EUROPEAN COMMISSION
SEVENTH FRAMEWORK PROGRAMME
FP7-ICT-2011-C
Grant agreement no.: 308830

Deliverable D2.1

Progress report for WP2: Architecture-driven decomposition

Project acronym	ADVENT
Project title	Architecture-driven verification of systems software
Funding scheme	FP7 FET Young Explorers
Scientific coordinator	Dr. Alexey Gotsman, IMDEA Software Institute, Alexey.Gotsman@imdea.org, +34 911 01 22 02

1 Summary

This work package has been concerned with developing techniques for decomposing the verification of systems software into that of its constituent components. We have considered this problem in several subdomains, developing techniques for separating the verification of:

- concurrent libraries from that of their clients in the presence of realistic interactions between the two [LMCS, ICALP'14];
- transactional memory implementations from that of programs using them [PODC'13, Draft];
- eventually consistent store implementations from that of programs storing data in them [POPL'14, TR];
- schedulers in preemptive operating system kernels from that of preemptable code [JFP].

2 Shared-Memory Concurrent Libraries [LMCS, ICALP'14]

The simplest way of structuring software is to implement a certain functionality as a library, which encapsulates the corresponding code and data. In systems software, key libraries are usually concurrent. Correspondingly, in this work package we have investigated methods of decomposing reasoning about programs with concurrent libraries into that about libraries and their clients considered in isolation. Correctness of concurrent libraries is usually stated using the notion of linearizability [7], which fixes a correspondence between a *concrete* library and a simpler *abstract* one, serving as its specification. Linearizability can be used for architecture-driven verification, since it satisfies the property of *observational refinement*: the behaviours of any client using the concrete library are contained in the behaviours of the client using the abstract one. Hence, while verifying a client of a concurrent library, we can soundly replace the library by its specification, thus simplifying the client verification. However, linearizability has so far only been defined in idealistic settings, assuming an almost complete isolation between the library and the client. We have proposed correctness conditions for concurrent libraries accounting for various ways in which they interact with their clients in systems software.

Linearizability with ownership transfer [LMCS]. Classical linearizability assumes a complete isolation between a library and its client, with interactions limited to passing values of a given data type as parameters or return values of library methods. This notion is not appropriate for low-level heap-manipulating languages, such as C/C++. There the library and the client run in a shared address space; thus, to prove the whole program correct, we need to verify that one of them does not corrupt the data structures used by the other. Program logics, such as separation logic [8], usually achieve this using the concept of *ownership* of data structures by a program component: the right to access a data structure is given only to a particular component. When verifying realistic programs, this ownership of data structures cannot be assigned statically; rather, it should be *transferred* between the client and the library at calls to and returns from the latter. Such interactions also exist in high-level languages providing basic memory protection, such as Java. For an example, a memory allocator accessible concurrently to multiple threads can be thought of as owning the blocks of memory on its free-list. Having allocated a block, a thread gets its exclusive ownership, which allows accessing it without interference from the other threads. When the thread frees the block, its ownership is returned to the allocator.

We have generalised linearizability to a setting where a library and its client execute in a shared address space, and boundaries between their data structures can change via ownership transfers. Linearizability is usually defined in terms of *histories*, which are sequences of calls to

and returns from a library in a given program execution, recording parameters and return values passed. To handle ownership transfer, histories also have to include descriptions of memory areas transferred. Furthermore, we should only consider *balanced* histories that repect the notion of ownership in a certain sense; e.g., a client should not transfer a piece of memory to the library that the latter already owns.

We have established that the proposed notion of linearizability implies observational refinement. The need to consider ownership transfer makes the proof of this fact highly non-trivial. This is because proving it requires us to convert a computation with a history produced by the concrete library into a computation with a history produced by the abstract one, which requires moving calls and returns to different points in the computation. In the setting without ownership transfer, these actions are thread-local and can be moved easily; however, once they involve ownership transfer, they become global and the justification of their moves becomes subtle, in particular, relying on the fact that the histories involved are balanced.

To avoid having to prove the new notion of linearizability from scratch for libraries that do not access some of the data structures transferred to them, such as concurrent containers, we also proposed a *frame rule for linearizability*. It ensures the linearizability of such libraries with respect to a specification with ownership transfer given their linearizability with respect to a specification without one.

The attached paper on the subject [LMCS] is a jounal version of a paper that appeared in CONCUR'12. It includes an extended presentation, full proofs and additional technical results.

Parameterised linearizability [ICALP'14]. Many concurrent libraries [5, 6, 9] are *parameterised*, meaning that they implement generic algorithms that take another library as a parameter and use it to implement more complex functionality. Reasoning about the correctness of parameterised libraries is challenging, as it requires considering all possible libraries that they can take as parameters. The classical linearizability is inapplicable to parameterised libraries, as it assumes that all of the library implementation is given. We have proposed a notion of *parameterised linearizability* that lifts this limitation. The key idea is to take into account not only interactions of a library with its client, but also with its parameter library, with the two types of interactions being subject to different conditions.

A challenge we had to deal with while generalising linearizability in this way is that parameterised libraries are often correct only under some assumptions about the context in which they are used. Thus, a parameterised library may assume that the library it takes as a parameter is *encapsulated*, meaning that clients cannot call its methods directly. A parameterised library may also accept as a parameter only libraries satisfying certain properties. For this reason, we actually proposed three notions of parameterised linearizability, appropriate for different situations: a general one, which does not make any assumptions about the client or the parameter library, a notion appropriate for the case when the parameter library is encapsulated, and *up-to linearizability*, which allows making assumptions about the parameter library. These notions differ in subtle ways: we find that there is a trade-off between the assumptions that parameterised libraries make about their environment and the conditions that a notion of linearizability has to impose on different types of interactions with it.

We have proved that the proposed notions of parameterised linearizability are *contextual*, i.e., closed under parameter instantiation. We also proved that parameterised linearizability is *compositional*: if several non-interacting libraries are linearizable, so is their composition. Finally, we showed that parameterised linearizability implies observational refinement. All these results allow modularising the reasoning about concurrent programs using parameterised libraries: contextuality and compositionality break the reasoning about complex parameterised libraries into that about individual libraries from which they are constructed; observational refinement then lifts this to complete programs, including clients.

To illustrate the applicability of our results, we proved the up-to linearizability of flat com-

bining [5], a generic algorithm for converting hard-to-parallelise sequential data structures into concurrent ones. As part of WP5 we have also proposed a logic that allows reasoning about parameterised libraries, which complements the results in this work package; see the corresponding deliverable.

3 Transactional Memory [PODC'13, Draft]

Transactional memory (TM) is a component that eases the task of writing concurrent applications by letting the programmer designate certain code blocks as *atomic*. TM allows developing a program and reasoning about its correctness as if each atomic block executed as a *transaction*—in one step and without interleaving with others—even though in reality the blocks can be executed concurrently.

The common approach to stating the correctness of TM implementations is through a *consistency condition* that restricts the possible TM executions. The main subtlety of formulating such a condition is the need to provide guarantees on the state of transactional objects observed by *live* transactions, i.e., those that have not yet committed or aborted. Because live transactions can always be aborted, one might think it unnecessary to provide any guarantees for them, as in fact done by common database consistency conditions. However, in the setting of transactional memory, this is unsatisfactory, since a live transaction that sees inconsistent data can commit a run-time *fault*, such as division by zero.

The question of which TM consistency condition to use is far from settled, with several candidates having been proposed. Furthermore, all such conditions have so far been given from the TM's point of view and have not been connected to the semantics of a programming language. As a consequence, it was not clear which of them provide the programmer with behaviors that correspond to the intuitive notion of atomic blocks, and which of them puts the minimal restrictions on TM implementations needed to achieve this.

We have bridged this gap by formalizing the intuitive expectations of a programmer as observational refinement between TM implementations [PODC'13], similarly to how we did this for concurrent libraries: a concrete TM observationally refines the abstract one if every behavior a user can observe of a program linked with the concrete TM can also be observed when the program is linked with the abstract TM instead. This allows the programmer to reason about the behavior of the program using the intuitive semantics formalized by the abstract TM; observational refinement implies that the conclusions will carry over to the case when the program uses the concrete TM. From the verification perspective, this allows us to prove the correctness of a TM separately from programs using it.

We have furthemore established which TM consistency conditions coincide with observational refinement for different choices of the programming language. Namely, we have showed that a variant of the *opacity* condition [4] coincides with observational refinement for a programming language in which local variables modified by a transaction are not rolled back upon an abort (e.g., Scala STM [10]) [PODC'13]. If the local variables are rolled back (as is done in most languages), then a variant of *transactional memory specification* (*TMS*) [2], which is weaker than opacity, coincides with observational refinement [Draft]. This formulates precise conditions that we need to discharge of the TM implementation and the program using it when verifying the whole system.

4 Eventually Consistent Replicated Stores [POPL'14, TR]

Our investigations have not been limited to shared-memory systems software, but also encompassed certain classes of software for distributed systems. Namely, we have considered *replicated stores*, which allow multiple clients to issue operations on shared data on a number of *replicas*, communicating changes to each other via message passing. Such stores are used in large-scale

Internet services, which *geo-replicate* data, placing its replicas in geographically distinct locations, and in mobile devices, which store replicas locally to support offline use. One benefit of such architectures is that the replicas remain locally available to clients even when network connections fail. Unfortunately, the famous CAP theorem [3] shows that such high **A**vailability and tolerance to network **P**artitions are incompatible with *strong* **C***onsistency*, i.e., the illusion of a single centralized replica handling all operations. For this reason, modern replicated stores often provide weaker forms of consistency, commonly dubbed *eventual consistency*. 'Eventual' usually refers to the guarantee that if clients stop issuing update requests, then the replicas will eventually reach a consistent state.

Unfortunately, the semantics of eventually consistent stores is poorly understood: the very term eventual consistency is a catch-all buzzword, and different stores claiming to be eventually consistent actually provide subtly different guarantees. One subtlety of their semantics is that different eventually consistent stores exhibit different degrees of deviation from strong consistency (this is similar to weak memory models; see below). Another subtlety is that, in replicated stores, clients can concurrently issue conflicting operations on the same data item at different replicas. For example, two users connected to replicas with different views of the shopping cart can also add and concurrently remove the same product. In such situations the store resolve conflicts consistently by means of special procedures encapsulated into *replicated data types*, which implement replicated objects such as registers, counters, sets or lists with various conflict resolution policies. For example, the *observed-remove set* processes concurrent operations trying to add and remove the same element so that an add always wins, an outcome that may be appropriate for a shopping cart.

The absence of precise specifications for eventually consistent stores makes it impossible to ignore their internals when verifying their clients. To address this problem, we have proposed a framework for formally defining the semantics of an eventually consistent store using:

- *replicated data type specifications*, determining the conflict resolution policies used by the store; and
- *consistency axioms*, constraining the level of anomalous behaviour exhibited by the store.

Replicated data type specifications specify the semantics of replicated objects declaratively, like abstract data types. This is achieved by defining the result of a data type operation not by a function of states, but of *operation contexts*—sets of events affecting the result of the operation, together with some relationships between them. We showed that our specifications are sufficiently flexible to handle data types representing a variety of conflict-resolution strategies: last-write-wins register, counter, multi-value register and observed-remove set.

Consistency axioms are able to define a variety of consistency models used in existing replicated stores, including a weak form of eventual consistency, session guarantees and different kinds of causal consistency. Furthermore, we found that, when specialized to the data type of last-writer-win register, these specifications were very close to weak memory models provided by modern processors and programming lanugages and, in particular, the one in the 2011 C/C++ standard [1]. Reasoning on weak memory models is studied in WP3, and the connection that we established suggests possible synergies between the two domains.

We attach the POPL'14 paper on the subject that describes the proposed specification framework. The part relevant to this deliverable consists of Sections 1-4 and 7-9. In the POPL'14 paper we also developed techniques for proving correctness of replicated data type implementations with respect to our specifications. This part of the paper is covered in the deliverable for WP5.

We have also made forrays into specifying the semantics of advanced features provided by eventually consistent stores, such as mixtures of different consistency levels and transactions. As part of this effort, we also justified our consistency axioms by proving that they are validated by

an example abstract implementation, based on algorithms used in eventually consistent stores. We attach a preliminary technical report on this effort [TR].

5 Preemptive OS Kernels [JFP]

Many interactions between parts of systems software do not fit into the standard paradigm of libraries or objects calling methods or sending messages to each other. Rather, one component may provide the illusion to the rest of the system of running on a higher-level machine, with the system organised into layers of such virtual machines. A prominent example of such a component is an operating system (OS) scheduler, which virtualises the computer into an abstract machine where every process has a dedicated CPU. We have proposed methods for decomposing the verification of OS kernels into verifying the scheduler and the rest of the kernel separately.

This is nontrivial since most major OS kernels are designed to run with multiple CPUs and are *preemptive*: it is possible for a process running in the kernel mode to get descheduled. As a consequence, the correctness of the scheduler is *interdependent* with the correctness of the rest of the kernel. This is because, when reasoning about a system call implementation in a preemptive kernel, we have to consider the possibility of context-switch code getting executed at almost every program point. Upon a context switch, the state of the system call will be stored in kernel data structures and subsequently loaded for execution again, possibly on a different CPU. A bug in the context switch code can load an incorrect state of the system call implementation upon a context switch, and a bug in the system call can corrupt the scheduler's data structures.

We have proposed a logic, based on separation logic [8], that is able to decompose the verification of safety properties of preemptive OS code into verifying preemptable code and the scheduler separately. This is the first logic that can achieve this in the presence of interdependencies between the scheduler and the kernel typical for mainstream OS kernels, such as those of Linux, FreeBSD and Mac OS X. The modularity of the logic is reflected in the structure of its proof system, which is partitioned into high-level and low-level parts. The high-level proof system verifies preemptable code under the illusion of an abstract machine where every process has its own virtual CPU—the control moves from one program point in the process code to the next without changing its state. This illusion is justified by verifying the scheduler code separately from the kernel in the low-level proof system.

We have also proposed a novel way of proving soundness of such logics: proofs in neither of the two proof systems comprising the logic are interpreted with respect to any semantics alone. Instead, our soundness statement interprets a proof of the kernel in the high-level system and a proof of the scheduler in the low-level one together with respect to the semantics of the concrete machine. Such a form of soundness statement allows us to cope with complex ownership transfers between the scheduler and the kernel, which would be difficult to accommodate had we tried to give meaning to proofs of the scheduler and the kernel separately.

Even though a scheduler is supposed to provide an illusion of running on a dedicated *virtual* CPU to every process, in practice, some features available to the kernel code can break through this abstraction: e.g., a process can disable preemption and become aware of the *physical* CPU on which it is currently executing. For example, some OS kernels use this to implement *per-CPU data structures*—arrays indexed by CPU identifiers such that a process can only access an entry in an array when it runs on the corresponding CPU. We have demonstrated that our approach can deal with such implementation exposures by extending the high-level proof system for the kernel code with axioms that allow reasoning about per-CPU data structures.

The attached paper on the subject [JFP] is a jounal version of a paper that appeared in ICFP'11. It includes an extended presentation, full proofs and additional technical results, including the treatment of per-CPU data structures.

References

[1] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, 2011.

[2] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, 25(5), 2013.

[3] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2), 2002.

[4] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPOPP*, 2008.

[5] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*, 2010.

[6] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPOPP*, 2008.

[7] M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3), 1990.

[8] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[9] C. Russo. The Joins concurrency library. In *PADL*, 2007.

[10] Scala STM Expert Group. Scala STM quick start guide. http://nbronson.github.io/scala-stm/quick_start.html.

List of Attached Papers

[LMCS] Alexey Gotsman and Hongseok Yang. Linearizability with ownership transfer. Logical Methods in Computer Science, 9(3:12).

[ICALP'14] Andrea Cerone, Alexey Gotsman, and Hongseok Yang. Parameterised Linearisability. ICALP'14: International Colloquium on Automata, Languages, and Programming, Copenhagen, Denmark. To appear.

[PODC'13] Hagit Attiya, Alexey Gotsman, Sandeep Hans, and Noam Rinetzky. A programming language perspective on transactional memory consistency. PODC'13: Symposium on Principles of Distributed Computing, Montreal, Canada, pages 309-318.

[Draft] Hagit Attiya, Alexey Gotsman, Sandeep Hans, and Noam Rinetzky. Safety of Live Transactions in Transactional Memory: TMS is Necessary and Sufficient.

[POPL'14] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: specification, verification, optimality. POPL'14: Symposium on Principles of Programming Languages, San Diego, CA, USA, pages 271-284.

[TR] Sebastian Burckhardt, Alexey Gotsman, and Hongseok Yang. Understanding eventual consistency, Microsoft Research Technical Report MSR-TR-2013-39.

[JFP] Alexey Gotsman and Hongseok Yang. Modular verification of preemptive OS kernels. Journal of Functional Programming, 23(4):452-514.

Logical Methods in Computer Science
Vol. 9(3:12)2013, pp. 1–47
www.lmcs-online.org

Submitted Feb. 15, 2013
Published Sep. 9, 2013

LINEARIZABILITY WITH OWNERSHIP TRANSFER

ALEXEY GOTSMAN [a] AND HONGSEOK YANG [b]

[a] IMDEA Software Institute, Madrid, Spain
e-mail address: Alexey.Gotsman@imdea.org

[b] University of Oxford, Oxford, UK
e-mail address: Hongseok.Yang@cs.ox.ac.uk

Abstract. Linearizability is a commonly accepted notion of correctness for libraries of concurrent algorithms. Unfortunately, it assumes a complete isolation between a library and its client, with interactions limited to passing values of a given data type. This is inappropriate for common programming languages, where libraries and their clients can communicate via the heap, transferring the ownership of data structures, and can even run in a shared address space without any memory protection.

In this paper, we present the first definition of linearizability that lifts this limitation and establish an Abstraction Theorem: while proving a property of a client of a concurrent library, we can soundly replace the library by its abstract implementation related to the original one by our generalisation of linearizability. This allows abstracting from the details of the library implementation while reasoning about the client. We also prove that linearizability with ownership transfer can be derived from the classical one if the library does not access some of data structures transferred to it by the client.

1. Introduction

The architecture of concurrent software usually exhibits some forms of modularity. For example, concurrent algorithms are encapsulated in libraries and complex algorithms are often constructed using libraries of simpler ones. This lets developers benefit from ready-made libraries of concurrency patterns and high-performance concurrent data structures, such as java.util.concurrent for Java and Threading Building Blocks for C++. To simplify reasoning about concurrent software, we need to exploit the available modularity. In particular, in reasoning about a client of a concurrent library, we would like to abstract from the details of a particular library implementation. This requires an appropriate notion of library correctness.

Correctness of concurrent libraries is commonly formalised by *linearizability* [18], which fixes a certain correspondence between the library and its specification. The latter is usually just another library, but implemented atomically using an abstract data type; the two libraries are called *concrete* and *abstract*, respectively. A good notion of linearizability should validate an *Abstraction Theorem* [12, 14]: the behaviours of any client using the

2012 ACM CCS: [**Theory of computiation**]: Semantics and reasoning.
Key words and phrases: Concurrency, Linearizability, Ownership Transfer, Semantics, Data Abstraction.

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.2168/LMCS-9(3:12)2013

concrete library are contained in the behaviours of the client using the abstract one. This makes it sound to replace a library by its specification in reasoning about its client.

Classical linearizability assumes a complete isolation between a library and its client, with interactions limited to passing values of a given data type as parameters or return values of library methods. This notion is not appropriate for low-level heap-manipulating languages, such as C/C++. There the library and the client run in a shared address space; thus, to prove the whole program correct, we need to verify that one of them does not corrupt the data structures used by the other. Type systems [8] and program logics [23] usually establish this using the concept of *ownership* of data structures by a program component: the right to access a data structure is given only to a particular component or a set of them. When verifying realistic programs, this ownership of data structures cannot be assigned statically; rather, it should be *transferred* between the client and the library at calls to and returns from the latter. The times when ownership is transferred are not determined operationally, but set by the proof method: as O'Hearn famously put it, "ownership is in the eye of the asserter" [23]. However, ownership transfer reflects actual interactions between program components via the heap, e.g., alternating accesses to a shared area of memory. Such interactions also exist in high-level languages providing basic memory protection, such as Java. In this case, we need to ensure that a client does not subvert a library by accessing a memory object after its ownership was transferred to the latter.

For an example of ownership transfer between concurrent libraries and their clients consider a memory allocator accessible concurrently to multiple threads. We can think of the allocator as owning the blocks of memory on its free-list; in particular, it can store free-list pointers in them. Having allocated a block, a thread gets its exclusive ownership, which allows accessing it without interference from the other threads. When the thread frees the block, its ownership is returned to the allocator. Trying to write to a memory cell after it was freed has dire consequences.

As another example, consider any container with concurrent access, such as a concurrent set from java.util.concurrent or Threading Building Blocks. A typical use of such a container is to store pointers to a certain type of data structures. However, when verifying a client of the container, we usually think of the latter as holding the ownership of the data structures whose addresses it stores [23]. Thus, when a thread inserts a pointer to a data structure into a container, its ownership is transferred from the thread to the container. When another thread removes a pointer from the container, it acquires the ownership of the data structure the pointer identifies. If the first thread tries to access a data structure after a pointer to it has been inserted into the container, this may result in a race condition. Unlike a memory allocator, the container code usually does not access the contents of the data structures its elements identify, but merely ferries their ownership between different threads. For this reason, correctness proofs for such containers [1,10,29] have so far established their classical linearizability, without taking ownership transfer into account.

We would like to use the notion of linearizability and, in particular, an Abstraction Theorem to reason about the above libraries and their clients in isolation, taking into account only the memory that they own. To this end, we would like the correctness of a library to constrain not only pointers that are passed between it and the client, but also the contents of the data structures whose ownership is transferred. So far, there has been no notion of linearizability that would allow this. In the case of concurrent containers, we have no way of using classical linearizability established for them to validate an Abstraction

Theorem that would be applicable to clients performing ownership transfer. This paper fills in these gaps.

Contributions. In this paper, we generalise linearizability to a setting where a library and its client execute in a shared address space, and boundaries between their data structures can change via ownership transfers. Linearizability is usually defined in terms of *histories*, which are sequences of calls to and returns from a library in a given program execution, recording parameters and return values passed. To handle ownership transfer, histories also have to include descriptions of memory areas transferred. However, in this case, some histories cannot be generated by any pair of a client and a library: while generating histories of a library we should only consider its executions in an environment that respects ownership. For example, a client that transfers an area of memory upon a call to a library not communicating with anyone else cannot then transfer the same area again before getting it back from the library upon a method return. We propose a notion of *balancedness* that characterises those histories that treat ownership transfer correctly. We then define a *linearizability relation* between balanced histories, matching histories of a concrete and an abstract library (Section 3).

This definition does not rely on a particular model of program states for describing memory areas transferred in histories, but assumes an arbitrary model defined by a *separation algebra* [7]. By picking a model with so-called *permissions* [5,9], we can allow clients and libraries to transfer non-exclusive rights to access certain memory areas in particular ways, instead of transferring their full ownership. This makes our results applicable even when libraries and their clients share access to some areas of memory. Our definition of balancedness for arbitrary separation algebras relies on a new formalisation of the notion of a *footprint* of a state, describing the amount of permissions the state includes (Section 2).

The rest of our technical development relies on a novel compositional semantics for a language with libraries that defines the denotation of a library or a client considered separately in an environment that communicates with the component correctly via ownership transfers (Sections 4 and 5). In particular, the semantics allows us to generate the set of all histories that can be produced by a library solely from its code, without considering all its possible clients. This, in its turn, allows us to lift the linearizability on histories to libraries and establish the Abstraction Theorem (Section 6). On the way, we also obtain an insight into the original definition of linearizability without ownership transfer, showing a surprising relationship between one of the ways of its formulation and the plain subset inclusion on the sets of histories produced by concrete and abstract libraries.

We note that the need to consider ownership transfer makes the proof of the Abstraction Theorem highly non-trivial. This is because proving the theorem requires us to convert a computation with a history produced by the concrete library into a computation with a history produced by the abstract one, which requires moving calls and returns to different points in the computation. In the setting without ownership transfer, these actions are thread-local and can be moved easily; however, once they involve ownership transfer, they become global and the justification of their moves becomes subtle, in particular, relying on the fact that the histories involved are balanced (Section 7).

To avoid having to prove the new notion of linearizability from scratch for libraries that do not access some of the data structures transferred to them, such as concurrent containers, we propose a *frame rule for linearizability* (Section 8). It ensures the linearizability of such libraries with respect to a specification with ownership transfer given their linearizability with respect to a specification without one.

We provide a glossary of notation at the end of the paper.

2. Footprints of States

2.1. **Separation Algebras.** Our results hold for a class of models of program states called *separation algebras* [7], which allow expressing the dynamic memory partitioning between libraries and clients.

Definition 2.1. A ***separation algebra*** is a set Σ, together with a partial commutative, associative and cancellative operation $*$ on Σ and a unit element $e \in \Sigma$. Here commutativity and associativity hold for the equality that means both sides are defined and equal, or both are undefined. The property of cancellativity says that for each $\sigma \in \Sigma$, the function $\sigma * \cdot : \Sigma \rightharpoonup \Sigma$ is injective.

We think of elements of a separation algebra Σ as *portions* of program states and the $*$ operation as combining such portions. The partial states allow us to describe parts of the program state belonging to a library or the client. When the $*$-combination of two states is defined, we call them ***compatible***. Incompatible states usually make contradictory claims about the ownership of memory. We sometimes use a pointwise lifting $* : 2^\Sigma \times 2^\Sigma \to 2^\Sigma$ of $*$ to sets of states: for $p, q \in 2^\Sigma$ we let $p * q = \{\sigma_1 * \sigma_2 \mid \sigma_1 \in p \wedge \sigma_2 \in q\}$.

Elements of separation algebras are often defined using partial functions. We use the following notation: $g(x)\downarrow$ means that the function g is defined on x, $g(x)\uparrow$ means that it is undefined on x, $\mathsf{dom}(g)$ denotes the set of arguments on which g is defined, $[\,]$ denotes a nowhere-defined function, and $g[x : y]$ denotes the function that has the same value as g everywhere, except for x, where it has the value y. We also write $\_$ for an expression whose value is irrelevant and implicitly existentially quantified.

Below is an example separation algebra RAM:

$$\mathsf{Loc} = \{1, 2, \ldots\}; \qquad \mathsf{Val} = \mathbb{Z}; \qquad \mathsf{RAM} = \mathsf{Loc} \rightharpoonup_{\mathit{fin}} \mathsf{Val}.$$

A (partial) state in this model consists of a finite partial function from allocated memory locations to the values they store. The $*$ operation on RAM is defined as the disjoint function union $\uplus$, with the nowhere-defined function $[\,]$ as its unit. Thus, $*$ combines disjoint pieces of memory.

More complicated separation algebras do not split memory completely, instead allowing heap parts combined by $*$ to overlap. This is done by associating so-called *permissions* [5] with memory cells in the model, which do not give their exclusive ownership, but allow accessing them in a certain way. Types of permissions range from read sharing [5] to accessing memory in an arbitrary way consistent with a given specification [9]. We now give an example of a separation algebra with permissions of the former kind.

We define the algebra RAM_π as follows:

$$\mathsf{Loc} = \{1, 2, \ldots\}; \qquad \mathsf{Val} = \mathbb{Z}; \qquad \mathsf{Perm} = (0, 1]; \qquad \mathsf{RAM}_\pi = \mathsf{Loc} \rightharpoonup_{\mathit{fin}} (\mathsf{Val} \times \mathsf{Perm}).$$

A state in this model consists of a finite partial function from allocated memory locations to values they store and so-called *permissions*—numbers from $(0, 1]$ that show "how much" of the memory cell belongs to the partial state [5]. The latter allow a library and its client to share access to some of memory cells. Permissions in RAM_π allow only read sharing: when defining the semantics of commands over states in RAM_π, the permissions strictly less than 1 are interpreted as permissions to read; the full permission 1 additionally allows

writing. The $*$ operation on RAM_π adds up permissions for memory cells. Formally, for $\sigma_1, \sigma_2 \in \mathsf{RAM}_\pi$, we write $\sigma_1 \sharp \sigma_2$ if:

$$\forall x \in \mathsf{Loc}.\, \sigma_1(x)\downarrow \wedge \sigma_2(x)\downarrow \implies (\exists u, \pi_1, \pi_2.\, \sigma_1(x) = (u, \pi_1) \wedge \sigma_2(x) = (u, \pi_2) \wedge \pi_1 + \pi_2 \leq 1).$$

If $\sigma_1 \sharp \sigma_2$, then we define

$$\sigma_1 * \sigma_2 = \{(x, (u, \pi)) \mid (\sigma_1(x) = (u, \pi) \wedge \sigma_2(x)\uparrow) \vee (\sigma_2(x) = (u, \pi) \wedge \sigma_1(x)\uparrow) \vee (\sigma_1(x) = (u, \pi_1) \wedge \sigma_2(x) = (u, \pi_2) \wedge \pi = \pi_1 + \pi_2)\};$$

otherwise, $\sigma_1 * \sigma_2$ is undefined. The unit for $*$ is the empty heap $[\,]$. This definition of $*$ allows us, e.g., to split a memory area into two disjoint parts. It also allows splitting a cell with a full permission 1 into two parts, carrying read-only permissions $1/2$ and agreeing on the value stored in the cell. These permissions can later be recombined to obtain the full permission, which allows both reading from and writing to the cell.

Since we develop all our results for an arbitrary separation algebra, by instantiating it with algebras similar to RAM_π, we can handle cases when a library and its client share access to some memory areas.

Consider an arbitrary separation algebra Σ with an operation $*$. We define a partial operation $\setminus : \Sigma \times \Sigma \rightharpoonup \Sigma$, called ***state subtraction***, as follows: $\sigma_2 \setminus \sigma_1$ is a state in Σ such that $\sigma_2 = (\sigma_2 \setminus \sigma_1) * \sigma_1$; if such a state does not exist, $\sigma_2 \setminus \sigma_1$ is undefined. The cancellativity of $*$ implies that $\sigma_2 \setminus \sigma_1$ is determined uniquely, and hence, the $\setminus$ operation is well-defined. When reasoning about ownership transfer between a library and a client, we use the $*$ operation to express a state change for the component that is receiving the ownership of memory, and the $\setminus$ operation for the one that is giving it up.

Proposition 2.2. *For all $\sigma_1, \sigma_2, \sigma_3 \in \Sigma$, if $\sigma_1 * \sigma_2$ and $\sigma_1 \setminus \sigma_3$ are defined, then*

$$(\sigma_1 * \sigma_2) \setminus \sigma_3 = (\sigma_1 \setminus \sigma_3) * \sigma_2.$$

2.2. **Footprints.** Our definition of linearizability uses a novel formalisation of a *footprint* of a state, which, informally, describes the amount of memory or permissions the state includes.

Definition 2.3. A ***footprint*** of a state σ in a separation algebra Σ is the set of states

$$\delta(\sigma) = \{\sigma' \mid \forall \sigma''.\, (\sigma' * \sigma'')\downarrow \iff (\sigma * \sigma'')\downarrow\}.$$

In the following, l ranges over footprints. The function δ computes the equivalence class of states with the same footprint as σ. In the case of RAM, we have $\delta(\sigma) = \{\sigma' \mid \mathsf{dom}(\sigma) = \mathsf{dom}(\sigma')\}$ for every $\sigma \in \mathsf{RAM}$. Thus, states with the same footprint contain the same memory cells. Definitions of δ for separation algebras with permissions are more complicated, taking into account not only memory cells present in the state, but also permissions for them. In the case of the algebra RAM_π, for $\sigma \in \mathsf{RAM}_\pi$ we have

$$\delta(\sigma) = \{\sigma' \mid \forall x.\, (\sigma(x)\downarrow \iff \sigma'(x)\downarrow) \wedge \forall u, \pi.\, (\sigma(x) = (u, \pi) \wedge \pi < 1 \implies \sigma(x) = \sigma'(x)) \wedge (\sigma(x) = (u, 1) \implies \sigma'(x) = (\_, 1))\}.$$

In other words, states with the same footprint contain the same memory cells with the identical permissions; in the case of memory cells on read permissions, the states also have to agree on their values.

Let $\mathcal{F}(\Sigma) = \{\delta(\sigma) \mid \sigma \in \Sigma\}$ be the set of footprints in a separation algebra Σ. We now lift the $*$ and $\setminus$ operations on Σ to $\mathcal{F}(\Sigma)$. First, we define the operation $\circ : \mathcal{F}(\Sigma) \times \mathcal{F}(\Sigma) \rightharpoonup \mathcal{F}(\Sigma)$ for adding footprints. Consider $l_1, l_2 \in \mathcal{F}(\Sigma)$ and $\sigma_1, \sigma_2 \in \Sigma$ such that $l_1 = \delta(\sigma_1)$ and $l_2 = \delta(\sigma_2)$. If $\sigma_1 * \sigma_2$ is defined, we let $l_1 \circ l_2 = \delta(\sigma_1 * \sigma_2)$; otherwise $l_1 \circ l_2$ is undefined.

Proposition 2.4. *The $\circ$ operation is well-defined.*

Proof. Consider $l_1, l_2 \in \mathcal{F}(\Sigma)$ and $\sigma_1, \sigma_2 \in \Sigma$ such that $l_1 = \delta(\sigma_1)$ and $l_2 = \delta(\sigma_2)$. Take another pair of states $\sigma_1', \sigma_2' \in \Sigma$ such that $l_1 = \delta(\sigma_1')$ and $l_2 = \delta(\sigma_2')$. Thus, $\sigma_1' \in \delta(\sigma_1)$ and $\sigma_2' \in \delta(\sigma_2)$, which implies:

$$(\sigma_1' * \sigma_2')\downarrow \iff (\sigma_1 * \sigma_2')\downarrow \iff (\sigma_1 * \sigma_2)\downarrow.$$

Furthermore, if $\sigma_1' * \sigma_2'$ and $\sigma_1 * \sigma_2$ are defined, then for all $\sigma' \in \Sigma$,

$$(\sigma_1' * \sigma_2' * \sigma')\downarrow \iff (\sigma_1 * \sigma_2' * \sigma')\downarrow \iff (\sigma_1 * \sigma_2 * \sigma')\downarrow.$$

Hence, $\delta(\sigma_1 * \sigma_2) = \delta(\sigma_1' * \sigma_2')$, so that $\circ$ is well-defined. □

For RAM, $\circ$ is just the pointwise lifting of the disjoint function union $\uplus$.

To define a subtraction operation on footprints, we use the following condition.

Definition 2.5. The $*$ operation of a separation algebra Σ is ***cancellative on footprints*** when for all $\sigma_1, \sigma_2, \sigma_1', \sigma_2' \in \Sigma$, if $\sigma_1 * \sigma_2$ and $\sigma_1' * \sigma_2'$ are defined, then

$$(\delta(\sigma_1 * \sigma_2) = \delta(\sigma_1' * \sigma_2') \wedge \delta(\sigma_1) = \delta(\sigma_1')) \implies \delta(\sigma_2) = \delta(\sigma_2').$$

For example, the $*$ operations on RAM and RAM_π satisfy this condition.

When the $*$ operation of an algebra Σ is cancellative on footprints, we can define an operation $\mathbin{\backslash\!\backslash} : \mathcal{F}(\Sigma) \times \mathcal{F}(\Sigma) \rightharpoonup \mathcal{F}(\Sigma)$ of ***footprint subtraction*** as follows. Consider $l_1, l_2 \in \mathcal{F}(\Sigma)$. If for some $\sigma_1, \sigma_2, \sigma \in \Sigma$, we have $l_1 = \delta(\sigma_1)$, $l_2 = \delta(\sigma_2)$ and $\sigma_2 = \sigma_1 * \sigma$, then we let $l_2 \mathbin{\backslash\!\backslash} l_1 = \delta(\sigma)$. When such $\sigma_1, \sigma_2, \sigma$ do not exist, $l_2 \mathbin{\backslash\!\backslash} l_1$ is undefined.

Proposition 2.6. *The $\mathbin{\backslash\!\backslash}$ operation is well-defined.*

Proof. Consider $l_1, l_2 \in \mathcal{F}(\Sigma)$ and $\sigma_1, \sigma_2, \sigma_1', \sigma_2', \sigma, \sigma' \in \Sigma$ such that $\sigma_1, \sigma_1' \in l_1$, $\sigma_2, \sigma_2' \in l_2$, $\sigma_1 = \sigma_2 * \sigma$ and $\sigma_1' = \sigma_2' * \sigma'$. We have:

$$\delta(\sigma_2 * \sigma) = \delta(\sigma_1) = l_1 = \delta(\sigma_1') = \delta(\sigma_2' * \sigma').$$

Since $\delta(\sigma_2) = \delta(\sigma_2')$, by Definition 2.5 this implies $\delta(\sigma) = \delta(\sigma')$, so that $\mathbin{\backslash\!\backslash}$ is well-defined. □

For RAM, the $\mathbin{\backslash\!\backslash}$ operation is defined as follows. For $l_1, l_2 \in \mathcal{F}(\mathsf{RAM})$, take any $\sigma_1, \sigma_2 \in \mathsf{RAM}$ such that $\delta(\sigma_1) = l_1$ and $\delta(\sigma_2) = l_2$. Then $l_2 \mathbin{\backslash\!\backslash} l_1 = \{\sigma \mid \mathsf{dom}(\sigma) \uplus \mathsf{dom}(\sigma_1) = \mathsf{dom}(\sigma_2)\}$, if $\mathsf{dom}(\sigma_1) \subseteq \mathsf{dom}(\sigma_2)$; otherwise, $l_1 \mathbin{\backslash\!\backslash} l_2$ is undefined.

We say that a footprint l_1 is ***smaller*** than l_2, written $l_1 \preceq l_2$, when $l_2 \mathbin{\backslash\!\backslash} l_1$ is defined. The $\circ$ and $\mathbin{\backslash\!\backslash}$ operations on footprints satisfy an analogue of Proposition 2.2.

Proposition 2.7. *For all $l_1, l_2, l_3 \in \mathcal{F}(\Sigma)$, if $l_1 \circ l_2$ and $l_1 \mathbin{\backslash\!\backslash} l_3$ are defined, then*

$$(l_1 \circ l_2) \mathbin{\backslash\!\backslash} l_3 = (l_1 \mathbin{\backslash\!\backslash} l_3) \circ l_2.$$

In the rest of the paper, we fix a separation algebra Σ with the $*$ operation cancellative on footprints.

3. Linearizability with Ownership Transfer

In the following, we consider descriptions of computations of a library providing several methods to a multithreaded client. We fix a set $\mathsf{ThreadID}$ of thread identifiers and a set Method of method names. As we explained in Section 1, a good definition of linearizability has to allow replacing a concrete library implementation with its abstract version while keeping client behaviours reproducible. For this, it should require that the two libraries have similar client-observable behaviours. Such behaviours are recorded using *histories*, which we now define in our setting.

Definition 3.1. An ***interface action*** ψ is an expression of the form $(t, \mathsf{call}\ m(\sigma))$ or $(t, \mathsf{ret}\ m(\sigma))$, where $t \in \mathsf{ThreadID}$, $m \in \mathsf{Method}$ and $\sigma \in \Sigma$. We denote the sets of all call and return actions by $\mathsf{CallAct}$ and RetAct, and the set of all interface actions by $\mathsf{CallRetAct}$.

An interface action records a call to or a return from a library method m by thread t. The component σ in $(t, \mathsf{call}\ m(\sigma))$ specifies the part of the state transferred upon the call from the client to the library; σ in $(t, \mathsf{ret}\ m(\sigma))$ is transferred in the other direction. For example, in the algebra RAM, the annotation $\sigma = [42 : 0]$ implies the transfer of the cell at the address 42 storing 0. In the algebra RAM_π, $\sigma = [42 : (0, 1/2)]$ implies the transfer of a read permission for this cell.

Definition 3.2. A ***history*** H is a finite sequence of interface actions such that for every thread t, its projection $H|_t$ to actions by t is a sequence of alternating call and return actions over matching methods that starts from a call action.

In the following, we use the standard notation for sequences: ε is the empty sequence, $\tau(i)$ is the i-th element of a sequence τ, $\tau\downarrow_k$ is the prefix of τ of length k, and $|\tau|$ is the length of τ.

Not all histories make intuitive sense with respect to the ownership transfer reading of interface actions. For example, let $\Sigma = \mathsf{RAM}$ and consider the history in Figure 1(a). The history is meant to describe *all* the interactions between the library and the client. According to the history, the cell at the address 10 was first owned by the client, and then transferred to the library by thread 1. However, before this state was transferred back to the client, it was again transferred from the client to the library, this time by thread 2. This is not consistent with the intuition of ownership transfer, as executing the second action requires the cell to be owned both by the library and by the client, which is impossible in RAM.

As we show in this paper, histories that do not respect the notion of ownership, such as the one above, cannot be generated by any program, and should not be taken into account when defining linearizability. We now use the notion of footprints of states from Section 2 to characterise formally the set of histories that respect ownership. A finite history H induces a partial function $[\![H]\!]^\sharp : \mathcal{F}(\Sigma) \rightharpoonup \mathcal{F}(\Sigma)$, which tracks how a computation with the history H changes the footprint of the library state:

$$
\begin{aligned}
[\![\varepsilon]\!]^\sharp l &= l;\\
[\![H\psi]\!]^\sharp l &= [\![H]\!]^\sharp l \circ \delta(\sigma), \quad \text{if } \psi = (\_, \mathsf{call}\ \_(\sigma)) \wedge ([\![H]\!]^\sharp l \circ \delta(\sigma))\downarrow;\\
[\![H\psi]\!]^\sharp l &= [\![H]\!]^\sharp l \setminus\!\setminus \delta(\sigma), \quad \text{if } \psi = (\_, \mathsf{ret}\ \_(\sigma)) \wedge ([\![H]\!]^\sharp l \setminus\!\setminus \delta(\sigma))\downarrow;\\
[\![H\psi]\!]^\sharp l &= \text{undefined}, \qquad \text{otherwise.}
\end{aligned}
$$

Using this function, we characterise histories respecting the notion of ownership as follows.

Definition 3.3. A history H is ***balanced*** from $l \in \mathcal{F}(\Sigma)$ if $[\![H]\!]^\sharp(l)$ is defined. We call subsets of $\mathsf{BHistory} = \{(l, H) \mid H \text{ is balanced from } l\}$ ***interface sets***.

An interface set can be used to describe all the behaviours of a library relevant to its clients. In the following, $\mathcal{H}$ ranges over interface sets.

To keep client behaviours reproducible when replacing a concrete library by an abstract one, we do not need to require the latter to reproduce the histories of the former exactly: the histories generated by the two libraries can be different in ways that are irrelevant for their clients. We now introduce a *linearizability relation* that matches a history of a concrete library with that of the abstract one that yields the same client-observable behaviour.

Definition 3.4. The ***linearization relation*** $\sqsubseteq$ on histories is defined as follows: $H \sqsubseteq H'$ holds if there exists a bijection $\rho\colon \{1, \ldots, |H|\} \to \{1, \ldots, |H'|\}$ such that

$$\forall i, j.\, (H(i) = H'(\rho(i))) \wedge ((i < j \wedge ((\exists t.\, H(i) = (t, \_) \wedge H(j) = (t, \_)) \vee (H(i) = (\_, \mathsf{ret}\ \_) \wedge H(j) = (\_, \mathsf{call}\ \_)))) \implies \rho(i) < \rho(j)).$$

We lift $\sqsubseteq$ to $\mathsf{BHistory}$ as follows: $(l, H) \sqsubseteq (l', H')$ holds if $l' \preceq l$ and $H \sqsubseteq H'$.

Finally, we lift $\sqsubseteq$ to interface sets as follows: $\mathcal{H}_1 \sqsubseteq \mathcal{H}_2$ holds if

$$\forall (l_1, H_1) \in \mathcal{H}_1.\, \exists (l_2, H_2) \in \mathcal{H}_2.\, (l_1, H_1) \sqsubseteq (l_2, H_2).$$

Thus, a history H is linearized by a history H' when the latter is a permutation of the former preserving the order of actions within threads and non-overlapping method invocations. The duration of a method invocation is defined by the interval from the method call action to the corresponding return action (or to the end of the history if there is none). An interface set $\mathcal{H}_1$ is linearized by an interface set $\mathcal{H}_2$, if every history in $\mathcal{H}_1$ may be reproduced in a linearized form by $\mathcal{H}_2$ without requiring more memory. We now discuss the definition in more detail.

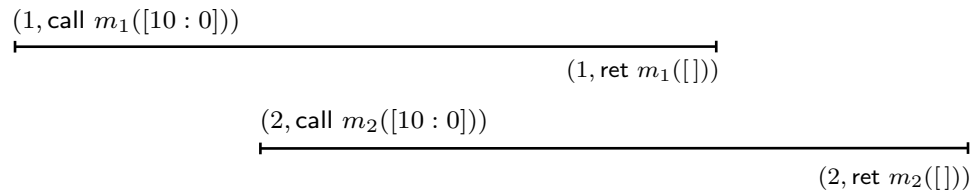
Definition 3.4 treats parts of memory whose ownership is passed between the library and the client in the same way as parameters and return values in the classical definition [18]: they are required to be the same in the two histories. In fact, the setting of the classical definition can be modelled in ours if we pass parameters and return values via the heap. Let $\Sigma = \mathsf{RAM}$ and let us fix distinct locations $\mathsf{arg}_t \in \mathsf{Loc}$ for $t \in \mathsf{ThreadID}$ meant for the transfer of parameters and return values. Then histories of the classical definition are represented in our setting by histories where all actions are of the form

$$(t, \mathsf{call}\ m([\mathsf{arg}_t : \mathit{param}])) \text{ or } (t, \mathsf{ret}\ m([\mathsf{arg}_t : \mathit{retval}])), \text{ where } \mathit{param}, \mathit{retval} \in \mathsf{Val}.$$

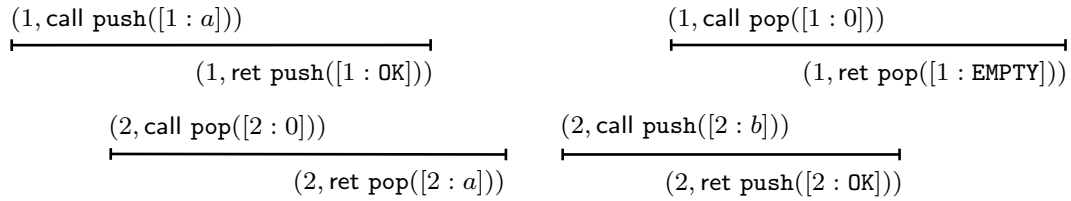
The novelty of our definition lies in restricting the histories considered to balanced ones, which are the only ones that can be produced by programs (we formalise this fact in Section 5). The notion of balancedness also plays a key role in proving the Abstraction Theorem in the presence of ownership transfer (Section 6.1).

The fact that the linearizability relation allows us to permute actions by different threads lets us arrange method invocations into a linear sequence. For example the history in Figure 1(b) is linearized by that in Figure 1(c). The former might correspond to a concurrent stack implementation, where threads 1 and 2 pass parameters and return values via locations 1 and 2, respectively. Histories such as the one in Figure 1(c), where a call to every method is immediately followed by the corresponding return, are called ***sequential***. Sequential histories correspond to abstract libraries with every method implemented atomically. When the histories in Figures 1(b) and 1(c) are members of interface sets defining the behaviour of concurrent and atomic stack implementations, the linearizability relationship

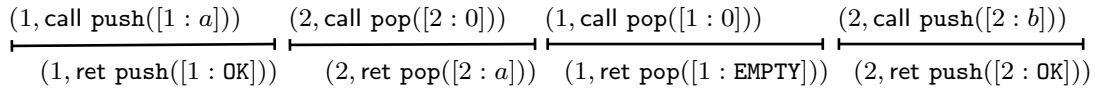
(a):



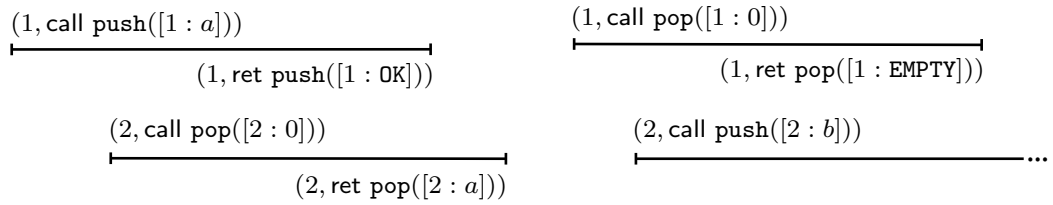
(b):



(c):



(d):



(e):

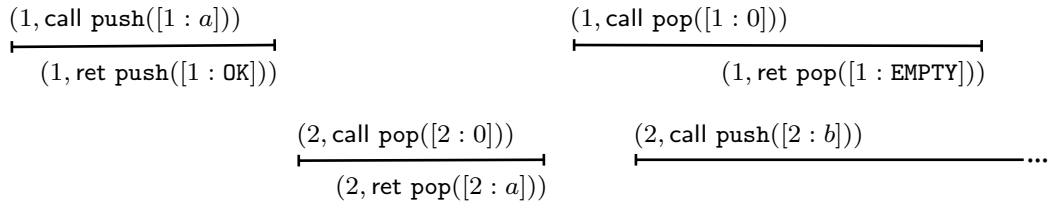


Figure 1: Example histories

between them allows us to justify that the call to `pop` by thread 1 in Figure 1(b) can return `EMPTY`, since this behaviour can be witnessed by the valid history of the atomic implementation in Figure 1(c). (In fact, the `pop` would also be allowed to return b, since the resulting history would be linearized by a sequential history with the `push` of b preceding the `pop`.)

The requirement that the order of non-overlapping method invocations be preserved is inherited from the classical notion of linearizability [18]. As shown by Filipović et al. [12], this requirement is essential to validate an Abstraction Theorem for clients that can communicate via shared client-side variables. For example, since in Figure 1(b) the `push` of a returns before the `push` of b is called, the order between these method invocations has to stay the same in any linearizing history, such as the one in Figure 1(c).

Following Filipović et al. [12], we do not require the abstract history H' to be sequential, like in the classical definition of linearizablity. This allows our definition to compare behaviours of two concurrent library implementations. We also allow a concrete history to contain calls without matching returns, arising, e.g., because the corresponding method invocation did not terminate. In this case, we require the same behaviour to be reproduced in the abstract history [14], which is possible because the latter does not have to be sequential. For example, the history in Figure 1(d) is linearized by that in Figure 1(e). This yields a simpler treatment of non-terminating calls than the use of completions in the classical definition of linearizability [18].

Definition 3.4 requires that the initial footprint of an abstract history H' be smaller than that of the concrete history H. This requirement is standard in data refinement [13]: it ensures that, when we replace a concrete library by an abstract one in a program, the library-owned memory stays disjoint from the client-owned one. It does not pose problems in practice, as the abstract library generating H' usually represents some of the data structures of the concrete library abstractly and, hence, more concisely.

So far we have defined the notion of linearizability on interface sets without taking into account library implementations that generate them. In the rest of the paper, we develop this notion for libraries written in a particular programming language and prove an Abstraction Theorem, which guarantees that a library can be replaced by another library linearizing it when we reason about its client program.

4. Programming Language

We consider a simple concurrent programming language:

$$\begin{array}{lcl} C & ::= & c \mid m \mid C; C \mid C + C \mid C^* \\ \mathcal{L} & ::= & \{m = C;\ \ldots;\ m = C\} \\ \mathcal{S} & ::= & \mathsf{let}\ \mathcal{L}\ \mathsf{in}\ C \parallel \ldots \parallel C \end{array}$$

A program consists of a single ***library*** $\mathcal{L}$ implementing methods $m \in \mathsf{Method}$ and its ***client*** $C_1 \parallel \ldots \parallel C_n$, given by a parallel composition of threads. The language is parameterised by a set of ***primitive commands*** $c \in \mathsf{PComm}$, meant to execute atomically. Commands also include method calls $m \in \mathsf{Method}$, sequential composition $C; C'$, nondeterministic choice $C + C'$ and finite iteration C^*. We use $+$ and * instead of conditionals and while loops for theoretical simplicity: as we show below, the latter can be defined in the language as syntactic sugar. Methods do not take arguments and do not return values, as these can be passed via special locations on the heap associated with the identifier of the thread calling the method (Section 3). We disallow nested method calls. We also assume that

every method called in the program is defined by the library, and thus call $\mathcal{S}$ a ***complete program***. An ***open program*** is a library $\mathcal{L}$ without a client, or a client $\mathcal{C}$ without a library implementation:

$$\begin{array}{rcl} \mathcal{C} & ::= & \mathsf{let}\ [-]\ \mathsf{in}\ C \parallel \ldots \parallel C \\ \mathcal{P} & ::= & \mathcal{S} \mid \mathcal{C} \mid \mathcal{L} \end{array}$$

In $\mathcal{C}$, we allow the client to call methods that are not defined in the program (but belong to the missing library). An open program represents a library or a client considered in isolation. The novelty of the kind of open programs we consider here is that we allow them to communicate with their environment via ownership transfers. We now define a way to specify a contract this communication follows.

4.1. **Method Specifications.** A ***predicate*** is a set of states from Σ, and a ***parameterised predicate*** is a mapping from thread identifiers to predicates. We use the same symbols p, q, r for ordinary and parameterised predicates; it should always be clear from the context which one we mean. When p is a parameterised predicate, we write p_t for the predicate obtained by applying p to a thread t. Both kinds of predicates can be described syntactically, e.g., using separation logic assertions [26].

We define possible ownership transfers between components with the aid of ***method specifications*** Γ, which are sets of Hoare triples $\{p\}\ m\ \{q\}$, at most one for each method. Here p and q are parameterised predicates such that p_t describes pieces of state transferred when thread t calls the method m, and q_t, those transferred at its return. Note that the intention of the pre- and postconditions in method specifications is only to identify the areas of memory transferred; in other words, they describe the "type" of the returned data structure, but not its "value". As usual for concurrent algorithms, a complete specification of a library is given by its abstract implementation (Section 6).

For example, as we discussed in Section 1, when programmers store pointers in a concurrent container, they often intend to transfer the ownership of the data structures these pointers identify at calls to and returns from the container's methods. In Figure 2(a) we give an example of such a container—a bounded stack represented by an array. For readability, we write examples in C instead of the minimalistic language introduced above. The library protects the array by a lock; more complicated algorithms allow a higher degree of concurrency [17]. Take $\Sigma = \mathsf{RAM}$ and for $x \in \mathsf{Loc}$ let $\mathsf{Obj}(x) \subseteq \mathsf{RAM}$ denote the set of states representing all well-formed data structures of a certain type allocated at the address x. For example, for objects with a single integer field we have $\mathsf{Obj}(x) = \{[x : y] \mid y \in \mathsf{Val}\}$. Then the specification of the stack when it stores pointers to such data structures can be given as follows:

$$\begin{array}{l} \{\exists x.\, \mathsf{arg}_t \mapsto x * \mathsf{Obj}(x)\}\ \mathtt{push}\ \{\mathsf{arg}_t \mapsto \mathtt{OK} \vee (\mathsf{arg}_t \mapsto \mathtt{FULL} * \mathsf{Obj}(x))\}; \\ \{\mathsf{arg}_t \mapsto \_\}\ \mathsf{pop}\ \{\exists x.\, \mathsf{arg}_t \mapsto x * ((x = \mathtt{EMPTY} \wedge \mathsf{emp}) \vee (x \neq \mathtt{EMPTY} \wedge \mathsf{Obj}(x)))\}. \end{array} \tag{4.1}$$

Here we use the separation logic syntax to describe predicates parameterised by the thread identifier t. Thus, emp denotes the empty heap $[\,]$, $x \mapsto y$ the heap $[x : y]$, and $*$ the combination of heaps with disjoint domains. In the following we also use the assertion $x..y \mapsto \_$ for $x \leq y$, denoting all the heaps with the domain $\{x, x+1, \ldots, y\}$. We use distinguished locations arg_t, $t \in \mathsf{ThreadID}$ to pass parameters and return values. According to the specification, the stack gets the ownership of an object when a pointer to it is pushed, and gives it up when the pointer is popped.

```
void *stack[SIZE];
int count = 0; // count of
// elements stored
Lock array_lock; // protects
// the array and the count

int push(void *arg) {
  lock(array_lock);
  if (count == SIZE) {
    unlock(array_lock);
    return FULL;
  }
  stack[count++] = arg;
  unlock(array_lock);
  return OK;
}

void *pop() {
  lock(array_lock);
  if (count == 0) {
    unlock(array_lock);
    return EMPTY;
  }
  void *obj = stack[--count];
  unlock(array_lock);
  return obj;
}
```

(a)

```
struct Node { Node *prev, *next; };
Node *free_list; // a cyclic doubly-
// linked list with a sentinel node
Lock *list_lock; // protects the list

void free(void *arg) {
  Node *block = (Node*)arg;
  lock(list_lock);
  block->prev = free_list;
  block->next = free_list->next;
  free_list->next->prev = block;
  free_list->next = block;
  unlock(list_lock);
}

void *alloc() {
  lock(list_lock);
  if (free_list->next == free_list) {
    unlock(list_lock);
    return NULL;
  }
  Node *block = free_list->next;
  free_list->next = block->next;
  block->next->prev = free_list;
  unlock(list_lock);
  return block;
}
```

(b)

Figure 2: Example concurrent library implementations: (a) a bounded stack storing pointers to objects; (b) a memory allocator managing memory blocks of a fixed size

Now take $size \in \mathbb{N}$ and let

$$\mathsf{Obj}(x) = \big\{\sigma \mid \mathsf{dom}(\sigma) = \{x, \ldots, x + size - 1\}\big\}.$$

We specify an allocator managing blocks of *size* memory cells as follows:

$$\begin{array}{l}\{\exists x.\, \mathsf{arg}_t \mapsto x * (x..(x + size - 1) \mapsto \_)\}\ \texttt{free}\ \{\mathsf{arg}_t \mapsto \_\};\\ \{\mathsf{arg}_t \mapsto \_\}\ \texttt{alloc}\ \{\exists x.\, \mathsf{arg}_t \mapsto x * ((x = 0 \wedge \mathsf{emp}) \vee (x \neq 0 \wedge (x..(x + size - 1) \mapsto \_)))\}.\end{array} \tag{4.2}$$

The specification corresponds to the ownership transfer reading of allocator calls explained in Section 1. In Figure 2(b) we give an example allocator corresponding to the specification (we have omitted initialisation code from the figure). Note that, unlike the stack, the allocator does access the blocks of memory transferred to it by the client, since it stores free-list pointers inside them.

To define the semantics of ownership transfers unambiguously (Section 5), we require pre- and postconditions in method specifications to be *precise* [23].

Definition 4.1. A predicate $r \in 2^\Sigma$ is ***precise*** if for every state σ there exists at most one substate σ_1 satisfying r, i.e., such that $\sigma_1 \in r$ and $\sigma = \sigma_1 * \sigma_2$ for some σ_2.

Since the $*$ operation is cancellative, when such a substate σ_1 exists, the corresponding substate σ_2 is unique and is denoted by $\sigma \setminus r$. A parameterised predicate r is precise if so is r_t for every t.

Informally, a precise predicate carves out a unique piece of the heap. For example, assuming the algebra RAM, the predicate $\{[42:0]\}$ and those used in the allocator specification are precise. However, the predicate $\{[42:0], [\,]\}$ is not: when $\sigma = [42:0]$ we can take either $\sigma_1 = [42:0]$ and $\sigma_2 = [\,]$, or $\sigma_1 = [\,]$ and $\sigma_2 = [42:0]$.

A ***specified open program*** is of the form $\Gamma \vdash \mathcal{C}$ or $\mathcal{L} : \Gamma$. In the former, the specification Γ describes all the methods that $\mathcal{C}$ may call. In the latter, Γ provides specifications for the methods in the open program that can be called by its external environment. In both cases, Γ specifies the type of another open program that can fill in the hole in $\mathcal{C}$ or $\mathcal{L}$. When we are not sure which form a program has, we write $\Gamma \vdash \mathcal{P} : \Gamma'$. In this case, if $\mathcal{P}$ does not have a client, then Γ is empty; if $\mathcal{P}$ does not have a library, then Γ' is empty; and if $\mathcal{P}$ is complete, then both Γ and Γ' are empty. For specified open programs

$$\Gamma \vdash \mathcal{C} \;=\; \Gamma \vdash \mathsf{let}\ [-]\ \mathsf{in}\ C_1 \parallel \ldots \parallel C_n$$

and

$$\mathcal{L} : \Gamma$$

agreeing on the specification Γ of library methods, we denote by $\mathcal{C}(\mathcal{L})$ the complete program

$$\mathsf{let}\ \mathcal{L}\ \mathsf{in}\ C_1 \parallel \ldots \parallel C_n.$$

4.2. **Primitive Commands.** We now discuss primitive commands in more detail. Consider the set $2^\Sigma \cup \{\top\}$ of subsets of Σ with a special element $\top$ used to denote an error state, resulting, e.g., from dereferencing an invalid pointer. We extend the $*$ operation on 2^Σ to $2^\Sigma \cup \{\top\}$ by letting $\top * p = p * \top = \top * \top = \top$ for all $p \in 2^\Sigma$.

We assume an interpretation of every primitive command $c \in \mathsf{PComm}$ as a transformer $f_c^t : \Sigma \to (2^\Sigma \cup \{\top\})$, which maps pre-states to states obtained when thread $t \in \mathsf{ThreadID}$ executes c from a pre-state. The fact that our transformers are parameterised by t allows atomic accesses to areas of memory indexed by thread identifiers. This idealisation simplifies the setting in that it lets us do without special thread-local or method-local storage for passing method parameters and return values.

Some typical primitive commands are:

$$\mathsf{skip}, \qquad [E] = E', \qquad \mathsf{assume}(E),$$

where expressions E are defined as follows:

$$E \;::=\; \mathbb{Z} \mid \mathsf{tid} \mid [E] \mid E + E \mid -E \mid !E \mid \ldots$$

Here tid refers to the identifier of the thread executing the command, $[E]$ returns the contents of the address E in memory, and $!E$ is the C-style negation of an expression E—it returns 1 when E evaluates to 0, and 0 otherwise. The $\mathsf{assume}(E)$ command filters out all the input states where E evaluates to 0. Hence, after $\mathsf{assume}(E)$ is executed, E always has a non-zero

$$
\begin{array}{llll}
\sigma, \mathsf{skip} & \rightsquigarrow_t & \sigma & \\
\sigma, [E] = E' & \rightsquigarrow_t & \sigma[[\![E]\!]_{\sigma,t} : [\![E']\!]_{\sigma,t}], & \text{if } [\![E]\!]_{\sigma,t} \in \mathsf{dom}(\sigma), [\![E']\!]_{\sigma,t} \in \mathsf{Val} \\
\sigma, [E] = E' & \rightsquigarrow_t & \top, & \text{if } [\![E]\!]_{\sigma,t} \notin \mathsf{dom}(\sigma) \text{ or } [\![E']\!]_{\sigma,t} = \top \\
\sigma, \mathsf{assume}(E) & \rightsquigarrow_t & \sigma, & \text{if } [\![E]\!]_{\sigma,t} \in \mathsf{Val} - \{0\} \\
\sigma, \mathsf{assume}(E) & \rightsquigarrow_t & \top, & \text{if } [\![E]\!]_{\sigma,t} = \top
\end{array}
$$

Figure 3: Transition relation for sample primitive commands in the RAM model. The result $\top$ indicates that the command faults.

$$
\begin{array}{llll}
\sigma, \mathsf{skip} & \rightsquigarrow_t & \sigma & \\
\sigma, [E] = E' & \rightsquigarrow_t & \sigma[[\![E]\!]_{\sigma,t} : ([\![E']\!]_{\sigma,t}, 1)], & \text{if } \sigma([\![E]\!]_{\sigma,t}) = (\_, 1), [\![E']\!]_{\sigma,t} \in \mathsf{Val} \\
\sigma, [E] = E' & \rightsquigarrow_t & \top, & \text{if the above condition does not hold} \\
\sigma, \mathsf{assume}(E) & \rightsquigarrow_t & \sigma, & \text{if } [\![E]\!]_{\sigma,t} \in \mathsf{Val} - \{0\} \\
\sigma, \mathsf{assume}(E) & \rightsquigarrow_t & \top, & \text{if } [\![E]\!]_{\sigma,t} = \top
\end{array}
$$

Figure 4: Transition relation for sample primitive commands in the RAM_π model. The evaluation of expressions $[\![E]\!]_{\sigma,t}$ ignores permissions in σ.

value. Using it, the standard commands for conditionals and loops can be defined in our language as follows:

$$
\begin{array}{rcl}
(\mathsf{if}\ E\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2) & = & (\mathsf{assume}(E); C_1) + (\mathsf{assume}(!E); C_2), \\
(\mathsf{while}\ E\ \mathsf{do}\ C) & = & (\mathsf{assume}(E); C)^*; \mathsf{assume}(!E).
\end{array}
\tag{4.3}
$$

For the above commands c and $t \in \mathsf{ThreadID}$, we define $f_c^t : \mathsf{RAM} \to 2^{\mathsf{RAM}} \cup \{\top\}$ using the transition relation $\rightsquigarrow_t$: $\mathsf{RAM} \times (\mathsf{RAM} \cup \{\top\})$ in Figure 3:

$$
f_c^t(\sigma) \;=\; \text{if } (\sigma, c \rightsquigarrow_t \top) \text{ then } \top \text{ else } \bigcup \{\sigma' \mid \sigma, c \rightsquigarrow_t \sigma'\}.
$$

In the figure, $[\![E]\!]_{\sigma,t} \in \mathsf{Val} \cup \{\top\}$ denotes the result of evaluating the expression E in the state σ with the current thread identifier t. When this evaluation dereferences illegal memory addresses, it results in the error value $\top$. We define $f_c^t : \mathsf{RAM}_\pi \to 2^{\mathsf{RAM}_\pi} \cup \{\top\}$ similarly, but using the transition relation $\rightsquigarrow_t$: $\mathsf{RAM}_\pi \times (\mathsf{RAM}_\pi \cup \{\top\})$ in Figure 4. The transformers formalise the semantics of permissions explained in Section 2: permissions less than 1 allow reading, and the full permission 1 additionally allows writing. Note that assume yields an empty set of post-states when its condition evaluates to zero, leading to the program getting stuck. Thus, even though, when executing the if statement (4.3), both branches of the non-deterministic choice will be explored, only the branch where the assume condition evaluates to true will proceed further.

For our results to hold, we need to place some restrictions on the transformers f_c^t for every primitive command $c \in \mathsf{PComm}$ and thread $t \in \mathsf{ThreadID}$:

Footprint Preservation: $\forall \sigma, \sigma' \in \Sigma.\, \sigma' \in f_c^t(\sigma) \implies \delta(\sigma') = \delta(\sigma)$.

Strong Locality: $\forall \sigma_1, \sigma_2 \in \Sigma.\, (\sigma_1 * \sigma_2)\!\downarrow \wedge\, f_c^t(\sigma_1) \neq \top \implies f_c^t(\sigma_1 * \sigma_2) = f_c^t(\sigma_1) * \{\sigma_2\}$.

Footprint Preservation prohibits primitive commands from allocating or deallocating memory. This does not pose a problem, since in the context of linearizability, an allocator is just another library and should be treated as such. The Strong Locality of f_c^t says that, if a

command c can be safely executed from a state σ_1, then when executed from a bigger state $\sigma_1 * \sigma_2$, it does not change the additional state σ_2 and its effect depends only on the state σ_1 and not on the additional state σ_2.

The Strong Locality is a strengthening of the locality property in separation logic [7]:

$$\forall \sigma_1, \sigma_2 \in \Sigma.\, (\sigma_1 * \sigma_2)\!\downarrow \wedge\, f_c^t(\sigma_1) \neq \top \implies f_c^t(\sigma_1 * \sigma_2) \subseteq f_c^t(\sigma_1) * \{\sigma_2\}.$$

Locality rules out commands that can check if a cell is allocated in the heap other than by trying to access it and faulting if it is not allocated. For example, let $\Sigma = \mathsf{RAM}$ and consider the following transformer $f^t : \mathsf{RAM} \to 2^{\mathsf{RAM}} \cup \{\top\}$:

$$f^t(\sigma) \;=\; \text{if } \sigma(1)\!\downarrow \text{ then } \{\sigma[1:0]\} \text{ else } \{\sigma\}.$$

The transformer f^t defines the denotation of a 'command' that writes 0 to the cell at the address 1 if it is allocated and acts as a no-op if it is not. This violates locality. Indeed, take $\sigma_1 = [\,]$ and $\sigma_2 = [1:1]$. Then

$$f^t(\sigma_1 * \sigma_2) = f^t([1:1]) = \{[1:0]\}$$

and

$$f^t(\sigma_1) * \{\sigma_2\} = f^t([\,]) * \{[1:1]\} = \{[\,]\} * \{[1:1]\} = \{[1:1]\}.$$

Hence, $f^t(\sigma_1 * \sigma_2) \subseteq f^t(\sigma_1) * \{\sigma_2\}$ does not hold.

While locality prohibits the command from changing the additional state, it permits the effect of the command to depend on this state [13]. The Strong Locality forbids such dependencies. To see this, consider another 'command' defined by the following transformer $f^t : \mathsf{RAM} \to 2^{\mathsf{RAM}} \cup \{\top\}$:

$$\begin{aligned} f^t(\sigma) \;=\; & \text{if } \sigma(1)\!\uparrow \text{ then } \top \\ & \text{else if } (\sigma(2)\!\downarrow) \text{ then } \{\sigma[1:0]\} \\ & \text{else } \{\sigma[1:0], \sigma[1:1]\}. \end{aligned}$$

The command does not access the cell at the address 2, since it does not fault if the cell is not allocated. However, when the cell is allocated, the effect of the command depends on its value. It is easy to check that f^t is local. However, it is not strongly local, since for $\sigma_1 = [1:0]$ and $\sigma_2 = [2:0]$, we have

$$f^t(\sigma_1 * \sigma_2) = f^t([1:0, 2:0]) = \{[1:0, 2:0]\}$$

and

$$f^t(\sigma_1) * \{\sigma_2\} = f^t([1:0]) * \{[2:0]\} = \{[1:0], [1:1]\} * \{[2:0]\} = \{[1:0, 2:0], [1:1, 2:0]\},$$

so that $f^t(\sigma_1 * \sigma_2) = f^t(\sigma_1) * \{\sigma_2\}$ does not hold. The property of Strong Locality subsumes the one of contents independence used in situations similar to ours in previous work on data refinement in a sequential setting [13].

The transformers for standard commands, except memory (de)allocation, satisfy the conditions of Footprint Preservation and Strong Locality.

5. Client-Local and Library-Local Semantics

We now give the semantics to complete and open programs. In the latter case, we define component-local semantics that include all behaviours of an open program under any environment satisfying the specification associated with it. In Section 6, we use these to lift linearizability to libraries and formulate the Abstraction Theorem.

Programs in our semantics denote sets of *traces*, recording every step in a computation. These include both internal actions by program components and calls and returns. We define program semantics in two stages. First, given a program, we generate the set of the possible execution traces of the program. This is done solely based on the structure of its statements, without taking into account restrictions arising from the semantics of primitive commands or ownership transfers. The next step filters out traces that are not consistent with the above restrictions using a trace evaluation process and, for open programs, annotates calls and returns appropriately.

5.1. Traces.

Traces consist of *actions*, which include primitive commands performed internally by a component and calls or returns, possibly annotated with states. Thus actions, include all interface actions ψ from Definition 3.1.

Definition 5.1. The set of ***actions*** is defined as follows:

$$\varphi \in \mathsf{Act} \ ::= \ \psi \mid (t, c) \mid (t, \mathsf{call}\ m) \mid (t, \mathsf{ret}\ m),$$

where $t \in \mathsf{ThreadID}$, $m \in \mathsf{Method}$ and $c \in \mathsf{PComm}$.

Definition 5.2. A ***trace*** τ is a finite sequence of actions such that for every thread t, the projection of τ to t's call and return actions is a sequence of alternating call and return actions over matching methods that starts from a call action.

We classify actions in a trace as those performed by the client and the library based on whether they happen inside a method.

Definition 5.3. For a trace τ and an index $i \in \{1, \ldots, |\tau|\}$, an action $\tau(i)$ is a ***client action*** if $\tau(i) = (t, c)$ for some thread t and a primitive command c and

$$\forall j.\, j < i \wedge \tau(j) = (t, \mathsf{call}\ \_) \implies \exists k.\, j < k < i \wedge \tau(k) = (t, \mathsf{ret}\ \_).$$

An action $\tau(i)$ is a ***library action*** if $\tau(i) = (t, c)$ for some t and c but $\tau(i)$ is not a client action, that is,

$$\exists j.\, j < i \wedge \tau(j) = (t, \mathsf{call}\ \_) \wedge \neg \exists k.\, j < k < i \wedge \tau(k) = (t, \mathsf{ret}\ \_).$$

A trace is a ***client trace***, if all of its actions of the form (t, c) are client actions; it is a ***library trace***, if they all of them are library actions.

In the following, κ denotes client traces, $\lambda, \zeta, \alpha, \beta$ library traces, and τ arbitrary ones. We write $\mathsf{client}(\tau)$ for the projection of τ to client, call and return actions, $\mathsf{lib}(\tau)$ for that to library, call and return actions, and $\mathsf{history}(\tau)$ for that to call and return actions.

$$
\begin{aligned}
(\!|c|\!)_t\eta &= \{(t, c)\} \\
(\!|C_1 + C_2|\!)_t\eta &= (\!|C_1|\!)_t\eta \cup (\!|C_2|\!)_t\eta \\
(\!|C^*|\!)_t\eta &= \left((\!|C|\!)_t\eta\right)^* \\
(\!|m|\!)_t\eta &= \{(t, \mathsf{call}\ m)\,\tau\,(t, \mathsf{ret}\ m) \mid \tau \in \eta(m, t)\} \\
(\!|C_1; C_2|\!)_t\eta &= \{\tau_1\tau_2 \mid \tau_1 \in (\!|C_1|\!)_t\eta \wedge \tau_2 \in (\!|C_2|\!)_t\eta\} \\
(\!|C_1 \parallel \ldots \parallel C_n|\!)\eta &= \bigcup\{\tau_1 \parallel \ldots \parallel \tau_n \mid \forall t \in \{1, \ldots, n\}.\, \tau_t \in (\!|C_t|\!)_t\eta\}
\end{aligned}
$$

$$
\begin{aligned}
(\!|\mathsf{let}\ \{m = C_m \mid m \in M\}\ \mathsf{in}\ C_1 \parallel \ldots \parallel C_n|\!) &= \mathsf{prefix}((\!|C_1 \parallel \ldots \parallel C_n|\!)(\boldsymbol{\lambda}(m, t).\, (\!|C_m|\!)_t(\_))) \\
(\!|\Gamma : \mathsf{let}\ [-]\ \mathsf{in}\ C_1 \parallel \ldots \parallel C_n|\!) &= \mathsf{prefix}((\!|C_1 \parallel \ldots \parallel C_n|\!)(\boldsymbol{\lambda}(m, t).\, \{\varepsilon\})) \\
(\!|\{m = C_m \mid m \in \{m_1, \ldots, m_j\}\} : \Gamma|\!) &= \\
&\mathsf{prefix}(\textstyle\bigcup_{k\geq 1}(\!|C_{\mathsf{mgc}} \parallel \ldots (k \text{ times}) \ldots \parallel C_{\mathsf{mgc}}|\!)(\boldsymbol{\lambda}(m, t).\, (\!|C_m|\!)_t(\_))) \\
&\qquad (\text{where } C_{\mathsf{mgc}} = (m_1 + \ldots + m_j)^*)
\end{aligned}
$$

Figure 5: Trace sets of commands and programs. Here $\mathsf{prefix}(T)$ is the prefix closure of T and $\tau \in \tau_1 \parallel \ldots \parallel \tau_n$ if and only if every action in τ is done by a thread $t \in \{1, \ldots, n\}$ and for all such t, we have $\tau|_t = \tau_t$. We use $\boldsymbol{\lambda}$ for functions, in contrast to λ for library traces.

5.2. **Trace Sets.** Consider a program $\Gamma \vdash \mathcal{P} : \Gamma'$ and let $M \subseteq \mathsf{Method}$ be the set of methods implemented by its library or called by its client. We define the trace set $(\!|\Gamma \vdash \mathcal{P} : \Gamma'|\!) \in 2^{\mathsf{Trace}}$ of $\mathcal{P}$ in Figure 5. We first define the trace set $(\!|C|\!)_t\eta$ of a command C, parameterised by the identifier t of the thread executing it and a mapping $\eta \in M \times \mathsf{ThreadID} \to 2^{\mathsf{Trace}}$ giving the trace set of the body of every method that C can call when executed by a given thread. The trace set of a client $(\!|C_1 \parallel \ldots \parallel C_n|\!)\eta$ is obtained by interleaving traces of its threads.

The trace set $(\!|\mathcal{C}(\mathcal{L})|\!)$ of a complete program is that of its client computed with respect to a mapping $\boldsymbol{\lambda}(m, t).\, (\!|C_m|\!)_t(\_)$ associating every method m with the trace set of its body C_m. Since we prohibit nested method calls, $(\!|C_m|\!)_t\eta$ does not depend on η. We prefix-close the resulting trace set to take into account incomplete executions. In particular, this allows the thread scheduler to be unfair: a thread can be preempted and never scheduled again.

A program $\Gamma \vdash \mathcal{C}$ generates client traces $(\!|\Gamma \vdash \mathcal{C}|\!)$, which do not include internal library actions. This is achieved by associating an empty trace with every library method. Finally, a program $\mathcal{L} : \Gamma'$ generates all possible library traces $(\!|\mathcal{L} : \Gamma'|\!)$. This is achieved by running the library under its *most general client*, where every thread executes an infinite loop, repeatedly invoking arbitrary library methods.

5.3. **Evaluation.** The set of traces generated using $(\!|\cdot|\!)$ may include those not consistent with the semantics of primitive commands or expected ownership transfers. We therefore define the meaning of a program $[\![\Gamma \vdash \mathcal{P} : \Gamma']\!] \in \Sigma \to (2^{\mathsf{Trace}} \cup \{\top\})$ by evaluating every trace in $(\!|\Gamma \vdash \mathcal{P} : \Gamma'|\!)$ from a given initial state to determine whether it is feasible. For open programs, this process also annotates calls and returns in a trace with states transferred.

The formal definition of $[\![\Gamma \vdash \mathcal{P} : \Gamma']\!]$ is given in Figure 6 with the aid of a trace evaluation function $[\![\Gamma \vdash \tau : \Gamma']\!] : \Sigma \to 2^{\Sigma \times \mathsf{Trace}} \cup \{\top\}$. Given an initial state, this either yields multiple final states and annotated traces, or fails and produces $\top$. If the resulting set

$$\begin{array}{l}
[\![\Gamma \vdash \mathcal{P} : \Gamma']\!] : \Sigma \to (2^{\mathsf{Trace}} \cup \{\top\}): \\
\qquad [\![\Gamma \vdash \mathcal{P} : \Gamma']\!]\sigma = \text{if } \exists \tau \in (\!|\Gamma \vdash \mathcal{P} : \Gamma'|\!).\, [\![\Gamma \vdash \tau : \Gamma']\!]\sigma = \top \text{ then } \top \\
\qquad\qquad\qquad\qquad \text{else } \{\tau' \mid \exists \tau \in (\!|\Gamma \vdash \mathcal{P} : \Gamma'|\!).\, (_, \tau') \in [\![\Gamma \vdash \tau : \Gamma']\!]\sigma\}
\end{array}$$

$$\begin{array}{rcl}
[\![\Gamma \vdash \tau : \Gamma']\!] : \Sigma \to (2^{\Sigma \times \mathsf{Trace}} \cup \{\top\}): & & \\
[\![\Gamma \vdash \varepsilon : \Gamma']\!]\sigma & = & \{(\sigma, \varepsilon)\} \\
[\![\Gamma \vdash \tau\varphi : \Gamma']\!]\sigma & = & \text{if } ([\![\Gamma \vdash \tau : \Gamma']\!]\sigma = \top) \text{ then } \top \\
& & \text{else if } (\exists(\sigma', _) \in [\![\Gamma \vdash \tau : \Gamma']\!]\sigma.\, [\![\Gamma \vdash \varphi : \Gamma']\!]\sigma' = \top) \text{ then } \top \\
& & \text{else } \{(\sigma'', \tau'\varphi') \mid \exists \sigma'.\, (\sigma', \tau') \in [\![\Gamma \vdash \tau : \Gamma']\!]\sigma \wedge (\sigma'', \varphi') \in [\![\Gamma \vdash \varphi : \Gamma']\!]\sigma'\}
\end{array}$$

$$\begin{array}{rcl}
[\![\Gamma \vdash \varphi : \Gamma']\!] : \Sigma \to (2^{\Sigma \times \mathsf{Act}} \cup \{\top\}): & & \\
[\![\Gamma \vdash (t, c) : \Gamma']\!]\sigma & = & \text{if } (f_c^t(\sigma) = \top) \text{ then } \top \text{ else } \{(\sigma', (t, c)) \mid \sigma' \in f_c^t(\sigma)\} \\
[\![(t, \mathsf{call}\ m)]\!]\sigma & = & \{(\sigma, (t, \mathsf{call}\ m))\} \\
[\![(t, \mathsf{ret}\ m)]\!]\sigma & = & \{(\sigma, (t, \mathsf{ret}\ m))\} \\
[\![(t, \mathsf{call}\ m) : (\{p\}\ m\ \{q\}), \Gamma']\!]\sigma & = & \{(\sigma * \sigma_p, (t, \mathsf{call}\ m(\sigma_p))) \mid \sigma_p \in p_t \wedge (\sigma * \sigma_p)\!\downarrow\} \\
[\![(t, \mathsf{ret}\ m) : (\{p\}\ m\ \{q\}), \Gamma']\!]\sigma & = & \text{if } (\sigma \setminus q_t)\!\uparrow \text{ then } \top \text{ else } \{(\sigma \setminus q_t, (t, \mathsf{ret}\ m(\sigma \setminus (\sigma \setminus q_t))))\} \\
[\![(\{p\}\ m\ \{q\}), \Gamma \vdash (t, \mathsf{call}\ m)]\!]\sigma & = & \text{if } (\sigma \setminus p_t)\!\uparrow \text{ then } \top \text{ else } \{(\sigma \setminus p_t, (t, \mathsf{call}\ m(\sigma \setminus (\sigma \setminus p_t))))\} \\
[\![(\{p\}\ m\ \{q\}), \Gamma \vdash (t, \mathsf{ret}\ m)]\!]\sigma & = & \{(\sigma * \sigma_q, (t, \mathsf{ret}\ m(\sigma_q))) \mid \sigma_q \in q_t \wedge (\sigma * \sigma_q)\!\downarrow\}
\end{array}$$

Figure 6: Semantics of programs

of state-trace pairs is empty, then the trace is infeasible and is discarded. If the evaluation produces $\top$ on any trace from $(\!|\Gamma \vdash \mathcal{P} : \Gamma'|\!)$, then the program has no semantics for the given initial state and its denotation is defined to be $\top$. The evaluation of τ is defined inductively on its length using a function $[\![\Gamma \vdash \varphi : \Gamma']\!] : \Sigma \to 2^{\Sigma \times \mathsf{Act}} \cup \{\top\}$ that evaluates a single action φ. We explain this function by considering separately the cases of a complete program, open program with a library and open program with a client.

The evaluation $[\![\varphi]\!]$ for an action in a complete program has the standard semantics, with the effects of primitive commands computed using their transformers from Section 4. In this case, calls and returns are left unannotated, since no ownership transfers to or from the external environment are performed.

The function $[\![\varphi : \Gamma']\!]$ gives a ***library-local*** semantics to the program $\mathcal{L} : \Gamma'$, in the sense that it generates library traces under any client respecting Γ'. When a method m from Γ' is called by thread t, the library receives the ownership of any state consistent with the method precondition p_t. This state has to be compatible with that of the library. After the method returns, the library has to give up the piece of state satisfying its postcondition. Since q_t is precise, this piece of state is determined uniquely. The evaluation faults if the state to be transferred is not available; thus, a library has no semantics if it violates the contract with its client given by Γ'. This also ensures that the histories produced by a library are balanced.

Proposition 5.4. *If* $\lambda \in [\![\mathcal{L} : \Gamma']\!]\sigma$*, then* $\mathsf{history}(\lambda)$ *is balanced from* $\delta(\sigma)$*.*

The function $[\![\Gamma \vdash \varphi]\!]$ gives a ***client-local*** semantics to $\Gamma \vdash \mathcal{C}$, in the sense that it generates traces of this client assuming any behaviour of the library consistent with Γ. When a thread t calls a method m in Γ, it transfers the ownership of a piece of state

satisfying the method precondition p_t to the library being called. As before, this piece is defined uniquely, because preconditions are precise. When such a piece of state is not available, the evaluation faults. This ensures that client respects the method specifications of the libraries it uses. When the method returns, the client receives the ownership of an arbitrary piece of state satisfying its postcondition q_t, compatible with the current state of the client.

5.4. **Connection between Local and Global Semantics.** We now formulate a lemma, used in the proof of the Abstraction Theorem (Section 6), that states the connection between the library-local and client-local semantics on one side and the semantics of complete programs on the other. We start by introducing some auxiliary definitions.

Definition 5.5. A program $\Gamma \vdash \mathcal{P} : \Gamma'$ is ***safe*** at σ, if $[\![\Gamma \vdash \mathcal{P} : \Gamma']\!]\sigma \neq \top$; $\mathcal{P}$ is safe for $I \subseteq \Sigma$, if it is safe at σ for all $\sigma \in I$.

For a set of initial states $I \subseteq \Sigma$, let

$$[\![(\Gamma \vdash \mathcal{P} : \Gamma'), I]\!] = \{(\sigma, \tau) \mid \sigma \in I \wedge \tau \in [\![\Gamma \vdash \mathcal{P} : \Gamma']\!]\sigma\}.$$

We define an operator $\otimes : 2^{\Sigma \times \mathsf{Trace}} \times 2^{\Sigma \times \mathsf{Trace}} \to 2^{\Sigma \times \mathsf{Trace}}$ combining the resulting sets X and Y of state-trace pairs produced by the client-local and library-local semantics into a set corresponding to the complete program:

$$X \otimes Y = \{(\sigma * \sigma', \tau) \mid \exists \kappa, \lambda.\, (\sigma, \kappa) \in X \wedge (\sigma', \lambda) \in Y \wedge (\sigma * \sigma')\downarrow \wedge \mathsf{cover}(\tau, \kappa, \lambda)\},$$

where

$$\mathsf{cover}(\tau, \kappa, \lambda) \iff \mathsf{history}(\kappa) = \mathsf{history}(\lambda) \wedge \mathsf{client}(\tau) = \mathsf{ground}(\kappa) \wedge \mathsf{lib}(\tau) = \mathsf{ground}(\lambda)$$

and ground is a function on traces that erases the state annotations from their interface actions.

Lemma 5.6. *Assume $\Gamma \vdash \mathcal{C}$ and $\mathcal{L} : \Gamma$ safe for I_0 and I_1, respectively. Then $\mathcal{C}(\mathcal{L})$ is safe for $I_0 * I_1$ and*

$$[\![\mathcal{C}(\mathcal{L}), I_0 * I_1]\!] = [\![\Gamma \vdash \mathcal{C}, I_0]\!] \otimes [\![\mathcal{L} : \Gamma, I_1]\!].$$

The lemma shows that the set of traces produced by $\mathcal{C}(\mathcal{L})$ can be obtained by combining pairs of traces with the same history produced by $\mathcal{C}$ and $\mathcal{L}$. Note that, since the semantics of $\mathcal{C}(\mathcal{L})$ does not annotate calls and returns with the states transferred, in cover we have to erase these annotations from the local traces κ or λ before comparing the traces with τ. Unpacking the definition of $\otimes$ and using the fact that

$$\forall \kappa, \lambda.\, \mathsf{history}(\kappa) = \mathsf{history}(\lambda) \implies \exists \tau.\, \mathsf{cover}(\tau, \kappa, \lambda),$$

from Lemma 5.6 we get the following two corollaries.

Corollary 5.7 (Decomposition)**.** *Assume $\Gamma \vdash \mathcal{C}$ and $\mathcal{L} : \Gamma$ safe for I_0 and I_1, respectively. Then $\mathcal{C}(\mathcal{L})$ is safe for $I_0 * I_1$ and*

$$\forall (\sigma, \tau) \in [\![\mathcal{C}(\mathcal{L}), I_0 * I_1]\!].\, \exists (\sigma_0, \kappa) \in [\![\Gamma \vdash \mathcal{C}, I_0]\!].\, \exists (\sigma_1, \lambda) \in [\![\mathcal{L} : \Gamma, I_1]\!].$$
$$\sigma = \sigma_0 * \sigma_1 \wedge \mathsf{cover}(\tau, \kappa, \lambda).$$

Corollary 5.8 (Composition). *If* $\Gamma \vdash \mathcal{C}$ *and* $\mathcal{L} : \Gamma$ *are safe for* I_0 *and* I_1*, respectively, then*

$$\forall (\sigma_1, \kappa) \in [\![\Gamma : \mathcal{C}, I_0]\!]. \forall (\sigma_2, \lambda) \in [\![\mathcal{L} : \Gamma, I_1]\!]. ((\sigma_0 * \sigma_1)\downarrow \wedge \mathsf{history}(\kappa) = \mathsf{history}(\lambda)) \implies \exists \tau. (\sigma_0 * \sigma_1, \tau) \in [\![\mathcal{C}(\mathcal{L}), I_0 * I_1]\!] \wedge \mathsf{cover}(\tau, \kappa, \lambda).$$

Corollary 5.7 can be viewed as carrying over properties of the local semantics, such as safety, to the global one, and in this sense is the statement of the soundness of the former with respect to the latter. The corollary also confirms that the client defined by $(\!|\mathcal{L} : \Gamma'|\!)$ and $[\![\lambda : \Gamma']\!]$ is indeed most general, as it reproduces library behaviours under any possible clients. Corollary 5.8 carries over properties of the global semantics to the local ones, stating the adequacy of the latter.

Lemma 5.6 is proved in Appendix A. Most of the proof deals with maintaining a splitting of the state of $\mathcal{C}(\mathcal{L})$ into the parts owned by $\mathcal{L}$ and $\mathcal{C}$, which changes during ownership transfers. The proof relies crucially on the safety of the client and the libraries and the Strong Locality property of primitive commands. In more detail, safety is defined by considering executions of a component in the library-local or the client-local semantics. These execute the component code only on the memory it owns, whose amount only changes with ownership transfers to and from its environment according to method specifications. Because of the Strong Locality property, commands fault when accessing memory cells that are not present in the state they are run from, and their execution does not depend on any additional memory that might be present in the state. Hence, when we use a component inside a complete program, its safety guarantees that the component code does not touch the part of the heap belonging to other components in the program, and its execution is not affected by the state of such components. This guarantees that the behaviour a component produces as part of the complete program can be reproduced when we execute it in isolation and vice versa, allowing us to establish Lemma 5.6. In practice, the safety of a program can be established using existing program logics, such as separation logic [23, 28].

6. Abstraction Theorem

We are now in a position to define the notion of linearizability on libraries and prove the central technical result of this paper—the Abstraction Theorem. We define linearizability between specified libraries $\mathcal{L} : \Gamma$, together with their sets of initial states I. First, using the library-local semantics of Section 5, we define the interface set describing all the behaviours of a library $\mathcal{L}$ when run from initial states in I:

$$\mathsf{interf}(\mathcal{L} : \Gamma, I) = \{(\delta(\sigma_0), \mathsf{history}(\tau)) \mid (\sigma_0, \tau) \in [\![(\mathcal{L} : \Gamma), I]\!]\} \subseteq \mathsf{BHistory}.$$

Definition 6.1. Consider $\mathcal{L}_1 : \Gamma$ and $\mathcal{L}_2 : \Gamma$ safe for I_1 and I_2, respectively. We say that $(\mathcal{L}_1 : \Gamma, I_1)$ ***is linearized by*** $(\mathcal{L}_2 : \Gamma, I_2)$, written $(\mathcal{L}_1 : \Gamma, I_1) \sqsubseteq (\mathcal{L}_2 : \Gamma, I_2)$, if, according to Definition 3.4,

$$\mathsf{interf}(\mathcal{L}_1 : \Gamma, I_1) \sqsubseteq \mathsf{interf}(\mathcal{L}_2 : \Gamma, I_2).$$

For an interface set $\mathcal{H}_2$ we say that $(\mathcal{L}_1 : \Gamma, I_1)$ ***is linearized by*** $\mathcal{H}_2$, written $(\mathcal{L}_1 : \Gamma, I_1) \sqsubseteq \mathcal{H}_2$, if

$$\mathsf{interf}(\mathcal{L}_1 : \Gamma, I_1) \sqsubseteq \mathcal{H}_2.$$

Thus, $(\mathcal{L}_1 : \Gamma, I_1)$ is linearized by $(\mathcal{L}_2 : \Gamma, I_2)$ if every history generated by the library-local semantics of the former may be reproduced in a linearized form by the library-local semantics of the latter without requiring more memory. The relation $(\mathcal{L}_1 : \Gamma, I_1) \sqsubseteq (\mathcal{L}_2 :$

```
Sequence<void*> stack;

int push(void *arg) {
  atomic {
    if (nondet()) { return FULL; }
    else {
      add_to_head(stack, arg);
      return OK;
    }
  }
}

void *pop() {
  atomic {
    if (!isEmpty(stack)) {
      void *obj = head(stack);
      stack = tail(stack);
      return obj;
    } else { return EMPTY; }
  }
}
```

(a)

```
Set<void*> free_list;

void free(void *arg) {
  atomic {
    add(free_list, arg);
  }
}

void *alloc() {
  atomic {
    if (!isEmpty(free_list)) {
      Node *block =
          (Node*)take(free_list);
      block->next = nondet();
      block->prev = nondet();
      return block;
    } else {
      return 0;
    }
  }
}
```

(b)

Figure 7: Specifications corresponding to the implementations in Figure 2: (a) a bounded stack storing pointers to objects; (b) a memory allocator managing memory blocks of a fixed size. The `Node` structure is defined in Figure 2(b).

Γ, I_2) allows us to specify a library by another piece of code, but possibly simpler than the original one. For example, the stack and the allocator from Figure 2 with method specifications (4.1) and (4.2) can be specified by the libraries in Figure 7. The libraries replace the array and the linked list in the implementations by the abstract data types of a sequence and a set (we assume a trivial extension of the RAM algebra from Section 2 to allow memory cells to store values of such types). Thus, the abstract libraries use less memory than the concrete ones. Instead of using locking, all operations on the abstract data types are done atomically; formally, we assume primitive commands corresponding to the code in the atomic blocks.

The other relation $(\mathcal{L}_1 : \Gamma, I_1) \sqsubseteq \mathcal{H}_2$ introduced in Definition 6.1 allows us to specify a library directly by an interface set, without fixing a piece of code generating it. The interface set $\mathcal{H}_2$ can still be simpler than that of $(\mathcal{L}_1 : \Gamma, I_1)$, e.g., containing only sequential histories. Even though the two forms of defining linearizability may seem very similar, as we show in Section 6.1, their mathematical properties are fundamentally different.

We now formulate two variants of the Abstraction Theorem, corresponding to the two ways of specifying libraries (we prove them in Section 6.1).

Theorem 6.2 (Abstraction—specification by code)**.** *If*

- $\mathcal{L}_1 : \Gamma$, $\mathcal{L}_2 : \Gamma$, $\Gamma \vdash \mathcal{C}$ *are safe for* I_1, I_2, I, *respectively, and*

- $(\mathcal{L}_1 : \Gamma, I_1) \sqsubseteq (\mathcal{L}_2 : \Gamma, I_2)$,

then

- $\mathcal{C}(\mathcal{L}_1)$ *and* $\mathcal{C}(\mathcal{L}_2)$ *are safe for* $I * I_1$ *and* $I * I_2$*, respectively, and*
- $\forall (\sigma_1, \tau_1) \in [\![\mathcal{C}(\mathcal{L}_1), I * I_1]\!]. \exists (\sigma_2, \tau_2) \in [\![\mathcal{C}(\mathcal{L}_2), I * I_2]\!]. \mathsf{client}(\tau_1) = \mathsf{client}(\tau_2)$.

Thus, when reasoning about a client $\mathcal{C}(\mathcal{L}_1)$ of a library $\mathcal{L}_1$, we can soundly replace $\mathcal{L}_1$ by a library $\mathcal{L}_2$ linearizing it: if a safety property over client traces holds of $\mathcal{C}(\mathcal{L}_2)$, it will also hold of $\mathcal{C}(\mathcal{L}_1)$. In practice, we are usually interested in ***atomicity abstraction***, a special case of this transformation when methods in $\mathcal{L}_2$ are atomic. An instance is replacing one of the libraries from Figure 2 by its specification from Figure 7. The requirement that $\mathcal{C}$ be safe in the theorem restricts its applicability to well-behaved clients that do not access memory owned by the library: you cannot replace a library by another one if the client can access its internal data structures and thereby "look inside the box". Similarly, the safety of the libraries ensures that they cannot corrupt the data structures owned by the client.

The other version of the Abstraction Theorem, allowing library specification by an interface set, guarantees that replacing a library by its specification leaves all the original client behaviours reproducible modulo the following notion of trace equivalence.

Definition 6.3. Client traces κ and κ' are ***equivalent***, written $\kappa \sim \kappa'$, if $\kappa|_t = \kappa'|_t$ for all $t \in \mathsf{ThreadID}$ and the projections of κ and κ' to non-interface actions are identical.

Theorem 6.4 (Abstraction—specification by an interface set)**.** *If*

- $\mathcal{L}_1 : \Gamma$ *and* $\Gamma \vdash \mathcal{C}$ *are safe for* I_1 *and* I*, respectively, and*
- $(\mathcal{L}_1, I_1) \sqsubseteq \mathcal{H}_2$,

then

- $\mathcal{C}(\mathcal{L}_1)$ *is safe for* $I * I_1$ *and*
- $\forall (\sigma, \tau_1) \in [\![\mathcal{C}(\mathcal{L}_1), I * I_1]\!]. \exists \kappa, l. (\sigma', \kappa) \in [\![\Gamma \vdash \mathcal{C}, I]\!] \wedge (l, \mathsf{history}(\kappa)) \in \mathcal{H}_2 \wedge (\delta(\sigma') \circ l)\downarrow \wedge \mathsf{client}(\tau_1) \sim \mathsf{ground}(\kappa)$.

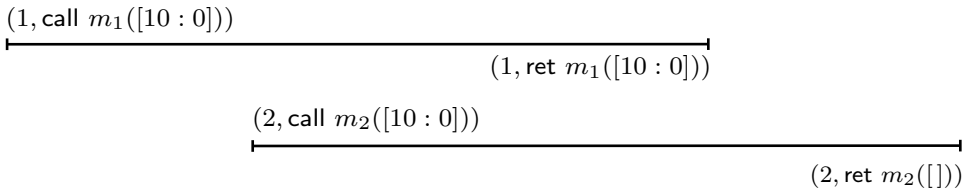
The theorem shows that client behaviours of $\mathcal{C}(\mathcal{L}_1)$ can be reproduced by the client-local semantics of $\mathcal{C}$ projected to histories in $\mathcal{H}_2$ with initial footprints compatible with initial client states. Note that $\mathsf{client}(\tau_1) = \mathsf{client}(\tau_2)$ in Theorem 6.2 implies that $\mathsf{history}(\tau_1) = \mathsf{history}(\tau_2)$, i.e., $\mathcal{C}(\mathcal{L}_2)$ can reproduce the history of $\mathcal{C}(\mathcal{L}_1)$ exactly. In contrast, Theorem 6.4 does not guarantee this, since $\kappa \sim \kappa'$ does not imply $\mathsf{history}(\kappa) = \mathsf{history}(\kappa')$; we only know that the projection to non-interface actions is reproduced. We discuss the reason for this discrepancy below.

6.1. **The Rearrangement Lemma and the Proof of the Abstraction Theorem.** The key component used for establishing Theorem 6.2 is the Rearrangement Lemma: if $H \sqsubseteq H'$, then every execution trace of a library producing H' can be transformed into another trace of the same library that differs from the original one only in the order of interface actions and produces H, instead of H'. Hence, the library specification can simulate any behaviour of its implementation the client can expect.

Lemma 6.5 (Rearrangement—library)**.** *If* $(\delta(\sigma), H) \sqsubseteq (\delta(\sigma'), H')$ *and* $\mathcal{L} : \Gamma$ *is safe at* σ'*, then*

$$\forall \lambda' \in [\![\mathcal{L} : \Gamma]\!]\sigma'. \mathsf{history}(\lambda') = H' \implies \exists \lambda \in [\![\mathcal{L} : \Gamma]\!]\sigma'. \mathsf{history}(\lambda) = H.$$

(a):



(b):

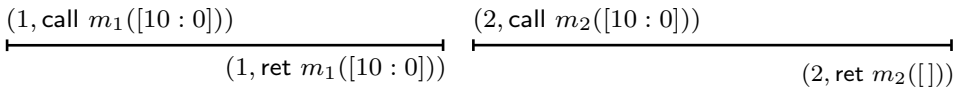


Figure 8: Counterexample showing the need for history balancedness in Lemma 6.5

The proof of the lemma is highly non-trivial and is a subject of Section 7. We point out Lemma 6.5 would not hold had we included unbalanced histories in our definition of linearizability. To show this, take $\Sigma = \mathsf{RAM}$ and consider the histories in Figure 8. In Figure 8(b) the library receives the cell 10 from the client, then returns it and then receives it again. Even though the history in Figure 8(a) is linearized by that in Figure 8(b), the former is not balanced, and by Proposition 5.4, cannot be produced by $\mathcal{L}$. This shows that Lemma 6.5 does not hold for unbalanced H.

Note that we have $H \sqsubseteq H$ for any history H. As a consequence, from Lemma 6.5 we obtain the following surprising result, stating that linearizability between libraries is equivalent to inclusion between the sets of histories they produce.

Corollary 6.6. *If $\mathcal{L}_1 : \Gamma$ and $\mathcal{L}_2 : \Gamma$ are safe for I_1 and I_2, respectively, then*

$$(\mathcal{L}_1 : \Gamma, I_1) \sqsubseteq (\mathcal{L}_2 : \Gamma, I_2) \iff \forall (l, H) \in \mathsf{interf}(\mathcal{L}_1 : \Gamma, I_1).\, \exists l'.\, l' \preceq l \wedge (l', H) \in \mathsf{interf}(\mathcal{L}_2 : \Gamma, I_2).$$

This fact is not a consequence of ownership transfer and also holds for the classical notion of linearizability. Intuitively, Lemma 6.5, and hence, Corollary 6.6 hold due to a closure property of the semantics of the language from Section 4. Namely, in this and other programming languages, there may always be a delay between the point when a library method is called and when it starts executing and, conversely, when it ends executing and when the control returns to the client. For example, when executing the code in Figure 7(a), there may be delays between a call to `push`, the execution of the atomic block and the return from `push`. Hence, this library can produce both of the histories in Figures 1(b) and 1(c).

Due to this property of the program semantics, a trace from $(\!|\mathcal{L}|\!)$, e.g., one producing the history in Figure 1(c), will stay valid if we execute some of the calls in it earlier and returns later, like in Figure 1(b). This is also (usually) safe given the ownership transfer reading of calls and returns in the library-local semantics defined by $[\![\mathcal{L} : \Gamma]\!]$: it just means that the library receives state from the client earlier and gives it up later. The proof of Lemma 6.5 uses such transformations on a history to "de-linearize" it, e.g., transforming

the history in Figure 1(c) into that in Figure 1(b). The above closure property is also the reason for Theorem 6.2 guaranteeing that $\mathcal{C}(\mathcal{L}_2)$ can reproduce the history of $\mathcal{C}(\mathcal{L}_1)$ exactly.

Given that replacing a library $\mathcal{L}_1$ by its linearization $\mathcal{L}_2$ does not simplify its interface set, can Theorem 6.2 really simplify reasoning about a complete program $\mathcal{C}(\mathcal{L}_1)$? Fortunately, the answer is yes, since the point of the theorem is to simplify *the code* of this program. For example, replacing the library in Figure 2(a) by the one in Figure 7(a) allows us to pretend in reasoning about a complete program that changes to the library state, shared between different threads, are atomic, and thus consider fewer possible thread interleavings. Calls and returns in such a complete program are merely thread-local operations that do not complicate reasoning. In Section 6.2, we discuss an example of using Theorem 6.2 to simplify proofs of complicated algorithms.

Specifying a library by an interface set $\mathcal{H}_2$ instead of code, as in Theorem 6.4, does not allow us to get results such as Lemma 6.5 and Corollary 6.6, since the set $\mathcal{H}_2$ is not guaranteed to satisfy any closure properties. For example, it might contain only sequential histories, where every call is immediately followed by the corresponding return without a delay. In fact, $\mathcal{H}_2$ has to be simpler than the interface set of $\mathcal{L}_1$ for Theorem 6.4 to be useful, since this is what the theorem replaces $\mathcal{L}_1$ by. Fortunately, to prove Theorem 6.4 we can exploit a closure property of the client-local semantics, formalised by the following variant of the Rearrangement Lemma: any client trace can be transformed into an equivalent one with a given history linearizing the history of the original one.

Lemma 6.7 (Rearrangement—client)**.** *If* $(l, H) \sqsubseteq (l', H')$ *and* $\Gamma \vdash \mathcal{C}$ *is safe at* σ*, then*

$$\forall \kappa \in [\![\Gamma \vdash \mathcal{C}]\!]\sigma.\, (\delta(\sigma) \circ l)\!\downarrow \wedge\, \mathsf{history}(\kappa) = H \implies \exists \kappa' \in [\![\mathcal{C}]\!]\sigma.\, \mathsf{history}(\kappa') = H' \wedge \kappa \sim \kappa'.$$

Intuitively, the lemma holds because, in the client-local semantics, it is safe to execute calls later and returns earlier. Like Lemma 6.5, this lemma would not hold if we allowed H' to be unbalanced.

In summary, when a library is specified by the code of its abstract implementation, the ability to linearize a concrete history while looking for a matching abstract one allowed by Definition 3.4 is not strictly needed. However, it is indispensable when the library is specified directly by a set of histories. We were able to obtain this insight into the original definition of linearizability by formalising the guarantees the linearizability of a library provides to its clients as Abstraction Theorems.

Using Lemmas 6.5 and 6.7, we now prove the two versions of the Abstraction Theorem.

Proof of Theorem 6.2. The safety of $\mathcal{C}(\mathcal{L}_1)$ and $\mathcal{C}(\mathcal{L}_2)$ follows from Corollary 5.7. Take $(\sigma, \tau_1) \in [\![\mathcal{C}(\mathcal{L}_1), I * I_1]\!]$. We transform the trace τ_1 of $\mathcal{C}(\mathcal{L}_1)$ into a trace τ_2 of $\mathcal{C}(\mathcal{L}_2)$ with the same client projection using the local semantics of $\mathcal{L}_1$, $\mathcal{L}_2$ and $\mathcal{C}$. Namely, we first apply Corollary 5.7 to generate a pair $(\sigma_{\mathbf{l}}^1, \lambda_1) \in [\![\mathcal{L}_1 : \Gamma, I_1]\!]$ of a library-local initial state and a trace and a client-local pair $(\sigma_{\mathbf{c}}, \kappa) \in [\![\Gamma \vdash \mathcal{C}, I]\!]$, such that

$$\sigma = \sigma_{\mathbf{c}} * \sigma_{\mathbf{l}}^1 \;\wedge\; \mathsf{client}(\tau_1) = \mathsf{ground}(\kappa) \wedge\; \mathsf{history}(\kappa) = \mathsf{history}(\lambda_1). \tag{6.1}$$

Since $(\mathcal{L}_1 : \Gamma, I_1) \sqsubseteq (\mathcal{L}_2 : \Gamma, I_2)$, for some $(\sigma_{\mathbf{l}}^2, \lambda_2) \in [\![\mathcal{L}_2 : \Gamma, I_2]\!]$, we have

$$(\delta(\sigma_{\mathbf{l}}^1), \mathsf{history}(\lambda_1)) \sqsubseteq (\delta(\sigma_{\mathbf{l}}^2), \mathsf{history}(\lambda_2)),$$

which implies $\delta(\sigma_{\mathbf{l}}^2) \preceq \delta(\sigma_{\mathbf{l}}^1)$. By Lemma 6.5, λ_2 can be transformed into a trace λ_2' such that

$$((\sigma_{\mathbf{l}}^2, \lambda_2') \in [\![\mathcal{L}_2 : \Gamma, I_2]\!]) \;\wedge\; (\mathsf{history}(\lambda_2') = \mathsf{history}(\lambda_1) = \mathsf{history}(\kappa)).$$

Since $\delta(\sigma_{\mathbf{l}}^2) \preceq \delta(\sigma_{\mathbf{l}}^1)$ and $(\sigma_{\mathbf{c}} * \sigma_{\mathbf{l}}^1)\downarrow$, we have $(\sigma_{\mathbf{c}} * \sigma_{\mathbf{l}}^2)\downarrow$. We then use Corollary 5.8 to compose the library-local trace λ_2' with the client-local one κ into a trace τ_2 such that

$$((\sigma_{\mathbf{c}} * \sigma_{\mathbf{l}}^2, \tau_2) \in [\![\mathcal{C}(\mathcal{L}_2), I * I_2]\!]) \ \wedge \ (\mathsf{client}(\tau_2) = \mathsf{ground}(\kappa) = \mathsf{client}(\tau_1)). \qquad \square$$

The above proof scheme can be described mnemonically as 'decompose, rearrange, compose'. We reuse its first two steps to prove Theorem 6.4.

Proof of Theorem 6.4. Take $(\sigma, \tau_1) \in [\![\mathcal{C}(\mathcal{L}_1), I * I_1]\!]$. Like in the proof of Theorem 6.2, we apply Corollary 5.7 to generate $(\sigma_{\mathbf{l}}^1, \lambda_1) \in [\![\mathcal{L}_1, I_1]\!]$ and $(\sigma_{\mathbf{c}}, \kappa) \in [\![\Gamma \vdash \mathcal{C}, I]\!]$ such that (6.1) holds. Since $(\mathcal{L}_1 : \Gamma, I_1) \sqsubseteq \mathcal{H}_2$, for some $(l_2, H_2) \in \mathcal{H}_2$, we have

$$(l_2 \preceq \delta(\sigma_{\mathbf{l}}^1)) \ \wedge \ ((\delta(\sigma_{\mathbf{l}}^1), \mathsf{history}(\kappa)) = (\delta(\sigma_{\mathbf{l}}^1), \mathsf{history}(\lambda_1)) \sqsubseteq (l_2, H_2)).$$

Then by Lemma 6.7, κ can be transformed into a trace κ', such that

$$(\sigma_{\mathbf{c}}, \kappa') \in [\![\Gamma \vdash \mathcal{C}, I]\!] \ \wedge \ \mathsf{history}(\kappa') = H_2 \ \wedge \ \kappa \sim \kappa'$$

Since $\mathsf{client}(\tau_1) = \mathsf{ground}(\kappa)$, we thus have $\mathsf{client}(\tau_1) \sim \mathsf{ground}(\kappa')$. Furthermore, since $l_2 \preceq \delta(\sigma_{\mathbf{l}}^1)$ and $(\sigma_{\mathbf{c}} * \sigma_{\mathbf{l}}^1)\downarrow$, we have $(\delta(\sigma_{\mathbf{c}}) \circ l_2)\downarrow$. Hence, κ' and l_2 are the required trace and footprint. $\square$

6.2. **Establishing and Using Linearizability with Ownership Transfer.** Our preliminary investigations show that linearizability with ownership transfer can be established by generalising existing proof systems for proving classical linearizability based on separation logic [28]. The details of such a generalisation are out of the scope of this paper; we plan to report on it in the future.

The Abstraction Theorem is not just a theoretical result: it enables compositional reasoning about complex concurrent algorithms that are challenging for existing verification methods. For example, the theorem can be used to justify Vafeiadis's compositional proof [28, Section 5.3] of the multiple-word compare-and-swap (MCAS) algorithm implemented using an auxiliary operation called RDCSS [16] (the proof used an abstraction of the kind enabled by Theorem 6.2 without justifying its correctness). If the MCAS algorithm were verified together with RDCSS, its proof would be extremely complicated. Fortunately, we can consider MCAS as a client of RDCSS, with the two components performing ownership transfers between them. The Abstraction Theorem then makes the proof tractable by allowing us to verify the linearizability of MCAS assuming an atomic specification of the inner RDCSS algorithm.

7. Proof of the Rearrangement Lemma

We only give the proof of Lemma 6.5, as that of Lemma 6.7 is completely symmetric.

The proof transforms λ' into λ by repeatedly swapping adjacent actions in it according to a certain strategy to make the history of the trace equal to H. The most subtle place in the proof is swapping

$$(t_1, \mathsf{ret}\ m_1(\sigma_1))\,(t_2, \mathsf{call}\ m_2(\sigma_2))$$

to yield

$$(t_2, \mathsf{call}\ m_2(\sigma_2))\,(t_1, \mathsf{ret}\ m_1(\sigma_1)),$$

where $t_1 \neq t_2$. This case is subtle for the following reason. Let the state of the library $\mathcal{L}$ before the return action be θ; then $(\theta \setminus \sigma_1)\!\downarrow$ and the state of the library after executing the return and the call is $(\theta \setminus \sigma_1) * \sigma_2$. For the swapping to be possible, we need $(\theta * \sigma_2)\!\downarrow$; then by Proposition 2.2

$$(\theta \setminus \sigma_1) * \sigma_2 = (\theta * \sigma_2) \setminus \sigma_1, \tag{7.1}$$

which can be used to establish that the resulting trace is still produced by $\mathcal{L}$. However, $(\theta * \sigma_2)\!\downarrow$ is not guaranteed if the history H is arbitrary. For example, take $\Sigma = \mathsf{RAM}$ and let H and H' be defined by Figures 8(a) and 8(b). Since H is unbalanced, it cannot be produced by any library, and hence, we cannot swap

$$(1, \mathsf{ret}\ m_1([10:0]))\,(2, \mathsf{call}\ m_2([10:0]))$$

in H'. In our proof we use the fact that the history H is balanced to show that a situation in which we cannot swap a return followed by a call while transforming λ' into λ cannot happen. This is non-trivial, as the problematic situation can potentially happen midway through the transformation. We only know that the target history H of λ is balanced, but this does not straightforwardly imply that the histories of the intermediate traces obtained while transforming λ' into λ are, since these histories might be quite different from H. Inferring their balancedness from that of H represents the most challenging part of the proof.

We therefore first do the proof under an assumption that allows swapping a return followed by a call easily and consider the general case later. This lets us illustrate the overall idea of the proof, which is then reused in the additional part of the proof dealing with the challenge presented by the general case. Namely, we make the following assumption:

$$\begin{array}{l}\Sigma = \mathsf{RAM} \text{ and for any } \zeta \in [\![\mathcal{L} : \Gamma]\!]\sigma' \text{ and interface actions } \psi_1 = (t_1, \_\ \_(\sigma_1)) \\ \text{and } \psi_2 = (t_2, \_\ \_(\sigma_2)) \text{ in } \zeta, \text{ if } t_1 \neq t_2, \text{ then } \mathsf{dom}(\sigma_1) \cap \mathsf{dom}(\sigma_2) = \emptyset.\end{array} \tag{7.2}$$

For example, this holds when states transferred between the client and the library are always thread-local. It is easy to check that in RAM, if $(\theta \setminus \sigma_1)\!\downarrow$, $((\theta \setminus \sigma_1) * \sigma_2)\!\downarrow$ and $(\sigma_1 * \sigma_2)\!\downarrow$, then $(\theta * \sigma_2)\!\downarrow$ and thus (7.1) holds. Hence, (7.2) allows us to justify swapping a return followed by a call in a trace easily. We now proceed to prove Lemma 6.5 under this assumption. In our proof, we use the assumption in a single place, which we note explicitly; the rest of the proof is independent from it.

Below we sometimes write $\sqsubseteq_\rho$ instead of $\sqsubseteq$ to make the bijection ρ used to establish the relation between histories in Definition 3.4 explicit. For a bijection ρ between histories H and H', we write $\mathsf{id}_k(\rho)$ if ρ is an identity on the first k actions in H.

Take $\sigma, \sigma' \in \Sigma$ and consider a trace $\lambda' \in [\![\mathcal{L} : \Gamma]\!]\sigma'$. Assume histories H, H' such that $\mathsf{history}(\lambda') = H'$ and $(\delta(\sigma), H) \sqsubseteq (\delta(\sigma'), H')$, so that H is balanced from $\delta(\sigma)$ and H' from $\delta(\sigma')$. We prove that there exists a trace $\lambda \in [\![\mathcal{L} : \Gamma]\!]\sigma'$ such that $\mathsf{history}(\lambda) = H$. To this end, we define a finite sequence of steps that transforms λ' into a such a trace λ. The main idea of the transformation is to make progressively longer prefixes of the trace have histories coinciding with prefixes of H. Namely, the transformation is done in stages, and on stage $k = 0, 1, 2, \ldots, |H|$ we obtain a trace $\alpha_k \in [\![\mathcal{L} : \Gamma]\!]\sigma'$, where $\alpha_0 = \lambda'$. Every one of these traces is such that for some prefix β_k of α_k we have:

$$\begin{array}{c}\mathsf{history}(\beta_k) = H\!\downarrow_k; \\ \exists \rho.\, ((\delta(\sigma), H) \sqsubseteq_\rho (\delta(\sigma'), \mathsf{history}(\alpha_k))) \wedge \mathsf{id}_k(\rho); \\ \forall j.\, 0 \leq j < k \implies (\beta_j \text{ is a prefix of } \beta_k).\end{array}$$

We let $\beta_0 = \varepsilon$, so that the above conditions are initially satisfied. Thus, during the transformation, progressively longer prefixes β_k of α_k have histories coinciding with prefixes of H, while the linearizability relation between the history H and that of α_k is preserved. We then take $\alpha_{|H|}$ as the desired trace λ.

The trace α_{k+1} is constructed from the trace α_k by applying the following lemma for $\lambda_1 = \beta_k$, $\lambda_1\lambda_2 = \alpha_k$, $H_1 = H\downarrow_k$, $H_1\psi H_2 = H$, $\alpha_{k+1} = \lambda_1\lambda_2'\lambda_2''$ and $\beta_{k+1} = \lambda_1\lambda_2'$.

Lemma 7.1. *Assume (7.2) holds. Consider a history $H_1\psi H_2$ and a trace $\lambda_1\lambda_2 \in [\![\mathcal{L} : \Gamma]\!]\sigma'$ such that*

$$\mathsf{history}(\lambda_1) = H_1; \tag{7.3}$$

$$\exists\rho.\,((\delta(\sigma), H_1\psi H_2) \sqsubseteq_\rho (\delta(\sigma'), \mathsf{history}(\lambda_1\lambda_2))) \wedge \mathsf{id}_{|H_1|}(\rho). \tag{7.4}$$

Then there exist traces λ_2' and λ_2'' such that $\lambda_1\lambda_2'\lambda_2'' \in [\![\mathcal{L} : \Gamma]\!]\sigma'$ and

$$\mathsf{history}(\lambda_1\lambda_2') = H_1\psi; \tag{7.5}$$

$$\exists\rho'.\,((\delta(\sigma), H_1\psi H_2) \sqsubseteq_{\rho'} (\delta(\sigma'), \mathsf{history}(\lambda_1\lambda_2'\lambda_2''))) \wedge \mathsf{id}_{|H_1\psi|}(\rho'). \tag{7.6}$$

To prove Lemma 7.1, we convert $\lambda_1\lambda_2$ into $\lambda_1\lambda_2'\lambda_2''$ by swapping adjacent actions in the trace a finite number of times while preserving its properties of interest. These transformations are described by the following proposition, which formalises the closure properties of the library-local semantics we alluded to in Section 6.1.

Proposition 7.2. *Let $\mathcal{L} : \Gamma$ be safe at σ_0 and consider $\zeta \in [\![\mathcal{L} : \Gamma]\!]\sigma_0$ and a history S such that $S \sqsubseteq_\rho \mathsf{history}(\zeta)$. Then swapping any two adjacent actions $\varphi_1\varphi_2$ in ζ executed by different threads such that*

(i) $\varphi_1 \in \mathsf{Act} - \mathsf{RetAct}$, $\varphi_2 \in \mathsf{CallAct}$*; or*
(ii) $\varphi_1 \in \mathsf{RetAct}$, $\varphi_2 \in \mathsf{CallAct}$, *$\varphi_2$ precedes φ_1 in S, and (7.2) holds; or*
(iii) $\varphi_1 \in \mathsf{RetAct}$, $\varphi_2 \in \mathsf{Act} - \mathsf{CallAct}$

yields a trace $\zeta' \in [\![\mathcal{L} : \Gamma]\!]\sigma_0$ such that $S \sqsubseteq_{\rho'} \mathsf{history}(\zeta')$ for the bijection ρ' defined as follows. If $\varphi_1 \notin \mathsf{CallRetAct}$ or $\varphi_2 \notin \mathsf{CallRetAct}$, then $\rho' = \rho$. Otherwise, let i be the index of φ_1 in S. Then $\rho'(i+1) = \rho(i)$, $\rho'(i) = \rho(i+1)$ and $\rho'(k) = \rho(k)$ for $k \notin \{i, i+1\}$.

Since, in the library-local semantics, the library gains state at a call and gives it up at a return, intuitively, the transformation in the proposition allows the library to gain state earlier (i, ii) and give it up later (iii). The assumption that φ_2 precede φ_1 in case (ii) is needed to ensure that the transformation does not violate the linearizability relation. The proof of case (ii) is the only place where the assumption (7.2) is used.

Proof sketch for Proposition 7.2. Consider $\zeta = \zeta_1\varphi_1\varphi_2\zeta_2 \in [\![\mathcal{L} : \Gamma]\!]\sigma_0$ and let $\zeta' = \zeta_1\varphi_2\varphi_1\zeta_2$. The proof of the required linearizability relationship is trivial. It therefore remains to show that $(\_, \zeta') \in [\![\mathcal{L} : \Gamma]\!]\sigma_0$. We know that for some α we have $\alpha \in (\!|\mathcal{L}|\!)$ and $(\_, \zeta) \in [\![\alpha]\!]\sigma_0$. Let $\alpha = \alpha_1\varphi_1'\varphi_2'\alpha_2$ and $\alpha' = \alpha_1\varphi_2'\varphi_1'\alpha_2$, where φ_1' and φ_2' correspond to φ_1 and φ_2. It is easy to see that $\alpha' \in (\!|\mathcal{L}|\!)$. It therefore remains to show that $\zeta' \in [\![\alpha']\!]\sigma_0$. The proof proceeds by case analysis on the kind of actions φ_1 and φ_2. The justification of the case when φ_1 is a return and φ_2 is a call follows from (7.2) by the argument given earlier. Out of the remaining cases, we only consider a single illustrative one: $\varphi_1 = (t_1, c)$ and $\varphi_2 = (t_2, \mathsf{call}\ m_2(\sigma))$ for $t_1 \neq t_2$.

Assume $(\theta', \zeta_1\varphi_1\varphi_2) \in [\![\alpha_1\varphi_1'\varphi_2']\!]\sigma_0$. Then for some θ we have $(\theta, \zeta_1) \in [\![\alpha_1]\!]\sigma_0$, $f_c^{t_1}(\theta) \neq \top$ and $\theta' \in f_c^{t_1}(\theta) * \{\sigma\}$. By the Footprint Preservation property, we get $(\theta * \sigma)\downarrow$. Then by

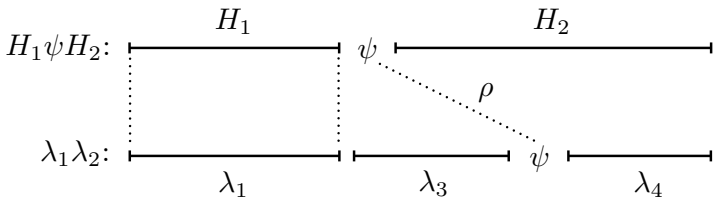


Figure 9: Illustration of the proof of Lemma 7.1

the Strong Locality property,

$$\theta' \in f_c^{t_1}(\theta) * \{\sigma\} = f_c^{t_1}(\theta * \sigma).$$

Hence, $(\theta', \zeta_1\varphi_2\varphi_1) \in [\![\alpha_1\varphi_2'\varphi_1']\!]\sigma_0$. Thus, Footprint Preservation and Strong Locality guarantee that a call can be safely executed earlier than a primitive command. □

Proof of Lemma 7.1. From (7.3) and (7.4) it follows that $\lambda_2 = \lambda_3\psi\lambda_4$ for some traces λ_3 and λ_4, where ρ maps ψ in $H_1\psi H_2$ to the ψ action shown in λ_2; see Figure 9. We consider two cases.

1. $\psi \in \mathsf{CallAct}$. Let t be the thread executing ψ. By (7.4) we have $(H_1\psi H_2)|_t = (\mathsf{history}(\lambda_1\lambda_2))|_t$. Then, since $\lambda_1\lambda_2$ is a library trace, (7.3) implies that there are no actions by thread t in λ_3. Furthermore, for any return action φ in λ_3, the action in $H_1\psi H_2$ corresponding to it according to ρ is in H_2. Thus, we can move the action ψ to the position between λ_1 and λ_3 by swapping it with adjacent actions a finite number of times as described in Proposition 7.2(i, ii). As a result, we obtain the trace $\lambda_1\psi\lambda_3\lambda_4 \in [\![\mathcal{L} : \Gamma]\!]\sigma'$. Conditions (7.5)–(7.6) then follow from Proposition 7.2(i, ii) for $\lambda_2' = \psi$ and $\lambda_2'' = \lambda_3\lambda_4$.

2. $\psi \in \mathsf{RetAct}$. Assume that λ_3 contains a call action φ, so that it precedes the return action ψ in $\mathsf{history}(\lambda_1\lambda_2)$. Then by (7.3) and (7.4) the action in $H_1\psi H_2$ corresponding to φ according to ρ is in H_2 and thus follows ψ in $H_1\psi H_2$. This violates the preservation of the order of non-overlapping method invocations required by (7.4). Hence, there are no call actions in λ_3. Since $\lambda_1\lambda_2$ is a library trace, this implies that for any action $\varphi = (t, \mathsf{ret}\ \_)$ in λ_3 there are no actions by the thread t in λ_3 following φ. Thus, we can move all return actions in the subtrace λ_3 of $\lambda_1\lambda_2$ to the position between ψ and λ_4 by swapping them with adjacent actions a finite number of times as described in Proposition 7.2(iii). We thus obtain the trace $\lambda_1\lambda_3'\psi\lambda_4'\lambda_4 \in [\![\mathcal{L} : \Gamma]\!]\sigma'$, where λ_4' consists of all return actions in λ_3, and λ_3' of the rest of actions in the subtrace; in particular, λ_3' does not contain any interface actions. Conditions (7.5)–(7.6) then follow from Proposition 7.2(iii) for $\lambda_2' = \lambda_3'\psi$ and $\lambda_2'' = \lambda_4'\lambda_4$. □

This completes the proof of Lemma 6.5 under the assumption (7.2). Let us now lift this assumption and consider the only place in the proof of the lemma that relies on it—that when we swap a return followed by a call using Proposition 7.2(ii) in case 1 in the proof of Lemma 7.1. Let us now identify precise conditions under which this situation happens. Let $\psi = (t_1, \mathsf{call}\ m_1(\sigma_1))$ and let the adjacent return action with which we are trying to swap it be $(t_2, \mathsf{ret}\ m_2(\sigma_2))$. Let S^1 be the target history $H_1\psi H_2$ from Lemma 7.1 and S^2 be the history of the trace in which we are trying to swap the return and the call. Then the two

histories are of the form

$$\begin{aligned} S^1 &= S\,(t_1, \mathsf{call}\ m_1(\sigma_1))\,S_1\,(t_2, \mathsf{ret}\ m_2(\sigma_2))\,S_2; \\ S^2 &= SS_1'\,(t_2, \mathsf{ret}\ m_2(\sigma_2))\,(t_1, \mathsf{call}\ m_1(\sigma_1))\,S_2' \end{aligned} \tag{7.7}$$

for some S, S_1, S_2, S_1', S_2'. Furthermore, from the conditions of Lemma 7.1, we have:

(i) $t_1 \neq t_2$;
(ii) S^1 and S^2 are balanced from some l_1 and l_2, respectively, such that $l_2 \preceq l_1$; and
(iii) $S^1 \sqsubseteq_\rho S^2$, where $\mathsf{id}_{|S|}(\rho)$ and ρ maps $(t_1, \mathsf{call}\ m_1(\sigma_1))$ and $(t_2, \mathsf{ret}\ m_2(\sigma_2))$ in S^1 to the corresponding actions shown in S^2.

As the following proposition shows, we can always do the desired transformation if the history

$$SS_1'\,(t_1, \mathsf{call}\ m_1(\sigma_1))\,(t_2, \mathsf{ret}\ m_2(\sigma_2))\,S_2' \tag{7.8}$$

resulting from swapping the return and the call in S^2 is balanced from l_2.

Proposition 7.3. *Let $\mathcal{L} : \Gamma$ be safe at σ_0. Consider traces*

$$\zeta = \zeta_1\,(t_2, \mathsf{ret}\ m_2(\sigma_2))\,(t_1, \mathsf{call}\ m_1(\sigma_1))\,\zeta_2 \in [\![\mathcal{L} : \Gamma]\!]\sigma_0$$

and

$$\zeta' = \zeta_1\,(t_1, \mathsf{call}\ m_1(\sigma_1))\,(t_2, \mathsf{ret}\ m_2(\sigma_2))\,\zeta_2.$$

If $\mathsf{history}(\zeta')$ is balanced from $\delta(\sigma_0)$, then $\zeta' \in [\![\mathcal{L} : \Gamma]\!]\sigma_0$.

Thus, the only problematic case we have is when the history (7.8) is not balanced from l_2. We summarise all the conditions under which such case can happen in the following definition.

Definition 7.4. Histories S^1 and S^2 of the form (7.7) are ***conflicting*** if the conditions (i)–(iii) above are satisfied and the history (7.8) is not balanced from l_2.

Given Proposition 7.3, the only case when the transformation in the proof of Lemma 7.1 can fail to convert the trace is when $H_1\psi H_2$ and the history currently being transformed are conflicting. Thus, with the assumption (7.2) lifted, Lemma 7.1 turns into

Lemma 7.5. *Consider a history $H_1\psi H_2$, and a trace $\lambda_1\lambda_2 \in [\![\mathcal{L} : \Gamma]\!]\sigma'$ such that (7.3) and (7.4) hold. Then either $H_1\psi H_2$ and another history composed of actions from $\mathsf{history}(\lambda_1\lambda_2)$ are conflicting, or there exist traces λ_2' and λ_2'' such that $\lambda_1\lambda_2'\lambda_2'' \in [\![\mathcal{L} : \Gamma]\!]\sigma'$ and (7.5) and (7.6) hold.*

We now show that no conflicting pairs of histories exist, hence guaranteeing that Lemma 7.5 can always be used to construct α_{k+1} from α_k in transforming λ' into λ. This completes the proof of Lemma 6.5 in the general case.

We first discuss the main idea of the proof. The history S^1 in (7.7) is similar to (7.8) in that the call precedes the return. We would like to use the fact that S^1 is balanced to prove that so is (7.8), thereby yielding a contradiction. As we noted at the beginning of this section, this is not straightforward due to the differences in the form of the histories S^1 and S^2 other than the precedence of the two call and return actions. We resolve this problem by adjusting the strategy we used above to transform λ' into λ under the assumption (7.2) to iron out the differences between S^1 and S^2. In particular, we use a variant of the transformation that, when the process of moving the call action ψ to the left in $\lambda_1\lambda_2$ gets stuck (Figure 9), leaves the corresponding action in $H_1\psi H_2$ unmatched and continues bringing the rest of the trace $\lambda_1\lambda_2$ in sync with the target history.

Lemma 7.6. *There are no conflicting pairs of histories.*

Proof. Consider histories S^1 and S^2 satisfying the conditions in Definition 7.4. Since S^2 is balanced from l_2,

$$[\![SS_1'\,(t_2, \mathsf{ret}\ m_2(\sigma_2))]\!]^\sharp l_2 = ([\![SS_1']\!]^\sharp l_2) \setminus\!\setminus \delta(\sigma_2)$$

is defined. Assume

$$[\![SS_1'\,(t_1, \mathsf{call}\ m_1(\sigma_1))]\!]^\sharp l_2 = ([\![SS_1']\!]^\sharp l_2) \circ \delta(\sigma_1)$$

is defined. Since $([\![SS_1']\!]^\sharp l_2) \setminus\!\setminus \delta(\sigma_2)$ is defined, by Proposition 2.7, we have:

$$\begin{aligned}[\![SS_1'\,(t_1, \mathsf{call}\ m_1(\sigma_1))\,(t_2, \mathsf{ret}\ m_2(\sigma_2))]\!]^\sharp l_2 &= (([\![SS_1']\!]^\sharp l_2) \circ \delta(\sigma_1)) \setminus\!\setminus \delta(\sigma_2) = \\ (([\![SS_1']\!]^\sharp l_2) \setminus\!\setminus \delta(\sigma_2)) \circ \delta(\sigma_1) &= [\![SS_1'\,(t_2, \mathsf{ret}\ m_2(\sigma_2))\,(t_1, \mathsf{call}\ m_1(\sigma_1))]\!]^\sharp l_2.\end{aligned}$$

Then (7.8) is balanced from l_2, contradicting our assumptions. Hence, $(([\![SS_1']\!]^\sharp l_2) \circ \delta(\sigma_1))\uparrow$.

A call action in S_1' cannot be in S_2: in this case it would follow $(t_2, \mathsf{ret}\ m_2(\sigma_2))$ in S^1, but precede it in S^2, contradicting $S^1 \sqsubseteq S^2$. Hence, all call actions in S_1' are in S_1. Let $S_1 = S_3S_4$, where S_3 is the minimal prefix of S_1 containing all call actions from S_1'. Then

$$\begin{aligned}S^1 &= S\,(t_1, \mathsf{call}\ m_1(\sigma_1))\,S_3S_4\,(t_2, \mathsf{ret}\ m_2(\sigma_2))\,S_2;\\ S^2 &= SS_1'\,(t_2, \mathsf{ret}\ m_2(\sigma_2))\,(t_1, \mathsf{call}\ m_1(\sigma_1))\,S_2'.\end{aligned}$$

If S_3 is non-empty, any return action in it precedes its last call action, which is also in S_1'. Since $S^1 \sqsubseteq S^2$, such a return action also has to be in S_1'. Thus, all return actions in S_3 are in S_1'.

The traces S^1 and S^2 are of the following more general form, obtained by letting $S_0 = S\,(t_1, \mathsf{call}\ m_1(\sigma_1))$ and $S_0' = S$:

$$\begin{aligned}S^1 &= S_0S_3S_4\,(t_2, \mathsf{ret}\ m_2(\sigma_2))\,S_2;\\ S^2 &= S_0'S_1'\,(t_2, \mathsf{ret}\ m_2(\sigma_2))\,(t_1, \mathsf{call}\ m_1(\sigma_1))\,S_2',\end{aligned}$$

where

- S^1 and S^2 are balanced from some l_1 and l_2, respectively, such that $l_2 \preceq l_1$;
- S_0 and S_0' are identical, except S_0 may have some extra call actions;
- $S^1 \sqsubseteq_\rho S^2$;
- ρ^{-1} maps all call actions in S_1' to actions in S_3;
- ρ maps all return actions in S_3 to actions in S_1';
- ρ^{-1} maps actions in S_0' to those in S_0, in particular $(t_1, \mathsf{call}\ m_1(\sigma_1))$ to an action in S_0, and $(t_2, \mathsf{ret}\ m_2(\sigma_2))$ to the same action shown in S^1; and
- $(([\![S_0'S_1']\!]^\sharp l_2) \circ \delta(\sigma_1))\uparrow$.

We denote this form by (**F**). The additional call actions in S_0 are the ones for which the transformation in Lemma 7.1 failed. The conditions relating S_3 and S_1' imply that S_3 may have more calls than S_1', and S_1' more returns than S_3. Thus, intuitively, S_0S_3 gains more state than $S_0'S_1'$, including that transferred by $(t_1, \mathsf{call}\ m_1(\sigma_1))$, and $S_0'S_1'$ gives up more than S_0S_3. In the following, we use this and the fact that S^1 is balanced from a bigger footprint than S^2 to show that $(([\![S_0'S_1']\!]^\sharp l_2) \circ \delta(\sigma_1))\downarrow$, thereby yielding a contradiction.

To this end, we describe a process that transforms the histories S^1 and S^2 into another pair of histories satisfying the conditions above, but such that S_3 is strictly smaller.

Repeatedly applying this process, we can make S_3 empty, obtaining histories satisfying (**F**):

$$\begin{array}{l} S_0 S_4 \, (t_2, \mathsf{ret}\ m_2(\sigma_2)) \, S_2; \\ S_0' S_1' \, (t_2, \mathsf{ret}\ m_2(\sigma_2)) \, (t_1, \mathsf{call}\ m_1(\sigma_1)) \, S_2'. \end{array} \tag{7.9}$$

In particular, $((\llbracket S_0' S_1' \rrbracket^\sharp l_2) \circ \delta(\sigma_1))\uparrow$. Before describing the transformation process, we show that, given the above pair of histories, we can obtain a contradiction. We use the following simple proposition, proved in Appendix A.1.

Proposition 7.7. *Assume S is identical to S', except it may have extra calls, and S and S' are balanced from l_1 and l_2, respectively, such that $l_2 \preceq l_1$. Then the $\circ$-combination l_c of footprints of states transferred at the extra call actions in S is defined, S' is balanced from l_1 and*

$$\llbracket S \rrbracket^\sharp l_1 = (\llbracket S' \rrbracket^\sharp l_1) \circ l_c \ \wedge\ \llbracket S' \rrbracket^\sharp l_2 \preceq \llbracket S' \rrbracket^\sharp l_1.$$

Consider the histories in (7.9). Since all calls from S_1' are in $S_3 = \varepsilon$, S_1' contains only returns. Since the histories are balanced from l_1 and l_2, respectively, $\llbracket S_0 \rrbracket^\sharp l_1$ and $\llbracket S_0' \rrbracket^\sharp l_2$ are defined. The history S_0 is identical to S_0', except it may have extra calls. By Proposition 7.7, the $\circ$-combination of footprints of states transferred at the extra call actions in S_0 is defined. Since an action $(t_1, \mathsf{call}\ m_1(\sigma_1))$ is in S_0, but not in S_0', this combination is of the form $\delta(\sigma_1) \circ l_c$ for some l_c; hence,

$$\llbracket S_0 \rrbracket^\sharp l_1 = (\llbracket S_0' \rrbracket^\sharp l_1) \circ \delta(\sigma_1) \circ l_c \ \wedge\ \llbracket S_0' \rrbracket^\sharp l_2 \preceq \llbracket S_0' \rrbracket^\sharp l_1.$$

Therefore, $(\llbracket S_0' \rrbracket^\sharp l_2) \circ \delta(\sigma_1)$ is defined. Since S_1' contains only return actions,

$$\llbracket S_0' S_1' \rrbracket^\sharp l_2 = (\llbracket S_0' \rrbracket^\sharp l_2) \setminus\!\setminus l',$$

where l' is the $\circ$-combination of the footprints of states transferred at these actions. This implies

$$\llbracket S_0' \rrbracket^\sharp l_2 = (\llbracket S_0' S_1' \rrbracket^\sharp l_2) \circ l',$$

where both expressions are defined. But then so is

$$(\llbracket S_0' \rrbracket^\sharp l_2) \circ \delta(\sigma_1) = (\llbracket S_0' S_1' \rrbracket^\sharp l_2) \circ l' \circ \delta(\sigma_1).$$

Hence, $(\llbracket S_0' S_1' \rrbracket^\sharp l_2) \circ \delta(\sigma_1)$ is defined, contradicting the opposite fact established above. This contradiction implies that a conflicting pair of histories does not exist.

Now assume arbitrary histories S^1 and S^2 satisfying (**F**):

$$\begin{array}{l} S^1 = S_0 S_3 S_4 \, (t_2, \mathsf{ret}\ m_2(\sigma_2)) \, S_2; \\ S^2 = S_0' S_1' \, (t_2, \mathsf{ret}\ m_2(\sigma_2)) \, (t_2, \mathsf{call}\ m_1(\sigma_1)) \, S_2', \end{array}$$

We show that from these we can construct another pair of histories satisfying (**F**), but with S_3 strictly smaller. We use the same transformation as in the proof of Lemma 7.1. When this transformation gets stuck, we obtain another pair of histories of the form (**F**), but again with a smaller S_3.

Let us make a case split on the next action in S_3.

- $S_3 = (t, \mathsf{call}\ m(\sigma)) \, S_5$, such that the action corresponding to $(t, \mathsf{call}\ m(\sigma))$ according to ρ is not in S_1'. In this case we let $S_0 := S_0 \, (t, \mathsf{call}\ m(\sigma))$ and $S_3 := S_5$. Thus, the call action $(t, \mathsf{call}\ m(\sigma))$ unmatched in S_1' becomes part of S_0.

- $S_3 = (t, \mathsf{call}\ m(\sigma))\, S_5$, such that the action corresponding to $(t, \mathsf{call}\ m(\sigma))$ according to ρ is in S_1'. Let $S_1' = S_3'\,(t, \mathsf{call}\ m(\sigma))S_4'$, so that
$$S^1 = S_0\,(t, \mathsf{call}\ m(\sigma))\, S_5 S_4\,(t_2, \mathsf{ret}\ m_2(\sigma_2))\, S_2;$$
$$S^2 = S_0' S_3'\,(t, \mathsf{call}\ m(\sigma))\, S_4'\,(t_2, \mathsf{ret}\ m_2(\sigma_2))\,(t_2, \mathsf{call}\ m_1(\sigma_1))\, S_2'.$$
Using the transformations from case 1 in the proof of Lemma 7.1, we can try to move the action $(t, \mathsf{call}\ m(\sigma))$ to the position between S_0' and S_3' while preserving the balancedness of the history. If this succeeds, we construct a new pair of histories of the form (**F**) by letting $S_0 := S_0\,(t, \mathsf{call}\ m(\sigma))$, $S_3 := S_5$, $S_0' := S_0'\,(t, \mathsf{call}\ m(\sigma))$ and $S_1' := S_3' S_4'$. Otherwise, we get a pair of conflicting histories, which are of the form (**F**) but with a smaller S_3. Again, in this case the unmatched call action $(t, \mathsf{call}\ m(\sigma))$ becomes part of S_0.
- $S_3 = (t, \mathsf{ret}\ m(\sigma))\, S_5$. Then the action corresponding to $(t, \mathsf{ret}\ m(\sigma))$ according to ρ is also in S_1', so that $S_1' = S_3'\,(t, \mathsf{ret}\ m(\sigma))\, S_4'$:
$$S^1 = S_0\,(t, \mathsf{ret}\ m(\sigma))\, S_5 S_4\,(t_2, \mathsf{ret}\ m_2(\sigma_2))\, S_2;$$
$$S^2 = S_0' S_3'\,(t, \mathsf{ret}\ m(\sigma))\, S_4'\,(t_2, \mathsf{ret}\ m_2(\sigma_2))\,(t_2, \mathsf{call}\ m_1(\sigma_1))\, S_2'.$$
Using the transformations from case 2 in the proof of Lemma 7.1, we can move the return action to the position between S_0' and S_3' while preserving the balancedness of the history. We thus obtain a pair of histories:
$$S^1 = S_0\,(t, \mathsf{ret}\ m(\sigma))\, S_5 S_4\,(t_2, \mathsf{ret}\ m_2(\sigma_2))\, S_2;$$
$$S^2 = S_0'\,(t, \mathsf{ret}\ m(\sigma))\, S_3' S_4'\,(t_2, \mathsf{ret}\ m_2(\sigma_2))\,(t_2, \mathsf{call}\ m_1(\sigma_1))\, S_2'.$$
Then we can let $S_0 := S_0\,(t, \mathsf{ret}\ m(\sigma))$, $S_3 := S_5$, $S_0' := S_0'\,(t, \mathsf{ret}\ m(\sigma))$ and $S_1' := S_3' S_4'$. □

8. Frame Rule for Linearizability

Libraries such as concurrent containers are used by clients to transfer the ownership of data structures, but do not actually access their contents. We show that for such libraries, the classical linearizability implies linearizability with ownership transfer.

Definition 8.1. A method specification $\Gamma' = \{\{r^m\}\ m\ \{s^m\} \mid m \in M\}$ ***extends*** a specification $\Gamma = \{\{p^m\}\ m\ \{q^m\} \mid m \in M\}$, if $\forall t.\, r_t^m \subseteq p_t^m * \Sigma \wedge s_t^m \subseteq q_t^m * \Sigma$.

For example, the method specification (4.1) of the stack in Figure 2(a) extends the following specification:

$$\begin{array}{l}\{\exists x.\, \mathsf{arg}_t \mapsto x\}\ \mathtt{push}\ \{\mathsf{arg}_t \mapsto \mathtt{OK} \vee \mathsf{arg}_t \mapsto \mathtt{FULL}\};\\ \{\exists y.\, \mathsf{arg}_t \mapsto y\}\ \mathtt{pop}\ \{\exists x.\, \mathsf{arg}_t \mapsto x\}.\end{array} \tag{8.1}$$

According to this specification, `push` just receives an arbitrary pointer x as a parameter; in contrast, the specification (4.1) additionally mandates that the object the pointer identifies be transferred to the library. We now identify conditions under which the linearizability between a pair of libraries satisfying Γ entails that of the same libraries satisfying an extended method specification Γ'. This yields a result somewhat analogous to the frame rule of separation logic [26].

We start by introducing some auxiliary definitions. We first define operations for mapping between histories corresponding to extended and non-extended method specifications.

For the method specification Γ from Definition 8.1, we define operations $\lfloor\!\lfloor\cdot\rfloor\!\rfloor_\Gamma$ and $\lceil\!\lceil\cdot\rceil\!\rceil_\Gamma$ on interface actions as follows:

$$\begin{aligned}
\lfloor\!\lfloor(t, \mathsf{call}\ m(\sigma))\rfloor\!\rfloor_\Gamma &= (t, \mathsf{call}\ m(\sigma \setminus (\sigma \setminus p_t^m)));\\
\lfloor\!\lfloor(t, \mathsf{ret}\ m(\sigma))\rfloor\!\rfloor_\Gamma &= (t, \mathsf{ret}\ m(\sigma \setminus (\sigma \setminus q_t^m)));\\
\lceil\!\lceil(t, \mathsf{call}\ m(\sigma))\rceil\!\rceil_\Gamma &= (t, \mathsf{call}\ m(\sigma \setminus p_t^m));\\
\lceil\!\lceil(t, \mathsf{ret}\ m(\sigma))\rceil\!\rceil_\Gamma &= (t, \mathsf{ret}\ m(\sigma \setminus q_t^m));
\end{aligned}$$

otherwise, the result is undefined. Thus, $\lfloor\!\lfloor\psi\rfloor\!\rfloor_\Gamma$ selects the part of the state in ψ that is required by Γ and $\lceil\!\lceil\psi\rceil\!\rceil_\Gamma$ the extra piece of state not required by it. We then lift $\lfloor\!\lfloor\cdot\rfloor\!\rfloor_\Gamma$ and $\lceil\!\lceil\cdot\rceil\!\rceil_\Gamma$ to traces by applying them to every interface action.

Given a history H_0 produced by a library $\mathcal{L} : \Gamma'$, we need to be able to check that the library does not modify the extra pieces of state not required by the original method specification Γ, which are given by $\lceil\!\lceil H_0\rceil\!\rceil_\Gamma$. To this end, we define an evaluation function similar to $\langle\cdot\rangle^\sharp$ from Section 3, which is meant to be applied to $\lceil\!\lceil H_0\rceil\!\rceil_\Gamma$. For an interface action ψ we define $\langle\psi\rangle : \Sigma \to (\Sigma \cup \{\top\})$ as follows:

$$\begin{aligned}
\langle(t, \mathsf{call}\ m(\sigma_0))\rangle\sigma &= \text{if } (\sigma * \sigma_0)\downarrow \text{ then } \sigma * \sigma_0 \text{ else } \top;\\
\langle(t, \mathsf{ret}\ m(\sigma_0))\rangle\sigma &= \text{if } (\sigma \setminus \sigma_0)\downarrow \text{ then } \sigma \setminus \sigma_0 \text{ else } \top.
\end{aligned}$$

We then define the evaluation $\langle H\rangle : \Sigma \to (\Sigma \cup \{\top\})$ of a history H as follows:

$$\langle\varepsilon\rangle\sigma \;=\; \sigma; \qquad \langle H\psi\rangle\sigma \;=\; \text{if } (\langle H\rangle\sigma \neq \top) \text{ then } \langle\psi\rangle(\langle H\rangle\sigma) \text{ else } \top.$$

Thus, if the evaluation does not fail, then the history respects the notion of ownership and the pieces of state transferred to the library are returned to the client unmodified.

Theorem 8.2 (Frame rule)**.** *Assume*
(1) Γ' *extends* Γ*;*
(2) *for all* $i \in \{1, 2\}$*,* $\mathcal{L}_i : \Gamma$ *and* $\mathcal{L}_i : \Gamma'$ *are safe for* I_i *and* $I_i * I$*, respectively;*
(3) $(\mathcal{L}_1 : \Gamma, I_1) \sqsubseteq (\mathcal{L}_2 : \Gamma, I_2)$*; and*
(4) *for every* $\sigma_0 \in I_1$*,* $\sigma_0' \in I$ *and* $\lambda \in [\![\mathcal{L}_1 : \Gamma']\!](\sigma_0 * \sigma_0')$*, we have* $\langle\lceil\!\lceil\mathsf{history}(\lambda)\rceil\!\rceil_\Gamma\rangle\sigma_0' \neq \top$*.*
Then $(\mathcal{L}_1 : \Gamma', I_1 * I) \sqsubseteq (\mathcal{L}_2 : \Gamma', I_2 * I)$*.*

The theorem allows us to establish the linearizability relation with respect to the extended specification Γ' given the relation with respect to Γ. This enables the use of the Abstraction Theorem for clients performing ownership transfer. However, the theorem does not guarantee the safety of the libraries with respect to Γ' for free, because it is not implied by their safety with respect to Γ. Intuitively, this is because Γ' extends both preconditions and postconditions in Γ; hence, not only does it guarantee to the library that the client will provide extra pieces of state at calls, but it also requires the library to provide (possibly different) extra pieces of state at returns. For example, Γ might assign the specification $\{\mathsf{arg}_t \mapsto \_\}\ m\ \{\mathsf{arg}_t \mapsto \_\}$ to every method m, and Γ', the specification $\{\mathsf{arg}_t \mapsto \_\}\ m\ \{\exists x.\, \mathsf{arg}_t \mapsto x * x \mapsto \_\}$. Unless a library already has all the memory required by the postconditions in Γ' in its initial state, it has no way of satisfying Γ'.

This situation is in contrast to the frame rule of separation logic [26], which guarantees the safety of a piece of code with respect to an extended specification. However, the frame rule requires the latter specification to extend both pre- and postconditions with the same piece of state, so that the code returns it immediately after termination. In our setting, a library can return the extra state to its client after a different method invocation and, possibly, in a different thread.

Finally, condition (4) in Theorem 8.2 ensures that the extra memory required by postconditions in Γ' comes from the extra memory provided in its preconditions and the extension of the initial state, not from the memory transferred according to Γ.

It can be shown that for the library $\mathcal{L}_1$ in Figure 2(a), the library $\mathcal{L}_2$ in Figure 7(a) and method specification Γ defined by (8.1), we have $(\mathcal{L}_1 : \Gamma, I_1) \sqsubseteq (\mathcal{L}_2 : \Gamma, I_2)$. It is also not difficult to prove (e.g., using separation logic) that $\mathcal{L}_1 : \Gamma'$ and $\mathcal{L}_2 : \Gamma'$ are safe. Condition (4) in Theorem 8.2 is satisfied, since the proof of safety of $\mathcal{L}_1 : \Gamma'$ would use only the extra state provided in the preconditions of Γ' to provide the extra state required by its postconditions. Hence, by Theorem 8.2 we have $(\mathcal{L}_1 : \Gamma', I_1) \sqsubseteq (\mathcal{L}_2 : \Gamma', I_2)$.

However, Theorem 8.2 is not applicable to the memory allocators in Figures 2(b) and 7(b): since the allocator implementation in Figure 2(b) stores free-list pointers inside the memory blocks, it is unsafe with respect to the variant of the method specification (4.2) that does not transfer their ownership.

The proof of Theorem 8.2 relies on the following two lemmas, proved in Appendices A.3 and A.4, that convert between library traces corresponding to extended and original method specifications. The first lemma shows that for a trace λ produced by $\mathcal{L} : \Gamma'$, the trace $\lfloor\!\lfloor \lambda \rfloor\!\rfloor_\Gamma$ can be produced by $\mathcal{L} : \Gamma$. The safety of the library with respect to Γ and condition (4) from Theorem 8.2 guarantee that the smaller preconditions specified by Γ are enough for the library to execute safely and that the extra pieces of state in Γ' do not influence its execution.

Lemma 8.3. *If $\lambda \in [\![\mathcal{L} : \Gamma']\!](\sigma_0 * \sigma_0')$, $\mathcal{L} : \Gamma$ is safe at σ_0, and $\langle \lceil\!\lceil \mathsf{history}(\lambda) \rceil\!\rceil_\Gamma \rangle \sigma_0' \neq \top$, then $\lfloor\!\lfloor \lambda \rfloor\!\rfloor_\Gamma \in [\![\mathcal{L} : \Gamma]\!]\sigma_0$.*

The other lemma gives conditions under which we can conclude that a trace λ is produced by $\mathcal{L} : \Gamma'$ given that $\lfloor\!\lfloor \lambda \rfloor\!\rfloor_\Gamma$ is produced by $\mathcal{L} : \Gamma$.

Lemma 8.4. *Assume $\lfloor\!\lfloor \lambda \rfloor\!\rfloor_\Gamma \in [\![\mathcal{L} : \Gamma]\!]\sigma_0$, $(\sigma_0 * \sigma_0')\downarrow$, $\mathsf{history}(\lambda)$ is balanced from $\delta(\sigma_0'' * \sigma_0')$ for $\delta(\sigma_0) \preceq \delta(\sigma_0'')$, and $\langle \lceil\!\lceil \mathsf{history}(\lambda) \rceil\!\rceil_\Gamma \rangle \sigma_0' \neq \top$. Then $\lambda \in [\![\mathcal{L} : \Gamma']\!](\sigma_0 * \sigma_0')$.*

Proof of Theorem 8.2. Consider a trace $\lambda_1 \in [\![\mathcal{L}_1 : \Gamma']\!](\sigma_1 * \sigma)$, where $\sigma_1 \in I_1$ and $\sigma \in I$. Then by (4) we have $\langle \lceil\!\lceil \mathsf{history}(\lambda_1) \rceil\!\rceil_\Gamma \rangle \sigma \neq \top$, and hence by Lemma 8.3 we have $\lfloor\!\lfloor \lambda_1 \rfloor\!\rfloor_\Gamma \in [\![\mathcal{L}_1 : \Gamma]\!]\sigma_1$. Since $(\mathcal{L}_1 : \Gamma, I_1) \sqsubseteq (\mathcal{L}_2 : \Gamma, I_2)$, for some $\sigma_2 \in I_2$ and $\lambda_2 \in [\![\mathcal{L}_2 : \Gamma]\!]\sigma_2$ we have

$$(\delta(\sigma_1), \mathsf{history}(\lfloor\!\lfloor \lambda_1 \rfloor\!\rfloor_\Gamma)) \sqsubseteq (\delta(\sigma_2), \mathsf{history}(\lambda_2)).$$

By Lemma 6.5, there exists $\lambda_2' \in [\![\mathcal{L}_2 : \Gamma]\!]\sigma_2$ such that $\mathsf{history}(\lambda_2') = \mathsf{history}(\lfloor\!\lfloor \lambda_1 \rfloor\!\rfloor_\Gamma)$. Let λ_2'' be the trace λ_2' with its interface actions replaced so that they form the history $\mathsf{history}(\lambda_1)$. Then $\lfloor\!\lfloor \lambda_2'' \rfloor\!\rfloor_\Gamma = \lambda_2'$ and $\lceil\!\lceil \mathsf{history}(\lambda_2'') \rceil\!\rceil_\Gamma = \lceil\!\lceil \mathsf{history}(\lambda_1) \rceil\!\rceil_\Gamma$. Since $\delta(\sigma_2) \preceq \delta(\sigma_1)$, we have $(\sigma_2 * \sigma)\downarrow$. Hence, by Lemma 8.4, $\lambda_2'' \in [\![\mathcal{L}_2 : \Gamma']\!](\sigma_2 * \sigma)$, from which the required follows. □

9. Related Work

The original definition of linearizability [18] was proposed in an abstract setting that did not consider a particular programming language and implicitly assumed a complete isolation between the states of the client and the library. Furthermore, at the time it was not clear what the linearizability of a library entails for its clients. Filipović et al. [12] were the first to observe that linearizability implies a form of contextual refinement; technically, their result is similar to our Lemma 6.7, but formulated over a highly idealistic semantics. In a

previous work [14], we generalised their result to a compositional proof method, formalised by an Abstraction Theorem, that allows one to replace a concrete library by an abstract one in reasoning about a complete program.

This paper is part of our recent push to propose notions of concurrent library correctness for realistic programming languages. So far we have developed such notions together with the corresponding Abstraction Theorems for supporting reasoning about liveness properties [14] and weak memory models [4, 6, 15]. All these results assumed that the library and its client operate in disjoint address spaces and, hence, are guaranteed not to interfere with each other and cannot communicate via the heap. Lifting this restriction is the goal of the present paper. Although the basic proof structure of Theorems 6.2 and 6.4 is the same as in [6,14], the formulations and proofs of the Abstraction Theorem and the required lemmas here have to deal with technical challenges posed by ownership transfer that did not arise in previous work. First, their formulations rely on the novel forms of client-local and library-local semantics (Section 5) that allow a component to communicate with its environment via ownership transfers. Proving Lemma 5.6 then involves a delicate tracking of a splitting between the parts of the state owned by the library and the client, and how ownership transfers affect it. Second, the key result needed to establish the Abstraction Theorem is the Rearrangement Lemma (Lemmas 6.5 and 6.7). What makes the proof of this lemma difficult in our case is the need to deal with subtle interactions between concurrency and ownership transfer that have not been considered in previous work. Namely, changing the history in the lemma requires commuting ownership transfer actions; justifying the correctness of these transformations is non-trivial and relies on the notion of history balancedness that we propose. These differences notwithstanding, we hope that techniques for handling ownership transfer proposed in this paper can be combined with the ones for handling other types of client-library interactions considered so far [4, 6, 14, 15].

Recently, there has been a lot of work on verifying linearizability of common algorithms; representative papers include [1, 10, 28]. All of them proved classical linearizability, where libraries and their clients exchange values of a given data type and do not perform ownership transfers. This includes even libraries, such as concurrent containers, that are actually used by client threads to transfer the ownership of data structures. The frame rule for linearizability we propose (Theorem 8.2) justifies that classical linearizability established for concurrent containers entails linearizability with ownership transfer. This makes our Abstraction Theorem applicable, enabling compositional reasoning about their clients.

Turon and Wand [27] have proposed a logic for establishing refinements between concurrent modules, likely equivalent to linearizability [12]. Their logic considers libraries and clients residing in a shared address space, but not ownership transfer. As a result, they do not support separate reasoning about a library and its client in realistic situations of the kind we consider.

Elmas et al. [10,11] have developed a system for verifying concurrent programs based on repeated applications of atomicity abstraction. They do not use linearizability to perform the abstraction. Instead, they check the commutativity of an action to be incorporated into an atomic block with *all* actions of other threads. In particular, to abstract a library implementation in a program by its atomic specification, their method would have to check the commutativity of every internal action of the library with all actions executed by the client code of other threads. Thus, the method of Elmas et al. does not allow decomposing the verification of a program into verifying libraries and their clients separately. In contrast, our Abstraction Theorem ensures the atomicity of a library under *any* safe client.

The most common approach of decomposing the verification of concurrent programs is using *thread-modular* reasoning methods, which consider every thread in the program in isolation under some assumptions on its environment [20, 24]. However, a single thread would usually make use of multiple program components. This work goes further by allowing a finer-grain *intrathread-modular* reasoning: separating the verification of a library and its client, the code from both of which may be executed by a single thread. Note that this approach is complementary to thread-modular reasoning, which can still be used to carry out the verification subtasks, such as establishing the linearizability of libraries and proving the safety of clients. Thread-modular techniques do enable a restricted form of intrathread-modular reasoning, since they allow reasoning about the control of a thread in a program while ignoring the possibility of its interruption by the other threads. Hence, they allow considering a library method called by the thread in isolation, e.g., by using the standard proof rules for procedures. However, such a decomposition is done under fixed assumptions on the environment of the thread and thus does not allow, e.g., increasing the atomicity of the environment's actions. As the example of MCAS shows (Section 6.2), this is necessary to deal with complex algorithms.

Ways of establishing relationships between different sequential implementations of the same library have been studied in *data refinement* [19, 25], including cases of interactions via ownership transfer [3, 13, 22]. Our results can be viewed as generalising data refinement to the concurrent setting. Moreover, when specialised to the sequential case, they provide a more flexible method of performing it in the presence of the heap and ownership transfer than previously proposed ones. In more detail, the way we define client safety (Section 5) is more general some of the ways used in data refinement [13]. There, it is typical to fix a (precise) invariant of a library and check that the client does not access the area of memory fenced off by the invariant. Here we do not require an explicit library invariant, using the client-local semantics instead: since primitive commands fault when accessing non-existent memory cells, the safety of the client in this semantics ensures that it does not access the internals of the library. We note that the approach requiring an invariant for library-local data structures does not generalise to the concurrent setting: while a precise invariant for the data structures *shared* among threads executing library code is not usually difficult to find, the state of data structures *local* to the threads depends on their program counters. Thus, an invariant insensitive to program positions inside the library code often does not exist. Such difficulties are one of reasons for using client- and library-local semantics in this paper.

Finally, we note that the applicability of our results is not limited to proving existing programs correct: they can also be used in the context of formal program development. In this case, instead of *abstracting* an existing library to an atomic specification while proving a complete program, the Abstraction Theorem allows *refining* an atomic library specification to a concrete concurrent implementation while developing a program top-down [2, 21]. Our work thus advances the method of atomicity refinement to a setting with concurrent components sharing an address space and communicating via ownership transfers.

Acknowledgements

We would like to thank Anindya Banerjee, Josh Berdine, Xinyu Feng, Hongjin Liang, Victor Luchangco, David Naumann, Peter O'Hearn, Matthew Parkinson, Noam Rinetzky and

Julles Villard for helpful comments. Gotsman was supported by the EU FET project ADVENT. Yang was supported by EPSRC.

References

[1] D. Amit, N. Rinetzky, T. W. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *CAV'07: Conference on Computer Aided Verification*, volume 4590 of *LNCS*, pages 477–490. Springer, 2007.

[2] R.-J. Back. On correct refinement of programs. *J. Comput. Syst. Sci.*, 23(1):49–68, 1981.

[3] A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence in object-oriented programs. *J. ACM*, 52(6):894–960, 2005.

[4] Mark Batty, Mike Dodds, and Alexey Gotsman. Library abstraction for C/C++ concurrency. In *POPL'13: Symposium on Principles of Programming Languages*, pages 235–248. ACM Press, 2013.

[5] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL'05: Symposium on Principles of Programming Languages*, pages 259–270. ACM Press, 2005.

[6] S. Burckhardt, A. Gotsman, M. Musuvathi, and H. Yang. Concurrent library correctness on the TSO memory model. In *ESOP'12: European Symposium on Programming*, volume 7211 of *LNCS*, pages 87–107. Springer, 2012.

[7] C. Calcagno, P. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS'07: Symposium on Logic in Computer Science*, pages 366–378. IEEE, 2007.

[8] D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In *ECOOP'01: European Conference on Object-Oriented Programming*, volume 2072 of *LNCS*, pages 53–76. Springer, 2001.

[9] M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *ESOP'09: European Symposium on Programming*, volume 5502 of *LNCS*, pages 363–377. Springer, 2009.

[10] T. Elmas, S. Qadeer, A. Sezgin, O. Subasi, and S. Tasiran. Simplifying linearizability proofs with reduction and abstraction. In *TACAS'10: Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *LNCS*, pages 296–311. Springer, 2010.

[11] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *POPL'09: Symposium on Principles of Programming Languages*, pages 2–15. ACM Press, 2009.

[12] I. Filipović, P. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010.

[13] I. Filipović, P. O'Hearn, N. Torp-Smith, and H. Yang. Blaiming the client: On data refinement in the presence of pointers. *Form. Asp. Comput.*, 22(5):547–583, 2010.

[14] A. Gotsman and H. Yang. Liveness-preserving atomicity abstraction. In *ICALP'11: International Colloquium on Automata, Languages and Programming*, volume 6756 of *LNCS*, pages 453–465. Springer, 2011.

[15] Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. Show no weakness: sequentially consistent specifications of TSO libraries. In *DISC'12: Symposium on Distributed Computing*, volume 7611 of *LNCS*, pages 31–45. Springer, 2012.

[16] T. Harris, K. Fraser, and I. Pratt. A practical multi-word compare-and-swap operation. In *DISC'02: Symposium on Distributed Computing*, volume 2508 of *LNCS*, pages 265–279. Springer, 2002.

[17] M. Herlihy and N. Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2008.

[18] M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[19] C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.

[20] C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[21] C. B. Jones. Splitting atoms safely. *Theor. Comput. Sci.*, 375(1-3):109–119, 2007.

[22] I. Mijajlovic and H. Yang. Data refinement with low-level pointer operations. In *APLAS'05: Asian Symposium on Programming Languages and Systems*, volume 3780 of *LNCS*, pages 19–36. Springer, 2005.

[23] P. O'Hearn. Resources, concurrency and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[24] A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems*, pages 123–144. Springer, 1985.

[25] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.
[26] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02: Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.
[27] A. Turon and M. Wand. A separation logic for refining concurrent objects. In *POPL'11: Symposium on Principles of Programming Languages*, pages 247–258. ACM Press, 2011.
[28] V. Vafeiadis. Modular fine-grained concurrency verification. PhD Thesis. University of Cambridge, 2008.
[29] V. Vafeiadis. Automatically proving linearizability. In *CAV'10: Conference on Computer Aided Verification*, volume 6174 of *LNCS*, pages 450–464. Springer, 2010.

Appendix A. Additional proofs

A.1. **Proof of Proposition 7.7.** We prove the required by induction on the length of S. If S is empty, then so is S' and $l_c = \delta(e)$. Assume the statement of the proposition is valid for all histories S of length less than $n > 0$. Consider a history $S = S_0\psi$ of length n and a corresponding history S' satisfying the conditions of the proposition. We now make a case split on the type of the action ψ.

- ψ is a call transferring σ_0 that is not in S'. Then S_0 and S' are identical except S_0 may have extra calls. Hence, by the induction hypothesis for S_0 and S', S' is balanced from l_1, $[\![S']\!]^\sharp l_2 \preceq [\![S']\!]^\sharp l_1$ and
$$[\![S]\!]^\sharp l_1 = [\![S_0\psi]\!]^\sharp l_1 = ([\![S_0]\!]^\sharp l_1) \circ \delta(\sigma_0) = ([\![S']\!]^\sharp l_1) \circ (l_c \circ \delta(\sigma_0)).$$
- ψ is a call transferring σ_0 also present in S'. Then $S' = S'_0\psi$, where S'_0 and S_0 are identical except S_0 may have extra calls. Hence, by the induction hypothesis for S_0 and S'_0, we have
$$[\![S]\!]^\sharp l_1 = [\![S_0\psi]\!]^\sharp l_1 = ([\![S_0]\!]^\sharp l_1) \circ \delta(\sigma_0) = ([\![S'_0]\!]^\sharp l_1) \circ l_c \circ \delta(\sigma_0) = ([\![S'_0\psi]\!]^\sharp l_1) \circ l_c = ([\![S']\!]^\sharp l_1) \circ l_c.$$
In particular, S' is balanced from l_1. By the induction hypothesis for S_0 and S'_0, we also have $[\![S'_0]\!]^\sharp l_2 \preceq [\![S'_0]\!]^\sharp l_1$. From this we get
$$[\![S']\!]^\sharp l_2 = [\![S'_0\psi]\!]^\sharp l_2 = ([\![S'_0]\!]^\sharp l_2) \circ \delta(\sigma_0) \preceq ([\![S'_0]\!]^\sharp l_1) \circ \delta(\sigma_0) = [\![S'_0\psi]\!]^\sharp l_1 = [\![S']\!]^\sharp l_1.$$
- ψ is a return transferring σ_0. Then it is also present in S', so that $S' = S'_0\psi$, where S_0 and S'_0 are identical except S_0 may have extra calls. Then by the induction hypothesis for S_0 and S'_0, we have:
$$[\![S]\!]^\sharp l_1 = [\![S_0\psi]\!]^\sharp l_1 = ([\![S_0]\!]^\sharp l_1) \setminus\!\setminus \delta(\sigma_0) = (([\![S'_0]\!]^\sharp l_1) \circ l_c) \setminus\!\setminus \delta(\sigma_0).$$
Since $S' = S'_0\psi$ is balanced from l_2, $([\![S'_0]\!]^\sharp l_2) \setminus\!\setminus \delta(\sigma_0)$ is defined. Furthermore, by the induction hypothesis for S_0 and S'_0, we also have $[\![S'_0]\!]^\sharp l_2 \preceq [\![S'_0]\!]^\sharp l_1$. Hence, $([\![S'_0]\!]^\sharp l_1) \setminus\!\setminus \delta(\sigma_0)$ is defined as well. By Proposition 2.7, we then have:
$$[\![S]\!]^\sharp l_1 = (([\![S'_0]\!]^\sharp l_1) \circ l_c) \setminus\!\setminus \delta(\sigma_0) = (([\![S'_0]\!]^\sharp l_1) \setminus\!\setminus \delta(\sigma_0)) \circ l_c = ([\![S'_0\psi]\!]^\sharp l_1) \circ l_c = ([\![S']\!]^\sharp l_1) \circ l_c.$$
In particular, S' is balanced from l_1. From $[\![S'_0]\!]^\sharp l_2 \preceq [\![S'_0]\!]^\sharp l_1$, it also follows that
$$[\![S']\!]^\sharp l_2 = [\![S'_0\psi]\!]^\sharp l_2 = ([\![S'_0]\!]^\sharp l_2) \setminus\!\setminus \delta(\sigma_0) \preceq ([\![S'_0]\!]^\sharp l_1) \setminus\!\setminus \delta(\sigma_0) = [\![S'_0\psi]\!]^\sharp l_1 = [\![S']\!]^\sharp l_1. \qquad \square$$

A.2. **Proof of Lemma 5.6.** Before delving into the proof of Lemma 5.6, we prove three important lemmas about our semantics that justify its key steps. The first concerns the evaluation of a call or a return action: intuitively, it says that the evaluation of such an action by the client matches that by the library.

Lemma A.1 (Preservation)**.** *Let Γ be a method specification, σ_0, σ_1 states, and φ an action describing a call to or return from a method specified in Γ such that*

$$[\![\Gamma \vdash \varphi]\!]\sigma_0 \neq \top \ \wedge \ [\![\varphi : \Gamma]\!]\sigma_1 \neq \top.$$

Then for all $\sigma_0', \sigma_1', \varphi'$,

$$((\sigma_0', \varphi') \in [\![\Gamma \vdash \varphi]\!]\sigma_0 \ \wedge \ (\sigma_1', \varphi') \in [\![\varphi : \Gamma]\!]\sigma_1) \implies ((\sigma_0' * \sigma_1')\downarrow \iff (\sigma_0 * \sigma_1)\downarrow).$$

If furthermore $\sigma_0 * \sigma_1$ is defined, then we have

$$\{\sigma_0 * \sigma_1\} = \{\sigma_0' * \sigma_1' \mid \exists \varphi'.\ (\sigma_0', \varphi') \in [\![\Gamma \vdash \varphi]\!]\sigma_0 \ \wedge \ (\sigma_1', \varphi') \in [\![\varphi : \Gamma]\!]\sigma_1 \ \wedge \ (\sigma_0' * \sigma_1')\downarrow\}.$$

Proof. Consider $\Gamma, \sigma_0, \sigma_1, \varphi$ satisfying the conditions in the lemma. We show the lemma only for the case when φ is a call action: for some t, m, p, we have $\varphi = (t, \mathsf{call}\ m)$ and $\{p\}\ m\ \{\_\} \in \Gamma$. The proof for the other case is symmetric.

To show the first claim of the lemma, consider $\sigma_0', \sigma_1', \varphi'$ such that

$$(\sigma_0', \varphi') \in [\![\Gamma \vdash \varphi]\!]\sigma_0 \ \wedge \ (\sigma_1', \varphi') \in [\![\varphi : \Gamma]\!]\sigma_1.$$

By the definition of the action evaluation, there exist σ_2, σ_3 such that

$$\varphi' = (t, \mathsf{call}\ m(\sigma_3)) \ \wedge \ (\sigma_2 * \sigma_3)\downarrow \wedge (\sigma_3 * \sigma_1)\downarrow \wedge \sigma_0 = \sigma_2 * \sigma_3 \ \wedge \ \sigma_0' = \sigma_2 \ \wedge \ \sigma_1' = \sigma_3 * \sigma_1.$$

Hence,

$$(\sigma_0' * \sigma_1')\downarrow \iff (\sigma_2 * (\sigma_3 * \sigma_1))\downarrow \iff ((\sigma_2 * \sigma_3) * \sigma_1)\downarrow \iff (\sigma_0 * \sigma_1)\downarrow.$$

Let us move on to the second claim of the lemma. Since $[\![\Gamma \vdash (t, \mathsf{call}\ m)]\!]\sigma_0 \neq \top$, p_t is precise and the $*$ operator is cancellative, there exists a unique splitting $\sigma_2 * \sigma_3 = \sigma_0$ of σ_0 such that $\sigma_3 \in p_t$. Let $\varphi_0 = (t, \mathsf{call}\ m(\sigma_3))$. Then

$$(\{(\sigma_2, \varphi_0)\} = [\![\Gamma \vdash (t, \mathsf{call}\ m)]\!]\sigma_0) \ \wedge \ (\forall \sigma_1'.\ (\sigma_1', \varphi_0) \in [\![(t, \mathsf{call}\ m) : \Gamma]\!]\sigma_1 \iff \sigma_1' = \sigma_3 * \sigma_1).$$

Hence,

$$\begin{aligned}&\{\sigma_0' * \sigma_1' \mid \exists \varphi'.\ (\sigma_0', \varphi') \in [\![\Gamma \vdash (t, \mathsf{call}\ m)]\!]\sigma_0 \wedge (\sigma_1', \varphi') \in [\![(t, \mathsf{call}\ m) : \Gamma]\!]\sigma_1 \wedge (\sigma_0' * \sigma_1')\downarrow\} \\ &= \{\sigma_2 * (\sigma_3 * \sigma_1)\} = \{\sigma_0 * \sigma_1\}.\end{aligned}$$

□

The second lemma describes the decomposition and composition properties of trace evaluation.

Lemma A.2 (Trace Decomposition and Composition)**.** *Consider traces τ, κ, λ without interface actions such that $\mathsf{cover}(\tau, \kappa, \lambda)$. For all states σ_0, σ_1, if*

$$(\sigma_0 * \sigma_1)\downarrow \ \wedge \ [\![\Gamma \vdash \kappa]\!]\sigma_0 \neq \top \ \wedge \ [\![\lambda : \Gamma]\!]\sigma_1 \neq \top, \tag{A.1}$$

then

$$[\![\tau]\!](\sigma_0 * \sigma_1) = \{(\sigma', \tau) \in [\![\Gamma \vdash \kappa]\!]\sigma_0 \otimes [\![\lambda : \Gamma]\!]\sigma_1\} \tag{A.2}$$

and

$$\forall \sigma_0', \sigma_1', \kappa', \lambda'.\, (\mathsf{cover}(\tau, \kappa', \lambda') \ \wedge \ (\sigma_0', \kappa') \in [\![\Gamma \vdash \kappa]\!]\sigma_0 \ \wedge \ (\sigma_1', \lambda') \in [\![\lambda : \Gamma]\!]\sigma_1) \implies (\sigma_0' * \sigma_1')\downarrow. \tag{A.3}$$

Proof. Consider $\tau, \kappa, \lambda, \sigma_0, \sigma_1, \Gamma$ satisfying the assumptions. We prove the lemma by induction on the length of τ. The base case of τ being the empty sequence is trivial.

Now suppose that $\tau = \tau'\varphi$ for some τ', φ. Then there exist κ' and λ' such that

$$\mathsf{cover}(\tau', \kappa', \lambda') \ \wedge \ ((\kappa = \kappa'\varphi \ \wedge \ \lambda = \lambda') \ \vee \ (\kappa = \kappa' \ \wedge \ \lambda = \lambda'\varphi) \ \vee \ (\kappa = \kappa'\varphi \ \wedge \ \lambda = \lambda'\varphi)).$$

By the assumption of the lemma, we have that

$$[\![\Gamma \vdash \kappa']\!]\sigma_0 \neq \top \ \wedge \ [\![\lambda' : \Gamma]\!]\sigma_1 \neq \top. \tag{A.4}$$

Hence, by the induction hypothesis, we have that

$$[\![\tau']\!](\sigma_0 * \sigma_1) = \{(\sigma', \tau') \in [\![\Gamma \vdash \kappa']\!]\sigma_0 \otimes [\![\lambda' : \Gamma]\!]\sigma_1\} \tag{A.5}$$

and

$$\forall \sigma_0', \sigma_1', \kappa'', \lambda''. \, (\mathsf{cover}(\tau', \kappa'', \lambda'') \ \wedge \ (\sigma_0', \kappa'') \in [\![\Gamma \vdash \kappa']\!]\sigma_0 \ \wedge \ (\sigma_1', \lambda'') \in [\![\lambda' : \Gamma]\!]\sigma_1) \\ \implies (\sigma_0' * \sigma_1')\downarrow. \tag{A.6}$$

From (A.5) it follows that:

$$[\![\tau']\!](\sigma_0 * \sigma_1) \neq \top. \tag{A.7}$$

Next, we prove that

$$[\![\tau'\varphi]\!](\sigma_0 * \sigma_1) \neq \top. \tag{A.8}$$

For the sake of contradiction, suppose this disequality does not hold. Because of (A.7), there exists σ'' such that

$$(\sigma'', \_) \in [\![\tau']\!](\sigma_0 * \sigma_1) \ \wedge \ [\![\varphi]\!]\sigma'' = \top. \tag{A.9}$$

By (A.5), this implies the existence of σ_0'', σ_1'' such that

$$(\sigma_0'', \_) \in [\![\Gamma \vdash \kappa']\!]\sigma_0 \ \wedge \ (\sigma_1'', \_) \in [\![\lambda' : \Gamma]\!]\sigma_1 \ \wedge \ \sigma'' = \sigma_0'' * \sigma_1''.$$

We split cases based on the relationships among κ, κ', λ and λ'.

(1) If $\kappa = \kappa'\varphi$ and $\lambda = \lambda'$, then $\varphi = (t, c)$ for some t, c. By (A.1), $[\![\Gamma \vdash \varphi]\!]\sigma_0'' \neq \top$, so that $f_c^t(\sigma_0'') \neq \top$. Hence, by the Strong Locality of f_c^t, $f_c^t(\sigma_0'' * \sigma_1'') \neq \top$, so that $[\![\varphi]\!](\sigma'') \neq \top$. But this contradicts (A.9).
(2) If $\kappa = \kappa'$ and $\lambda = \lambda'\varphi$, then $\varphi = (t, c)$ for some t, c. This case is symmetric to the previous one.
(3) If $\kappa = \kappa'\varphi$ and $\lambda = \lambda'\varphi$, then φ is a call or a return action. Then, by the definition of evaluation, $[\![\varphi]\!]\sigma'' = \{(\sigma'', \varphi)\} \neq \top$. This gives the desired contradiction.

The remainder of the proof is again done by a case analysis on the relationships among κ, κ', λ and λ'. We consider three cases.

1. $\kappa = \kappa'\varphi$ and $\lambda = \lambda'$. In this case, $\varphi = (t, c)$ for some t, c. As shown in (A.8), $[\![\tau'\varphi]\!](\sigma_0 * \sigma_1) \neq \top$. Pick σ'' such that

$$(\sigma'', \tau'\varphi) \in [\![\tau'\varphi]\!](\sigma_0 * \sigma_1). \tag{A.10}$$

By the definition of the trace evaluation and the induction hypothesis in (A.5), there exist σ_0'', σ_1'', κ'' and λ'' such that

$$(\sigma'', \varphi) \in [\![\varphi]\!](\sigma_0'' * \sigma_1'') \ \wedge \ (\sigma_0'', \kappa'') \in [\![\Gamma \vdash \kappa']\!]\sigma_0 \ \wedge \ (\sigma_1'', \lambda'') \in [\![\lambda' : \Gamma]\!]\sigma_1 \ \wedge \ \mathsf{cover}(\tau', \kappa'', \lambda''). \tag{A.11}$$

Then

$$\mathsf{cover}(\tau' \, (t, c), \kappa'' \, (t, c), \lambda'').$$

We have $[\![\Gamma \vdash (t,c)]\!]\sigma_0'' \neq \top$, because $\kappa = \kappa'(t,c)$ and $[\![\Gamma \vdash \kappa]\!]\sigma_0 \neq \top$. Hence $f_c^t(\sigma_0'') \neq \top$ and, furthermore, $\sigma'' \in f_c^t(\sigma_0'' * \sigma_1'')$. Hence, by the Strong Locality of f_c^t, there exists σ_0''' such that

$$\sigma_0''' \in f_c^t(\sigma_0'') \ \wedge \ \sigma'' = \sigma_0''' * \sigma_1''.$$

This implies

$$(\sigma_0''', \kappa''(t,c)) \in [\![\Gamma \vdash \kappa'(t,c)]\!]\sigma_0 = [\![\Gamma \vdash \kappa]\!]\sigma_0.$$

From what we have shown so far, it follows that

$$(\sigma'', \tau'(t,c)) = (\sigma_0''' * \sigma_1'', \tau'(t,c)) \in [\![\Gamma \vdash \kappa]\!]\sigma_0 \otimes [\![\lambda : \Gamma]\!]\sigma_1.$$

Thus,

$$[\![\tau]\!](\sigma_0 * \sigma_1) \subseteq \{(\sigma'', \tau) \in [\![\Gamma \vdash \kappa]\!]\sigma_0 \otimes [\![\lambda : \Gamma]\!]\sigma_1\}. \tag{A.12}$$

To show the other inclusion and the (A.3) part of the lemma, consider $\sigma_0'', \sigma_1'', \kappa'', \lambda''$ such that

$$(\sigma_0'', \kappa'') \in [\![\Gamma \vdash \kappa'(t,c)]\!]\sigma_0 \ \wedge \ (\sigma_1'', \lambda'') \in [\![\lambda' : \Gamma]\!]\sigma_1 \ \wedge \ \mathsf{cover}(\tau'(t,c), \kappa'', \lambda'').$$

Then for some κ''' we have

$$\kappa'' = \kappa'''(t,c) \ \wedge \ \mathsf{cover}(\tau', \kappa''', \lambda'').$$

By the definition of the evaluation function, there exists σ_0''' such that

$$(\sigma_0''', \kappa''') \in [\![\Gamma \vdash \kappa']\!]\sigma_0 \ \wedge \ (\sigma_0'', (t,c)) \in [\![\Gamma \vdash (t,c)]\!]\sigma_0'''.$$

By the induction hypothesis in (A.5) and (A.6),

$$(\sigma_0''' * \sigma_1'')\downarrow \wedge (\sigma_0''' * \sigma_1'', \tau') \in [\![\tau']\!](\sigma_0 * \sigma_1).$$

Now by the Footprint Preservation property of f_c^t, the first conjunct above implies that $(\sigma_0'' * \sigma_1'')\downarrow$, which proves (A.3). By the Strong Locality of f_c^t,

$$(\sigma_0'' * \sigma_1'', (t,c)) \in [\![(t,c)]\!](\sigma_0''' * \sigma_1'').$$

From what we have shown above it follows that

$$(\sigma_0'' * \sigma_1'', \tau'(t,c)) \in [\![\tau]\!](\sigma_0 * \sigma_1).$$

Hence,

$$[\![\tau]\!](\sigma_0 * \sigma_1) \supseteq \{(\sigma'', \tau) \in [\![\Gamma \vdash \kappa]\!]\sigma_0 \otimes [\![\lambda : \Gamma]\!]\sigma_1\}. \tag{A.13}$$

2. $\kappa = \kappa'$ and $\lambda = \lambda'\varphi$. This case is symmetric to the previous one.

3. $\kappa = \kappa'\varphi$ and $\lambda = \lambda'\varphi$. In this case φ is a call to or a return from a method in Γ. As shown in (A.8), $[\![\tau'\varphi]\!](\sigma_0 * \sigma_1) \neq \top$. Pick σ'' such that (A.10) holds. By the definition of evaluation and the induction hypothesis in (A.5), there exist σ_0'', σ_1'', κ'' and λ'' such that (A.11) holds. But

$$[\![\Gamma \vdash \varphi]\!]\sigma_0'' \neq \top \ \wedge \ [\![\varphi : \Gamma]\!]\sigma_1'' \neq \top.$$

Furthermore, $\sigma'' = \sigma_0'' * \sigma_1''$. Hence, by Lemma A.1 and the definition of the evaluation, there exist $\sigma_0''', \sigma_1''', \varphi'$ such that

$$\sigma'' = \sigma_0''' * \sigma_1''' \ \wedge \ (\sigma_0''', \varphi') \in [\![\Gamma \vdash \varphi]\!]\sigma_0'' \ \wedge \ (\sigma_1''', \varphi') \in [\![\varphi : \Gamma]\!]\sigma_1''$$

This in turn implies that

$$(\sigma_0''', \kappa''\varphi') \in [\![\Gamma \vdash \kappa'\varphi]\!]\sigma_0 \ \wedge \ (\sigma_1''', \lambda''\varphi') \in [\![\lambda'\varphi : \Gamma]\!]\sigma_1.$$

From what we have shown so far, it follows that

$$(\sigma'', \tau'\varphi) = (\sigma_0''' * \sigma_1''', \tau'\varphi) \in [\![\Gamma \vdash \kappa]\!]\sigma_0 \otimes [\![\lambda : \Gamma]\!]\sigma_1.$$

Thus, (A.12) holds.

To show the other inclusion and the (A.3) part of the lemma, consider

$$\sigma_0'', \sigma_1'', \sigma_0''', \sigma_1''', \kappa'', \lambda'', \varphi'$$

such that

$$(\sigma_0'', \kappa'') \in [\![\Gamma \vdash \kappa']\!]\sigma_0 \ \wedge\ (\sigma_1'', \lambda'') \in [\![\lambda' : \Gamma]\!]\sigma_1 \ \wedge\ (\sigma_0''', \varphi') \in [\![\Gamma \vdash \varphi]\!]\sigma_0''$$
$$\wedge\ (\sigma_1''', \varphi') \in [\![\varphi : \Gamma]\!]\sigma_1'' \ \wedge\ \mathsf{cover}(\tau'\varphi, \kappa''\varphi', \lambda''\varphi').$$

We need to show that

$$(\sigma_0''' * \sigma_1''')\downarrow \wedge (\sigma_0''' * \sigma_1''', \tau'\varphi) \in [\![\tau'\varphi]\!](\sigma_0 * \sigma_1).$$

In particular, this establishes (A.13).

Since $\mathsf{cover}(\tau'\varphi, \kappa''\varphi', \lambda''\varphi')$ and φ' is a call to or a return from a method in Γ,

$$\mathsf{cover}(\tau', \kappa'', \lambda'') \ \wedge\ \varphi = \mathsf{ground}(\varphi').$$

We now use the induction hypothesis in (A.5) and (A.6) and derive that

$$(\sigma_0'' * \sigma_1'')\downarrow \wedge (\sigma_0'' * \sigma_1'', \tau') \in [\![\tau']\!](\sigma_0 * \sigma_1).$$

But $[\![\Gamma \vdash \varphi]\!]\sigma_0'' \neq \top$ and $[\![\varphi : \Gamma]\!]\sigma_1'' \neq \top$. Hence, by Lemma A.1,

$$(\sigma_0''' * \sigma_1''')\downarrow \wedge (\sigma_0'' * \sigma_1'' = \sigma_0''' * \sigma_1''').$$

By the definition of evaluation,

$$[\![\varphi]\!](\sigma_0'' * \sigma_1'') = \{(\sigma_0'' * \sigma_1'', \varphi)\} = \{(\sigma_0''' * \sigma_1''', \varphi)\}.$$

From what we have shown, it follows that

$$(\sigma_0''' * \sigma_1''', \tau'\varphi) \in [\![\tau'\varphi]\!](\sigma_0 * \sigma_1),$$

as required. □

The following lemma shows that the trace-set generation of our semantics also satisfies the decomposition and composition properties.

Lemma A.3. $\forall \tau.\, \tau \in (\!|\mathcal{C}(\mathcal{L})|\!) \iff (\exists \kappa, \lambda.\, \kappa \in (\!|\Gamma \vdash \mathcal{C}|\!) \ \wedge\ \lambda \in (\!|\mathcal{L} : \Gamma|\!) \ \wedge\ \mathsf{cover}(\tau, \kappa, \lambda))$.

Proof. Let

$$\mathcal{C} = \mathsf{let}\ [-]\ \mathsf{in}\ C_1 \parallel \ldots \parallel C_n;\ \ \mathcal{L} = \{m = C_m \mid m \in \{m_1, \ldots, m_j\}\};\ \ C_{\mathsf{mgc}} = (m_1 + \ldots + m_j)^*.$$

First, consider $\tau \in (\!|\mathcal{C}(\mathcal{L})|\!)$. By the definition of the semantics, for some trace τ', τ is a prefix of τ',

$$\forall t \in \{1, \ldots, n\}.\, \tau'|_t \in (\!|C_t|\!)_t(\boldsymbol{\lambda}(m, t).\, (\!|C_m|\!)_t(\_))$$

and all actions in τ' are done by some thread $t \in \{1, \ldots, n\}$. Then

$$\forall t \in \{1, \ldots, n\}.\, \mathsf{client}(\tau'|_t) \in (\!|C_t|\!)_t(\boldsymbol{\lambda}(m, t).\, \{\varepsilon\}) \ \wedge\ \mathsf{lib}(\tau'|_t) \in (\!|C_{\mathsf{mgc}}|\!)_t(\boldsymbol{\lambda}(m, t).\, (\!|C_m|\!)_t(\_)).$$

Since all actions in τ' are done by some thread $t \in \{1, \ldots, n\}$, we have

$$\mathsf{client}(\tau') \in (\mathsf{client}(\tau'|_1) \parallel \ldots \parallel \mathsf{client}(\tau'|_n)) \ \wedge\ \mathsf{lib}(\tau') \in (\mathsf{lib}(\tau'|_1) \parallel \ldots \parallel \mathsf{lib}(\tau'|_n)).$$

Hence,

$$\mathsf{client}(\tau') \in (\!|\Gamma \vdash \mathcal{C}|\!) \ \wedge\ \mathsf{lib}(\tau') \in (\!|\mathcal{L} : \Gamma|\!).$$

Since $\mathsf{client}(\tau)$ is a prefix of $\mathsf{client}(\tau')$ and $\mathsf{lib}(\tau)$ is a prefix of $\mathsf{lib}(\tau')$, this implies

$$\mathsf{client}(\tau) \in (\!|\Gamma \vdash \mathcal{C}|\!) \ \wedge\ \mathsf{lib}(\tau) \in (\!|\mathcal{L} : \Gamma|\!).$$

Furthermore, $\mathsf{cover}(\tau, \mathsf{client}(\tau), \mathsf{lib}(\tau))$, as desired.

Assume now that

$$\kappa \in (\!|\Gamma \vdash \mathcal{C}|\!) \ \wedge\ \lambda \in (\!|\mathcal{L} : \Gamma|\!) \ \wedge\ \mathsf{cover}(\tau, \kappa, \lambda).$$

Then for some traces κ' and λ', κ is a prefix of κ', λ is a prefix of λ', and

$$\forall t \in \{1, \ldots, n\}.\, \kappa'|_t \in (\!|C_t|\!)_t(\boldsymbol{\lambda}(m, t).\, \{\varepsilon\}) \ \wedge\ \lambda'|_t \in (\!|C_{\mathsf{mgc}}|\!)_t(\boldsymbol{\lambda}(m, t).\, (\!|C_m|\!)_t(\_)).$$

The definition of our semantics in Figure 5 allows us to choose λ' in such a way that for some trace τ', τ is a prefix of τ' and $\mathsf{cover}(\tau', \kappa', \lambda')$. Then

$$\forall t \in \{1, \ldots, n\}.\, \tau'|_t \in (\!|C_t|\!)_t(\boldsymbol{\lambda}(m, t).\, (\!|C_m|\!)_t(\_))$$

and $\tau' \in (\tau'|_1 \parallel \ldots \parallel \tau'|_n)$. Thus, $\tau' \in (\!|\mathcal{C}(\mathcal{L})|\!)$, which implies $\tau \in (\!|\mathcal{C}(\mathcal{L})|\!)$, as desired. □

Proof of Lemma 5.6. We first show that $\mathcal{C}(\mathcal{L})$ is safe for $I_0 * I_1$. Pick states σ_0, σ_1, τ such that

$$\sigma_0 \in I_0 \ \wedge\ \sigma_1 \in I_1 \ \wedge\ (\sigma_0 * \sigma_1)\!\downarrow \wedge\, \tau \in (\!|\mathcal{C}(\mathcal{L})|\!).$$

By Lemma A.3, there exist traces κ, λ such that

$$\kappa \in (\!|\Gamma \vdash \mathcal{C}|\!) \ \wedge\ \lambda \in (\!|\mathcal{L} : \Gamma|\!) \ \wedge\ \mathsf{cover}(\tau, \kappa, \lambda). \tag{A.14}$$

By our assumptions, $\Gamma \vdash \mathcal{C}$ and $\mathcal{L} : \Gamma$ are safe for I_0 and I_1, respectively. Hence, $[\![\Gamma \vdash \kappa]\!]\sigma_0 \neq \top$ and $[\![\lambda : \Gamma]\!]\sigma_1 \neq \top$. By Lemma A.2, these disequalities imply that $[\![\tau]\!](\sigma_0 * \sigma_1) \neq \top$. We have just shown the safety of $\mathcal{C}(\mathcal{L})$ for $I_0 * I_1$.

Next, we show that

$$[\![\mathcal{C}(\mathcal{L}), I_0 * I_1]\!] \subseteq [\![\Gamma \vdash \mathcal{C}, I_0]\!] \otimes [\![\mathcal{L} : \Gamma, I_1]\!].$$

Pick $(\sigma, \tau) \in [\![\mathcal{C}(\mathcal{L}), I_0 * I_1]\!]$. Then for some σ_0, σ_1 we have

$$\sigma_0 \in I_0 \ \wedge\ \sigma_1 \in I_1 \ \wedge\ \sigma = \sigma_0 * \sigma_1 \ \wedge\ \tau \in (\!|\mathcal{C}(\mathcal{L})|\!) \ \wedge\ (\_, \tau') \in [\![\tau]\!](\sigma_0 * \sigma_1).$$

By Lemma A.3, there are κ, λ such that (A.14) holds. We use Lemma A.2 and deduce that for some κ', λ' we have

$$(\_, \kappa') \in [\![\Gamma \vdash \kappa]\!]\sigma_0 \ \wedge\ (\_, \lambda') \in [\![\lambda : \Gamma]\!]\sigma_1 \ \wedge\ \mathsf{cover}(\tau, \kappa', \lambda').$$

Furthermore, $\sigma_0 \in I_0$, $\sigma_1 \in I_1$ and $(\sigma_0 * \sigma_1)\!\downarrow$. Hence,

$$(\sigma, \tau) = (\sigma_0 * \sigma_1, \tau) \in [\![\Gamma \vdash \mathcal{C}, I_0]\!] \otimes [\![\mathcal{L} : \Gamma, I_1]\!],$$

as desired.

Finally, we prove that

$$[\![\mathcal{C}(\mathcal{L}), I_0 * I_1]\!] \supseteq [\![\Gamma \vdash \mathcal{C}, I_0]\!] \otimes [\![\mathcal{L} : \Gamma, I_1]\!].$$

Pick $(\sigma, \tau) \in [\![\Gamma \vdash \mathcal{C}, I_0]\!] \otimes [\![\mathcal{L} : \Gamma, I_1]\!]$. By the definition of the $\otimes$ operator and our semantics, there exist $\sigma_0, \sigma_1, \sigma_0', \sigma_1', \kappa, \lambda, \kappa', \lambda'$ such that

$$\begin{aligned}\sigma = \sigma_0 * \sigma_1 \ \wedge\ \sigma_0 \in I_0 \ \wedge\ \sigma_1 \in I_1 \ \wedge\ \kappa \in (\!|\Gamma \vdash \mathcal{C}|\!) \ \wedge\ \lambda \in (\!|\mathcal{L} : \Gamma|\!)\\ \wedge\ (\sigma_0', \kappa') \in [\![\Gamma \vdash \kappa]\!]\sigma_0 \ \wedge\ (\sigma_1', \lambda') \in [\![\lambda : \Gamma]\!]\sigma_1 \ \wedge\ \mathsf{cover}(\tau, \kappa', \lambda').\end{aligned}$$

By the definition of our semantics, $\kappa = \mathsf{ground}(\kappa')$ and $\lambda = \mathsf{ground}(\lambda')$. Because of this and $\mathsf{cover}(\tau, \kappa', \lambda')$ we have $\mathsf{cover}(\tau, \kappa, \lambda)$. By Lemma A.3, this implies $\tau \in (\!|\mathcal{C}(\mathcal{L})|\!)$. Also, by Lemma A.2, we have that

$$(\sigma_0' * \sigma_1')\!\downarrow \wedge\, (\sigma_0' * \sigma_1', \tau) \in [\![\tau]\!](\sigma_0 * \sigma_1).$$

Hence,

$$(\sigma, \tau) = (\sigma_0 * \sigma_1, \tau) \in [\![\mathcal{C}(\mathcal{L}), I_0 * I_1]\!],$$

as desired. □

A.3. **Proof of Lemma 8.3.** Consider $\lambda \in [\![\mathcal{L} : \Gamma']\!](\sigma_0 * \sigma_0')$. Then there exist σ_1 and $\zeta \in (\!|\mathcal{L}|\!)$ such that $(\sigma_1, \lambda) \in [\![\zeta : \Gamma']\!](\sigma_0 * \sigma_0')$. We show that for some σ_2 we have

$$((\sigma_2, \lambda\!\!\rfloor_\Gamma) \in [\![\zeta : \Gamma]\!]\sigma_0) \ \wedge \ (\sigma_1 = \sigma_2 * (\langle \lceil\!\lceil \mathsf{history}(\lambda) \rceil\!\rceil_\Gamma \rangle \sigma_0')).$$

We proceed by induction on the length of ζ. The base case of $\zeta = \varepsilon$ is trivial. Assume that the above holds for some $\lambda, \zeta, \sigma_1, \sigma_2$ and consider $\varphi, \varphi', \sigma_1'$ such that

$$\zeta\varphi \in (\!|\mathcal{L}|\!) \ \wedge \ (\sigma_1', \varphi') \in [\![\varphi : \Gamma']\!]\sigma_1 \ \wedge \ \langle \lceil\!\lceil \mathsf{history}(\lambda\varphi') \rceil\!\rceil_\Gamma \rangle \sigma_0' \neq \top.$$

We show that for some σ_2' we have

$$((\sigma_2', \lambda\varphi'\!\!\rfloor_\Gamma) \in [\![\zeta\varphi : \Gamma]\!]\sigma_0) \ \wedge \ (\sigma_1' = \sigma_2' * (\langle \lceil\!\lceil \mathsf{history}(\lambda\varphi') \rceil\!\rceil_\Gamma \rangle \sigma_0')).$$

We consider three cases, depending on the type of the actions φ and φ'.

- $\varphi = \varphi' = (t, c)$. Then $\mathsf{history}(\lambda\varphi') = \mathsf{history}(\lambda)$. Since $\mathcal{L} : \Gamma$ is safe at σ_0, $f_c^t(\sigma_2) \neq \top$. Hence, by the Strong Locality property, we have

  $$\sigma_1' \in f_c^t(\sigma_1) = f_c^t(\sigma_2 * (\langle \lceil\!\lceil \mathsf{history}(\lambda) \rceil\!\rceil_\Gamma \rangle \sigma_0')) = f_c^t(\sigma_2) * \{\langle \lceil\!\lceil \mathsf{history}(\lambda) \rceil\!\rceil_\Gamma \rangle \sigma_0'\} = f_c^t(\sigma_2) * \{\langle \lceil\!\lceil \mathsf{history}(\lambda\varphi') \rceil\!\rceil_\Gamma \rangle \sigma_0'\}.$$

  Then $\sigma_1' = \sigma_2' * (\langle \lceil\!\lceil \mathsf{history}(\lambda\varphi') \rceil\!\rceil_\Gamma \rangle \sigma_0')$ for some $\sigma_2' \in f_c^t(\sigma_2)$.
- $\varphi = (t, \mathsf{call}\ m)$ and $\varphi' = (t, \mathsf{call}\ m(\sigma_p * \sigma_p'))$, where $\sigma_p \in p_t^m$ and $\sigma_p * \sigma_p' \in r_t^m$. Then

  $$\sigma_1' = \sigma_1 * \sigma_p * \sigma_p' = (\sigma_2 * \sigma_p) * ((\langle \lceil\!\lceil \mathsf{history}(\lambda) \rceil\!\rceil_\Gamma \rangle \sigma_0') * \sigma_p') = (\sigma_2 * \sigma_p) * (\langle \lceil\!\lceil \mathsf{history}(\lambda\varphi') \rceil\!\rceil_\Gamma \rangle \sigma_0').$$

  Hence, the required holds for $\sigma_2' = \sigma_2 * \sigma_p$.
- $\varphi = (t, \mathsf{ret}\ m)$ and $\varphi' = (t, \mathsf{ret}\ m(\sigma_q * \sigma_q'))$, where $\sigma_q \in q_t^m$ and $\sigma_q * \sigma_q' \in s_t^m$. Then

  $$\sigma_2 * (\langle \lceil\!\lceil \mathsf{history}(\lambda) \rceil\!\rceil_\Gamma \rangle \sigma_0') = \sigma_1 = \sigma_1' * \sigma_q * \sigma_q'.$$

  Since $\mathcal{L} : \Gamma$ is safe at σ_0, $\sigma_2 = \sigma_2' * \sigma_q$ for some σ_2', so that

  $$\sigma_2' * \sigma_q * (\langle \lceil\!\lceil \mathsf{history}(\lambda) \rceil\!\rceil_\Gamma \rangle \sigma_0') = \sigma_1' * \sigma_q * \sigma_q'.$$

  By the cancellativity of $*$, this entails

  $$\sigma_2' * (\langle \lceil\!\lceil \mathsf{history}(\lambda) \rceil\!\rceil_\Gamma \rangle \sigma_0') = \sigma_1' * \sigma_q'.$$

  We also know that $\langle \lceil\!\lceil \mathsf{history}(\lambda\varphi') \rceil\!\rceil_\Gamma \rangle \sigma_0' \neq \top$, so that

  $$\langle \lceil\!\lceil \mathsf{history}(\lambda\varphi') \rceil\!\rceil_\Gamma \rangle \sigma_0' = (\langle \lceil\!\lceil \mathsf{history}(\lambda) \rceil\!\rceil_\Gamma \rangle \sigma_0') \setminus \sigma_q'$$

  is defined. Hence, $\sigma_1' = \sigma_2' * (\langle \lceil\!\lceil \mathsf{history}(\lambda\varphi') \rceil\!\rceil_\Gamma \rangle \sigma_0')$. □

A.4. **Proof of Lemma 8.4.** In the following, we extend the $[\![\cdot]\!]_\Gamma$ operation to non-interface actions by assuming that it does not change them.

Consider $\lambda, \sigma_0, \sigma_0', \sigma_0''$ satisfying the conditions of the lemma. Then $\mathsf{history}(\lambda)$ is balanced from $\delta(\sigma_0'' * \sigma_0')$ for $\delta(\sigma_0) \preceq \delta(\sigma_0'')$, and there exist σ and $\zeta \in (\!|\mathcal{L}|\!)$ such that $(\sigma, [\![\lambda]\!]_\Gamma) \in [\![\zeta : \Gamma]\!]\sigma_0$. We show that

$$(\sigma * (\langle \lceil\!\lceil \mathsf{history}(\lambda) \rceil\!\rceil_\Gamma \rangle \sigma_0'), \lambda) \in [\![\zeta : \Gamma']\!](\sigma_0 * \sigma_0')$$

by induction on the length of ζ. The base case of $\zeta = \varepsilon$ is trivial. Assume that the above holds for some $\lambda, \sigma_0, \sigma_0', \sigma_0'', \sigma, \zeta$ and consider $\varphi, \varphi', \sigma'$ such that

$$(\zeta\varphi \in (\!|\mathcal{L}|\!)) \ \wedge \ ((\sigma', [\![\varphi']\!]_\Gamma) \in [\![\varphi : \Gamma]\!]\sigma) \ \wedge$$
$$(\mathsf{history}(\lambda\varphi') \text{ is balanced from } \delta(\sigma_0'' * \sigma_0') \text{ for } \delta(\sigma_0) \preceq \delta(\sigma_0'')) \ \wedge \ (\langle \lceil\!\lceil \mathsf{history}(\lambda\varphi') \rceil\!\rceil_\Gamma \rangle \sigma_0' \neq \top).$$

We show that

$$(\sigma' * (\langle \lceil\!\lceil \mathsf{history}(\lambda\varphi') \rceil\!\rceil_\Gamma \rangle \sigma_0'), \lambda\varphi') \in [\![\zeta\varphi : \Gamma']\!](\sigma_0 * \sigma_0').$$

We consider three cases, depending on the type of the actions φ and φ'.

- $\varphi = \varphi' = (t, c)$. Then $\mathsf{history}(\lambda) = \mathsf{history}(\lambda\varphi')$ and $\sigma' \in f_c^t(\sigma)$. Since
$$(\sigma * (\langle \lceil\!\lceil \mathsf{history}(\lambda) \rceil\!\rceil_\Gamma \rangle \sigma_0'))\!\downarrow,$$
by the Footprint Preservation property, $(\sigma' * (\langle \lceil\!\lceil \mathsf{history}(\lambda) \rceil\!\rceil_\Gamma \rangle \sigma_0'))\!\downarrow$. Then by the Strong Locality property,
$$\sigma' * (\langle \lceil\!\lceil \mathsf{history}(\lambda\varphi') \rceil\!\rceil_\Gamma \rangle \sigma_0') = \sigma' * (\langle \lceil\!\lceil \mathsf{history}(\lambda) \rceil\!\rceil_\Gamma \rangle \sigma_0') \in$$
$$f_c^t(\sigma) * \{\langle \lceil\!\lceil \mathsf{history}(\lambda) \rceil\!\rceil_\Gamma \rangle \sigma_0'\} = f_c^t(\sigma * (\langle \lceil\!\lceil \mathsf{history}(\lambda) \rceil\!\rceil_\Gamma \rangle \sigma_0')).$$
- $\varphi = (t, \mathsf{call}\ m)$ and $\varphi' = (t, \mathsf{call}\ m(\sigma_p * \sigma_p'))$, where $\sigma_p \in p_t^m$ and $\sigma_p * \sigma_p' \in r_t^m$. In this case we have $\sigma' = \sigma * \sigma_p$. Since $\mathsf{history}(\lambda\varphi')$ is balanced from $\delta(\sigma_0'' * \sigma_0')$, we have $(\sigma * (\langle \lceil\!\lceil \mathsf{history}(\lambda) \rceil\!\rceil_\Gamma \rangle \sigma_0') * \sigma_p * \sigma_p')\!\downarrow$. Then

$$\sigma * (\langle \lceil\!\lceil \mathsf{history}(\lambda) \rceil\!\rceil_\Gamma \rangle \sigma_0') * \sigma_p * \sigma_p' = \sigma' * ((\langle \lceil\!\lceil \mathsf{history}(\lambda) \rceil\!\rceil_\Gamma \rangle \sigma_0') * \sigma_p') = \sigma' * \langle \lceil\!\lceil \mathsf{history}(\lambda\varphi') \rceil\!\rceil_\Gamma \rangle \sigma_0'.$$

- $\varphi = (t, \mathsf{ret}\ m)$ and $\varphi' = (t, \mathsf{ret}\ m(\sigma_q * \sigma_q'))$, where $\sigma_q \in q_t^m$ and $\sigma_q * \sigma_q' \in s_t^m$. In this case we have $\sigma' = \sigma \setminus \sigma_q$. We know $\langle \lceil\!\lceil \mathsf{history}(\lambda\varphi') \rceil\!\rceil_\Gamma \rangle \sigma_0' \neq \top$. Thus, $((\langle \lceil\!\lceil \mathsf{history}(\lambda) \rceil\!\rceil_\Gamma \rangle \sigma_0') \setminus \sigma_q')\!\downarrow$. Since $\sigma * (\langle \lceil\!\lceil \mathsf{history}(\lambda) \rceil\!\rceil_\Gamma \rangle \sigma_0')$ is defined, so is
$$\sigma' * (\langle \lceil\!\lceil \mathsf{history}(\lambda\varphi') \rceil\!\rceil_\Gamma \rangle \sigma_0') = \sigma' * ((\langle \lceil\!\lceil \mathsf{history}(\lambda) \rceil\!\rceil_\Gamma \rangle \sigma_0') \setminus \sigma_q') =$$
$$(\sigma \setminus \sigma_q) * ((\langle \lceil\!\lceil \mathsf{history}(\lambda) \rceil\!\rceil_\Gamma \rangle \sigma_0') \setminus \sigma_q') = (\sigma * (\langle \lceil\!\lceil \mathsf{history}(\lambda) \rceil\!\rceil_\Gamma \rangle \sigma_0')) \setminus (\sigma_q * \sigma_q'). \ \square$$

Glossary

Symbol	Meaning and section
Σ	separation algebra, 2.1
$*$	operation accompanying a separation algebra, 2.1
e	unit of a separation algebra, 2.1
σ, θ	state, 2.1
p, q, r, s	predicate on states, 2.1; parameterised predicate, 4.1
$g(x)\!\downarrow$	function g is defined on x, 2.1
$g(x)\!\uparrow$	function g is undefined on x, 2.1
$g[x : y]$	the function that has the same value as g everywhere,

	except for x, where it has the value y, 2.1
π	permission, 2.1
$\backslash$	subtraction: on states, 2.1; on a state and a predicate, 4.1
l	footprint, 2.2
$\delta(\sigma)$	footprint of a state σ, 2.2
$\mathcal{F}(\Sigma)$	the set of all footprints in a separation algebra Σ, 2.2
$\circ$	addition operation on footprints, 2.2
$\backslash\backslash$	subtraction operation on footprints, 2.2
$\preceq$	"smaller-than" relation on footprints, 2.2
ψ	interface action, 3
t	thread identifier, 3
m	library method, 3
H, S	history, 3
ε	empty history or trace, 3
$\tau(i)$	the i-th element of τ, 3
$\tau\downarrow_k$	the prefix of τ of length k, 3
$\lvert\tau\rvert$	is the length of τ, 3
$[\![H]\!]^\sharp$	footprint tracking function, 3
$\mathcal{H}$	interface set, 3
$\sqsubseteq$	linearizability: on histories, 3; interface sets, 3; libraries, 6
ρ	bijection on history indices, 3
c	primitive command, 4
C	command, 4
$\mathcal{L}$	library, 4
$\mathcal{S}$	complete program, 4
$\mathcal{C}$	open program with a client, 4
$\mathcal{P}$	open or complete program, 4
Γ	method specification, 4.1
$\top$	error state, 4.2
f_c^t	transformer for a primitive command c and thread identifier t, 4.2
φ	action, 5.1
τ	trace, 5.1
κ	client trace, 5.1
$\lambda, \zeta, \alpha, \beta$	library trace, 5.1
$\mathsf{lib}(\tau)$	projection of τ to library actions, calls and returns, 5.1
$\mathsf{client}(\tau)$	projection of τ to client actions, calls and returns, 5.1
$\mathsf{history}(\tau)$	projection of τ to calls and returns, 5.1
η	mapping from methods to trace sets, 5.2
$(\!\lvert \Gamma \vdash \mathcal{P} : \Gamma' \rvert\!)$	trace set, 5.2
$[\![\Gamma \vdash \mathcal{P} : \Gamma']\!]$	denotation of a program $\mathcal{P}$, 5.3
$[\![\Gamma \vdash \tau : \Gamma']\!]$	evaluation of a trace τ, 5.3
$[\![\Gamma \vdash \varphi : \Gamma']\!]$	evalutation of an action φ, 5.3
I	set of initial states, 5.4
$[\![\mathcal{P}, I]\!]$	set of traces of $\mathcal{P}$ run from states in I, 5.4
$\mathsf{ground}(\tau)$	erasure of state annotations from actions in τ, 5.4
$\mathsf{interf}(\mathcal{L}, I)$	interface set of $\mathcal{L}$ run from initial states in I, 6
$\sim$	equivalence of client traces, 6

$[\![\lambda]\!]$ selects state annotations corresponding to unextended specifications, 8
$[\![\lambda]\!]$ selects state annotations recording extra state, 8
$\langle H \rangle$ evaluation of a history H recording extra state, 8

Parameterised Linearisability

Andrea Cerone[1], Alexey Gotsman[1], and Hongseok Yang[2]

[1] IMDEA Software Institute
[2] University of Oxford

Abstract Many concurrent libraries are parameterised, meaning that they implement generic algorithms that take another library as a parameter. In such cases, the standard way of stating the correctness of concurrent libraries via linearisability is inapplicable. We generalise linearisability to parameterised libraries and investigate subtle trade-offs between the assumptions that such libraries can make about their environment and the conditions that linearisability has to impose on different types of interactions with it. We prove that the resulting parameterised linearisability is closed under instantiating parameter libraries and composing several non-interacting libraries, and furthermore implies observational refinement. These results allow modularising the reasoning about concurrent programs using parameterised libraries and confirm the appropriateness of the proposed definitions. We illustrate the applicability of our results by proving the correctness of a parameterised library implementing flat combining.

1 Introduction

Concurrent libraries encapsulate high-performance concurrent algorithms and data structures behind a well-defined interface, providing a set of methods for clients to call. Many such libraries [6,7,13] are *parameterised*, meaning that they implement generic algorithms that take another library as a parameter and use it to implement more complex functionality (we give a concrete example in §2). Reasoning about the correctness of parameterised libraries is challenging, as it requires considering all possible libraries that they can take as parameters.

Correctness of concurrent libraries is usually stated using *linearisability* [8], which fixes a certain correspondence between the *concrete* library implementation and a (possibly simpler) *abstract* library, whose behaviour the concrete one is supposed to simulate. For example, a high-performance concurrent stack that allows multiple push and pop operations to access the data structure at the same time may be specified by an abstract library where each operation takes effect atomically. However, linearisability considers only *ground* libraries, where all of the library implementation is given, and is thus inapplicable to parameterised ones. In this paper we propose a notion of *parameterised linearisability* (§3 and §4) that lifts this limitation. The key idea is to take into account not only interactions of a library with its client, but also with its parameter library, with the two types of interactions being subject to different conditions.

A challenge we have to deal with while generalising linearisability in this way is that parameterised libraries are often correct only under some assumptions about the context in which they are used. Thus, a parameterised library may assume that the library it takes as a parameter is *encapsulated*, meaning that clients cannot call its methods directly. A parameterised library may also accept as a parameter only libraries satisfying certain properties. For this reason, we actually present three notions of parameterised

linearisability, appropriate for different situations: a general one, which does not make any assumptions about the client or the parameter library, a notion appropriate for the case when the parameter library is encapsulated, and *up-to linearisability*, which allows making assumptions about the parameter library. These notions differ in subtle ways: we find that there is a trade-off between the assumptions that parameterised libraries make about their environment and the conditions that a notion of linearisability has to impose on different types of interactions with it.

We prove that the proposed notions of parameterised linearisability are *contextual* (§5), i.e., closed under parameter instantiation. This includes the case when the parameter library is itself parameterised. On the other hand, when the parameter is an ordinary ground library, this result allows us to derive the classical linearisability of the instantiated library from our notion for the parameterised one. We also prove that parameterised linearisability is *compositional* (§5): if several non-interacting libraries are linearisable, so is their composition. Finally, we show that parameterised linearisability implies *observational refinement* (§6): the behaviours of any complete program using a concrete parameterised library can be reproduced if the program uses a corresponding abstract one instead. All these results allow modularising the reasoning about concurrent programs using parameterised libraries: contextuality and compositionality break the reasoning about complex parameterised libraries into that about individual libraries from which they are constructed; observational refinement then lifts this to complete programs, including clients. The properties of parameterised linearisability we establish also serve to confirm the appropriateness of the proposed definitions.

We illustrate the applicability of our results by proving the up-to linearisability of flat combining [6] (§4), a generic algorithm for converting hard-to-parallelise sequential data structures into concurrent ones.

Due to space constraints, we defer the proofs of most theorems to [1, §B].

2 Parameterised Libraries

We consider ***parameterised libraries*** (or simply libraries) L, which provide some ***public methods*** to their ***clients***. The latter are multi-threaded programs that can call the methods in parallel. In §4 and §6 we introduce a particular syntax for libraries and clients; for now it suffices to treat them abstractly. Our libraries are called parameterised because we allow their method implementations to call ***abstract methods***, whose implementation is left unspecified. Abstract methods are meant to be implemented by another library provided by L's client, which we call the ***parameter library*** of L.

We identify methods by names from a set $\mathcal{M}$, ranged over by m, and threads by identifiers from a set $\mathcal{T}$, ranged over by t. For the sake of simplicity, we assume that methods take a single integer as a parameter and always return an integer. We annotate libraries with types as in $L : M \to M'$, where $M, M' \subseteq \mathcal{M}$ give the sets of abstract and public methods of L, respectively. If $M = \emptyset$ we call L a ***ground library***. The sets M and M' do not have to be disjoint: methods in $M \cap M'$ may be called by L's clients, but their implementation is inherited from the one given by the parameter library.

Example: Flat Combining. Flat combining [6] is a recent synchronisation paradigm, which can be viewed [14] as a parameterised library $\mathsf{FC} : \{m_i\}_{i=1}^n \to \{\mathsf{do\_}m_i\}_{i=1}^n$ for a given set of methods $\{m_i\}_{i=1}^n$. In Figure 1 we show a pseudocode of its implementation, which simplifies the original one in ways orthogonal to our goals. FC takes a library,

whose methods m_i are meant to be executed sequentially, and efficiently turns it into a library with methods do_m_i that can be called concurrently.

As usual, this is achieved by means of mutual exclusion, implemented using a lock, but in a way that is more sophisticated than just acquiring it before calling a method m_i. A thread executing do_m_i first publishes the operation it would like to execute and its parameter in its entry of the requests array. It then spins, trying to acquire the global lock. Having acquired a lock, the thread becomes a *combiner*: it performs the operations requested by all threads, stored in requests, by calling methods m_i of the parameter library and writing the values returned into the retval field of the corresponding entries in requests. Each spinning thread periodically checks this field and stops if some other thread has performed the operation it requested (for simplicity, we assume that nil is a special value that is never returned by any method). This algorithm benefits from cache locality when the combiner executes several operations in sequence, and thus yields good performance even for hard-to-parallelise data structures, such as stacks and queues.

```
LOCK lock;
struct{op,param,retval} requests[NThread];

do_m_i(int z):
  requests[mytid()].op = i;
  requests[mytid()].param = z;
  requests[mytid()].retval = nil;
  do:
    if (lock.tryacquire()):
      for (t = 0; t < NThread; t++):
        if (requests[t].retval == nil):
          int j = requests[t].op;
          int w = requests[t].param;
          requests[t].retval = m_j(w);
      lock.release();
  while (requests[mytid()].retval == nil);
  return requests[mytid()].retval;
```

Figure 1. Flat combining: implementation FC.

In this paper, we develop a framework for specifying and verifying parameterised concurrent libraries. For flat combining, our framework suggests using an ***abstract*** library $\mathsf{FC}^\sharp : \{m_i\}_{i=1}^n \to \{\mathsf{do}\_m_i\}_{i=1}^n$ in Figure 2 as a specification for the ***concrete*** library in Figure 1. $\mathsf{FC}^\sharp$ specifies the expected behaviour of flat combining by using the naive mutual exclusion. Showing that the implementation satisfies this specification in our framework amounts to proving that it is related to $\mathsf{FC}^\sharp$ by *parameterised linearisability*, which we present next.

```
LOCK lock;
do_m_i(int z):
  lock.acquire();
  int retval = m_i(z);
  lock.release();
  return retval;
```

Figure 2. Flat combining: specification $\mathsf{FC}^\sharp$.

3 Histories and Parameterised Linearisability

Histories. Informally, for a concrete library (such as the one in Figure 1) to be correct with respect to an abstract one (such as the one in Figure 2), the two should interact with their environment—the client and the parameter library—in similar ways. In this paper, we assume that different libraries and their clients access disjoint portions of memory, and thus interactions between them are limited to passing parameters and return values at method calls and returns. This is a standard assumption [8], which we believe can be relaxed using existing techniques [5]; see §7 for discussion. We record interactions of a parameterised library $L : M \to M'$ with its environment using *histories* (Definition 1 below), which are certain sequences of ***actions*** of the form

$$\mathsf{Act} ::= (t, \mathsf{call?}\, m'(z)) \mid (t, \mathsf{ret!}\, m'(z)) \mid (t, \mathsf{call!}\, m(z)) \mid (t, \mathsf{ret?}\, m(z)),$$

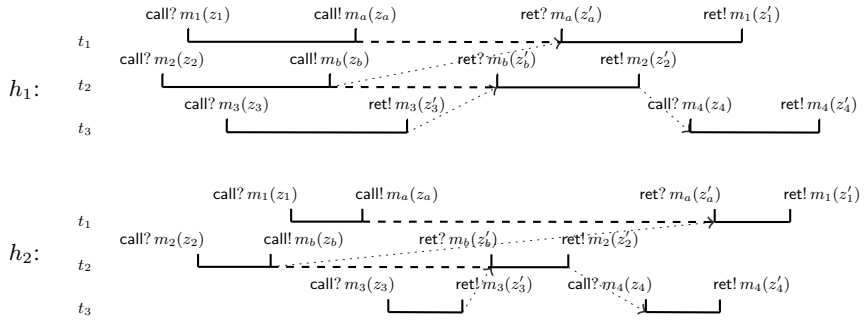


Figure 4. Illustration of histories and parameterised linearisability. A solid line represents a thread executing the code of the parameterised library, and a dashed one, the parameter library.

where $t \in \mathcal{T}$ is the thread performing the action, $m' \in M'$ or $m \in M$ is the method involved, and $z \in \mathbb{Z}$ is the method parameter or a return value.

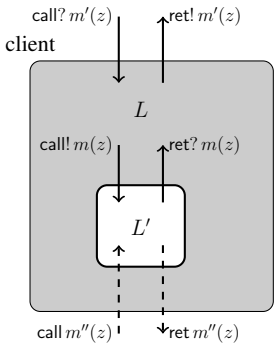


Figure 3. Interactions of a library L with its client and parameter library L'.

We illustrate the meaning of the actions in Figure 3: call? and ret! describe the client invoking public methods m' of the parameterised library L, and call! and ret? the library L invoking implementations of abstract methods m provided by a parameter library L'. We denote the sets of actions corresponding to interactions with these two entities by ClAct and AbsAct, respectively. In the spirit of the opponent-proponent distinction in game semantics [9,11], we annotate actions by ! or ? depending on whether the action was initiated by L or by an external entity, and we denote the corresponding sets of actions by Act! and Act?. We also use sets ActCall?, ActRet!, ActCall! and ActRet? with the expected meaning. Clients can also call methods $m'' \in M \cap M'$ directly, as represented by the dashed lines in the figure. Since such interactions do not involve the library L, we do not include them into Act. Histories are finite sequences of actions with invocations of abstract methods properly nested inside those of public ones.

DEFINITION 1 (Histories) *A **history** $h : M \to M'$ is a finite sequence of actions such that for every t, the projection of h to t's actions is a prefix of a sequence generated by the grammar* SHist *below, where $m \in M$ and $m' \in M'$:*

$$\begin{array}{lll} \mathsf{SHist} & ::= \varepsilon \mid (t, \mathsf{call?}\, m'(z))\, \mathsf{IntSHist}\, (t, \mathsf{ret!}\, m'(z')) \mid \mathsf{SHist}\ \mathsf{SHist} \\ \mathsf{IntSHist} & ::= \varepsilon \mid (t, \mathsf{call!}\, m(z))\, (t, \mathsf{ret?}\, m(z')) \mid \mathsf{IntSHist}\ \mathsf{IntSHist} \end{array}$$

We denote the set of histories by Hist. See Figure 4 for examples. In this paper, we focus on safety properties of libraries and thus let histories be finite. This assumption is also taken by the classical notion of linearisability [8] and can be relaxed as described in [4] (§7). For a history h and $A \subseteq \mathsf{Act}$, we let $h|_A$ be the projection of h onto actions in A and we denote the i-th action in h by $h(i)$.

Parameterised Linearisability. We would like the notion of correctness of a concrete library $L : M \to M'$ with respect to an abstract one $L^\sharp : M \to M'$ to imply *observational refinement*. Informally, this property means that $L^\sharp$ can be used to replace L in any program (consisting of a client, the library and an instantiation of the parameter library) while keeping its observable behaviours reproducible; a formal definition is given in §6. While this notion is intuitive, establishing it between two libraries directly is challenging because of the quantification over all possible programs they can be used by. We therefore set out to find a correctness criterion that compares the concrete and abstract libraries in isolation and thus avoids this quantification. For ground libraries, ***linearisability*** [8] formulates such a criterion by matching a history h_1 of L with a history h_2 of $L^\sharp$ that yields the same client-observable behaviour. The following definition generalises it to parameterised libraries.

DEFINITION 2 (Parameterised linearisability: general case) *A history $h_1 : M \to M'$ is **linearised** by another one $h_2 : M \to M'$, written $h_1 \sqsubseteq h_2$, if there exists a permutation $\pi : \mathbb{N} \to \mathbb{N}$ such that*

$$\forall i.\, h_1(i) = h_2(\pi(i)) \wedge (\forall j.\, i < j \wedge ((\exists t.\, h_1(i) = (t, -) \wedge h_2(j) = (t, -)) \vee (h_1(i) \in \mathsf{Act!} \wedge h_1(j) \in \mathsf{Act?})) \implies \pi(i) < \pi(j)).$$

For sets of histories H_1, H_2 we let $H_1 \sqsubseteq H_2 \iff \forall h_1 \in H_1.\, \exists h_2 \in H_2.\, h_1 \sqsubseteq h_2$.

In §4 we show how to generate all histories of a library in a particular language and define linearisability on libraries by the $\sqsubseteq$ relation on their sets of histories. For now we explain the above abstract definition. According to it, a history h_1 is linearised by a history h_2 when the latter is a permutation of the former preserving the order of actions within threads and the precedence relation between the actions initiated by the library and those initiated by its environment. As we explain below, we have $h_1 \sqsubseteq h_2$ for the histories h_1, h_2 in Figure 4. Hence, parameterised linearisability is able to match a history of a concurrent library with a simpler one where every contiguous block of library execution (e.g., the one between $(t_1, \mathsf{call?}\, m_1(z_1))$ and $(t_1, \mathsf{call!}\, m_a(z_a))$) is executed without interleaving with other such blocks. On the other hand, $h_2 \not\sqsubseteq h_1$, since $(t_1, \mathsf{call!}\, m_a(z_a))$ precedes $(t_3, \mathsf{call?}\, m_3(z_3))$ in h_2, but not in h_1.

When $h_1, h_2 : \emptyset \to M'$, i.e., these are histories of a ground library and thus contain only call? and ret! actions, Definition 2 coincides with a variant of the classical linearisability [8], which requires preserving the order between ret! and call? actions. For example, Definition 2 requires preserving the order between $(t_2, \mathsf{ret!}\, m_2(z_2'))$ and $(t_3, \mathsf{call?}\, m_4(z_4))$ in h_1 from Figure 4 (shown by a diagonal arrow). This requirement is needed for linearisability to imply observational refinement: informally, during the interval of time between $(t_2, \mathsf{ret!}\, m_2(z_2'))$ and $(t_3, \mathsf{call?}\, m_4(z_4))$ in an execution of a program producing h_1, both threads t_2 and t_3 execute pieces of client code, which can communicate via the client memory. To preserve the behaviour of the client when replacing the concrete library in the program by an abstract one in observational refinement, this communication must not be affected, and, for this, the abstract library has to admit a history in which the order between the above actions is preserved.

When $h_1, h_2 : M \to M'$ correspond to a non-ground parameterised library, i.e., $M \neq \emptyset$, a similar situation arises with communication between the methods of the parameter library executing in different threads. For this reason, our generalisation of linearisability requires preserving the order between call! and ret? actions, such as

$(t_2, \mathsf{call!}\, m_b(z_b))$ and $(t_1, \mathsf{ret?}\, m_a(z'_a))$ in Figure 4; this requirement is dual to the one considered in classical linearisability. It is not enough, however. Definition 2 also requires preserving the order between call! and call?, as well as ret! and ret? actions, e.g., $(t_3, \mathsf{ret!}\, m_3(z'_3))$ and $(t_2, \mathsf{ret?}\, m_b(z'_b))$ in Figure 4. In the case when $M \cap M' \neq \emptyset$, this is also required to validate observational refinement. For example, during the interval of time between $(t_3, \mathsf{ret!}\, m_3(z'_3)$ and $(t_2, \mathsf{ret?}\, m_b(z'_b))$ in an execution producing h_1, the client code in thread t_3 can call a method $m'_b \in M \cap M'$ of the parameter library (cf. the dashed arrows in Figure 3). The code of the method m'_b executed by t_3 can then communicate with that of the method m_b executed by t_2, and to preserve this communication, we need to preserve the order between $(t_3, \mathsf{ret!}\, m_3(z'_3))$ and $(t_2, \mathsf{ret?}\, m_b(z'_b))$.

In §5 and §6 we prove that the above notion of linearisability indeed validates observational refinement. If the library $L : M \to M'$ producing the histories h_1, h_2 in Definition 2 is such that $M \cap M' = \emptyset$, then the client cannot directly call methods of its parameter library, and, as we show, parameterised linearisability can be weakened without invalidating observational refinement.

DEFINITION 3 (Parameterised linearisability: encapsulated case) *For $h_1, h_2 : M \to M'$ with $M \cap M' = \emptyset$ we let $h_1 \sqsubseteq_{\mathsf{e}} h_2$ if there exists a permutation $\pi : \mathbb{N} \to \mathbb{N}$ such that*

$$\forall i.\, h_1(i) = h_2(\pi(i)) \wedge (\forall j.\, i < j \wedge ((\exists t.\, h_1(i) = (t, -) \wedge h_2(j) = (t, -)) \vee (h_1(i), h_1(j)) \in (\mathsf{ActRet!} \times \mathsf{ActCall?}) \cup (\mathsf{ActCall!} \times \mathsf{ActRet?})) \implies \pi(i) < \pi(j)).$$

Since this definition does not take into account the order between $(t_1, \mathsf{call!}\, m_a(z_a))$ and $(t_3, \mathsf{call?}\, m_3(z_3))$ in h_2 from Figure 4, we have $h_2 \sqsubseteq_{\mathsf{e}} h_1$ even though $h_2 \not\sqsubseteq h_1$.

Definitions 2 and 3 do not make any assumptions about the implementation of the parameter library. However, sometimes the correctness of a parameterised library can only be established under certain assumptions about the behaviour of its parameter. In particular, this is the case for the flat combining library from §2. In its implementation FC from Figure 1, a request by a thread t to execute a method m_i of the parameter library can be fulfilled by another thread t' who happens to act as a combiner; in contrast, the specification $\mathsf{FC}^\sharp$ in Figure 2 pretends that m_i is executed in the requesting thread. Thus, FC and $\mathsf{FC}^\sharp$ will behave differently if we supply as their parameter a library whose methods depend on the identifiers of executing threads (e.g., with m_i implemented as "return mytid()"). As a consequence, FC does not simulate $\mathsf{FC}^\sharp$. On the other hand, this will be the case if we restrict ourselves to parameter libraries whose behaviour is independent of thread identifiers. The following version of parameterised linearisability allows us to use such assumptions, formulated as closure properties on histories of interactions between a parameterised library and its parameter. Given a history h, let $\overline{h}$ be the history obtained by swapping ! and ? actions in h.

DEFINITION 4 (Up-to linearisability) *For $h_1, h_2 : M \to M'$ such that $M \cap M' = \emptyset$ and a binary relation $\mathcal{R}$ on histories of type $\emptyset \to M$, we say that h_1 **is linearised by** h_2 **up to** $\mathcal{R}$, written $h_1 \sqsubseteq_{\mathcal{R}} h_2$, if $(h_1|_{\mathsf{ClAct}}) \sqsubseteq (h_2|_{\mathsf{ClAct}})$ and $\overline{(h_1|_{\mathsf{AbsAct}})}\ \mathcal{R}\ \overline{(h_2|_{\mathsf{AbsAct}})}$.*

For flat combining, a suitable relation $\mathcal{R}_{\mathsf{t}}$ relates two histories if one can be obtained from the other by replacing thread identifiers of some pairs of a call and a corresponding (if any) return action. There are other useful choices of $\mathcal{R}$, such as equivalence up to commuting abstract method invocations [7].

So far we have defined our notions of linearisability abstractly, on sets of histories. We next introduce a language for parameterised libraries and show how to generate sets

of histories of a library in this language. This lets us lift the notion of linearisability to libraries and prove that FC in Figure 1 is indeed linearised up to $\mathcal{R}_\mathsf{t}$ by $\mathsf{FC}^\sharp$ in Figure 2.

4 Lifting Linearisability to Libraries

Library Syntax. We use the following language to define libraries:

$$L ::= \langle \mathsf{public} : B; \mathsf{private} : B \rangle \qquad B ::= \varepsilon \mid (m \Leftarrow C); B \mid (\mathsf{abstract}\ m); B$$
$$C ::= c \mid m() \mid C; C \mid \mathsf{if}(E)\ \mathsf{then}\ C\ \mathsf{else}\ C \mid \mathsf{while}(E)\ C$$

A parameterised library L is a collection of methods, some implemented by commands C and others declared as abstract, meant to be implemented by a parameter library. Methods can be public or private, with only the former made available to clients. In §5 and §6 we extend the language to complete programs, consisting of a multithreaded client using a parameterised library with its parameter instantiated. In particular, we introduce private methods here to define parameter library instantiation in §5.

In commands, c ranges over ***primitive commands*** from a set PComm, and E over expressions, whose set we leave unspecified. The command $m()$ invokes the method m; it does not mention its parameter or return value, since, as we explain below, these are passed via dedicated thread-local memory locations. We consider only well-formed libraries where a method is declared at most once and every method called is declared. We identify libraries up to the order of method declarations and α-renaming of private non-abstract methods. For a library $L = \langle \mathsf{public} : B_\mathsf{pub}; \mathsf{private} : B_\mathsf{pvt} \rangle$ we have $L : \mathsf{Abs}(L) \to \mathsf{Pub}(L)$, where $\mathsf{Pub}(L)$ is the set of methods declared in B_pub, and $\mathsf{Abs}(L)$ of those declared as abstract in B_pub or B_pvt.

Linearisability on Libraries and the Semantics Idea. We now show how to generate the set of histories $[\![L]\!] \in 2^{\mathsf{Hist}}$ of a library L. Then we let a library L_1 be ***linearised by*** a library L_2, written $L_1 \sqsubseteq L_2$, if $[\![L_1]\!] \sqsubseteq [\![L_2]\!]$; similarly for $\sqsubseteq_\mathsf{e}$ and $\sqsubseteq_\mathcal{R}$.

We actually generate all library *traces*, which, unlike histories, also record its internal actions. Let us extend the set of actions Act with elements of the forms (t, c) for $c \in \mathsf{PComm}$, $(t, \mathsf{call}\, m(z))$ and $(t, \mathsf{ret}\, m(z))$, leading to a set TrAct. The latter two kinds of actions correspond to calls and returns between methods implemented inside the library. A ***trace*** τ is a finite sequence of elements in TrAct; we let $\mathsf{Traces} = \mathsf{TrAct}^*$.

The denotation $[\![L]\!]$ of a library $L : M \to M'$ includes the histories extracted from traces that L produces in any possible environment, i.e., assuming that client threads perform any sequences of calls to methods in M' with arbitrary parameter values and that abstract methods in M return arbitrary values. The definition of $[\![L]\!]$ follows the intuitive semantics of our programming language. An impatient reader can skip it on first reading and jump directly to Theorem 1 at the end of this section.

Heaps and Primitive Command Semantics. Let Locs be the set of memory locations. As we noted in §3, we impose a standard restriction that different libraries and their clients access different sets of memory locations, except the ones used for method parameter passing. Formally, we assume that each library L is associated with a set of its locations $\mathsf{Locs}_L \subseteq \mathsf{Locs}$. The state of L is thus given by a ***heap*** $\sigma \in \mathsf{Locs}_L \to \mathbb{Z}$. We assume a special subset of locations $\{\mathsf{arg}_t\}_{t \in \mathcal{T}}$ belonging to every Locs_L, which we use to pass parameters and return values for method invocations in thread t.

We assume that the execution of primitive commands and the evaluation of expressions are atomic. The semantics of a primitive command $c \in \mathsf{PComm}$ used by a

Traces of commands $(\!|C|\!)_t : (\mathcal{M} \times \mathcal{T} \to 2^{\mathsf{Traces}}) \to 2^{\mathsf{Traces}}$

$$(\!|c|\!)_t\eta = \{(t,c)\} \qquad (\!|C_1; C_2|\!)_t\eta = \{\tau_1\tau_2 \mid \tau_1 \in (\!|C_1|\!)_t\eta \wedge \tau_2 \in (\!|C_2|\!)_t\eta\}$$

$$(\!|\mathsf{if}(E)\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2|\!)_t\eta = (t, \mathsf{assume}(E))\,((\!|C_1|\!)_t\eta) \cup (t, \mathsf{assume}(!E))((\!|C_2|\!)_t\eta)$$

$$(\!|\mathsf{while}(E)\ C|\!)_t\eta = ((t, \mathsf{assume}(E))([\![C]\!]\eta))^*(t, \mathsf{assume}(!E))$$

$$(\!|m()|\!)_t\eta = \begin{cases} \{(t, \mathsf{call}!\, m(z))\,\tau\,(t, \mathsf{ret}?\, m(z')) \mid \tau \in \eta(m,t) \wedge z, z' \in \mathbb{Z}\}, & \text{if } m \in M \\ \{(t, \mathsf{call}\, m(z))\,\tau\,(t, \mathsf{ret}\, m(z')) \mid \tau \in \eta(m,t) \wedge z, z' \in \mathbb{Z}\}, & \text{otherwise} \end{cases}$$

Traces of library bodies

$$\mathcal{F} : (\mathcal{M} \times \mathcal{T} \to 2^{\mathsf{Traces}}) \to (\mathcal{M} \times \mathcal{T} \to 2^{\mathsf{Traces}})$$

$$(\mathcal{F}(\eta))(m,t) = \begin{cases} \eta(m,t) \cup ((\!|C|\!)_t\eta), & \text{if } (m \Leftarrow C) \text{ appears in } L \\ \{\varepsilon\}, & \text{if } m \in M \\ \emptyset, & \text{otherwise} \end{cases}$$

$$(\!|B|\!) : \mathcal{M} \times \mathcal{T} \to 2^{\mathsf{Traces}}$$

$$(\!|B_{\mathsf{pub}}; B_{\mathsf{pvt}}|\!) = \mathsf{lfp}(\mathcal{F})$$

Traces of libraries $(\!|L : M \to M'|\!) : 2^{\mathsf{Traces}}$

$$(\!|L|\!) = \mathsf{prefix}\left(\bigcup\nolimits_{k>0} \|_{t=1}^{k} \left(\bigcup_{\substack{z,z' \in \mathbb{Z} \\ m \in M' \setminus M}} (t, \mathsf{call}?\, m(z))\,((\!|B_{\mathsf{pub}}; B_{\mathsf{pvt}}|\!)(m,t))\,(t, \mathsf{ret}!\, m(z'))\right)^*\right)$$

Figure 5. Possible traces of a library $L = \langle \mathsf{public} : B_{\mathsf{pub}}; \mathsf{private} : B_{\mathsf{pvt}} \rangle : M \to M'$. Here $\|_{t=1}^{k} T_t$ denotes the set of all interleavings of traces from the sets $T_1, \ldots, T_k$.

$$\sigma \rightsquigarrow^{L}_{\mathsf{call}\ m(z),t} \sigma' \text{ iff } \sigma' = \sigma, \sigma(\mathsf{arg}_t) = z \qquad \sigma \rightsquigarrow^{L}_{\mathsf{ret}\ m(z),t} \sigma' \text{ iff } \sigma' = \sigma, \sigma(\mathsf{arg}_t) = z$$

$$\sigma \rightsquigarrow^{L}_{\mathsf{call}?\ m(z),t} \sigma' \text{ iff } \sigma' = \sigma[\mathsf{arg}_t \mapsto z] \qquad \sigma \rightsquigarrow^{L}_{\mathsf{ret}!\ m(z),t} \sigma' \text{ iff } \sigma' = \sigma, \sigma(\mathsf{arg}_t) = z$$

$$\sigma \rightsquigarrow^{L}_{\mathsf{call}!\ m(z),t} \sigma' \text{ iff } \sigma' = \sigma, \sigma(\mathsf{arg}_t) = z \qquad \sigma \rightsquigarrow^{L}_{\mathsf{ret}?\ m(z),t} \sigma' \text{ iff } \sigma' = \sigma[\mathsf{arg}_t \mapsto z]$$

Figure 6. Transformers for calls and returns to, from and inside a library L.

library L is defined by a family of transformers $\{\rightsquigarrow^{L}_{c,t}\}_{t \in \mathcal{T}}$, where $\rightsquigarrow^{L}_{c,t} \subseteq (\mathsf{Locs}_L \to \mathbb{Z}) \times (\mathsf{Locs}_L \to \mathbb{Z})$ describes how c affects the state of the library. The fact that the transformers are defined on locations from Locs_L formalises our assumption that L accesses only these locations. We assume that the transformers satisfy some standard properties [15], deferred to [1, §A] due to space constraints. To define the semantics of expressions, we assume that for each E the set PComm contains a special command $\mathsf{assume}(E)$, used only in defining the semantics, that allows the computation to proceed only if E is non-zero: $\sigma \rightsquigarrow^{L}_{\mathsf{assume}(E),t} \sigma'$ iff $\sigma' = \sigma$ and E is non-zero in σ.

Library Denotations. The set of traces of a library is generated in two stages. First, we generate a superset $(\!|L|\!) \subseteq 2^{\mathsf{Traces}}$ of traces produced by L, defined in Figure 5. If we think of commands as control-flow graphs, these traces contain interleavings of all possible paths through the control-flow graphs of L's methods, invoked in an arbitrary sequence. We then select those traces in $(\!|L|\!)$ that correspond to valid executions starting in a given heap using a predicate $[\![\tau]\!]_L : (\mathsf{Locs}_L \to \mathbb{Z}) \to \{\mathsf{true}, \mathsf{false}\}$. We define $[\![\cdot]\!]_L$ by generalising $\rightsquigarrow$ to calls and returns as shown in Figure 6 and letting

$$[\![\varepsilon]\!]_L\sigma = \mathsf{true}; \quad [\![(t,a)\,\tau]\!]_L\sigma = \text{if } (\exists\sigma'.\, \sigma \rightsquigarrow^{L}_{a,t} \sigma' \wedge [\![\tau]\!]_L\sigma' = \mathsf{true}) \text{ then } \mathsf{true} \text{ else } \mathsf{false}.$$

Finally, we let the set of histories $[\![L]\!]$ of a library L consist of those obtained from traces representing its valid executions from a heap with all locations set to 0:

$$[\![L]\!] = \mathsf{history}(\{\tau \in (\!|L|\!) \mid [\![\tau]\!]_L(\lambda x \in \mathsf{Locs}_L.\, 0) = \mathsf{true}\}),$$

where history projects to actions in Act.

THEOREM 1 (Correctness of flat combining) *For the libraries* FC *in Figure 1 and* $\mathsf{FC}^\sharp$ *in Figure 2 and the relation* $\mathcal{R}_\mathsf{t}$ *from §3 we have* $\mathsf{FC} \sqsubseteq_{\mathcal{R}_\mathsf{t}} \mathsf{FC}^\sharp$.

PROOF SKETCH. Consider $h \in [\![\mathsf{FC}]\!]$. In such a history, any invocation of an abstract method $(t, \mathsf{call}!\, m_i(z_i))\,(t, \mathsf{ret}?\, m_i(z_i'))$ happens within the execution of the corresponding wrapper method $(t', \mathsf{call}?\, \mathsf{do\_}m_i(z_i))\,(t', \mathsf{ret}!\, \mathsf{do\_}m_i(z_i'))$ (or just $(t', \mathsf{call}?\, \mathsf{do\_}m_i(z_i))$ if the execution of the method is uncompleted in h), though not necessarily in the same thread. This correspondence is one-to-one, as different invocations of abstract methods correspond to different requests to perform them. Furthermore, abstract methods in h are executed sequentially. We then construct a history h' by replacing every abstract method call $(t, \mathsf{call}!\, m_i(z_i))\,(t, \mathsf{ret}?\, m_i(z_i'))$ in $h|_{\mathsf{AbsAct}}$ by

$$(t', \mathsf{call}?\, \mathsf{do\_}m_i(z_i))\,(t', \mathsf{call}!\, m_i(z_i))\,(t', \mathsf{ret}?\, m_i(z_i'))\,(t', \mathsf{ret}!\, \mathsf{do\_}m_i(z_i')),$$

where t' is the thread identifier of the corresponding wrapper method invocation (similarly for uncompleted invocations). It is easy to see that $\overline{(h|_{\mathsf{AbsAct}})}\ \mathcal{R}_\mathsf{t}\ \overline{(h'|_{\mathsf{AbsAct}})}$ and $h' \in [\![\mathsf{FC}^\sharp]\!]$. Since the execution of an abstract method in h happens within the execution of the corresponding wrapper method, we also have $(h|_{\mathsf{ClAct}}) \sqsubseteq (h'|_{\mathsf{ClAct}})$. □

5 Instantiating Library Parameters and Contextuality

We now define how library parameters are instantiated and show that our notions of linearisability are preserved under such instantiations. To this end, we introduce a partial operation $\circ$ on libraries of §4: informally, for $L_1 : M \to M'$ and $L_2 : M' \to M''$ the library $L_2 \circ L_1 : M \to M''$ is obtained by instantiating abstract methods in L_2 with their implementations from L_1. Note that L_1 can itself have abstract methods M, which are left unimplemented in $L_2 \circ L_1$. Since we assume that different libraries operate in disjoint address spaces, for $\circ$ to be defined we require that the sets of locations of L_1 and L_2 be disjoint, with the exception of those used for method parameter passing. To avoid name clashes, we also require that public non-abstract methods of L_2 not be declared as abstract in L_1 (private non-abstract methods are not an issue, since we identify libraries up to their α-renaming); this also disallows recursion between L_2 and L_1.

DEFINITION 5 (Parameter library instantiation) *Consider* $L_1 : M \to M'$ *and* $L_2 : M' \to M''$ *such that* $(M'' \setminus M') \cap M = \emptyset$ *and* $\mathsf{Locs}_{L_1} \cap \mathsf{Locs}_{L_2} = \{\mathsf{arg}_t\}_{t \in \mathcal{T}}$. *Then* $L_2 \circ L_1 : M \to M''$ *is the library with* $\mathsf{Locs}_{L_2 \circ L_1} = \mathsf{Locs}_{L_1} \cup \mathsf{Locs}_{L_2}$ *obtained by erasing the declarations for methods in* M' *from* L_2*, reclassifying the methods from* $M' \setminus M''$ *in* L_1 *as private, and concatenating the method declarations of the resulting two libraries. We write* $(L_2 \circ L_1)\!\downarrow$ *when* $L_2 \circ L_1$ *is defined.*

We now show that the notions of parameterised linearisability we proposed are ***contextual***, i.e., closed under library instantiations. This property is useful in that it allows us to break the reasoning about a complex library into that about individual libraries from which it is constructed. As we show in §6, contextuality also helps us establish observational refinement.

THEOREM 2 (Contextuality of parameterised linearisability: general case) *For* $L_1, L_2 : M \to M'$ *such that* $L_1 \sqsubseteq L_2$*:*

(i) $\forall L : M'' \to M.\, (L_1 \circ L)\!\downarrow \wedge\, (L_2 \circ L)\!\downarrow \implies L_1 \circ L \sqsubseteq L_2 \circ L.$

(ii) $\forall L : M' \to M''.\, (L \circ L_1)\!\downarrow \wedge\, (L \circ L_2)\!\downarrow \implies L \circ L_1 \sqsubseteq L \circ L_2.$

THEOREM 3 (Contextuality of parameterised linearisability: encapsulated case) *For $L_1, L_2 : M \to M'$ such that $M \cap M' = \emptyset$ and $L_1 \sqsubseteq_{\mathsf{e}} L_2$:*

(i) $\forall L : M'' \to M.\, (L_1 \circ L)\!\downarrow \wedge\, (L_2 \circ L)\!\downarrow \implies L_1 \circ L \sqsubseteq_{\mathsf{e}} L_2 \circ L.$

(ii) $\forall L : M' \to M''.\, (L \circ L_1)\!\downarrow \wedge\, (L \circ L_2)\!\downarrow \implies L \circ L_1 \sqsubseteq_{\mathsf{e}} L \circ L_2.$

The restriction on method names in Definition 5 ensures that the library compositions in Theorem 3 have no public abstract methods and can thus be compared by $\sqsubseteq_{\mathsf{e}}$. Note that if L is ground, then so are $L_1 \circ L$ and $L_2 \circ L$. In this case, Theorems 2(i) and 3(i) allow us to establish classical linearisability from parameterised one.

Stating the contextuality of $\sqsubseteq_{\mathcal{R}}$ is more subtle. The relationship $L_1 \sqsubseteq_{\mathcal{R}} L_2$ allows the use of abstract methods by L_1 and L_2 to differ according to $\mathcal{R}$. As a consequence, for a non-ground parameter library L, their use by $L_1 \circ L$ and $L_2 \circ L$ may also differ according to another relation $\mathcal{G}$. We now introduce a property of L ensuring that a change in L's interactions with its client according to $\mathcal{R}$ (the *rely*) leads to a change in L's interactions with its abstract methods according to $\mathcal{G}$ (the *guarantee*).

DEFINITION 6 (Rely-guarantee closure) *Let $\mathcal{R}, \mathcal{G}$ be relations between histories of type $\emptyset \to M'$ and $\emptyset \to M$, respectively. A library $L : M \to M'$ is $\binom{\mathcal{R}}{\mathcal{G}}$**-closed** if for all $h \in [\![L]\!]$ and $h' : \emptyset \to M'$ we have*

$$(h|_{\mathsf{ClAct}})\ \mathcal{R}\ h' \implies \exists h'' \in [\![L]\!].\, (h''|_{\mathsf{ClAct}} = h') \wedge \overline{(h|_{\mathsf{AbsAct}})}\ \mathcal{G}\ \overline{(h''|_{\mathsf{AbsAct}})}.$$

Due to space constraints, we state contextuality of $\sqsubseteq_{\mathcal{R}}$ only for the case in which library parameters do not have public abstract methods. A more general statement which relaxes this assumption is given in [1, §B].

THEOREM 4 (Contextuality of linearisability up to $\mathcal{R}$) *For $L_1, L_2 : M \to M'$ such that $M \cap M' = \emptyset$ and a relation $\mathcal{R}$ such that $L_1 \sqsubseteq_{\mathcal{R}} L_2$:*

(i) $\forall L : M'' \to M.\, \forall \mathcal{G}.\, M'' \cap M = \emptyset \wedge (L \text{ is } \binom{\mathcal{R}}{\mathcal{G}}\text{-closed}) \wedge$
$(L_1 \circ L)\!\downarrow \wedge\, (L_2 \circ L)\!\downarrow \implies L_1 \circ L \sqsubseteq_{\mathcal{G}} L_2 \circ L.$

(ii) $\forall L : M' \to M''.\, (L \circ L_1)\!\downarrow \wedge\, (L \circ L_2)\!\downarrow \implies L \circ L_1 \sqsubseteq_{\mathcal{R}} L \circ L_2.$

When L in Theorem 4(i) is ground, $\mathcal{G}$ becomes irrelevant. In this case we say that L is $\mathcal{R}$***-closed*** if it is $\binom{\mathcal{R}}{\{(\varepsilon,\varepsilon)\}}$-closed. Hence, from Theorems 1 and 4(i) we get that for any $\mathcal{R}_{\mathsf{t}}$-closed (§3) library L we have $\mathsf{FC} \circ L \sqsubseteq \mathsf{FC}^{\sharp} \circ L$: instantiating flat combining with a library insensitive to thread identifiers, e.g., a sequential stack or a queue, yields a concurrent library linearisable in the classical sense.

Given two libraries $L_1 : M_1 \to M_1'$ and $L_2 : M_2 \to M_2'$ that do not interact, i.e., $(M_1 \cup M_1') \cap (M_2 \cup M_2') = \emptyset$, we may wish to compose them by merging their method declarations into a library $L_1 \uplus L_2 : M_1 \uplus M_2 \to M_1' \uplus M_2'$, as originally proposed in [8]. Our notions of linearisability are also closed under this composition.

THEOREM 5 (Compositionality of parameterised linearisability) *For $L_1, L_1' : M_1 \to M_1'$ and $L_2, L_2' : M_2 \to M_2'$ such that $(M_1 \cup M_1') \cap (M_2 \cup M_2') = \emptyset$:*

(i) $L_1 \sqsubseteq L_1' \wedge L_2 \sqsubseteq L_2' \implies L_1 \uplus L_2 \sqsubseteq L_1' \uplus L_2'.$

(ii) $L_1 \sqsubseteq_{\mathsf{e}} L_1' \wedge L_2 \sqsubseteq_{\mathsf{e}} L_2' \implies L_1 \uplus L_2 \sqsubseteq_{\mathsf{e}} L_1' \uplus L_2'.$

(iii) $\forall \mathcal{R}, \mathcal{G}.\, L_1 \sqsubseteq_{\mathcal{R}} L_1' \wedge L_2 \sqsubseteq_{\mathcal{G}} L_2' \implies L_1 \uplus L_2 \sqsubseteq_{\mathcal{R} \otimes \mathcal{G}} L_1' \uplus L_2'$, *where $\mathcal{R} \otimes \mathcal{G}$ relates histories if their projections to M_1 actions are related by $\mathcal{R}$ and the projections to M_2 actions are related by $\mathcal{G}$.*

6 Clients and Observational Refinement

A ***program*** P has the form $\mathsf{let}\ L\ \mathsf{in}\ C_1 \parallel \ldots \parallel C_n$, where $L : \emptyset \to M$ is a ground library and $C_1 \parallel \ldots \parallel C_n$ is a client such that $C_1, \ldots, C_n$ call only methods in M, written $(C_1 \parallel \ldots \parallel C_n) : M$. Using the contextuality results from §5, we now show that our notions of linearisability imply observational refinement for such programs.

The semantics of a program P is given by the set of its traces $[\![P]\!] \in 2^{\mathsf{Traces}}$, which include actions (t, c) recording the execution of primitive commands c by client threads C_t and the library L, as well as $(t, \mathsf{call}\, m(z))$ and $(t, \mathsf{ret}\, m(z))$ actions corresponding to the former invoking methods of the latter. The semantics $[\![P]\!]$ is defined similarly to that of libraries in §4. In particular, we assume that client threads C_t access only locations in a set $\mathsf{Locs}_{\mathsf{client}}$ such that $\mathsf{Locs}_{\mathsf{client}} \cap \mathsf{Locs}_L = \{\mathsf{arg}_t\}_{t \in \mathcal{T}}$ for any L. Due to space constraints, we defer the definition of $[\![P]\!]$ to [1, §A]. We define the ***observable behaviour*** $\mathsf{obs}(\tau)$ of a trace $\tau \in [\![P]\!]$ as its projection to client actions, i.e., those outside method invocations, and lift obs to sets of traces as expected.

DEFINITION 7 (Observational refinement) *For $L_1, L_2 : M \to M'$ we say that L_1 **observationally refines** L_2, written $L_1 \sqsubseteq_{\mathsf{obs}} L_2$, if for any ground library $L : \emptyset \to M$ and client $(C_1 \parallel \ldots \parallel C_n) : M'$ we have*

$$\mathsf{obs}([\![\mathsf{let}\ (L_1 \circ L)\ \mathsf{in}\ C_1 \parallel \ldots \parallel C_n]\!]) \subseteq \mathsf{obs}([\![\mathsf{let}\ (L_2 \circ L)\ \mathsf{in}\ C_1 \parallel \ldots \parallel C_n]\!]).$$

*For a binary relation $\mathcal{R}$ on histories we say that L_1 **observationally refines** L_2 **up to** $\mathcal{R}$, written $L_1 \sqsubseteq^{\mathcal{R}}_{\mathsf{obs}} L_2$, if the above is true under the assumption that L is $\mathcal{R}$-closed.*

Thus, $L_1 \sqsubseteq_{\mathsf{obs}} L_2$ means that L_1 can be replaced by L_2 in any program that uses it while keeping observable behaviours reproducible. This allows us to check a property of a program using L_1 (e.g., the flat combining implementation in Figure 1) by checking this property on a program with L_1 replaced by a possibly simpler L_2 (e.g., the flat combining specification in Figure 2). Using Theorems 2–4, we can show that our notions of linearisability validate observational refinement.

THEOREM 6 (Observational refinement) *For any libraries $L_1, L_2 : M \to M'$:*

(i) $L_1 \sqsubseteq L_2 \implies L_1 \sqsubseteq_{\mathsf{obs}} L_2$.
(ii) $M \cap M' = \emptyset \wedge L_1 \sqsubseteq_{\mathsf{e}} L_2 \implies L_1 \sqsubseteq_{\mathsf{obs}} L_2$.
(iii) $\forall \mathcal{R}.\, M \cap M' = \emptyset \wedge L_1 \sqsubseteq_{\mathcal{R}} L_2 \implies L_1 \sqsubseteq^{\mathcal{R}}_{\mathsf{obs}} L_2$.

7 Related Work

Linearisability has recently been extended to handle liveness properties, ownership transfer and weak memory models [4,5,10]. Most of these extensions have exploited the connection between linearisability and observational refinement [2]. The same methodology is adopted in the present work, but for studying two previously unexplored topics: parameterised libraries and the impact that common restrictions on their contexts have on the definition of linearisability. We believe that our results are compatible with the existing ones and can thus be extended to cover liveness and ownership transfer [4,5].

Our work shares techniques with game semantics of concurrent programming languages [12,3] and Jeffrey and Rathke's semantics of concurrent objects [11] (in particular, we use the ? and ! notation from the latter). The proofs of our contextuality theorems rely on the fact that library denotations satisfy certain closure properties related to $\sqsubseteq$, $\sqsubseteq_{\mathsf{e}}$

and $\sqsubseteq_{\mathcal{R}}$, which are similar to those exploited in these prior works. However, there are two important differences. First, prior work has not studied common restrictions on library contexts (such as the encapsulation and closure conditions in Definitions 3 and 4) and the induced stronger notions of refinement between libraries, the two key topics of this paper. Second, prior works have considered all higher-order functions, while our parameterised libraries are limited to second order. Our motivation for constraining the setting in this way is to use a simple semantics and study the key issues involved in linearisability of parameterised libraries without using sophisticated machinery from game semantics, such as justification pointers and views [9], designed for accurately modelling higher-order features. However, it is definitely a promising direction to look for appropriate notions of linearisability for full higher-order concurrent libraries by combining the ideas from this paper with those from game semantics.

Turon et al. proposed CaReSL [14], a logic that allows proving observational refinements between higher-order concurrent programs directly, without going via linearisability. Their work is complimentary to ours: it provides efficient proof techniques, whereas we identify obligations to prove, independent of a particular proof system.

Acknowledgements. We thank Thomas Dinsdale-Young and Ilya Sergey for comments that helped improve the paper. This work was supported by the EU FET project ADVENT.

References

1. Andrea Cerone, Alexey Gotsman, and Hongseok Yang. Parameterised linearisability (extended version). Available from http://software.imdea.org/∼gotsman/.
2. Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52), 2010.
3. Dan R. Ghica and Andrzej S. Murawski. Angelic semantics of fine-grained concurrency. *Ann. Pure Appl. Logic*, 151(2-3), 2008.
4. Alexey Gotsman and Hongseok Yang. Liveness-preserving atomicity abstraction. In *ICALP*, 2011.
5. Alexey Gotsman and Hongseok Yang. Linearizability with ownership transfer. In *CONCUR*, 2012.
6. Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*, 2010.
7. Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPOPP*, 2008.
8. Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3), 1990.
9. J. M. E. Hyland and C.-H. Luke Ong. On full abstraction for PCF: I, II, and III. *Inf. Comput.*, 163(2), 2000.
10. Radha Jagadeesan, Gustavo Petri, Corin Pitcher, and James Riely. Quarantining weakness - compositional reasoning under relaxed memory models. In *ESOP*, 2013.
11. Alan Jeffrey and Julian Rathke. A fully abstract may testing semantics for concurrent objects. *Theor. Comput. Sci.*, 338(1-3), 2005.
12. James Laird. A game semantics of idealized CSP. *ENTCS*, 45, 2001.
13. Claudio Russo. The Joins concurrency library. In *PADL*, 2007.
14. Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, 2013.
15. Hongseok Yang and Peter W. O'Hearn. A semantic basis for local reasoning. In *FoSSaCS*, 2002.

A Programming Language Perspective on Transactional Memory Consistency

Hagit Attiya
Technion

Alexey Gotsman
IMDEA Software Institute

Sandeep Hans
Technion

Noam Rinetzky
Tel-Aviv University

ABSTRACT

Transactional memory (TM) has been hailed as a paradigm for simplifying concurrent programming. While several consistency conditions have been suggested for TM, they fall short of formalizing the intuitive semantics of atomic blocks, the interface through which a TM is used in a programming language.

To close this gap, we formalize the intuitive expectations of a programmer as *observational refinement* between TM implementations: a concrete TM observationally refines an abstract one if every user-observable behavior of a program using the former can be reproduced if the program uses the latter. This allows the programmer to reason about the behavior of a program using the intuitive semantics formalized by the abstract TM; the observational refinement relation implies that the conclusions will carry over to the case when the program uses the concrete TM. We show that, for a particular programming language and notions of observable behavior, a variant of the well-known consistency condition of opacity is sufficient for observational refinement, and its restriction to complete histories is furthermore necessary.

Our results suggest a new approach to evaluating and comparing TM consistency conditions. They can also reduce the effort of proving that a TM implements its programming language interface correctly, by only requiring its developer to show that it satisfies the corresponding consistency condition.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.4 [**Software Engineering**]: Software/Program Verification

Keywords

Transactional memory; atomic blocks; observational refinement

1. INTRODUCTION

Transactional memory (TM) eases the task of writing concurrent applications by letting the programmer designate certain code blocks as *atomic*. TM allows designing a program and reasoning about its correctness as if each atomic block executed as a *transaction*—in one step and without interleaving with others. As an example, Figure 1 shows how atomic blocks yield simple code for pushing an element onto a stack represented as a singly-linked list: in this case it is possible to read the top-of-the-stack pointer, point the new element to it, and change the top-of-the-stack pointer, all at once. Many TM implementations have been proposed [10], using a myriad of design approaches that, for efficiency, may execute transactions concurrently, yet aim to provide the programmer with an illusion that they are executed atomically. This illusion is not always perfect—for example, as evident from Figure 1, transactions can abort due to conflicts with concurrently running ones and need to be restarted.

```
node := new(StackNode);
node.val := val;
result := abort;
while (result == abort) do {
    result := atomic {
        node.next = Top.read();
        Top = node;
    }
}
```

Figure 1: Example of transactional memory usage

How can we be sure that a TM indeed implements atomic blocks correctly? So far, researchers have tried to achieve this through a *consistency condition* that restricts the possible TM executions. Several such conditions have been proposed, including *opacity* [8, 9], *virtual world consistency* [16], *TMS* [5, 18] and *DU-opacity* [3]. Opacity is the best-known of them; roughly speaking, it requires that for any sequence of interactions between the program and the TM, dubbed a *history*, there exist another history where:

(i) the interactions of every separate thread are the same as in the original history;

(ii) the order of non-overlapping transactions present in the original history is preserved; and

(iii) each transaction executes atomically.

Unfortunately, this definition is given from the TM's point of view, as a restriction on the set of histories it can produce, and is not connected to the semantics of a programming language. The situation for other TM consistency conditions is the same and, in fact, it is not clear which of them provide the programmer with behaviors that correspond to the intuitive notion of atomic blocks, and which of them puts the minimal restrictions on TM implementations needed to achieve this.

In this paper, we aim to bridge this gap by formalizing the intuitive expectations of a programmer as *observational refinement* [13, 14] between TM implementations. Consider two TM implementations—a *concrete* one, such as an efficient TM, and an

PODC'13, July 22–24, 2013, Montréal, Québec, Canada.

abstract one, such as a TM executing every atomic block atomically. Informally, the concrete TM observationally refines the abstract one if every behavior a user can observe of a program P linked with the concrete TM can also be observed when P is linked with the abstract TM instead. This allows the programmer to reason about the behavior of P using the intuitive semantics formalized by the abstract TM; the observational refinement relation implies that the conclusions will carry over to the case when P uses the concrete TM.

We show that one TM implementation observationally refines another if they are in an *opacity relation*. The relation requires that every history of the concrete TM have a matching history of the abstract TM satisfying the conditions (i) and (ii) above. By instantiating it with an abstract TM implementation that executes transactions atomically, we obtain the existing notion of opacity. However, our definition also allows comparing two TM implementations that execute transactions concurrently, such as a more and a less optimized one. Furthemore, we show that observational refinement between two TM implementations implies the opacity relation between thir restriction to *complete* histories, i.e., ones in which every transaction either commits or aborts.

We note that the formalization of observational refinement, and thus our results, depend on the particular choices of programming language and the notion of observations. In this first treatment of this topic, we consider a basic programming language and particular forms of observations. The features of the language are as follows:

- Threads can access shared global variables outside transactions, but not inside them. However, thread-local variables (such as node in Figure 1) can be accessed in both cases.
- An aborted transaction is not restarted automatically and modifications to thread-local variables that it may have performed are not rolled back.
- A program cannot explicitly ask the TM to abort a transaction.
- Nesting of atomic blocks is not allowed.

As observable behaviors of a program, our result allows one to take either the set of its reachable states or the set of all sequences of actions performed by its finite computations. This allows a programmer to reason about safety, but not about liveness properties.

It is likely that for other programming languages or notions of observations, other consistency criteria will be necessary or sufficient for observational refinement, resulting in different trade-offs between the efficiency of TM implementations and the flexibility of their programming interfaces (see Section 8 for discussion). We hope that the link between TM consistency conditions and programming language abstractions we establish in this paper will enable TM implementors and language designers to make informed decisions about such trade-offs. Our approach can also reduce the effort of proving that a TM implements its programming interface correctly, by only requiring its developer to show that it satisfies the corresponding consistency condition.

2. PROGRAMMING LANGUAGE

We develop our results for a simple concurrent programming language with programs consisting of a fixed, but arbitrary, number m of ***threads***, identified by $\mathsf{ThreadID} = \{1, \ldots, m\}$. Every thread $t \in \mathsf{ThreadID}$ has a private set of ***local variables*** $\mathsf{LVar}_t = \{x, y, \ldots\}$, and all threads share access to a set of ***global variables*** $\mathsf{GVar} = \{g, \ldots\}$. For simplicity, we assume that all variables are of type integer. Let $\mathsf{Var} = \mathsf{GVar} \uplus \biguplus_{t=1}^{m} \mathsf{LVar}_t$ be the set of all program variables (where $\uplus$ denotes disjoint union). In addition to variables, threads can access software or hardware transactional memory, which from now on we refer to as the ***transactional system***. The system manages a fixed collection of ***transactional objects*** $\mathsf{Obj} = \{o, \ldots\}$, each having a set of ***methods*** $\mathsf{Method} = \{f, \ldots\}$ that threads can call. For simplicity, we assume that each method takes one integer parameter and returns an integer value, and that all objects have the same set of methods.

The syntax of the language is as follows:

$$\begin{array}{lcl} C & ::= & c \mid C; C \mid \texttt{if}\,(b)\,\texttt{then}\,C\,\texttt{else}\,C \mid \\ & & \texttt{while}\,(b)\,\texttt{do}\,C \mid x := \texttt{atomic}\,\{C\} \mid x := o.f(e) \\ P & ::= & C_1 \parallel \ldots \parallel C_m \end{array}$$

where b and e denote Boolean and integer expressions over local variables, left unspecified. A program P is a parallel composition of ***sequential commands*** $C_1, \ldots, C_m$, which can include ***primitive commands*** c from a set Pcomm, sequential compositions, conditionals, loops, `atomic` blocks and object method invocations.

Primitive commands are meant to execute atomically. We do not fix their set Pcomm, but assume that it at least includes assignments to local and global variables: e.g., $g := x$. We partition the set Pcomm into $2m$ classes: $\mathsf{Pcomm} = \biguplus_{t=1}^{m}(\mathsf{LPcomm}_t \uplus \mathsf{GPcomm}_t)$. The intention is that commands from LPcomm_t can access only the local variables of thread t (LVar_t); commands from GPcomm_t can additionally access global variables ($\mathsf{LVar}_t \uplus \mathsf{GVar}$). We formalize these restriction in Section 4. We forbid a thread t from accessing local variables of other threads. Thus, the thread cannot mention such variables in the conditions of `if` and `while` commands and can only use primitive commands from $\mathsf{LPcomm}_t \uplus \mathsf{GPcomm}_t$.

An ***atomic block*** $x := \texttt{atomic}\,\{C\}$ executes the command C as a ***transaction***, which the transactional system can decide to ***commit*** or ***abort***. The system's decision is returned in the local variable x, which gets assigned distinguished values committed or aborted. We forbid nested atomic blocks and, hence, nested transactions. Inside an atomic block (and only there), the program can invoke methods on transactional objects, as in $x := o.f(e)$. Here the expression e gives the value of the method parameter, and x gets assigned the return value after the method terminates. The transactional system may decide to abort a transaction initiated by $x := \texttt{atomic}\,\{C\}$ not only upon reaching the end of the `atomic` block, but also during the execution of a method on a transactional object. Once this happens, the execution of C terminates. We do not allow programs in our language to abort a transaction explicitly. A typical pattern of using the transactional system is to execute a transaction repeatedly until it commits, as shown in Figure 1.

We forbid accessing global variables inside `atomic` blocks; thus, a thread t can use primitive commands from GPcomm_t only outside them. A transaction can use local variables of the current thread; if the transaction is aborted, these variables are not rolled back to their initial values, and the values written to them by the transaction can thus be observed by the following non-transactional code. We note that, whereas transactional objects are managed by the transactional system, global variables are not. Thus, threads can communicate via the transactional system inside `atomic` blocks, and directly via global variables outside them. We also note that the transactional system is not part of a program, but is a library used by it. Hence, the state of the transactional system is separate from the variables in Var to which the program has access.

A correct transactional system implementation has to ensure that the program behaves as though `atomic` blocks indeed execute atomically, i.e., without interleaving with actions of other threads, and that operations invoked on transactional objects in aborted transactions have no effect. This does not require the implementation to execute `atomic` blocks like this internally; it only has to

provide the illusion of their atomicity to the rest of the program. In the rest of the paper, we prove that a variant of a well-known criterion of transactional system correctness, opacity, is sufficient and necessary to validate this illusion for our programming language.

3. THE OPACITY RELATION

Histories

In this section we formalize the notion of opacity in our setting, and along the way, show how to make it more flexible. To this end, we introduce the notion of a history, which records all the interactions a program in the language of Section 2 has with the transactional system in one of its executions. A ***history***, ranged over by H and S, is a finite[1] sequence of interface actions, defined as follows.

DEFINITION 1. *An **interface action** ψ is an expression of one of the following forms:*

Request actions	*Response actions*
$(t, \mathsf{txbegin})$	(t, OK)
$(t, \mathsf{txcommit})$	$(t, \mathsf{committed}) \mid (t, \mathsf{aborted})$
$(t, \mathsf{call}\ o.f(n))$	$(t, \mathsf{ret}(n')\ o.f) \mid (t, \mathsf{aborted})$

where $t \in \mathsf{ThreadID}$, $o \in \mathsf{Obj}$, $f \in \mathsf{Method}$ and $n, n' \in \mathbb{Z}$.

Interface actions denote the control flow crossing the boundary between the program and the transactional system: request actions correspond to the control being transferred from the program to the transactional system, and response actions correspond to the control being transferred the other way around. A $(t, \mathsf{txbegin})$ action denotes a thread t requesting the transactional system to start executing a transaction; this action is generated upon entering an `atomic` block. An OK action is the only possible response by the transactional system. A txcommit action is issued when a transaction tries to commit upon exiting an `atomic` block. The transactional system responds with a committed or aborted action, depending on the result. Actions call and ret denote a call to and a return from an invocation of a method on a transactional object; they are annotated with the parameter or the return value. As we noted in Section 2, the transactional system may also decide to abort a transaction while executing a method on a transactional object. In such cases, the corresponding call action is followed by an aborted action instead of a ret one.

We use the following notation: ε is the empty history; $H(i)$ is the i-th element of a history H; $H|_t$ is the projection of H onto actions of thread t; $H|_{\neg t}$ is the projection of H onto actions of threads other than t; $H|_o$ is the projection of H onto call and ret actions on object o; $|H|$ is the length of H; $H\lfloor_i$ is the prefix of H containing i actions; $H_1 H_2$ is the concatenation of H_1 and H_2. We denote by _ an expression that is irrelevant and implicitly existentially quantified.

The interactions of programs in the language of Section 2 with the transactional system are not arbitrary; they are recorded by histories that satisfy certain well-formedness properties, summarized in the following definition.

DEFINITION 2. *A history H is **well-formed** if*

- *request and response actions are properly matched: for every thread t, $H|_t$ consists of alternating request and corresponding response actions, starting from a request action;*
- *actions denoting beginning and end of transactions are properly matched: for every thread t, in the projection of $H|_t$ to* txbegin, committed *and* aborted *actions,* txbegin *alternates with* committed *or* aborted, *starting from* txbegin*; and*
- call *and* ret *actions occur only inside transactions: for every thread t, if $H|_t = H_1\ \psi\ H_2$ for a* call *or* ret *action ψ, then $H_1 = H_1'\ \psi'\ H_1''$ for some* txbegin *action ψ', and histories H_1' and H_1'' such that H_1'' does not contain* txbegin, txcommit, committed *or* aborted *actions.*

Program executions that run utill completion are described by *complete* histories, in which every transaction either aborts or commits. This class of histories is of a particular importance to us because our necessity result holds only for this class.

DEFINITION 3. *A well-formed history H is **complete** if all transactions in it have completed: if $H = H_1\ (t, \mathsf{txbegin})\ H_2$, then H_2 contains a $(t, \mathsf{committed})$ or $(t, \mathsf{aborted})$ action.*

We specify the behavior of a transactional system implementation by the set of possible interactions it can have with its clients—its ***history set*** $\mathcal{H}$, which is a prefix-closed set of well-formed histories. For our purposes, this specifies the behaviour of a transactional system completely; thus, in the following, we often conflate the notion of a transactional system and its history set.

We denote the ***complete subset*** of a history set $\mathcal{H}$ by

$$\mathcal{H}|_{\mathsf{complete}} = \{H \mid H \in \mathcal{H} \wedge (H \text{ is complete})\}.$$

The definition of the opacity relation

We define the notion of opacity in a slightly more flexible way than the original one [8, 9], inspired by the approach taken when defining the correctness of concurrent libraries via linearizability [15]. Namely, we define the correctness of a transactional system implementation by relating its history set to that of an ***abstract*** implementation, whose behavior it has to simulate; in this context, we call the original implementation ***concrete***. The abstract implementation is typically one in which `atomic` blocks actually execute atomically and methods called by aborted transactions have no effect. As we show below, we can obtain the original definition of opacity by instantiating ours with such an abstract implementation; however, our definition can also be used to compare two arbitrary implementations. To disambiguate, in the following we refer to our notion as the ***opacity relation***, instead of just opacity.

According to the following definition, a concrete transactional system $\mathcal{H}_C$ is in the opacity relation with an abstract transactional system $\mathcal{H}_A$, if every history H from $\mathcal{H}_C$ can be matched by a history S from $\mathcal{H}_A$ that "looks similar" to H from the perspective of the program. The similarity is formalized by a relation $H \sqsubseteq S$, which requires S to be a permutation of H preserving the order of actions within a thread and that of non-overlapping transactions (whether committed or aborted). Here the duration of a transaction is defined by the interval from its txbegin action to the corresponding committed or aborted action (or to the end of the history if there is none).

DEFINITION 4. *A well-formed history H is in the **opacity relation** with a well-formed history S, denoted $H \sqsubseteq S$, if there is a bijection $\theta : \{1, \ldots, |H|\} \to \{1, \ldots, |S|\}$ such that $\forall i.\, H(i) = S(\theta(i))$ and*

$$\begin{aligned}\forall i, j.\, i < j \wedge ((\exists t.\, H(i) = (t, \_) \wedge H(j) = (t, \_)) \vee \\ (H(i) \in \{(\_, \mathsf{committed}), (\_, \mathsf{aborted})\} \wedge \\ H(j) = (\_, \mathsf{txbegin}))) \\ \implies \theta(i) < \theta(j).\end{aligned}$$

[1] We do not consider infinite computations in this paper; see Section 8 for a discussion.

*A transactional system $\mathcal{H}_C$ is in the **opacity relation** with a transactional system $\mathcal{H}_A$, denoted $\mathcal{H}_C \sqsubseteq \mathcal{H}_A$, if*

$$\forall H \in \mathcal{H}_C.\, \exists S \in \mathcal{H}_A.\, H \sqsubseteq S.$$

In the following, for $i < j$ we say that the actions $H(i)$ and $H(j)$ are in the ***per-thread order*** in H when they are by the same thread and in the ***real-time order*** when $H(i) = (\_, \mathsf{committed})$ or $H(i) = (\_, \mathsf{aborted})$ and $H(j) = (\_, \mathsf{txbegin})$. Thus, $H \sqsubseteq S$ requires that the per-thread and real-time orders between actions in H be preserved in S. We now make some comments concerning the choices taken in this definition.

- As we show in Section 7, preserving the real-time order in Definition 4 is necessary to validate observational refinement due to the fact that, in our programming language, threads can access global variables outside transactions and, hence, can notice the order of non-overlapping transactions.
- Definition 4 treats committed and aborted transactions uniformly. This is a characteristic feature of opacity: it ensures that the results returned by methods of transactional objects inside aborted transactions are as consistent as those returned inside committed transactions (we return to this point in Sections 7 and 8).
- The abstract history S in Definition 4 is not required to be sequential, i.e., it may have overlapping executions of transactions. This allows the definition to compare behaviors of two realistic transactional system implementations that actually execute transactions concurrently. We also allow H to contain uncompleted transactions (without a final committed or aborted action) arising, e.g., because the corresponding thread has been preempted. In this case, we require the same behavior to be reproduced in the matching history S, which is possible because the latter does not have to be sequential [7].

Comparison with the original opacity definition

We now show how the original notion of opacity [8, 9] can be obtained from ours by instantiating Definition 4 with an abstract transactional system $\mathcal{H}_{\mathsf{atomic}}$ in which `atomic` blocks execute atomically and methods called by aborted transactions have no effect. We first introduce the ingredients needed to define this system. We start by defining a special class of *non-interleaved* histories.

DEFINITION 5. *A well-formed history H is **non-interleaved** if actions by any two transactions do not overlap: if $H = H_1\,(t, \mathsf{txbegin})\,H_2\,(t', \mathsf{txbegin})\,H_3$, where H_2 does not contain* txbegin *actions, then either H_2 contains a $(t, \mathsf{committed})$ or a $(t, \mathsf{aborted})$ action, or there are no actions by thread t in H_3.*

Note that a non-interleaved history does not have to be complete. In fact, the history set $\mathcal{H}_{\mathsf{atomic}}$ we are about to define contains only non-interleaved histories, but some of them are incomplete. This is because a concrete transactional system may produce histories with incomplete transactions, and our opacity relation requires these transactions to stay incomplete in the matching history of the abstract system. To check whether an incomplete history should be included into $\mathcal{H}_{\mathsf{atomic}}$, we first complete it with the aid of the operation defined below, which aborts every transaction that has not tried to commit yet and commits or aborts every transaction that has tried to commit (as witnessed by a txcommit action), but has not yet got a response.

DEFINITION 6. *A history H is a **completing history** for an interface action ψ, if the following holds:*

- *if $\psi = (t, \mathsf{call}\ o.f(n))$, then $H = (t, \mathsf{aborted})$;*
- *if $\psi = (t, \mathsf{txbegin})$, then $H = (t, \mathsf{OK})\,(t, \mathsf{txcommit})\,(t, \mathsf{aborted})$;*
- *if $\psi \in \{(t, \mathsf{ret}(n)\ o.f), (t, \mathsf{OK})\}$, then $H = (t, \mathsf{txcommit})\,(t, \mathsf{aborted})$;*
- *if $\psi = (t, \mathsf{txcommit})$, then $H = (t, \mathsf{committed})$ or $H = (t, \mathsf{aborted})$;*
- *otherwise, $H = \varepsilon$.*

DEFINITION 7. *A history H_c is a **non-interleaved completion** of a non-interleaved history H, if H_c is a non-interleaved complete history that can be constructed from H by adding a completing history for the last action of every thread right after this action. We denote the set of non-interleaved completions of H by $\mathsf{nicomp}(H)$.*

To define $\mathcal{H}_{\mathsf{atomic}}$, we also need to know the intended semantics of operations on transactional objects. We describe the semantics for an object $o \in \mathsf{Obj}$ by fixing all sequences of actions on o that are considered correct when executed by a sequential program. More precisely, a ***sequential specification*** of an object o is a set of histories $[\![o]\!]$ such that:

- $[\![o]\!]$ is prefix-closed;
- each $H \in [\![o]\!]$ consists of alternating call and ret actions on o, starting from a call action, where every ret is by the same thread as the preceding call; and
- $[\![o]\!]$ is insensitive to thread identifiers: for any $H \in [\![o]\!]$, changing the thread identifier in call-ret pair of adjacent actions in H yields a history in $[\![o]\!]$.

For example, $[\![o]\!]$ for a register object o would consist of histories where each read method invocation returns the value written by the latest preceding write method invocation (or the default value if there is none).

Using sequential specifications for all objects, we now define when a complete and non-interleaved history H respects the object semantics. Let $H(i)$ be a call or ret action on an object o. We say that $H(i)$ is ***legal*** in H if $H'|_o \in [\![o]\!]$, where H' is the history obtained from H by projecting $H{\downarrow_i}$ on all actions by committed transactions and the transaction containing $H(i)$. A complete and non-interleaved history H is ***legal*** if all call and ret actions in H are legal. We now let $\mathcal{H}_{\mathsf{atomic}}$ to be the set of all non-interleaved histories that can be completed to a legal history:

$$\mathcal{H}_{\mathsf{atomic}} = \{H \mid H \text{ is non-interleaved} \wedge \exists \text{ legal } H_c \in \mathsf{nicomp}(H)\}.$$

Thus, we can say that a transactional system $\mathcal{H}_C$ establishes the illusion of atomicity for transactions if $\mathcal{H}_C \sqsubseteq \mathcal{H}_{\mathsf{atomic}}$. Note that, since we require the history set of a transactional system to be prefix-closed, this criterion checks every prefix of any history produced by $\mathcal{H}_C$, just like opacity as formulated in [9]. However, in other aspects this definition is of a different form than opacity, which we can formulate in our setting as follows.

DEFINITION 8. *A history H_c is a **suffix completion** of a history H, if it is a complete history, H is a prefix of H_c, and H_c can be constructed from H by appending to it a completing history for the last action of every thread. We denote the set of suffix-completions of H by $\mathsf{scomp}(H)$.*

DEFINITION 9. *A transactional system $\mathcal{H}_C$ is **opaque** if for every history $H \in \mathcal{H}_C$, there exists a history $H_c \in \mathsf{scomp}(H)$ and a complete, non-interleaved and legal history S_c such that $H_c \sqsubseteq S_c$.*

The main difference is that Definition 9 first completes a history from $\mathcal{H}_C$ and then finds its match according to the opacity relation;

our criterion $\mathcal{H}_C \sqsubseteq \mathcal{H}_{\mathsf{atomic}}$ first finds the match and then completes the matching history. For technical reasons, Definition 9 also uses a slightly different completion, putting completing histories at the end to avoid creating new real-time orderings.

Fortunately, completion and matching commute, and thus the two formulations of opacity are equivalent.

PROPOSITION 10. *A transactional system $\mathcal{H}_C$ is opaque if and only if $\mathcal{H}_C \sqsubseteq \mathcal{H}_{\mathsf{atomic}}$.*

We prove the proposition in [2, Appendix A]. The formulation $\mathcal{H}_C \sqsubseteq \mathcal{H}_{\mathsf{atomic}}$ is more convenient for us, since the statement of observational refinement we give in Section 5 below requires us to leave the histories of the concrete transactional system intact. Stating transactional system consistency in this way also avoids the need to bake in completions (Definition 7) into the definition of the opacity relation (Definition 4): the treatment of incomplete transactions can be deferred to the choice of the abstract transactional system.

4. PROGRAMMING LANGUAGE SEMANTICS

Our goal is to establish a connection between conditions on transactional system implementations, such as the opacity relation from Section 3, and the behavior of programs that use these systems, such as those in the language of Section 2. To this end, in this section we define the *semantics* of our programming language, i.e., what kinds of computations can result when a program executes with a particular transactional system.

A program computation is captured by a ***trace*** τ, which is a finite sequence of *actions*, each describing a single computation step.

DEFINITION 11. *An **action** φ is an expression of one of the following forms: $\varphi ::= \psi \mid (t, c)$, where $t \in \mathsf{ThreadID}$ and $c \in \mathsf{Pcomm}$. We denote that set of all actions by* Action.

In addition to interface actions, we have actions of the form (t, c), which denote the execution of a primitive command by thread t. To denote the evaluation of conditions in `if` and `while` statements, we assume that the sets LPcomm_t contain special primitive commands $\mathsf{assume}(b)$, where b is a Boolean expression over local variables of thread t, defining the condition. We state their semantics formally below; informally, $\mathsf{assume}(b)$ does nothing if b holds in the current program state, and stops the computation otherwise. Thus, it allows the computation to proceed only if b holds. The assume commands are only used in defining the semantics of the programming language; hence, we forbid threads from using them directly.

We denote by $\mathsf{history}(\tau)$ the history obtained by projecting a trace τ to interface actions. We use various operations on histories defined in Section 3 for traces as well. As is the case for histories, programs in the language of Section 2 do not generate arbitrary traces, but only those satisfying certain conditions summarized in the following definition.

DEFINITION 12. *A trace τ is **well-formed** if*

- *the history* $\mathsf{history}(\tau)$ *is well-formed;*
- *thread t does not access local variables of other threads: if $\tau = \tau_1 (t, c) \tau_2$, then $c \in \mathsf{LPcomm}_t \uplus \mathsf{GPcomm}_t$; and*
- *commands in τ do not access global variables inside a transaction: if $\tau = \tau_1 (t, c) \tau_2$ for $c \in \mathsf{GPcomm}_t$, then it is not the case that $\tau_1 = \tau_1' (t, \mathsf{txbegin}) \tau_1''$, where τ_1'' does not contain* committed *or* aborted *actions.*

We denote the set of well-formed traces by WTrace.

We use two additional operations on traces. For a trace τ, we define $\mathsf{trans}(\tau)$ and $\mathsf{nontrans}(\tau)$ as the subsequence of actions in τ executed inside transactions (including txbegin, committed and aborted actions), respectively, outside them (excluding txbegin, committed and aborted actions). Formally, we include an action $\varphi = (t, \_)$ such that $\tau|_t = \tau_1 \varphi \tau_2$ into $\mathsf{trans}(\tau)$ if:

- φ is a txbegin, committed or aborted action; or
- $\tau_1 = \tau_1' (t, \mathsf{txbegin}) \tau_1''$, where τ_1'' does not contain committed or aborted actions.

All other actions form $\mathsf{nontrans}(\tau)$. Actions in τ that are in $\mathsf{trans}(\tau)$ are ***transactional*** and all others are ***non-transactional***.

A ***state*** of a program records the values of all its variables: $s \in \mathsf{State} = \mathsf{Var} \to \mathbb{Z}$. The semantics of a program $P = C_1 \parallel \cdots \parallel C_m$ is given by the set of traces $[\![P]\!](s, \mathcal{H}) \in \mathcal{P}(\mathsf{WTrace})$ it produces when run with a transactional system $\mathcal{H}$ from an initial state s. We define this set in two stages. First, we define the set $[\![P]\!](s) \in \mathcal{P}(\mathsf{WTrace})$ that a program produces when run from s with the behaviors of the transactional system unrestricted. We then compute the set of traces produced by P when run with a given transactional system $\mathcal{H}$ by selecting those traces that interact with the transactional system in a way consistent with $\mathcal{H}$:

$$[\![P]\!](s, \mathcal{H}) = \{\tau \mid \tau \in [\![P]\!](s) \wedge \mathsf{history}(\tau) \in \mathcal{H}\}. \quad (1)$$

The set $[\![P]\!](s)$ is itself computed in two stages[2]. First, we compute a trace set $A(P) \in \mathcal{P}(\mathsf{WTrace})$ that resolves all issues regarding sequential control flow and interleaving. Intuitively, if one thinks of each sequential command C_t in P as a control-flow graph, then $A(P)$ contains all possible interleavings of paths in the control-flow graph of all the commands C_t starting from their source nodes. The set $A(P)$ is a superset of all the traces that can actually be executed: e.g., if a thread executes the command

$$x := 1;\ \texttt{if}\ (x = 1)\ y := 1\ \texttt{else}\ y := 2 \quad (2)$$

where x is a local variable, then $A(P)$ will contain a trace where $y := 2$ is executed instead of $y := 1$. To filter out such nonsensical traces, we *evaluate* every trace to determine whether its control flow is consistent with the expected behavior of its actions. This is formalized by a function $\mathsf{eval} : \mathsf{State} \times \mathsf{WTrace} \to \mathcal{P}(\mathsf{State})$ that, given an initial state and a trace, produces the set of states resulting from executing the actions in the trace, or an empty set if the trace is infeasible. Then we let

$$[\![P]\!](s) = \{\tau \mid \tau \in A(P) \wedge \mathsf{eval}(s, \tau) \neq \emptyset\}. \quad (3)$$

The rest of this section defines the trace set $A(P)$ and the evaluation function eval formally. The definitions follow the intuitive semantics of our programming language and can be skipped on first reading (they are only used in the proofs of Lemmas 18 and 20 in Section 6 and in the detailed proof of Theorem 24 in [2, Appendix B]).

Trace set $A(P)$

The function $A(\cdot)$ in Figure 2 maps commands and programs to the set of their possible traces. $A(C)t$ gives the set of traces produced by a command C when it is executed by thread t. To define $A(P)$, we first compute the set of all the interleavings of traces produced by the threads constituting P. Formally, $\tau \in \mathsf{interleave}(\tau_1, \ldots, \tau_m)$ if and only if every action in τ is performed by some thread $t \in \{1, \ldots, m\}$, and $\tau|_t = \tau_t$ for every thread $t \in \{1, \ldots, m\}$. We then let $A(P)$ be the set of all prefixes of the resulting traces, as denoted by the prefix operator. We take prefix

[2]Here we define the set $[\![P]\!](s)$ in a denotational style; a definition using structural operational semantics would also be appropriate.

$$
\begin{array}{rcl}
A(c)t & = & \{(t,c)\} \\
A(C_1;C_2)t & = & \{\tau_1\,\tau_2 \mid \tau_1 \in A(C_1)t \wedge \tau_2 \in A(C_2)t\} \\
A(\texttt{if}\ (b)\ \texttt{then}\ C_1\ \texttt{else}\ C_2)t & = & \{(t,\mathsf{assume}(b))\,\tau_1 \mid \tau_1 \in A(C_1)t\} \cup \{(t,\mathsf{assume}(\neg b))\,\tau_2 \mid \tau_2 \in A(C_2)t\} \\
A(\texttt{while}\ (b)\ \texttt{do}\ C)t & = & \{((t,\mathsf{assume}(b))\,(A(C)t))^*\,(t,\mathsf{assume}(\neg b))\} \\
A(x := o.f(e))t & = & \{(t,\mathsf{assume}(e=n))\,(t,\mathsf{call}\ o.f(n))\,(t,\mathsf{ret}(n')\ o.f)\,(t,x:=n') \mid n,n' \in \mathbb{Z}\} \cup \\
 & & \{(t,\mathsf{assume}(e=n))\,(t,\mathsf{call}\ o.f(n))\,(t,\mathsf{aborted}) \mid n \in \mathbb{Z}\} \\
A(x := \texttt{atomic}\ \{C\})t & = & \{(t,\mathsf{txbegin})\,(t,\mathsf{OK})\,\tau\,(t,\mathsf{aborted})\,(t,x:=\mathsf{aborted}) \mid \tau\,(t,\mathsf{aborted}) \in A(C)t\} \cup \\
 & & \{(t,\mathsf{txbegin})\,(t,\mathsf{OK})\,\tau\,(t,\mathsf{txcommit})\,(t,\mathsf{committed})\,(t,x:=\mathsf{committed}) \mid \tau \in A(C)t \wedge \tau \neq \_\,(t,\mathsf{aborted})\} \cup \\
 & & \{(t,\mathsf{txbegin})\,(t,\mathsf{OK})\,\tau\,(t,\mathsf{txcommit})\,(t,\mathsf{aborted})\,(t,x:=\mathsf{aborted}) \mid \tau \in A(C)t \wedge \tau \neq \_\,(t,\mathsf{aborted})\} \\
A(C_1 \parallel \ldots \parallel C_m) & = & \mathsf{prefix}(\bigcup\{\mathsf{interleave}(\tau_1,\ldots,\tau_m) \mid \forall t.\, 1 \le t \le m \implies \tau_t \in A(C_t)t\})
\end{array}
$$

Figure 2: The function $A(\cdot)$ mapping commands and programs to the set of all their possible traces

closure here to account for incomplete program computations as well as those in which the scheduler preempts a thread forever.

$A(c)t$ returns a singleton set with the action corresponding to the primitive command c (recall that primitive commands execute atomically). $A(C_1;C_2)t$ concatenates all possible traces corresponding to C_1 with those corresponding to C_2. The set of traces for a conditional considers cases where either branch is taken. We record the decision using an assume action; at the evaluation stage, this allows us to ensure that this decision is consistent with the program state. The trace set for a loop is defined using the Kleene closure operator * to produce all possible unfoldings of the loop body. Again, we record branching decisions using assume actions.

The trace set of a method invocation $x := f(e)$ includes both traces where the method executes successfully and where the current transaction is aborted. The former set is constructed by nondeterministically choosing two integers n and n' to describe the parameter n and the return value n' for the method call. To ensure that e indeed evaluates to n, we insert $\mathsf{assume}(e = n)$ before the call action, and to ensure that x gets the return value n', we add the assignment $x := n'$ after the ret action. Note that some of the choices here might not be feasible: the chosen n might not be the value of the parameter expression e when the method is invoked, or the method might never return n' when called with n. Such infeasible choices are filtered out at the following stages of the semantics definition: the former at the evaluation stage (3) by the semantics of assume, and the latter in (1) by selecting the traces from $[\![P]\!](s)$ that interact with the transactional system correctly.

The trace set of $x := \texttt{atomic}\ \{C\}$ contains traces in which C is aborted in the middle of its execution (at an object operation) and those in which C executes until completion and then the transaction commits or aborts. From the restrictions on programs introduced in Section 2, we immediately get:

PROPOSITION 13. *For any program P, the set $A(P)$ contains only well-formed traces.*

Semantics of primitive commands

To define evaluation, we assume a semantics of every command $c \in \mathsf{Pcomm}$, given by a function $[\![c]\!]$ that defines how the program state is transformed by executing c. As we noted in Section 2, different classes of primitive commands are supposed to access only certain subsets of variables. To ensure that this is indeed the case, we define $[\![c]\!]$ as a function of only those variables that c is allowed to access. Namely, the semantics of $c \in \mathsf{LPcomm}_t$ is given by

$$[\![c]\!] : (\mathsf{LVar}_t \to \mathbb{Z}) \to \mathcal{P}(\mathsf{LVar}_t \to \mathbb{Z}),$$

and the semantics of $c \in \mathsf{GPcomm}_t$, by

$$[\![c]\!] : ((\mathsf{LVar}_t \uplus \mathsf{GVar}) \to \mathbb{Z}) \to \mathcal{P}((\mathsf{LVar}_t \uplus \mathsf{GVar}) \to \mathbb{Z}).$$

For a valuation q of variables that c is allowed to access, $[\![c]\!](q)$ yields the set of their valuations that can be obtained by executing c from a state with variable values q. Note that this allows c to be non-deterministic. For example, an assignment command $x := g$ has the following semantics:

$$[\![x := g]\!](q) = q[x \mapsto q(g)],$$

where $q[x \mapsto q(g)]$ is the function that has the same value as q everywhere, except for x, where it has the value $q(g)$. We define the semantics of assume commands following the informal explanation given at the beginning of this section: for example,

$$[\![\mathsf{assume}(x = n)]\!](q) = \begin{cases} \{q\}, & \text{if } q(x) = n; \\ \emptyset, & \text{otherwise.} \end{cases} \tag{4}$$

Thus, when the condition in assume does not hold of q, the command stops the computation by not producing any output states.

We lift functions $[\![c]\!]$ to full states by keeping the variables that c is not allowed to access unmodified. For example, if $c \in \mathsf{GPcomm}_t$, then for all $u \in \mathsf{Var}$ we let

$$[\![c]\!](s)(u) = \begin{cases} [\![c]\!](s|_{\mathsf{LVar}_t \uplus \mathsf{GVar}})(u), & \text{if } u \in \mathsf{LVar}_t \uplus \mathsf{GVar}; \\ s(u), & \text{otherwise.} \end{cases}$$

where $s|_{\mathsf{LVar}_t \uplus \mathsf{GVar}}$ is the restriction of s to variables in $\mathsf{LVar}_t \uplus \mathsf{GVar}$.

Trace evaluation

Using the semantics of primitive commands, we first define the evaluation of a single action on a given state:

$$
\begin{array}{rcl}
\multicolumn{3}{c}{\mathsf{eval} : \mathsf{State} \times \mathsf{Action} \to \mathcal{P}(\mathsf{State})} \\
\mathsf{eval}(s,(t,c)) & = & [\![c]\!](s); \\
\mathsf{eval}(s,\varphi) & = & \{s\} \text{ for all other actions } \varphi.
\end{array}
$$

Note that this does not change the state s as a result of interface actions, since their return values are assigned to local variables by separate actions introduced when generating $A(P)$. We then lift eval to traces as follows:

$$
\begin{array}{rcl}
\multicolumn{3}{c}{\mathsf{eval} : \mathsf{State} \times \mathsf{WTrace} \to \mathcal{P}(\mathsf{State})} \\
\mathsf{eval}(s,\varepsilon) & = & \{s\}; \\
\mathsf{eval}(s,\tau\varphi) & = & \{s'' \mid \exists s'.\, s' \in \mathsf{eval}(s,\tau) \wedge s'' \in \mathsf{eval}(s',\varphi)\}.
\end{array}
$$

This allows us to define $[\![P]\!](s)$ as the set of those traces from $A(P)$ that can be evaluated from s without getting stuck, as formalized

by (3). Note that this definition enables the semantics of assume defined by (4) to filter out traces that make branching decisions inconsistent with the program state. For example, consider the program (2). The set $A(P)$ includes traces where both branches are explored. However, due to the semantics of the assume actions added to the traces according to Figure 2, only the trace executing $y := 1$ will result in a non-empty set of final states after the evaluation and, therefore, only this trace will be included into $[\![P]\!](s)$.

5. OBSERVATIONAL REFINEMENT

Informally, a concrete transactional system $\mathcal{H}_C$ *observationally refines* an abstract transactional system $\mathcal{H}_A$, if replacing $\mathcal{H}_C$ by $\mathcal{H}_A$ in a program leaves all its original user-observable behaviors reproducible. The formal definition depends on which aspects of program behavior we consider observable. One possibility is to allow the user to observe the values of all variables during the program execution. The corresponding notion of observational refinement is stated as follows.

DEFINITION 14. *A transactional system $\mathcal{H}_C$ **observationally refines** a transactional system $\mathcal{H}_A$ **with respect to states**, denoted by $\mathcal{H}_C \preceq_{\mathsf{state}} \mathcal{H}_A$, if*

$$\forall P.\, \forall s.\, \forall \tau \in [\![P]\!](s, \mathcal{H}_C).\, \exists \tau' \in [\![P]\!](s, \mathcal{H}_A).\ \mathsf{eval}(s, \tau) = \mathsf{eval}(s, \tau').$$

Note that, since the semantics of a program P includes traces corresponding to its incomplete computations (see the use of prefix-closure in Figure 2), we allow observing intermediate program states as well as final ones. Definition 14 implies that, if a program using the abstract transactional system $\mathcal{H}_A$ satisfies a correctness property stated in terms of such states, then it will still satisfy the property if it uses the concrete system $\mathcal{H}_C$ instead.

In some situations, we may also be interested in observing separate program actions and the order between them, rather than the set of all reachable program states. In particular, this is desirable for checking the validity of linear-time temporal properties over program traces. We formulate the corresponding notion of observational refinement as follows.

DEFINITION 15. *Well-formed traces τ and τ' are **observationally equivalent**, written $\tau \sim \tau'$, if*

$$(\forall t \in \mathsf{ThreadID}.\, \tau|_t = \tau'|_t) \wedge (\mathsf{nontrans}(\tau) = \mathsf{nontrans}(\tau')).$$

*A transactional system $\mathcal{H}_C$ **observationally refines** a transactional system $\mathcal{H}_A$ **with respect to traces**, denoted by $\mathcal{H}_C \preceq_{\mathsf{trace}} \mathcal{H}_A$, if*

$$\forall P.\, \forall s.\, \forall \tau \in [\![P]\!](s, \mathcal{H}_C).\, \exists \tau' \in [\![P]\!](s, \mathcal{H}_A).\, \tau \sim \tau'.$$

Traces related by $\sim$ are thus considered indistinguishable to the user, and, hence, the user can observe which equivalence class over $\sim$ the trace executed by the program belongs to.

We are now in a position to state our main result.

THEOREM 16.

$$\mathcal{H}_C \sqsubseteq \mathcal{H}_A \implies \mathcal{H}_C \preceq_{\mathsf{trace}} \mathcal{H}_A \implies \mathcal{H}_C \preceq_{\mathsf{state}} \mathcal{H}_A \implies \mathcal{H}_C|_{\mathsf{complete}} \sqsubseteq \mathcal{H}_A|_{\mathsf{complete}}.$$

Thus, for our programming language, the opacity relation implies observational refinement with respect to traces or states, and either of these implies that the complete subsets of the transactional systems are in the opacity relation.

The first and the second implications from Theorem 16 are proved in Section 6, and the third one, in Section 7.

6. SUFFICIENCY OF THE OPACITY RELATION

Our goal in this section is to show that the opacity relation implies observational refinement with respect to traces. The next lemma (proved below) is key in establishing this: it shows that a trace τ_H with a history H can be transformed into an equivalent trace τ_S with a history S that is in the opacity relation with H.

LEMMA 17 (REARRANGEMENT). *For any well-formed histories H and S:*

$$H \sqsubseteq S \implies (\forall \textit{well-formed } \tau_H.\, \mathsf{history}(\tau_H) = H \implies \exists \textit{well-formed } \tau_S.\, \mathsf{history}(\tau_S) = S \wedge \tau_H \sim \tau_S).$$

We also rely on the following lemma, which straightforwardly implies that observational refinement with respect to traces implies that with respect to states. The proof of the lemma (given below) relies on the restrictions on accesses to variables in Definition 12.

LEMMA 18. *For all well-formed traces τ_H and τ_S,*

$$\tau_H \sim \tau_S \wedge \mathsf{eval}(s, \tau_H) \neq \emptyset \implies \forall s.\, \mathsf{eval}(s, \tau_H) = \mathsf{eval}(s, \tau_S).$$

COROLLARY 19. $\mathcal{H}_C \preceq_{\mathsf{trace}} \mathcal{H}_A \implies \mathcal{H}_C \preceq_{\mathsf{state}} \mathcal{H}_A$.

Lemma 18 also allows us to conclude that the trace τ_S resulting from the transformation in Lemma 17 can be produced by a program P if so can the original trace τ_H.

LEMMA 20. *If $\tau_H \in [\![P]\!](s)$ and $\tau_H \sim \tau_S$, then $\tau_S \in [\![P]\!](s)$.*

PROOF. Let $P = C_1 \parallel \ldots \parallel C_m$. Consider s, τ_H and τ_S such that $\tau_H \in [\![P]\!](s)$ and $\tau_H \sim \tau_S$. Then for some τ' we have $\tau_H\, \tau' \in A(P)$ and $\mathsf{eval}(s, \tau_H) \neq \emptyset$. This implies $(\tau_H\, \tau')|_t \in A(C_t)t$ for any thread t. Since $\tau_H \sim \tau_S$, we have that, $\tau_S|_t = \tau_H|_t$. Then $(\tau_S\, \tau')|_t \in A(C_t)t$ and by the definition of $A(P)$ in Figure 2 we get $\tau_S \in A(P)$. Furthermore, by Lemma 18, $\mathsf{eval}(s, \tau_S) \neq \emptyset$, so that $\tau_S \in [\![P]\!](s)$. □

THEOREM 21. $\mathcal{H}_C \sqsubseteq \mathcal{H}_A \implies \mathcal{H}_C \preceq_{\mathsf{trace}} \mathcal{H}_A$.

PROOF. Assume $\mathcal{H}_C \sqsubseteq \mathcal{H}_A$ and consider a program P, a state s and a trace $\tau_H \in [\![P]\!](s, \mathcal{H}_C)$. Let $\mathsf{history}(\tau_H) = H$; then $\tau_H \in [\![P]\!](s)$ and $H \in \mathcal{H}_C$. Since $\mathcal{H}_C \sqsubseteq \mathcal{H}_A$, there exists a history $S \in \mathcal{H}_A$ such that $H \sqsubseteq S$. Since H and S are well-formed, Lemma 17 implies that there is a well-formed trace τ_S such that $\tau_H \sim \tau_S$ and $\mathsf{history}(\tau_S) = S$. Then by Lemma 20 we have $\tau_S \in [\![P]\!](s)$. Since $S \in \mathcal{H}_A$, this implies $\tau_S \in [\![P]\!](s, \mathcal{H}_A)$. □

Proof of Lemma 17 (Rearrangement)

Consider H, S and τ_H such that $H \sqsubseteq S$ and $\mathsf{history}(\tau_H) = H$. Note that $|H| = |S|$. To obtain the desired trace τ_S, we inductively construct a sequence of well-formed traces τ^i, $i = 0..|S|$ with well-formed histories $H^i = \mathsf{history}(\tau^i)$ such that

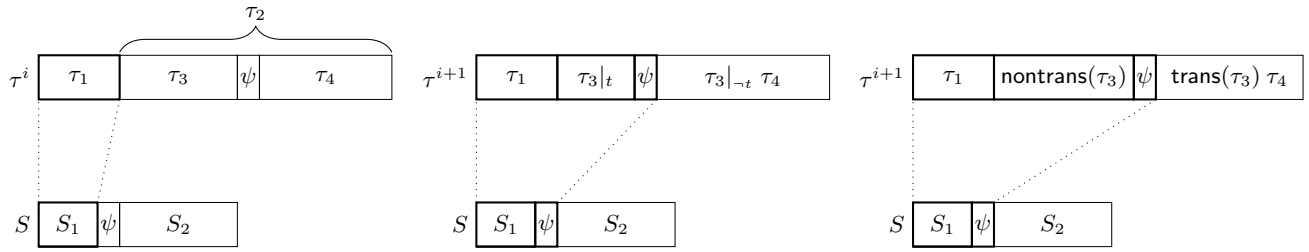
$$H^i\downarrow_i = S\downarrow_i; \quad H^i \sqsubseteq S; \quad \tau_H \sim \tau^i. \tag{5}$$

We then let $\tau_S = \tau^{|S|}$, so that $\tau_H \sim \tau^{|S|}$ and

$$\mathsf{history}(\tau^{|S|}) = H^{|S|} = H^{|S|}\downarrow_{|S|} = S\downarrow_{|S|} = S,$$

as required. Note that the condition $H^i \sqsubseteq S$ in (5) is not used to establish the required properties of τ_S; we add it so that the induction goes through.

We start the construction of the sequence of traces τ^i with $\tau^0 = \tau_H$, so that $H^0 = H$ and all the requirements in (5) hold trivially. Assume a trace τ^i satisfying (5) was constructed; we get τ^{i+1} from τ^i by applying the following lemma. Since $\sim$ is transitive, the conclusion of the lemma implies the desired properties of τ^{i+1}.



(a) Trace τ^i and history S (b) Case of $\psi \neq (t, \mathsf{txbegin})$ (c) Case of $\psi = (t, \mathsf{txbegin})$

Figure 3: An illustration of the transformations performed in the proof of Lemma 22

LEMMA 22. *Assume well-formed histories H^i and S and a well-formed trace τ^i such that*

$$\mathsf{history}(\tau^i) = H^i; \quad H^i\downarrow_i = S\downarrow_i; \quad H^i \sqsubseteq S.$$

Then there exist some well-formed history H^{i+1} and a well-formed trace τ^{i+1} such that

$$\mathsf{history}(\tau^{i+1}) = H^{i+1}; \quad H^{i+1}\downarrow_{i+1} = S\downarrow_{i+1};$$
$$H^{i+1} \sqsubseteq S; \quad \tau^i \sim \tau^{i+1}.$$

PROOF. Let $S = S_1\,\psi\,S_2$, where $|S_1| = i$. By assumption, $\mathsf{history}(\tau^i)\downarrow_i = H^i\downarrow_i = S\downarrow_i = S_1$. Thus, for some traces τ_1 and τ_2, we have $\tau^i = \tau_1\,\tau_2$, where τ_1 is the minimal prefix of τ^i such that $\mathsf{history}(\tau_1) = S_1$. We also have $H^i \sqsubseteq S$, and, hence, there exists a bijection $\theta : \{1, \ldots, |H^i|\} \to \{1, \ldots, |S|\}$ satisfying the conditions of Definition 4. Since θ preserves the per-thread order of actions and $\mathsf{history}(\tau_1) = S_1$, we can assume that θ maps actions in $\mathsf{history}(\tau_1)$ to the corresponding actions in S_1. Hence, for some traces τ_3 and τ_4, we have

$$\mathsf{history}(\psi) = \psi, \qquad \tau_2 = \tau_3\,\psi\,\tau_4, \qquad \tau^i = \tau_1\,\tau_2 = \tau_1\,\tau_3\,\psi\,\tau_4,$$

and θ maps the ψ in $\mathsf{history}(\tau^i)$ to ψ in S, i.e., $\theta(|\mathsf{history}(\tau_1\,\tau_3)| + 1) = |S_1| + 1$; see Figure 3(a).

Let $\psi = (t, \_)$. We note that, since θ preserves the per-thread order of actions and $\mathsf{history}(\tau_1) = S_1$, we have $\mathsf{history}(\tau_3|_t) = \varepsilon$. We proceed by case analysis on the kind of ψ.

Case I: $\psi \neq (t, \mathsf{txbegin})$. Let $\tau^{i+1} = \tau_1\,(\tau_3|_t)\,\psi\,(\tau_3|_{\neg t})\,\tau_4$ and $H^{i+1} = \mathsf{history}(\tau^{i+1})$; see Figure 3(b). Intuitively, τ^{i+1} is obtained from $\tau^i = \tau_1\,\tau_3\,\psi\,\tau_4$ by moving all the actions in τ_3 performed by thread t, together with ψ, to the position right after τ_1.

Since $\mathsf{history}(\tau_3|_t) = \varepsilon$, $\mathsf{history}(\tau_1) = S_1$ and $|S_1| = i$, we get:

$$\begin{array}{rcl} H^{i+1}\downarrow_{i+1} & = & (\mathsf{history}(\tau_1\,(\tau_3|_t)\,\psi\,(\tau_3|_{\neg t})\,\tau_4))\downarrow_{i+1} \\ & = & S_1\,\psi = S\downarrow_{i+1}, \end{array}$$

as required. We also have:

$$\begin{array}{rcl} \tau^{i+1}|_t & = & (\tau_1\,(\tau_3|_t)\,\psi\,(\tau_3|_{\neg t})\,\tau_4)|_t \\ & = & (\tau_1|_t)\,(\tau_3|_t)\,\psi\,(\tau_4|_t) \\ & = & (\tau_1\,\tau_3\,\psi\,\tau_4)|_t \\ & = & \tau^i|_t; \end{array}$$

$$\begin{array}{rcl} \tau^{i+1}|_{\neg t} & = & (\tau_1\,(\tau_3|_t)\,\psi\,(\tau_3|_{\neg t})\,\tau_4)|_{\neg t} \\ & = & (\tau_1|_{\neg t})\,(\tau_3|_{\neg t})\,(\tau_4|_{\neg t}) \\ & = & (\tau_1\,\tau_3\,\psi\,\tau_4)|_{\neg t} \\ & = & \tau^i|_{\neg t}. \end{array}$$

Hence, for any thread t', we have $\tau^i|_{t'} = \tau^{i+1}|_{t'}$ and $H^{i+1}|_{t'} = H^i|_{t'} = S|_{t'}$. Any real-time order between two actions that exists in H^{i+1}, but not in H^i, also exists between the actions in S corresponding to them according to θ. Thus, $H^{i+1} \sqsubseteq S$.

Since $\psi \neq (t, \mathsf{txbegin})$ and $\mathsf{history}(\tau_3|_t) = \varepsilon$, all the actions performed by t in the subtrace τ_3 of τ^i are transactional. Hence,

$$\begin{array}{rcl} \mathsf{nontrans}(\tau^{i+1}) & = & \mathsf{nontrans}(\tau_1\,(\tau_3|_t)\,\psi\,(\tau_3|_{\neg t})\,\tau_4) \\ & = & \mathsf{nontrans}(\tau_1\,\tau_3\,\psi\,\tau_4) \\ & = & \mathsf{nontrans}(\tau^i) \end{array}$$

and therefore $\tau^i \sim \tau^{i+1}$.

Case II: $\psi = (t, \mathsf{txbegin})$. Note that the subtrace τ_3 of τ^i does not contain any committed or aborted actions ψ'. Indeed, assume otherwise. Since $\mathsf{history}(\tau_1) = S_1$, the action in S corresponding to ψ' according to θ would be in S_2. This would mean that the real-time order between ψ' and ψ in H^i is not preserved in S, contradicting our assumption that $H^i \sqsubseteq S$. Thus, for any thread $t' \neq t$, $\tau_3|_{t'}$ consists of some number of non-transactional actions followed by some number of transactional ones, and $\tau_3|_t$ does not contain any transactional actions. Motivated by these observations, we let

$$\tau^{i+1} = \tau_1\,\mathsf{nontrans}(\tau_3)\,\psi\,\mathsf{trans}(\tau_3)\,\tau_4$$

and $H^{i+1} = \mathsf{history}(\tau^{i+1})$; see Figure 3(c).[3] Intuitively, τ^{i+1} is obtained from $\tau^i = \tau_1\,\tau_3\,\psi\,\tau_4$ by moving all transactional actions in τ_3 to the position right before τ_4.

Since $\mathsf{history}(\mathsf{nontrans}(\tau_3)) = \varepsilon$, $\mathsf{history}(\tau_1) = S_1$ and $|S_1| = i$, we get:

$$\begin{array}{rcl} H^{i+1}\downarrow_{i+1} & = & (\mathsf{history}(\tau_1\,\mathsf{nontrans}(\tau_3)\,\psi\,\mathsf{trans}(\tau_3)\,\tau_4)\downarrow_{i+1} \\ & = & S_1\,\psi = S\downarrow_{i+1}, \end{array}$$

as required.

Since for every thread $t' \neq t$, $\tau_3|_{t'}$ consists of non-transactional actions followed by transactional ones,

$$(\mathsf{nontrans}(\tau_3)\,\psi\,\mathsf{trans}(\tau_3))|_{t'} = (\tau_3\,\psi)|_{t'}.$$

Since $\tau_3|_t$ does not contain any transactional actions, $\tau_3 = \mathsf{nontrans}(\tau_3)$ and, hence,

$$(\mathsf{nontrans}(\tau_3)\,\psi\,\mathsf{trans}(\tau_3))|_t = (\tau_3\,\psi)|_t.$$

Thus, for any t'' we have

$$\begin{array}{rcl} \tau^{i+1}|_{t''} & = & (\tau_1\,\mathsf{nontrans}(\tau_3)\,\psi\,\mathsf{trans}(\tau_3)\,\tau_4)|_{t''} \\ & = & (\tau_1\,\tau_3\,\psi\,\tau_4)|_{t''} = \tau^i|_{t''} \end{array}$$

[3] Here we are applying nontrans to a part τ_3 of a well-formed trace τ^i. Note that, in the absence of txbegin actions in τ_3, the definition of nontrans given in Section 4 considers all actions in τ_3 non-transactional.

and $H^{i+1}|_{t''} = H^i|_{t''} = S|_{t''}$. Any real-time order between actions in H^{i+1} also exists in H^i, and hence, $H^{i+1} \sqsubseteq S$. Finally,

$$\begin{array}{rcl} \mathsf{nontrans}(\tau^{i+1}) & = & \mathsf{nontrans}(\tau_1\, \mathsf{nontrans}(\tau_3)\, \psi\, \mathsf{trans}(\tau_3)\, \tau_4) \\ & = & \mathsf{nontrans}(\tau_1\, \tau_3\, \psi\, \tau_4) \\ & = & \mathsf{nontrans}(\tau^i), \end{array}$$

so that $\tau^i \sim \tau^{i+1}$. $\square$

Proof of Lemma 18

Consider s, τ_H and τ_S such that $\tau_H \sim \tau_S$ and $\mathsf{eval}(s, \tau_H) \neq \emptyset$. We need to show that $\mathsf{eval}(s, \tau_S) \neq \emptyset$. The proof of this fact is similar in its structure to that of Lemma 17. We inductively construct a sequence of well-formed traces τ^i, $i = 0..|\tau_S|$ such that

$$\tau^i\downarrow_i = \tau_S\downarrow_i; \quad \tau^i \sim \tau_S; \quad \mathsf{eval}(s, \tau^i) = \mathsf{eval}(s, \tau_H) \neq \emptyset. \quad (6)$$

Then for $i = |\tau_S|$ we get $\tau^i = \tau_S$, which implies the required.

To construct the sequence of traces τ^i, we let $\tau^0 = \tau_H$, so that all the requirements in (6) hold trivially. Assume now that a trace τ^i satisfying (6) has been constructed. Let $\tau_S = \tau_1\, \varphi\, \tau_2$, where $|\tau_1| = i$, and $\varphi = (t, \_)$. By assumption, $\tau^i\downarrow_i = \tau_S\downarrow_i$ and, since $\tau^i \sim \tau_S$, we also have $\tau^i|_t = \tau_S|_t$. Hence, for some traces τ_2' and τ_2'', we get

$$\tau^i = \tau_1\, \tau_2'\, \varphi\, \tau_2'',$$

where τ_2' does not contain any actions by thread t. Let

$$\tau^{i+1} = \tau_1\, \varphi\, \tau_2'\, \tau_2'';$$

then $\tau^{i+1}\downarrow_{i+1} = \tau_S\downarrow_{i+1}$. We now show that $\tau^{i+1} \sim \tau^i$ and $\mathsf{eval}(s, \tau^{i+1}) = \mathsf{eval}(s, \tau^i)$, which implies the required.

First note that $\tau^i|_{t'} = \tau^{i+1}|_{t'}$ for any thread t' since τ_2' does not contain any actions by thread t. Next, consider the case when the action φ in τ^i is non-transactional. Since $\tau^i|_t = \tau_S|_t$, so is the corresponding action φ in τ_S. Assume that some action $\varphi' = (t', \_)$ in the subtrace τ_2' of τ^i is non-transactional as well, where $t' \neq t$. Let φ' be the j-th action by thread t' in τ^i. Since $\tau^i|_{t'} = \tau_S|_{t'}$, φ' is also the j-th action by thread t' in τ_S and is non-transactional in this trace. But then φ' has to be in the subtrace τ_2 of τ_S and thus follow φ, contradicting $\mathsf{nontrans}(\tau^i) = \mathsf{nontrans}(\tau_S)$. Hence, if the action φ in τ^i is non-transactional, then the subtrace τ_2' of τ^i does not contain any non-transactional actions. This implies $\mathsf{nontrans}(\tau^{i+1}) = \mathsf{nontrans}(\tau^i)$ and thus $\tau^{i+1} \sim \tau^i$.

The restrictions on accesses to variables by commands from LPcomm_t and GPcomm_t (stated in Section 2 and formalized in Section 4) imply

PROPOSITION 23. *Assume φ_1 and φ_2 are actions by different threads and, if $\varphi_1 = (t_1, c_1)$ and $\varphi_2 = (t_2, c_2)$, then*

$$(c_1 \in \mathsf{LPcomm}_{t_1} \wedge c_2 \in \mathsf{LPcomm}_{t_2} \uplus \mathsf{GPcomm}_{t_2}) \vee (c_1 \in \mathsf{LPcomm}_{t_1} \uplus \mathsf{GPcomm}_{t_1} \wedge c_2 \in \mathsf{LPcomm}_{t_2}).$$

Then $\mathsf{eval}(s, \varphi_1\, \varphi_2) = \mathsf{eval}(s, \varphi_2\, \varphi_1)$ for any state s.

Since τ^i is well-formed, by Definition 12 for any action (t', c) in it we have that $c \in \mathsf{LPcomm}_t \uplus \mathsf{GPcomm}_t$ and, if $c \in \mathsf{GPcomm}_t$, then the action is non-transactional. Given this and the properties of τ_2' established above, by applying Proposition 23 repeatedly we get that $\mathsf{eval}(s', \varphi\, \tau_2') = \mathsf{eval}(s', \tau_2'\, \varphi)$ for any state s'. Hence,

$$\mathsf{eval}(s, \tau^{i+1}) = \mathsf{eval}(s, \tau_1\, \varphi\, \tau_2'\, \tau_2'') = \mathsf{eval}(s, \tau_1\, \tau_2'\, \varphi\, \tau_2'') = \mathsf{eval}(s, \tau^i). \quad \square$$

7. NECESSITY OF THE OPACITY RELATION

In this section we prove that observational refinement with respect to states implies the opacity relation restricted to the complete subsets of the transactional systems. We only give a proof sketch here and defer the full proof to [2, Appendix B].

THEOREM 24. $\mathcal{H}_C \preceq_{\mathsf{state}} \mathcal{H}_A \implies \mathcal{H}_C|_{\mathsf{complete}} \sqsubseteq \mathcal{H}_A|_{\mathsf{complete}}$.

PROOF SKETCH. For every complete history $H \in \mathcal{H}_C$ we construct a program P_H where every thread performs the sequence of transactions specified by H, records the return values obtained from methods of transactional objects, and monitors whether the real-time order between actions includes that in H.

In more detail, given that the history H is well-formed, we construct the code of every thread t as a sequence of `atomic` blocks, corresponding to txbegin, committed and aborted actions in $H|_t$, which perform the sequence of method invocations determined by call and ret actions in $H|_t$. We record the return value for every method invocation in a dedicated variable local to the corresponding thread, which is never rewritten (this is possible because H is finite). The return status of every transaction is also recorded in a dedicated local variable. As a result, from the final state of an execution of P_H, we can reconstruct the interaction of each thread with the transactional system. (We rely here on the fact that the histories considered are complete; our notion of observation is not strong enough to distinguish between, e.g., histories $H\, \varphi\, H'$ and $H\, H'$ where φ is a call action that does not return.)

To check whether an execution of P_H complies with the real-time order in H, we exploit the ability of threads to communicate via global variables outside transactions. For each transaction in H, we introduce a global variable g, which is initially 0 and is set to 1 by the thread executing the transaction right after the transaction completes, by a command following the corresponding `atomic` block. Before starting a transaction, each thread checks whether all transactions preceding this one in the real-time order in H have finished by reading the corresponding g variables. The outcome is recorded in a dedicated local variable.

Let s be a state with all variables set to distinguished initial values. From the above it follows that, given any trace $\tau \in [\![P_H]\!](s)$, by observing the states in $\mathsf{eval}(s, \tau)$, we can unambiguously determine whether the per-thread projections of $\mathsf{history}(\tau)$ are equal to those of H and whether the real-time order in $\mathsf{history}(\tau)$ includes that in H.

Now given $H \in \mathcal{H}_C$, we construct a trace $\tau \in [\![P_H]\!](s, \mathcal{H}_C)$ such that $\mathsf{history}(\tau) = H$ and $\mathsf{eval}(s, \tau) = \{s'\}$, where the state s' signals the above correspondence of the trace to H. By Definition 14, there exists a trace $\tau' \in [\![P_H]\!](s, \mathcal{H}_A)$ such that $\mathsf{eval}(s, \tau') = \{s'\}$. But then the per-thread projections of $S = \mathsf{history}(\tau')$ are equal to those of H and the real-time order in H is preserved in S. By Definition 4, this implies $H \sqsubseteq S$. $\square$

The proof highlights the features of our programming model that lead to the necessity result. First, the ability to access global variables outside transactions allows us to monitor the real-time order. Second, we use the ability of programs to observe values assigned to local variables during the execution of a transaction, even if the transaction aborts: P_H records the return values of method invocations by aborted transactions in special local variables, which can then be observed in final states. This necessitates treating aborted transactions in the same way as committed ones in the consistency definition.

8. RELATED WORK AND DISCUSSION

Previous work has studied transactional memory consistency by:

- investigating the semantics of different programming languages with atomic blocks and the feasibility of their efficient implementation [1, 11, 19]; or
- defining consistency conditions for TM [3,5,8,9,18] and proving that particular TM implementations validate them [4, 9].

Thus, previous work has tended to address the issue from the perspective of either programming languages or TM implementations and has not tried to relate these two levels in a formal manner. An exception is the work by Harris et al. [12], which proved that a specific TM implementation, Bartok-STM, validates a particular semantics of atomic blocks in a programming language.

This paper tries to fill in the gap in existing studies by relating the semantics of a programming language with atomic blocks to that of a TM system implementing them. Our work is complementary to previous proofs that certain TM systems satisfy opacity [4], as it lifts such results to the language level. Our work is also more general than that of Harris et al. [12], since our results allow establishing observational refinement for *any* TM implementation satisfying opacity. However, some of the above-mentioned papers [1, 19] investigated advanced language interfaces that we do not consider, such as nested transactions and access to shared data both inside and outside transactions.

This paper employs a well-known technique from the theory of programming languages, observational refinement [13, 14], to explore the most appropriate way to specify TM consistency. Observational refinement has previously been used to characterize correctness criteria for libraries of concurrent data structures. Thus, Filipovic et al. [6] proved that, in this setting sequential consistency [17] is necessary and sufficient for observational refinement, and so is linearizability [15] when client programs can interact via shared global variables. Gotsman and Yang [7] adjusted linearizability to account for infinite computations and showed its sufficiency for observational refinement in the case when the client can observe the validity of liveness properties. Our work takes this approach from the simpler setting of concurrent libraries to the more elaborate setup of transactional memory. Since we allow the abstract transactional system to have incomplete transactions, we hope that in the future we can generalize our consistency condition to specify liveness properties, along the lines of [7].

Opacity requires the TM behavior observed by aborted transactions to be as consistent as that observed by committed ones. Other proposed consistency conditions tried to relax this requirement. For example, *virtual world consistency (VWC)* [16] requires the behavior observed by an individual aborted transaction to be consistent only with the committed transactions from which it reads and those previously committed by the same thread. Our necessity result implies that VWC does not imply observational refinement for our programming language. However, this does not rule out the viability of VWC and related notions as a consistency condition for TM. VWC may well imply observational refinement for a programming language in which aborted transactions do not affect the rest of the computation (in particular, their modifications to local variables are rolled back) and a weaker notion of observations. This paper establishes an approach for evaluating and comparing TM consistency conditions, and in future work, we hope to apply it to VWC and other conditions, such as TMS [5, 18] and DU-opacity [3]. We also intend to investigate the behaviour of incomplete histories with respect to observational refinement, whose treatment by the opacity relation causes our result to fall short of the strict equivalence between the relation and the observational refinement.

Acknowledgements

This work is supported by EU FP7 projects TRANSFORM (238639) and ADVENT (308830). We thank Victor Luchangco, Eran Yahav, Hongseok Yang and the reviewers for helpful comments.

9. REFERENCES

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL*, pages 63–74, 2008.

[2] H. Attiya, A. Gotsman, S. Hans, and N. Rinetzky. A programming language perspective on transactional memory consistency. Technical Report CS-2013-04, Department of Computer Science, Technion, 2013.

[3] H. Attiya, S. Hans, P. Kuznetsov, and S. Ravi. Safety of deferred update in transactional memory. In *ICDCS*, 2013. To appear.

[4] A. Bieniusa and P. Thiemann. Proving isolation properties for software transactional memory. In *ESOP*, pages 38–56, 2011.

[5] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, pages 1–31, March 2012.

[6] I. Filipovic, P. W. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. In *ESOP*, pages 252–266, 2009.

[7] A. Gotsman and H. Yang. Liveness-preserving atomicity abstraction. In *ICALP (2)*, pages 453–465, 2011.

[8] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPOPP*, pages 175–184, 2008.

[9] R. Guerraoui and M. Kapalka. *Principles of Transactional Memory*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool, 2011.

[10] T. Harris, J. Larus, and R. Rajwar. *Transactional memory*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[11] T. Harris, S. Marlow, S. L. P. Jones, and M. Herlihy. Composable memory transactions. In *PPOPP*, pages 48–60, 2005.

[12] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI*, pages 14–25, 2006.

[13] J. He, C. Hoare, and J. Sanders. Data refinement refined. In *ESOP*, pages 187–196, 1986.

[14] J. He, C. Hoare, and J. Sanders. Prespecification in data refinement. *Information Processing Letters*, 25(2):71 – 76, 1987.

[15] M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[16] D. Imbs, J. R. G. de Mendívil, and M. Raynal. Brief announcement: virtual world consistency: a new condition for STM systems. In *PODC*, pages 280–281, 2009.

[17] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[18] M. Lesani, V. Luchangco, and M. Moir. Putting opacity in its place. In *WTTM*, 2012.

[19] K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *POPL*, pages 51–62, 2008.

Safety of Live Transactions in Transactional Memory: TMS is Necessary and Sufficient

Hagit Attiya* Alexey Gotsman† Sandeep Hans* Noam Rinetzky‡

Abstract

The main challenge of stating the correctness of transactional memory (TM) systems is the need to provide guarantees on the system state observed by *live* transactions, i.e., those that have not yet committed or aborted. A TM correctness condition should not be too restrictive, to allow flexibility in implementation, yet strong enough to disallow undesirable TM behavior, which can lead to run-time errors in live transactions. The latter feature is formalized by *observational refinement* between TM implementations, stating that properties of a program using a concrete TM implementation can be established by analyzing its behavior with an abstract TM, serving as a specification of the concrete one.

We show that a variant of *transactional memory specification* (*TMS*) is equivalent to observational refinement for the common programming model in which local variables are rolled back upon a transaction abort and, hence, is the weakest consistency condition for this case. This is challenging due to the nontrivial formulation of TMS, which allows different aborted and live transactions to have different views on the system state. Our proof reveals some natural, but subtle, assumptions on the TM required for the equivalence result.

This is a submission to the regular track.
It can be considered for the best student paper award: Sandeep Hans is a full-time student.

*Technion - Israel Institute of Technology, Israel, {hagit,sandeep}@cs.technion.ac.il
†IMDEA Software Institute, Spain, alexey.gotsman@imdea.org
‡Tel Aviv University, Israel, maon@cs.tau.ac.il

1 Introduction

Transactional memory (TM) eases the task of writing concurrent applications by letting the programmer designate certain code blocks as *atomic*. TM allows developing a program and reasoning about its correctness as if each atomic block executes as a *transaction*—in one step and without interleaving with others—even though in reality the blocks can be executed concurrently. Figure 1 shows how atomic blocks yield simple code for computations involving several shared *transactional objects* X, Y and Z, access to which is mediated by the TM.

```
result := abort;
while (result == abort) {
  result := atomic {
    x = X.read();
    y = Y.read();
    z = 42 / (x - y);
    Z.write(z);
  } }
```

Figure 1: TM usage

The common approach to stating TM correctness is through a *consistency condition* that restricts the possible TM executions. The main subtlety of formulating such a condition is the need to provide guarantees on the state of transactional objects observed by *live* transactions, i.e., those that have not yet committed or aborted. Because live transactions can always be aborted, one might think it unnecessary to provide any guarantees for them, as in fact done by common database consistency conditions [12]. However, in the setting of transactional memory, this is often unsatisfactory. For example, in Figure 1 the programmer may rely on the fact that $\mathsf{X} \neq \mathsf{Y}$, and, correspondingly, make sure that every committing transaction preserves this invariant. If we allow the transaction to read values of X and Y violating the invariant (counting on it to abort later, due to inconsistency), this will lead to the program *faulting* due to a division by zero.

The question of which TM consistency condition to use is far from settled, with several candidates having been proposed [2,3,6,10]. An ideal condition should be as weak as possible, to allow flexibility in TM implementations, yet strong enough to satisfy the intuitive expectations of the programmer and, in particular, to disallow undesirable behaviors such as the one described above. *Observational refinement* [7, 8] allows formalizing the programmer's expectations and thereby evaluating consistency conditions systematically. Consider two TM implementations—a *concrete* one, such as an efficient TM, and an *abstract* one, such as a TM executing every atomic block atomically. Informally, the concrete TM observationally refines the abstract one for a given programming language if every behavior a user can observe of a program P in this language linked with the concrete TM can also be observed when P is linked with the abstract TM instead. This allows the programmer to reason about the behavior of P (e.g., the preservation of the invariant $\mathsf{X} \neq \mathsf{Y}$) using the expected intuitive semantics formalized by the abstract TM; the observational refinement relation implies that the conclusions (e.g., the safety of the division in Figure 1) will carry over to the case when P uses the concrete TM.

In prior work [1] we showed that a variant of the *opacity* condition [6] coincides with observational refinement for a particular programming language and, hence, is the weakest consistency condition for this language. Roughly speaking, a concrete TM implementation is in the opacity relation with an abstract one if for any sequence of interactions with the concrete TM, dubbed a *history*, there exists a history of the abstract TM where: (i) the actions of every separate thread are the same as in the original history; and (ii) the order of non-overlapping transactions present in the original history is preserved. However, our result considered a programming language in which local variables modified by a transaction are not rolled back upon an abort. Although this assumption holds in some situations (e.g., for Scala STM [13]), it is non-standard and most TM systems do not satisfy it. In this paper, we consider a variant of *transactional memory specification* (*TMS*) [3], a condition weaker than opacity,[1] and show that, under some natural assumptions on the TM, it coincides with observational refinement for a programming language in which local variables do get rolled back upon an abort.

This result is not just a straightforward adjustment of the one about opacity to a more realistic setting: TMS weakens opacity in a nontrivial way, which makes reasoning about its relationship with observational refinement much more intricate. In more detail, the key feature of opacity is that the behavior of *all* transactions in a history of the concrete TM, including aborted and live ones, has to be explained by a single history of the abstract TM. TMS relaxes this requirement by requiring only committed transactions in the concrete history to be explained by a single abstract one obeying (i)–(ii) above; every response obtained from the TM in an aborted or live transaction may be explained by a separate abstract history. The constraints on the choice of the abstract history are subtle: on one

[1] The condition we present here is actually called TMS1 in [3, 11]. These papers also propose another condition, called TMS2, but it is stronger than opacity [11] and therefore not considered here.

hand, somewhat counter-intuitively, TMS allows it to include transactions that aborted in the concrete history, with their status changed to committed, and exclude some that committed; on the other hand, this is subject to certain carefully chosen constraints. The flexibility in the choice of the abstract history is meant to allow the concrete TM implementation to perform as many optimizations as possible. However, it is not straightforward to establish that this flexibility does not invalidate observational refinement (and hence, the informal guarantees that programmers expect from a TM) or that the TMS definition cannot be weakened further.

Our results ensure that this is indeed the case. Informally, if local variables are not rolled back when transactions abort, threads can communicate to each other the observations they make inside aborted transactions about the state of transactional objects. This requires the TM to provide a consistent view of this state across all transactions, as formalized by the use of a single abstract history in opacity. However, if local variables are rolled back upon an abort, no information can leak out of an uncommitted transaction, possibly apart from the fact that the code in the transaction has faulted, stopping the computation. To get observational refinement in this case we only need to make sure that a fault in the transaction occurring with the concrete TM could be reproduced with the abstract one. For this it is sufficient to require that the state of transactional objects seen by every live transaction can be explained by some abstract history; different transactions can be explained by different histories.

Technically, we prove that TMS is sufficient for observational refinement by establishing a nontrivial property of the set of computations of a program, showing that a live transaction cannot notice the changes in the committed/aborted status of transactions concurrent with it that are allowed by TMS (Lemma 9, Section 6.1). Proving that TMS is necessary for observational refinement is challenging as well, as this requires us to devise multiple programs that can observe whether the subtle constraints governing the change of transaction status in TMS are fulfilled by the TM. We have identified several closure properties on the set of histories produced by the abstract TM required for these results to hold. Although intuitive, these properties are not necessarily provided by an arbitrary TM, and our results demonstrate their importance.

To concentrate on the core goal of this paper, the programming language we consider does not allow explicit transaction aborts or transaction nesting and assumes a static separation of transactional and non-transactional shared memory. Extending our development to lift these restrictions is an interesting avenue for future work. Also, due to space constraints, we defer some of the proofs to Appendix D.

2 Programming Language Syntax

We consider a language where a program $P = C_1 \parallel \cdots \parallel C_m$ is a parallel composition of ***threads*** C_t, $t \in \mathsf{ThreadID} = \{1, \ldots, m\}$. Every thread $t \in \mathsf{ThreadID}$ has a set of ***local variables*** $\mathsf{LVar}_t = \{x, y, \ldots\}$ and threads share a set of ***global variables*** $\mathsf{GVar} = \{g, \ldots\}$, all of type integer. We let $\mathsf{Var} = \mathsf{GVar} \uplus \biguplus_{t=1}^{m} \mathsf{LVar}_t$ be the set of all program variables. Threads can also access a transactional memory, which manages a fixed collection of ***transactional objects*** $\mathsf{Obj} = \{o, \ldots\}$, each with a set of ***methods*** $\mathsf{Method} = \{f, \ldots\}$ that threads can call. For simplicity, we assume that each method takes one integer parameter and returns an integer value, and that all objects have the same set of methods. The syntax of commands C is standard:

$$C \ ::= \ c \mid C; C \mid \texttt{while}\,(b)\,\texttt{do}\,C \mid \texttt{if}\,(b)\,\texttt{then}\,C\,\texttt{else}\,C \mid x := \texttt{atomic}\,\{C\} \mid x := o.f(e)$$

where b and e denote Boolean and integer expressions over local variables, left unspecified. The syntax includes ***primitive commands*** c from a set PComm, sequential composition, conditionals, loops, `atomic` blocks and object method invocations. Primitive commands execute atomically, and they include assignments to local and global variables and a special `fault` command, which stops the execution of the program in an error state. Thus, `fault` encodes illegal computations, such as division by zero or dereference of `null`-valued pointers.

An ***atomic block*** $x := \texttt{atomic}\,\{C\}$ executes C as a ***transaction***, which the TM can ***commit*** or ***abort***. The system's decision is returned in the local variable x, which gets assigned distinguished values committed or aborted. We do not allow programs in our language to abort a transaction explicitly and forbid nested atomic blocks and, hence, nested transactions. We also assume that a program can invoke methods on transactional objects only inside atomic blocks and access global variables only outside them. Local variables can be accessed in both cases; however, threads cannot access local variables of other threads. Due to space constraints, we defer the formalisation of

the rules on variable accesses to Appendix A. When we later define the semantics of our programming language, we mandate that, if a transaction is aborted, local variables are rolled back to the values they had at its start, and hence, the values written to them by the transaction cannot be observed by the following non-transactional code.

3 Model of Computations

To define the notion of observational refinement for our programming language and the TMS consistency condition, we need a formal model for program computations. To this end, we introduce *traces*, which are certain finite sequences of *actions*, each describing a single computation step (we do not consider infinite computations).

DEFINITION 1. *Let* ActionId *be a set of action identifiers. A* ***TM interface action*** ψ *has one of the following forms:*

Request actions	*Matching response actions*
$(a, t, \mathsf{txbegin})$	$(a, t, \mathsf{OK}) \mid (a, t, \mathsf{aborted})$
$(a, t, \mathsf{txcommit})$	$(a, t, \mathsf{committed}) \mid (a, t, \mathsf{aborted})$
$(a, t, \mathsf{call}\ o.f(n))$	$(a, t, \mathsf{ret}(n')\ o.f) \mid (a, t, \mathsf{aborted})$

where $a \in \mathsf{ActionId}$, $t \in \mathsf{ThreadID}$, $o \in \mathsf{Obj}$, $f \in \mathsf{Method}$ *and* $n, n' \in \mathbb{Z}$. *A* ***primitive action*** ϕ *has the form* (a, t, c), *where* $c \in \mathsf{PComm}$ *is a primitive command. We use* φ *to range over actions of either type.*

TM interface actions denote the control flow of a thread t crossing the boundary between the program and the TM: ***request*** actions correspond to the control being transferred from the former to the latter, and ***response*** actions, the other way around. A txbegin action is generated upon entering an `atomic` block, and a txcommit action when a transaction tries to commit upon exiting an `atomic` block. Actions call and ret denote a call to and a return from an invocation of a method on a transactional object and are annotated with the method parameter or return value. The TM may abort a transaction at anys point when it is in control; this is recorded by an aborted response action.

A ***trace*** τ is a finite sequence of actions satisfying certain natural well-formedness conditions (stated informally due to space constraints; see Appendix B): every action in τ has a unique identifier; no action follows a `fault`; request and response actions are properly matched; actions denoting the beginning and end of transactions are properly matched; call and ret actions occur only inside transactions; and commands in τ do not access local variables of other threads and do not access global variables when inside a transaction. We denote the set of traces by Trace. A ***history*** is a trace containing only TM interface actions; we use H, S to range over histories. We specify the behavior of a TM implementation by the set of possible interactions it can have with programs: a ***transactional memory*** $\mathcal{T}$ is a set of histories that is prefix-closed and closed under renaming action identifiers.

We denote irrelevant expressions by _ and use the following notation: $\tau(i)$ is the i-th element of τ; $\tau|_t$ is the projection of τ onto actions of the form $(\_, t, \_)$; $|\tau|$ is the length of τ; $\tau_1\tau_2$ is the concatenation of τ_1 and τ_2. We say that an action φ is in τ, denoted by $\varphi \in \tau$, if $\tau = \_\varphi\_$. The empty sequence of actions is denoted ε.

A ***transaction*** T is a non-empty trace such that it contains actions by the same thread, begins with a txbegin action and only its last action can be a committed or an aborted action. We use the following terminology for transactions. A transaction T is: ***committed*** if it ends with a committed action, ***aborted*** if it ends with aborted, ***commit-pending*** if it ends with txcommit, and ***live***, in all other cases. We refer to this as T's ***status***. A transaction T is ***completed*** if it is either committed or aborted, and ***visible*** if it contains a txcommit action. A transaction T is ***in a trace*** τ, written $T \in \tau$, if $\tau|_t = \tau_1 T \tau_2$ for some t, τ_1 and τ_2, where either T is completed or τ_2 is empty. We denote the set of all transactions in τ by $\mathsf{tx}(\tau)$ and use self-explanatory notation for various subsets of transactions: $\mathsf{committed}(\tau)$, $\mathsf{aborted}(\tau)$, $\mathsf{pending}(\tau)$, $\mathsf{live}(\tau)$, $\mathsf{visible}(\tau)$. For $\varphi \in \tau$, the ***transaction of*** φ ***in*** τ, denoted $\mathsf{txof}(\varphi, \tau)$, is the subsequence of τ comprised of all actions that are in the same transaction as φ (undefined if φ does not belong to a transaction).

4 Transactional Memory Specification (TMS)

In this section we formalise the TMS [3] correctness condition in our setting. TMS was originally formulated in an operational manner, using I/O automata; here we present a more abstract definition appropriate for our goals (we

provide further comparison in Section 7). As is common in consistency conditions for shared-memory concurrency, such as opacity [6] or linearizability [9], a crucial building block in the TMS definition is the following notion of the *real-time order*, which captures the order between non-overlapping transactions in a history.

DEFINITION 2. *Let $\psi = (\_, t, \_)$ and $\psi' = (\_, t', \_)$ be two actions in a history H; ψ **is before** ψ' **in the real-time order** in H, denoted by $\psi \prec_H \psi'$, if $H = H\psi H_2 H_2' \psi' H_3$ and either (i) $t = t'$ or (ii) $(\_, t', \mathsf{txbegin}) \in H_2'\psi'$ and either $(\_, t, \mathsf{committed}) \in \psi H_2$ or $(\_, t, \mathsf{aborted}) \in \psi H_2$. A transaction T **is before** an action ψ' **in the real-time order** in H, denoted by $T \prec_H \psi'$, if $\psi \prec_H \psi'$ for any $\psi \in T$. A transaction T **is before** a transaction T' **in the real-time order** in H, denoted by $T \prec_H T'$, if $T \prec_H T'(1)$.*

Given a history H of program interactions with a concrete TM, TMS requires us to explain the behavior of all committed transactions in H by a single history S of the abstract TM, and to explain every response action ψ in H by an abstract history S_ψ. As we show in this paper, the existence of such explanations ensures that TMS imply observational refinement between the two TMs: the behavior of a program during some transaction in the history H of the program's interactions with the concrete TM can be reproduced when the program interacts with the abstract TM according to the history S or S_ψ. Below we use this insight when explaining the rationale for key TMS features.

The history S_ψ used to explain a response action ψ includes the transaction of ψ and a subset of transactions from H whose actions justify the response ψ. The following notion of a *possible past* of a history $H = H_1\psi$ defines all sets of transactions from H that can form S_ψ. Note that, if a transaction selected by this definition is aborted or commit-pending in H, its status is changed to committed when constructing S_ψ, as formalized later in Definition 4. Informally, the response ψ is given as if all the transactions in its possible past have taken effect and all the others have not. We first give the formal definition of a possible past, and then explain it on an example.

DEFINITION 3. *A history $H_\psi = H_1'\psi$ is a **possible past** of a history $H = H_1\psi$, where ψ is a response action, if:*

(i) H_1' is a subsequence of H_1;

(ii) H_ψ is comprised of the transaction of ψ and some of the visible transactions in H:

$$\mathsf{tx}(H_\psi) \subseteq \{\mathsf{txof}(\psi, H)\} \cup \mathsf{visible}(H).$$

(iii) for every transaction $T \in H_\psi$, out of all transactions preceding T in the real-time order in H, the history H_ψ includes exactly the committed ones:

$$\forall T \in \mathsf{tx}(H_\psi).\, \forall T'.\, T' \prec_{H_\psi} T \iff T' \prec_H T \wedge T' \in \mathsf{committed}(H).$$

We denote the set of possible pasts of H by $\mathsf{TMSpast}(H)$.

We explain the definition using the history H of the trace shown in Figure 2; one of its possible pasts H_ψ consists of the transactions T_1, T_4 and T_5. According to (ii), the transaction of ψ (T_5 in Figure 2) is always included into any possible past, and live transactions are excluded: since they have not made an attempt to commit, they should not have an effect on ψ. Out of the visible transactions in H, we are allowed to select which ones to include (and, hence, treat as committed), subject to (iii): if we include a transaction T then, out of all transactions preceding T in the real-time order in H, we have to include exactly the committed ones. For example, since T_4 and T_5 are included in H_ψ, T_1 must also be included and T_3 must not. This condition is necessary for TMS to imply observational refinement. Informally, T_3 cannot be included into H_ψ because, in a program producing H, in between T_3 aborting and T_5 starting, thread t_2 could have communicated to thread t_3 the fact that T_3 has aborted, e.g., using a global variable g, as illustrated in Figure 2. When executing ψ, the code in T_5 may thus expect that T_3 did not take effect; hence, the result of ψ has to reflect this, so that the code behavior is preserved when replacing the concrete TM by an abstract one in observational refinement. This is a key idea used in our proof that TMS is necessary for observational refinement (Section 6.2). In contrast to T_3, we can include T_4 into H_ψ even if it is aborted or commit-pending. Since our language does not allow accessing global variables inside transactions, there is no way for the code in T_5 to find out about the status of T_4 from thread t_2, and hence, this code will not notice if the status of T_4 is changed to committed when replacing the concrete TM by an abstract one in observational

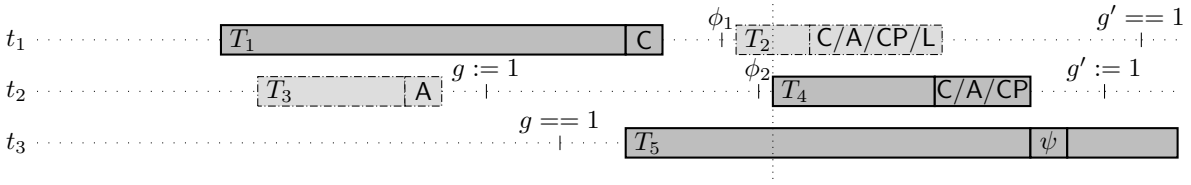


Figure 2: Transactions T_1, T_4 and T_5 form one possible past of the history H of the trace shown. Allowed status of transactions in H is denoted as follows: committed – C, aborted – A, commit-pending – CP, live – L. The transaction T_5 executes only primitive actions after ψ in the trace.

refinement. For similar reasons, we can exclude T_2 from H_ψ even if it is committed. This idea is used in our proof that TMS is sufficient for observational refinement (Section 6.1).

Before giving the definition of TMS, we introduce operations used to change the status of transactions in a possible past of a history to committed. *Suffix commit completion* below converts commit-pending transactions into committed; then *completed possible past* defines a possible past with all transactions committed.

DEFINITION 4. *A history H^c is a **suffix completion** of a history $H\psi$ if $H^c = H\psi H'$, any action in H' is either* committed *or* aborted*, and every transaction in H^c except possibly that of ψ, is completed. It is a **suffix commit completion** of H if H' consists of* committed *actions only. The sets of suffix completions and suffix commit completions of H are denoted* $\mathsf{comp}(H)$ *and* $\mathsf{comcomp}(H)$*, respectively.*

*A history H^c_ψ is a **completed possible past** of a history $H = H_1\psi$, if H^c_ψ is a suffix commit completion of a history obtained from a possible past $H'_1\psi$ of H by replacing all the* aborted *actions in H'_1 by* committed *actions. The set of completed possible pasts of H is denoted* $\mathsf{cTMSpast}(H)$*:*

$$\mathsf{cTMSpast}(H_1\psi) = \{H^c_\psi \mid \exists H'_1.\, H'_1\psi \in \mathsf{TMSpast}(H_1\psi) \wedge H^c_\psi \in \mathsf{comcomp}(\mathsf{com}(H'_1)\,\psi)\},$$

where $|\mathsf{com}(H'_1)| = |H'_1|$ *and* $\mathsf{com}(H'_1)(i) = (\text{if } (H'_1(i) = (a, t, \mathsf{aborted})) \text{ then } (a, t, \mathsf{committed}) \text{ else } H'_1(i))$.

For example, one completed possible past of the history in Figure 2 consists of the transactions T_1, T_4 and T_5, with the status of the latter changed to committed if it was previously aborted or commit-pending. Note that a history H has a suffix completion only if H is of the form $H = H_1\psi$ where all the transactions in $H_1\psi$, except possibly that of ψ, are commit-pending or completed. Also, $\mathsf{cTMSpast}(H_1\psi) \neq \emptyset$ only if, in addition, ψ is a response action.

The following definition of the *TMS relation* between TMs matches a history H arising from a concrete TM with a similar history S of an abstract TM. As part of this matching, we require that S be a permutation of H preserving the real-time order, which is formalised by the *opacity relation* [1, 6]. As in Definition 3(iii), this requirement is necessary to ensure observational refinement between the TMs: preserving the real-time order is necessary to preserve communication between threads when replacing the concrete TM with the abstract one.

DEFINITION 5. *A history H is in the **opacity relation** with a history S, denoted by $H \sqsubseteq_{\mathsf{op}} S$, if*

$$\forall \psi, \psi'.\, (\psi \in S \iff \psi \in H) \wedge (\psi \prec_H \psi' \implies \psi \prec_S \psi').$$

DEFINITION 6. *A history H is in the **TMS relation** with TM $\mathcal{T}$, denoted $H \sqsubseteq_{\mathsf{tms}} \mathcal{T}$, if:*

(i) $\exists H^c \in \mathsf{comp}(H|_{\neg\mathsf{live}})$, $S \in \mathcal{T}.\, H^c|_{\mathsf{com}} \sqsubseteq_{\mathsf{op}} S$, where $\cdot|_{\neg\mathsf{live}}$ and $\cdot|_{\mathsf{com}}$ are the projections to actions by transactions that are not live and by committed transactions, respectively; and

(ii) for every response action ψ such that $H = H_1\psi H_2$: $\exists H^c_\psi \in \mathsf{cTMSpast}(H_1\psi).\, \exists S_\psi \in \mathcal{T}.\, H^c_\psi \sqsubseteq_{\mathsf{op}} S_\psi$.

*A TM $\mathcal{T}_C$ is in the **TMS relation** with a TM $\mathcal{T}_A$, denoted by $\mathcal{T}_C \sqsubseteq_{\mathsf{tms}} \mathcal{T}_A$, if $\forall H \in \mathcal{T}_C.\, H \sqsubseteq_{\mathsf{tms}} \mathcal{T}_A$.*

5 Observational Refinement

Our main result relates TMS to *observational refinement*, which we introduce in this section. This requires defining the semantics of the programming language, i.e., the set of traces that computations of programs produce. A ***state*** of a program records the values of all its variables: $s \in \mathsf{State} = \mathsf{Var} \to \mathbb{Z}$. The semantics of a program $P = C_1 \parallel \cdots \parallel C_m$ is given by the set of traces $[\![P, \mathcal{T}]\!](s) \subseteq \mathsf{Trace}$ it produces when run with a TM $\mathcal{T}$ from an

initial state s. To define this set, we first define the set of traces $[\![P]\!](s) \subseteq \mathsf{Trace}$ that a program can produce when run from s with the behavior of the TM unrestricted, i.e., considering all possible values the TM can return to object method invocations and allowing transactions to commit or abort arbitrarily. We then restrict to the set of traces produced by P when run with $\mathcal{T}$ by selecting those traces that interact with the TM in a way consistent with $\mathcal{T}$: $[\![P, \mathcal{T}]\!](s) = \{\tau \mid \tau \in [\![P]\!](s) \wedge \mathsf{history}(\tau) \in \mathcal{T}\}$, where $\mathsf{history}(\cdot)$ projects to TM interface actions. Due to space constraints, we defer the formal definition of $[\![P]\!](s)$ to Appendix C. The definition follows the intuitive semantics of our programming language. In particular, it mandates that local variables be rolled back upon a transaction abort and includes traces corresponding to incomplete program computations into $[\![P]\!](s)$.

We can now define the notions of *observations* and *observational refinement*. Informally, given a trace τ of a client program, we consider observable: (i) the sequence of actions performed outside transactions in τ; (ii) the per-thread sequence of actions in τ that are not part of uncommitted transactions; and (iii) whether a τ ends with `fault` or not. Then observational refinement between a concrete TM $\mathcal{T}_C$ and an abstract one $\mathcal{T}_A$ states that every observable behavior of a program P using $\mathcal{T}_C$ can be reproduced when P uses $\mathcal{T}_A$. Hence, any conclusion about its observable behavior that a programmer makes assuming $\mathcal{T}_A$ will carry over to $\mathcal{T}_C$. Since our notion of observations excludes actions performed inside aborted or live transactions other than faulting, the programmer cannot make any conclusions about them. But, crucially, the programmer can be sure that, if a program is non-faulting under $\mathcal{T}_A$, it will stay so under $\mathcal{T}_C$. Let us call an action $\varphi \in \tau$ ***transactional*** if $\varphi \in T$ for some $T \in \tau$, and ***non-transactional*** otherwise. We denote by $\tau|_{\mathsf{trans}}$ and $\tau|_{\neg\mathsf{trans}}$ the projections of τ to transactional and non-transactional actions.

DEFINITION 7. *The **thread-local observable behavior** of thread t in a trace τ, denoted by $\mathsf{observable}_t(\tau)$, is $\lightning$ if $\tau|_t$ ends with a* `fault` *action, and $(\tau|_t)|_{\mathsf{obs}}$ otherwise, where $\cdot|_{\mathsf{obs}}$ denotes the projection to non-transactional actions and actions by committed transactions. A TM $\mathcal{T}_C$ **observationally refines** a TM $\mathcal{T}_A$, denoted by $\mathcal{T}_C \preceq \mathcal{T}_A$, if for every program P, state s and trace $\tau \in [\![P, \mathcal{T}_C]\!](s)$ we have: (i) $\exists \tau' \in [\![P, \mathcal{T}_A]\!](s).\, \tau'|_{\neg\mathsf{trans}} = \tau|_{\neg\mathsf{trans}}$; and (ii) $\forall t.\, \exists \tau'_t \in [\![P, \mathcal{T}_A]\!](s).\, \mathsf{observable}_t(\tau'_t) = \mathsf{observable}_t(\tau)$.*

6 Main Result

The main result of this paper is that the TMS relation can be characterized in terms of observational refinement for a class of TMs that enjoy certain natural closure properties.

- A TM $\mathcal{T}$ is ***closed under immediate aborts*** if it allows adding arbitrary ***immediately aborted*** transactions to any history $H \in \mathcal{T}$: if $H_1 H_2 H_3 \in \mathcal{T}$, $H_2|_t = \varepsilon$ and $H_1(\_, t, \mathsf{txbegin}) H_2(\_, t, \mathsf{aborted}) H_3$ is a well-formed history, then the latter belongs to $\mathcal{T}$.
- A TM $\mathcal{T}$ is ***closed under removing transaction responses*** if whenever $H_1(\_, t, \mathsf{aborted}) H_2 \in \mathcal{T}$ or $H_1(\_, t, \mathsf{committed}) H_2 \in \mathcal{T}$ for H_2 not containing actions by t, we also have $H_1 H_2 \in \mathcal{T}$.
- A TM $\mathcal{T}$ is ***closed under removing live and aborted transactions*** if whenever $H \in \mathcal{T}$, we also have $H' \in \mathcal{T}$ for any history H' which is a subsequence of H such that $\mathsf{committed}(H') = \mathsf{committed}(H)$, $\mathsf{pending}(H') = \mathsf{pending}(H)$, $\mathsf{live}(H') \subseteq \mathsf{live}(H)$ and $\mathsf{aborted}(H') \subseteq \mathsf{aborted}(H)$.
- A TM $\mathcal{T}$ is ***closed under commit-pending transactions completion*** if whenever $H \in \mathcal{T}$ and $\mathsf{comp}(H) \neq \emptyset$, we have $\mathsf{comp}(H) \cap \mathcal{T} \neq \emptyset$.
- A TM $\mathcal{T}$ is ***closed under aborting live transactions*** if whenever $H_1 \psi \in \mathcal{T}$ and $\psi \in \{(\_, t, \mathsf{OK}), (\_, t, \mathsf{ret}\_)\}$, we have $H_1 \psi \psi_c(\_, t, \mathsf{aborted}) \in \mathcal{T}$, where ψ_c is an arbitrary request action by t.

These properties are satisfied by the expected TM specification that executes every transaction atomically [1].

THEOREM 8. *Let $\mathcal{T}_C$ and $\mathcal{T}_A$ be transactional memories.*

(i) If $\mathcal{T}_A$ is closed under immediate aborts and removing transaction responses, then $\mathcal{T}_C \sqsubseteq_{\mathsf{tms}} \mathcal{T}_A \Longrightarrow \mathcal{T}_C \preceq \mathcal{T}_A$.

(ii) If $\mathcal{T}_A$ is closed under removing live and aborted transactions, aborting live transactions and commit-pending transactions completion, then $\mathcal{T}_C \preceq \mathcal{T}_A \Longrightarrow \mathcal{T}_C \sqsubseteq_{\mathsf{tms}} \mathcal{T}_A$.

6.1 Proof of Theorem 8(i) (Sufficiency)

Let us fix a program $P = C_1 \parallel \ldots \parallel C_m$, state s and TMs $\mathcal{T}_C$ and $\mathcal{T}_A$ such that $\mathcal{T}_C \sqsubseteq_{\mathsf{tms}} \mathcal{T}_A$ and $\mathcal{T}_A$ satisfies the closure conditions stated in the theorem. As we have noted before, the main subtlety of TMS lies in explaining the

behavior of a live transaction under $\mathcal{T}_C$ by a history of $\mathcal{T}_A$ where the committed/aborted status of some transactions is changed, as formalized by the use of $\mathsf{cTMSpast}$ in Definition 6(ii). Correspondingly, the most challenging part of the proof is to show that a trace from $[\![P, \mathcal{T}_C]\!](s)$ with a `fault` inside a live transaction can be transformed into a trace with the `fault` from $[\![P, \mathcal{T}_A]\!](s)$. The following lemma describes the first and foremost step of this transformation: it converts a trace $\tau \in [\![P]\!](s)$ with a live transaction into another trace from $[\![P]\!](s)$ that contains the same live transaction, but whose history of non-aborted transactions is a given element of $\mathsf{cTMSpast}(\mathsf{history}(\tau))$. In other words, this establishes that the live transaction cannot notice the changes of the status of other transactions done by $\mathsf{cTMSpast}$. Let $\tau|_{\neg\mathsf{abortedtx}}$ be the projection of τ excluding aborted transactions.

LEMMA 9 (Live transaction insensitivity). *Let $\tau = \tau_1\psi\tau_2 \in [\![P]\!](s)$ be such that ψ is a response action by thread t_0 that is not a* committed *or* aborted *action and τ_2 contains only primitive actions by thread t_0. Consider $H^c_\psi \in \mathsf{cTMSpast}(\mathsf{history}(\tau))$. There exists $\tau_\psi \in [\![P]\!](s)$ such that $\mathsf{history}(\tau_\psi)|_{\neg\mathsf{abortedtx}} = H^c_\psi$ and $\tau_\psi|_{t_0} = \tau|_{t_0}$.*

PROOF. We first show how to construct τ_ψ and then prove that it satisfies the required properties. We illustrate the idea of its construction using the trace τ in Figure 2. Let $\mathsf{history}(\tau) = H_1\psi$. Since $H^c_\psi \in \mathsf{cTMSpast}(H)$, by Definition 4 there exist histories H'_1, H''_1, and H^{cc} such that

$$H'_1\psi \in \mathsf{TMSpast}(H_1\psi) \quad \wedge \quad H''_1 = \mathsf{com}(H'_1) \quad \wedge \quad H^c_\psi = H''_1\psi H^{cc} \in \mathsf{comcomp}(H''_1\psi).$$

Recall that, for the τ in Figure 2, $H'_1\psi$ consists of the transactions T_1, T_4 and T_5. Then H''_1 is obtained from H'_1 by changing the last action of T_4 to committed if it was aborted; H^c_ψ is obtained by completing T_4 with a committed action if it was commit-pending. The trickiness of the proof comes from the fact that just mirroring these transformations on τ may not yield a trace of the program P: for example, if T_4 aborted, the code in thread t_2 following T_4 may rely on this fact, communicated to it by the TM via a local variable. Fortunately, we show that it is possible to construct the required trace by erasing certain suffixes of every thread and therefore getting rid of the actions that could be sensitive to the changes of transaction status, such as those following T_4. This erasure has to be performed carefully, since threads can communicate via global variables: for example, the value written by the assignment to g' in the code following T_4 may later be read by t_1, and, hence, when erasing the the former, the latter action has to be erased as well. We now explain how to truncate τ consistently.

Let ψ^b be the last txbegin action in $H'_1\psi$; then for some traces τ^b_1 and τ^b_2 we have $\tau = \tau^b_1\psi^b\tau^b_2\psi\tau_2$. For the τ in Figure 2, ψ^b is the txbegin action of T_4. Our idea is, for every thread other than t_0, to erase all its actions that follow the last of its transactions included into $H'_1\psi$ or its last non-transactional action preceding ψ^b, whichever is later. Formally, for every thread t, let τ^I_t denote the prefix of $\tau|_t$ that ends with the last TM interface action of t in $H'_1\psi$, or ε if no such action exists. For example, in Figure 2, $\tau^I_{t_1}$ and $\tau^I_{t_2}$ end with the last TM interface actions of T_1 and T_4, respectively. Similarly, let τ^N_t denote the prefix of $\tau|_t$ that ends in the last non-transactional action of t in τ^b_1, or ε if no such action exists. For example, in Figure 2, $\tau^N_{t_1}$ and $\tau^N_{t_2}$ end with ϕ_1 and ϕ_2, respectively. Let $\tau_{t_0} = \tau|_{t_0}$ and for any $t \neq t_0$ let τ_t be τ^I_t, if $|\tau^N_t| < |\tau^I_t|$, and τ^N_t, otherwise. We then let the truncated trace τ' be the subsequence of τ such that $\tau'|_t = \tau_t$ for any t. Thus, for the τ in Figure 2, in the corresponding trace τ' the actions of t_1 end with ϕ_1 and those of t_2 with the last action of T_4; note that this erases both operations on g'. To construct τ_ψ from τ', we mirror the transformations of H'_1 into H''_1 and H^c_ψ. Let τ'' be defined by $|\tau''| = |\tau'|$ and

$$\tau''(i) = (\text{if } (\tau'(i) = (a, t, \mathsf{aborted}) \wedge \tau'(i) \in H'_1)) \text{ then } (a, t, \mathsf{committed}) \text{ else } \tau'(i)).$$

Then we let $\tau_\psi = \tau''H^{cc}$.

We first prove that $\tau_\psi|_{t_0} = \tau|_{t_0}$. Let $T = \mathsf{txof}(\psi, H_1\psi)$; then by Definition 3(ii), $T \in H'_1\psi$. Hence, by Definition 3(iii) we have

$$\forall T'.\, T' \prec_{H'_1\psi} T \iff T' \prec_{H_1\psi} T \wedge T' \in \mathsf{committed}(H_1\psi), \tag{1}$$

so that $(H'_1\psi)|_{t_0}$ does not contain aborted transactions and $\tau''|_{t_0} = \tau'|_{t_0} = \tau|_{t_0}$. Besides, $H^{cc}|_{t_0} = \varepsilon$ and, hence, $\tau_\psi|_{t_0} = \tau''|_{t_0} = \tau|_{t_0}$.

We now sketch the proof that $\tau_\psi \in [\![P]\!](s)$, appealing to the intuitive understanding of the programming language semantics. To this end, we show that τ' and then τ'' belong to $[\![P]\!](s)$. We start by analyzing how the

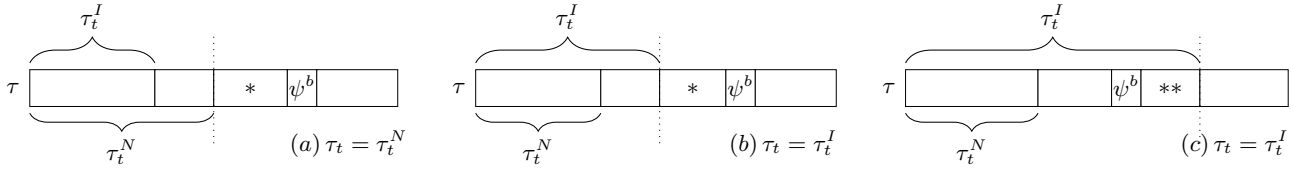


Figure 3: Cases in the proof of Lemma 9. * all actions by t are transactional; ** all actions by t come from a single transaction, started before or by ψ^b

trace $\tau|_t$ is truncated to τ_t for every thread $t \neq t_0$. Let us make a case split on the relative positions of τ_t^N, τ_t^I and ψ^b in τ. There are three cases, shown in Figure 3. Either $\tau_t = \tau_t^N$ (a, thread t_1 in Figure 2) or $\tau_t = \tau_t^I$ (b, c). In the former case, ψ^b has to come after the end of τ_t^N. In the latter case, either ψ^b comes after the end of τ_t^I (b) or is its last action or precedes the latter (c, thread t_2 in Figure 2).

By the choice of τ_t^N, in (a) and (b) the fragment of τ in between the end of τ_t^N and ψ^b can contain only those actions by t that are transactional (T_2 in Figure 2). By the choice of τ_t^I and ψ^b, in (c) the fragment of τ in between ψ^b and the end of τ_t^I cannot contain a $\mathsf{txbegin}$ action by t and, hence, can contain only those actions by t that are transactional. Furthermore, these have to come from a single transaction, started either by ψ^b or before it (T_4 in Figure 2). Finally, by the choice of ψ^b the actions of t_0 following ψ^b are transactional and come from the transaction of ψ, also started either by ψ^b or before it (T_5 in Figure 2). Given this analysis, the transformation from τ to τ' can be viewed as a sequence of two: (i) erase all actions following ψ^b, except those in some of transactions that were already ongoing at this time; (ii) erase some suffixes of threads containing only transactional actions. Since transactional actions do not access global variables, they are not affected by the actions of other threads. Furthermore, as we noted in Section 5, $[\![P]\!](s)$ includes incomplete program computations. This allows us to conclude that $\tau' \in [\![P]\!](s)$.

We now show that τ'' is valid, again referring to cases (a-c). Let $T = \mathsf{txof}(\psi^b, H_1\psi)$; then $T \in H_1'\psi$ by the choice of ψ^b and by Definition 3(iii) we get (1). Hence, for threads t falling into cases (a) or (b), $\tau'|_t$ may not contain aborted transactions that are also in $H_1'\psi$. For threads t falling into case (c), the only aborted transaction included into $H_1'\psi$ can be the last one in $\tau'|_t$. Finally, above we established that $(H_1'\psi)|_{t_0}$ does not contain aborted transactions. Hence, transactions in τ' whose status is changed from aborted to committed when switching to τ'' do not have any actions following them in τ'. Furthermore, $[\![P]\!](s)$ allows committing or aborting transactions arbitrarily. This allows us to conclude that $\tau'' \in [\![P]\!](s)$. For the same reason, we get $\tau_\psi \in [\![P]\!](s)$.

Finally, we show that $\mathsf{history}(\tau_\psi)|_{\neg\mathsf{abortedtx}} = H_\psi^c$. It is sufficient to show that $\mathsf{history}(\tau'')|_{\neg\mathsf{abortedtx}} = H_1''\psi$; since $\tau_\psi = \tau'' H^{cc}$ and H^{cc} contains only $\mathsf{committed}$ actions, this would imply

$$\mathsf{history}(\tau_\psi)|_{\neg\mathsf{abortedtx}} = \mathsf{history}(\tau'' H^{cc})|_{\neg\mathsf{abortedtx}} = \mathsf{history}(\tau'')|_{\neg\mathsf{abortedtx}} H^{cc} = H_1''\psi H^{cc} = H_\psi^c.$$

By the choice of τ_t^I for $t \neq t_0$, every transaction in $(H_1'\psi)|_t$ is also in τ_t^I. Hence, $H_1'\psi$ is a subsequence of $\mathsf{history}(\tau')$. By the definition of τ'' and H_1'', $H_1''\psi$ is a subsequence of $\mathsf{history}(\tau'')$. Then since $H_1''\psi$ does not contain aborted transactions, $H_1''\psi$ is a subsequence of $\mathsf{history}(\tau'')|_{\neg\mathsf{abortedtx}}$.

Thus, to prove $\mathsf{history}(\tau'')|_{\neg\mathsf{abortedtx}} = H_1''\psi$ it remains to show that every non-aborted transaction in $\mathsf{history}(\tau'')$ is in $H_1''\psi$. Since the construction of τ'' from τ' changes the status of only those transactions that belong to $H_1'\psi$, it is sufficient to show that every non-aborted transaction in $\mathsf{history}(\tau')$ is in $H_1'\psi$. Here we only consider the case when such a transaction is by a thread $t \neq t_0$ and $\tau'|_t = \tau_t^N \neq \varepsilon$; we cover the other cases in Appendix D. Let ϕ_t^N be the last action in τ_t^N and $T = \mathsf{txof}(\psi^b, H_1\psi) \in H_1'\psi$. Then by Definition 3(iii) we get (1). Since ϕ_t^N comes before ψ^b in $H_1\psi$, any transaction T' in $\tau'|_t$ is such that $T' \prec_{H_1\psi} T$, which together with (1) implies the required. This concludes the proof that $\mathsf{history}(\tau'')|_{\neg\mathsf{abortedtx}} = H_1''\psi$. ∎

We now give the other lemmas necessary for the proof. Definition 6 matches a history of $\mathcal{T}_C$ with one of $\mathcal{T}_A$ using the opacity relation, possibly after transforming the former with $\mathsf{cTMSpast}$. The following lemma is used to transform a trace of P accordingly. The lemma shows that, if we consider only traces where aborted

transactions abort immediately (i.e., are of the form $(\_, \_, \mathsf{txbegin})\ (\_, \_, \mathsf{aborted})$), then the opacity relation implies observational refinement with respect to observing non-transactional actions and thread-local trace projections. This result is a simple adjustment of the one about the sufficiency of opacity for observational refinement to our setting [1, Theorem 16] (it was proved in [1] for a language where local variables are *not* rolled back upon a transaction abort; this difference, however, does not matter if aborted transactions abort immediately).

LEMMA 10. *Consider $\tau \in [\![P]\!](s)$ such that all the aborted transactions in τ abort immediately. Let S be such that $\mathsf{history}(\tau) \sqsubseteq_{\mathsf{op}} S$. Then there exists $\tau' \in [\![P]\!](s)$ such that $\mathsf{history}(\tau') = S$, $\tau|_{\neg\mathsf{trans}} = \tau'|_{\neg\mathsf{trans}}$ and $\forall t.\, \tau'|_t = \tau|_t$.*

Let $\tau|_{\neg\mathsf{abortact}}$ be the trace obtained from τ by removing all actions inside aborted transactions, so that every such transaction aborts immediately. We can benefit from Lemma 10 because local variables are rolled back if a transaction aborts, and, hence, applying $\cdot|_{\neg\mathsf{abortact}}$ to a trace preserves its validity.

PROPOSITION 11. $\forall \tau.\, \tau \in [\![P]\!](s) \implies \tau|_{\neg\mathsf{abortact}} \in [\![P]\!](s)$.

Finally, Definition 6 matches only histories of committed transactions, but the histories of the traces in Lemma 10 also contain aborted transactions. Fortunately, the following lemma allows us to add empty aborted transactions into the abstract history while preserving the opacity relation.

LEMMA 12. *Let H be a history where all aborted transactions abort immediately and S be such that $H|_{\neg\mathsf{abortedtx}} \sqsubseteq_{\mathsf{op}} S$. There exists a history S' such that $S'|_{\neg\mathsf{abortedtx}} = S$ and $H \sqsubseteq_{\mathsf{op}} S'$.*

Using Definition 6(i), Proposition 11 and Lemmas 10 and 12, we can show that the TMS relation preserves non-transactional actions and thread-local observable behavior of threads whose last action is not a `fault`.

LEMMA 13. $\forall \tau \in [\![P, \mathcal{T}_C]\!](s).\, \exists \tau' \in [\![P, \mathcal{T}_A]\!](s).\, (\tau'|_{\neg\mathsf{trans}} = \tau|_{\neg\mathsf{trans}}) \wedge (\forall t.\, (\tau'|_t)|_{\mathsf{obs}} = (\tau|_t)|_{\mathsf{obs}})$.

PROOF OF THEOREM 8(I). Given Lemma 13, we only need to establish the preservation of faults inside transactions. Consider $\tau_0 \in [\![P, \mathcal{T}_C]\!](s)$ such that $\tau_0 = \tau_1 \psi \tau_2 \phi$, where $\phi = (\_, t_0, \mathtt{fault})$ is transactional and ψ is the last TM interface action by thread t_0. Then $\tau_2|_{t_0}$ consists of transactional actions and thus does not contain accesses to global variables. Hence, $\tau = \tau_1 \psi (\tau_2|_{t_0}) \phi \in [\![P, \mathcal{T}_C]\!](s)$. By our assumption, $\mathcal{T}_C \sqsubseteq_{\mathsf{tms}} \mathcal{T}_A$. Then there exists $H^c_\psi \in \mathsf{cTMSpast}(\mathsf{history}(\tau))$ and $S \in \mathcal{T}_A$ such that $H^c_\psi \sqsubseteq_{\mathsf{op}} S$. By Lemma 9, for some trace τ_ψ we have $\tau_\psi \in [\![P]\!](s)$, $\mathsf{history}(\tau_\psi)|_{\neg\mathsf{abortedtx}} = H^c_\psi$ and $\tau_\psi|_{t_0} = \tau|_{t_0}$. By Proposition 11, $\tau_\psi|_{\neg\mathsf{abortact}} \in [\![P]\!](s)$. Using Lemma 12, we get a history S' such that $\mathsf{history}(\tau_\psi|_{\neg\mathsf{abortact}}) \sqsubseteq_{\mathsf{op}} S'$ and $S'|_{\neg\mathsf{abortedtx}} = S$. Since $S \in \mathcal{T}_A$ and $\mathcal{T}_A$ is closed under immediate aborts, we get $S' \in \mathcal{T}_A$. Hence, by Lemma 10, for some $\tau' \in [\![P, \mathcal{T}_A]\!](s)$ we have $\tau'|_{t_0} = \tau_\psi|_{t_0} = \tau|_{t_0} = \_\phi$, as required. ∎

6.2 Proof Sketch for Theorem 8(ii) (Necessity)

Consider $\mathcal{T}_C$ and $\mathcal{T}_A$ such that $\mathcal{T}_C \preceq \mathcal{T}_A$ and $\mathcal{T}_A$ satisfies the closure conditions stated in the theorem. To show that for any $H_0 \in \mathcal{T}_C$ we have $H_0 \sqsubseteq_{\mathsf{tms}} \mathcal{T}_A$, we have to establish conditions (i) and (ii) from Definition 6. We sketch the more interesting case of (ii) (cf. Lemma 17), in which $H_0 = H_1 \psi H_2 = H H_2 \in \mathcal{T}_C$, where ψ is a response action by a thread t_0. We assume that ψ is not a committed or aborted action: the full proof considers the case of aborted actions separately (Lemma 18). We need to find $H^c \in \mathsf{cTMSpast}(H)$ and $S \in \mathcal{T}_A$ such that $H^c \sqsubseteq_{\mathsf{op}} S$.

To this end, we construct a program P_H (as we explain further below) where every thread t performs the sequence of transactions specified in $H|_t$. The program monitors certain properties of the TM behavior, e.g., checking that the return values obtained from methods of transactional objects in committed transactions correspond to those in H and that the real-time order between actions includes that in H. If these properties hold, thread t_0 ends by executing the `fault` command. Let s be a state with all variables set to distinguished values. We next construct a trace $\tau \in [\![P_H, \mathcal{T}_C]\!](s)$ such that $\mathsf{history}(\tau) = H$ and t_0 faults in τ. By Definition 7, there exists $\tau' \in [\![P_H, \mathcal{T}_A]\!](s)$ such that t_0 faults in τ'. However, the program P_H is constructed so that t_0 can fault in τ' only if the properties of the TM behaviour the program monitors hold, and thus H is related to $\mathsf{history}(\tau')$ in a certain way. This relationship allows us to construct $H^c \in \mathsf{cTMSpast}(H)$ from H and $S \in \mathcal{T}_A$ from $\mathsf{history}(\tau')$ such that $H^c \sqsubseteq_{\mathsf{op}} S$.

In more detail, thread t_0 in P_H monitors the return status of every transaction and the return values obtained inside the atomic blocks corresponding to transactions committed in $H|_{t_0}$ and the (live) transaction of ψ. If there

is a mismatch with $H|_{t_0}$, this is recorded in a special local variable. At the end of the transaction of ψ, t_0 checks the variable and faults if the TM behavior matched $H|_{t_0}$. This construction is motivated by the fact that faulting is the only observation Definition 7 allows us to make about the behavior of the live transaction of ψ. Since the definition does not correlate actions by threads t other than t_0 between τ and τ', such threads monitor TM behavior differently: if there is a mismatch with $H|_t$, a thread t faults immediately. Since a trace can have at most one `fault` and t_0 faults in τ', this ensures that any committed transaction in τ' behaves as in H.

To check whether an execution of P_H complies with the real-time order in H, for each transaction in H, we introduce a global variable g, which is initially 0 and is set to 1 by the thread executing the transaction right after the transaction completes, by a command following the corresponding atomic block. Before starting a transaction, each thread checks whether all transactions preceding this one in the real-time order in H have finished by reading the corresponding g variables. Thread t_0 records the outcome in the special local variable checked at the end; all other threads fault upon detecting a mismatch.

This construction of P_H allows us to infer that: (i) the projection of $\mathsf{history}(\tau')|_{t_0}$ to committed transactions and $\mathsf{txof}(\varphi, \tau')$ is equal to the corresponding projection of $H|_{t_0}$; (ii) for all other threads t a similar relationship holds for the prefix of $\mathsf{history}(\tau')|_t$ ending with the last transaction preceding $\mathsf{txof}(\varphi, \tau')$ in the real-time order; (iii) the real-time order in $\mathsf{history}(\tau')$ includes that in H. Transactions concurrent with $\mathsf{txof}(\varphi, \tau')$ in τ' may behave differently from H. However, checks done by P_H inside these transactions ensure that, if such a transaction T is visible in τ', then the return values inside T match those in H. The checks on the global variables g done right before T also ensure that all transactions preceding T in the real-time order in H commit or abort in τ' as prescribed by H. This relationship between H and $\mathsf{history}(\tau')$ allows us to establish the requirements of Definition 6(ii). ∎

7 Related Work

When presenting TMS [3], Doherty et al. discuss why it allows programmers to think only of serial executions of their programs, in which the actions of a transaction appear consecutively. This discussion—corresponding to our sufficiency result—is informal, since the paper lacks a formal model for programs and their semantics. Most of it explains how Definition 6(i) ensures the correctness of committed transactions. The discussion of the most challenging case of live transactions—corresponding to Definition 6(ii) and our Lemma 9—is one paragraph long. It only roughly sketches the construction of a trace with an abstract history allowed by TMS and does not give any reasoning for why this trace is a valid one, but only claims that constraints in Definition 6(ii) ensure this. This reasoning is very delicate, as indicated by our proof of Lemma 9, which carefully selects which actions to erase when transforming the trace. Moreover, Doherty et al. do not try to argue that TMS is the weakest condition possible, as we established by our necessity result.

Another TM consistency condition, weaker than opacity but incomparable to TMS, is *virtual world consistency* (*VWC*) [10]. Like TMS, VWC allows every operation in a live or aborted transaction to be explained by a separate abstract history. However, it places different constraints on the choice of abstract histories, which do not take into account the real-time order between actions. Because of this, VWC does not imply observational refinement for our programming language: taking into account the real-time order is necessary when threads can communicate via global variables outside transactions.

Our earlier paper [1] has laid the groundwork for relating TM consistency and observational refinement, and it includes a detailed comparison with related work on opacity and observational refinement. The present paper considers a much more challenging case of a language where local variables are rolled back upon an abort. To handle this case, we have developed new techniques, such as establishing the live transaction insensitivity property (Lemma 9) to prove sufficiency and proposing monitor programs for the nontrivial constraints used in the TMS definition to prove necessity. Similarly to [1] and other papers using observational refinement to study consistency conditions [4, 5], we reformulate TMS so that it is not restricted to an abstract TM $\mathcal{T}_A$ that executes transactions atomically. This generality, not allowed by the original TMS definition, has two benefits. First, our reformulation can be used to compare two TM implementations, e.g., an optimized and an unoptimized one. Second, dealing with the general definition forces us to explicitly state the closure properties required from the abstract TM, rather than having them follow implicitly from its atomic behavior.

References

[1] H. Attiya, A. Gotsman, S. Hans, and N. Rinetzky. A programming language perspective on transactional memory consistency. In *PODC*, pages 309–318, 2013.

[2] H. Attiya, S. Hans, P. Kuznetsov, and S. Ravi. Safety of deferred update in transactional memory. In *ICDCS*, 2013.

[3] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, 25(5):769–799, 2013.

[4] I. Filipovic, P. W. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. In *ESOP*, pages 252–266, 2009.

[5] A. Gotsman and H. Yang. Liveness-preserving atomicity abstraction. In *ICALP (2)*, pages 453–465, 2011.

[6] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPOPP*, pages 175–184, 2008.

[7] J. He, C. Hoare, and J. Sanders. Data refinement refined. In *ESOP*, pages 187–196, 1986.

[8] J. He, C. Hoare, and J. Sanders. Prespecification in data refinement. *Information Processing Letters*, 25(2):71 – 76, 1987.

[9] M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[10] D. Imbs and M. Raynal. Virtual world consistency: A condition for STM systems (with a versatile protocol with invisible read operations). *Theor. Comput. Sci.*, 444:113–127, 2012.

[11] M. Lesani, V. Luchangco, and M. Moir. Putting opacity in its place. In *WTTM*, 2012.

[12] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26:631–653, October 1979.

[13] Scala STM Expert Group. Scala STM quick start guide, 2012. http://nbronson.github.io/scala-stm/quick_start.html.

A Restrictions on Variable Accesses

To formalise restrictions on accesses to variables by primitive commands, we partition the set $\mathsf{PComm} - \{\texttt{fault}\}$ into $2m$ classes: $\mathsf{PComm} - \{\texttt{fault}\} = \biguplus_{t=1}^{m}(\mathsf{LPcomm}_t \uplus \mathsf{GPcomm}_t)$. The intention is that commands from LPcomm_t can access only the local variables of thread t (LVar_t); commands from GPcomm_t can additionally access global variables ($\mathsf{LVar}_t \uplus \mathsf{GVar}$). We formalize these restrictions in Appendix C. To ensure that a thread t does not access local variables of other threads, we require that the thread cannot mention such variables in the conditions of `if` and `while` commands and can only use primitive commands from $\mathsf{LPcomm}_t \uplus \mathsf{GPcomm}_t$.

B Formal Definition of Traces

DEFINITION 14 (Traces). *A **trace** τ is a finite sequence of actions, satisfying the following conditions:*

(i) every action in τ has a unique identifier: if $\tau = \tau_1(a_1, \_, \_)\tau_2(a_2, \_, \_)\tau_3$ then $a_1 \neq a_2$.

(ii) no action follows a `fault`*: if $\tau = \tau'\varphi$ then τ' does not contain a* `fault` *action.*

(iii) request and response actions are properly matched: for every thread t, $\mathsf{history}(\tau)|_t$ consists of alternating request and corresponding response actions, starting from a request action;

(iv) actions denoting the beginning and end of transactions are properly matched: for every thread t, in the projection of $\mathsf{history}(\tau)|_t$ to txbegin*,* committed *and* aborted *actions,* txbegin *alternates with* committed *or* aborted*, starting from* txbegin*;*

(v) call *and* ret *actions occur only inside transactions: for every thread t, if $\tau|_t = \tau_1\psi\tau_2$ for a* call *or* ret *action ψ, then $\tau_1 = \tau_1'\psi'\tau_1''$ for some* txbegin *action ψ', and τ_1' and τ_1'' such that τ_1'' does not contain* committed *or* aborted *actions;*

(vi) commands in τ do not access local variables of other threads: if $(\_, t, c) \in \tau$ then $c \in \mathsf{LPcomm}_t \uplus \mathsf{GPcomm}_t \uplus \{\texttt{fault}\}$;

(vii) commands in τ do not access global variables inside a transaction: if $\tau = \tau_1\,(\_, t, c)\,\tau_2$ for $c \in \mathsf{GPcomm}_t$, then it is not the case that $\tau_1 = \tau_1'\,(\_, t, \mathsf{txbegin})\,\tau_1''$, where τ_1'' does not contain committed *or* aborted *actions.*

C Formal Definition of the Semantics of the Programming Language

This section formally defines the set $[\![P]\!](s)$. It is computed in two stages. First, we compute a set $A(P)$ of traces that resolves all issues regarding sequential control flow and interleaving. Intuitively, if one thinks of each thread C_t in P as a control-flow graph, then $A(P)$ contains all possible interleavings of paths in the graphs of C_t, $t \in \mathsf{ThreadID}$ starting from their initial nodes. The set $A(P)$ is a superset of all the traces that can actually be executed: e.g., if a thread executes the command "$x := 1;$ `if` $(x = 1)$ $y := 1$ `else` $y := 2$" where x, y are local variables, then $A(P)$ will contain a trace where $y := 2$ is executed instead of $y := 1$. To filter out such nonsensical traces, we *evaluate* every trace to determine whether it is ***valid***, i.e., whether its control flow is consistent with the effect of its actions on program variables. This is formalized by a function $\mathsf{eval} : \mathsf{State} \times \mathsf{Trace} \to \mathcal{P}(\mathsf{State}) \cup \{\lightning\}$ that, given an initial state and a trace, produces the set of states resulting from executing the actions in the trace, an empty set if the trace is invalid, or a special state $\lightning$ if the trace contains a `fault` action. Thus, $[\![P]\!](s) = \{\tau \in A(P) \mid \mathsf{eval}(s, \tau) \neq \emptyset\}$.

When defining the semantics, we encode the evaluation of conditions in `if` and `while` statements with assume commands. More specifically, we expect that the sets LPcomm_t contain special primitive commands $\mathsf{assume}(b)$, where b is a Boolean expression over local variables of thread t, defining the condition. We state their semantics formally below; informally, $\mathsf{assume}(b)$ does nothing if b holds in the current program state, and stops the computation otherwise. Thus, it allows the computation to proceed only if b holds. The assume commands are only used in defining the semantics of the programming language; hence, we forbid threads from using them directly.

The trace set $A(P)$. The function $A'(\cdot)$ in Figure 4 maps commands and programs to sequences of actions they may produce. Technically, $A'(\cdot)$ might contain sequences that are not traces, e.g., because they do not have unique identifiers or continue beyond a `fault` command. This is resolved by intersecting the set $A'(P)$ with the set of all traces to define $A(P)$. $A'(C)t$ gives the set of action sequences produced by a command C when it is executed

$$
\begin{array}{ll}
A'(c)t & = \{(\_, t, c)\} \\
A'(C_1; C_2)t & = \{\tau_1\,\tau_2 \mid \tau_1 \in A'(C_1)t \wedge \tau_2 \in A'(C_2)t\} \\
A'(\texttt{if}\ (b)\ \texttt{then}\ C)t & = \{(\_, t, \mathsf{assume}(b))\,(A'(C)t)\} \\
A'(\texttt{if}\ (b)\ \texttt{then}\ C_1\ \texttt{else}\ C_2)t & = \{(\_, t, \mathsf{assume}(b))\,\tau_1 \mid \tau_1 \in A'(C_1)t\} \cup \{(\_, t, \mathsf{assume}(\neg b))\,\tau_2 \mid \tau_2 \in A'(C_2)t\} \\
A'(\texttt{while}\ (b)\ \texttt{do}\ C)t & = \{((\_, t, \mathsf{assume}(b))\,(A'(C)t))^*\,(\_, t, \mathsf{assume}(\neg b))\} \\
A'(x := o.f(e))t & = \{(\_, t, \mathsf{assume}(e = n))\,(\_, t, \mathsf{call}\ o.f(n))\,(\_, t, \mathsf{ret}(n')\ o.f)\,(\_, t, x := n') \mid n, n' \in \mathbb{Z}\} \cup \\
 & \quad \{(\_, t, \mathsf{assume}(e = n))\,(\_, t, \mathsf{call}\ o.f(n))\,(\_, t, \mathsf{aborted}) \mid n \in \mathbb{Z}\} \\
A'(x := \texttt{atomic}\,\{C\})t & = \{(\_, t, \mathsf{txbegin})\,(\_, t, \mathsf{aborted})\,(\_, t, x := \mathsf{aborted})\} \cup \\
 & \{(\_, t, \mathsf{txbegin})\,(\_, t, \mathsf{OK})\,\tau\,(\_, t, \mathsf{aborted})\,(\_, t, x := \mathsf{aborted}) \mid \tau\,(\_, t, \mathsf{aborted})\,\tau' \in A'(C)t \wedge (\_, t, \mathsf{aborted}) \notin \tau\} \cup \\
 & \{(\_, t, \mathsf{txbegin})\,(\_, t, \mathsf{OK})\,\tau\,(\_, t, \mathsf{txcommit})\,(\_, t, r)\,(\_, t, x := r) \mid \tau \in A'(C)t \wedge \\
 & \qquad\qquad (\_, t, \mathsf{aborted}) \notin \tau \wedge (r = \mathsf{committed} \vee r = \mathsf{aborted})\} \\
A'(C_1 \parallel \ldots \parallel C_m) & = \mathsf{prefix}(\bigcup\{\mathsf{interleave}(\tau_1, \ldots, \tau_m) \mid \forall t.\, 1 \le t \le m \implies \tau_t \in A'(C_t)t\}) \\
A(P) & = A'(P) \cap \mathsf{Trace}
\end{array}
$$

Figure 4: The definition of $A(P)$.

by thread t. To define $A'(P)$, we first compute the set of all the interleavings of action sequences produced by the threads constituting P. Formally, $\tau \in \mathsf{interleave}(\tau_1, \ldots, \tau_m)$ if and only if every action in τ is performed by some thread $t \in \{1, \ldots, m\}$, and $\tau|_t = \tau_t$ for every thread $t \in \{1, \ldots, m\}$. We then let $A'(P)$ be the set of all prefixes of the resulting sequences, as denoted by the prefix operator. We take prefix closure here to account for incomplete program computations as well as those in which the scheduler preempts a thread forever.

$A'(c)t$ returns a singleton set with the action corresponding to the primitive command c (primitive commands execute atomically). $A'(C_1; C_2)t$ concatenates all possible action sequences corresponding to C_1 with those corresponding to C_2. The set of action sequences of a conditional considers cases where either branch is taken. We record the decision using an assume action; at the evaluation stage, this allows us to ensure that this decision is consistent with the program state. The set of action sequences for a loop is defined using the Kleene closure operator * to produce all possible unfoldings of the loop body. Again, we record branching decisions using assume actions.

The set of action sequences of a method invocation $x := f(e)$ includes both sequences where the method executes successfully and where the current transaction is aborted. The former set is constructed by nondeterministically choosing two integers n and n' to describe the parameter n and the return value n' for the method call. To ensure that e indeed evaluates to n, we insert $\mathsf{assume}(e = n)$ before the call action, and to ensure that x gets the return value n', we add the assignment $x := n'$ after the ret action. Note that some of the choices here might not be feasible: the chosen n might not be the value of the parameter expression e when the method is invoked, or the method might never return n' when called with n. Such infeasible choices are filtered out at the following stages of the semantics definition: the former in the definition of $[\![P]\!](s)$ by the use of evaluation and the semantics of assume, and the latter in the definition of $[\![P, \mathcal{T}]\!](s)$ by selecting the sequences from $[\![P]\!](s)$ that interact with the transactional memory correctly. The set of action sequences of $x := \texttt{atomic}\,\{C\}$ contains those in which C is aborted in the middle of its execution (at an object operation or right after it begins) and those in which C executes until completion and then the transaction commits or aborts.

Semantics of primitive commands. To define evaluation, we assume a semantics of every command $c \in \mathsf{PComm} - \{\texttt{fault}\}$, given by a function $[\![c]\!]$ that defines how the program state is transformed by executing c. As we noted before, different classes of primitive commands are supposed to access only certain subsets of variables. To ensure that this is indeed the case, we define $[\![c]\!]$ as a function of only those variables that c is allowed to access. Namely, the semantics of $c \in \mathsf{LPcomm}_t$ is given by

$$[\![c]\!] : (\mathsf{LVar}_t \to \mathbb{Z}) \to \mathcal{P}(\mathsf{LVar}_t \to \mathbb{Z}).$$

The semantics of $c \in \mathsf{GPcomm}_t$ is given by

$$[\![c]\!] : ((\mathsf{LVar}_t \uplus \mathsf{GVar}) \to \mathbb{Z}) \to \mathcal{P}((\mathsf{LVar}_t \uplus \mathsf{GVar}) \to \mathbb{Z}).$$

Note that we allow c to be non-deterministic.

For a valuation q of variables that c is allowed to access, $[\![c]\!](q)$ yields the set of their valuations that can be obtained by executing c from a state with variable values q. For example, an assignment command $x := g$ has the following semantics:

$$[\![x := g]\!](q) = \{q[g \mapsto q(g)]\} \,.$$

We define the semantics of assume commands following the informal explanation given at the beginning of this section: for example,

$$[\![\mathsf{assume}(x = n)]\!](q) = \begin{cases} \{q\}, & \text{if } q(x) = n; \\ \emptyset, & \text{otherwise.} \end{cases} \tag{2}$$

Thus, when the condition in assume does not hold of q, the command stops the computation by not producing any output.

We lift functions $[\![c]\!]$ to full states by keeping the variables that c is not allowed to access unmodified and producing $\lightning$ if c faults. For example, if $c \in \mathsf{LPcomm}_t$, then

$$[\![c]\!](s) = \{s|_{\mathsf{LVar} \setminus \mathsf{LVar}_t} \uplus q \mid q \in [\![c]\!](s|_{\mathsf{LVar}_t})\},$$

where $s|_V$ is the restriction of s to variables in V. Finally, we let

$$[\![\texttt{fault}]\!](s) = \lightning \,,$$

so that the only way a program can fault is by executing the `fault` command.

Trace evaluation. Using the semantics of primitive commands, we first define the evaluation of a single action on a given state:

$$\begin{aligned} &\mathsf{eval} : \mathsf{State} \times \mathsf{Action} \to \mathcal{P}(\mathsf{State}) \cup \{\lightning\} \\ &\mathsf{eval}(s, (\_, t, c)) \;=\; [\![c]\!](s); \\ &\mathsf{eval}(s, \psi) \;=\; \{s\}. \end{aligned}$$

Note that this does not change the state s as a result of TM interface actions, since their return values are assigned to local variables by separate actions introduced when generating $A(P)$. We then lift eval to traces as follows:

$$\begin{aligned} &\mathsf{eval} : \mathsf{State} \times \mathsf{Trace} \to \mathcal{P}(\mathsf{State}) \cup \{\lightning\} \\ &\mathsf{eval}(s, \tau) = \begin{cases} \emptyset, & \text{if } \tau = \tau'\varphi \wedge \mathsf{eval}(s, \tau') = \emptyset; \\ \mathsf{evalna}(s, \tau|_{\neg \mathsf{abortact}}), & \text{otherwise,} \end{cases} \end{aligned}$$

where

$$\mathsf{evalna}(s, \tau) = \begin{cases} \{s\}, & \text{if } \tau = \varepsilon; \\ \{s'' \in \mathsf{eval}(s', \varphi) \mid s' \in \mathsf{evalna}(s, \tau')\}, & \text{if } \tau = \tau'\varphi. \end{cases}$$

The set of states resulting from evaluating trace τ from state s is effectively computed by the helper function $\mathsf{evalna}(s, \tau)$, which ignores actions inside aborted transactions to model local variable roll-back. However, ignoring the contents of aborted transactions completely poses a risk that we might consider traces including sequences of actions inside aborted transactions that yield an empty set of states. To mitigate this, $\mathsf{eval}(s, \tau)$ recursively evaluates every prefix of τ, thus ensuring that sequences of actions inside aborted transaction are valid.

As we explained in Section 5, we define $[\![P]\!](s)$ as the set of those traces from $A(P)$ that can be evaluated from s without getting stuck, as formalized by eval. Note that this definition enables the semantics of assume defined by (2) to filter out traces that make branching decisions inconsistent with the program state. For example, consider again the program "$x := 1$; `if` $(x = 1)$ $y := 1$ `else` $y := 2$". The set $A(P)$ includes traces where both branches are explored. However, due to the semantics of the assume actions added to the traces according to Figure 4, only the trace executing $y := 1$ will result in a non-empty set of final states after the evaluation and, therefore, only this trace will be included into $[\![P]\!](s)$.

D Additional Proofs

D.1 Remaining Cases from the Proof of Lemma 9

- $t \neq t_0$ is such that $\tau'|_t = \tau^I_t \neq \varepsilon$. Let ψ^I_t be the last action in τ^I_t. Let $T = \mathsf{txof}(\psi^I_t, H_1\psi)$. By the choice of τ^I_t we have $T \in H'_1\psi$; then by Definition 3(iii) we get (1). Since any transaction T' in $\mathsf{history}(\tau'|_t)$ is either T or is such that $T' \prec_{(H_1\psi)|_t} T$, this implies the required.
- $t = t_0$. Let $T = \mathsf{txof}(\psi, H_1\psi) \in H'_1\psi$. Then by Definition 3(iii) we get (1). Since any transaction T' in $\mathsf{history}(\tau'|_{t_0})$ is either T or is such that $T' \prec_{(H_1\psi)|_{t_0}} T$, this implies the required.

D.2 Proof of Lemma 12

Let n be the number of aborted transactions in H. To construct the desired S', we inductively construct a sequence of histories S_i, $i = 0..n$ such that

$$|\mathsf{aborted}(S_i)| = i; \qquad S_i|_{\neg\mathsf{abortedtx}} = S; \qquad \{\psi \mid \psi \in S_i\} \subseteq \{\psi \mid \psi \in H\}; \\ \forall \psi_1, \psi_2 \in S_i.\, \psi_1 \prec_H \psi_2 \implies \psi_1 \prec_{S_i} \psi_2. \tag{3}$$

We then let $S' = S_n$, so that $H \sqsubseteq_{\mathsf{op}} S'$.

For $i = 0$, we take $S_0 = S$, and all the requirements in (3) hold vacuously. Assume a history S_i satisfying (3) was constructed; we get S_{i+1} from S_i by the following construction. Let $H = H_1\psi_b H_2 \psi_a H_3$, where $\psi_b = (\_, t, \mathsf{txbegin})$, $\psi_a = (\_, t, \mathsf{aborted})$, $H_2|_t = \varepsilon$ and

$$\neg\exists \psi'.\, \psi' = (\_, \_, \mathsf{txbegin}) \in H_1 \wedge \mathsf{txof}(\psi', H) \in \mathsf{aborted}(H) \wedge \psi' \notin S_i.$$

That is, out of all aborted transactions in H that are not in S_i, $\psi_b\psi_a$ is the one with the earliest txbegin. We now consider two cases.

Case I: H_1 does not contain a committed or an aborted action.

In this case, let $S_{i+1} = \psi_b\psi_a S_i$. We only need to show that for any $\psi' \in S_i$ we have $\psi' \prec_H \psi_b \implies \psi' \prec_{S_{i+1}} \psi_b$ and $\psi_a \prec_H \psi' \implies \psi_a \prec_{S_{i+1}} \psi'$. The latter holds by the construction of S_{i+1}. To show the former, observe that, since H_1 does not contain a committed or aborted action, it cannot contain actions by thread t. Hence, we cannot have $\psi' \prec_H \psi_b$ for any ψ'.

Case II: H_1 contains a committed or an aborted action.

Let ψ be the last committed or aborted action in S_i that is also in H_1 and let $S_i = S'\psi S''$. We then let $S_{i+1} = S'\psi\psi_b\psi_a S''$. We again need to show that for any $\psi' \in S_i$ we have $\psi' \prec_H \psi_b \implies \psi' \prec_{S_{i+1}} \psi_b$ and $\psi_a \prec_H \psi' \implies \psi_a \prec_{S_{i+1}} \psi'$.

Assume $\psi' \prec_H \psi_b$ for some $\psi' \in S_i$; then $\psi' \in H_1$. By the choice of ψ_b and ψ_a, all the committed and aborted actions in H_1 are in S_i, and by the choice of ψ, all such actions are in $S'\psi$. Hence, if ψ' is a committed or an aborted action, then $\psi' \in S'\psi$ and, hence, $\psi' \prec_{S_{i+1}} \psi_b$. If ψ' is by thread t, then it is either a committed or an aborted action (and, hence, $\psi' \prec_{S_{i+1}} \psi_b$) or it precedes such an action $\psi'' \in S_i$ by t in H_1: $\psi' \prec_{H_1} \psi''$. Then $\psi' \prec_{S_{i+1}} \psi''$ and $\psi'' \prec_{S_{i+1}} \psi_b$, which implies $\psi' \prec_{S_{i+1}} \psi_b$.

Now assume $\psi_a \prec_H \psi'$ for some $\psi' \in S_i$; then $\psi' \in H_3$. If ψ' is a txbegin action, then $\psi \prec_H \psi'$. Hence, $\psi \prec_{S_i} \psi'$, i.e., $\psi' \in S''$, which implies $\psi_a \prec_{S_{i+1}} \psi'$. If ψ' is by thread t, then it is either a txbegin action (and, hence, $\psi_a \prec_{S_{i+1}} \psi'$) or it follows such an action $\psi'' \in S_i$ by thread t in H_3: $\psi'' \prec_{H_3} \psi'$. Then $\psi'' \prec_{S_{i+1}} \psi'$ and $\psi_a \prec_{S_{i+1}} \psi''$, which implies $\psi_a \prec_{S_{i+1}} \psi'$.

■

D.3 Proof of Lemma 13

Let $H = \mathsf{history}(\tau)$. By assumption, $\mathcal{T}_C \sqsubseteq_{\mathsf{tms}} \mathcal{T}_A$. Hence, there exist histories $H^c \in \mathsf{comp}(H|_{\neg\mathsf{live}})$ and $S \in \mathcal{T}_A$ such that $H^c|_{\mathsf{com}} \sqsubseteq_{\mathsf{op}} S$. Then $H^c = (H|_{\neg\mathsf{live}})H'$ for some H'. Let τ^c be the trace obtained from τ in the same way as H^c is obtained from H: $\tau^c = \tau_0 H'$, where τ_0 is obtained from τ by discarding all live transactions; thus, $\mathsf{history}(\tau^c) = H^c$. It is easy to see that $\tau^c \in [\![P]\!](s)$. Besides, $\tau^c|_{\neg\mathsf{trans}} = \tau|_{\neg\mathsf{trans}}$ and

$(\tau|_t)|_{\mathsf{obs}}$ is a prefix of $(\tau^c|_t)|_{\mathsf{obs}}$ for any t. Let $\tau^{na} = \tau^c|_{\neg\mathsf{abortact}}$. By Proposition 11 we get $\tau^{na} \in [\![P]\!](s)$. Since $(H^c|_{\neg\mathsf{abortact}})|_{\neg\mathsf{abortedtx}} = H^c|_{\mathsf{com}} \sqsubseteq_{\mathsf{op}} S$, by Lemma 12, for some history S' we have $\mathsf{history}(\tau^{na}) = H^c|_{\neg\mathsf{abortact}} \sqsubseteq_{\mathsf{op}} S'$ and $S'|_{\neg\mathsf{abortedtx}} = S$. Since $\mathcal{T}_A$ is closed under immediate aborts and $S \in \mathcal{T}_A$, we have $S' \in \mathcal{T}_A$. We have $\tau^{na} \in [\![P]\!](s)$; hence, by Lemma 10, there exists a trace $\tau'' \in [\![P]\!](s)$ such that $\mathsf{history}(\tau'') = S' \in \mathcal{T}_A$,

$$\tau''|_{\neg\mathsf{trans}} = \tau^{na}|_{\neg\mathsf{trans}} = \tau^c|_{\neg\mathsf{trans}} = \tau|_{\neg\mathsf{trans}},$$

and $\tau''|_t = \tau^{na}|_t$ for any t. Let τ' be the history obtained from τ'' by discarding the actions in H^c, which are last by the corresponding threads. Then

$$\tau'|_{\neg\mathsf{trans}} = \tau''|_{\neg\mathsf{trans}} = \tau|_{\neg\mathsf{trans}},$$

$\tau' \in [\![P]\!](s)$ and, since $\mathcal{T}_A$ is closed under removing transaction responses, $\mathsf{history}(\tau') \in \mathcal{T}_A$. Given $\tau''|_t = \tau^{na}|_t$, it is also easy to check that $(\tau'|_t)|_{\mathsf{obs}} = (\tau|_t)|_{\mathsf{obs}}$, as required. ∎

D.4 Proof of Theorem 8(ii) (Necessity)

Let $\tau\downarrow_i$ denote the prefix of a trace τ containing i actions and let $\tau\downarrow_\varphi$ denote the prefix of τ until (and including) the action φ.

DEFINITION 15. *Two traces τ and τ' are **equivalent up to identifiers**, denoted $\tau \equiv \tau'$, if $|\tau| = |\tau'|$ and for every $i = 1 \ldots |\tau|$, actions $\tau(i)$ and $\tau'(i)$ may differ only in their action identifiers.*

LEMMA 16. *Let $\mathcal{T}_C$ and $\mathcal{T}_A$ be transactional memories such that $\mathcal{T}_C \preceq \mathcal{T}_A$, and $\mathcal{T}_A$ is closed under removing live and aborted transactions and commit-pending transactions completion. Let H be a history in $\mathcal{T}_C$. There exists a history $H^c \in \mathsf{comp}(H|_{\neg\mathsf{live}})$ and a history $S \in \mathcal{T}_A$, such that $H^c|_{\mathsf{com}} \sqsubseteq_{\mathsf{op}} S$.*

Proof: Let us choose an integer value $u \neq 1$ which does not appear in H. We use the following shorthands:

- We let m be the largest thread identifier occurring in H.
- We denote by k^t the number of transactions started by thread t in H, i.e., the number of $(\_, t, \mathsf{txbegin})$ actions in H.
- We partition $H|_t$ into k^t subsequences: $H|_t = H^t_1 \ldots H^t_{k^t}$, where H^t_i is comprised of the actions in the i-th transaction of t. Specifically, $H^t_i(1) = (\_, t, \mathsf{txbegin})$.
- We let c^t_i be the outcome of the i-th transaction of thread t, i.e., $c^t_i = \mathsf{committed}$ or $c^t_i = \mathsf{aborted}$. If the transaction is not completed, c^t_i is undefined.
- We denote by q^t_i the number of call actions of thread t in its i-th transaction, i.e., in H^t_i.
- We let $(\_, t, \mathsf{call}\ o^t_{i,j}.f^t_{i,j}(n^t_{i,j}))$ be the j-th call action of thread t in its i-th transaction.
- We let $(\_, t, \mathsf{ret}(r^t_{i,j})\ o^t_{i,j}.f^t_{i,j})$ be the j-th ret action of thread t in its i-th transaction. If the response to $(\_, t, \mathsf{call}\ o^t_{i,j}.f^t_{i,j}(n^t_{i,j}))$ is an aborted action, we let $r^t_{i,j} = \mathsf{aborted}$. If there is no response to $(\_, t, \mathsf{call}\ o^t_{i,j}.f^t_{i,j}(n^t_{i,j}))$, i.e., the transaction is live and $(\_, t, \mathsf{call}\ o^t_{i,j}.f^t_{i,j}(n^t_{i,j}))$ is its last action, then we let $r^t_{i,j} = u$.
- We denote by $\mathsf{lasttx}(t, i, t')$ the number of transactions of thread t' in H that either committed or aborted before the i-th transaction of thread t started, i.e., the number of $(\_, t', \mathsf{committed})$ and $(\_, t', \mathsf{aborted})$ actions preceding the i-th $(\_, t, \mathsf{txbegin})$ action in H.

For every thread $t = 1..m$ we construct a straight-line command

$$C^t_H = GP^t_1; CP^t_1; GP^t_2; CP^t_2; \ldots; GP^t_{k^t}; CP^t_{k^t},$$

where GP^t_i and CP^t_i are defined in Figure 5. Here the g^t_i variables are global and all others are local. The g^t_i variables are used to monitor the real-time order: g^t_i is written only by thread t and is used to signal that the i-th transaction of thread t ended. The variable $z^t_{i,t'}$ is used to record whether the $\mathsf{lasttx}(t, i, t')$-th transaction of thread t' signaled that it had ended before the i-th transaction of thread t started. As $\mathsf{lasttx}(t, i, t')$ might be 0, we add a dummy variable g^t_0 for every thread t. Later in the proof we execute the program from a state in which g^t_0 is initialised to 1. The variable w^t_i records whether the i-th transaction of thread t committed or aborted. The variables $y^t_{i,j}$ record the return value of the j-th object method invocation in the i-th transaction of thread t.

$$GP_i^t \;=\; z_{i,1}^t := g^1_{\mathsf{lasttx}(t,i,1)}; \texttt{if}(z_{i,1}^t \neq 1)\ \texttt{then fault};$$
$$\ldots$$
$$z_{i,m}^t := g^m_{\mathsf{lasttx}(t,i,m)}; \texttt{if}(z_{i,m}^t \neq 1)\ \texttt{then fault};$$

- If H_i^t is a committed or aborted transaction, then

$$\begin{aligned} CP_i^t \;=\; & w_i^t := \texttt{atomic}\,\{\, y_{i,1}^t := o_{i,1}^t.f_{i,1}^t(n_{i,1}^t);\ \texttt{if}(y_{i,1}^t \neq r_{i,1}^t)\ \texttt{then fault}; \\ & \qquad \ldots; \\ & \qquad y_{i,q_i^t}^t := o_{i,q_i^t}^t.f_{i,q_i^t}^t(n_{i,q_i^t}^t);\ \texttt{if}(y_{i,q_i^t}^t \neq r_{i,q_i^t}^t)\ \texttt{then fault};\,\} \\ & \texttt{if}(w_i^t \neq c_i^t)\ \texttt{then fault else}\ g_i^t := 1; \end{aligned}$$

- If H_i^t is a live transaction, then

$$\begin{aligned} CP_i^t \;=\; & w_i^t := \texttt{atomic}\,\{\, y_{i,1}^t := o_{i,1}^t.f_{i,1}^t(n_{i,1}^t); \\ & \qquad \ldots; \\ & \qquad y_{i,q_i^t}^t := o_{i,q_i^t}^t.f_{i,q_i^t}^t(n_{i,q_i^t}^t); \\ & \qquad \texttt{fault};\,\} \end{aligned}$$

- If H_i^t is a commit-pending transaction, then

$$\begin{aligned} CP_i^t \;=\; & w_i^t := \texttt{atomic}\,\{\, y_{i,1}^t := o_{i,1}^t.f_{i,1}^t(n_{i,1}^t);\ \texttt{if}(y_{i,1}^t \neq r_{i,1}^t)\ \texttt{then fault}; \\ & \qquad \ldots; \\ & \qquad y_{i,q_i^t}^t := o_{i,q_i^t}^t.f_{i,q_i^t}^t(n_{i,q_i^t}^t);\ \texttt{if}(y_{i,q_i^t}^t \neq r_{i,q_i^t}^t)\ \texttt{then fault};\,\} \end{aligned}$$

Figure 5: The construction of CP_i^t and GP_i^t for Lemma 16 and Lemma 17 (case of $t \neq t_0$).

Thus, the command $GP_i^t; CP_i^t$ begins by reading the signals of the last transaction of every thread that, according to H, should end before the i-th transaction of thread t starts. It then performs an `atomic` block in which it invokes the sequence of object method invocations induced by H_i^t. After the `atomic` block ends, the command signals the end of the transaction. The desired program P_H is:

$$P_H \;=\; C_H^1 \parallel \ldots \parallel C_H^m.$$

We now construct a particular trace τ of P_H. We build τ by first constructing a trace τ^t for every sequential command C_H^t and then interleaving the traces $\tau^1, \ldots, \tau^m$ in a particular way. Consider the set of traces $A(C_H^t)t$ of the sequential command C_H^t, and let $\tau^t \in A(C_H^t)t$ be the maximal trace without any `fault` actions such that $H|_t = \mathsf{history}(\tau^t)$. The trace τ^t exists, since by construction of C_H^t and the definition of the trace set of a sequential command (Figure 4), there is a trace in $A(C_H^t)t$ for every possible parameter and return value of object method invocations and `atomic` blocks in C_H^t; in particular, $A(C_H^t)t$ contains a trace where the parameters and return values of object method invocations and the return values of transactions are as in $H|_t$. We now partition every τ^t into $\big|H|_t\big|$ subsequences that we later interleave to create τ:

$$\tau^t = \tau_1^t \ldots \tau_{|H|_t|}^t.$$

Formally, for every $i = 1..\big|H|_t\big|$ there is exactly one TM interface action ψ_i^t in τ_i^t and the conditions in Figure 6 hold. This defines τ_i^t uniquely and ensures that, if τ_i^t ends with $\psi_i^t = (\_, t, \mathsf{txbegin})$, then it contains all the actions that are used to read the global signaling variables that precede ψ_i^t in τ^t. The desired trace is constructed by interleaving the subsequences of the traces $\tau^1, \ldots, \tau^m$ according to the order induced by H. Formally,

$$\tau = \tau_{j_1}^{t_1} \ldots \tau_{j_{|H|}}^{t_{|H|}},$$

$$\tau_i^t = \begin{cases} \_\,\psi_i^t, & \psi_i^t \in \{(\_, t, \mathsf{txbegin}), (\_, t, \mathsf{call}\ o.f(n)), (\_, t, \mathsf{txcommit})\}; \\ \psi_i^t, & \psi_i^t = (\_, t, \mathsf{OK}); \\ \psi_i^t\,(\_, t, y_{i,\_}^t := n), & \psi_i^t = (\_, t, \mathsf{ret}(n)\ o.f) \wedge \mathsf{txof}(\psi_i^t, \tau^t) \in \mathsf{live}(\tau^t); \\ \psi_i^t\,(\_, t, y_{i,\_}^t := n)\,(\_, t, \mathsf{assume}(y_{i,\_}^t = r_{i,\_}^t)), & \psi_i^t = (\_, t, \mathsf{ret}(n)\ o.f) \wedge \mathsf{txof}(\psi_i^t, \tau^t) \notin \mathsf{live}(\tau^t); \\ \psi_i^t\,(\_, t, w_i := c_i^t)\,(\_, t, \mathsf{assume}(w_i = c_i^t))\,(\_, t, g_i^t := 1), & \psi_i^t \in \{(\_, t, \mathsf{committed}), (\_, t, \mathsf{aborted})\}. \end{cases}$$

Figure 6: The construction of τ_i^t for Lemma 16 and Lemma 17.

where $H(i) = (\_, t_i, \_)$ and $j_i = \left|(H\downarrow_i)|_{t_i}\right|$. Note that by construction $\mathsf{history}(\tau) = H$.

Since $\tau^t \in A(C_H^t)t$, we have $\tau \in A(P_H)$. Let s be the state where all the local variables are set to u and for all t, $g_i^t = 0$ for $i \neq 0$ and $g_0^t = 1$. By the construction of τ, we have $\mathsf{eval}(s, \tau) \neq \emptyset$. Then, since $\mathsf{history}(\tau) = H \in \mathcal{T}_C$, we have $\tau \in [\![P_H]\!](s, \mathcal{T}_C)$. Since $\tau \in [\![P_H]\!](s, \mathcal{T}_C)$ and $\mathcal{T}_C \preceq \mathcal{T}_A$, by Definition 7 there exists a trace $\tau' \in [\![P_H]\!](s, \mathcal{T}_A)$ such that $\tau' \in A(P_H)$, $\tau'|_{\neg\mathsf{trans}} = \tau|_{\neg\mathsf{trans}}$ and $S' = \mathsf{history}(\tau') \in \mathcal{T}_A$.

We now analyze the relationship between a transaction in τ and the one in τ' arising from the execution of the same commands. The construction in Figure 5 ensures that:

- If a transaction in τ is completed, then so is the corresponding transaction in τ', since every completed transaction in τ is followed by a non-transactional action assigning to g_i^t and $\tau'|_{\neg\mathsf{trans}} = \tau|_{\neg\mathsf{trans}}$. Furthermore, a transaction in τ is committed if and only if so is the corresponding transaction in τ', and in this case the return values for transactional actions inside the transactions match.
- If a transaction in τ is live, then the corresponding transaction in τ' is live, aborted or does not appear in τ' at all.
- If a transaction in τ is commit-pending, then the corresponding transaction in τ' can have any status or may not appear in τ' at all. However, if the transaction in τ' is visible (in particular, commit-pending or committed), then the return values for transactional actions inside the transactions match.

Let $H_1 = (H|_{\neg\mathsf{live}})|_{\neg\mathsf{abortedtx}}$ and $S^r = (S'|_{\neg\mathsf{live}})|_{\neg\mathsf{abortedtx}}$. Since $S' \in \mathcal{T}_A$ and $\mathcal{T}_A$ is closed under removing live and aborted transactions, $S^r \in \mathcal{T}_A$. Let p_t', respectively, p_t'' be the index of last $\mathsf{txcommit}$ or $\mathsf{committed}$ action in $H_1|_t$, respectively, $S^r|_t$; if there is no such action, the corresponding index is 0. Let $p_t = \min(p_t', p_t'')$. From the above analysis it follows that $(H_1|_t)\downarrow_{p_t} \equiv (S^r|_t)\downarrow_{p_t}$.

Let S_1 be a history obtained from S^r by renaming the action identifiers such that $S_1 \equiv S^r$, $(H_1|_t)\downarrow_{p_t} = (S_1|_t)\downarrow_{p_t}$ and all actions in S_1 that do not belong to $(S^r|_t)\downarrow_{p_t}$ for some t have identifiers that do not appear in H. Since $S^r \in \mathcal{T}_A$ and $\mathcal{T}_A$ is closed under renaming action identifiers, we get $S_1 \in \mathcal{T}_A$.

The history S_1 does not contain live transactions so that $\mathsf{comp}(S_1) \neq \emptyset$. Then, since $S_1 \in \mathcal{T}_A$ and $\mathcal{T}_A$ is closed under commit-pending transactions completion, we get that there exists a history $S_2 \in \mathsf{comp}(S_1) \cap \mathcal{T}_A$. Then $S_2 = S_1 S_1'$ for some history S_1' that contains only $\mathsf{committed}$ and $\mathsf{aborted}$ actions; without loss of generality we can assume that identifiers of actions in S_1' do not appear in H. Let S_1'' be the subsequence of $\mathsf{committed}$ actions in S_1 that are not in H_1. The history $H_1 S_1'' S_1'$ contains only visible transactions. We construct the desired history H^c by aborting all the commit-pending transactions in $H_1 S_1'' S_1'$. Let $H^c = H_1 S_1'' S_1' H_a$, where H_a consists of actions $(\_, t, \mathsf{aborted})$ for every thread t ending with a commit-pending transaction in $H_1 S_1'' S_1'$; these actions have unique identifiers that do not appear in H. We have that $H^c \in \mathsf{comp}(H|_{\neg\mathsf{live}})$, for the following reasons:

(i) S_1'' completes the commit-pending transactions in H that get committed in S'.
(ii) S_1' completes the commit-pending transactions in H that stay commit-pending in S'.
(iii) H_a aborts the commit-pending transactions in H that become live or aborted in S' or do not appear there at all.

Let $S = S_2|_{\mathsf{com}}$. Since $S_2 \in \mathcal{T}_A$ has only completed transactions and $\mathcal{T}_A$ is closed under removing live and aborted transactions, we get that $S = S_2|_{\mathsf{com}} \in \mathcal{T}_A$. We now show $H^c|_{\mathsf{com}} \sqsubseteq_{\mathsf{op}} S$, thus completing the proof. We first show that $\forall t \in \mathsf{ThreadID}.\ (H^c|_t)|_{\mathsf{com}} = S|_t$ by considering several cases.

- $H|_t$ does not contain commit-pending transactions. Then $H_1|_t = S_1|_t$, $H_1|_t$ contains only committed transactions and $S_1' = S_1'' = H_a = \varepsilon$. Hence,

$$(H^c|_t)|_{\mathsf{com}} = (H_1|_t)|_{\mathsf{com}} = H_1|_t = S_1|_t = (S_1|_t)|_{\mathsf{com}} = (S_2|_t)|_{\mathsf{com}} = S|_t.$$

- $H|_t$ ends with a commit-pending transaction and $(S_1''S_1'H_a)|_t = S_1''|_t = (\_, t, \mathsf{committed})$. Then $H_1|_t = (H_1|_t)\downarrow_{p_t}$ and $S_1|_t = ((S_1|_t)\downarrow_{p_t})(S_1''|_t)$. Hence,

$$H^c|_t = (H_1|_t)\,(S_1''|_t) = ((H_1|_t)\downarrow_{p_t})\,(S_1''|_t) = ((S_1|_t)\downarrow_{p_t})\,(S_1''|_t) = S_1|_t = S_2|_t.$$

  Then $(H^c|_t)|_{\mathsf{com}} = (S_2|_t)|_{\mathsf{com}} = S|_t$.
- $H|_t$ ends with a commit-pending transaction and $(S_1''S_1'H_a)|_t = S_1'|_t$. Then $H_1|_t = S_1|_t$. Hence,

$$H^c|_t = (H_1|_t)\,(S_1'|_t) = (S_1|_t)\,(S_1'|_t) = S_2|_t.$$

  Then $(H^c|_t)|_{\mathsf{com}} = (S_2|_t)|_{\mathsf{com}} = S|_t$.
- $H|_t$ ends with a commit-pending transaction and $(S_1''S_1'H_a)|_t = H_a|_t = (\_, t, \mathsf{aborted})$. Then $((H_1|_t)\,(H_a|_t))|_{\mathsf{com}} = ((H_1|_t)\downarrow_{p_t})|_{\mathsf{com}}$ and $S_1|_t = (S_1|_t)\downarrow_{p_t}$. Hence,

$$(H^c|_t)|_{\mathsf{com}} = ((H_1|_t)\,(H_a|_t))|_{\mathsf{com}} = ((H_1|_t)\downarrow_{p_t})|_{\mathsf{com}} = ((S_1|_t)\downarrow_{p_t})|_{\mathsf{com}} = S_1|_t = S_2|_t.$$

This shows $\forall t \in \mathsf{ThreadID}.\ (H^c|_t)|_{\mathsf{com}} = S|_t$. Finally, for every completed transaction T in τ, the value of g_i^t is set to 1 after the transaction completes and is read before every transaction T' that begins after the completion of T. Hence, the transaction in τ' corresponding to T completes before the transaction corresponding to T' begins. Then the real-time order in H^c is preserved in S, which implies $H^c|_{\mathsf{com}} \sqsubseteq_{\mathsf{op}} S$. ■

LEMMA 17. *Let $\mathcal{T}_C$ and $\mathcal{T}_A$ be transactional memories such that $\mathcal{T}_C \preceq \mathcal{T}_A$ and $\mathcal{T}_A$ is closed under live transactions omission and pending transactions completion. Let $H = H'\psi \in \mathcal{T}_C$, where ψ is a response action and $\psi \notin \{(\_, \_, \mathsf{committed}), (\_, \_, \mathsf{aborted})\}$. There exists a history $S \in \mathcal{T}_A$ such that $H^c \sqsubseteq_{\mathsf{op}} S$, where $H^c \in \mathsf{cTMSpast}(H)$.*

Proof: Let ψ be an action of thread t_0. As in Lemma 16, we build a program P_H for H that records the interaction of every thread t with the transactional memory and monitors whether the real-time order includes that in H. The sequential commands C_H^i constructed for all the threads except t_0 are the same as in Lemma 16, but C_H^0, the sequential command built for t_0, is different.

As in Lemma 16, the command C_H^0 records the arguments passed to method calls and the return values during a transaction the program using local variables that are never rewritten. However, C_H^0 observes these values *after* the transaction completes using conditional command. This makes the sequence of actions inside a committed transaction observable to thread t_0. As a result, if P_H is evaluated with a history H' which has a different per-thread sequence of actions inside committed transactions than that of H, then the trace would reflect the difference by either a `fault` by some thread other than t_0 or by the non-transactional actions of t_0. Histories H and H' might perform different actions inside aborted transactions, but this is fine for our purposes because the actions inside aborted transactions are not observable. Monitoring the real-time order is done using the same monitoring mechanism as in Lemma 16.

The challenge here is to construct a program that can observe the sequence of actions inside the transaction of ψ, which is live. The only one way to get this observation is to use faults. Hence, given an incomplete history H, we construct a program which faults *if* the local observations of t_0 are the same as in H.

Now we formalize the intuitive proof sketch. To make the proof self-contained we repeat some parts of the constructions that were already discussed in Lemma 16. Let us choose an integer value $u \neq 1$ which does not appear in H. We use the following shorthands:

- We denote by m the number of threads that have an action in H, and for simplicity, let m be the largest thread identifier occurring in H.

$$GP_i^{t_0} \quad = \quad z_{i,1}^{t_0} := g^1_{\mathsf{lasttx}(t_0,i,1)}; \texttt{if}(z_{i,1}^{t_0} \neq 1) \texttt{ then mismatch} := 1;$$
$$\ldots$$
$$z_{i,m}^{t_0} := g^m_{\mathsf{lasttx}(t_0,i,m)}; \texttt{if}(z_{i,m}^{t_0} \neq 1) \texttt{ then mismatch} := 1;$$

- For the live transaction of ψ, $CP^{t_0}_{k^{t_0}}$ is constructed as follows:

$$\begin{aligned} CP^{t_0}_{k^{t_0}} \quad = \quad & w^{t_0}_{k^{t_0}} := \texttt{atomic}\,\{\, y^{t_0}_{k^{t_0},1} := o^{t_0}_{k^{t_0},1}.f^{t_0}_{k^{t_0},1}(n^{t_0}_{k^{t_0},1}); \\ & \qquad \texttt{if}(y^{t_0}_{k^{t_0},1} \neq r^{t}_{k^{t_0},1}) \texttt{ then mismatch} := 1; \\ & \qquad \ldots; \\ & \qquad y^{t_0}_{k^{t_0},q^{t_0}_{k^{t_0}}} := o^{t_0}_{k^{t_0},q^{t_0}_{k^{t_0}}}.f^{t_0}_{k^{t_0},q^{t_0}_{k^{t_0}}}(n^{t_0}_{k^{t_0},q^{t_0}_{k^{t_0}}}); \\ & \qquad \texttt{if}(y^{t}_{k^{t_0},q^{t}_{k^{t_0}}} \neq r^{t}_{k^{t_0},q^{t}_{k^{t_0}}}) \texttt{ then mismatch} := 1; \\ & \qquad \texttt{if}(\mathsf{mismatch} \neq 1) \texttt{ then fault};\, \} \end{aligned}$$

- For an aborted or a committed transaction by t_0, CP^t_i is constructed as follows:

$$\begin{aligned} CP^{t_0}_{i} \quad = \quad & w^{t_0}_{i} := \texttt{atomic}\,\{\, y^{t_0}_{i,1} := o^{t_0}_{i,1}.f^{t_0}_{i,1}(n^{t_0}_{i,1});\ \texttt{if}(y^{t_0}_{i,1} \neq r^{t_0}_{i,1}) \texttt{ then mismatch} := 1; \\ & \qquad \ldots; \\ & \qquad y^{t_0}_{i,q^{t_0}_i} := o^{t_0}_{i,q^{t_0}_i}.f^{t_0}_{i,q^{t_0}_i}(n^{t_0}_{i,q^{t_0}_i});\ \texttt{if}(y^{t_0}_{i,q^{t_0}_i} \neq r^{t_0}_{i,q^{t_0}_i}) \texttt{ then mismatch} := 1;\, \} \\ & \texttt{if}(w^{t_0}_i \neq c^{t_0}_i) \texttt{ then mismatch} := 1 \texttt{ else } g^{t_0}_i := 1; \end{aligned}$$

Figure 7: The construction of $CP^{t_0}_i$ and $GP^{t_0}_i$ for Lemma 17.

- We denote by k^t the number of transactions started by thread t in H, i.e., the number of $(\_, t, \mathsf{txbegin})$ actions in H.
- We partition $H|_t$ into k^t subsequences: $H|_t = H^t_1 \ldots H^t_{k^t}$, where H^t_i is comprised of the actions in the i-th transaction of t. Specifically, $H^t_i(1) = (\_, t, \mathsf{txbegin})$.
- We let c^t_i be the outcome of the i-th transaction of thread t, i.e., $c^t_i = \mathsf{committed}$ or $c^t_i = \mathsf{aborted}$. If the transaction is not completed, c^t_i is undefined.
- We denote by q^t_i the number of call actions of thread t in its i-th transaction, i.e., in H^t_i.
- We let $(\_, t, \mathsf{call}\ o^t_{i,j}.f^t_{i,j}(n^t_{i,j}))$ be the j-th call action of thread t in its i-th transaction.
- We let $(\_, t, \mathsf{ret}(r^t_{i,j})\ o^t_{i,j}.f^t_{i,j})$ be the j-th ret action of thread t in its i-th transaction.
- If the response to $(\_, t, \mathsf{call}\ o^t_{i,j}.f^t_{i,j}(n^t_{i,j}))$ is an aborted action, we let $r^t_{i,j}$ be aborted. If there is no response to $(\_, t, \mathsf{call}\ o^t_{i,j}.f^t_{i,j}(n^t_{i,j}))$, i.e. the transaction is live and $(\_, t, \mathsf{call}\ o^t_{i,j}.f^t_{i,j}(n^t_{i,j}))$ is the last action in the transaction, then $r^t_{i,j}$ is set to u.
- We denote by $\mathsf{lasttx}(t, i, t')$ the number of transactions of thread t' in H that either committed or aborted before the i-th transaction of thread t started, i.e., the number of $(\_, t', \mathsf{committed})$ and $(\_, t', \mathsf{aborted})$ actions preceding the i-th $(\_, t, \mathsf{txbegin})$ action in H.

For every thread $t = 1..m$ we construct a straight-line command

$$C^t_H = GP^t_1; CP^t_1; GP^t_2; CP^t_2; \ldots; GP^t_{k^t}; CP^t_{k^t}\ .$$

CP^t_i is a sequence of commands constructed according to the i-th transaction of thread t in H. The construction of $CP^{t_0}_i$ and $GP^{t_0}_i$ is shown in Figure 7, and the construction of CP^t_i and GP^t_i, when $t \neq t_0$, is shown in Figure 5.

Here the g variables are global and all others are local. The g and z variables are used to monitor the real-time order. The variable g^t_i is written only by thread t and is used to signal that the i-th transaction of thread t ended: g^t_i is set to 1 only after t's i-th transaction either committed or aborted. The variable $z^t_{i,t'}$ is local to thread t. It is used to record if the $\mathsf{lasttx}(t, i, t')$-th transaction of thread t' signaled that it had ended before the i-th transaction of

thread t started. As $\mathsf{lasttx}(t,i,t')$ might be 0, we add a dummy variable g_0^t for every thread t. Later in the proof we execute the program from a state in which g_0^t is initialised to 1. The variable w_i^t records whether the i-th transaction of thread t committed or aborted. The variables $y_{i,j}^t$ record the return value of the j-th object method invocation in the i-th transaction of thread t. Variable mismatch is a local variable of thread t_0. It us used to check that the return values values for object methods by thread t_0 are as expected: it is set to 1 if there is a mismatch. It will help in producing a different trace if there is a mismatch.

Thus, the command C_H^t begins by reading the signals of the last transaction of every thread that, according to H, should end before the i-th transaction of thread t starts. It then performs an `atomic` block in which it invokes the sequence of object method invocations induced by H_i^t. After the `atomic` block ends, CP_i^t signals the end of the transaction. The desired program P_H is the parallel composition of the straight-line commands of the different threads:

$$P_H \;=\; C_H^1 \parallel \ldots \parallel C_H^m.$$

We now construct a particular trace τ of P_H. We build τ by first constructing a trace τ^t for every sequential command C_H^t and then interleaving the traces $\tau^1, \ldots, \tau^m$ in a particular way.

Consider the set of traces $A(C_H^t)t$ (cf. Figure 4) of the sequential command C_H^t, and let $\tau^t \in A(C_H^t)t$ be the maximal trace such that $H|_t = \mathsf{history}(\tau_c^t)$. The trace τ^t exists, since by construction of C_H^t and the definition of the trace set of a sequential command, there is a trace in $A(C_H^t)t$ for every possible parameter and return value of object method invocations and `atomic` blocks in C_H^t; in particular, $A(C_H^t)t$ contains a trace where the parameters and return values of object method invocations and the return values of transactions are as in $H|_t$.

We now partition every τ^t into $\left|H|_t\right|$ subsequences that we later interleave to create τ:

$$\tau^t = \tau_1^t \ldots \tau_{|H|_t|}^t.$$

Formally, for every $i = 1..\left|H|_t\right|$ there is exactly one TM interface action ψ_i^t in τ_i^t. The construction of τ_i^t is shown in Figure 6. Note that this defines $\tau_1^t \ldots \tau_{\left|H|_t\right|}^t$ uniquely. The above definition also ensures that, if τ_i^t ends with $\psi_i^t = (\_, t, \mathsf{txbegin})$, then it contains all the actions that are used to read the global signaling variables that precede ψ_i^t in τ^t.

The desired trace is constructed by interleaving the subsequences of the traces $\tau^1, \ldots, \tau^m$ according to the order induced by H. Formally,

$$\tau = \tau_{j_1}^{t_1} \ldots \tau_{j_{|H|}}^{t_{|H|}},$$

where $H(i) = (\_, t_i, \_)$ and $j_i = \left|H\downarrow_i |_{t_i}\right|$. Note that by construction $\mathsf{history}(\tau) = H$.

This ends the construction, which is similar to Lemma 16. The rest of the proof diverges from the proof of Lemma 16.

Since $\tau^t \in A(C_H^t)t$, we have $\tau \in A(P_H)$. Since $\mathsf{history}(\tau) = H \in \mathcal{T}_C$, we have $\tau \in [\![P_H]\!](s, \mathcal{T}_C)$.

Since $\tau \in [\![P_H]\!](s, \mathcal{T}_C)$ has a `fault` after the action ψ and $\mathcal{T}_C \preceq \mathcal{T}_A$, by Definition 7 there exists a trace $\tau' \in [\![P_H]\!](s, \mathcal{T}_A)$ that ends with a fault by thread t_0. Hence, $\tau' \in A(P_H)$ and $S' = \mathsf{history}(\tau')$ is in $\mathcal{T}_A$. Since τ and τ' have only one fault, the global variables in τ and τ' have same values, and thus the real-time order of actions in H is preserved in S'. Also, the actions inside committed transactions in S' are same as in H and all the committed transactions in τ are committed in τ' since we check the local values w_i^t outside the atomic block.

Now we create a history H^c using the history S'. Let $H_1 = H|_{\mathsf{v}\cup\psi}$ and $S^r = S'|_{\mathsf{v}\cup\psi}$, respectively, where $\cdot|_{\mathsf{v}\cup\psi}$ is the projection to actions by all the visible transactions and the transaction of ψ.

Since the return values of object invocations are checked inside the completed transactions in H_1 and that of ψ, all the return values inside completed transactions that completed before $\mathsf{txbegin}$ of $\mathsf{txof}(\psi, S^r)$ in S^r and the transaction of ψ are same as in H_1; otherwise, the trace τ would have a `fault` by a thread $t \neq t_0$, or would not have a `fault` after ψ by thread t_0.

Thus, for thread t_0, $(S^r\downarrow_\psi)|_{t_0} \equiv (H_1\downarrow_\psi)|_{t_0}$ and for every thread $t \neq t_0$, if there is a $\mathsf{txcommit}$ action in $S^r|_t$, then $(S^r\downarrow_{k_t})|_t \equiv (H_1\downarrow_{k_t})|_t$, where k_t is the index of the k^{th} $\mathsf{txcommit}$ action, ψ_t, in $H_1|_t$ and k is the number of $\mathsf{txcommit}$ actions in $S^r|_t$. Let S_1 be a history with unique action identifiers obtained from S^r by renaming the

action identifiers in S^r such that $S_1 \equiv S^r$ and $\forall i = 1 \ldots |(H_1\downarrow_\psi)|_{t_0}|$, $(S_1|_{t_0})(i) = (H_1|_{t_0})(i)$, and for every thread $t \neq t_0$, if there is a txcommit action in S^r, then $\forall i = 1 \ldots k_t$, $(S_1|_t)(i) = (H_1|_t)(i)$ and all actions in S_1 that are not in H_1 do not have action identifier same as any action in H_1. Note that if there is not txcommit action in $S^r|_t$, then $S_1|_t = S^r|_t = \varepsilon$. Note that since $S_1|_t$ contains only visible transactions, either $S_1|_t = \varepsilon$ or $S_1|_t = (S_1|_t)\downarrow_{\psi_t}$ or $S_1|_t = (S_1|_t)\downarrow_{\psi_t} \psi'_t$, where $\psi'_t = (\_, t, \mathsf{committed})$ or $(\_, t, \mathsf{aborted})$.

Let H_p be the subsequence of H_1 such that for thread t_0, $H_p|_{t_0} = H_1|_{t_0}$ and for all other threads, if $S^r|_t$ does not contains a txcommit action then $H_p|_t = \varepsilon$, if $S^r|_t$ contains a txcommit action then $H_p|_t = H_1|_t$ if ψ_t is the last action in $H_1|_t$; otherwise $H_p|_t = (H_1|_t)\downarrow_{\psi_t} \psi''_t$, where ψ''_t is the response of ψ_t in H_1. Let H'_1 be a subsequence of H_1 such that for thread t_0, $H'_1|_{t_0} = H_1|_{t_0}$ and for all other threads, if $S^r|_t$ does not contains a txcommit action then $H'_1|_t = \varepsilon$, otherwise $H'_1|_t = (H_1|_t)\downarrow_{\psi_t}$. Note that for every thread $t \neq t_0$, either $S_1|_t$ does not contain a txcommit action and $H'_1|_t = H_p|_t = \varepsilon$, or $S_1|_t$ contains a txcommit action and $H'_1|_t = (H_p|_t)\downarrow_{\psi_t} = (H_1|_t)\downarrow_{\psi_t}$, where ψ_t is the last txcommit action in $S_1|_t$.

Note that a transaction in τ that does not complete before txbegin of $\mathsf{txof}(\psi, \tau)$ may have different response in τ'. Also note that a transaction that is in H_1 but not in H'_1 cannot complete before the txbegin of $\mathsf{txof}(\psi, H_1)$. The reason for this is that for every completed transaction, either committed or aborted, that completed in H_1 before txbegin of $\mathsf{txof}(\psi, H_1)$, there is a completed transaction, with the same return value, in S_1. This is because for every committed and aborted transaction in τ that completed before txbegin of $\mathsf{txof}(\psi, \tau)$, τ' contains assignment to the variable g following the completion of the transaction, and the $\mathsf{txof}(\psi, H_1)$ has checks in τ before it began on the number of transactions completed by other threads. However, for a transaction in τ that does not complete before txbegin of $\mathsf{txof}(\psi, \tau)$ may have only a prefix or no action executed in τ', and if it is completed in both τ and τ', it may have different response. Note that for any transaction in H'_1, all its preceding transactions are in S_1 with the same response actions, for the same reason.

Since $S' \in \mathcal{T}_A$ and $\mathcal{T}_A$ is closed under removing live and aborted transactions, $S^r \in \mathcal{T}_A$. Since $\mathcal{T}_A$ is closed under renaming action identifiers, we get $S_1 \in \mathcal{T}_A$. Since $S_1 \in \mathcal{T}_A$ and $\mathcal{T}_A$ is closed under commit-pending transactions completion, there exists a history $S_2 \in \mathsf{comp}(S_1) \cap \mathcal{T}_A$. Let $S_2 = S_1 S'_1 \in \mathcal{T}_A$ be a history such that every action in S'_1 is either committed or aborted and has unique action identifiers with no action identifier that appear in S_1 or H_1, and every transaction in S_2, except that of ψ, is complete. Let $S = S_2|_{\mathsf{c}\cup\psi}$, where $\cdot|_{\mathsf{c}\cup\psi}$ is the projection to actions by all the committed transactions and the transaction of ψ. Let S''_1 be the subsequence of committed and aborted actions in S_1 which are not in H'_1. There are committed and aborted actions in S_1 which are not in H'_1 because some of the commit-pending transactions in τ are executed to completion in τ' after txbegin of $\mathsf{txof}(\psi, \tau)$.

Let $H^c = (H'_1 S''_1 S'_1)|_{\mathsf{c}\cup\psi}$. Note that all the transactions in H^c and S are completed, except for the transaction of ψ. The reason for this is for every thread $t \neq t_0$, the last transaction in $H'_1|_t$ is a completed transaction or a commit-pending transaction, which gets completed either by S''_1 if it is completed in S_1, or it gets completed by S'_1 if it is commit-pending in S_1. Since $S_2 \in \mathcal{T}_A$ and $\mathcal{T}_A$ is closed under removing live and aborted transactions, in particular removing aborted transactions, we get that $S = S_2|_{\mathsf{c}\cup\psi} \in \mathcal{T}_A$. Now we show that $H^c \in \mathsf{cTMSpast}(H)$ (Claim 17.1) and then show Claim 17.2 and Claim 17.3 to prove that $H^c \sqsubseteq_{\mathsf{op}} S$.

CLAIM 17.1. $H^c \in \mathsf{cTMSpast}(H)$.

Proof: Let $H''_1\psi$ be the subsequence of H_p obtained by removing some transactions in the following way:

- a committed transaction T_c is removed if T_c is not in H^c.
- an aborted transaction T_a is removed if there is no txbegin action, by any thread in $H''_1\psi$, after T_a completes.

Consider the following about the history $H''_1\psi$:

(i) $H''_1\psi$ is a subsequence of H_p, which is a subsequence of H.

(ii) H''_1 contains some of the visible transactions in H and also includes $\mathsf{txof}(\psi, H)$. This is because H_1 contains all the visible transactions in H and $\mathsf{txof}(\psi, H)$.

(iii) Since for any transaction in H_p, all its preceding transactions are in S_1 with the same response actions of preceding transactions. (The response of the last transaction by a thread $t \neq t_0$ in H_p might have different response in S_1.). Thus, we get that for a transaction in H_p (and in H), if it is included in H_p then all of its preceding transactions in H_p (and in H) are also included. Since all the committed transactions in H_p, which

have a transaction after them in real-time order in $H_1''\psi$, are included in $H_1''\psi$, it follows that

$$\forall T \in \mathsf{tx}(H_1''\psi).\, \forall T'.\, T' \prec_{H_1''\psi} T \iff (T' \prec_H T \wedge T' \in \mathsf{committed}(H)).$$

Therefore, $H_1''\psi \in \mathsf{TMSpast}(H)$, by Definition 3.

Note that for every aborted transaction by a thread $t \neq t_0$, that is last in H_p and included in $H_1''\psi$, the response action is removed in H_1' and replaced by a committed or aborted action in S_1''. Therefore, $(H_1'S_1''S_1')|_{\mathsf{c}\cup\psi} \in \mathsf{comcomp}(\mathsf{com}(H_1'')\psi)$, and hence $H^c \in \mathsf{comcomp}(\mathsf{com}(H_1'')\psi)$.

Since $H_1''\psi \in \mathsf{TMSpast}(H)$ and $H^c \in \mathsf{comcomp}(\mathsf{com}(H_1'')\psi)$, Definition 4 implies that $H^c \in \mathsf{cTMSpast}(H)$. ■

CLAIM 17.2. *For every thread $t \neq t_0$, if there is a* txcommit *action in S_1, then $(S_1\downarrow_{\psi_t})|_t = (H_1'\downarrow_{\psi_t})|_t$, where ψ_t is the last* txcommit *action in $S_1|_t$.*

Proof: Since $(S^r\downarrow_{k_t})|_t \equiv (H_1\downarrow_{k_t})|_t$, where $H_1|_t(k_t) = \psi_t$, and by the construction of S_1, for every thread $t \neq t_0$ it holds that $\forall i = 1 \ldots k_t$, $(S_1|_t)(i) = (H_1|_t)(i)$, we get that $(S_1\downarrow_{\psi_t})|_t = (H_1\downarrow_{\psi_t})|_t$. By construction of H_1', $H_1'|_t\downarrow_{\psi_t} = H_p|_t\downarrow_{\psi_t} = H_1|_t\downarrow_{\psi_t}$. Therefore, $(S_1\downarrow_{\psi_t})|_t = (H_1'\downarrow_{\psi_t})|_t$. ■

CLAIM 17.3. *$H^c|_t = S|_t$, for every thread $t \neq t_0$.*

Proof: If $S_1|_t$ does not contain a txcommit, then $S_1|_t = H_1'|_t = \varepsilon$. Also, $H^c|_t = \varepsilon$ and $S|_t = \varepsilon$, and the claim holds vacuously.

If there is a txcommit action in $S_1|_t$, Claim 17.2 implies that $(S_1\downarrow_{\psi_t})|_t = (H_1'\downarrow_{\psi_t})|_t$, where ψ_t is the last txcommit action in $S_1|_t$. Let $S_1|_t = (S_1\downarrow_{\psi_t})|_t S_t$, for some history S_t. Note that either $S_t = \varepsilon$ or $S_t = (\_, t, \mathsf{committed})$ or $(\_, t, \mathsf{aborted})$.

If $S_t = (\_, t, \mathsf{committed})$ or $(\_, t, \mathsf{aborted})$, then $S_1|_t$ is a completed history and hence, $S_2|_t = S_1|_t$ and $S_1'|_t = \varepsilon$. Since S_1'' contains all committed and aborted transactions in S_1 which are not in H_1', it follows that $S_1''|_t = S_t$. Hence, $S_2|_t = S_1|_t = (H_1'S_1'')|_t$, and

$$H^c|_t = ((H_1'S_1''S_1')|_{\mathsf{c}\cup\psi})|_t = ((H_1'S_1'')|_{\mathsf{c}\cup\psi})|_t = (S_1|_{\mathsf{c}\cup\psi})|_t = (S_2|_{\mathsf{c}\cup\psi})|_t = S|_t$$

If $S_t = \varepsilon$, then the last transaction in $S_1|_t$ is either commit-pending or completed in both $H_1'|_t$ and $S_1|_t$, and hence, $S_2|_t = (S_1S_1')|_t$ and $S_1''|_t = \varepsilon$. Since $(S_1\downarrow_{\psi_t})|_t = (H_1'\downarrow_{\psi_t})|_t$, it follows that $(S_1S_1')|_t = (H_1'S_1')|_t$, and

$$H^c|_t = ((H_1'S_1''S_1')|_{\mathsf{c}\cup\psi})|_t = ((H_1'S_1')|_{\mathsf{c}\cup\psi})|_t = ((S_1S_1')|_{\mathsf{c}\cup\psi})|_t = (S_2|_{\mathsf{c}\cup\psi})|_t = S|_t$$

■

Since thread t_0 faults only if all the checks that ensures it succeeded, we get that $H_1'|_{t_0} = S_1|_{t_0}$. Therefore,

$$H^c|_{t_0} = (H_1'S_1'S_1'')|_{t_0} = H_1'|_{t_0} = S_1|_{t_0} = S_2|_{t_0} = S|_{t_0}.$$

Claim 17.3 implies that $H^c|_t = S|_t$, for all other threads $t \neq t_0$.

Since for every complete transaction T in τ the value of g_i^t is set to 1 after the transaction completes and is read by every transaction T' that begin after the completion of T, we get that T completes in τ' before the transaction T' begins; otherwise, it would generate a `fault` by a thread $t \neq t_0$, and will not generate a `fault` by t_0. Thus, the real-time order in H^c is preserved in S, implying that $H^c \sqsubseteq_{\mathsf{op}} S$ and completing the proof. ■

LEMMA 18. *Let $\mathcal{T}_C$ and $\mathcal{T}_A$ be transactional memories such that $\mathcal{T}_C \preceq \mathcal{T}_A$ and $\mathcal{T}_A$ is closed under live transactions omission, pending transactions completion and aborting live transactions. Let $H = H'\psi \in \mathcal{T}_C$, where $\psi = (t_0, \_, \mathsf{aborted})\}$. There exists a history $S \in \mathcal{T}_A$ such that $H^c \sqsubseteq_{\mathsf{op}} S$, where $H^c \in \mathsf{cTMSpast}(H)$.*

Proof: Let $H_0 = H\lfloor_{\psi_r}$, where ψ_r is the last response action by $\mathsf{txof}(\psi, H)$. Let $H_1 = H_0\psi_c\psi$, where ψ_c is the last request action in H by $\mathsf{txof}(\psi, H)$. Note that ψ_c may not necessarily be a $\mathsf{txcommit}$ action. It can be any arbitrary request action that got aborted. Since every prefix of H is in $\mathcal{T}_C$, the history H_0 is also in $\mathcal{T}_C$. Lemma 17 yields a history $H_0^c \in \mathsf{cTMSpast}(H_0)$ and a history $S_0 \in \mathcal{T}_A$ such that $H_0^c \sqsubseteq_{\mathsf{op}} S_0$, where $H_0^c \in \mathsf{cTMSpast}(H_\psi)$.

Let $H^c = H_0\psi_c\psi$ and $S = S_0\psi_c\psi$. Since $S_0 \in \mathcal{T}_A$ and $\mathcal{T}_A$ is closed under aborting live transactions, we get that $S \in \mathcal{T}_A$. Since $H_0^c \in \mathsf{cTMSpast}(H_0)$, we get $H^c \in \mathsf{cTMSpast}(H)$, implying that $H^c \sqsubseteq_{\mathsf{op}} S$, since $H_0^c \sqsubseteq_{\mathsf{op}} S_0$. ■

PROOF OF THEOREM 8(II). Let $H \in \mathcal{T}_C$. Since $\mathcal{T}_C \preceq \mathcal{T}_A$, Lemma 16 gives a history $H^c \in \mathsf{comp}(H|_{\neg\mathsf{live}})$ and a history $S \in \mathcal{T}_A$ such that $H^c|_{\mathsf{com}} \sqsubseteq_{\mathsf{op}} S$.

Since every prefix of H is in $\mathcal{T}_C$, for every response ψ action in H, $H_\psi = H\lfloor_\psi \in \mathcal{T}_C$.

If ψ is a $\mathsf{committed}$ action, then Lemma 16 gives a history $H^c \in \mathsf{comp}(H_\psi|_{\neg\mathsf{live}})$ and a history $S \in \mathcal{T}_A$ such that $H^c|_{\mathsf{com}} \sqsubseteq_{\mathsf{op}} S$.

If ψ is a response action, but neither $\mathsf{committed}$ nor $\mathsf{aborted}$, then Lemma 17 gives a history $S_\psi \in \mathcal{T}_A$ such that $H_\psi^c \sqsubseteq_{\mathsf{op}} S_\psi$, where $H_\psi^c \in \mathsf{cTMSpast}(H_\psi)$.

If ψ is an $\mathsf{aborted}$ action, then Lemma 18 gives a history $S_\psi \in \mathcal{T}_A$ such that $H_\psi^c \sqsubseteq_{\mathsf{op}} S_\psi$, where $H_\psi^c \in \mathsf{cTMSpast}(H_\psi)$.

In all cases, $H \sqsubseteq_{\mathsf{tms}} \mathcal{T}_A$. ■

Replicated Data Types: Specification, Verification, Optimality

Sebastian Burckhardt
Microsoft Research

Alexey Gotsman
IMDEA Software Institute

Hongseok Yang
University of Oxford

Marek Zawirski
INRIA & UPMC-LIP6

Abstract

Geographically distributed systems often rely on replicated eventually consistent data stores to achieve availability and performance. To resolve conflicting updates at different replicas, researchers and practitioners have proposed specialized consistency protocols, called replicated data types, that implement objects such as registers, counters, sets or lists. Reasoning about replicated data types has however not been on par with comparable work on abstract data types and concurrent data types, lacking specifications, correctness proofs, and optimality results.

To fill in this gap, we propose a framework for specifying replicated data types using relations over events and verifying their implementations using replication-aware simulations. We apply it to 7 existing implementations of 4 data types with nontrivial conflict-resolution strategies and optimizations (last-writer-wins register, counter, multi-value register and observed-remove set). We also present a novel technique for obtaining lower bounds on the worst-case space overhead of data type implementations and use it to prove optimality of 4 implementations. Finally, we show how to specify consistency of replicated stores with multiple objects axiomatically, in analogy to prior work on weak memory models. Overall, our work provides foundational reasoning tools to support research on replicated eventually consistent stores.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Replication; eventual consistency; weak memory

1. Introduction

To achieve availability and scalability, many networked computing systems rely on *replicated stores*, allowing multiple clients to issue operations on shared data on a number of *replicas*, which communicate changes to each other using message passing. For example, large-scale Internet services rely on *geo-replication*, which places data replicas in geographically distinct locations, and applications for mobile devices store replicas locally to support offline use. One benefit of such architectures is that the replicas remain locally available to clients even when network connections fail. Unfortunately, the famous CAP theorem [19] shows that such high **A**vailability and tolerance to network **P**artitions are incompatible with *strong* **C***onsistency*, i.e., the illusion of a single centralized replica handling all operations. For this reason, modern replicated stores often provide weaker forms of consistency, commonly dubbed *eventual consistency* [36]. 'Eventual' usually refers to the guarantee that

> if clients stop issuing update requests, then the replicas will eventually reach a consistent state. (1)

Eventual consistency is a hot research area, and new replicated stores implementing it appear every year [1, 13, 16, 18, 23, 27, 33, 34, 37]. Unfortunately, their semantics is poorly understood: the very term eventual consistency is a catch-all buzzword, and different stores claiming to be eventually consistent actually provide subtly different guarantees. The property (1), which is a form of *quiescent consistency*, is too weak to capture these. Although it requires the replicas to converge to the same state eventually, it doesn't say which one it will be. Furthermore, (1) does not provide any guarantees in realistic scenarios when updates never stop arriving. The difficulty of reasoning about the behavior of eventually consistent stores comes from a multitude of choices to be made in their design, some of which we now explain.

Allowing the replicas to be temporarily inconsistent enables eventually consistent stores to satisfy clients' requests from the local replica immediately, and broadcast the changes to the other replicas only after the fact, when the network connection permits this. However, this means that clients can concurrently issue conflicting operations on the same data item at different replicas; furthermore, if the replicas are out-of-sync, these operations will be applied to its copies in different states. For example, two users sharing an online store account can write two different zip codes into the delivery address; the same users connected to replicas with different views of the shopping cart can also add and concurrently remove the same product. In such situations the store needs to ensure that, after the replicas exchange updates, the changes by different clients will be merged and all conflicts will be resolved in a meaningful way. Furthermore, to ensure eventual consistency (1), the conflict resolution has to be uniform across replicas, so that, in the end, they converge to the same state.

The protocols achieving this are commonly encapsulated within *replicated data types* [1, 10, 16, 18, 31, 33, 34] that implement *objects*, such as registers, counters, sets or lists, with various conflict-resolution strategies. The strategies can be as simple as establishing a total order on all operations using timestamps and letting the last writer win, but can also be much more subtle. Thus, a data type can detect the presence of a conflict and let the client deal with it: e.g., the *multi-value register* used in Amazon's Dynamo key-value store [18] would return both conflicting zip codes in the above example. A data type can also resolve the conflict in an application-specific way. For example, the *observed-remove set* [7, 32] processes concurrent operations trying to add and remove the same element so that an add always wins, an outcome that may be appropriate for a shopping cart.

Replicated data type implementations are often nontrivial, since they have to maintain not only client-observable object state, but also *metadata* needed to detect and resolve conflicts and to handle network failures. This makes reasoning about their behavior challenging. The situation gets only worse if we consider multi-

POPL '14, January 22–24, 2014, San Diego, CA, USA.

http://dx.doi.org/10.1145/2535838.2535848

ple replicated objects: in this case, asynchronous propagation of updates between replicas may lead to counterintuitive behaviors—*anomalies*, in database terminology. The following code illustrates an anomaly happening in real replicated stores [1, 18]:

$$\begin{array}{l} \text{Replica } r_1 \to x.\texttt{wr}(\textit{post}) \\ \qquad\quad y.\texttt{wr}(\textit{comment}) \end{array} \;\Big\|\; \begin{array}{l} i = y.\texttt{rd}\ \textit{// comment} \leftarrow \text{Replica } r_2 \\ j = x.\texttt{rd}\ \textit{// empty} \end{array} \tag{2}$$

We have two clients reading from and writing to register objects x and y at two different replicas; i and j are client-local variables. The first client makes a post by writing to x at replica r_1 and then comments on the post by writing to y. After every write, replica r_1 might send a message with the update to replica r_2. If the messages carrying the writes of *post* to x and *comment* to y arrive to replica r_2 out of the order they were issued in, the second client can see the comment, but not the post. Different replicated stores may allow such an anomaly or not, and this has to be taken into account when reasoning about them.

In this paper, we propose techniques for reasoning about eventually consistent replicated stores in the following three areas.

1. Specification. We propose a comprehensive framework for specifying the semantics of replicated stores. Its key novel component is *replicated data type specifications* (§3), which provide the first way of specifying the semantics of replicated objects with advanced conflict resolution declaratively, like abstract data types [25]. We achieve this by defining the result of a data type operation not by a function of states, but of *operation contexts*—sets of events affecting the result of the operation, together with some relationships between them. We show that our specifications are sufficiently flexible to handle data types representing a variety of conflict-resolution strategies: last-write-wins register, counter, multi-value register and observed-remove set.

We then specify the semantics of a whole store with multiple objects, possibly of different types, by *consistency axioms* (§7), which constrain the way the store processes incoming requests in the style of weak shared-memory models [2] and thus define the anomalies allowed. As an illustration, we define consistency models used in existing replicated stores, including a weak form of eventual consistency [1, 18] and different kinds of causal consistency [23, 27, 33, 34]. We find that, when specialized to last-writer-wins registers, these specifications are very close to fragments of the C/C++ memory model [5]. Thus, our specification framework generalizes axiomatic shared-memory models to replicated stores with nontrivial conflict resolution.

2. Verification. We propose a method for proving the correctness of replicated data type implementations with respect to our specifications and apply it to seven existing implementations of the four data types mentioned above, including those with nontrivial optimizations. Reasoning about the implementations is difficult due to the highly concurrent nature of a replicated store, with multiple replicas simultaneously updating their object copies and exchanging messages. We address this challenge by proposing *replication-aware simulations* (§5). Like classical simulations from data refinement [21], these associate a concrete state of an implementation with its abstract description—structures on events, in our case. To combat the complexity of replication, they consider the state of an object at a single replica or a message in transit separately and associate it with abstract descriptions of only those events that led to it. Verifying an implementation then requires only reasoning about an instance of its code running at a single replica.

Here, however, we have to deal with another challenge: code at a single replica can access both the state of an object and a message at the same time, e.g., when updating the former upon receiving the latter. To reason about such code, we often need to rely on certain *agreement properties* correlating the abstract descriptions of the message and the object state. Establishing these properties requires global reasoning. Fortunately, we find that agreement properties needed to prove realistic implementations depend only on basic facts about their messaging behavior and can thus be established once for broad classes of data types. Then a particular implementation within such a class can be verified by reasoning purely locally.

By carefully structuring reasoning in this way, we achieve easy and intuitive proofs of single data type implementations. We then lift these results to stores with multiple objects of different types by showing how consistency axioms can be proved given properties of the transport layer and data type implementations (§7).

3. Optimality. Replicated data type designers strive to optimize their implementations; knowing that one is optimal can help guide such efforts in the most promising direction. However, proving optimality is challengingly broad as it requires quantifying over all possible implementations satisfying the same specification.

For most data types we studied, the primary optimization target is the size of the metadata needed to resolve conflicts or handle network failures. To establish optimality of metadata size, we present a novel method for proving lower bounds on the *worst-case metadata overhead* of replicated data types—the proportion of metadata relative to the client-observable content. The main idea is to find a large family of executions of an arbitrary correct implementation such that, given the results of data type operations from a certain fixed point in any of the executions, we can recover the previous execution history. This implies that, across executions, the states at this point are distinct and thus must have some minimal size.

Using our method, we prove that four of the implementations we verified have an optimal worst-case metadata overhead among all implementations satisfying the same specification. Two of these (counter, last-writer-wins register) are well-known; one (optimized observed-remove set [6]) is a recently proposed nontrivial optimization; and one (optimized multi-value register) is a small improvement of a known implementation [33] that we discovered during a failed attempt to prove optimality of the latter. We summarize all the bounds we proved in Fig. 10.

We hope that the theoretical foundations we develop will help in exploring the design space of replicated data types and replicated eventually consistent stores in a systematic way.

2. Replicated Data Types

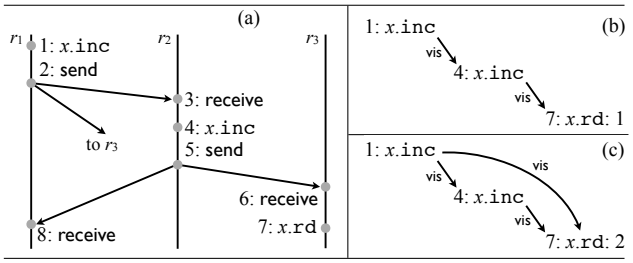
We now describe our formal model for replicated stores and introduce *replicated data type implementations*, which implement operations on a single object at a replica and the protocol used by replicas to exchange updates to this object. Our formalism follows closely the models used by replicated data type designers [33].

A replicated store is organized as a collection of named ***objects*** $\mathsf{Obj} = \{x, y, z, \ldots\}$. Each object is hosted at all ***replicas*** $r, s \in \mathsf{ReplicaID}$. The sets of objects and replicas may be infinite, to model their dynamic creation. Clients interact with the store by performing ***operations*** on objects at a specified replica. Each object $x \in \mathsf{Obj}$ has a ***type*** $\tau = \mathsf{type}(x) \in \mathsf{Type}$, whose ***type signature*** $(\mathsf{Op}_\tau, \mathsf{Val}_\tau)$ determines the set of supported operations Op_τ (ranged over by o) and the set of their return values Val_τ (ranged over by a, b, c, d). We assume that a special value $\bot \in \mathsf{Val}_\tau$ belongs to all sets Val_τ and is used for operations that return no value. For example, we can define a counter data type `ctr` and an integer register type `intreg` with operations for reading, incrementing or writing an integer a: $\mathsf{Val}_{\texttt{ctr}} = \mathsf{Val}_{\texttt{intreg}} = \mathbb{Z} \cup \{\bot\}$, $\mathsf{Op}_{\texttt{ctr}} = \{\texttt{rd}, \texttt{inc}\}$ and $\mathsf{Op}_{\texttt{intreg}} = \{\texttt{rd}\} \cup \{\texttt{wr}(a) \mid a \in \mathbb{Z}\}$.

We also assume sets $\mathsf{Message}$ of messages (ranged over by m) and timestamps $\mathsf{Timestamp}$ (ranged over by t). For simplicity, we let timestamps be positive integers: $\mathsf{Timestamp} = \mathbb{N}_1$.

DEFINITION 1. *A **replicated data type implementation** for a data type τ is a tuple $\mathcal{D}_\tau = (\Sigma, \vec{\sigma}_0, M, \mathsf{do}, \mathsf{send}, \mathsf{receive})$, where $\vec{\sigma}_0$:*

Figure 1. Illustrations of a concrete (a) and two abstract executions (b, c)



$\mathsf{ReplicaID} \to \Sigma$, $M \subseteq \mathsf{Message}$ *and*

$$\mathsf{do} : \mathsf{Op}_\tau \times \Sigma \times \mathsf{Timestamp} \to \Sigma \times \mathsf{Val}_\tau;$$
$$\mathsf{send} : \Sigma \to \Sigma \times M; \qquad \mathsf{receive} : \Sigma \times M \to \Sigma.$$

We denote a component of $\mathcal{D}_\tau$, such as do, by $\mathcal{D}_\tau.\mathsf{do}$. A tuple $\mathcal{D}_\tau$ defines the class of implementations of objects with type τ, meant to be instantiated for every such object in the store. Σ is the set of states (ranged over by σ) used to represent the current state of the object, including metadata, at a single replica. The initial state at every replica is given by $\vec{\sigma}_0$.

$\mathcal{D}_\tau$ provides three methods that the rest of the store implementation can call at a given replica; we assume that these methods execute atomically. We visualize store executions resulting from repeated calls to the methods as in Fig. 1(a), by arranging the calls on several vertical timelines corresponding to replicas at which they occur and denoting the delivery of messages by diagonal arrows. In §4, we formalize them as sequences of transitions called *concrete executions* and define the store semantics by their sets; the intuition given by Fig. 1(a) should suffice for the following discussion.

A client request to perform an operation $o \in \mathsf{Op}_\tau$ triggers the call $\mathsf{do}(o, \sigma, t)$ (e.g., event 1 in Fig. 1(a)). This takes the current state $\sigma \in \Sigma$ of the object at the replica where the request is issued and a timestamp $t \in \mathsf{Timestamp}$ provided by the rest of the store implementation and produces the updated object state and the return value of the operation. The data type implementation can use the timestamp provided, e.g., to implement the last-writer-wins conflict-resolution strategy mentioned in §1, but is free to ignore it.

Nondeterministically, in moments when the network is able to accept messages, a replica calls send. Given the current state of the object at the replica, send produces a message in M to broadcast to all other replicas (event 2 in Fig. 1(a)); sometimes send also alters the state of the object. Using broadcast rather than point-to-point communication does not limit generality, since we can always tag messages with the intended receiver. Another replica that receives the message generated by send calls receive to merge the enclosed update into its copy of the object state (event 3 in Fig. 1(a)).

We now reproduce three replicated data type implementations due to Shapiro et al. [33].They fall into two categories: in ***op-based*** implementations, each message carries a description of the latest operations that the sender has performed, and in ***state-based*** implementations, a description of *all* operations it knows about.

Op-based counter (`ctr`). Fig. 2(a) shows an implementation of the `ctr` data type. A replica stores a pair $\langle a, d\rangle$, where a is the current value of the counter, and d is the number of increments performed since the last broadcast (we use angle brackets for tuples representing states and messages). The send method returns d and resets it; the receive method adds the content of the message to a. This implementation is correct, as long as each message is delivered exactly once (we show how to prove this in §5). Since `inc` operations commute, they never conflict: applying them in different orders at different replicas yields the same final state.

State-based counter (`ctr`). The implementation in Fig. 2(b) summarizes the currently known history by recording the contribution of every replica to the counter value separately (reminiscent of vector clocks [29]). A replica stores its identifier r and a vector v such that for each replica s the entry $v(s)$ gives the number of increments made by clients at s that have been received by r. A `rd` operation returns the sum of all entries in the vector. An `inc` operation increments the entry for the current replica. We denote by $v[i \mapsto j]$ the function that has the same value as v everywhere, except for i, where it has the value j. The send method returns the vector, and the receive method takes the maximum of each entry in the vectors v and v' given to it. This is correct because an entry for s in either vector reflects a prefix of the sequence of increments done at replica s. Hence, we know that $\min\{v(s), v'(s)\}$ increments by s are taken into account both in $v(s)$ and in $v'(s)$.

Figure 2. Three replicated data type implementations

(a) Op-based counter (`ctr`)

Σ	$= \mathbb{N}_0 \times \mathbb{N}_0$	$\mathsf{do}(\mathtt{rd}, \langle a, d\rangle, t)$	$= (\langle a, d\rangle, a)$
M	$= \mathbb{N}_0$	$\mathsf{do}(\mathtt{inc}, \langle a, d\rangle, t)$	$= (\langle a+1, d+1\rangle, \bot)$
$\vec{\sigma}_0$	$= \lambda r.\, \langle 0, 0\rangle$	$\mathsf{send}(\langle a, d\rangle)$	$= (\langle a, 0\rangle, d)$
		$\mathsf{receive}(\langle a, d\rangle, d')$	$= \langle a + d', d\rangle$

(b) State-based counter (`ctr`)

Σ	$=$	$\mathsf{ReplicaID} \times (\mathsf{ReplicaID} \to \mathbb{N}_0)$
σ_0	$=$	$\lambda r.\, \langle r, \lambda s.\, 0\rangle$
M	$=$	$\mathsf{ReplicaID} \to \mathbb{N}_0$
$\mathsf{do}(\mathtt{rd}, \langle r, v\rangle, t)$	$=$	$(\langle r, v\rangle, \sum\{v(s) \mid s \in \mathsf{ReplicaID}\})$
$\mathsf{do}(\mathtt{inc}, \langle r, v\rangle, t)$	$=$	$(\langle r, v[r \mapsto v(r)+1]\rangle, \bot)$
$\mathsf{send}(\langle r, v\rangle)$	$=$	$(\langle r, v\rangle, v)$
$\mathsf{receive}(\langle r, v\rangle, v')$	$=$	$\langle r, (\lambda s.\, \max\{v(s), v'(s)\})\rangle$

(c) State-based last-writer-wins register (`intreg`)

Σ	$=$	$\mathbb{Z} \times (\mathsf{Timestamp} \uplus \{0\})$
$\vec{\sigma}_0$	$=$	$\lambda r.\, \langle 0, 0\rangle$
M	$=$	Σ
$\mathsf{do}(\mathtt{rd}, \langle a, t\rangle, t')$	$=$	$(\langle a, t\rangle, a)$
$\mathsf{do}(\mathtt{wr}(a'), \langle a, t\rangle, t')$	$=$	if $t < t'$ then $(\langle a', t'\rangle, \bot)$ else $(\langle a, t\rangle, \bot)$
$\mathsf{send}(\langle a, t\rangle)$	$=$	$(\langle a, t\rangle, \langle a, t\rangle)$
$\mathsf{receive}(\langle a, t\rangle, \langle a', t'\rangle)$	$=$	if $t < t'$ then $\langle a', t'\rangle$ else $\langle a, t\rangle$

State-based last-writer-wins (LWW) register (`intreg`). Unlike counters, registers have update operations that are not commutative. To resolve conflicts, the implementation in Fig. 2 uses the last-writer-wins strategy, creating a total order on writes by associating a unique timestamp with each of them. A state contains the current value, returned by `rd`, and the timestamp at which it was written (initially, we have 0 instead of a timestamp). A $\mathtt{wr}(a')$ compares its timestamp t' with the timestamp t of the current value a and sets the value to the one with the highest timestamp. Note that here we have to allow for $t' < t$, since we do not make any assumptions about timestamps apart from uniqueness: e.g., the rest of the store implementation can compute them using physical or Lamport clocks [22]. We show how to state assumptions about timestamps in §4. The send method just returns the state, and the receive method chooses the winning value by comparing the timestamps in the current state and the message, like `wr`.

State-based vs. op-based. State-based implementations converge to a consistent state faster than op-based implementations because they are ***transitively delivering***, meaning that they can propagate updates indirectly. For example, when using the counter in Fig. 2(b), in the execution in Fig. 1(a) the read at r_3 (event 7) returns 2, even though the message from r_1 has not arrived yet, because r_3 learns about r_1's update via r_2. State-based implementations are also resilient against transport failures like message *loss, reordering, or duplication*. Op-based implementations require the replicated store using them to mask such failures (e.g., using message sequence numbers, retransmission buffers, or reorder buffers).

The potential weakness of state-based implementations is the size of states and messages, which motivates our examination of space optimality in §6. For example, we show that the counter in Fig. 2(b) is optimal, meaning that no counter implementation satisfying the same requirements (transitive delivery and resilience against message loss, reordering, and duplication) can do better.

3. Specifying Replicated Data Types and Stores

Consider the concrete execution in Fig. 1(a). What are valid return values for the read in event 7? Intuitively, 1 or 2 can be justifiable, but not 100. We now present a framework for specifying the expected outcome declaratively, without referring to implementation details. For example, we give a specification of a replicated counter that is satisfied by both implementations in Fig. 2(a, b).

In presenting the framework, we rely on the intuitive understanding of the way a replicated store executes given in §2. Later we define the store semantics formally (§4), which lets us state what it means for a store to satisfy our specifications (§4 and §7).

3.1 Abstract Executions and Specification Structure

We define our specifications on *abstract executions*, which include only user-visible events (corresponding to do calls) and describe the other information about the store processing in an implementation-independent form. Informally, we consider a concrete execution correct if it can be justified by an abstract execution satisfying the specifications that is "similar" to it and, in particular, has the same operations and return values.

Abstract executions are inspired by axiomatic definitions of weak shared-memory models [2]. In particular, we use their previously proposed reformulation with visibility and arbitration relations [13], which are similar to the reads-from and coherence relations from weak shared-memory models. We provide a comparison with shared-memory models in §7 and with [13] in §8.

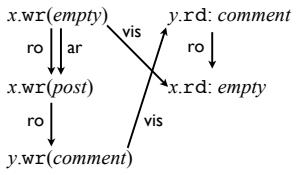
DEFINITION 2. *An **abstract execution** is a tuple* $A = (E, \mathsf{repl}, \mathsf{obj}, \mathsf{oper}, \mathsf{rval}, \mathsf{ro}, \mathsf{vis}, \mathsf{ar})$*, where*

- $E \subseteq \mathsf{Event}$ *is a set of events from a countable universe* Event*;*
- *each event* $e \in E$ *describes a replica* $\mathsf{repl}(e) \in \mathsf{ReplicaID}$ *performing an operation* $\mathsf{oper}(e) \in \mathsf{Op}_{\mathsf{type}(\mathsf{obj}(e))}$ *on an object* $\mathsf{obj}(e) \in \mathsf{Obj}$*, which returns the value* $\mathsf{rval}(e) \in \mathsf{Val}_{\mathsf{type}(\mathsf{obj}(e))}$*;*
- $\mathsf{ro} \subseteq E \times E$ *is a **replica order**, which is a union of transitive, irreflexive and total orders on events at each replica;*
- $\mathsf{vis} \subseteq E \times E$ *is an acyclic **visibility relation** such that* $\forall e, f \in E.\, e \xrightarrow{\mathsf{vis}} f \implies \mathsf{obj}(e) = \mathsf{obj}(f)$*;*
- $\mathsf{ar} \subseteq E \times E$ *is an **arbitration relation**, which is a union of transitive, irreflexive and total orders on events on each object.*

We also require that ro*,* vis *and* ar *be well-founded.*

In the following, we denote components of A and similar structures as in $A.\mathsf{repl}$. We also use $(e, f) \in \mathsf{r}$ and $e \xrightarrow{\mathsf{r}} f$ interchangeably.

Informally, $e \xrightarrow{\mathsf{vis}} f$ means that f is aware of e and thus e's effect can influence f's return value. In implementation terms, this may be the case if the update performed by e has been delivered to the replica performing f before f is issued. The exact meaning of "delivered", however, depends on how much information messages carry in the implementation. For example, as we explain in §3.2, the return value of a read from a counter is equal to the number of `inc` operations visible to it. Then, as we formalize in §4, the abstract execution illustrated in Fig. 1(b) justifies the op-based implementation in Fig. 2(a) reading 1 in the concrete execution in Fig. 1(a). The abstract execution in Fig. 1(c) justifies the state-based implementation in Fig. 2(b) reading 2 due to transitive delivery (§2). There is no abstract execution that would justify reading 100.



The ar relation represents the ordering information provided by the store, e.g., via timestamps. On the right we show an abstract execution corresponding to a variant of the anomaly (2). The ar edge means that any replica that sees both writes to x should assume that *post* overwrites *empty*.

We give a store specification by two components, constraining abstract executions:

1. ***Replicated data type specifications*** determine return values of operations in an abstract execution in terms of its vis and ar relations, and thus define conflict-resolution policies for individual objects in the store. The specifications are the key novel component of our framework, and we discuss them next.
2. ***Consistency axioms*** constrain vis and ar and thereby disallow anomalies and extend the semantics of individual objects to that of the entire store. We defer their discussion to §7. See Fig. 13 for their flavor; in particular, COCV prohibits the anomaly above.

Each of these components can be varied separately, and our specifications will define the semantics of any possible combination. Given a specification of a store, we can determine whether a set of events can be observed by its users by checking if there is an abstract execution with this set of events satisfying the data type specifications and consistency axioms.

3.2 Replicated Data Type Specifications

In a sequential setting, the semantics of a data type τ can be specified by a function $\mathcal{S}_\tau : \mathsf{Op}_\tau^+ \to \mathsf{Val}_\tau$, which, given a non-empty sequence of operations performed on an object, specifies the return value of the last operation. For a register, read operations return the value of the last preceding write, or zero if there is no prior write. For a counter, read operations return the number of preceding increments. Thus, for any sequence of operations ξ:

$$\begin{aligned} \mathcal{S}_{\mathtt{intreg}}(\xi\,\mathtt{rd}) &= a, \text{ if } \mathtt{wr}(0)\,\xi = \xi_1\,\mathtt{wr}(a)\,\xi_2 \text{ and} \\ &\qquad \xi_2 \text{ does not contain } \mathtt{wr} \text{ operations;} \\ \mathcal{S}_{\mathtt{ctr}}(\xi\,\mathtt{rd}) &= (\text{the number of } \mathtt{inc} \text{ operations in } \xi); \\ \mathcal{S}_{\mathtt{intreg}}(\xi\,\mathtt{inc}) &= \mathcal{S}_{\mathtt{ctr}}(\xi\,\mathtt{wr}(a)) = \bot. \end{aligned}$$

In a replicated store, the story is more interesting. We specify a data type τ by a function $\mathcal{F}_\tau$, generalizing $\mathcal{S}_\tau$. Just like $\mathcal{S}_\tau$, this determines the return value of an operation based on prior operations performed on the object. However, $\mathcal{F}_\tau$ takes as a parameter not a sequence, but an *operation context*, which includes all we need to know about a store execution to determine the return value of a given operation o—the set E of all events that are visible to o, together with the operations performed by the events and visibility and arbitration relations on them.

DEFINITION 3. *An **operation context** for a data type τ is a tuple $L = (o, E, \mathsf{oper}, \mathsf{vis}, \mathsf{ar})$, where $o \in \mathsf{Op}_\tau$, E is a finite subset of* Event*,* $\mathsf{oper} : E \to \mathsf{Op}_\tau$*,* $\mathsf{vis} \subseteq E \times E$ *is acyclic and* $\mathsf{ar} \subseteq E \times E$ *is transitive, irreflexive and total.*

We can extract the context of an event $e \in A.E$ in an abstract execution A by selecting all events visible to it according to $A.\mathsf{vis}$:

$$\mathsf{ctxt}(A, e) = (A.\mathsf{oper}(e), G, (A.\mathsf{oper})|_G, (A.\mathsf{vis})|_G, (A.\mathsf{ar})|_G),$$

where $G = (A.\mathsf{vis})^{-1}(e)$ and $\cdot|_G$ is the restriction to events in G. Thus, in the abstract execution in Fig. 1(b), the operation context of the read from x includes only one increment event; in the execution in Fig. 1(c) it includes two.

DEFINITION 4. *A **replicated data type specification** for a type τ is a function $\mathcal{F}_\tau$ that, given an operation context L for τ, specifies a return value $\mathcal{F}_\tau(L) \in \mathsf{Val}_\tau$.*

Note that $\mathcal{F}_\tau(o, \emptyset, \ldots)$ returns the value resulting from performing o on the initial state for the data type (e.g., 0 for the LWW-register).

We specify multiple data types used in a replicated store by a partial function $\mathbb{F}$ mapping them to data type specifications.

DEFINITION 5. *An abstract execution A **satisfies** $\mathbb{F}$, written $A \models \mathbb{F}$, if the return value of every event in A is computed on its context by the specification for the type of the object the event accesses:*

$$\forall e \in A.E.\ (A.\mathsf{rval}(e) = \mathbb{F}(\mathsf{type}(A.\mathsf{obj}(e)))(\mathsf{ctxt}(A, e))).$$

We specify a whole store by $\mathbb{F}$ and a set of consistency axioms (§7). This lets us determine if its users can observe a given set of events by checking if there is an abstract execution with these events that satisfies $\mathbb{F}$ according to the above definition, as well as the axioms.

Note that $\mathcal{F}_\tau$ is deterministic. This does not mean that so is an outcome of an operation on a store; rather, that all the nondeterminism arising due to its distributed nature is resolved by vis and ar in the context passed to $\mathcal{F}_\tau$. These relations are chosen arbitrarily subject to consistency axioms. Due to the determinacy property, two events that perform the same operation and see the same set of events produce the same return values. As we show in §7, this property ensures that our specifications can formalize eventual consistency in the sense of (1).

We now give four examples of data type specifications, corresponding to the four conflict-resolution strategies mentioned in §1 and §2: (1) operations commute, so no conflicts arise; (2) last writer wins; (3) all conflicting values are returned; and (4) conflicts are resolved in an application-specific way. We start by specifying the data types whose implementations we presented in §2.

1. Counter (`ctr`) is defined by

$$\begin{aligned} \mathcal{F}_{\texttt{ctr}}(\texttt{inc}, E, \mathsf{oper}, \mathsf{vis}, \mathsf{ar}) &= \bot; \\ \mathcal{F}_{\texttt{ctr}}(\texttt{rd}, E, \mathsf{oper}, \mathsf{vis}, \mathsf{ar}) &= \big|\{e \in E \mid \mathsf{oper}(e) = \texttt{inc}\}\big|. \end{aligned} \quad (3)$$

Thus, according to Def. 5 the executions in Fig. 1(b) and 1(c) satisfy the counter specification: both 1 and 2 are valid return values for the read from x when there are two concurrent increments.

2. LWW-register (`intreg`) is defined by

$$\mathcal{F}_{\texttt{intreg}}(o, E, \mathsf{oper}, \mathsf{vis}, \mathsf{ar}) = \mathcal{S}_{\texttt{intreg}}(E^{\mathsf{ar}} o), \quad (4)$$

where E^{ar} denotes the sequence obtained by ordering the operations performed by the events in E according to ar. Thus, the return value is determined by establishing a total order of the visible operations and applying the regular sequential semantics. For example, by Def. 5 in the example execution from §3.1 the read from x has to return *empty*; if we had a vis edge from the write of *post* to the read from x, then the read would have to return *post*. As we show in §7, weak shared-memory models are obtained by specializing our framework to stores with only LWW-registers.

We can obtain a concurrent semantics $\mathcal{F}_\tau$ of any data type τ based on its sequential semantics $\mathcal{S}_\tau$ similarly to (4). For example, $\mathcal{F}_{\texttt{ctr}}$ defined above is equivalent to what we obtain using this generic construction. The next two examples go beyond this.

3. Multi-value register (`mvr`). This register [1, 18] has the same operations as the LWW-register, but its reads return a set of values:

$$\begin{aligned} \mathcal{F}_{\texttt{mvr}}(\texttt{rd}, E, \mathsf{oper}, \mathsf{vis}, \mathsf{ar}) = \{a \mid \exists e \in E.\ \mathsf{oper}(e) = \texttt{wr}(a) \\ \wedge \neg\exists f \in E.\ \mathsf{oper}(f) = \texttt{wr}(\_) \wedge e \xrightarrow{\mathsf{vis}} f\}. \end{aligned}$$

(We write _ for an expression whose value is irrelevant.) A read returns the values written by currently conflicting writes, defined as those that are not superseded in vis by later writes; ar is not used. For example, a `rd` would return $\{2, 3\}$ in the context on the right.

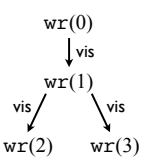


Figure 3. The set of configurations Config and the transition relation $\longrightarrow_{\mathbb{D}}$: Config × Event × Config for a data type library $\mathbb{D}$. We use $e : \{h_1 = u_1, h_2 = u_2\}$ to abbreviate $h_1(e) = u_1$ and $h_2(e) = u_2$. We uncurry $R \in \mathsf{RState}$ where convenient.

$$\begin{aligned} \mathsf{Obj}_\tau &= \{x \in \mathsf{Obj} \mid \mathsf{type}(x) = \tau\} \\ \mathsf{RState} &= \textstyle\bigcup_{X \subseteq \mathsf{Obj}} \prod_{x \in X} (\mathsf{ReplicaID} \to \mathbb{D}(\mathsf{type}(x)).\Sigma) \\ \mathsf{TState} &= \mathsf{MessageID} \rightharpoonup \textstyle\bigcup_{\tau \in \mathsf{Type}} (\mathsf{ReplicaID} \times \mathsf{Obj}_\tau \times \mathbb{D}(\tau).M) \\ \mathsf{Config} &= \mathsf{RState} \times \mathsf{TState} \end{aligned}$$

$$\frac{\mathbb{D}(\mathsf{type}(x)).\mathsf{do}(o, \sigma, t) = (\sigma', a) \quad e : \{\mathsf{act} = \mathsf{do},\ \mathsf{obj} = x,\ \mathsf{repl} = r,\ \mathsf{oper} = o,\ \mathsf{time} = t,\ \mathsf{rval} = a\}}{(R[(x,r) \mapsto \sigma],\ T) \xrightarrow{e}_{\mathbb{D}} (R[(x,r) \mapsto \sigma'],\ T)}$$

$$\frac{\mathbb{D}(\mathsf{type}(x)).\mathsf{send}(\sigma) = (\sigma', m) \qquad mid \notin \mathsf{dom}(T) \quad e : \{\mathsf{act} = \mathsf{send},\ \mathsf{obj} = x,\ \mathsf{repl} = r,\ \mathsf{msg} = mid\}}{(R[(x,r) \mapsto \sigma],\ T) \xrightarrow{e}_{\mathbb{D}} (R[(x,r) \mapsto \sigma'],\ T[mid \mapsto (r, x, m)])}$$

$$\frac{\mathbb{D}(\mathsf{type}(x)).\mathsf{receive}(\sigma, m) = \sigma' \qquad r \neq r' \quad e : \{\mathsf{act} = \mathsf{receive},\ \mathsf{obj} = x,\ \mathsf{repl} = r,\ \mathsf{srepl} = r',\ \mathsf{msg} = mid\}}{(R[(x,r) \mapsto \sigma],\ T[mid \mapsto (r', x, m)]) \xrightarrow{e}_{\mathbb{D}} (R[(x,r) \mapsto \sigma'],\ T[mid \mapsto (r', x, m)])}$$

4. Observed-remove set (`orset`). How do we specify a replicated set of integers? The operations of adding and removing different elements commute and thus do not conflict. Conflicts arise from concurrently adding and removing the same element. For example, we need to decide what `rd` will return as the contents of the set in the context $(\texttt{rd}, \{e, f\}, \mathsf{oper}, \mathsf{vis}, \mathsf{ar})$, where $\mathsf{oper}(e) = \texttt{add}(42)$ and $\mathsf{oper}(f) = \texttt{remove}(42)$. If we use the generic construction from the LWW-register, the result will depend on the arbitration relation: $\emptyset$ if $e \xrightarrow{\mathsf{ar}} f$, and $\{42\}$ otherwise. An application may require a more consistent behavior, e.g., that an add operation always win against concurrent remove operations. Observed-remove (OR) set [7, 32] achieves this by mandating that `remove` operations cancel only the `add` operations that are visible to them:

$$\begin{aligned} \mathcal{F}_{\texttt{orset}}(\texttt{rd}, E, \mathsf{oper}, \mathsf{vis}, \mathsf{ar}) = \{a \mid \exists e \in E.\ \mathsf{oper}(e) = \texttt{add}(a) \\ \wedge \neg\exists f \in E.\ \mathsf{oper}(f) = \texttt{remove}(a) \wedge e \xrightarrow{\mathsf{vis}} f\}, \end{aligned} \quad (5)$$

In the above operation context `rd` will return $\emptyset$ if $e \xrightarrow{\mathsf{vis}} f$, and $\{42\}$ otherwise. The rationale is that, in the former case, $\texttt{add}(42)$ and $\texttt{remove}(42)$ are not concurrent: the user who issued the `remove` knew that 42 was in the set and thus meant to remove it. In the latter case, the two operations are concurrent and thus `add` wins.

As the above examples illustrate, our specifications can describe the semantics of data types and their conflict-resolution policies declaratively, without referring to the internals of their implementations. In this sense the specifications generalize the concept of an *abstract data type* [25] to the replicated setting.

4. Store Semantics and Data Type Correctness

A ***data type library*** $\mathbb{D}$ is a partial mapping from types τ to data type implementations $\mathbb{D}(\tau)$ from Def. 1. We now define the semantics of a replicated store with a data type library $\mathbb{D}$ as a set of its *concrete executions*, previously introduced informally by Fig. 1(a). We then state what it means for data type implementations of §2 to satisfy their specifications of §3.2 by requiring their concrete executions to be justified by abstract ones. In §7 we generalize this to the correctness of the whole store with multiple object with respect to both data type specifications and consistency axioms.

Semantics. We define the semantics using the relation $\longrightarrow_{\mathbb{D}}$: Config × Event × Config in Fig. 3, which describes a single

step of the store execution. The relation transforms ***configurations*** $(R, T) \in \mathsf{Config}$ describing the store state: R gives the object state at each replica, and T the set of messages in transit between them, each identified by a message identifier $mid \in \mathsf{MessageID}$. A message is annotated by the origin replica and the object to which it pertains. We allow the store to contain only some objects from Obj and thus allow R to be partial on them. We use a number of functions on events, such as act, obj, etc., to record the information about the corresponding transitions, so that $\longrightarrow_{\mathbb{D}}$ is implicitly parameterized by them; we give their full list in Def. 6 below.

The first rule in Fig. 3 describes a replica r performing an operation o on an object x using the do method of the corresponding data type implementation. We record the return value using the function rval. To communicate the change to other replicas, we can at any time perform a transition defined by the second rule, which puts a new message m created by a call to send into the set of messages in transit. The third rule describes the delivery of such a message to a replica r other than the origin replica r', which triggers a call to $\mathsf{receive}$. Note that the relation $\longrightarrow_{\mathbb{D}}$ does not make any assumptions about message delivery: messages can be delivered in any order, multiple times, or not at all. These assumptions can be introduced separately, as we show later in this section. A concrete execution can be thought of as a finite or infinite sequence of transitions:

$$(R_0, T_0) \xrightarrow{e_1} (R_1, T_1) \xrightarrow{e_2} \dots \xrightarrow{e_n} (R_n, T_n) \dots,$$

where all events e_i are distinct. To ease mapping between concrete and abstract executions in the future, we formalize it as a structure on events, similarly to Def. 2.

DEFINITION 6. *A **concrete execution** of a store with a data type library $\mathbb{D}$ is a tuple*

$$C = (E, \mathsf{eo}, \mathsf{pre}, \mathsf{post}, \mathsf{act}, \mathsf{obj}, \mathsf{repl}, \mathsf{oper}, \mathsf{time}, \mathsf{rval}, \mathsf{msg}, \mathsf{srepl}).$$

*Here $E \subseteq \mathsf{Event}$, the **execution order** eo is a well-founded, transitive, irreflexive and total order on E, relating the events according to the order of the transitions they describe, time is injective and $\mathsf{pre}, \mathsf{post} : E \to \mathsf{Config}$ form a valid sequence of transitions:*

$$(\forall e \in E.\, \mathsf{pre}(e) \xrightarrow{e}_{\mathbb{D}} \mathsf{post}(e)) \wedge$$
$$(\forall e, f \in E.\, e \xrightarrow{\mathsf{eo}} f \wedge \neg\exists g.\, e \xrightarrow{\mathsf{eo}} g \xrightarrow{\mathsf{eo}} f \implies \mathsf{post}(e) = \mathsf{pre}(f)).$$

We have omitted the types of functions on events, which are easily inferred from Fig. 3: e.g., $\mathsf{act} : E \to \{\mathsf{do}, \mathsf{send}, \mathsf{receive}\}$ and $\mathsf{time} : E \rightharpoonup \mathsf{Timestamp}$, defined only on e with $\mathsf{act}(e) = \mathsf{do}$.

We denote the initial configuration of C by $\mathsf{init}(C) = C.\mathsf{pre}(e_0)$, where e_0 is the minimal event in $C.\mathsf{eo}$. If $C.E$ is finite, we denote the final configuration of C by $\mathsf{final}(C) = C.\mathsf{post}(e_\mathbf{f})$, where $e_\mathbf{f}$ is the maximal event in $C.\mathsf{eo}$. The ***semantics*** $[\![\mathbb{D}]\!]$ of $\mathbb{D}$ is the set of all its concrete executions C that start in a configuration with an empty set of messages and all objects in initial states, i.e.,

$$\exists X \subseteq \mathsf{Obj}.\, \mathsf{init}(C) = ((\lambda x \in X.\, \mathbb{D}(\mathsf{type}(x)).\vec{\sigma}_0), [\,]),$$

where $[\,]$ is the everywhere-undefined function.

Transport layer specifications. Data type implementations such as the op-based counter in Fig. 2(a) can rely on some guarantees concerning the delivery of messages ensured by the rest of the store implementation. They may similarly assume certain properties of timestamps other than uniqueness (guaranteed by the injectivity of time in Def. 6). We take such assumptions into account by admitting only a subset of executions from $[\![\mathbb{D}]\!]$ that satisfy a ***transport layer specification*** $\mathcal{T}$, which is a predicate on concrete executions. Thus, we consider a ***replicated store*** to be defined by a pair $(\mathbb{D}, \mathcal{T})$ and the set of its executions be $[\![\mathbb{D}]\!] \cap \mathcal{T}$.

Even though our definition of $\mathcal{T}$ lets it potentially restrict data type implementation internals, the particular instantiations we use only restrict message delivery and timestamps. For technical reasons, we assume that $\mathcal{T}$ always satisfies certain closure properties: for every $C \in \mathcal{T}$, the projection of C onto events on a given object or a subset of events forming a prefix in the eo order is also in $\mathcal{T}$.

As an example, we define a transport layer specification ensuring that a message is delivered to any single replica at most once, as required by the implementation in Fig. 2(a). Let the ***delivery relation*** $\mathsf{del}(C) \in C.E \times C.E$ pair events sending and receiving the same message:

$$e \xrightarrow{\mathsf{del}(C)} f \iff e \xrightarrow{C.\mathsf{eo}} f \wedge C.\mathsf{act}(e) = \mathsf{send} \wedge C.\mathsf{act}(f) = \mathsf{receive} \wedge C.\mathsf{msg}(e) = C.\mathsf{msg}(f).$$

Then the desired condition on concrete executions C is

$$\forall e, f, g \in C.E.\, e \xrightarrow{\mathsf{del}(C)} f \wedge e \xrightarrow{\mathsf{del}(C)} g \wedge C.\mathsf{repl}(f) = C.\mathsf{repl}(g) \implies f = g. \quad (\mathsf{T\text{-}Unique})$$

Data type implementation correctness. We now state what it means for an implementation $\mathcal{D}_\tau$ of type τ from Def. 1 to satisfy a specification $\mathcal{F}_\tau$ from Def. 4. To this end, we consider the behavior of $\mathcal{D}_\tau$ under the "most general" client and transport layer, performing all possible operations and message deliveries. Formally, let $[\![\mathcal{D}_\tau]\!]$ be the set of executions $C \in [\![\tau \mapsto \mathcal{D}_\tau]\!]$ of a store containing a single object x of a type τ with the implementation $\mathcal{D}_\tau$, i.e., $\mathsf{init}(C) = (R, [\,])$ for some R such that $\mathsf{dom}(R) = \{x\}$.

Then $\mathcal{D}_\tau$ should satisfy $\mathcal{F}_\tau$ under a transport specification $\mathcal{T}$ if for every concrete execution $C \in [\![\mathcal{D}_\tau]\!] \cap \mathcal{T}$ we can find a "similar" abstract execution satisfying $\mathcal{F}_\tau$ and, in particular, having the same operations and return values. As it happens, all components of the abstract execution except visibility are straightforwardly determined by C; as explained in §3.1, we have some freedom in choosing visibility. We define the choice using a ***visibility witness*** $\mathcal{V}$, which maps a concrete execution $C \in [\![\mathcal{D}_\tau]\!]$ to an acyclic relation on $(C.E)|_{\mathsf{do}}$ defining visibility (here $\cdot|_{\mathsf{do}}$ is the restriction to events e with $C.\mathsf{act}(e) = \mathsf{do}$). Let

$$e \xrightarrow{\mathsf{ro}(C)} f \iff e \xrightarrow{C.\mathsf{eo}} f \wedge C.\mathsf{repl}(e) = C.\mathsf{repl}(f);$$
$$e \xrightarrow{\mathsf{ar}(C)} f \iff e, f \in (C.E)|_{\mathsf{do}} \wedge C.\mathsf{obj}(e) = C.\mathsf{obj}(f) \wedge C.\mathsf{time}(e) < C.\mathsf{time}(f).$$

Then the abstract execution justifying $C \in [\![\mathcal{D}_\tau]\!]$ is defined by

$$\mathsf{abs}(C, \mathcal{V}) = (C.E|_{\mathsf{do}}, E.\mathsf{repl}|_{\mathsf{do}}, E.\mathsf{obj}|_{\mathsf{do}}, E.\mathsf{oper}|_{\mathsf{do}}, E.\mathsf{rval}|_{\mathsf{do}}, \mathsf{ro}(C)|_{\mathsf{do}}, \mathcal{V}(C), \mathsf{ar}(C)).$$

DEFINITION 7. *A data type implementation $\mathcal{D}_\tau$ **satisfies** a specification $\mathcal{F}_\tau$ with respect to $\mathcal{V}$ and $\mathcal{T}$, written $\mathcal{D}_\tau\ \mathsf{sat}[\mathcal{V}, \mathcal{T}]\ \mathcal{F}_\tau$, if $\forall C \in [\![\mathcal{D}_\tau]\!] \cap \mathcal{T}.\, (\mathsf{abs}(C, \mathcal{V}) \models [\tau \mapsto \mathcal{F}_\tau])$, where $\models$ is defined in Def. 5.*

As we explained informally in §3.1, the visibility witness depends on how much information the implementation puts into messages. Since state-based implementations, such as the ones in Fig. 2(b, c), are transitively delivering (§2), for them we use the witness $\mathcal{V}^{\mathsf{state}}(C) = (\mathsf{ro}(C) \cup \mathsf{del}(C))^+|_{\mathsf{do}}$. By the definition of $\mathsf{ro}(C)$ and $\mathsf{del}(C)$, $(\mathsf{ro}(C) \cup \mathsf{del}(C))$ is acyclic, so $\mathcal{V}^{\mathsf{state}}$ is well-defined. State-based implementations do not make any assumptions about the transport layer: in this case we write $\mathcal{T} = \mathsf{T\text{-}Any}$. In contrast, op-based implementations, such as the one in Fig. 2(a), require $\mathcal{T} = \mathsf{T\text{-}Unique}$. Since such implementations are not transitively delivering, the witness $\mathcal{V}^{\mathsf{state}}$ is not appropriate for them. We could attempt to define a witness for them by straightforwardly lifting the delivery relation:

$$\mathcal{V}^{\mathsf{op}}(C) = \mathsf{ro}(C)|_{\mathsf{do}} \cup \{(e, f) \mid e, f \in (C.E)|_{\mathsf{do}} \wedge \exists e', f'.\, e \xrightarrow{\mathsf{ro}(C)} e' \xrightarrow{\mathsf{del}(C)} f' \xrightarrow{\mathsf{ro}(C)} f\}.$$

However, we need to be more careful, since for op-based implementations $e \xrightarrow{\mathsf{ro}(C)} e' \xrightarrow{\mathsf{del}(C)} f' \xrightarrow{\mathsf{ro}(C)} f$ does not ensure that the update of e is taken into account by f: if there is another send event e'' in between e and e', then e'' will capture the update of e and e' will not. Hence, we define the witness as:

$$\begin{aligned}\mathcal{V}^{\mathsf{op}}(C) = \mathsf{ro}(C)|_{\mathsf{do}} \cup \{(e,f) \mid e,f \in (C.E)|_{\mathsf{do}} \\ \wedge\, \exists e', f'.\, e \xrightarrow{\mathsf{ro}(C)} e' \xrightarrow{\mathsf{del}(C)} f' \xrightarrow{\mathsf{ro}(C)} f \\ \wedge\, \neg\exists e''.\, e \xrightarrow{\mathsf{ro}(C)} e'' \xrightarrow{\mathsf{ro}(C)} e' \wedge C.\mathsf{act}(e'') = \mathsf{send}\}.\end{aligned}$$

We next present a method for proving data type implementation correctness in the sense of Def. 7. In §7 we lift this to stores with multiple objects and take into account consistency axioms.

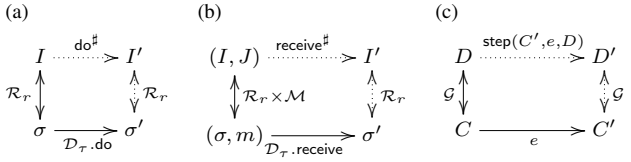
5. Proving Data Type Implementations Correct

The straightforward approach to proving correctness in the sense of Def. 7 would require us to consider global store configurations in executions C, including object states at all replicas and all messages in transit, making the reasoning non-modular and unintuitive. To deal with this challenge, we focus on a single component of a store configuration using *replication-aware simulation relations* $\mathcal{R}_r$ and $\mathcal{M}$, analogous to simulation (aka coupling) relations used in data refinement [21]. The $\mathcal{R}_r$ relation associates the object state at a replica r with an abstract execution that describes *only those events that led to this state*; $\mathcal{M}$ does the same for a message. For example, when proving $\mathcal{D}_{\mathtt{ctr}}$ in Fig. 2(b) with respect to $\mathcal{F}_{\mathtt{ctr}}$ in (3), $\mathcal{M}$ associates a message carrying a vector v with executions in which each replica s makes $v(s)$ increments. As part of a proof of $\mathcal{D}_\tau$, we require checking that the effect of its methods, such as $\mathcal{D}_\tau.\mathsf{do}$, can be simulated by appropriately transforming related abstract executions while preserving the relations. We define these transformations using *abstract methods* $\mathsf{do}^\sharp$, $\mathsf{send}^\sharp$ and $\mathsf{receive}^\sharp$ as illustrated in Fig. 4(a, b). For example, if a replica r executes $\mathcal{D}_\tau.\mathsf{do}$ from a state σ related by $\mathcal{R}_r$ to an abstract execution I (we explain the use of I instead of A later), we need to find an I' related by $\mathcal{R}_r$ to the resulting state σ'. We also need to check that the value returned by $\mathcal{D}_\tau.\mathsf{do}$ on σ is equal to that returned by $\mathcal{F}_\tau$ on I.

These conditions consider the behavior of an implementation method on a single state and/or message and its effect on only the relevant part of the abstract execution. However, by localizing the reasoning in this way, we lose some global information that is actually required to verify realistic implementations. In particular, this occurs when discharging the obligation for $\mathsf{receive}$ in Fig. 4(b). Taking a global view, σ and m there are meant to come from the same configuration in a concrete execution C; correspondingly, I and J are meant to be fragments of the same abstract execution $\mathsf{abs}(C, \mathcal{V})$. In this context we may know certain ***agreement properties*** correlating I and J, e.g., that the union of their visibility relations is itself a well-formed visibility relation and is thus acyclic. Establishing them requires global reasoning about whole executions C and $\mathsf{abs}(C, \mathcal{V})$. Fortunately, we find that this can be done knowing only the abstract methods, not the implementation $\mathcal{D}_\tau$. Furthermore, these methods state basic facts about the messaging behavior of implementations and are thus common to broad classes of them, such as state-based or op-based. This allows us to establish agreement properties using global reasoning once for a given class of implementations; at this stage we can also benefit from the transport layer specification $\mathcal{T}$ and check that the abstract methods construct visibility according to the given witness $\mathcal{V}$. Then a particular implementation within the class can be verified by discharging local obligations, such as those in Fig. 4(a, b), while assuming agreement properties. This yields easy and intuitive proofs.

To summarize, we deal with the challenge posed by a distributed data type implementation by decomposing reasoning about it into global reasoning done once for a broad class of implementations and local implementation-specific reasoning. We start by presenting the general form of obligations to be discharged for a single implementation within a certain class (§5.1) and the particular form they take for the class of state-based implementations (§5.2), together with some examples (§5.3). We then formulate the obligations to be discharged for a class of implementations (§5.4), which in particular, establish the agreement properties assumed in the per-implementation obligations. In [12, §B], we give the obligations for op-based implementations, together with a proof of the counter in Fig. 2(a). An impatient reader can move on to §6 after finishing §5.3, and come back to §5.4 later.

Figure 4. Diagrams illustrating replication-aware simulations



Since Def. 7 considers only single-object executions, we fix an object x of type τ and consider only concrete and abstract executions over x, whose sets we denote by $\mathsf{CEx}[x]$ and $\mathsf{AEx}[x]$.

5.1 Replication-Aware Simulations

As is typical for simulation-based proofs, we need to use auxiliary state to record information about the computation history. For this reason, actually our simulation relations associate a state or a message with an ***instrumented execution***—a pair $(A, \mathsf{info}) \in \mathsf{IEx}$ of an abstract execution $A \in \mathsf{AEx}[x]$ and a function $\mathsf{info} : A.E \to \mathsf{AInfo}$, tagging events with auxiliary information from a set AInfo. As we show below, AInfo can be chosen once for a class of data type implementations: e.g., $\mathsf{AInfo} = \mathsf{Timestamp}$ for state-based ones (§5.2). We use I and J to range over instrumented executions and shorten, e.g., $I.A.E$ to $I.E$. For a partial function h we write $h(x)\!\downarrow$ for $x \in \mathsf{dom}(h)$, and adopt the convention that $h(x) = y$ implies $h(x)\!\downarrow$.

DEFINITION 8. *A **replication-aware simulation** between $\mathcal{D}_\tau$ and $\mathcal{F}_\tau$ with respect to* info *and **abstract methods** $\mathsf{do}^\sharp$, $\mathsf{send}^\sharp$ and $\mathsf{receive}^\sharp$ is a collection of relations $\{\mathcal{R}_r, \mathcal{M} \mid r \in \mathsf{ReplicaID}\}$ satisfying the conditions in Fig. 5.*

Here info and abstract methods are meant to be fixed for a given class of implementations, such as state or op-based. To prove a particular implementation within this class, one needs to find simulation relations satisfying the conditions in Fig. 5. For example, as we show in §5.3, the following relation lets us prove the correctness of the counter in Fig. 2(b) with respect to info and abstract methods appropriate for state-based implementations:

$$\begin{aligned}&\langle s, v\rangle\ [\mathcal{R}_r]\ I \iff (r = s) \wedge (v\ [\mathcal{M}]\ I);\\ &v\ [\mathcal{M}]\ ((E, \mathsf{repl}, \mathsf{obj}, \mathsf{oper}, \mathsf{rval}, \mathsf{ro}, \mathsf{vis}, \mathsf{ar}), \mathsf{info}) \iff \\ &\qquad \forall s.\, v(s) = \big|\{e \in E \mid \mathsf{oper}(e) = \mathtt{inc} \wedge \mathsf{repl}(e) = s\}\big|.\end{aligned} \tag{6}$$

INIT in Fig. 5 associates the initial state at a replica r with the execution having an empty set of events. DO, SEND and RECEIVE formalize the obligations illustrated in Fig. 4(a, b). Note that $\mathsf{do}^\sharp$ is parameterized by an event e (required to be fresh in instantiations) and the information about the operation performed.

The abstract methods are partial and the obligations in Fig. 5 *assume* that their applications are defined. When instantiating $\mathsf{receive}^\sharp$ for a given class of implementations, we let it be defined only when its arguments satisfy the agreement property for this class, which we establish separately (§5.4). While doing this, we can also establish some ***execution invariants***, holding of single ex-

Figure 5. Definition of a replication-aware simulation $\{\mathcal{R}_r, \mathcal{M}\}$ between $\mathcal{D}_\tau$ and $\mathcal{F}_\tau$. All free variables in each condition are implicitly universally quantified and have the following types: $\sigma, \sigma' \in \Sigma$, $m \in M$, $I, I', J \in \mathsf{IEx}$, $e \in \mathsf{Event}$, $r \in \mathsf{ReplicaID}$, $o \in \mathsf{Op}_\tau$, $a \in \mathsf{Val}_\tau$, $t \in \mathsf{Timestamp}$.

$$\mathcal{R}_r \subseteq \Sigma \times \mathsf{IEx},\ r \in \mathsf{ReplicaID}; \qquad \mathcal{M} \subseteq M \times \mathsf{IEx}$$

$$\mathsf{do}^\sharp : \mathsf{IEx} \times \mathsf{Event} \times \mathsf{ReplicaID} \times \mathsf{Op}_\tau \times \mathsf{Val}_\tau \times \mathsf{Timestamp} \rightharpoonup \mathsf{IEx}$$

$$\mathsf{send}^\sharp : \mathsf{IEx} \rightharpoonup \mathsf{IEx} \times \mathsf{IEx}; \qquad \mathsf{receive}^\sharp : \mathsf{IEx} \times \mathsf{IEx} \rightharpoonup \mathsf{IEx}$$

INIT: $\mathcal{D}_\tau.\vec{\sigma}_0(r)\ [\mathcal{R}_r]\ I_\emptyset$, where $I_\emptyset.E = \emptyset$

DO: $(\mathsf{do}^\sharp(I, e, r, o, a, t) = I' \ \wedge\ \mathcal{D}_\tau.\mathsf{do}(o, \sigma, t) = (\sigma', a) \ \wedge\ \sigma\ [\mathcal{R}_r]\ I)$
$\implies (\sigma'\ [\mathcal{R}_r]\ I' \ \wedge\ a = \mathcal{F}_\tau(\mathsf{ctxt}(I'.A, e)))$

SEND: $(\mathsf{send}^\sharp(I) = (I', J) \ \wedge\ \mathcal{D}_\tau.\mathsf{send}(\sigma) = (\sigma', m) \ \wedge\ \sigma\ [\mathcal{R}_r]\ I)$
$\implies (\sigma'\ [\mathcal{R}_r]\ I' \ \wedge\ m\ [\mathcal{M}]\ J)$

RECEIVE: $(\mathsf{receive}^\sharp(I, J)\downarrow \ \wedge\ \sigma\ [\mathcal{R}_r]\ I \ \wedge\ m\ [\mathcal{M}]\ J)$
$\implies (\mathcal{D}_\tau.\mathsf{receive}(\sigma, m)\ [\mathcal{R}_r]\ \mathsf{receive}^\sharp(I, J))$

ecutions supplied as parameters to $\mathsf{do}^\sharp$ and $\mathsf{send}^\sharp$. We similarly assume them in Fig. 5 via the definedness of these abstract methods.

5.2 Instantiation for State-Based Implementations

Fig. 6 defines the domain AInfo and abstract methods appropriate for state-based implementations. In §5.4 we show that the existence of a simulation of Def. 8 with respect to these parameters implies $\mathcal{D}_\tau\ \mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathsf{T\text{-}Any}]\ \mathcal{F}_\tau$ (Theorems 9 and 10). The $\mathsf{do}^\sharp$ method adds a fresh event e with the given attributes to I; its timestamp t is recorded in info. In the resulting execution I', the event e is the last one by its replica, observes all events in I and occupies the place in arbitration consistent with t. The $\mathsf{send}^\sharp$ method just returns I, which formalizes the intuition that, in state-based implementations, send returns a message capturing all the information about the object available at the replica. The $\mathsf{receive}^\sharp$ method takes the component-wise union $I \sqcup J$ of executions I related to the current state and J related to the message, applied recursively to the components of $I.A$ and $J.A$. We also assume that $I \sqcup J$ recomputes the arbitration relation in the resulting execution from the timestamps. This is the reason for recording them in info: we would not be able to construct $\mathsf{receive}^\sharp(I, J).\mathsf{ar}$ solely from $I.\mathsf{ar}$ and $J.\mathsf{ar}$.

The agreement property $\mathsf{agree}(I, J)$ guarantees that $I \sqcup J$ is well-formed (e.g., its visibility relation is acyclic) and that, for each replica, I describes a computation extending J or vice versa. The latter follows from the observation we made when explaining the state-based counter in §2: a message or a state in a state-based implementation reflects a prefix of the sequence of events performed at a given replica. The first conjunct of the execution invariant inv requires arbitration to be consistent with event timestamps; the second conjunct follows from the definition of $\mathcal{V}^{\mathsf{state}}$ (§4). When discharging the obligations in Fig. 5 with respect to the parameters in Fig. 6 for a particular implementation, we can rely on the agreement property and the execution invariant.

5.3 Examples

We illustrate the use of the instantiation from §5.2 on the state-based counter, LWW-register and OR-set. In [12, §A] we also give proofs of two multi-value register implementations.

Counter: $\mathcal{D}_{\mathtt{ctr}}$ in Fig. 2(b) and $\mathcal{F}_{\mathtt{ctr}}$ in (3). Discharging the obligations in Fig. 5 for the simulation (6) is easy. The key case is RECEIVE, where the first conjunct of agree in Fig. 6 ensures that $\min\{v(s), v'(s)\}$ increments by a replica s are taken into account both in $v(s)$ and in $v'(s)$:

$$(\langle r, v\rangle\ [\mathcal{R}_r]\ I) \wedge (v'\ [\mathcal{M}]\ J) \implies (\forall s.\ \min\{v(s), v'(s)\} = |\{e \in I.E \cap J.E \mid I.\mathsf{oper}(e) = \mathtt{inc} \wedge I.\mathsf{repl}(e) = s\}|).$$

Figure 6. Instantiation for state-based data type implementations. In $\mathsf{do}^\sharp$ we omit the straightforward definition of ar' in terms of $I.\mathsf{info}$ and t.

$$\mathsf{AInfo} = \mathsf{Timestamp}$$

$$\mathsf{agree}(I, J) \iff \forall r.\ (\{e \in I.E \mid I.\mathsf{repl}(e) = r\}, I.\mathsf{ro}) \text{ is a prefix of } (\{e \in J.E \mid J.\mathsf{repl}(e) = r\}, J.\mathsf{ro}) \text{ or vice versa}) \wedge (I \sqcup J \in \mathsf{IEx})$$

$$\mathsf{inv}(I) \iff (\forall e, f \in I.E.\ (e, f) \in I.\mathsf{ar} \iff I.\mathsf{info}(e) < I.\mathsf{info}(f)) \wedge ((I.\mathsf{vis} \cup I.\mathsf{ro})^+ \subseteq I.\mathsf{vis})$$

$$\mathsf{do}^\sharp(I, e, r, o, a, t) = I', \quad \text{if } \mathsf{inv}(I) \wedge e \notin I.E \wedge I' \in \mathsf{IEx}$$
$$\text{where } I' = ((I.E \cup \{e\},\ I.\mathsf{repl}[e \mapsto r],\ I.\mathsf{obj}[e \mapsto x],\ I.\mathsf{oper}[e \mapsto o],\ I.\mathsf{rval}[e \mapsto a],\ I.\mathsf{ro} \cup \{(f, e) \mid f \in I.E \wedge I.\mathsf{repl}(f) = r\},\ I.\mathsf{vis} \cup \{(f, e) \mid f \in I.E\},\ \mathsf{ar}'),\ I.\mathsf{info}[e \mapsto t])$$

$$\mathsf{send}^\sharp(I) = (I, I), \quad \text{if } \mathsf{inv}(I)$$

$$\mathsf{receive}^\sharp(I, J) = I \sqcup J, \quad \text{if } \mathsf{inv}(I) \wedge \mathsf{inv}(J) \wedge \mathsf{agree}(I, J)$$

This allows establishing $\mathsf{receive}(\langle r, v\rangle, v')\ \ [\mathcal{R}_r]\ \ (I \sqcup J)$, thus formalizing the informal justification of correctness we gave in §2.

LWW-register: $\mathcal{D}_{\mathtt{intreg}}$ in Fig. 2(c) and $\mathcal{F}_{\mathtt{intreg}}$ in (4). We associate a state or a message $\langle a, t\rangle$ with any execution that contains a $\mathtt{wr}(a)$ event with the timestamp t maximal among all other $\mathtt{wr}$ events (as per info). By inv in Fig. 6, this event is maximal in arbitration, which implies that $\mathtt{rd}$ returns the correct value; the other obligations are also discharged easily. Formally, $\forall r.\ \mathcal{R}_r = \mathcal{M}$ and

$$\langle a, t\rangle\ [\mathcal{M}]\ ((E, \mathsf{repl}, \mathsf{obj}, \mathsf{oper}, \mathsf{rval}, \mathsf{ro}, \mathsf{vis}, \mathsf{ar}), \mathsf{info}) \iff (t = 0 \wedge a = 0 \wedge (\neg\exists e \in E.\ \mathsf{oper}(e) = \mathtt{wr}(\_))) \vee (t > 0 \wedge (\exists e \in E.\ \mathsf{oper}(e) = \mathtt{wr}(a) \wedge \mathsf{info}(e) = t) \wedge (\forall f \in E.\ \mathsf{oper}(f) = \mathtt{wr}(\_) \implies \mathsf{info}(f) \leq t)).$$

Optimized OR-set: $\mathcal{D}_{\mathtt{orset}}$ in Fig. 7 and $\mathcal{F}_{\mathtt{orset}}$ in (5). A problem with implementing a replicated set is that we often cannot discard the information about an element from a replica state after it has been removed: if another replica unaware of the removal sends us a snapshot of its state containing this element, the semantics of the set may require our receive to keep the element out of the set. As we prove in §6, for the OR-set keeping track of information about removed elements cannot be fully avoided, which makes its space-efficient implementation very challenging. Here we consider a recently-proposed OR-set implementation [6] that, as we show in §6, has an optimal space complexity. It improves on the original implementation [32], whose complexity was suboptimal (we have proved the correctness of the latter as well; see [12, §A]).

An additional challenge posed by the OR-set is that, according to $\mathcal{F}_{\mathtt{orset}}$, a `remove` operation may behave differently with respect to different events adding the same element to the set, depending on whether it sees them or not. This causes the implementation to treat internally each `add` operation as generating a unique *instance* of the *element* being added, further increasing the space required. To combat this, the implementation concisely summarizes information about instances. An instance is represented by a unique *instance identifier* that is generated when a replica performs an `add` and consists of the replica identifier and the number of `add`s (of any elements) performed at the replica until then. In a state $\langle r, V, w\rangle$, the vector w determines the identifiers of all instances that the current replica r has ever observed: for any replica s, the replica r has seen $w(s)$ successive identifiers $(s, 1), (s, 2), \ldots, (s, w(s))$ generated at s. To generate a new identifier in $\mathsf{do}(\mathtt{add}(a'))$, the replica r increments $w(r)$. The connection between the vector w in a state or a message and `add` events $e_{s,k}$ in corresponding executions is formalized in lines 1-3 of the simulation relation, also shown in Fig. 7. In receive we take the pointwise maximum of the two vectors w and w'. Like for the counter, the first conjunct of agree implies that this preserves the clauses in lines 1-3.

Figure 7. Optimized OR-set implementation [6] and its simulation

$\Sigma = \mathsf{ReplicaID} \times ((\mathbb{Z} \times \mathsf{ReplicaID}) \to \mathbb{N}_0) \times (\mathsf{ReplicaID} \to \mathbb{N}_0)$
$\vec{\sigma}_0 = \lambda r.\, \langle r, (\lambda a, s.\, 0), (\lambda s.\, 0)\rangle$
$M = ((\mathbb{Z} \times \mathsf{ReplicaID}) \to \mathbb{N}_0) \times (\mathsf{ReplicaID} \to \mathbb{N}_0)$
$\mathsf{do}(\mathtt{add}(a'), \langle r, V, w\rangle, t) = (\langle r, (\lambda a, s.\, \text{if } a = a' \wedge s = r \text{ then } w(r) + 1 \text{ else } V(a, s)), w[r \mapsto w(r) + 1]\rangle, \bot)$
$\mathsf{do}(\mathtt{remove}(a'), \langle r, V, w\rangle, t) = (\langle r, (\lambda a, s.\, \text{if } a = a' \text{ then } 0 \text{ else } V(a, s)), w\rangle, \bot)$
$\mathsf{do}(\mathtt{rd}, \langle r, V, w\rangle, t) = (\langle r, V, w\rangle, \{a \mid \exists s.\, V(a, s) > 0\})$
$\mathsf{send}(\langle r, V, w\rangle) = (\langle r, V, w\rangle, \langle V, w\rangle)$
$\mathsf{receive}(\langle r, V, w\rangle, \langle V', w'\rangle) = \langle r, (\lambda a, s.\, \text{if } (V(a, s) = 0 \wedge w(s) \geq V'(a, s)) \vee (V'(a, s) = 0 \wedge w'(s) \geq V(a, s)) \text{ then } 0 \text{ else } \max\{V(a, s), V'(a, s)\}), (\lambda s.\, \max\{w(s), w'(s)\})\rangle$

$\langle s, V, w\rangle\ [\mathcal{R}_r]\ I \iff (r = s) \wedge (\langle V, w\rangle\ [\mathcal{M}]\ I)$

$\langle V, w\rangle\ [\mathcal{M}]\ ((E, \mathsf{repl}, \mathsf{obj}, \mathsf{oper}, \mathsf{rval}, \mathsf{ro}, \mathsf{vis}, \mathsf{ar}), \mathsf{info}) \iff$
1: $\exists$ distinct $e_{s,k}.\, (\{e_{s,k} \mid s \in \mathsf{ReplicaID} \wedge 1 \leq k \leq w(s)\} =$
2: $\quad \{e \in E \mid \mathsf{oper}(e) = \mathtt{add}(\_)\}) \wedge$
3: $(\forall s, k, j.\, (\mathsf{repl}(e_{s,k}) = s) \wedge (e_{s,j} \xrightarrow{\mathsf{ro}} e_{s,k} \iff j < k)) \wedge$
4: $(\forall a, s.\, (V(a, s) \leq w(s)) \wedge (V(a, s) \neq 0 \implies$
5: $\quad (\mathsf{oper}(e_{s,V(a,s)}) = \mathtt{add}(a)) \wedge$
6: $\quad (\neg\exists k.\, V(a, s) < k \leq w(s) \wedge \mathsf{oper}(e_{s,k}) = \mathtt{add}(a)) \wedge$
7: $\quad (\neg\exists f \in E.\, \mathsf{oper}(f) = \mathtt{remove}(a) \wedge e_{s,V(a,s)} \xrightarrow{\mathsf{vis}} f))) \wedge$
8: $(\forall a, s, k.\, e_{s,k} \in E \wedge \mathsf{oper}(e_{s,k}) = \mathtt{add}(a) \implies$
9: $\quad (k \leq V(a, s) \vee \exists f \in E.\, \mathsf{oper}(f) = \mathtt{remove}(a) \wedge e_{s,k} \xrightarrow{\mathsf{vis}} f))$

The component w in $\langle r, V, w\rangle$ records identifiers of both of those instances that have been removed and those that are still in the set (are *active*). The component V serves to distinguish the latter. As it happens, we do not need to store all active instances of an element a: for every replica s, it is enough to keep the last active instance identifier generated by an $\mathtt{add}(a)$ at this replica. If $V(a, s) \neq 0$, this identifier is $(s, V(a, s))$; if $V(a, s) = 0$, all instances of a generated at s that the current replica knows about are inactive. The meaning of V is formalized in the simulation: each instance identifier given by V is covered by w (line 4) and, if $V(a, s) \neq 0$, then the event $e_{s,V(a,s)}$ performs $\mathtt{add}(a)$ (line 5), is the last $\mathtt{add}(a)$ by replica s (line 6) and has not been observed by a $\mathtt{remove}(a)$ (line 7). Finally, the $\mathtt{add}(a)$ events that are not seen by a $\mathtt{remove}(a)$ in the execution are either the events $e_{s,V(a,s)}$ or those superseded by them (lines 8-9). This ensures that returning all elements with an active instance in $\mathtt{rd}$ matches $\mathcal{F}_{\mathtt{orset}}$.

When a replica r performs $\mathsf{do}(\mathtt{add}(a'))$, we update $V(a', r)$ to correspond to the new instance identifier. Conversely, in $\mathsf{do}(\mathtt{remove}(a'))$, we clear all entries in $V(a')$, thereby deactivating all instances of a'. However, after this their identifiers are still recorded in w, and so we know that they have been previously removed. This allows us to address the problem with implementing receive we mentioned above: if we receive a message with an active instance $(s, V'(a, s))$ of an element a that is not in the set at our replica ($V(a, s) = 0$), but previously existed ($w(s) \geq V'(a, s)$), this means that the instance has been removed and should not be active in the resulting state (the entry for (a, s) should be 0). We also do the same check with the state and the message swapped.

As the above explanation shows, our simulation relations are useful not only for proving correctness of data type implementations, but also for explaining their designs. Discharging obligations in Fig. 5 requires some work for the OR-set; due to space constraints, we defer this to [12, §A].

Figure 8. Function step that mirrors the effect of an event $e \in C'.E$ from $C' \in \mathsf{CEx}[x]$ in $D \in \mathsf{DEx}$, defined when so is the abstract method used

$\mathsf{step}(C', e, D) = D[r \mapsto \mathsf{do}^\sharp(D(r), e, r, C'.\mathsf{oper}(e), C'.\mathsf{rval}(e), C'.\mathsf{time}(e))]$,
  if $C'.\mathsf{act}(e) = \mathsf{do} \wedge C'.\mathsf{repl}(e) = r$
$\mathsf{step}(C', e, D) = D[r \mapsto I, C'.\mathsf{msg}(e) \mapsto J]$,
  if $C'.\mathsf{act}(e) = \mathsf{send} \wedge C'.\mathsf{repl}(e) = r \wedge C'.\mathsf{msg}(e) \notin \mathsf{dom}(D) \wedge \mathsf{send}^\sharp(D(r)) = (I, J)$
$\mathsf{step}(C', e, D) = D[r \mapsto \mathsf{receive}^\sharp(D(r), D(C'.\mathsf{msg}(e)))]$,
  if $C'.\mathsf{act}(e) = \mathsf{receive} \wedge C'.\mathsf{repl}(e) = r$

5.4 Soundness and Establishing Agreement Properties

We present conditions on AInfo and abstract methods ensuring the soundness of replication-aware simulations over them and, in particular, establishing the agreement property and execution invariants assumed via the definedness of abstract operations in Fig. 5.

THEOREM 9 (Soundness). *Assume* AInfo, $\mathsf{do}^\sharp$, $\mathsf{send}^\sharp$, $\mathsf{receive}^\sharp$, $\mathcal{V}$ *and* $\mathcal{T}$ *that satisfy the conditions in Fig. 9 for some* $\mathcal{G}$. *If there exists a replication-aware simulation between* $\mathcal{D}_\tau$ *and* $\mathcal{F}_\tau$ *with respect to these parameters, then* $\mathcal{D}_\tau\ \mathsf{sat}[\mathcal{V}, \mathcal{T}]\ \mathcal{F}_\tau$.

Conditions in Fig. 9 require global reasoning, but can be discharged once for a class of data types. For example, they hold of the instantiation for state-based implementations from §5.2, as well as one for op-based implementations presented in [12, §B].

THEOREM 10. *There exists* $\mathcal{G}$ *such that, for all* $\mathcal{D}_\tau$, *the parameters in Fig. 6 satisfy the conditions in Fig. 9 with respect to this* $\mathcal{G}$, $\mathcal{V} = \mathcal{V}^{\mathsf{state}}$, $\mathcal{T} = \mathsf{T\text{-}Any}$.

The proofs of Theorems 9 and 10 are given in [12, §B]. To explain the conditions in Fig. 9, here we consider the proof strategy for Theorem 9. To establish $\mathcal{D}_\tau\ \mathsf{sat}[\mathcal{V}, \mathcal{T}]\ \mathcal{F}_\tau$, for any $C \in [\![\mathcal{D}_\tau]\!] \cap \mathcal{T}$ we need to show $\mathsf{abs}(C, \mathcal{V}) \models [\tau \mapsto \mathcal{F}_\tau]$. We prove this by induction on the length of C. To use the localized conditions in Fig. 5, we require a relation $\mathcal{G}$ associating C with a ***decomposed execution***—a partial function $D : (\mathsf{ReplicaID} \cup \mathsf{MessageID}) \rightharpoonup \mathsf{IEx}$ that gives fragments of $\mathsf{abs}(C, \mathcal{V})$ corresponding to replica states and messages in the final configuration of C. We write DEx for the set of all decomposed executions, so that $\mathcal{G} \subseteq \mathsf{CEx}[x] \times \mathsf{DEx}$. The existence of a decomposed execution D such that $C\ [\mathcal{G}]\ D$ forms the core of our induction hypothesis. G-CTXT in Fig. 9 checks that the abstract methods construct visibility according to $\mathcal{V}$: it requires the context of any event e by a replica r to be the same in $D(r)$ and $\mathsf{abs}(C, \mathcal{V})$. Together with DO in Fig. 5, this ensures $\mathsf{abs}(C, \mathcal{V}) \models [\tau \mapsto \mathcal{F}_\tau]$.

We write $C' \sim (C \xrightarrow{e} (R, T))$ when C' is an extension of C in the following sense: $C'.E = C.E \uplus \{e\}$, the other components of C are those of C' restricted to $C.E$, e is last in $C'.\mathsf{eo}$ and $C'.\mathsf{post}(e) = (R, T)$. For the induction step, assume $C\ [\mathcal{G}]\ D$ and $C' \sim (C \xrightarrow{e} (R, T))$; see Fig. 4(c). Then the decomposed execution D' corresponding to C' is given by $\mathsf{step}(C', e, D)$, where the function step in Fig. 8 mirrors the effect of the event e from C' in D using the abstract methods. G-STEP ensures that it preserves the relation $\mathcal{G}$. Crucially, G-STEP also requires us to establish the definedness of step and thus the corresponding abstract method. This justifies the agreement property and execution invariants encoded by the definedness and allows us to use the conditions in Fig. 5 to complete the induction. We also require G-INIT, which establishes the base case, and G-VIS, which formulates a technical restriction on $\mathcal{V}$. Finally, the conditions in Fig. 9 allow us to use the transport specification $\mathcal{T}$ by considering only executions C satisfying it.

6. Space Bounds and Implementation Optimality

Object states in replicated data type implementations include not only the current client-observable content, but also metadata

Figure 9. Proof obligations for abstract methods. Free variables are implicitly universally quantified and have the following types: $C, C' \in \mathsf{CEx}[x] \cap \mathcal{T}$, $D \in \mathsf{DEx}$, $r \in \mathsf{ReplicaID}$, $e \in \mathsf{Event}$, $(R, T) \in \mathsf{Config}$.

G-CTXT: $(C\ [\mathcal{G}]\ D \wedge e \in \mathsf{abs}(C, \mathcal{V}).E \wedge \mathsf{abs}(C, \mathcal{V}).\mathsf{repl}(e) = r)$
$\implies \mathsf{ctxt}(D(r).A, e) = \mathsf{ctxt}(\mathsf{abs}(C, \mathcal{V}), e)$

G-STEP: $(C' \sim (C \xrightarrow{e} (R, T)) \wedge (C\ [\mathcal{G}]\ D))$
$\implies (\mathsf{step}(C', e, D)\downarrow \wedge C'\ [\mathcal{G}]\ \mathsf{step}(C', e, D))$

G-INIT: $(C.E = \{e\} \wedge C.\mathsf{pre}(e) = (\_, [\,]))$
$\implies (\mathsf{step}(C, e, D_\emptyset)\downarrow \wedge C\ [\mathcal{G}]\ \mathsf{step}(C, e, D_\emptyset))$,
where $D_\emptyset$ is such that $\mathsf{dom}(D_\emptyset) = \mathsf{ReplicaID} \wedge \forall r \in \mathsf{ReplicaID}.\ D_\emptyset(r).E = \emptyset$

G-VIS: $(e \in \mathsf{abs}(C, \mathcal{V}).E \wedge (C \text{ is a prefix of } C' \text{ under } C'.\mathsf{eo}))$
$\implies \mathsf{ctxt}(\mathsf{abs}(C, \mathcal{V}), e) = \mathsf{ctxt}(\mathsf{abs}(C', \mathcal{V}), e)$

needed for conflict resolution or masking network failures. Space taken by this metadata is a major factor determining their efficiency and feasibility. As illustrated by the OR-set in §5.3, this is especially so for state-based implementations, i.e., those that satisfy their data type specifications with respect to the visibility witness $\mathcal{V}^{\mathsf{state}}$ and the transport layer specification T-Any. We now present a general technique for proving lower bounds on this space overhead, which we use to prove optimality of four state-based implementations (we leave other implementation classes for future work; see §9). As in §5, we only consider executions over a fixed object x of type τ.

6.1 Metadata Overhead

To measure space, we need to consider how data are represented. An ***encoding*** of a set S is an injective function $\mathsf{enc} : S \to \Lambda^+$, where Λ is some suitably chosen fixed finite set of characters (left unspecified). Sometimes, we clarify the domain being encoded using a subscript: e.g., $\mathsf{enc}_{\mathbb{N}_0}(1)$. For $s \in S$, we let $\mathsf{len}_S(s)$ be the length of $\mathsf{enc}_S(s)$. The length can vary: e.g., for an integer k, $\mathsf{len}_{\mathbb{N}_0}(k) \in \Theta(\lg k)$. We use standard encodings (listed in [12, §C]) for return values $\mathsf{enc}_{\mathsf{Val}_\tau}$ of the data types τ we consider and assume an arbitrary but fixed encoding of object states $\mathsf{enc}_{\mathcal{D}_\tau.\Sigma}$.

To distinguish metadata from the client-observable content of the object, we assume that each data type has a special `rd` operation that returns the latter, as is the case in the examples considered so far. For a concrete execution $C \in [\![\mathcal{D}_\tau]\!]$ over the object x and a read event $e \in (C.E)|_{\mathtt{rd}}$, we define $\mathsf{state}(e)$ to be the state of the object accessed at e: $\mathsf{state}(e) = R(x, C.\mathsf{repl}(e))$ for $(R, \_) = C.\mathsf{pre}(e)$.

We now define the metadata overhead as a ratio, by dividing the size of the object state by the size of the observable state. We then quantify the worst-case overhead by taking the maximum of this ratio over all read operations in all executions with given numbers of replicas n and update operations m. To define the latter, we assume that each data type τ specifies a set $\mathsf{Upd}_\tau \subset \mathsf{Op}_\tau$ of update operations; for all examples in this paper $\mathsf{Upd}_\tau = \mathsf{Op}_\tau \setminus \{\mathtt{rd}\}$.

DEFINITION 11. *The **maximum metadata overhead** of an execution $C \in [\![\mathcal{D}_\tau]\!]$ of an implementation $\mathcal{D}_\tau$ is*

$$\mathsf{mmo}(\mathcal{D}_\tau, C) = \max\left\{\frac{\mathsf{len}_{\mathcal{D}_\tau.\Sigma}(\mathsf{state}(e))}{\mathsf{len}_{\mathsf{Val}_\tau}(C.\mathsf{rval}(e))} \mid e \in (C.E)|_{\mathtt{rd}}\right\}.$$

*The **worst-case metadata overhead** of an implementation $\mathcal{D}_\tau$ over all executions with n replicas and m updates ($2 \le n \le m$) is*

$$\begin{aligned}\mathsf{wcmo}(\mathcal{D}_\tau, n, m) = \max\{&\mathsf{mmo}(\mathcal{D}_\tau, C) \mid C \in [\![\mathcal{D}_\tau]\!] \wedge \\ &n = |\{C.\mathsf{repl}(e) \mid e \in C.E\}| \wedge \\ &m = |\{e \in C.E \mid C.\mathsf{oper}(e) \in \mathsf{Upd}_\tau\}|\}.\end{aligned}$$

We consider only executions with $m \ge n$, since we are interested in the asymptotic overhead of executions where all replicas are mutated (i.e., perform at least one update operation).

Figure 10. Summary of bounds on metadata overhead for stated-based implementations, as functions of the number of replicas n and updates m

Type	Existing implementation			Any implementation
	algorithm	ref.	overhead	overhead
`ctr`	Fig. 2(b)	[32]	$\widehat{\Theta}(n)$	$\widehat{\Omega}(n)$
`orset`	Fig. 7	[6]	$\widehat{\Theta}(n \lg m)$	$\widehat{\Omega}(n \lg m)$
	Fig. 15, [12, §A]	[32]	$\widehat{\Theta}(m \lg m)$	
`intreg`	Fig. 2(c)	[32]	$\widehat{\Theta}(\lg m)^\dagger$	$\widehat{\Omega}(\lg m)$
`mvr`	Fig. 17, [12, §A]	new‡	$\widehat{\Theta}(n \lg m)$	$\widehat{\Omega}(n \lg m)$
	Fig. 16, [12, §A]	[32]	$\widehat{\Theta}(n^2 \lg m)$	

† Assuming timestamp encoding is $O(\lg m)$, satisfied by Lamport clocks.
‡ An optimization of [32] discovered during the optimality proof.

DEFINITION 12. *Assume $\mathcal{D}_\tau$ and a positive function $f(n, m)$.*

- *f is an **asymptotic upper bound** ($\mathcal{D}_\tau \in \widehat{O}(f(n, m))$) if $\sup_{n,m\to\infty}(\mathsf{wcmo}(\mathcal{D}_\tau, n, m)/f(n, m)) < \infty$, i.e.,*
$$\exists K > 0.\, \forall m \ge n \ge 2.\, \mathsf{wcmo}(\mathcal{D}_\tau, n, m) < Kf(n, m);$$
- *f is an **asymptotic lower bound** ($\mathcal{D}_\tau \in \widehat{\Omega}(f(n, m))$) if $\lim_{n,m\to\infty}(\mathsf{wcmo}(\mathcal{D}_\tau, n, m)/f(n, m)) \ne 0$, i.e.,*
$$\exists K > 0.\, \forall m_0 \ge n_0 \ge 2.\, \exists n \ge n_0, m \ge n_0.\ \mathsf{wcmo}(\mathcal{D}_\tau, n, m) > Kf(n, m);$$
- *f is an **asymptotically tight bound** ($\mathcal{D}_\tau \in \widehat{\Theta}(f(n, m))$) if it is both an upper and a lower asymptotic bound.*

Fig. 10 summarizes our results; as described in §5, we have proved all the implementations correct. Matching lower and upper bounds indicate worst-case optimality of an implementation (note that this is different from optimality in all cases). The derivation of upper bounds relies on standard techniques and is deferred to [12, §C]. We now proceed to the main challenge: how to derive lower bounds that apply to *any* implementation of τ. We present proofs for `ctr` and `orset`; `intreg` and `mvr` are covered in [12, §C].

6.2 Experiment Families

The goal is to show that for any correct implementation $\mathcal{D}_\tau$ (i.e., such that $\mathcal{D}_\tau\ \mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathsf{T\text{-}Any}]\ \mathcal{F}_\tau$), the object state must store some minimum amount of information. We achieve this by constructing an *experiment family*, which is a collection of executions $\mathbb{C}_\alpha$, where $\alpha \in Q$ for some index set Q. Each experiment contains a distinguished read event $\mathbf{e}_\alpha$. The experiments are designed in such a way that the object states $\mathsf{state}(\mathbf{e}_\alpha)$ must be distinct, which then implies a lower bound $\lg_{|\Lambda|} |Q|$ on the size of their encoding. To prove that they are distinct, we construct black-box tests that execute the methods of $\mathcal{D}_\tau$ on the states and show that the tests must produce different results for each $\mathsf{state}(\mathbf{e}_\alpha)$ provided $\mathcal{D}_\tau$ is correct. Formally, the tests induce a read-back function rb that satisfies $\mathsf{rb}(\mathsf{state}(\mathbf{e}_\alpha)) = \alpha$. We encapsulate the core argument in the following lemma.

DEFINITION 13. *An **experiment family** for an implementation $\mathcal{D}_\tau$ is a tuple $(Q, n, m, \mathbb{C}, \mathbf{e}, \mathsf{rb})$ where Q is a finite set, $2 \le n \le m$, and for each $\alpha \in Q$, $\mathbb{C}_\alpha \in [\![\mathcal{D}_\tau]\!]$ is an execution with n replicas and m updates, $\mathbf{e}_\alpha \in (\mathbb{C}_\alpha.E)|_{\mathtt{rd}}$ and $\mathsf{rb} : \mathcal{D}_\tau.\Sigma \to Q$ is a function satisfying $\mathsf{rb}(\mathsf{state}(\mathbf{e}_\alpha)) = \alpha$.*

LEMMA 14. *If $(Q, n, m, \mathbb{C}, \mathbf{e}, \mathsf{rb})$ is an experiment family, then*

$$\mathsf{wcmo}(\mathcal{D}_\tau, n, m) \ge \lfloor \lg_{|\Lambda|} |Q| \rfloor \,/\, (\max_{\alpha\in Q} \mathsf{len}(\mathbb{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha))).$$

PROOF. Since $\mathsf{rb}(\mathsf{state}(\mathbf{e}_\alpha)) = \alpha$, the states $\mathsf{state}(\mathbf{e}_\alpha)$ are pairwise distinct and so are their encodings $\mathsf{enc}(\mathsf{state}(\mathbf{e}_\alpha))$. Since there are fewer than $|Q|$ strings of length strictly less than $\lfloor \lg_{|\Lambda|} |Q| \rfloor$, for

some $\alpha \in Q$ we have $\mathsf{len}(\mathsf{enc}(\mathsf{state}(\mathbf{e}_\alpha))) \geq \lfloor \lg_{|\Lambda|} |Q| \rfloor$. Then

$$\mathsf{wcmo}(\mathcal{D}_\tau, n, m) \geq \mathsf{mmo}(\mathcal{D}_\tau, \mathbb{C}_\alpha) \geq \frac{\mathsf{len}(\mathsf{state}(\mathbf{e}_\alpha))}{\mathsf{len}(\mathbb{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha))} \geq \frac{\lfloor \lg_{|\Lambda|} |Q| \rfloor}{\max_{\alpha' \in Q} \mathsf{len}(\mathbb{C}_{\alpha'}.\mathsf{rval}(\mathbf{e}_{\alpha'}))}. \quad \square$$

To apply this lemma to the best effect, we need to find experiment families with $|Q|$ as large as possible and $\mathsf{len}(\mathbb{C}_{\alpha'}.\mathsf{rval}(\mathbf{e}_{\alpha'}))$ as small as possible. Finding such families is challenging, as there is no systematic way to derive them. We relied on intuitions about "which situations force replicas to store a lot of information" when searching for experiment families.

Driver programs. We define experiment families using ***driver programs*** (e.g., see Fig. 11). These are written in imperative pseudocode and use traditional constructs like loops and conditionals. As they execute, they construct concrete executions of the data type library $[\tau \mapsto \mathcal{D}_\tau]$ by means of the following instructions, each of which triggers a uniquely-determined transition from Fig. 3:

$\mathsf{do}_r\ o^{\,t}$	do operation o on x at replica r with timestamp t
$u \leftarrow \mathsf{do}_r\ o^{\,t}$	same, but assign the return value to u
$\mathsf{send}_r(mid)$	send a message for x with identifier mid at r
$\mathsf{receive}_r(mid)$	receive the message mid at replica r

Programs explicitly supply timestamps for do and message identifiers for send and receive. We require that they do this correctly, e.g., respect uniqueness of timestamps. When a driver program terminates, it may produce a return value. For a program P, an implementation $\mathcal{D}_\tau$, and a configuration (R, T), we let $\mathsf{exec}(\mathcal{D}_\tau, (R, T), P)$ be the concrete execution of the data type library $[\tau \mapsto \mathcal{D}_\tau]$ starting in (R, T) that results from running P; we define $\mathsf{result}(\mathcal{D}_\tau, (R, T), P)$ as the return value of P in this run.

6.3 Lower Bound for State-Based Counter (ctr)

THEOREM 15. *If* $\mathcal{D}_{\mathtt{ctr}}\ \mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathsf{T\text{-}Any}]\ \mathcal{F}_{\mathtt{ctr}}$, *then* $\mathcal{D}_{\mathtt{ctr}}$ *is* $\widehat{\Omega}(n)$.

We start by formulating a suitable experiment family.

LEMMA 16. *If* $\mathcal{D}_{\mathtt{ctr}}\ \mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathsf{T\text{-}Any}]\ \mathcal{F}_{\mathtt{ctr}}$, $n \geq 2$ *and* $m \geq n$ *is a multiple of* $(n-1)$, *then tuple* $(Q, n, m, \mathbb{C}, \mathbf{e}, \mathsf{rb})$ *as defined in the left column of Fig. 11 is an experiment family.*

The idea of the experiments is to force replica 1 to remember one number for each of the other replicas in the system, which then introduces an overhead proportional to n; cf. the implementation in Fig. 2(b). We show one experiment in Fig. 12. All experiments start with a common initialization phase, defined by $init$, where each of the replicas $2..n$ performs $m/(n-1)$ increments and sends a message after each increment. All messages remain undelivered until the second phase, defined by $exp(\alpha)$. There replica 1 receives exactly one message from each replica $r = 2..n$, selected using $\alpha(r)$. An experiment concludes with the read $\mathbf{e}_\alpha$ on the first replica.

The read-back works by performing separate tests for each of the replicas $r = 2..n$, defined by $test(r)$. For example, to determine which message was sent by replica 2 during the experiment in Fig. 12, the program $test(2)$: reads the counter value at replica 1, getting 12; delivers the final message by replica 2 to it; and reads the counter value at replica 1 again, getting 14. By observing the difference, the program can determine the message sent during the experiment: $\alpha(2) = 5 - (14 - 12) = 3$.

PROOF OF LEMMA 16. The only nontrivial obligation is to prove $\mathsf{rb}(\mathsf{state}(\mathbf{e}_\alpha)) = \alpha$. Let $(R_\alpha, T_\alpha) = \mathsf{final}(\mathbb{C}_\alpha)$. Then

$$\begin{aligned}
\alpha(r) &\overset{\text{(i)}}{=} \mathsf{result}(\mathcal{D}_{\mathtt{ctr}}, (R_0, T_0), (init; exp(\alpha); test(r))) \\
&= \mathsf{result}(\mathcal{D}_{\mathtt{ctr}}, (R_\alpha, T_\alpha), test(r)) \\
&\overset{\text{(ii)}}{=} \mathsf{result}(\mathcal{D}_{\mathtt{ctr}}, (R_{init}[(x,1) \mapsto R_\alpha(x,1)], T_{init}), test(r)) \\
&= \mathsf{rb}(R_\alpha(x,1))(r) \;=\; \mathsf{rb}(\mathsf{state}(\mathbf{e}_\alpha))(r),
\end{aligned}$$

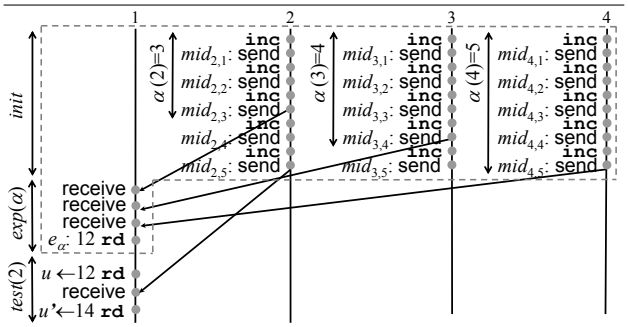
Figure 11. Experiment families $(Q, n, m, \mathbb{C}, \mathbf{e}, \mathsf{rb})$ used in the proofs of Theorem 15 (ctr) and Theorem 17 (orset)

ctr	orset
Conditions on n, m (number of replicas/updates)	
$m \geq n \geq 2$	$m \geq n \geq 2$
$m \mod (n-1) = 0$	$(m-1) \mod (n-1) = 0$
Index set Q	
$Q = ([2..n] \to [1..\frac{m}{n-1}])$	$Q = ([2..n] \to [1..\frac{m-1}{n-1}])$
Family size $\lvert Q \rvert$	
$\lvert Q \rvert = (\frac{m}{n-1})^{n-1}$	$\lvert Q \rvert = (\frac{m-1}{n-1})^{n-1}$
Driver programs	

procedure $init$
  for all $r \in [2..n]$
  for all $i \in [1..\frac{m}{n-1}]$
  do_r `inc` $^{rm+i}$
  $\mathsf{send}_r(mid_{r,i})$
procedure $exp(\alpha)$
  for all $r \in [2..n]$
  $\mathsf{receive}_1(mid_{r,\alpha(r)})$
  do_1 `rd` $^{(n+2)m}$ // read $\mathbf{e}_\alpha$
procedure $test(r)$
  $u \leftarrow \mathsf{do}_1$ `rd` $^{(n+3)m}$
  $\mathsf{receive}_1(mid_{r,\frac{m}{n-1}})$
  $u' \leftarrow \mathsf{do}_1$ `rd` $^{(n+4)m}$
  return $\frac{m}{n-1} - (u' - u)$
	procedure $init$
	  for all $r \in [2..n]$
	  for all $i \in [1..\frac{m-1}{n-1}]$
	  do_r `add(0)` $^{rm+i}$
	  $\mathsf{send}_r(mid_{r,i})$
	procedure $exp(\alpha)$
	  for all $r \in [2..n]$
	  $\mathsf{receive}_1(mid_{r,\alpha(r)})$
	  do_1 `remove(0)` $^{(n+2)m}$
	  do_1 `rd` $^{(n+3)m}$ // read $\mathbf{e}_\alpha$
	procedure $test(r)$
	  for all $i \in [1..(\frac{m-1}{n-1})]$
	  $\mathsf{receive}_1(mid_{r,i})$
	  $u \leftarrow \mathsf{do}_1$ `rd` $^{(n+4)m+i}$
	  if $0 \in u$
	  return $i - 1$
	  return $\frac{m-1}{n-1}$

Definition of executions $\mathbb{C}_\alpha$	
$\mathbb{C}_\alpha = \mathsf{exec}(\mathcal{D}_\tau, (R_0, T_0), init; exp(\alpha))$ where $(R_0, T_0) = ([x \mapsto \mathcal{D}_\tau.\vec{\sigma}_0], \emptyset)$	
Definition of read-back function $\mathsf{rb} : \mathcal{D}_\tau.\Sigma \to Q$	
$\mathsf{rb}(\sigma) = \lambda r : [2..n].\mathsf{result}(\mathcal{D}_\tau, (R_{init}[(x,1) \mapsto \sigma], T_{init}), test(r))$ where $(R_{init}, T_{init}) = \mathsf{post}(\mathsf{exec}(\mathcal{D}_\tau, (R_0, T_0), init))$	

Figure 12. Example experiment ($n = 4$ and $m = 15$) and test for ctr. Gray dashed lines represent the configuration $(R_{init}[(x,1) \mapsto R_\alpha(x,1)], T_{init})$ where the $test$ driver program is applied.



where:

(i) This is due to $\mathcal{D}_{\mathtt{ctr}}\ \mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathsf{T\text{-}Any}]\ \mathcal{F}_{\mathtt{ctr}}$, as we explained informally above. Let

$$C'_\alpha = \mathsf{exec}(\mathcal{D}_{\mathtt{ctr}}, (R_0, T_0), (init; exp(\alpha); test(r))).$$

Then the operation context in $\mathsf{abs}(C'_\alpha, \mathcal{V}^{\mathsf{state}})$ of the first read in $test(r)$ contains $\sum_{r=2}^{n} \alpha(r)$ increments, while that of the second read contains $(m/(n-1)) - \alpha(r)$ more increments.

(ii) We have $T_\alpha = T_{init}$ because $exp(\alpha)$ does not send any messages. Also, R_α and $R_{init}[(x,1) \mapsto R_\alpha(x,1)]$ can differ only

in the states of the replicas $2..n$. These cannot influence the run of $test(r)$, since it performs events on replica 1 only. □

PROOF OF THEOREM 15. Given n_0, m_0, we pick $n = n_0$ and some $m \geq n_0$ such that m is a multiple of $(n-1)$ and $m \geq n^2$. Take the experiment family $(Q, n, m, \mathbb{C}, \mathbf{e}, \mathsf{rb})$ given by Lemma 16. Then for any α, $\mathbb{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha)$ is at most the total number of increments m in $\mathbb{C}_\alpha$. Using Lemma 14 and $m \geq n^2$, for some constants K_1, K_2, K_3, K independent from n_0, m_0 we get:

$$\mathsf{wcmo}(\mathcal{D}_{\mathtt{ctr}}, n, m) \geq \lfloor \lg_{|\Lambda|} |Q| \rfloor / (\max_{\alpha \in Q} \mathsf{len}(\mathbb{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha))) \geq$$

$$K_1 \frac{\lg_{|\Lambda|}(\frac{m}{n-1})^{n-1}}{\mathsf{len}_{\mathbb{N}_0}(m)} \geq K_2 \frac{n \lg(m/n)}{\lg m} \geq K_3 \frac{n \lg \sqrt{m}}{\lg m} \geq Kn. \quad \square$$

6.4 Lower Bound for State-Based OR-Set (`orset`)

THEOREM 17. *If* $\mathcal{D}_{\mathtt{orset}}\ \mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathsf{T\text{-}Any}]\ \mathcal{F}_{\mathtt{orset}}$*, then* $\mathcal{D}_{\mathtt{orset}}$ *is* $\widehat{\Omega}(n \lg m)$.

LEMMA 18. *If* $\mathcal{D}_{\mathtt{orset}}\ \mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathsf{T\text{-}Any}]\ \mathcal{F}_{\mathtt{orset}}$*,* $n \geq 2$ *and* $m \geq n$ *is such that* $(m-1)$ *is a multiple of* $(n-1)$*, then the tuple* $(Q, n, m, \mathbb{C}, \mathbf{e}, \mathsf{rb})$ *on the right in Fig. 11 is an experiment family.*

The proof is the same as that of Lemma 16, except for obligation (i). We therefore give only informal explanations.

The main idea of the experiments defined in the lemma is to force replica 1 to remember element instances even after they have been removed at that replica; cf. our explanation of the challenges of implementing the OR-set from §5.3. The experiments follow a similar pattern to those for `ctr`, but use different operations. In the common *init* phase, each replica $2..n$ performs $\frac{m-1}{n-1}$ operations adding a designated element 0, which are interleaved with sending messages. In the experiment phase $exp(\alpha)$, one message from each replica $r = 2..n$, selected by $\alpha(r)$, is delivered to replica 1. At the end of execution, replica 1 removes 0 from the set and performs the read $\mathbf{e}_\alpha$. The return value of this read is always the empty set.

To perform the read-back of $\alpha(r)$ for $r = 2..n$, $test(r)$ delivers all messages by replica r to replica 1 in the order they were sent and, after each such delivery, checks if replica 1 now reports the element 0 as part of the set. From $\mathcal{D}_{\mathtt{orset}}\ \mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathsf{T\text{-}Any}]\ \mathcal{F}_{\mathtt{orset}}$ and the definition (5) of $\mathcal{F}_{\mathtt{orset}}$, we get that exactly the first $\alpha(r)$ such deliveries will have no effect on the contents of the set: the respective add operations have already been observed by the remove operation that replica 1 performed in the experiment phase. Thus, if 0 appears in the set right after delivering the i-th message of replica r, then $\alpha(r) = i-1$, and if 0 does not appear by the time the loop is finished, then $\alpha(r) = (m-1)/(n-1)$.

PROOF OF THEOREM 17. Given n_0, m_0, we pick $n = n_0$ and some $m \geq n_0$ such that $(m-1)$ is a multiple of $(n-1)$ and $m \geq n^2$. Take the experiment family $(Q, n, m, \mathbb{C}, \mathbf{e}, \mathsf{rb})$ given by Lemma 18. For any $\alpha \in Q$, $\mathbb{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha) = \emptyset$, which can be encoded with a constant length. Using Lemma 14 and $m \geq n^2$, for some constants K_1, K_2, K we get:

$$\mathsf{wcmo}(\mathcal{D}_{\mathtt{orset}}, n, m) \geq \lfloor \lg_{|\Lambda|} |Q| \rfloor \,/\, (\max_{\alpha \in Q} \mathsf{len}(\mathbb{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha)))$$
$$\geq K_1 n \lg(m/n) \geq K_2 n \lg \sqrt{m} \geq Kn \lg m. \quad \square$$

7. Store Correctness and Consistency Axioms

Recall that we define a replicated store by a data type library $\mathbb{D}$ and a transport layer specification $\mathcal{T}$ (§4), and we specify its behavior by a function $\mathbb{F}$ from types $\tau \in \mathsf{dom}(\mathbb{D})$ to data type specifications and a set of consistency axioms (§3). The axioms are just constraints over abstract executions, such as those shown in Fig. 13; from now on we denote their sets by X. So far we have concentrated on single data type specifications $\mathbb{F}(\tau)$ and their correspondence to implementations $\mathbb{D}(\tau)$, as stated by Def. 7. In this section we consider consistency axioms and formulate the notion of correctness of the whole store $(\mathbb{D}, \mathcal{T})$ with respect to its specification $(\mathbb{F}, \mathsf{X})$.

Our first goal is to lift the statement of correctness given by Def. 7 to a store $(\mathbb{D}, \mathcal{T})$ with multiple objects of different data types. To this end, we assume a function $\mathbb{V}$ mapping each type $\tau \in \mathsf{dom}(\mathbb{D})$ to its visibility witness $\mathbb{V}$. This allows us to construct the visibility relation for a concrete execution $C \in [\![\mathbb{D}]\!] \cap \mathcal{T}$ by applying $\mathbb{V}(\tau)$ to its projection onto the events on every object of type τ:

$$\mathsf{witness}(\mathbb{V}) = \lambda C. \bigcup \{\mathbb{V}(\mathsf{type}(x))(C|_x) \mid x \in \mathsf{Obj}\},$$

where $\cdot|_x$ projects to events over x. Then the correctness of every separate data type τ in the store with respect to $\mathbb{F}(\tau)$ according to Def. 7 automatically ensures that the behavior of the whole store is consistent with $\mathbb{F}$ in the sense of Def. 5.

PROPOSITION 19. $(\forall \tau \in \mathsf{dom}(\mathbb{D}).\ \mathbb{D}(\tau)\ \mathsf{sat}[\mathbb{V}(\tau), \mathcal{T}]\ \mathbb{F}(\tau)) \implies$
$(\forall C \in [\![\mathbb{D}]\!] \cap \mathcal{T}.\ \mathsf{abs}(C, \mathsf{witness}(\mathbb{V})) \models \mathbb{F}).$

This motivates the following definition of store correctness. Let us write $A \models \mathsf{X}$ when the abstract execution A satisfies the axioms X.

DEFINITION 20. *A store* $(\mathbb{D}, \mathcal{T})$ *is* ***correct*** *with respect to a specification* $(\mathbb{F}, \mathsf{X})$*, if for some* $\mathbb{V}$*:*

(i) $\forall \tau \in \mathsf{dom}(\mathbb{D}).\ (\mathbb{D}(\tau)\ \mathsf{sat}[\mathbb{V}(\tau), \mathcal{T}]\ \mathbb{F}(\tau))$*; and*
(ii) $\forall C \in [\![\mathbb{D}]\!] \cap \mathcal{T}.\ (\mathsf{abs}(C, \mathsf{witness}(\mathbb{V})) \models \mathsf{X})$.

We showed how to discharge (i) in §5. The validity of axioms X required by (ii) most often depends on the transport layer specification $\mathcal{T}$: e.g., to disallow the anomaly (2) from §1, $\mathcal{T}$ needs to provide guarantees on how messages pertaining to different objects are delivered. However, data type implementations can also enforce axioms by putting enough information into messages: e.g., implementations correct with respect to $\mathcal{V}^{\mathsf{state}}$ from §4 ensure that vis is transitive regardless of the behavior of the transport layer. Fortunately, to establish (ii) in practice, we do not need to consider the internals of data type implementations in $\mathbb{D}$—just knowing the visibility witnesses used in the statements of their correctness is enough, as formulated in the following definition.

DEFINITION 21. *A set* W *of visibility witnesses and a transport layer specification* $\mathcal{T}$ ***validate*** *axioms* X*, if*

$$\forall C, \mathbb{V}.\ (C \in \mathcal{T}) \wedge (\{\mathbb{V}(\tau) \mid \tau \in \mathsf{dom}(\mathbb{V})\} \subseteq W) \implies$$
$$(\mathsf{abs}(C, \mathsf{witness}(\mathbb{V})) \models \mathsf{X}).$$

Since visibility witnesses are common to wide classes of data types (e.g., state- or op-based), our proofs of the validity of axioms will not have to be redone if we add new data type implementations to the store from a class already considered.

We next present axioms formalizing several variants of eventual consistency used in replicated stores (Fig. 13 and 14) and W and $\mathcal{T}$ that validate them. We then use this as a basis for discussing connections with weak shared-memory models. Due to space constraints, we defer technical details and proofs to [12, §D].

Basic eventual consistency. EVENTUAL and THINAIR define a weak form of eventual consistency. EVENTUAL ensures that an event cannot be invisible to infinitely many other events on the same object and thus implies (1) from §1: informally, if updates stop, then reads at all replicas will eventually see all updates and will return the same values (§3.2). However, EVENTUAL is stronger than quiescent consistency: the latter does not provide any guarantees at all for executions with infinitely many updates to the store, whereas our specification implies that the return values are computed according to $\mathbb{F}(\tau)$ using increasingly up-to-date view of the store state. We formalize these relationships in [12, §D].

THINAIR prohibits values from appearing "out-of-thin-air" [28], like 42 in Fig. 14(a) (recall that registers are initialized to 0). Cycles in $\mathsf{ro} \cup \mathsf{vis}$ that lead to out-of-thin-airs usually arise

Figure 13. A selection of consistency axioms over an execution $(E, \mathsf{repl}, \mathsf{obj}, \mathsf{oper}, \mathsf{rval}, \mathsf{ro}, \mathsf{vis}, \mathsf{ar})$

Auxiliary relations

$\mathsf{sameobj}(e, f) \iff \mathsf{obj}(e) = \mathsf{obj}(f)$

Per-object causality (aka happens-before) order:

$\mathsf{hbo} = ((\mathsf{ro} \cap \mathsf{sameobj}) \cup \mathsf{vis})^+$

Causality (aka happens-before) order: $\mathsf{hb} = (\mathsf{ro} \cup \mathsf{vis})^+$

Axioms

EVENTUAL:

$\forall e \in E. \neg(\exists \text{ infinitely many } f \in E. \mathsf{sameobj}(e, f) \wedge \neg(e \xrightarrow{\mathsf{vis}} f))$

THINAIR: $\mathsf{ro} \cup \mathsf{vis}$ is acyclic

POCV (Per-Object Causal Visibility): $\mathsf{hbo} \subseteq \mathsf{vis}$

POCA (Per-Object Causal Arbitration): $\mathsf{hbo} \subseteq \mathsf{ar}$

COCV (Cross-Object Causal Visibility): $(\mathsf{hb} \cap \mathsf{sameobj}) \subseteq \mathsf{vis}$

COCA (Cross-Object Causal Arbitration): $\mathsf{hb} \cup \mathsf{ar}$ is acyclic

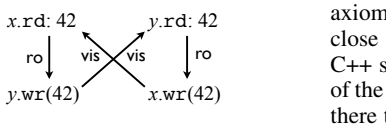
Figure 14. Anomalies allowed or disallowed by different axioms

(a) Disallowed by THINAIR:

$x, y : \texttt{intreg}$

$i = x.\texttt{rd} \;\|\; j = y.\texttt{rd}$

$y.\texttt{wr}(i) \;\|\; x.\texttt{wr}(j)$

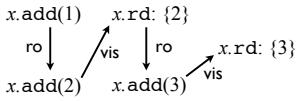


(b) Disallowed by POCV:

$x : \texttt{orset}$

$x.\texttt{add}(1) \;\|\; i = x.\texttt{rd} \;\|\; j = x.\texttt{rd}$

$x.\texttt{add}(2) \;\|\; x.\texttt{add}(3) \;\|$

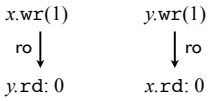


(c) Allowed by COCV and COCA:

$x, y : \texttt{intreg}$

$x.\texttt{wr}(1) \;\|\; y.\texttt{wr}(1)$

$i = y.\texttt{rd} \;\|\; j = x.\texttt{rd}$



from effects of speculative computations, which are done by some older replicated stores [36].

THINAIR is validated by $\{\mathcal{V}^{\mathsf{state}}, \mathcal{V}^{\mathsf{op}}\}$ and T-Any, and EVENTUAL by $\{\mathcal{V}^{\mathsf{state}}, \mathcal{V}^{\mathsf{op}}\}$ and the following condition on C ensuring that every message is eventually delivered to all other replicas and every operation is followed by a message generation:

$$(\forall e \in C.E. \forall r, r'. C.\mathsf{act}(e) = \mathsf{send} \wedge C.\mathsf{repl}(e) = r \wedge r \neq r' \implies \exists f. C.\mathsf{repl}(f) = r' \wedge e \xrightarrow{\mathsf{del}(C)} f) \wedge$$
$$(\forall e \in C.E. C.\mathsf{act}(e) = \mathsf{do} \implies \exists f. \mathsf{act}(f) = \mathsf{send} \wedge e \xrightarrow{\mathsf{roo}(C)} f),$$

where $\mathsf{roo}(C)$ is $\mathsf{ro}(C)$ projected to events on the same object.

Causality guarantees. Many replicated stores achieve availability and partition tolerance while providing stronger guarantees, which we formalize by the other axioms in Fig. 13. We call an execution ***per-object***, respectively, ***cross-object causally consistent***, if it is eventually consistent (as per above) and satisfies the axioms POCV and POCA, respectively, COCV and COCA. POCV guarantees that an operation sees all operations connected to it by a causal chain of events on the same object; COCV also considers causal chains via different objects. Thus, POCV disallows the execution in Fig. 14(b), and COCV the one in §3.1, corresponding to (2) from §1. POCA and COCA similarly require arbitration to be consistent with causality. The axioms highlight the principle of formalizing stronger consistency models: including more edges into vis and ar, so that clients have more up-to-date information.

Cross-object causal consistency is implemented by, e.g., COPS [27] and Gemini [23]. It is weaker than strong consistency, as it allows reading stale data. For example, it allows the execution in Fig. 14(c), where both reads fetch the initial value of the register, despite writes to it by the other replica. It is easy to check that this outcome cannot be produced by any interleaving of the events at the two replicas, and is thus not strongly consistent.

An interesting feature of per-object causal consistency is that state-based data types ensure most of it just by the definition of $\mathcal{V}^{\mathsf{state}}$: POCV is validated by $\{\mathcal{V}^{\mathsf{state}}\}$ and T-Any. If the witness set is $\{\mathcal{V}^{\mathsf{state}}, \mathcal{V}^{\mathsf{op}}\}$, then we need $\mathcal{T}$ to guarantee the following: informally, if a send event e and another event f are connected by a causal chain of events on the same object, then the message created by e is delivered to $C.\mathsf{repl}(f)$ by the time f is done. POCA is validated by $\{\mathcal{V}^{\mathsf{state}}, \mathcal{V}^{\mathsf{op}}\}$ and the transport layer specification $(\mathsf{roo}(C) \cup \mathsf{del}(C))^+|_{\mathsf{do}} \subseteq \mathsf{ar}(C)$. This states that timestamps of events on every object behave like a Lamport clock [22]. Conditions for COCV and COCA are similar.

There also exist consistency levels in between basic eventual consistency and per-object causal consistency, defined using so-called ***session guarantees*** [35]. We cover them in [12, §D].

Comparison with shared-memory consistency models. Interestingly, the specializations of the consistency levels defined by the axioms in Fig. 13 to the type `intreg` of LWW-registers are very close to those adopted by the memory model in the 2011 C and C++ standards [5]. Thus, POCA and POCV define the semantics of the fragment of C/C++ restricted to so-called *relaxed* operations; there this semantics is defined using *coherence* axioms, which are analogous to session guarantees [35]. COCV and COCA are close to the semantics of C/C++ restricted to *release-acquire* operations. However, C/C++ does not have an analog of EVENTUAL and does not validate THINAIR, since it makes the effects of speculations visible to the programmer [4]. We formalize the correspondence to C/C++ in [12, §D]. In the future, this correspondence may open the door to applying technology developed for shared-memory models to eventually consistent systems; promising directions include model checking [3, 9], automatic inference of required consistency levels [26] and compositional reasoning [4].

8. Related Work

For a comprehensive overview of replicated data type research we refer the reader to Shapiro et al. [32]. Most papers proposing new data type implementations [6, 31–33] do not provide their formal declarative specifications, save for the expected property (1) of quiescent consistency or first specification attempts for sets [6, 7]. Formalizations of eventual consistency have either expressed quiescent consistency [8] or gave low-level operational specifications [17].

An exception is the work of Burckhardt et al. [10, 13], who proposed an axiomatic model of causal eventual consistency based on visibility and arbitration relations and an operational model based on revision diagrams. Their store specification does not provide customizable consistency guarantees, and their data type specifications are limited to the sequential $\mathcal{S}$ construction from §3.2, which cannot express advanced conflict resolution used by the multi-value register, the OR-set and many other data types [32]. More significantly, their operational model does not support general op- or state-based implementations, and is thus not suited for studying the correctness or optimality of these commonly used patterns.

Simulation relations have been applied to verify the correctness of sequential [25] and shared-memory concurrent data type implementations [24]. We take this approach to the more complex setting of a replicated store, where the simulation needs to take into account multiple object copies and messages and associate them with structures on events, rather than single abstract states. This poses technical challenges not considered by prior work, which we address by our novel notion of replication-aware simulations.

The distributed computing community has established a number of asymptotic lower bounds on the complexity of implementing certain distributed or concurrent abstractions, including one-

shot timestamp objects [20] and counting protocols [15, 30]. These works have considered either programming models or metrics significantly different from ours. An exception is the work of Charron-Bost [14], who proved that the size of vector clocks [29] is optimal to represent the happens-before relation of a computation (similar to the visibility relation in our model). Specifications of `mvr` and `orset` rely on visibility; however, Charron-Bost's result does not directly translate into a lower bound on their implementation complexity, since a specification may not require complete knowledge about the relation and an implementation may represent it in an arbitrary manner, not necessarily using a vector.

9. Conclusion and Future Work

We have presented a comprehensive theoretical toolkit to advance the study of replicated eventually consistent stores, by proposing methods for (1) specifying the semantics of replicated data types and stores abstractly, (2) verifying implementations of replicated data types, and (3) proving that such implementations have optimal metadata overhead. By proving both correctness and optimality of four nontrivial data type implementations, we have demonstrated that our methods can indeed be productively applied to the kinds of patterns used by practitioners and researchers in this area.

Although our work marks a big step forward, it is only a beginning, and creates plenty of opportunities for future research. We have already made the first steps in extending our specification framework with more features, such as mixtures of consistency levels [23] and transactions [34, 37]; see [11]. In the future we would also like to study more data types, such as lists used for collaborative editing [32], and to investigate metadata bounds for data type implementations other than state-based ones, including more detailed overhead metrics capturing optimizations invisible to the worst-case overhead analysis. Even though our execution model for replicated stores follows the one used by replicated data type designers [33], there are opportunities for bringing it closer to actual implementations. Thus, we would like to verify the algorithms used by store implementations [27, 34, 37] that our semantics abstracts from. This includes fail-over and session migration protocols, which permit clients to interact with multiple physical replicas, while being provided the illusion of a single virtual replica.

Finally, by bringing together prior work on shared-memory models and data replication, we wish to promote an exchange of ideas and results between the research communities of programming languages and verification on one side and distributed systems on the other.

Acknowledgements. We thank Hagit Attiya, Anindya Banerjee, Carlos Baquero, Lindsey Kuper and Marc Shapiro for comments that helped improve the paper. Gotsman was supported by the EU FET project ADVENT, and Yang by EPSRC.

References

[1] Riak key-value store. http://basho.com/products/riak-overview/.

[2] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12), 1996.

[3] J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software verification for weak memory via program transformation. In *ESOP*, 2013.

[4] M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In *POPL*, 2013.

[5] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, 2011.

[6] A. Bieniusa, M. Zawirski, N. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. An optimized conflict-free replicated set. Technical Report 8083, INRIA, 2012.

[7] A. Bieniusa, M. Zawirski, N. M. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. Brief announcement: Semantics of eventually consistent replicated sets. In *DISC*, 2012.

[8] A.-M. Bosneag and M. Brockmeyer. A formal model for eventual consistency semantics. In *IASTED PDCS*, 2002.

[9] S. Burckhardt, R. Alur, and M. M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *PLDI*, 2007.

[10] S. Burckhardt, M. Fähndrich, D. Leijen, and B. P. Wood. Cloud types for eventual consistency. In *ECOOP*, 2012.

[11] S. Burckhardt, A. Gotsman, and H. Yang. Understanding eventual consistency. Technical Report MSR-TR-2013-39, Microsoft Research, 2013.

[12] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: specification, verification, optimality (extended version), 2013. http://research.microsoft.com/apps/pubs/?id=201602.

[13] S. Burckhardt, D. Leijen, M. Fähndrich, and M. Sagiv. Eventually consistent transactions. In *ESOP*, 2012.

[14] B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39(1), 1991.

[15] J.-Y. Chen and G. Pandurangan. Optimal gossip-based aggregate computation. In *SPAA*, 2010.

[16] N. Conway, R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *SOCC*, 2012.

[17] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-serializable data services. In *PODC*, 1996.

[18] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[19] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2), 2002.

[20] M. Helmi, L. Higham, E. Pacheco, and P. Woelfel. The space complexity of long-lived and one-shot timestamp implementations. In *PODC*, 2011.

[21] C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1, 1972.

[22] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7), 1978.

[23] C. Li, D. Porto, A. Clement, R. Rodrigues, N. Preguiça, and J. Gehrke. Making geo-replicated systems fast if possible, consistent when necessary. In *OSDI*, 2012.

[24] H. Liang, X. Feng, and M. Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *POPL*, 2012.

[25] B. Liskov and S. Zilles. Programming with abstract data types. In *ACM Symposium on Very High Level Languages*, 1974.

[26] F. Liu, N. Nedev, N. Prisadnikov, M. T. Vechev, and E. Yahav. Dynamic synthesis for relaxed memory models. In *PLDI*, 2012.

[27] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *SOSP*, 2011.

[28] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL*, 2005.

[29] F. Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1989.

[30] S. Moran, G. Taubenfeld, and I. Yadin. Concurrent counting. In *PODC*, 1992.

[31] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel Distrib. Comput.*, 71(3), 2011.

[32] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report 7506, INRIA, 2011.

[33] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *SSS*, 2011.

[34] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, 2011.

[35] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. Session guarantees for weakly consistent replicated data. In *PDIS*, 1994.

[36] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, 1995.

[37] M. Zawirski, A. Bieniusa, V. Balegas, S. Duarte, C. Baquero, M. Shapiro, and N. Preguiça. SwiftCloud: Fault-tolerant geo-replication integrated all the way to the client machine. Technical Report 8347, INRIA, 2013.

Understanding Eventual Consistency

Sebastian Burckhardt
Microsoft Research

Alexey Gotsman
IMDEA Software Institute

Hongseok Yang
University of Oxford

Abstract. Modern geo-replicated databases underlying large-scale Internet services guarantee immediate availability and tolerate network partitions at the expense of providing only weak forms of consistency, commonly dubbed eventual consistency. At the moment there is a lot of confusion about the semantics of eventual consistency, as different systems implement it with different sets of features and in subtly different forms, stated either informally or using disparate and low-level formalisms.

We address this problem by proposing a framework for formal and declarative specification of the semantics of eventually consistent systems using axioms. Our framework is fully customisable: by varying the set of axioms, we can rigorously define the semantics of systems that combine any subset of typical guarantees or features, including conflict resolution policies, session guarantees, causality guarantees, multiple consistency levels and transactions. We prove that our specifications are validated by an example abstract implementation, based on algorithms used in real-world systems. These results demonstrate that our framework provides system architects with a tool for exploring the design space, and lays the foundation for formal reasoning about eventually consistent systems.

1. Introduction

Modern large-scale Internet services rely on distributed database systems that maintain multiple replicas of data. Often such systems are *geo-replicated*, meaning that the replicas are located in geographically distinct locations. Geo-replication requires the systems to tolerate network partitions, yet end-user applications also require them to provide immediate availability. Ideally, we would like to achieve these two requirements while also providing *strong consistency*, which roughly guarantees that the outcome of a set of concurrent requests to the database is the same as what one can obtain by executing these requests atomically in some sequence. Unfortunately, the famous CAP theorem [18] shows that this is impossible. For this reason, modern geo-replicated systems provide weaker forms of consistency, commonly dubbed *eventual consistency* [32]. Here the word 'eventual' refers to the guarantee that

> if update requests stop arriving to the database, then it will eventually reach a consistent state. (1)

Geo-replication is a hot research area, and new architectures for eventually consistent systems appear every year [5, 14, 15, 17, 21, 24, 29, 30]. Unfortunately, whereas consistency models of classical relational databases have been well-studied [9, 26], those of geo-replicated systems are poorly understood. The very term eventual consistency is a catch-all buzzword, and different systems claiming to be eventually consistent actually provide subtly different guarantees and features. Commonly used ways of their specification are inadequate for several reasons:

- **Disparate and low-level formalisms.** Specifications of consistency models proposed for various systems are stated informally or using disparate formalisms, often tied to system implementations. This makes it hard to compare guarantees provided by different systems or apply ideas from one of them in another.
- **Weak guarantees.** More declarative attempts to formalise eventual consistency [29] have identified it with property (1), which actually corresponds to a form of *quiescent consistency* from distributed computing [19]. However, such reading of eventual consistency does not allow making conclusions about the behaviour of the database in realistic scenarios, when updates *never* stop arriving.
- **Conflict resolution policies.** To satisfy the requirement of availability, geo-replicated systems have to allow making updates to the same object on different, potentially disconnected replicas. The systems then have to resolve conflicts, arising when replicas exchange the updates, according to certain policies, often encapsulated in *replicated data types* [27, 29]. The use of such policies complicates the semantics provided by eventually consistent systems and makes its formal specification challenging.
- **Combinations of different consistency levels.** Even in applications where basic eventual consistency is sufficient most of the time, stronger consistency may be needed occasionally. This has given rise to a wide variety of features for strengthening consistency on demand. Thus, some systems now provide a mixture of eventual and strong consistency [1, 13, 21], and researchers have argued for doing the same with different forms of eventual consistency [5]. Other systems have allowed strengthening consistency by implementing transactions, usually not provided by geo-replicated systems [14, 24, 30]. Understanding the semantics of such features and their combinations is very difficult.

The absence of a uniform and widely applicable specification formalism complicates the development and use of eventually consistent systems. Currently, there is no easy way for developers of such systems to answer basic questions when designing their programming interfaces: Are the requirements of my application okay with a given form of eventual consistency? Can I use a replicated data type implemented in a system X in a different system Y? What is the semantics of combining two given forms of eventual consistency?

We address this problem by proposing a formal and declarative framework for specifying the semantics of eventually consistent systems. In our proposal, the specification of a consistency model consists of the following components:

- **Replicated data type specifications** define the basic data types supported by the database and determine the conflict resolution policies at the level of individual objects (§2). We show how to specify abstractly a variety of data types, including registers, counters, multi-value registers [17] and observed-remove sets [28].
- **Consistency specification** extends the semantics of individual objects to that of the entire database (§3). It is defined by a set of *axioms*, constraining requests submitted to the database by its clients and relations on these requests that abstract the way the database processes them. We show how to specify consistency guarantees implemented by a range of existing eventually consistent systems, including weak forms of eventual consistency [2, 17], session guarantees [31] and different kinds of causal consistency [14, 24, 29, 30].
- **Interfaces for strengthening consistency** can include *consistency annotations*, which allow users to specify the consistency level of a single operation (§4.1), *fences*, which affect the consistency level of multiple operations (§4.2), and *transactions*, which ensure that a group of operations executes atomically (§5). We provide the analysis of the trade-offs between some of these features (§4.3).

The main technical challenge we have to deal with is that the above aspects of the system behaviour interact in subtle ways, making it difficult to specify each of them separately. We resolve this problem by representing the information about the database execution that all of the specification components rely on abstractly, in a way not tied to the database implementation (visibility and arbitration relations in §2). This interface between different specification components yields a fully customisable framework, which can define the semantics of any combination of typical guarantees or features of eventually consistent systems. Our technique is a generalisation of approaches used in the context of *weak shared-memory consistency models* [3, 8], which have been extensively studied by the programming languages and verification communities[1]. Overall, we make the following contributions:

- We systematise the knowledge about the existing forms of eventual consistency and provide a single specification framework that can express many guarantees and features of existing systems [2, 13, 14, 17, 21, 24, 29–31]. While some of the concepts that we specify have been formally defined before individually, to our best knowledge, our framework is the first to allow handling all of them uniformly and drawing conclusions about their interactions. This provides the developers of eventually consistent systems with a tool for exploring the design space.
- We justify the correspondence of our specifications to real-world systems by proving that the specifications are validated by an example abstract implementation, based on algorithms used in such systems.
- We prove that our specifications are more powerful than quiescent consistency, allowing us to make conclusions about the database behaviour even in the presence of a continuous stream of updates arriving from its clients(§3.4).
- We prove several results demonstrating how our framework supports rigorous comparisons between various features and guarantees (Theorem 6, Theorem 8, Proposition 9). In particular, we establish that fences are sufficient to recover sequential consistency and serializable transactions in an eventually consistent system. An analogous result for modern shared-memory models was proved only recently [7].

Due to the volume of our technical development, we have relegated all proofs to §A. The definition of the abstract implementation and the proof of its correspondence with our specifications are deferred to §B; however, we present the algorithms used in the implementation informally throughout the paper.

[1] Although our axioms may appear different on the surface, we discover that, when specialised to the integer register data type, common forms of eventual consistency correspond almost exactly to fragments of the memory model adopted in the 2011 C and C++ standards [8] (§3.4). This suggests that existing techniques for testing and verifying programs running on weak memory models [6, 12, 23] may prove beneficial for eventually consistent systems.

2. Replicated Data Types

In this paper we consider a replicated database storing ***objects*** $\mathsf{Obj} = \{x, y, \dots\}$ that have ***values*** from a set Val. For simplicity, we assume that the set of objects in the database is fixed. Every object $x \in \mathsf{Obj}$ has a ***type*** $\mathsf{type}(x) \in \mathsf{Type}$ that determines the set $\mathsf{Op}_{\mathsf{type}(x)}$ of ***operations*** that clients of the database can perform on it. For example, the data types can include a counter ctr and an integer register intreg with operations for reading, incrementing or writing an integer k: $\mathsf{Op}_{\mathsf{ctr}} = \{\mathsf{rd}, \mathsf{inc}\}$ and $\mathsf{Op}_{\mathsf{intreg}} = \{\mathsf{rd}\} \cup \{\mathsf{wr}(k) \mid k \in \mathbb{Z}\}$.

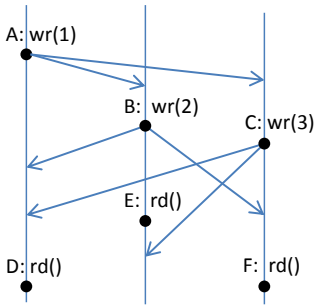
Even though the database may store the same object on multiple replicas, its interface is agnostic to this: a client just requests an operation on an object, without specifying a replica it is stored on. We now present a way of defining the semantics of common replicated data types that matches this level of abstraction by hiding implementation aspects, such as the replication strategy, network topology or transport layer. This allows us to understand how the same data type can be implemented on different system architectures. Our discussion focuses on the behaviour of individual operations on a single object.

In strongly consistent systems, the outcome of a set of concurrent operations on an object can be obtained by executing them atomically in some sequence. In this setting, the semantics of a data type τ can be completely specified by a function $\mathcal{S}_\tau : \mathsf{Op}_\tau^+ \to \mathsf{Val}$, which, given a nonempty sequence of operations performed on an object, specifies the return value of the last operation (we assume that $\bot \in \mathsf{Val}$ is used for operations that return no value). For a counter, read operations return the number of preceding increment operations. For an integer register, read operations return the value of the last preceding write, or zero if there was no prior write. We can thus define $\mathcal{S}_{\mathsf{ctr}}$ and $\mathcal{S}_{\mathsf{intreg}}$ as follows: for any sequence of operations σ,

$$\begin{aligned}
\mathcal{S}_{\mathsf{ctr}}(\sigma\,\mathsf{rd}) &= (\text{the number of } \mathsf{inc} \text{ operations in } \sigma);\\
\mathcal{S}_{\mathsf{intreg}}(\sigma\,\mathsf{rd}) &= k, \text{ if } \mathsf{wr}(0)\,\sigma = \sigma_1\,\mathsf{wr}(k)\,\sigma_2 \text{ and}\\
&\qquad \sigma_2 \text{ does not contain } \mathsf{wr} \text{ operations};\\
\mathcal{S}_{\mathsf{ctr}}(\sigma\,\mathsf{wr}(k)) &= \mathcal{S}_{\mathsf{intreg}}(\sigma\,\mathsf{inc}) = \bot.
\end{aligned}$$

In an eventually consistent system, data type semantics is more complicated. In such a system, operations on the same

object can be issued concurrently at multiple replicas. To achieve availability and partition tolerance, the system performs operations at each replica immediately, and communicates this to other replicas only after the fact. The following diagram illustrates an execution of a system with integer registers by arranging labelled operations on several vertical timelines corresponding to replicas, with the delivery of updates to other replicas shown as diagonal arrows:



Previous work has described a variety of implementations of *replicated data types* that operate in this manner (see [28] for a survey). Since replicas in these implementations cannot be immediately aware of all operations, they may at times be inconsistent. The key challenge is to ensure that, once all updates are delivered to all replicas, they resolve conflicts between them uniformly and converge to the same state. Exactly how this is achieved varies greatly; we identify the following strategies:

1. **Make concurrent operations commutative**. We require that all operations be commutative, so that we can apply them in any order using the standard sequential semantics. This strategy works for counters, but not integer registers.
2. **Order concurrent operations**. The system totally orders all concurrent operations in some disciplined way, e.g., using timestamps. This allows applying the sequential semantics.
3. **Flag conflicts**. The system detects the presence of a conflict and lets the user deal with it. For example, the multi-value register used in Amazon's Dynamo key-value store [17] defines the return value of the read D in the above example as the set $\{2, 3\}$ of conflicting values.
4. **Resolve conflicts semantically**. The system detects a conflict and takes some data-type-dependent action to resolve it [27, 29]. For example, a replicated data type of *observed-remove set* [10] resolves conflicting operations trying to add and remove the same element so that an add always wins.

We now present a single formalization that is sufficiently general to represent all the four strategies and give an example of a replicated data type for each of them. We specify a data type τ by a function $\mathcal{F}_\tau$, generalizing $\mathcal{S}_\tau$. Just like with $\mathcal{S}_\tau$, we want $\mathcal{F}_\tau$ to determine the return value of an operation based on prior operations performed on the object. However, there are some important differences:

1. In the sequential setting, an operation takes into account the effect of all operations preceding it in the sequence. In the concurrent setting, the result depends on what subset of operations is visible in a given context. For example, in the diagram above, C is visible to D, but not to E.
2. In the concurrent setting, the result may also depend on additional information used to order events. For example, the relative order of the concurrent writes B and C in the diagram above may be determined using a timestamp, thus guaranteeing that reads D and F return the same result, even though they receive B and C in different orders.

The main insight of our formalisation is that we can specify the information about such relationships between events declaratively, without referring to implementation-level concepts, such as replicas or messages. Namely, $\mathcal{F}_\tau$ takes as a parameter not a sequence, but an *operation context*, which encapsulates "all we need to know" about a system execution to determine the return value of a given operation.

DEFINITION 1. *An **operation context** for a data type τ is a tuple $C = (f, V, \mathsf{ar}, \mathsf{vis})$, where*

- *$f \in \mathsf{Op}_\tau$ is the operation about to be performed.*
- *V is a set of **operation events** of the form (e, g), where e is a unique event identifier and $g \in \mathsf{Op}_\tau$. The set V includes all operations that are visible to f and can thus affect its result.*
- *$\mathsf{vis} \subseteq V \times V$ is a **visibility relation**, recording the relationships between operations in V motivated by point 1 above.*
- *$\mathsf{ar} \subseteq V \times V$ is a total and irreflexive **arbitration relation**, recording the relationships motivated by point 2 above.*

For a relation r we write $(u, v) \in \mathsf{r}$ and $u \xrightarrow{\mathsf{r}} v$ interchangeably. The vis relation describes the relative visibility of events in V: $u \xrightarrow{\mathsf{vis}} v$ means that the effect of u is visible to v. As we show below, it is needed to define certain data types. In implementation terms, an event u might be considered visible to an event v if the update performed by u has been delivered to the replica performing v before v is issued. The ar relation represents the ordering information provided by the system; $u \xrightarrow{\mathsf{ar}} v$ means that u is ordered before v. In an implementation it can be constructed using timestamps or tie-breaking mechanisms.

DEFINITION 2. *A **replicated data type specification** for a type τ is a function $\mathcal{F}_\tau$ that, given an operation context C for τ, specifies a return value $\mathcal{F}_\tau(C) \in \mathsf{Val}$, and that does not depend on the operation identifiers in C.*

Note that $\mathcal{F}_\tau(f, \emptyset, \emptyset, \emptyset)$ returns the value resulting from performing f on the default state for the data type (e.g., zero for an integer register). Also, $\mathcal{F}_\tau$ is deterministic: all the nondeterminism present in the distributed system is resolved by the operation context. We emphasise that our specifications do not prescribe a particular way in which the visibility and arbitration relations are actually represented in an implementation: the above references to implementations are only illustrative. Thus, the specifications can be viewed as generalising the concept of an *abstract data type* [22] to the replicated case. We now give several examples of data type specifications, corresponding to the four implementation strategies mentioned earlier.

Example 1: Counter (ctr) is defined by

$$\mathcal{F}_{\mathsf{ctr}}(\mathsf{inc}, V, \mathsf{vis}, \mathsf{ar}) = \bot;$$
$$\mathcal{F}_{\mathsf{ctr}}(\mathsf{rd}, V, \mathsf{vis}, \mathsf{ar}) = (\text{the number of inc operations in } V).$$

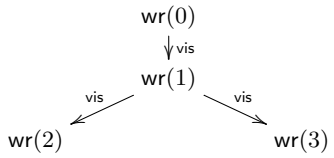
Note that the result does not depend on ar or vis, but only on V. Hence, this example is representative of strategy 1.

Example 2: Integer register (intreg) is defined by $\mathcal{F}_{\mathsf{intreg}}(f, V, \mathsf{vis}, \mathsf{ar}) = \mathcal{S}_{\mathsf{intreg}}(V^{\mathsf{ar}} f)$, where V^{ar} denotes the sequence obtained by ordering the operations in the set V of events visible to f according to the arbitration relation ar. This represents strategy 2: the return value is determined by establishing a total order of the visible operations and applying the regular sequential semantics. Note that the relative visibility between events in V, given by vis, is not used. We can similarly obtain a concurrent semantics $\mathcal{F}_\tau$ of any data type τ based on its sequential semantics $\mathcal{S}_\tau$. For example, $\mathcal{F}_{\mathsf{ctr}}$ as defined in Example 1 is equivalent to what we obtain using this generic construction. The next two examples go beyond this.

Example 3: Multi-value register (mvr)**.** A multi-value register has the same operations as the integer register, but its reads return a set of values, rather than a single one:

$$\mathcal{F}_{\mathsf{mvr}}(\mathsf{rd}, V, \mathsf{vis}, \mathsf{ar}) = \{k \mid \exists e.\, (e, \mathsf{wr}(k)) \text{ is maximal in } V\},$$

where an element $v \in V$ is maximal in V if there exists no $v' \in V$ such that $v \xrightarrow{\mathsf{vis}} v'$. Here a read returns all versions of the object that are not superseded by later writes, as determined by vis; the ar relation is not used. A multi-value register thus detects conflicting writes, as in strategy 3. For example, a rd operation would return $\{2, 3\}$ in the following context:



Example 4: Observed-remove set (orset)**.** Assume we want to implement a set of integers. Consider the operation context

$$(\mathsf{rd}, \{(\mathsf{A}, \mathsf{add}(42)), (\mathsf{B}, \mathsf{remove}(42))\}, \mathsf{vis}, \mathsf{ar}).$$

If we use the generic construction from Example 2, the result of rd will depend on the arbitration relation: $\emptyset$ if $\mathsf{A} \xrightarrow{\mathsf{ar}} \mathsf{B}$, and $\{42\}$ otherwise. In some cases, the application semantics may require a different outcome, e.g., that an add operation always win against concurrent remove operations. Bieniusa et al. [10] propose the ***observed-remove set*** orset that achieves this semantics by mandating that remove operations cancel out only the add operations that are visible to them:

$$\mathcal{F}_{\mathsf{orset}}(\mathsf{rd}, V, \mathsf{vis}, \mathsf{ar}) = \{k \mid \exists \text{ live } v = (e, \mathsf{add}(k)) \in V\},$$

where $v = (e, \mathsf{add}(k)) \in V$ is live if there does not exist $v' = (e', \mathsf{remove}(k)) \in V$ such that $v \xrightarrow{\mathsf{vis}} v'$. In the above operation context rd will return $\emptyset$ if $\mathsf{A} \xrightarrow{\mathsf{vis}} \mathsf{B}$, and $\{42\}$ otherwise.

Although replicated data types can simplify the programming of eventually consistent systems, they are not by themselves sufficient for general systems that contain more than one object. For example, consider a client program that updates a set object `friends` and then a list object `wall`:

```
friends.remove(boss); wall.append(photo);
```

In this case, we may want the database to ensure that the order of the updates is preserved. In terms of our framework, such guarantees depend on properties of vis and ar relations that are not defined by data type specifications. In the following sections we provide means of their specification.

3. Axiomatic Specification Framework

We extend our framework for specifying replicated data types (§2) to defining the semantics of the entire database. Our specifications can capture a range of forms of eventual consistency:

- **Basic eventual consistency**, which only guarantees that the effect of every operation will eventually become visible to all participants (§3.2).
- **Ordering guarantees**, which ensure that the system preserves the order in which operations are performed (§3.3) .
- **Interfaces for on-demand consistency strengthening**, which include consistency annotations and fences (§4), as well as transactions (§5).

We start by formalising client-database interactions (§3.1). We describe a run of the database by a *history*, which records all (possibly infinitely many) such interactions. We then specify its consistency model by a number of *axioms* (see Figure 1), which define the set of all histories it can produce.

3.1 Client Interaction Model

Clients often wish to perform multiple operations within some context, and to capture this, we introduce the notion of a ***session***, identified by session identifiers $\mathsf{SId} = \{1, 2, 3, \dots\}$. Sessions are different from transactions (introduced in §5): rather than providing advanced guarantees, such as atomicity or isolation, they are a means of tracking client identity across multiple requests. The purpose of sessions is to satisfy basic ordering expectations [31], such as whether a read following a write is guaranteed to see the effects of that write. In some systems, this is ensured by permanently binding a session to a single replica.

Each client operation is described in a history by an *action*, which enriches operation events (Definition 1) with session identifiers and return values.

DEFINITION 3. *An **action** is a tuple $(e, s, [x.f : k])$, where e is a unique action identifier, $s \in \mathsf{SId}$ is the identifier of the session the action takes place in, $f \in \mathsf{Op}_{\mathsf{type}(x)}$ is an operation performed on an object $x \in \mathsf{Obj}$, and $k \in \mathsf{Val}$ is its return value. We denote the set of actions by* Act.

We omit return values equal to $\bot$. For $a = (e, s, [x.f : k])$, we let $\mathsf{ses}(a) = s$, $\mathsf{obj}(a) = x$, $\mathsf{op}(a) = f$, $\mathsf{rval}(a) = k$, $\mathsf{type}(a) = \mathsf{type}(x)$ and $\mathsf{event}(a) = (e, f)$. We also use the event projection on sets of actions and relations over them.

A history consists of a set of actions, together with a ***session order***, relating actions within a session according to the order in which they were issued by the client.

DEFINITION 4. *A **history** is a pair (A, so), where $A \subseteq \mathsf{Act}$, A has no duplicate action identifiers, $\mathsf{so} \subseteq A \times A$, and* so *satisfies the axiom* SOWF *in Figure 1.*

Histories abstract from internal database execution completely. However, in order to define concepts arising in differ-

ent forms of eventual consistency, we need to know more about how operations relate to each other. To this end, we enrich histories with additional information about the internal database execution. As in §2, we record this information in an abstract form using the visibility and arbitration relations.

DEFINITION 5. *An **execution** is a tuple* $X = (A, \mathsf{so}, \mathsf{vis}, \mathsf{ar})$*, where* (A, so) *is a history, and* $\mathsf{vis}, \mathsf{ar} \subseteq A \times A$ *are such that the axioms* VISWF *and* ARWF *from Figure 1 hold.*

The vis and ar relations have the same intuitive meaning as those in §2. The difference is that, previously, we only considered these relations on a set of events needed to determine the outcome of a particular operation. In contrast, here the relations are defined on the set of all operations on all objects accessed in a run of a database. Given $X = (A, \mathsf{so}, \mathsf{vis}, \mathsf{ar})$ and $a \in A$, we can easily extract the operation context of a by selecting all actions visible to it according to vis:

$$\mathsf{ctxt}(a) = (\mathsf{op}(a), \mathsf{event}(B), \mathsf{event}(\mathsf{vis}|_B), \mathsf{event}(\mathsf{ar}|_B)),$$

where $B = \mathsf{vis}^{-1}(a)$. ARWF implies that the arbitration relation is sufficient to fully order all operations in $\mathsf{vis}^{-1}(a)$, and, thus, $\mathsf{ctxt}(a)$ is indeed an operation context.

Figure 2(a) shows an example of an execution, corresponding to the scenario of posting a photo from §2, but expressed using integer registers. In diagrams throughout this paper, we omit action and session identifiers. In the execution, a session first writes to a register x, setting the access permission to `all`. Some time later, it changes the permission to `noboss` and then writes to a register y to post a photo. The arbitration relation ar states that any session that sees both writes to x should assume that `noboss` overwrites `all`. However, as shown by the vis edges, in this example another session sees the photo, but not the updated access permission. In an implementation this anomaly can happen when the replica that session is connected to receives the updates corresponding to the writes of `noboss` and `photo` out of the order they were issued in [2, 17].

Our notion of an execution is similar to structures used for defining memory consistency models of hardware [3] and programming languages [8]. In particular, the visibility relation is similar to the "reads-from" relation used in such models. Unlike "reads-from", our visibility relation captures *all* delivered updates, so as to handle replicated data types.

3.2 Axiomatic System Specifications

The notion of a history corresponds straightforwardly to runs of a real-world database. Thus, we consider a ***system specification*** to be simply a set of histories. We can check if a particular database correctly implements the specification by determining whether, for all its runs, the corresponding history is in the set. In our specification framework, we obtain a system specification by choosing a set of ***axioms*** A from Figure 1 that constrain executions. The corresponding specification includes all histories that can be extended to an execution satisfying A:

$$(A, \mathsf{so}) \in [\![\mathsf{A}]\!] \iff \exists \mathsf{vis}, \mathsf{ar}.\ (\mathsf{A}, \mathsf{so}, \mathsf{vis}, \mathsf{ar}) \text{ satisfies } \mathsf{A}.$$

The fewer axioms are chosen, the weaker the consistency model is. The weakest sensible specification in our framework is $[\![\text{SOWF}, \text{VISWF}, \text{ARWF}, \text{RVAL}, \text{EVENTUAL}, \text{THINAIR}]\!]$. We call it ***basic eventual consistency***, and executions satisfying its axioms, ***eventually consistent***. The specification includes no ordering guarantees, but satisfies basic expectations, and is implemented, e.g., by Dynamo [17]. Its axioms have the following purpose:

- The well-formedness axioms SOWF, VISWF, ARWF state basic properties of the relations that we are working with.
- RVAL ensures that data types behave according to their specifications, i.e., the return value of every action a is computed on its operation context using the specification $\mathcal{F}_{\mathsf{type}(a)}$ for the type of the object accessed in a (§2). For example, in the execution in Figure 2(a) the operation context of the read from x includes only the write of `all`, so by RVAL and $\mathcal{F}_{\mathsf{intreg}}$, the read returns `all`. If the execution contained a vis edge from the write of `noboss` to the read from x, then the context would include both writes to x, with `noboss` taking precedence over `all` according to ar. By RVAL and $\mathcal{F}_{\mathsf{intreg}}$, the read would then have to return `noboss`.
- EVENTUAL ensures that an action cannot be invisible to infinitely many other actions on the same object and thus formalises (1) from §1.
- THINAIR rules out counterintuitive behaviours that are not produced by most real-world eventually consistent systems. For example, the execution in Figure 2(b) is allowed by RVAL and EVENTUAL, but not by THINAIR. Here the value 42 appears "out-of-thin-air", due to a cycle in $\mathsf{so} \cup \mathsf{vis}$ (recall that registers are initialised to 0). Such cycles usually arise from effects of speculative computations, which are done by some older eventually consistent systems [32], but not by modern ones. In this paper, we restrict ourselves to forms of eventual consistency implemented by the latter [29].

Our framework allows checking easily whether a given outcome is allowed by the consistency model without considering the system implementation: one just needs to ensure that all the axioms are satisfied. For example, the execution in Figure 2(a) is allowed by the axioms of basic eventual consistency.

Like in §2, our formalisation does not prescribe a particular way of representing visibility and arbitration in an implementation. Hence, these relations form an abstract interface between replicated data types used for conflict resolution and the underlying consistency model of the database. This makes our specification framework fully customisable and allows exploring the design space of consistency models easily: each of these two aspects can be varied separately, and the framework will define the semantics of any possible combination. We achieve this level of modularity even though, in real-world systems, implementations of the two aspects are often tightly interlinked [28].

Abstract implementation. We justify the soundness of our axioms by proving that they are validated by an abstract implementation based on algorithms used in existing systems [2, 14, 17, 24, 30, 31]: the set of histories it produces is included into the corresponding system specification (Theorem 11, §B). Our implementation of basic eventual consistency corresponds straightforwardly to the operational explanations we have given so far. The database consists of an arbi-

Figure 1. Axioms of eventual consistency. Here $\mathsf{r}|_B$ denotes the projection of a relation r to B: $\mathsf{r}|_B = (\mathsf{r} \cap (B \times B))$.

WELL-FORMEDNESS AXIOMS

SOWF: so is the union of transitive, irreflexive and total orders on actions by each session

VISWF: $\forall a, b.\ a \xrightarrow{\mathsf{vis}} b \implies \mathsf{obj}(a) = \mathsf{obj}(b)$

ARWF: $\forall a, b.\ a \xrightarrow{\mathsf{ar}} b \implies \mathsf{obj}(a) = \mathsf{obj}(b)$, ar is transitive and irreflexive, and $\mathsf{ar}|_{\mathsf{vis}^{-1}(a)}$ is a total order for all $a \in A$

DATA TYPE AXIOM

RVAL: $\forall a \in A.\ \mathsf{rval}(a) = \mathcal{F}_{\mathsf{type}(a)}(\mathsf{ctxt}(a))$

BASIC EVENTUAL CONSISTENCY AXIOMS

EVENTUAL:
$\forall a \in A.\ \neg(\exists \text{ infinitely many } b \in A.\ \mathsf{sameobj}(a, b) \wedge \neg(a \xrightarrow{\mathsf{vis}} b))$

THINAIR: so ∪ vis is acyclic

AUXILIARY RELATIONS

Per-object session order: $\mathsf{soo} = (\mathsf{so} \cap \mathsf{sameobj})$

Per-object causality order: $\mathsf{hbo} = (\mathsf{soo} \cup \mathsf{vis})^+$

Causality order: $\mathsf{hb} = (\mathsf{so} \cup \mathsf{vis})^+$

SESSION GUARANTEES

RYW (Read Your Writes). An operation sees all previous operations by the same session: $\mathsf{soo} \subseteq \mathsf{vis}$

MR (Monotonic Reads). An operation sees all operations previously seen by the same session: $(\mathsf{vis}; \mathsf{soo}) \subseteq \mathsf{vis}$

WFRV (Writes Follow Reads in Visibility). Operations are made visible at other replicas after operations on the same object that were previously seen by the same session: $(\mathsf{vis}; \mathsf{soo}^*; \mathsf{vis}) \subseteq \mathsf{vis}$

WFRA (Writes Follow Reads in Arbitration). Arbitration orders an operation after other operations previously seen by the same session: $(\mathsf{vis}; \mathsf{soo}^*) \subseteq \mathsf{ar}$

MWV (Monotonic Writes in Visibility). Operations are made visible at other replicas after all previous operations on the same object by the same session: $(\mathsf{soo}; \mathsf{vis}) \subseteq \mathsf{vis}$

MWA (Monotonic Writes in Arbitration). Arbitration orders an operation after all previous operations by the same session: $\mathsf{soo} \subseteq \mathsf{ar}$

CAUSALITY AXIOMS

POCV (Per-Object Causal Visibility): $\mathsf{hbo} \subseteq \mathsf{vis}$

POCA (Per-Object Causal Arbitration): $\mathsf{hbo} \subseteq \mathsf{ar}$

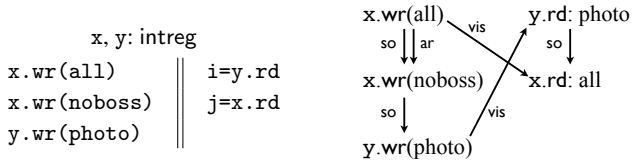
COCV (Cross-Object Causal Visibility): $(\mathsf{hb} \cap \mathsf{sameobj}) \subseteq \mathsf{vis}$

COCA (Cross-Object Causal Arbitration): hb ∪ ar is acyclic

trary number of replicas, each storing a log of actions. Every client session is connected to some replica, although a session can switch to another one at any time. When a session issues an operation, its return value is computed immediately on the basis of the state of the replica the session is connected to, and the corresponding action is appended to its log. From time to time, each replica broadcasts updates to the others, subject to a fairness constraint that every update will eventually get delivered to every replica (needed to validate EVENTUAL). Then $a \xrightarrow{\mathsf{vis}} b$ if a has been delivered to the replica performing b before b is issued. Thus, so and vis edges go forwards in time, so that THINAIR is validated. The arbitration relation is computed using Lamport timestamps [20].

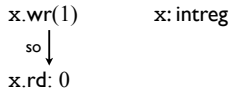
Figure 2. Anomalies allowed or disallowed by different axioms from Figure 1. In some cases, we show the code of client sessions that could produce the execution (where `i` and `j` are client-local variables).

(a) Disallowed by COCV:

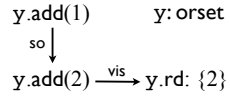


(b) Disallowed by THINAIR:

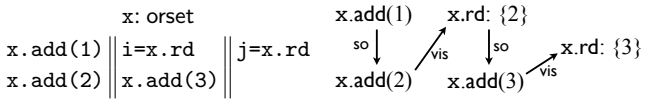
(c) Disallowed by RYW:



(d) Disallowed by MWV:



(e) Disallowed by POCV:



(f) Allowed by all axioms in Figure 1:

3.3 Classification of Ordering Guarantees

Basic eventual consistency provides very few guarantees to clients, allowing the anomaly in Figure 2(a) and, in fact, even more straightforward anomalies shown in Figures 2(c) and 2(d). In Figure 2(c), a session does not see a write it made (as signified by the absence of a vis edge) and thus reads the initial value of the register `x`. In an implementation this can happen when the read connects to another replica, which has not yet received the update corresponding to the write operation. In Figure 2(d), a session inserts 1 and then 2 into a set `y`; another session sees the first insertion, but not the second. In an implementation this can happen when the two insertions are propagated to other replicas out of order.

Many systems achieve availability and partition tolerance while providing stronger guarantees on the ordering of operations. In this paper we formalise the following ones:

Form of eventual consistency	Implementations
Basic eventual consistency	[2, 17]
Session guarantees	[31]
Per-object causal consistency	[29]
(Cross-object) causal consistency	[14, 21, 24, 30]

The differences can be subtle; it is one of our contributions to provide a means for precise comparisons. The cornerstones of our discussion are the axioms shown in Figure 1 and the anomalies shown in Figure 2.

Session guarantees. Let r^* denote the reflexive and transitive closure of a relation r, $\mathsf{r}_1; \mathsf{r}_2$ the composition of binary relations r_1 and r_2, and let $\mathsf{sameobj}(a, b) \Leftrightarrow \mathsf{obj}(a) = \mathsf{obj}(b)$. For

an execution $X = (A, \mathsf{so}, \mathsf{vis}, \mathsf{ar})$, we define the ***per-object session order*** as follows: $\mathsf{soo} = (\mathsf{so} \cap \mathsf{sameobj})$.

The axioms RYW–MWA in Figure 1 formalise ***session guarantees***, ensuring that operations within a session observe a view of the database that is consistent with their own actions, even if, in an implementation, they can access various, potentially inconsistent replicas. The guarantees are due to Terry et al. [31], who defined them in a low-level operational framework. Here we recast them into axioms appropriate for arbitrary replicated data types. However, we have preserved the original terminology, and thus refer to reads and writes in the names of the axioms. As an illustration, RYW and MWV disallow the executions in Figures 2(c) and 2(d), respectively. Note also that WFRV implies that vis is transitive, and WFRA implies that $\mathsf{vis} \subseteq \mathsf{ar}$. In our **abstract implementation**, we implement session guarantees as proposed by Terry et al. [31]. For example, to ensure RYW, each session maintains the "write set" of actions it has issued; the session can then only connect to replicas that contain all of its write set.

The axioms for session guarantees highlight the general principle of formalising stronger consistency models: mandating that certain edges be included into vis and ar, so that clients have more up-to-date information. We now use the same approach to formalise other models.

Per-object causal consistency. Let $\mathsf{hbo} = (\mathsf{soo} \cup \mathsf{vis})^+$ be the ***per-object causality order***, which orders every action after those that can affect it via a chain of computation involving the same object; in other contexts, the name *happens-before* is used instead of causality. For example, in Figure 2(e), $\mathsf{add}(1)$ in the first session and rd in the third are related by hbo.

An execution is ***per-object causally consistent***, if it is eventually consistent and satisfies the axioms POCV and POCA in Figure 1. POCV guarantees that an operation sees all operations on the same object that causally affect it, and POCA correspondingly restricts the arbitration relation. POCV disallows the executions in Figures 2(c), 2(d) and 2(e). In fact, using our formalisation we can show that per-object causal consistency is equivalent to all of Terry et al.'s session guarantees.

THEOREM 6. POCV *is equivalent to the conjunction of* RYW*,* MR*,* WFRV *and* MWV. POCA *is equivalent to the conjunction of* WFRA *and* MWA.

(Cross-object) causal consistency. We define a consistency model that preserves a stronger notion of causality than per-object one, and is implemented, e.g., by COPS [24], Walter [30] and Concurrent Revisions [14]. Let the ***causality order*** $\mathsf{hb} = (\mathsf{so} \cup \mathsf{vis})^+$ be the transitive closure of the session and visibility relations. Unlike hbo, this relation considers any chain of computation, possibly involving multiple objects, to be a causal dependency. For example, in Figure 2(a) the write of `noboss` to x in the first session hb-precedes the read from x in the second; these actions are not related by hbo.

An execution is ***(cross-object) causally consistent***, if it is eventually consistent and satisfies the axioms COCV and COCA in Figure 1. These axioms are similar to those for per-object causal consistency, but without restrictions to causal dependencies via the same object (the use of acyclicity in COCA is explained below). For example, the execution in Figure 2(a), is allowed by per-object causal consistency, but not by cross-object causal consistency. Causal consistency is weaker than strong consistency, as it allows reading stale data—it is this feature that allows implementing it while guaranteeing availability and partition tolerance. For example, it allows the execution in Figure 2(f), where both reads fetch the initial value of the register, despite writes to it by the other session. It is easy to check that this outcome cannot be produced by any interleaving of the sessions' actions, and is thus not strongly consistent.

Abstract implementation of causal consistency. Our abstract implementation of per-object and cross-object causal consistency is based on the algorithm used in the COPS system [24]. When propagating actions between replicas, the algorithm tags every one of them with a list of other actions it causally depends on: those preceding it in hbo for per-object causal consistency, and in hb for cross-object one. A replica receiving an update performed by an action must wait until it receives all of the action's dependencies before making it available to clients. For the case of cross-object causal consistency, in Figure 2(a) the write of the photo by the first session will depend on the write of `noboss`. Hence, when the replica the second session is connected to receives the photo, it will have to wait until it receives the write of `noboss` before making the photo available to the client. The arbitration relation is computed using a system-wide Lamport clock [20]. This guarantees the acyclicity of $\mathsf{hb} \cup \mathsf{ar}$ and is the reason for not formulating COCA in the same way as POCA, i.e., $\mathsf{hb} \cap \mathsf{sameobj} \subseteq \mathsf{ar}$.

3.4 Comparison to Other Definitions

Quiescent consistency. We now show that our specifications of eventual consistency describe the semantics of the system more precisely than quiescent consistency, as stated by (1). To formalise the latter, assume that all operations are divided into queries (Query) and updates (Update), and that return values computed by $\mathcal{F}_\tau$ are insensitive to queries: $\mathcal{F}_\tau(f, V, \mathsf{vis}, \mathsf{ar}) = \mathcal{F}_\tau(f, V_u, \mathsf{vis}_u, \mathsf{ar}_u)$ where V_u, vis_u and ar_u are the restrictions of V, vis and ar to update operations. The following straightforward proposition shows that even the basic notion of eventual consistency in Figure 1 implies quiescent consistency.

PROPOSITION 7. *Consider an eventually consistent execution* $(A, \mathsf{so}, \mathsf{vis}, \mathsf{ar})$ *with finitely many update actions. Then there are only finitely many query actions* $a \in A$ *that do not see all updates on the object they are accessing:*

$$\neg(\forall b \in A.\, \mathsf{op}(b) \in \mathsf{Update} \wedge \mathsf{sameobj}(a, b) \implies b \xrightarrow{\mathsf{vis}} a).$$

Furthermore, all other query actions return the same value.

The converse is not true. Consider a program with a counter object x (initially zero), where the first session adds 1 to x, and the second continuously adds 2 to x until it is odd:

```
x.add(1)  ||  do { x.add(2) } while(x.read() % 2 == 0)
```

Under basic eventual consistency as defined above, termination is guaranteed: the axiom EVENTUAL guarantees that all but finitely many reads of x must see the $x.\mathsf{add}(1)$ operation, and thus the loop cannot repeat forever. However, under quiescent consistency termination is not guaranteed: since x is updated continuously, the system is not quiescent, and all bets are off.

Shared-memory consistency models. Interestingly, the specialisations of the consistency levels defined by the axioms in Figure 1 to the type intreg from §2 correspond almost exactly to those adopted by the memory model in the 2011 C and C++ standards [8]. Thus, POCA and POCV define the semantics of so-called *relaxed* operations in C/C++, which provide the weakest consistency. COCV and COCA are close to the semantics of *release-acquire* operations, providing consistency in between relaxed and strong. However, C/C++ does not validate THINAIR, since it makes the effects of processor or compiler speculation visible to the programmer. We formalise the correspondence to the C/C++ memory model in §C.

The similarity to C/C++ might come as a surprise. It stems from the fact that modern multiprocessors have a complicated microarchitecture, including a hierarchy of caches with coherence maintained via message passing—under the hood, a processor is really a distributed system. Optimisations that processors perform produce the same effects as delays and reorderings of messages occurring in a distributed system.

4. Combining Different Consistency Levels

Even though a given flavour of eventual consistency may be sufficient for many applications, stronger consistency levels are needed from time to time. For example, consider a shopping cart in an e-commerce application: while a user is shopping, it is acceptable for the information about the items in the shopping cart to be temporarily inconsistent; however, during a check-out, we need to be sure the user is paying only for what has actually been ordered. Therefore, Amazon's Dynamo [1] allows requesting strong consistency for some operations.

In a similar vein, causal consistency is desirable for many applications, but algorithms used to implement it (§3.3) increase the latency of propagating operations through the system. Bailis et al. [5] have argued that, given the huge size of causality graphs in real-world applications, this makes it problematic to provide causal consistency throughout the system and suggested letting the programmer request it on demand.

Due to the complexity of eventually consistent models, formulating their combinations precisely and choosing the programming interfaces for requesting stronger consistency are delicate. These issues have not been addressed by existing proposals for combining different models of eventual consistency [5]. Our contribution in this section is to show how, using our specification framework, we can define the semantics of such combinations and assess the trade-offs between different design choices. In particular, we present two mechanisms for requesting stronger consistency, widely used in shared-memory models [3, 8], and discuss their trade-offs in the context of eventually consistent systems. We first illustrate the techniques for combining consistency levels using the example of requesting cross-object causal consistency on demand in a system providing per-object causal consistency (§4.1). We then add on-demand strong consistency (§4.2). Combinations with weaker consistency levels could be handled similarly.

4.1 Consistency Annotations

Consider a system providing per-object causal consistency, i.e., satisfying the axioms SOWF–POCA in Figure 1. We wish to give the programmer the ability to request cross-object causal consistency on demand, as defined by COCV and COCA. One way to achieve this is to annotate every operation accepted by the database by a ***consistency annotation***, which classifies the operation as ordinary or causal. Accordingly, we change the form of actions from §3 to $a = (e, s, [x.f_\mu : k])$, where $\mu \in \{\mathsf{ORD}, \mathsf{CSL}\}$. We let $\mathsf{level}(a) = \mu$, and let the event selector ignore μ. The intention is that, when an action a annotated by CSL is visible to another action b, this establishes a causal relationship between a and b. Formally, consider an execution $X = (A, \mathsf{so}, \mathsf{vis}, \mathsf{ar})$. We define a ***causal visibility relation*** as follows:

$$\forall a, b.\, a \xrightarrow{\mathsf{cvis}} b \iff a \xrightarrow{\mathsf{vis}} b \wedge \mathsf{level}(a) = \mathsf{CSL}.$$

We then redefine the causality order as the transitive closure of the session and causal visibility relations: $\mathsf{hb} = (\mathsf{so} \cup \mathsf{cvis})^+$. Consistency annotations thus allow the programmer to specify which visibility relationships represent causal dependencies that the system has to preserve (we could similarly let the programmer treat only a subset of so as causal dependencies).

The combined model is given by all of the axioms in Figure 1, but with hb defined as above. Note that, since hbo is still defined as in Figure 1, per-object causal consistency is always guaranteed. Also, if every access is annotated as causal, then $\mathsf{cvis} = \mathsf{vis}$ and hb becomes defined as in Figure 1, so we come back to the causal consistency model from §3.3.

To illustrate the combined model, consider the execution in Figure 2(a), previously discussed in §3. To rule out this execution and make sure that a session that sees the photo will also see the updated access permission, we do not need to require all accesses to be causally consistent: only posting the photo has to be a causal operation. Indeed, in this case, the vis edge from the write to y in the first session to the read from it in the second will be included into cvis. Since the $\mathsf{so} \subseteq \mathsf{hb}$, the write of `noboss` to x in the first session will causally precede the read from x in the second. Then, by COCV, the read from x in the second session will see the write of `noboss` and fetch the correct permission. This is ensured even though the write to x is annotated as ordinary. Note that, if we performed additional ordinary operations in between the writes to x and y in the first session and then read them in the second, we would get the same guarantees.

Abstract implementation. To provide the combined consistency model in our abstract implementation, we modify the COPS algorithm (§3.3) to tag actions with their $(\mathsf{hb} \cup \mathsf{hbo})$-predecessors. Because of the tigher definition of hb, this decreases the number of dependencies in comparison to providing causal consistency throughout the system.

4.2 Fences

We now extend the the consistency model from §4.1 to allow the programmer to request strong consistency as well. Here we illustrate a different approach: instead of consistency annotations, we use ***fences***, which affect multiple actions instead of a single one. We extend the set of actions with $a = (e, s, \mathsf{fence})$, for which we let $\mathsf{op}(a) = \mathsf{fence}$.

In our **abstract implementation**, we treat a fence as a two-phase commit across all replicas, whereby the replica that ex-

ecutes it propagates all the updates it knows about to the other replicas, and does not proceed further until they acknowledge the receipt. Note that, as expected, this violates the availability requirement: if some replica becomes disconnected, the execution of a fence will have to wait until it reconnects.

We show the axioms for the consistency model with fences in Figure 3, where we also integrated the new axioms from §4.1. To define the semantics of fences, we adjust the notion of an execution to contain an additional relation sc satisfying the SCWF axiom: $X = (A, \mathsf{so}, \mathsf{vis}, \mathsf{ar}, \mathsf{sc})$. In implementation terms, sc can be viewed as the total order in which the two-phase commits initiated by fences take place. We modify VISWF and ARWF so that vis and ar relations did not relate fence actions, as they do not make sense for fences; we also assume that the sameobj relation from Figure 1 does not relate fence actions. We adjust THINAIR to take sc into account.

The role of fences is to provide additional guarantees about the visibility of certain actions, which we capture in our axioms two ways. First, we redefine the hb relation to include sc. It is then used by COCV and COCA, which allows taking sc into account when determining visibility and arbitration. To illustrate this, consider the execution in Figure 2(f), which is causally consistent, but not strongly consistent. If each session executes a fence after its write, then the outcome shown will be disallowed. This is because, by SCWF, the total order sc has to order the two fences one way or another; then by COCV, the write of the session whose fence comes first in sc is guaranteed to be seen by the read in the other session. Fences can thus be used to avoid anomalies where sessions read stale values. In implementation terms, we can justify including sc into hb as follows: if fences are implemented by two-phase commits, then a replica R_1 that executed a fence after a replica R_2 did is guaranteed to see all the updates that were visible to R_2 at the time it issued the fence.

Another way in which fences strengthen consistency is captured by a new clause into the definition of cvis: if an operation b hbo-follows an operation c in another session following a fence, then it causally depends on the fence. In implementation terms, this is justified as follows: $a \xrightarrow{\mathsf{so}} c \xrightarrow{\mathsf{hbo}} b$ guarantees that b executes after the two-phase commit triggered by the fence a has been completed. This two-phase commit propagates all updates known to the replica on which a was issued to all others, including the replica of b, which lets us include (a, b) into hb.

To illustrate this guarantee provided by fences, consider the execution in Figure 2(a) and assume that all writes are annotated as ordinary, but the first session issues a fence in between writing `noboss` and `photo`. In this case, by the definition of cvis and COCV, the second session will be guaranteed to see the correct access permissions. Thus, a fence subsumes the effect of annotating the write of the photo as causal. In fact, a fence has a much stronger effect. For example, if the first session posts several photos using ordinary write operations, then the fence would ensure that a session reading any of these photos will see the updated permissions. To achieve the same effect using consistency annotations, we would have to annotate all the writes of the photos as causal. This is the fundamental difference between fences and consistency annotations: the latter affect only a single operation, whereas the former affect many.

Figure 3. Axioms for a combination of per-object causal, cross-object causal and strong consistency. The definition of hbo and the axioms SOWF, RVAL, EVENTUAL, POCV, POCA, COCV, COCA are the same as in Figure 1, but with hb defined here.

VISWF: $\forall a, b.\, a \xrightarrow{\mathsf{vis}} b \implies \mathsf{obj}(a) = \mathsf{obj}(b)$ and vis does not have edges involving fence actions

ARWF: $\forall a, b.\, a \xrightarrow{\mathsf{ar}} b \implies \mathsf{obj}(a) = \mathsf{obj}(b)$, ar is transitive and irreflexive, does not have edges involving fence actions, and $\mathsf{ar}|_{\mathsf{vis}^{-1}(a)}$ is a total order for all $a \in A$

SCWF: sc is a total, transitive and irreflexive relation on fence actions

THINAIR: $\mathsf{so} \cup \mathsf{vis} \cup \mathsf{sc}$ is acyclic

Causal visibility relation:

$$\forall a, b.\, a \xrightarrow{\mathsf{cvis}} b \iff (a \xrightarrow{\mathsf{vis}} b \wedge \mathsf{level}(a) = \mathsf{CSL}) \vee (\exists c.\, \mathsf{op}(a) = \mathsf{fence} \wedge a \xrightarrow{\mathsf{so}} c \xrightarrow{\mathsf{hbo}} b)$$

Causality order: $\mathsf{hb} = (\mathsf{so} \cup \mathsf{cvis} \cup \mathsf{sc})^+$

As we show in §4.3, choosing one over the other to request stronger consistency impacts the implementation.

We now justify that fences enforce strong consistency: in a system with integer registers, putting a fence in between every pair of operations in a session guarantees strong consistency.

THEOREM 8. *Assume an execution* $(A, \mathsf{so}, \mathsf{vis}, \mathsf{ar}, \mathsf{sc})$ *with only* intreg *objects such that it satisfies the axioms in Figure 3 and* $\forall a, b \in A.\, \mathsf{op}(a) \neq \mathsf{fence} \wedge \mathsf{op}(b) \neq \mathsf{fence} \wedge a \xrightarrow{\mathsf{so}} b \implies \exists c.\, \mathsf{op}(c) = \mathsf{fence} \wedge a \xrightarrow{\mathsf{so}} c \xrightarrow{\mathsf{so}} b$. *Then there exists a total, transitive and irreflexive order* r *on all actions in* A *such that every read* r *from a register* x *fetches the value written by the last write to* x *preceding* r *in* r, *or* 0 *if there is no such write.*

4.3 Consistency Annotations vs Fences

In §4.1 and §4.2, we illustrated different mechanisms for requesting a stronger level of consistency: consistency annotations for causal consistency and fences for strong consistency. We now argue that the choice of the mechanisms impacts the implementation significantly.

Consider a per-object causally consistent system. Instead of using consistency annotations to request cross-object causal consistency, we could introduce a special kind of a fence, $\mathsf{fence}_{\mathsf{CSL}}$, and define cvis similarly to how it was done in §4.2:

$$\forall a, b.\, a \xrightarrow{\mathsf{cvis}} b \iff \exists c.\, \mathsf{op}(a) = \mathsf{fence}_{\mathsf{CSL}} \wedge a \xrightarrow{\mathsf{so}} c \xrightarrow{\mathsf{vis}} b.$$

Then a session that sees *any* operation c following a fence a is guaranteed to see all operations preceding the fence. To achieve this effect using consistency annotations, we would have to annotate every such operation c as causal. Hence, requesting causal consistency using fences makes it easier for the programmer to mark a series of operations as enforcing causal relationships. However, this can potentially affect the efficiency of the implementation. For the COPS algorithm (§3.3) to validate the above axiom, we would have to tag *every* action following a fence in a session with the list of actions visible to it at the time the fence was issued. A replica receiving such an action would then have to wait until all its dependencies are satisfied before making it available to clients. Unlike a consistency annotation, a fence thus increases the number of dependencies for all actions following it, with a potential impact on latency.

Such considerations need to be taken into account when designing programming interfaces for combined consistency models. A system may also provide both consistency annotations and fences (as done, e.g., in C/C++ [8]), to give the programmer maximal control at the expense of complicating the programming model. In all cases, our framework helps in exploring this design space by allowing a system developer to quickly establish the semantic consequences of various decisions and their impact on programmability.

5. Transactions

The semantics of transactions has been extensively studied in the context of databases, and various consistency models have been proposed for them, including ANSI SQL isolation levels, snapshot isolation [9] and serializability [26]. However, classical consistency models, such as the latter two, cannot be implemented while satisfying the requirements of availability and partition tolerance. For this reason, eventually consistent systems implement transactions with weaker guarantees [24, 30] that do not contradict these requirements. In this section, we show how they can be specified in our framework.

We illustrate our approach by extending the combined consistency model from §4 to accommodate transactions. The resulting semantics is similar to snapshot isolation [9], but without write-write conflict detection. Since we focus on the use of replicated data types in this paper, we do not need to disallow write-write conflicts: when an appropriate data type is used, its semantics automatically resolves conflicts, merging the two conflicting versions if needed, and the "lost update" anomaly does not occur. Hence, we do not have to consider the consequences of transactions aborting due to conflicts with others.

When a fence is used inside every transaction, our semantics coincides with serializability (Proposition 9 below). When specialised to causal consistency, it coincides with a variant of *parallel snapshot isolation*, implemented in Walter [30] and, for read-only transactions, in COPS [24]. We have not yet incorporated transactions into the single abstract implementation we use for the other features (§2–4). Hence, we justify the correspondence of our definitions to existing systems by a separate proof that their specialisation to causal consistency is equivalent to the abstract implementation of parallel snapshot isolation given by Sovran et al. [30]. The main idea of this implementation is to append writes performed by a transaction to the state of the replica it executes on atomically, and to propagate these writes to other replicas together. This ensures that transactions take effect atomically, but possibly with a delay. We formalise and prove the correspondence in §D.

To define the semantics of transactions, we assume that every action in an execution belongs to a transaction. Let us again extend the set of actions with one that separates different transactions: $a = (e, s, \mathsf{commit})$, for which we let $\mathsf{op}(a) = \mathsf{commit}$. We do not consider explicit abort operations. The changes to the axioms from Figure 3 needed to accommodate transactions are shown in Figure 4. We assume that vis, ar and sameobj relations do not relate commit actions.

Intuitively, if several actions are executed within a transaction, then the system should treat them as a single atomic action. Our key insight is that this can be captured by factoring the relations in an execution over an equivalence relation $\sim$, grouping actions in the same transaction. For a relation r, its factoring $\mathsf{r}/\sim$ includes the edges from r and those obtained from such edges by relating any other actions coming from the same transactions as their endpoints. The latter excludes the case when the endpoints themselves are from the same transaction. For vis, ar and sc we require that they be preserved under factoring (TRANSACT). We also include factoring explicitly into the definition of hb and COCA. Finally, ISOLATION ensures that uncommitted transactions are invisible to other sessions. In the following, we explain the semantics of transactions and justify the particular ways in which we use factoring over $\sim$ to define it by examples, summarised in Figure 5.

We start by showing that our notion of transactions provides basic isolation properties, ensured by factoring vis over $\sim$. Consider the execution in Figure 5(a), similar to the one in Figure 2(a), but where the user posts a photo and changes the access permission in a different order and within the same transaction. This execution is disallowed by the clause for vis in TRANSACT: even though both writes are ordinary, a session that sees the photo is guaranteed to see the updated permission. Thus, every session sees a transaction as happening either completely, or not at all. Furthermore, if the first transaction in the execution in Figure 5(a) also read the permission it wrote, it would be guaranteed to see its own write, since we assume per-object causality by default. If the second transaction read the photo twice, it would be guaranteed to see the same result due to the clause for vis in TRANSACT. Hence, transactions cannot read uncommitted data or observe a non-repeatable read.

The execution in Figure 5(b), similar to the one in Figure 2(f), illustrates the analogy with snapshot isolation by showing that our transactions allow the write skew anomaly typical for it. There, each transaction reads the initial value of a register, despite a write to it by the other transaction. This outcome would not be allowed if transactions were serializable. The execution in Figure 5(c) shows that, unlike classical snapshot isolation, our notion allows the lost update anomaly *for integer registers*. To eliminate this anomaly while satisfying the requirements of availability and partition tolerance, we can use an appropriate data type, as in the execution in Figure 5(d).

Finally, we illustrate how our notion of transactions interacts with different consistency levels in the underlying consistency model. If in the examples in Figures 2(a) and 2(f) every write or read were a transaction writing to or reading from multiple registers, then the anomalies shown in both examples would still be allowed: transactions enforce atomicity, but do not strengthen the causality guarantees among different transactions in comparison to that provided by the underlying non-transactional operations. However, ways of strengthening consistency guarantees presented §4 achieve the same effect with transactions. For example, if in the above transactional variant of the execution in Figure 2(a) any write in the photo-writing transaction were annotated as CSL, then the second session would be guaranteed to see the updated permissions.

We can also use fences from §4.2 to achieve serializability for transactions. Figure 5(e) illustrates this by implementing an operation that atomically tests if a register has reach a limit and, and if not, increments it, e.g., to make a reservation.

Figure 4. Changes to Figure 3 needed to accommodate transactions

"Same transaction" relation:
$a \sim b \iff \mathsf{op}(a) \neq \mathsf{commit} \wedge \mathsf{op}(b) \neq \mathsf{commit} \wedge$
$((a \xrightarrow{\mathsf{so}}{}^{*} b \wedge \neg\exists c.\, a \xrightarrow{\mathsf{so}} c \xrightarrow{\mathsf{so}} b \wedge \mathsf{op}(c) = \mathsf{commit}) \vee$
$(b \xrightarrow{\mathsf{so}}{}^{*} a \wedge \neg\exists c.\, b \xrightarrow{\mathsf{so}} c \xrightarrow{\mathsf{so}} a \wedge \mathsf{op}(c) = \mathsf{commit}))$

Factoring:
$\mathsf{r}/\sim = \mathsf{r} \cup \{(a,b) \mid a \not\sim b \wedge \exists a', b'.\, a \sim a' \wedge b \sim b' \wedge (a', b') \in \mathsf{r}\}$

TRANSACT: $(\mathsf{vis}/\sim) \cap \mathsf{sameobj} = \mathsf{vis}, \quad (\mathsf{ar}/\sim) \cap \mathsf{sameobj} = \mathsf{ar},$
$(\mathsf{sc}/\sim) \cap (\{a \mid \mathsf{op}(a) = \mathsf{fence}\})^2 = \mathsf{sc}$

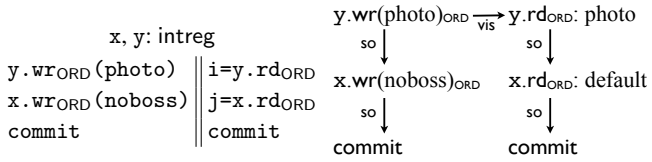
Causality order: $\mathsf{hb} = ((\mathsf{so} \cup \mathsf{cvis} \cup \mathsf{sc})/\sim)^{+}$

COCA. $(\mathsf{hb} \cup \mathsf{ar})/\sim$ is acyclic

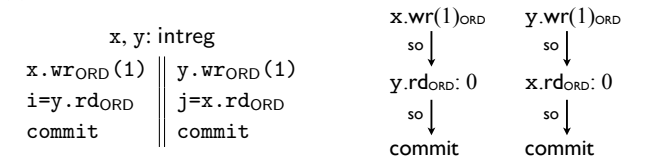
ISOLATION: $\forall a, b \in A.\, (a \xrightarrow{\mathsf{vis}} b \wedge \neg\exists c.\, a \xrightarrow{\mathsf{so}} c \wedge \mathsf{op}(c) = \mathsf{commit})$
$\implies \mathsf{ses}(a) = \mathsf{ses}(b)$

Figure 5. Anomalies allowed or disallowed by our transactions. We have omitted unimportant vis edges.

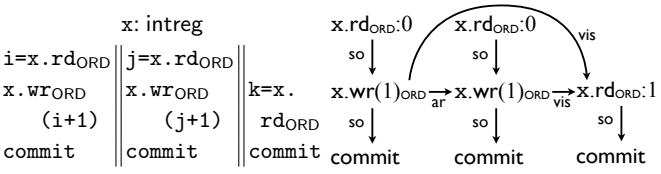
(a) Disallowed by TRANSACT:



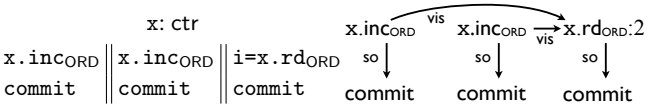
(b) Write skew is allowed:



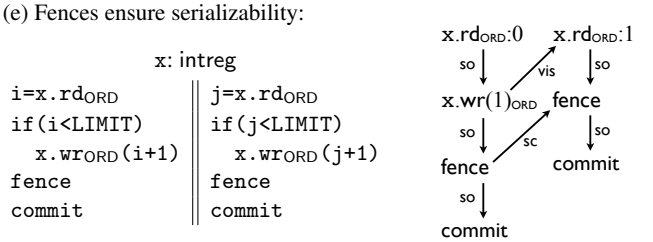
(c) Lost update can happen with integer registers:



(d) Lost update absent with an appropriate choice of data types:



(e) Fences ensure serializability:



In that execution, which is allowed by our axioms, we assume $\texttt{LIMIT} = 1$, and thus only one transaction can succeed in incrementing the register. This is ensured by the use of factoring in the definition of hb: the sc edge between the fences yields an hb edge from the write to x in the first transaction to the read from it in the second; by COCV this means that the second transaction is guaranteed to see the increment by the first one and will thus not perform increment itself. Note that we cannot guarantee this result when using the replicated counter data type: if in Figure 5(d) the first and the second transaction read the counter before incrementing it, they would both read 0. Making sure that only one transaction does an increment requires the use of fences, and correspondingly, giving up availability or partition tolerance. We formalise the guarantee provided by fences for transactions as follows.

PROPOSITION 9. *If an execution satisfies the consistency model in Figure 4, contains only* intreg *objects and has at least one fence inside every transaction, then it is serializable.*

6. Related Work

There exist several formal definitions of eventually consistent models, often proposed together with systems implementing them. In a nutshell, the difference with our work is that such specifications have so far been tied to particular data types or consistency levels, and were often very low-level.

For example, Shapiro et al. [10, 28, 29] described algorithms for a number of replicated data types on per-object causal consistency. As the correctness criterion, they consider quiescent consistency, which is less expressive than our specifications (§3.4). Bosneag and Brockmeyer [11] defined the consistency model of Bayou [32], also concentrating on a form of quiescent consistency. They handle Bayou's speculative operation execution, which we do not cover (§3.2). Fekete et al. [16] specified an eventually consistent model in an operational style, similar to our abstract implementation. Burckhardt et al. [14] defined the consistency model of the Concurrent Revisions system. Like us, they use axioms, but handle only causal consistency and data types with the semantics obtained from the generic construction using arbitration (§2).

Causal consistency was originally defined by Ahamad et al. [4]. However, their definition allows different replicas to have different, though causally consistent, views on the system evolution and thus diverge forever. Recently, Mahajan et al. [25] and Lloyd et al. [24] defined a stronger version of causal consistency that ensures convergence, which we use in this paper. Mahajan et al. define convergence in an operational model, and Lloyd et al. using explicit conflict handling functions. We give a declarative specification, with conflict handling encapsulated in a replicated data type.

Sovran et al. [30] defined a consistency model for transactions called parallel snapshot isolation, which they implemented in the Walter system. Our semantics of transactions for the case of causal consistency is equivalent to the variant of parallel snapshot isolation that Sovran et al. define for a *counting set* replicated data type (§5). In contrast to ours, their definition is given in a low-level operational style.

There exist a number of specifications of shared-memory models weaker than strong consistency [3], including those combining several consistency levels (e.g., [8]). All such models assume read-write memory cells, corresponding to our intreg data type. We handle arbitrary replicated data types and, as a consequence, our notion of executions is somewhat different from the one used to define shared-memory models (§3.1).

There have been some proposals for combining multiple consistency levels within geo-replicated databases. Li et al. [21] proposed *red-blue consistency*, similar to our on-

demand strong consistency (§4.2). In contrast to us, they assume a particular strategy of resolving conflicts using commutativity (§2). Bailis et al. [5] sketched an interface that allows a programmer to specify causality explicitly. Our formalisation of on-demand causal consistency in §4.1 and §4.3 complements their proposal with a formal semantics and a discussion of the trade-offs between different ways of specifying causality.

7. Conclusion

We have presented a flexible specification framework for eventually consistent systems that incorporates and unifies the guarantees and features that have appeared in a wide array of previous work [2, 13, 14, 17, 21, 24, 29–31]. In particular, our framework supports replicated data types and conflict resolution, session guarantees, various causality guarantees, consistency annotations, fences, and transactions.

We have illustrated how our specifications allow programmers to determine if a certain behavior is possible without considering the details of a system's implementation. Moreover, we have shown how our framework enables precise statements and proofs about how various features and guarantees are related, thus providing system architects with a tool for exploring the design space.

Finally, we hope that our use of shared-memory model techniques will help to bridge research communities and promote an exchange of ideas and results. For instance, we plan to apply several techniques developed for shared-memory models to eventually consistent systems, such as the testing and verification of programs [12], automatic inference of consistency annotations and fences [23] and compositional reasoning about components [6].

References

[1] Amazon DynamoDB. http://aws.amazon.com/dynamodb/.

[2] Basho Riak. http://basho.com/products/riak-overview/.

[3] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12), 1996.

[4] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9, 1995.

[5] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. The potential dangers of causal consistency and an explicit solution (vision paper). In *SOCC*, 20012.

[6] M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In *POPL*, 2013. To appear.

[7] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In *POPL*, 2012.

[8] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, 2011.

[9] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, 1995.

[10] A. Bieniusa, M. Zawirski, N. M. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. Brief announcement: Semantics of eventually consistent replicated sets. In *DISC*, 2012.

[11] A.-M. Bosneag and M. Brockmeyer. A formal model for eventual consistency semantics. In *IASTED PDCS*, 2002.

[12] S. Burckhardt, R. Alur, and M. M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *PLDI*, 2007.

[13] S. Burckhardt, M. Fähndrich, D. Leijen, and B. P. Wood. Cloud types for eventual consistency. In *ECOOP*, 2012.

[14] S. Burckhardt, D. Leijen, M. Fähndrich, and M. Sagiv. Eventually consistent transactions. In *ESOP*, 2012.

[15] N. Conway, R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *SOCC*, 2012.

[16] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-serializable data services. In *PODC*, 1996.

[17] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[18] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2), 2002.

[19] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. 2008.

[20] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7), 1978.

[21] C. Li, D. Porto, A. Clement, R. Rodrigues, N. Preguiça, and J. Gehrke. Making geo-replicated systems fast if possible, consistent when necessary. In *OSDI*, 2012.

[22] B. Liskov and S. Zilles. Programming with abstract data types. In *ACM Symposium on Very High Level Languages*, 1974.

[23] F. Liu, N. Nedev, N. Prisadnikov, M. T. Vechev, and E. Yahav. Dynamic synthesis for relaxed memory models. In *PLDI*, 2012.

[24] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *SOSP*, 2011.

[25] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, and convergence. Technical Report TR-11-22, UT Austin, 2011.

[26] C. Papadimitriou. *The theory of database concurrency control*. 1986.

[27] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel Distrib. Comput.*, 71(3), 2011.

[28] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report 7506, INRIA, 2011.

[29] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *SSS*, 2011.

[30] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, 2011.

[31] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. Session guarantees for weakly consistent replicated data. In *PDIS*, 1994.

[32] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, 1995.

A. Proofs

A.1 Proof of Theorem 6

POCV is equivalent to the conjunction of RYW, MR, WFRV and MWV. Pick an excution $X = (A, \mathsf{so}, \mathsf{vis}, \mathsf{ar})$. Let soo and hbo be, respectively, the per-object session order and the per-object causality order, both induced by X. Consider binary relations r on actions in A that access the same object, i.e.,

$$r \subseteq \{(a,b) \in A \times A \mid \mathsf{sameobj}(a,b)\}. \tag{2}$$

We define two operators H, G on such relations as follows:

$$H(r) = \mathsf{soo} \cup (r; \mathsf{soo}) \cup (r; \mathsf{soo}^*; r) \cup (\mathsf{soo}; r);$$
$$G(r) = (r \cup \mathsf{soo})^+.$$

We will prove that for every r satisfying (2) we have

$$H(r) \subseteq r \iff G(r) \subseteq r.$$

This gives us the required, because POCV is equivalent to $G(\mathsf{vis}) \subseteq \mathsf{vis}$, and the conjunction of RYW, MR, WFRV and MWV to $H(\mathsf{vis}) \subseteq \mathsf{vis}$.

Since $H(r) \subseteq G(r)$ for every r satisfying (2), we have that $G(r) \subseteq r \implies H(r) \subseteq r$. For the other direction, assume that $H(r) \subseteq r$. We will now show that $G(r) \subseteq r$. Pick $a, a' \in A$ such that $(a, a') \in G(r)$. We have to prove that $(a, a') \in r$. By the definition of G and the transitivity of r and soo, we have the following cases of (a, a'), which we handle separately in this proof.

If $(a, a') \in \mathsf{soo}$, then $(a, a') \in H(r)$, because $\mathsf{soo} \subseteq H(r)$. But $H(r) \subseteq r$ by our choice of r. Hence, $(a, a') \in r$, as desired.

If $(a, a') \notin \mathsf{soo}$, then there exist $a_1, b_1, \ldots, a_n, b_n$ with $n \geq 1$ such that

$$\begin{aligned}(a, a_1) \in \mathsf{soo}^* \wedge (b_n, a') \in \mathsf{soo}^* \\ \wedge (\forall i \in \{1, \ldots, n\}.\, (a_i, b_i) \in r) \\ \wedge (\forall i \in \{1, \ldots, n-1\}.\, (b_i, a_{i+1}) \in \mathsf{soo}).\end{aligned}$$

Using our assumption that $H(r) \subseteq r$, we can prove the desired $(a, a') \in r$ as follows:

$$\begin{aligned}&(a, a_1) \in \mathsf{soo}^* \wedge (a_1, b_1) \in r \wedge (b_1, a_2) \in \mathsf{soo} \wedge \ldots \\ &\qquad\qquad (a_n, b_n) \in r \wedge (b_n, a') \in \mathsf{soo}^* \\ &\implies (a, a_1) \in \mathsf{soo}^* \wedge (a_1, b_n) \in r \wedge (b_n, a') \in \mathsf{soo}^* \\ &\implies (a, a_1) \in \mathsf{soo}^* \wedge (a_1, a') \in r \\ &\implies (a, a') \in r.\end{aligned}$$

The first implication comes from the fact that $(r; \mathsf{soo}^*; r) \subseteq H(r)$, the second from $(r; \mathsf{soo}) \subseteq H(r)$, and the third from $(\mathsf{soo}; r) \subseteq H(r)$.

POCA is equivalent to the conjunction of WFRA and MWA. Pick an execution $X = (A, \mathsf{so}, \mathsf{vis}, \mathsf{ar})$. Let soo and hbo be, respectively, the per-object session order and the per-object causality order, both induced by X. Then,

$$\mathsf{hbo} = (\mathsf{soo} \cup \mathsf{vis})^+ = (\mathsf{soo} \cup (\mathsf{vis}; \mathsf{soo}^*))^+.$$

Thus, the lower bound on ar set by WFRA and MWA is included in that given by POCA. This means that POCA implies WFRA and MWA. It remains to prove that WFRA and MWA together imply POCA. Consider $(a, b) \in \mathsf{hbo}$. By the definition of hbo, there exist $a_1, \ldots, a_n$ with $n \geq 2$ such that

$$\begin{aligned}a_1 = a \wedge a_n = b \wedge \\ \forall i \in \{1, \ldots, n-1\}.\, (a_i, a_{i+1}) \in (\mathsf{soo} \cup \mathsf{vis}).\end{aligned}$$

Since WFRA and MWA hold, the third conjunct above implies that

$$\forall i \in \{1, \ldots, n-1\}.\, (a_i, a_{i+1}) \in \mathsf{ar}.$$

Furthermore, ar is transitive. Hence, we have $(a, b) \in \mathsf{ar}$, as desired. □

A.2 Proof of Theorem 8

In the following r ranges over read actions, w over write actions and f over fence actions. Consider an execution $X = (A, \mathsf{so}, \mathsf{vis}, \mathsf{ar}, \mathsf{sc})$ satisfying the assumptions of the theorem. For simplicity, we consider only the case when every read in the execution reads a non-default value. Let

$$\begin{aligned}\mathsf{vis}' &= \{(w, r) \in \mathsf{vis} \mid \mathsf{op}(w) = \mathsf{wr} \wedge \mathsf{op}(r) = \mathsf{rd}\}; \\ \mathsf{ar}' &= \{(w_1, w_2) \in \mathsf{ar} \mid \mathsf{op}(w_1) = \mathsf{wr} \wedge \mathsf{op}(w_2) = \mathsf{wr}\}.\end{aligned}$$

Let us show that $\mathsf{ar}' \cup \mathsf{vis}' \cup \mathsf{so} \cup \mathsf{sc}$ is acyclic. Assume there is a cycle in this relation. By the assumptions of the theorem, we can assume that any so edge on the cycle has a fence as one of its endpoints. Then there is at least one fence on the cycle, since ar' and vis' cannot form one due to the types of their endpoints. Consider any segment of the cycle starting an ending with a fence that has no fences in the middle. If this segment has edges other than so and sc, then it has to have one of the following forms:

- $f_1 \xrightarrow{\mathsf{so}} w_1 \xrightarrow{\mathsf{ar}'} w_2 \xrightarrow{\mathsf{vis}'} r \xrightarrow{\mathsf{so}} f_2$;
- $f_1 \xrightarrow{\mathsf{so}} w_1 \xrightarrow{\mathsf{ar}'} w_2 \xrightarrow{\mathsf{so}} f_2$;
- $f_1 \xrightarrow{\mathsf{so}} w_1 \xrightarrow{\mathsf{so}} f_2$.

Assume $(f_2, f_1) \in \mathsf{sc}$. Then the second configuration contradicts COCA, and the last one THINAIR. By POCV and the transitivity of vis, the first configuration yields $(w_2, w_1) \in \mathsf{vis}$, which contradicts POCA. Hence $(f_1, f_2) \in \mathsf{sc}$. We can thus convert the cycle into one in $\mathsf{so} \cup \mathsf{sc}$, contradicting THINAIR.

Let ar'' be any relation that is total on write actions to the same object and contains $\mathsf{ar}' \cup \mathsf{vis}' \cup \mathsf{so} \cup \mathsf{sc}$. Then $\mathsf{ar}'' \cup \mathsf{vis}' \cup \mathsf{so} \cup \mathsf{sc}$ is acyclic. Let

$$\begin{aligned}\mathsf{fr} = \{(r, w) \mid \exists w'.\, (w', r) \in \mathsf{vis} \wedge (w', w) \in \mathsf{ar} \wedge \\ \neg \exists w''.\, (w'', r) \in \mathsf{vis} \wedge (w', w'') \in \mathsf{ar}\}.\end{aligned}$$

We now show that

$$\mathsf{ar}'' \cup \mathsf{vis}' \cup \mathsf{so} \cup \mathsf{sc} \cup \mathsf{fr} \tag{3}$$

is acyclic. Assume the contrary. Then the cycle contains at least one fr edge.

Consider first the case when the cycle does not contain so or sc edges. We can assume that the cycle does not have any ar

edges, as they can only follow fr edges and can be merged with them.

Let us show that $\mathsf{fr};\mathsf{vis}';\mathsf{fr} \subseteq \mathsf{fr}$. Take $(r_1, w_4) \in \mathsf{fr};\mathsf{vis}';\mathsf{fr}$, then for some w_1, w_2, r_2, w_3 we have

$$r_1 \xleftarrow{\mathsf{vis}'} w_1 \xrightarrow{\mathsf{ar}''} w_2 \xrightarrow{\mathsf{vis}'} r_2 \xleftarrow{\mathsf{vis}'} w_3 \xrightarrow{\mathsf{ar}''} w_4$$

and $(w_2, w_3) \in \mathsf{ar}$, $\neg\exists w'.\, (w', r_1) \in \mathsf{vis}' \wedge (w_1, w') \in \mathsf{ar}$. Then $(w_1, w_4) \in \mathsf{ar}$ and hence, $(r_1, w_4) \in \mathsf{fr}$.

Thus, we can assume that the cycle has only a single fr edge. Then it also has a single vis' edge. Hence, for some r, w we have $(r, w) \in \mathsf{fr}$ and $(w, r) \in \mathsf{vis}'$. But the former implies that for some w' we have $(w', r) \in \mathsf{vis}'$, $(w', w) \in \mathsf{ar}$ and $\neg\exists w''.\, (w'', r) \in \mathsf{vis}' \wedge (w', w'') \in \mathsf{ar}$, which is a contradiction.

Assume now that the cycle in (3) has at least one so or sc edge. Consider any segment of the cycle starting and ending by a fence that does not contain fences in the middle. Then the segment does not contain any so or sc edges, except the first and the last one. Any ar edges except possibly the second one on the segment can only follow fr edges and can be merged with them. Since $\mathsf{fr};\mathsf{vis}';\mathsf{fr} \subseteq \mathsf{fr}$, we can thus assume that the segment has a single fr edge. Hence, the segment can only be of one of the following forms:

1. $f_1 \xrightarrow{\mathsf{so}} w_1 \xrightarrow{(\mathsf{ar}'')^*} w_2 \xrightarrow{\mathsf{vis}'} r_1 \xrightarrow{\mathsf{fr}} w_3 \xrightarrow{\mathsf{vis}'} r_2 \xrightarrow{\mathsf{so}} f_2$;
2. $f_1 \xrightarrow{\mathsf{so}} w_1 \xrightarrow{(\mathsf{ar}'')^*} w_2 \xrightarrow{\mathsf{vis}'} r_1 \xrightarrow{\mathsf{fr}} w_3 \xrightarrow{\mathsf{so}} f_2$;
3. $f_1 \xrightarrow{\mathsf{so}} r_1 \xrightarrow{\mathsf{fr}} w_3 \xrightarrow{\mathsf{vis}'} r_2 \xrightarrow{\mathsf{so}} f_2$;
4. $f_1 \xrightarrow{\mathsf{so}} r_1 \xrightarrow{\mathsf{fr}} w_3 \xrightarrow{\mathsf{so}} f_2$;
5. $f_1 \xrightarrow{\mathsf{so}} f_2$;
6. $f_1 \xrightarrow{\mathsf{sc}} f_2$;
7. $f_1 \xrightarrow{\mathsf{so}} w_1 \xrightarrow{(\mathsf{ar}'')^*} w_2 \xrightarrow{\mathsf{so}} f_2$;
8. $f_1 \xrightarrow{\mathsf{so}} w_1 \xrightarrow{(\mathsf{ar}'')^*} w_2 \xrightarrow{\mathsf{vis}'} r_1 \xrightarrow{\mathsf{so}} f_2$.

We now show that in all cases we must have $(f_1, f_2) \in \mathsf{sc}$. Note that this means that we can convert the cycle into one in $\mathsf{so} \cup \mathsf{sc}$, contradicting THINAIR and thus implying the acyclicity of (3).

Assume the contrary, i.e., $(f_2, f_1) \in \mathsf{sc}$. Then cases 5 and 6 contradict THINAIR, and case 7 contradicts COCA. By POCV and the transitivity of vis, case 8 contradict POCA. By the definition of hbo, in cases 1 and 2 the segment has the form

$$f_1 \xrightarrow{\mathsf{so}} w_1 \xrightarrow{(\mathsf{ar}'')^*} w_2 \xrightarrow{\mathsf{vis}'} r_1 \xleftarrow{\mathsf{vis}'} w_4 \xrightarrow{\mathsf{ar}''} w_3 \xrightarrow{(\mathsf{vis}';\mathsf{so})\cup\mathsf{so}} f_2$$

and $(w_2, w_4) \in \mathsf{ar}''$. Hence,

$$f_1 \xrightarrow{\mathsf{so}} w_1 \xrightarrow{\mathsf{ar}''} w_3 \xrightarrow{(\mathsf{vis}';\mathsf{so})\cup\mathsf{so}} f_2$$

By POCV, COCV and the transitivity of vis, $(f_2, f_1) \in \mathsf{sc}$ entails $(w_3, w_1) \in \mathsf{vis} \subseteq \mathsf{hbo}$, which contradicts POCA.

In cases 3 and 4 the segment has the form

$$f_1 \xrightarrow{\mathsf{so}} r_1 \xleftarrow{\mathsf{vis}'} w_4 \xrightarrow{\mathsf{ar}''} w_3 \xrightarrow{(\mathsf{vis}';\mathsf{so})\cup\mathsf{so}} f_2$$

and $\neg\exists w'.\, (r_1, w') \in \mathsf{vis}' \wedge (w_4, w') \in \mathsf{ar}''$. By POCV, COCV and the transitivity of vis, $(f_2, f_1) \in \mathsf{sc}$ entails $(w_3, r_1) \in \mathsf{vis}'$, which yields a contradiction.

Hence, (3) is acyclic. Let us take any total relation including (3) as the desired r. Assume that for some $r, w, w' \in A$ such that $\mathsf{obj}(w) = \mathsf{obj}(w')$ we have $(w, r) \in \mathsf{vis}$ and $(w, w') \in \mathsf{r}$. Then $(w, r) \in \mathsf{r}$, $(w, w') \in \mathsf{ar}$, and hence, $(r, w') \in \mathsf{fr}$. From this it follows that $(r, w') \in \mathsf{r}$. Thus, every read reads the most recent value according to r. □

A.3 Proof of Proposition 9

Consider an execution $X = (A, \mathsf{so}, \mathsf{vis}, \mathsf{ar}, \mathsf{sc})$. Due to the use of factoring in the definition of hb, from the assumption of the theorem we have

$$\begin{aligned}&\forall a, b \in A.\, a \not\sim b \implies\\ &\quad((\forall a', b' \in A.\, a \sim a' \wedge b \sim b' \implies (a', b') \in \mathsf{hb}) \vee\\ &\qquad(\forall a', b' \in A.\, a \sim a' \wedge b \sim b' \implies (b', a') \in \mathsf{hb})).\end{aligned}$$

Hence, hb is total on A and every transaction is contiguous in it, which for the case of integer registers implies serializability.

B. Abstract Implementation and Correspondence Theorem

THEOREM 10. *Our abstract implementation generates executions satisfying axioms in the main text of the paper, when its parameters satisfy appropriate conditions. These conditions are described in Propositions 21, 22, 23, 24, 25, 26, 27 and 28.*

In this section, we describe an abstract implementation of an eventually-consistent database system, and show that it corresponds to the axiomatic specification given in the main text of the paper. Our implementation is highly parameterised. When instantiated with appropriate parameters, it meets different levels of eventual consistency, which we have described axiomatically in the paper. The implementation is defined in terms of rules for transforming sessions and replicas. One of such rules is concerned with the interaction between a session and a replica, and it supports the availability of a replica to a session in the sense that the rule does not involve any communication among replicas.

Throughout this section, we assume a finite totally-ordered set of replica ids:

$$(\mathsf{DId}, <), \text{ ranged over by } d.$$

Also, we call a relation r is ***functional*** if

$$\begin{aligned}&\forall x, y_1, \ldots, y_n, y_1', \ldots, y_n'.\\ &\quad(x, y_1, \ldots, y_n) \in r \wedge (x, y_1', \ldots, y_n') \in r\\ &\qquad\implies y_1 = y_1' \wedge \ldots \wedge y_n = y_n'.\end{aligned}$$

Finally, for any set X, we let $\mathcal{P}_{\mathsf{fin}}(X)$ be the collection of all finite subsets of X.

B.1 Three Main Components

We start with three main components of our implementation: distributed time, message and data type implementation.

A ***distributed time*** is a totally-ordered set of the form

$$\mathsf{LTime} \times \mathsf{DId}.$$

Here LTime is a countably-infinite totally-ordered set with a function $\mathsf{next} : \mathsf{LTime} \to \mathsf{LTime}$ satisfying

$$\forall t, t' \in \mathsf{LTime}.\ \mathsf{next}(t) = t' \implies t < t'$$

and $(\mathsf{LTime} \times \mathsf{DId})$ is ordered lexicographically. We call elements in $(\mathsf{LTime} \times \mathsf{DId})$ ***distributed timestamps***.

Our abstract implementation assumes that the set of action ids is defined by a distributed time:

$$\mathsf{AId} = \mathsf{LTime} \times \mathsf{DId}.$$

A ***message*** is a tuple $m = (x, e, f, W, E)$ in the following set:

$$\mathsf{Obj} \times \mathsf{AId} \times (\bigcup_\tau \mathsf{Op}_\tau) \times \mathcal{P}_{\mathsf{fin}}(\mathsf{AId}) \times \mathcal{P}_{\mathsf{fin}}(\mathsf{AId}).$$

In our intended usage, a message (x, e, f, W, E) represents an operation f performed on the object x at the timestamp e. When this operation was originally issued in a replica, the message says, the actions with ids in W were visible in the replica. When the operation is later propagated to another remote replica, it will stay in the target replica without being executed until all the actions with ids in E are performed on the replica. We will write Msg for the set of messages.

A ***data type implementation*** is a family of tuples

$$\begin{aligned}(&\mathsf{St}_\tau,\ \mathsf{init}_\tau : \mathsf{St}_\tau,\\ &\quad \mathsf{eval}_\tau : \mathsf{AId} \times \mathsf{Op}_\tau \times \mathcal{P}_{\mathsf{fin}}(\mathsf{AId}) \times \mathsf{St}_\tau \to \mathsf{Val} \times \mathsf{St}_\tau,\\ &\quad \mathsf{id}_\tau : \mathsf{St}_\tau \to \mathcal{P}_{\mathsf{fin}}(\mathsf{AId}))\end{aligned}$$

indexed by each data type τ, such that for every τ,

$$\begin{aligned}&(\mathsf{id}_\tau(\mathsf{init}_\tau) = \emptyset) \wedge\\ &(\forall e, f, W.\, \forall \sigma, \sigma' \in \mathsf{St}_\tau.\\ &\quad \mathsf{eval}_\tau(e, f, W, \sigma) = (\_, \sigma') \implies \mathsf{id}_\tau(\sigma') = \mathsf{id}_\tau(\sigma) \cup \{e\}).\end{aligned}$$

The init_τ is the initial state of a data type τ, the next $\mathsf{eval}_\tau(e, f, W, \sigma)$ describes the outcome of running the operation f on the state σ, where the id of the associated action is e and this action is issued when actions in W are visible from the replica. The last $\mathsf{id}_\tau(\sigma)$ returns the ids of all the actions that have influenced on the computation leading to σ.

B.2 Database Implementation

Our implementation of a distributed database assumes the three components that we have just described. It defines the information stored in sessions and replicas, and specifies a protocol that these replicas and sessions need to follow in order to guarantee a desired level of eventual consistency. In the following, we describe the implementation step-by-step:

1. Firstly, we assume a distributed time, a next operator and a data type implementation:

   $$\begin{aligned}&\mathsf{LTime} \times \mathsf{DId}, \qquad \mathsf{next} : \mathsf{LTime} \to \mathsf{LTime},\\ &\{(\mathsf{St}_\tau, \mathsf{init}_\tau, \mathsf{eval}_\tau, \mathsf{id}_\tau)\}_\tau.\end{aligned}$$

2. Secondly, we define configurations for replicas, sessions and entire systems.

   - The set of replica configurations, DB, is given by

     $$\begin{aligned}&\mathsf{Table} = \mathsf{Obj} \to \bigcup_\tau \mathsf{St}_\tau,\\ &\mathsf{DB} = \mathsf{DId} \times \mathsf{LTime} \times \mathsf{Table} \times \mathcal{P}_{\mathsf{fin}}(\mathsf{Msg}) \times \mathcal{P}_{\mathsf{fin}}(\mathsf{Msg}).\end{aligned}$$

     A tuple $(d, t, \rho, M, N) \in \mathsf{DB}$ represents a status of a replica. The first component d is the id of the replica, and (t, d) is the timestamp that the replica maintains for itself. The next ρ is a table for storing objects. The following M consists of messages about actions that were performed on the replica and need to be propagated to other remote replicas. The last component N contains the other kind of messages, those that describe operations performed in other remote replicas and are propagated from them to the current replica.

   - The set of session configurations, $\mathsf{Session}$, is defined as follows:

     $$\mathsf{Session} = \mathsf{SId} \times \mathsf{SCtx},\ \text{ ranged over by } \theta \text{ or } (s, \kappa),$$

     Here SCtx is an unspecified set and it contains session contexts, which store session-specific information. We do not fix what goes into a session context. Various choices of SCtx will emerge as we later consider axioms of eventual consistency.

   - A system configuration is a pair

     $$\mathcal{S} \mid \mathcal{D}$$

     where $\mathcal{S}$ is a subset of $\mathsf{Session}$ and $\mathcal{D}$ is a subset of DB. We require that both $\mathcal{S}$ and $\mathcal{D}$ be functional, and they meet the following condition:

     $$\begin{aligned}\forall e \in \mathsf{id}(\mathcal{S}|\mathcal{D}).\, \exists t.\, \exists (d, t', \_, \_, \_) \in \mathcal{D}.\\ e = (t, d) \wedge e < (t', d),\end{aligned}$$

     where the set $\mathsf{id}(\mathcal{S}|\mathcal{D})$ consists of all action ids appearing in $\mathcal{S}|\mathcal{D}$ and is formally defined by:

     $$\begin{aligned}&\mathsf{id}(\mathcal{S}|\mathcal{D}) =\\ &\{e \mid \exists (\_, \_, \rho, M, N) \in \mathcal{D}.\\ &\quad (\exists x \in \mathsf{Obj}.\, e \in \mathsf{id}_{\mathsf{type}(x)}(\rho(x)))\\ &\quad \vee \exists (\_, e', \_, W, E) \in M \cup N.\, e \in W \cup E \cup \{e'\}\}.\end{aligned}$$

     This condition means that according to the distributed time, every action in $\mathcal{D}$ are performed in the past, if we use timestamps of replicas as our baseline.

     We use comma to mean the disjoint union, and represent a singleton set $\{x\}$ simply by its element x. For instance, $(\mathcal{S}, \theta)$ means the union of $\mathcal{S}$ and the singleton set $\{\theta\}$ where θ does not belong to $\mathcal{S}$. Also, we use the following operation on system configurations:

     $$\mathsf{time}(\mathcal{S}|\mathcal{D}) = \{(t, d) \mid (d, t, \_, \_, \_) \in \mathcal{D}\}.$$

3. Thirdly, we assume the presence of the following operations

for all data types τ:

$$\begin{aligned}
\mathsf{instr} &: \mathsf{Session} \to \mathsf{Obj} \times \mathsf{Op} \\
\mathsf{update}_\tau &: \mathsf{Obj}_\tau \times \mathsf{AId} \times \mathsf{Op}_\tau \times \mathsf{St}_\tau \times \mathsf{Table} \times \mathsf{SCtx} \to \mathsf{SCtx} \\
\mathsf{depend}_\tau &: \mathsf{AId} \times \mathsf{Op}_\tau \times \mathsf{St}_\tau \times \mathsf{Table} \times \mathsf{SCtx} \to \mathcal{P}_{\mathsf{fin}}(\mathsf{AId}) \\
\mathsf{enable} &: \mathsf{Obj} \times \mathsf{Session} \times \mathsf{DB} \to \{\mathsf{true}, \mathsf{false}\}
\end{aligned}$$

where $\mathsf{Obj}_\tau = \{x \in \mathsf{Obj} \mid \mathsf{type}(x) = \tau\}$ and the depend operator is required to satisfy the condition below:

$$\mathsf{depend}_\tau(e, f, \sigma, \rho, \kappa) \subseteq (\mathsf{id}_\tau(\sigma) \cup \bigcup_{x \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(x)}(\rho(x))).$$

The first operation selects the next instruction to be executed by a session, and the second describes how running this instruction changes the session's status. The third operation takes the id of an action, and computes the ids of other actions that should be performed on a replica before the given action. The last is a predicate that checks whether a session can access a replica without violating a desired level of eventual consistency.

The last three operations are the knobs on our abstract implementation, which can be adjusted to achieve a different level of eventual consistency. The details will be given in the rest of this section.

4. Finally, we define our implementation in terms of rules for transforming system configurations:

$$\frac{\begin{array}{l}\delta_i = (d_i, t_i, \rho_i, M_i, N_i) \quad m = (x, e, f, W, E) \in M_1 \\ e \notin \mathsf{id}_{\mathsf{type}(x)}(\rho_2(x)) \\ \delta_2' = (d_2, \mathsf{max}(\mathsf{next}(t_1), t_2), \rho_2, M_2, N_2 \cup \{m\})\end{array}}{\mathcal{S} \mid \mathcal{D}, \delta_1, \delta_2 \xrightarrow{m} \mathcal{S} \mid \mathcal{D}, \delta_1, \delta_2'} \textsc{Prop}$$

$$\frac{\begin{array}{l}\delta = (d, t, \rho, M, N) \qquad (x, e, f, W, E) \in N \\ \tau = \mathsf{type}(x) \qquad E \subseteq \bigcup_{y \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(y)}(\rho(y)) \\ e \notin \mathsf{id}_\tau(\rho(x)) \qquad (\_, \sigma') = \mathsf{eval}_\tau(e, f, W, \rho(x)) \\ \delta' = (d, t, \rho[x \mapsto \sigma'], M, N)\end{array}}{\mathcal{S} \mid \mathcal{D}, \delta \xrightarrow{()} \mathcal{S} \mid \mathcal{D}, \delta'} \textsc{OpD}$$

$$\frac{\begin{array}{l}\theta = (s, \kappa) \qquad \delta = (d, t, \rho, M, N) \qquad e = (t, d) \\ \mathsf{instr}(\theta) = (x, f) \quad \tau = \mathsf{type}(x) \quad \mathsf{enable}(x, \theta, \delta) \\ \kappa' = \mathsf{update}_\tau(x, e, f, \rho(x), \rho, \kappa) \qquad \theta' = (s, \kappa') \\ W' = \mathsf{id}_\tau(\rho(x)) \qquad E' = \mathsf{depend}_\tau(e, f, \rho(x), \rho, \kappa) \\ \mathsf{eval}_\tau(e, f, W', \rho(x)) = (w, \sigma') \quad a = (e, s, [x.f : w]) \\ \delta' = (d, \mathsf{next}(t), \rho[x \mapsto \sigma'], M \cup \{(x, e, f, W', E')\}, N)\end{array}}{\mathcal{S}, \theta \mid \mathcal{D}, \delta \xrightarrow{(W', a)} \mathcal{S}, \theta' \mid \mathcal{D}, \delta'} \textsc{OpS}$$

The first rule describes the communication between two replicas, which propagates a message from one replica to another. When such a message arrives, the receiving replica waits until an associated causality condition for the message is met. Once this waiting condition is met, the replica updates its local object table according to the second rule. The last rule describes the access to a replica by a session. This generates a new action, which is later propagated to other replicas in the form of a message. Also, it changes the configurations of session and replica that interact, as described by the rule.

LEMMA 11. *If $\mathcal{S}|\mathcal{D} \xrightarrow{\iota} \mathcal{S}'|\mathcal{D}'$ and $\mathcal{S}|\mathcal{D}$ is well-formed, so is $\mathcal{S}'|\mathcal{D}'$.*

Proof: Assume that $\mathcal{S}|\mathcal{D}$ is well-formed and $\mathcal{S}|\mathcal{D} \xrightarrow{\iota} \mathcal{S}'|\mathcal{D}'$. We should show that $\mathcal{S}'|\mathcal{D}'$ is also well-formed. This means two properties of $\mathcal{S}'|\mathcal{D}'$. First, both $\mathcal{S}'$ and $\mathcal{D}'$ are functional. Second,

$$\begin{aligned}\forall e \in \mathsf{id}(\mathcal{S}'|\mathcal{D}').\, \exists t.\, \exists (d, t', \_, \_, \_) \in \mathcal{D}'. \\ e = (t, d) \wedge e < (t', d).\end{aligned}$$

The first property follows from the well-formedness of $\mathcal{S}|\mathcal{D}$ and the fact that all three rules in our abstract implementation do not create any session configurations nor replica configurations. For the second property, we show it by case analysis on the rule used to obtain $\mathcal{S}'|\mathcal{D}'$.

- If the rule is PROP, we have

$$\begin{aligned}&\mathsf{id}(\mathcal{S}|\mathcal{D}) = \mathsf{id}(\mathcal{S}'|\mathcal{D}') \\ &\wedge (\forall (t, d) \in \mathsf{time}(\mathcal{S}|\mathcal{D}).\, \exists t'. \\ &\qquad (t', d) \in \mathsf{time}(\mathcal{S}'|\mathcal{D}') \wedge (t, d) \leq (t', d')).\end{aligned}$$

  The desired property of $\mathcal{S}'|\mathcal{D}'$ follows from these two conjuncts and the well-formedness of $\mathcal{S}|\mathcal{D}$.
- If the rule is OPD, we have

$$\mathsf{id}(\mathcal{S}|\mathcal{D}) = \mathsf{id}(\mathcal{S}'|\mathcal{D}') \wedge \mathsf{time}(\mathcal{S}|\mathcal{D}) = \mathsf{time}(\mathcal{S}'|\mathcal{D}'). \quad (4)$$

  Here the second conjunct holds because the rule does not change the timestamps of any replicas. The first conjunct holds because the OPD rule changes only one replica by running eval, but

$$\begin{aligned}(\_, \sigma') = \mathsf{eval}_\tau(e, f, W, \rho(x)) \\ \implies \mathsf{id}_\tau(\sigma') = \mathsf{id}_\tau(\rho(x)) \cup \{e\}.\end{aligned}$$

  The second property of $\mathcal{S}'|\mathcal{D}'$ follows from (4) and the well-formedness of $\mathcal{S}|\mathcal{D}$.
- The remaining case is that the rule is OPS. By the definition of OPS, we have

$$\begin{aligned}\forall (t, d) \in \mathsf{time}(\mathcal{S}|\mathcal{D}).\, \exists t'. \\ (t', d) \in \mathsf{time}(\mathcal{S}'|\mathcal{D}') \wedge (t, d) \leq (t', d').\end{aligned} \quad (5)$$

  Pick $e' \in \mathsf{id}(\mathcal{S}'|\mathcal{D}')$. Let (x, e, f, W', E') be a message generated by the rule. If e' does not belong to $\{e\} \cup W' \cup E'$, it should be in $\mathsf{id}(\mathcal{S}|\mathcal{D})$, and the desired property of $\mathcal{S}'|\mathcal{D}'$ follows from (5) and the well-formedness of $\mathcal{S}|\mathcal{D}$. If $e' = e$, by the definition of the rule, we have

$$\exists d, t.\, (t, d) = e' \wedge (d, \mathsf{next}(t), \_, \_, \_) \in \mathcal{D}',$$

  from which follows the desired property of $\mathcal{S}'|\mathcal{D}'$. Finally, if $e' \in (W' \cup E') \setminus \{e\}$, we use the condition on the depend operator and the definition of W', and derive that

$$\begin{aligned}e' \in ((W' \cup E') \setminus \{e\}) &\subseteq \bigcup_{y \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(y)}(\rho(y)) \\ &\subseteq \mathsf{id}(\mathcal{S}|\mathcal{D}),\end{aligned}$$

  where ρ is the fourth argument used for computing E' in the OPS rule. Now the well-formedness of $\mathcal{S}|\mathcal{D}$ and the formula in (5) give the desired property of $\mathcal{S}'|\mathcal{D}'$.

□

Let $\mathsf{id}(\iota) = \{\mathsf{id}(a) \mid (\_, a) = \iota\}$. We show that the rules of our implementation increase the set of action ids occurring in a system configuration.

LEMMA 12. *If* $\mathcal{S}|\mathcal{D} \xrightarrow{\iota} \mathcal{S}'|\mathcal{D}'$, *then*

$$\mathsf{id}(\mathcal{S}|\mathcal{D}) \subseteq \mathsf{id}(\mathcal{S}'|\mathcal{D}') \wedge \mathsf{id}(\mathcal{S}|\mathcal{D}) \cup \mathsf{id}(\iota) = \mathsf{id}(\mathcal{S}'|\mathcal{D}').$$

Proof: We do the case analysis on the rule used in the transition:

$$\mathcal{S}|\mathcal{D} \xrightarrow{\iota} \mathcal{S}'|\mathcal{D}'.$$

The first case is that the rule is PROP. The rule PROP only delivers an existing message to a new target replica, and does not change the object table of any replica. Furthermore, $\mathsf{id}(\iota) = \emptyset$. The lemma follows from these properties of PROP.

The next case is OPS. As in the previous case, OPS only adds a new message (x, e, f, W', E') to some replica, where all action ids in $W' \cup E'$ already appear in the same replica before the rule. Also, although OPS changes the object table of a replica, this change happens only via the execution of eval, which satisfies the following property:

$$(\_, \sigma') = \mathsf{eval}_\tau(e, f, W, \sigma) \implies \mathsf{id}_\tau(\sigma) \cup \{e\} = \mathsf{id}_\tau(\sigma').$$

Since $\mathsf{id}(\iota) = \{e\}$, the claim of this lemma follows from what we have just described.

The final case is OPD. In this case, no message is removed from or added to a replica. The object table of some replica is changing by OPD, but this is done via the execution of eval. Hence, as explained in the previous case, this change only increases the set of action ids in the object table of a replica, and this increment is the action id e of the message (x, e, f, W, E) that is selected by the rule. Note that this message already exists in the replica before the application of the rule, so e is not a new action id. From the observations that we have just made follows that $\mathsf{id}(\mathcal{S}|\mathcal{D}) \subseteq \mathsf{id}(\mathcal{S}'|\mathcal{D}')$. Since $\mathsf{id}(\iota) = \emptyset$, the claim of this lemma holds. □

A message (x, e, f, W, E) ***respects time*** if

$$\forall e' \in E \cup W.\, e' < e.$$

A system configuration $\mathcal{S}|\mathcal{D}$ ***respects time*** if all messages in the configuration respect time and the following condition is met for every replica $(d, t, \rho, M, N) \in \mathcal{D}$:

$$(\forall x \in \mathsf{Obj}.\, \forall e \in \mathsf{id}_{\mathsf{type}(x)}(\rho(x)).\, e < (t, d)) \wedge$$
$$(\forall (\_, e, \_, W, E) \in M \cup N.\, \forall e' \in \{e\} \cup W \cup E.\, e' < (t, d)).$$

LEMMA 13. *If* $\mathcal{S}|\mathcal{D} \xrightarrow{\iota} \mathcal{S}'|\mathcal{D}'$ *and* $\mathcal{S}|\mathcal{D}$ *respects time, then* $\mathcal{S}'|\mathcal{D}'$ *also respects time.*

Proof: We do the case analysis on the rule used in the transition:

$$\mathcal{S}|\mathcal{D} \xrightarrow{\iota} \mathcal{S}'|\mathcal{D}'.$$

The first case is PROP. The rule delivers a message (x, e, f, W, E) from one replica to another, but it does not create any new messages. Thus, all the messages in $\mathcal{S}'|\mathcal{D}'$ are also in $\mathcal{S}|\mathcal{D}$, so they should respect time. It remains to prove that the condition on a replica's timestamp holds for all replicas in $\mathcal{D}'$. If a replica in $\mathcal{D}'$ is not modified by the application of PROP, this condition follows from our assumption on $\mathcal{S}|\mathcal{D}$. Let $\delta = (d, t, \rho, M, N) \in \mathcal{D}'$ be the replica modified by PROP, and let $m = (x, e, f, W, E)$ be the message delivered by PROP. Pick an action id e_0 such that

$$(\exists (\_, e', \_, W', E') \in M \cup N.\, e_0 \in \{e'\} \cup W' \cup E')$$
$$\vee\, (\exists x \in \mathsf{Obj}.\, e_0 \in \mathsf{id}_{\mathsf{type}(x)}(\rho(x))).$$

If the second disjunct holds or the first disjunct is true with a witness different from m, the condition on the timestamp of δ follows from our assumption on $\mathcal{S}|\mathcal{D}$. Otherwise,

$$e_0 \in \{e\} \cup W \cup E.$$

Let (d', t') be the timestamp of a replica that sent the message m in our application of PROP. By our assumption on $\mathcal{S}|\mathcal{D}$,

$$e_0 < (t', d').$$

But by the definition of PROP, $(t', d') < (t, d)$. Hence, $e_0 < (t, d)$ as desired.

The second case is OPD. This rule does not create any new messages. Since every message in $\mathcal{S}|\mathcal{D}$ respects time, so do the ones in $\mathcal{S}'|\mathcal{D}'$. We can thus complete the proof of this case if we show that the condition on the timestamp of a replica holds for every replica in $\mathcal{D}'$. Note that the OPD rule changes only one replica. For all the other unchanged replicas in $\mathcal{D}'$, the condition on their timestamps holds, because these replicas are already present in $\mathcal{S}|\mathcal{D}$ and every replica in $\mathcal{S}|\mathcal{D}$ satisfies the condition on its timestamp. Let δ, δ' be replicas in $\mathcal{D}$ and $\mathcal{D}'$, respectively, such that the OPD rule changes δ to δ'. Then, there are $d, t, \rho, M, N, x, e, f, W, E, \sigma'$ such that

$$\delta = (d, t, \rho, M, N)$$
$$\wedge\, \delta' = (d, t, \rho[x \mapsto \sigma'], M, N)$$
$$\wedge\, (x, e, f, W, E) \in N \wedge (\_, \sigma') = \mathsf{eval}_{\mathsf{type}(x)}(e, f, W, \rho(x)).$$

By the condition imposed on eval, the above formula implies that every action id e' appearing in δ' also occurs in δ. By the assumption on $\mathcal{S}|\mathcal{D}$, this implies that

$$e' < (t, d)$$

which is precisely what we have to prove.

The final case is OPS. Let δ, δ' be replicas in $\mathcal{D}$ and $\mathcal{D}'$ such that the OPS rule transforms δ to δ'. Then, there exist $d, t, \rho, M, N, x, \sigma', f, W', E'$ satisfying the following properties:

$$(\delta = (d, t, \rho, M, N)) \wedge$$
$$((\_, \sigma') = \mathsf{eval}_{\mathsf{type}(x)}(\_, \_, \_, \rho(x))) \wedge$$
$$(W' = \mathsf{id}_{\mathsf{type}(x)}(\rho(x))) \wedge$$
$$(E' = \mathsf{depend}_{\mathsf{type}(x)}(\_, \_, \rho(x), \rho, \_)) \wedge$$
$$(\delta' = (d, \mathsf{next}(t), \rho[x \mapsto \sigma'], M \cup \{(x, (t, d), f, W', E')\}, N))$$

Every message in $\mathcal{D}'$ other than $m = (x, (t, d), f, W', E')$ exists in $\mathcal{D}$, so it reflects time. By the condition imposed on depend and the definition of W',

$$(E' \cup W') \subseteq \bigcup_{y \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(y)}(\rho(y))$$

Hence, by our assumption on $\mathcal{S}|\mathcal{D}$,

$$\forall e' \in E' \cup W'.\, e' < (t, d).$$

Hence, m also respects time. It remains to show the condition on the timestamp of each replica in $\mathcal{D}'$. This condition is met for all replicas in $\mathcal{D}'$ other than δ', because they are already present in $\mathcal{D}$ and every replica in $\mathcal{D}$ satisfies the same condition on its timestamp. Pick an action id e' in δ'. Then, e' already appears in δ, or

$$\begin{aligned} e' &\in \mathsf{id}_{\mathsf{type}(x)}(\sigma') \cup \{(t, d)\} \cup E' \cup W' \\ &= \{(t, d)\} \cup \bigcup_{y \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(y)}(\rho(y)) \end{aligned}$$

where the equality above uses the conditions on id and depend. If $e' = (t, d)$, then

$$e' = (t, d) < (\mathsf{next}(t), d).$$

Otherwise, e' appears in δ already. So in this case, by our assumption on $\mathcal{S}|\mathcal{D}$, we have

$$e' < (t, d) < (\mathsf{next}(t), d).$$

In both cases, we proved that the condition on the timestamp of δ' holds, as desired. □

A configuration $\mathcal{S}|\mathcal{D}$ is ***empty*** if for all $(\_, \_, \rho, M, N) \in \mathcal{D}$,

$$(M = N = \emptyset) \wedge (\forall x \in \mathsf{Obj}.\, \rho(x) = \mathsf{init}_{\mathsf{type}(x)}).$$

COROLLARY 14. *If $\mathcal{S}_0|\mathcal{D}_0$ is empty and*

$$\mathcal{S}_0|\mathcal{D}_0 \xrightarrow{\iota_1} \mathcal{S}_1|\mathcal{D}_1 \xrightarrow{\iota_2} \dots \xrightarrow{\iota_n} \mathcal{S}_n|\mathcal{D}_n$$

then $\mathcal{S}_n|\mathcal{D}_n$ respects time.

Proof: This corollary follows from Lemma 13 and the fact that all empty system configurations respect time. □

B.3 Fair Trace

We represent a computation of our implementation in terms of a trace:

DEFINITION 15. *A* **trace** *is a tuple*

$$(\mathcal{S}_0, \mathcal{D}_0, \{(\iota_k, \mathcal{S}_k, \mathcal{D}_k)\}_{1 \le k \le n})$$

for $n \in \mathbb{N} \cup \{\omega\}$, such that $\mathcal{S}_0|\mathcal{D}_0$ is empty and

$$\forall k.\, (1 \le k \le n) \implies (\mathcal{S}_{k-1}|\mathcal{D}_{k-1} \xrightarrow{\iota_k} \mathcal{S}_k|\mathcal{D}_k).$$

We will often present traces in the following more readable form:

$$\tau = \mathcal{S}_0|\mathcal{D}_0 \xrightarrow{\iota_1} \mathcal{S}_1|\mathcal{D}_1 \xrightarrow{\iota_2} \dots \xrightarrow{\iota_n} \mathcal{S}_n|\mathcal{D}_n$$

LEMMA 16. *Every trace*

$$\tau = \mathcal{S}_0|\mathcal{D}_0 \xrightarrow{\iota_1} \mathcal{S}_1|\mathcal{D}_1 \xrightarrow{\iota_2} \dots \xrightarrow{\iota_n} \mathcal{S}_n|\mathcal{D}_n,$$

satisfies the following properties:

1. *For every i,*

$$\mathsf{id}(\mathcal{S}_i|\mathcal{D}_i) = \{\mathsf{id}(a) \mid \exists k.\, (\_, a) = \iota_k \wedge k \le i\}.$$

   This implies that $\mathsf{id}(\mathcal{S}_i|\mathcal{D}_i) \subseteq \mathsf{id}(\mathcal{S}_j|\mathcal{D}_j)$ for all i, j with $i < j$.
2. *For all $i \in \{1, \dots, n\}$ and $a \in \mathsf{Act}$, if $(\_, a) = \iota_i$, then*

$$\mathsf{id}(a) \notin \mathsf{id}(\mathcal{S}_{i-1}|\mathcal{D}_{i-1}) \wedge \mathsf{id}(a) \in \mathsf{id}(\mathcal{S}_i|\mathcal{D}_i).$$

3. *For all $i, j \in \{1, \dots, n\}$ and $a, b \in \mathsf{Act}$, if*

$$(\_, a) = \iota_i \wedge (\_, b) = \iota_j \wedge i \ne j$$

   then $\mathsf{id}(a) \ne \mathsf{id}(b)$.

Proof: The first property follows from Lemma 12 and the fact that $\mathsf{id}(\mathcal{S}_0|\mathcal{D}_0) = \emptyset$. To prove the second, consider i, a such that $(\_, a) = \iota_i$. By the definition of our rules, $\mathsf{id}(a)$ should come from the timestamp and the id of a replica in the configuration $\mathcal{S}_{i-1}|\mathcal{D}_{i-1}$. By the well-formedness of $\mathcal{S}_{i-1}|\mathcal{D}_{i-1}$, this means that $\mathsf{id}(a)$ is not in $\mathsf{id}(\mathcal{S}_{i-1}|\mathcal{D}_{i-1})$. Also, by the definition of our rules, $\mathsf{id}(a)$ should be in $\mathsf{id}(\mathcal{S}_i|\mathcal{D}_i)$. We have just shown that the second property holds. The last property in the lemma is a consequence of the other two properties. □

LEMMA 17. *Consider a trace*

$$\tau = \mathcal{S}_0|\mathcal{D}_0 \xrightarrow{\iota_1} \mathcal{S}_1|\mathcal{D}_1 \xrightarrow{\iota_2} \dots \xrightarrow{\iota_n} \mathcal{S}_n|\mathcal{D}_n.$$

For all i, $(\_, \_, \_, M, N) \in \mathcal{D}_i$ and $(\_, e, \_, W, \_) \in M \cup N$, there exist a, j such that

$$j \le i \wedge (W, a) = \iota_j \wedge \mathsf{id}(a) = e.$$

Proof: By our condition on the trace, there are no messages in the initial configuration $\mathcal{S}_0|\mathcal{D}_0$. Hence, all messages in some $\mathcal{D}_i$ are generated by the application of the OPS rule. So, for every replica $(\_, \_, \_, M, N) \in \mathcal{D}_i$ and every message $m = (\_, e, \_, W, \_) \in M \cup N$ in this replica, there is an index k such that the message is generated by OPS in the k-th step. This k and the action generated at this step are the desired j and a in this lemma. □

LEMMA 18. *Consider a trace*

$$\tau = \mathcal{S}_0|\mathcal{D}_0 \xrightarrow{\iota_1} \mathcal{S}_1|\mathcal{D}_1 \xrightarrow{\iota_2} \dots \xrightarrow{\iota_n} \mathcal{S}_n|\mathcal{D}_n.$$

For all i, $e \in \mathsf{id}(\mathcal{S}_i|\mathcal{D}_i)$, $(\_, \_, \rho, M, N) \in \mathcal{D}_i$ and $x \in \mathsf{Obj}$, if

$$e \in \mathsf{id}_{\mathsf{type}(x)}(\rho(x)) \vee (\exists W.\, (x, \_, \_, W, \_) \in M \cup N \wedge e \in W)$$

then

$$\exists k \le i.\, \exists a.\, (\_, a) = \iota_k \wedge \mathsf{id}(a) = e \wedge \mathsf{obj}(a) = x.$$

Proof: Pick i, ρ, M, N, e, x such that

$$((\_, \_, \rho, M, N) \in \mathcal{D}_i) \wedge$$
$$(e \in \mathsf{id}_{\mathsf{type}(x)}(\rho(x)) \vee \exists W.\, (x, \_, \_, W, \_) \in M \cup N \wedge e \in W).$$

By definition, the initial configuration of the trace τ should be empty. This implies that $\mathsf{id}(\mathcal{S}_0|\mathcal{D}_0) = \emptyset$. Hence, by Lemma 12, there exist k, a such that

$$k \leq i \wedge (\_, a) = \iota_k \wedge \mathsf{id}(a) = e.$$

It remains to show that $\mathsf{obj}(a) = x$. We show this by induction on $i - k$. Suppose that $i = k$. Then, by the well-formedness of $\mathcal{S}_{i-1}|\mathcal{D}_{i-1}$, e is not in $\mathsf{id}(\mathcal{S}_{i-1}|\mathcal{D}_{i-1})$. This means that the state $\rho(x)$ or the message $(x, \_, \_, W, \_)$ is produced by the rule at the k-th step. This can happen only when $x = \mathsf{obj}(a)$. Now suppose that $i > k$. If e appears in a message, this message must already exist in $\mathcal{S}_{i-1}|\mathcal{D}_{i-1}$. In this case, by induction hypothesis, we get $x = \mathsf{obj}(a)$. Otherwise, e is in $\mathsf{id}_{\mathsf{type}(x)}(\rho(x))$. Let ρ' be the object table of the same replica in $\mathcal{S}_{i-1}|\mathcal{D}_{i-1}$. If e is already in $\mathsf{id}_{\mathsf{type}(x)}(\rho'(x))$, we can use induction hypothesis and deduce that $x = \mathsf{obj}(a)$ as desired. If not, this means that the OPD rule was applied at the i-th step to $\rho'(x)$ and some message (x, e, f', W', E'), and produced $\rho(x)$. Now we can apply the induction hypothesis on the message (x, e, f', W', E'), and obtain the desired conclusion that $x = \mathsf{obj}(a)$. □

LEMMA 19. *Consider a trace*

$$\tau = \mathcal{S}_0|\mathcal{D}_0 \xrightarrow{\iota_1} \mathcal{S}_1|\mathcal{D}_1 \xrightarrow{\iota_2} \ldots \xrightarrow{\iota_n} \mathcal{S}_n|\mathcal{D}_n.$$

Every configuration $\mathcal{S}_i|\mathcal{D}_i$ respects time.

Proof: Every empty configuration respects time. Furthermore, by Lemma 13, all the rules in our abstract implementation transforms time-respecting configurations to time-respecting ones. This lemma follows from these two observations. □

DEFINITION 20. *A trace*

$$\tau = \mathcal{S}_0|\mathcal{D}_0 \xrightarrow{\iota_1} \mathcal{S}_1|\mathcal{D}_1 \xrightarrow{\iota_2} \ldots \xrightarrow{\iota_n} \mathcal{S}_n|\mathcal{D}_n$$

is **fair** *if for every action a such that $(\_, a) = \iota_k$ for some k, there is a threshold $1 \leq k_0 \leq n$ satisfying the following condition:*

$$\forall i \geq k_0.\, \forall (\_, \_, \rho, \_, \_) \in \mathcal{D}_i.$$
$$\exists x \in \mathsf{Obj}.\, x = \mathsf{obj}(a) \wedge \mathsf{id}(a) \in \mathsf{id}_{\mathsf{type}(x)}(\rho(x)).$$

B.4 Correspondence: Basic Axioms

We now relate the abstract implementation with the axiomatic description given in the main part of this paper. Our plan is to show that every fair trace generates an execution in our axiomatic semantics that satisfies all the well-formedness axioms, the data type axiom and the basic eventual consistency axioms in Figure 1. Once this basic correspondence is established, we describe further conditions on the parameters of our implementation, and show that they validate axioms for session guarantees and causality.

Consider a fair trace

$$\tau = (\mathcal{S}_0, \mathcal{D}_0, \{(\iota_k, \mathcal{S}_k, \mathcal{D}_k)\}_k).$$

We generate an execution $(A, \mathsf{so}, \mathsf{vis}, \mathsf{ar})$ from this trace τ as follows:

$$A = \{a \mid \exists k.\, (\_, a) = \iota_k\},$$
$$\mathsf{vis} = \{(a, b) \in A \times A \mid \exists k, W.\, (W, b) = \iota_k \wedge \mathsf{id}(a) \in W\},$$
$$\mathsf{so} = \{(a, b) \in A \times A \mid \exists k, l.\, k < l \wedge (\_, a) = \iota_k \wedge (\_, b) = \iota_l \wedge \mathsf{ses}(a) = \mathsf{ses}(b)\},$$
$$\mathsf{ar} = \{(a, b) \in A \times A \mid \mathsf{id}(a) < \mathsf{id}(b) \wedge \mathsf{obj}(a) = \mathsf{obj}(b)\}.$$

This execution assumes a replicated data type specification $\mathcal{F}$ that is compatible with a data type implementation

$$\{(\mathsf{St}_\tau, \mathsf{init}_\tau, \mathsf{eval}_\tau, \mathsf{id}_\tau)\}_\tau$$

in the following sense: for every type τ, there exists a relation $\mathcal{R}_\tau$ between the last three components of contexts for the type τ and data type states in St_τ such that

1. $((\emptyset, \emptyset, \emptyset), \mathsf{init}_\tau) \in \mathcal{R}_\tau$;
2. if

$$((V, r_0, r_1), \sigma) \in \mathcal{R}_\tau$$
$$\wedge\, \mathsf{eval}_\tau(e, f, W, \sigma) = (w, \sigma') \wedge (\forall g \in \mathsf{Op}_\tau.\, (e, g) \notin V)$$
$$\wedge\, r_0' \supseteq (r_0 \cup \{((e', g), (e, f)) \mid e' \in W \wedge (e', g) \in V\})$$
$$\wedge\, r_1' = r_1 \cup \{((e', g), (e, f)) \mid e' < e \wedge (e', g) \in V\} \cup \{((e, f), (e', g)) \mid e < e' \wedge (e', g) \in V\}$$

then

$$((V \cup \{(e, f)\}, r_0', r_1'), \sigma') \in \mathcal{R}_\tau \wedge \mathcal{F}_\tau(f, V, r_0, r_1) = w;$$

PROPOSITION 21. *All the fair traces generate executions satisfying* SOWF*,* VISWF*,* ARWF*,* EVENTUAL *and* THINAIR*.*

Proof: Pick a fair trace τ:

$$\tau = \mathcal{S}_0|\mathcal{D}_0 \xrightarrow{\iota_1} \mathcal{S}_1|\mathcal{D}_1 \xrightarrow{\iota_2} \ldots \xrightarrow{\iota_n} \mathcal{S}_n|\mathcal{D}_n$$

where $n \in \mathbb{N} \cup \{\omega\}$. Let $(A, \mathsf{vis}, \mathsf{so}, \mathsf{ar})$ be an execution constructed from this trace according to our recipe described above. We will show that this execution satisfies all the axioms mentioned in the proposition.

Firstly, we show that SOWF holds. By the definition of so, if $a \xrightarrow{\mathsf{so}} b$, then $\mathsf{ses}(a) = \mathsf{ses}(b)$. Furthermore, in this case, $a \neq b$ because of Lemma 16. Hence, so is irreflexive. Furthermore, for every $a, b \in A$, if $a \neq b$ but $\mathsf{ses}(a) = \mathsf{ses}(b)$, we have that

$$(a \xrightarrow{\mathsf{so}} b) \vee (a \xrightarrow{\mathsf{so}} b)$$

by the definition of so. It remains to show that so is transitive. Consider $a, b, c \in A$ such that $a \xrightarrow{\mathsf{so}} b$ and $b \xrightarrow{\mathsf{so}} c$. By the definition of so, there are i, j, k, l such that

$$i < j \wedge k < l \wedge \iota_i = (\_, a) \wedge \iota_j = (\_, b)$$
$$\wedge\, \iota_k = (\_, b) \wedge \iota_l = (\_, c).$$

But $j = k$ because of Lemma 16. Hence, $a \xrightarrow{\mathsf{so}} c$ by the definition of so.

Secondly, we consider the VISWF axiom. This follows from Lemma 18 and the definition of the OPS rule.

Thirdly, we prove the ARWF axiom. Suppose that $a \xrightarrow{\mathsf{ar}} b$. By the definition of ar, $\mathsf{obj}(a) = \mathsf{obj}(b)$. The irreflexivity and the transitivity of ar follow from the same properties of $<$ on action ids. Since $<$ is total and the ids of different actions in A are different (Lemma 16), ar relates any two actions in A that work on the same object. Furthermore, for every $a \in A$, $\mathsf{vis}^{-1}(a)$ contains only those actions working on $\mathsf{obj}(a)$, because of the VISWF axiom that we just proved above. Hence, $\mathsf{ar}|_{\mathsf{vis}^{-1}(a)}$ is a total order for every $a \in A$, as required in the last part of the ARWF axiom.

Fourthly, we show the EVENTUAL axiom. If A is finite, the axiom becomes vacuous and it holds. Suppose that A is infinite. Note that this supposition implies $n = \omega$. Pick an action $a \in A$. Let $x = \mathsf{obj}(a)$. Then, there exists k such that

$$(\_, a) = \iota_k.$$

Since the trace is fair, there exists k_0 such that

$$\forall i \geq k_0.\, \forall (\_, \_, \rho, \_, \_) \in \mathcal{D}_i.\, \mathsf{id}(a) \in \mathsf{id}_{\mathsf{type}(x)}(\rho(x)).$$

To show the axiom holds for a, we should prove that there are at most finitely many $b \in A$ satisfying the condition below:

$$\mathsf{obj}(b) = x \wedge \neg(a \xrightarrow{\mathsf{vis}} b).$$

Notice that if an action $b \in A$ satisfies the condition above,

$$\exists k_1.\, k_1 \leq k_0 \wedge (\_, b) = \iota_{k_1}.$$

But the number of possible witnesses k_1 in the above formula is at most k_0. So only finitely many b's can satisfy the condition above.

Fifthly, we prove that the THINAIR axiom is satisfied. We define a total order $<_\tau$ on A by

$$a <_\tau b \iff \exists i, j.\, i < j \wedge \iota_i = (\_, a) \wedge \iota_j = (\_, b).$$

By Lemma 16, $<_\tau$ is an irreflexive and transitive total order on A. We will prove that

$$\forall a, b \in A.\, (a \xrightarrow{\mathsf{vis}} b \vee a \xrightarrow{\mathsf{so}} b) \implies a <_\tau b. \tag{6}$$

To see why the THINAIR axiom follows from this formula, note that the formula implies

$$(\mathsf{vis} \cup \mathsf{so})^+ \subseteq (<_\tau)^+$$

but the RHS of this subset relationship is included in $<_\tau$ because $<_\tau$ is transitive. Now the irreflexivity of $<_\tau$ implies that $(\mathsf{vis} \cup \mathsf{so})^+$ is irreflexive as well, as required by the THINAIR axiom. Let us go back to the proof of the formula in (6). Pick $a, b \in A$. If $a \xrightarrow{\mathsf{so}} b$, then $a <_\tau b$ by the definition of so. Now suppose that $a \xrightarrow{\mathsf{vis}} b$. This means that there exists i such that

$$a \in \mathsf{id}(\mathcal{S}_i|\mathcal{D}_i) \wedge (\_, b) = \iota_{i+1}.$$

By Lemma 12, there should be k such that

$$a \notin \mathsf{id}(\mathcal{S}_k|\mathcal{D}_k) \wedge (\_, a) = \iota_{k+1}.$$

By Lemma 16,

$$k + 1 < i + 1.$$

Hence, $a <_\tau b$, as desired. □

PROPOSITION 22. *All the fair traces generate executions satisfying* RVAL.

Proof: Pick a trace τ:

$$\tau = \mathcal{S}_0|\mathcal{D}_0 \xrightarrow{\iota_1} \mathcal{S}_1|\mathcal{D}_1 \xrightarrow{\iota_2} \dots \xrightarrow{\iota_n} \mathcal{S}_n|\mathcal{D}_n$$

where $n \in \mathbb{N} \cup \{\omega\}$. Let $(A, \mathsf{vis}, \mathsf{so}, \mathsf{ar})$ be an execution constructed from this trace according to our recipe described above.

We will first show that

$$\begin{aligned}
&\forall i.\, \forall (_, _, \rho, _, _) \in \mathcal{D}_i.\, \forall x \in \mathsf{Obj}.\\
&\quad \exists \tau, \sigma, B.\, \tau = \mathsf{type}(x) \wedge \sigma = \rho(x)\\
&\qquad \wedge B = \{a \in A \mid \mathsf{id}(a) \in \mathsf{id}_\tau(\sigma)\}\\
&\qquad \wedge ((\mathsf{event}(B), \mathsf{event}(\mathsf{vis}|_B), \mathsf{event}(\mathsf{ar}|_B)), \sigma) \in \mathcal{R}_\tau.
\end{aligned} \tag{7}$$

Our proof is by induction on i. Pick ρ and x as described in the formula above, and define τ, σ, B again as described in the formula. We need to show that the last conjunct in the formula holds. When $i = 0$,

$$\sigma = \mathsf{init}_\tau \wedge B = \emptyset.$$

Hence, the claimed relationship by $\mathcal{R}_\tau$ in the formula becomes

$$((\emptyset, \emptyset, \emptyset), \mathsf{init}_\tau) \in \mathcal{R}_\tau,$$

which holds because it is precisely one of the conditions assumed on $\mathcal{R}_\tau$. Now assume that $i > 0$, and that the formula in (7) holds for all $0 \leq j < i$. We do the case analysis on the rule used at the i-th step. If the rule is PROP, no object tables of replicas change during the i-th step. The last conjunct in (7) follows from induction hypothesis. The other cases are that OPS and OPD are used. In these cases, the induction hypothesis gives the desired conclusion except when ρ is the table of a replica changed by the rule and x is the object affected by the rule. To take care of this exception, assume that ρ and x are the table and the object updated by the rule. Then, there exist e, f, W_0, σ_0, B_0 and $a_0 \in A$ such that

$$\begin{aligned}
&(\mathsf{eval}_\tau(e, f, W_0, \sigma_0) = (_, \sigma))\\
&\wedge B_0 = \{a \in A \mid \mathsf{id}(a) \in \mathsf{id}_\tau(\sigma_0)\}\\
&\wedge (\forall g.\, (e, g) \notin \mathsf{event}(B_0))\\
&\wedge ((\mathsf{event}(B_0), \mathsf{event}(\mathsf{vis}|_{B_0}), \mathsf{event}(\mathsf{ar}|_{B_0})), \sigma_0) \in \mathcal{R}_\tau\\
&\wedge \mathsf{event}(a_0) = (e, f)\\
&\wedge (B_0 \cup \{a_0\} = B)\\
&\wedge \{b \in B_0 \mid \mathsf{id}(b) \in W_0\} \subseteq \mathsf{vis}^{-1}(a_0).
\end{aligned} \tag{8}$$

The third conjunct comes from the well-formedness of configurations (OPS) or the conditions in the rule (OPD). The fourth

conjunct holds because of induction hypothesis. The fifth conjunct is true since for every action id such as e in a fair trace, there are a, k with $\iota_k = (\_, a) \wedge \mathsf{id}(a) = e$ (Lemma 12). The sixth conjunct follows from our assumption on eval_τ, the fact that σ is obtained from σ' and the following properties of B and B_0:

$$(B = \{a \in A \mid \mathsf{id}(a) \in \mathsf{id}_\tau(\sigma)\}) \\ \wedge (B_0 = \{a \in A \mid \mathsf{id}(a) \in \mathsf{id}_\tau(\sigma_0)\}).$$

The seventh conjunct holds because of Lemma 17 (OPD) or the definition of the rule (OPS). We note two consequences of the formula in (8):

$$\begin{aligned}&\mathsf{event}(\mathsf{vis}|_{B_0}) \cup \{((e', f'), (e, f)) \mid (e', f') \in \mathsf{event}(B_0) \\ &\qquad\qquad \wedge e' \in W_0\} \\ &= \mathsf{event}(\mathsf{vis}|_{B_0}) \cup \mathsf{event}(\{(b, a_0) \mid b \in B_0 \wedge \mathsf{id}(b) \in W_0\}) \\ &\subseteq \mathsf{event}(\mathsf{vis}|_{B_0 \cup \{a_0\}}) \\ &= \mathsf{event}(\mathsf{vis}|_B).\end{aligned}$$

And

$$\begin{aligned}&\mathsf{event}(\mathsf{ar}|_{B_0}) \\ &\quad \cup \{((e', f'), (e, f)) \mid (e', f') \in \mathsf{event}(B_0) \wedge e' < e\} \\ &\quad \cup \{((e, f), (e', f')) \mid (e', f') \in \mathsf{event}(B_0) \wedge e < e'\} \\ &= \mathsf{event}(\mathsf{ar}|_{B_0}) \\ &\qquad \cup \mathsf{event}(\{(b, a_0) \mid b \in B_0 \wedge \mathsf{id}(b) < \mathsf{id}(a_0)\}) \\ &\qquad \cup \mathsf{event}(\{(a_0, b) \mid b \in B_0 \wedge \mathsf{id}(a_0) < \mathsf{id}(b)\}) \\ &= \mathsf{event}(\mathsf{ar}|_{B_0 \cup \{a_0\}}) = \mathsf{event}(\mathsf{ar}|_B).\end{aligned}$$

By the second condition on $\mathcal{R}_\tau$, what we have shown so far imply that

$$((\mathsf{event}(B), \mathsf{event}(\mathsf{vis}|_B), \mathsf{event}(\mathsf{ar}|_B)), \sigma_0) \in \mathcal{R}_\tau,$$

as desired.

Next we use the formula in (7) and prove that this execution satisfies the RVAL axiom. Pick $a \in A$. By the definition of A, there exists a unique k such that

$$\iota_k = (\_, a).$$

Let ρ and x be the table and the object that the OPS rule has used for creating the action a in the k-th step. Define τ, σ, B as follows:

$$\tau = \mathsf{type}(x) \wedge \sigma = \rho(x) \wedge B = \{b \in A \mid \mathsf{id}(b) \in \mathsf{id}_\tau(\sigma)\}.$$

Then, by (7),

$$((\mathsf{event}(B), \mathsf{event}(\mathsf{vis}|_B), \mathsf{event}(\mathsf{ar}|_B)), \sigma) \in \mathcal{R}_\tau. \quad (9)$$

Meanwhile, $B = \mathsf{vis}^{-1}(a)$ by the definition of vis, so that

$$\mathsf{ctxt}(a) = (\mathsf{op}(a), \mathsf{event}(B), \mathsf{event}(\mathsf{vis}|_B), \mathsf{event}(\mathsf{ar}|_B)).$$

Furthermore, $\mathsf{id}(a) \notin \{\mathsf{id}(b) \mid b \in B\}$ by the well-formedness of the configuration at the $(k-1)$-th step and the definition of the OPS rule. Hence, the relationship in (9) implies

$$(\mathcal{F}_\tau(\mathsf{ctxt}(a)), \_) = \mathsf{eval}_\tau(\mathsf{id}(a), \mathsf{op}(a), \mathsf{id}_\tau(\sigma), \sigma)).$$

Since $\mathsf{rval}(a)$ is defined to be the value computed by the RHS function above in the definition of OPS, this in turn gives $\mathsf{rval}(a) = \mathcal{F}_\tau(\mathsf{ctxt}(a))$, as desired. □

B.5 Correspondence: Axioms for Session Guarantees

Next, we consider axioms about session guarantees. To do so, we make a few assumptions:

1. We assume that SCtx has the following form:

   $$\mathsf{SCtx} = (\mathsf{Obj} \to \mathcal{P}_{\mathsf{fin}}(\mathsf{AId})) \times (\mathsf{Obj} \to \mathcal{P}_{\mathsf{fin}}(\mathsf{AId})) \times \mathsf{Code}$$

   A session context is a tuple of two tables and code, (R, U, K), where R records all the actions read for each object, U does the same for actions performed by the session and K is the piece of the remaining code to be executed by the session.
2. We require the following condition on update:

   $$\mathsf{update}_\tau(x, e, f, \sigma, \rho, (R, U, K)) \sqsupseteq \\ (R[x \mapsto R(x) \cup \mathsf{id}_\tau(\sigma)], U[x \mapsto U(x) \cup \{e\}], K).$$

   where $(R', U', K') \sqsupseteq (R'', U'', K'')$ means that

   $$\forall x \in \mathsf{Obj}.\, R'(x) \supseteq R''(x) \wedge U'(x) \supseteq U''(x).$$

PROPOSITION 23. *The following conditions on the* enable *predicate imply corresponding session-guarantee axioms as described:*

1. *For every fair trace τ, the* RYW *axiom holds for the execution generated from τ if*

   $$\forall x, U, \rho.\, \mathsf{enable}(x, (\_, (\_, U, \_)), (\_, \_, \rho, \_, \_)) \\ \implies U(x) \subseteq \mathsf{id}_{\mathsf{type}(x)}(\rho(x)).$$

2. *For every fair trace τ, the* MR *axiom holds for the execution generated from τ if*

   $$\forall x, R, \rho.\, \mathsf{enable}(x, (\_, (R, \_, \_)), (\_, \_, \rho, \_, \_)) \\ \implies R(x) \subseteq \mathsf{id}_{\mathsf{type}(x)}(\rho(x)).$$

3. *For every fair trace τ, the* WFRA *axiom holds for the execution generated from τ if*

   $$\forall x, R, \rho.\, \mathsf{enable}(x, (\_, (R, \_, \_)), (\_, \_, \rho, \_, \_)) \\ \implies R(x) \subseteq \mathsf{id}_{\mathsf{type}(x)}(\rho(x)).$$

4. *For every fair trace τ, the* MWA *axiom holds for the execution generated from τ if*

   $$\forall x, U, \rho.\, \mathsf{enable}(x, (\_, (\_, U, \_)), (\_, \_, \rho, \_, \_)) \\ \implies U(x) \subseteq \mathsf{id}_{\mathsf{type}(x)}(\rho(x)).$$

Proof: Consider a fair trace

$$\tau = \mathcal{S}_0|\mathcal{D}_0 \xrightarrow{\iota_1} \mathcal{S}_1|\mathcal{D}_1 \xrightarrow{\iota_2} \dots \xrightarrow{\iota_n} \mathcal{S}_n|\mathcal{D}_n.$$

Let $(A, \mathsf{so}, \mathsf{vis}, \mathsf{ar})$ be the execution constructed from this trace τ according to our recipe. We go through each item of the list in the proposition, and show that if the condition in the item holds, this execution satisfies the axiom mentioned in the item.

Let us start with the first item. Suppose that

$$\forall x, U, \rho.\, \mathsf{enable}(x, (\_, (\_, U, \_)), (\_, \_, \rho, \_, \_)) \\ \implies U(x) \subseteq \mathsf{id}_{\mathsf{type}(x)}(\rho(x)).$$

Pick $a, b \in A$ such that

$$a \xrightarrow{\mathsf{soo}} b.$$

By the definition of soo,

$$(a \xrightarrow{\mathsf{so}} b) \wedge (\mathsf{obj}(a) = \mathsf{obj}(b)).$$

We unpack the definition of so in the first conjunct:

$$\exists k, l.\, k < l \wedge (\_, a) = \iota_k \wedge (\_, b) = \iota_l \wedge \mathsf{ses}(a) = \mathsf{ses}(b).$$

Then, there are

$$\theta \in \mathcal{S}_{l-1} \;\text{ and }\; \delta \in \mathcal{D}_{l-1}$$

such that the OpS rule is applied for θ and δ at the l step in the trace τ. We note one simple consequence:

$$(\mathsf{ses}(a), \_) = (\mathsf{ses}(b), \_) = \theta.$$

Let U and ρ be the write table and the object table of θ and δ, respectively:

$$(\_, (\_, U, \_)) = \theta \wedge (\_, \_, \rho, \_, \_) = \delta.$$

The rules in our abstract implementation only increase sets stored in the U tables of all sessions. Furthermore, the U table of a session contains the ids of all the operations performed in the session. These two facts imply that

$$\mathsf{id}(a) \in U(\mathsf{obj}(a)) = U(\mathsf{obj}(b)).$$

Since the OpS rule is applied at the l step, the following check by the enable predicate should hold:

$$\mathsf{enable}(\mathsf{obj}(b), \theta, \delta).$$

Hence, by our supposition on the predicate,

$$\mathsf{id}(a) \in U(\mathsf{obj}(b)) \subseteq \mathsf{id}_{\mathsf{type}(\mathsf{obj}(b))}(\rho(\mathsf{obj}(b))).$$

Recall that the definition of the vis relation says that

$$\forall a' \in A.\ (\mathsf{id}(a') \in \mathsf{id}_{\mathsf{type}(\mathsf{obj}(b))}(\rho(\mathsf{obj}(b)))) \implies (a' \xrightarrow{\mathsf{vis}} b).$$

Since $\mathsf{id}(a) \in \mathsf{id}_{\mathsf{type}(\mathsf{obj}(b))}(\rho(\mathsf{obj}(b)))$, we have $(a \xrightarrow{\mathsf{vis}} b)$, as desired.

Next, we prove the item about the MR axiom. Suppose that

$$\forall x, R, \rho.\, \mathsf{enable}(x, (\_, (R, \_, \_)), (\_, \_, \rho, \_, \_)) \implies R(x) \subseteq \mathsf{id}_{\mathsf{type}(x)}(\rho(x)).$$

Consider actions $a, b, c \in A$ such that

$$a \xrightarrow{\mathsf{vis}} b \xrightarrow{\mathsf{soo}} c.$$

By the definition of the soo relation, there are k, l such that

$$k < l \wedge (\_, b) = \iota_k \wedge (\_, c) = \iota_l \wedge \mathsf{ses}(b) = \mathsf{ses}(c) \wedge \mathsf{obj}(b) = \mathsf{obj}(c).$$

Let θ_k and δ_k be the session in $\mathcal{S}_{k-1}$ and the replica in $\mathcal{D}_{k-1}$ used in the k-th step of our trace. Similarly, let θ_l and δ_l be the session in $\mathcal{S}_{l-1}$ and the replica in $\mathcal{D}_{l-1}$ engaged in the l-th step of our trace. Let R_k, R_l and ρ_k, ρ_l be tables such that

$$(\_, (R_k, \_, \_)) = \theta_k \wedge (\_, \_, \rho_k, \_, \_) = \delta_k \wedge (\_, (R_l, \_, \_)) = \theta_l \wedge (\_, \_, \rho_l, \_, \_) = \delta_l.$$

By our supposition on the enable predicate,

$$R_l(\mathsf{obj}(c)) \subseteq \mathsf{id}_{\mathsf{type}(\mathsf{obj}(c))}(\rho_l(\mathsf{obj}(c))).$$

Hence, to complete this case, we only need to show that

$$\mathsf{id}(a) \in R_l(\mathsf{obj}(c)),$$

because every action with its id in $\rho_l(\mathsf{obj}(c))$ becomes related to c by the vis relation. Since $a \xrightarrow{\mathsf{vis}} b$, by the definition of the OpS rule,

$$\mathsf{id}(a) \in \mathsf{id}_{\mathsf{type}(\mathsf{obj}(b))}(\rho_k(\mathsf{obj}(b))).$$

This means that a gets included in the $\mathsf{obj}(b)$ entry of the R table of the session $\mathsf{ses}(b)$ after the k-th step. But $\mathsf{ses}(b) = \mathsf{ses}(c)$ and the R table of each session only grows by the rules of our abstract implementation. Hence, $\mathsf{id}(a) \in R_l(\mathsf{obj}(b))$. Since $\mathsf{obj}(b) = \mathsf{obj}(c)$, we get the desired membership of a.

We move on to the WFRA axiom. Suppose that

$$\forall x, R, \rho.\, \mathsf{enable}(x, (\_, (R, \_, \_)), (\_, \_, \rho, \_, \_)) \implies R(x) \subseteq \mathsf{id}_{\mathsf{type}(x)}(\rho(x)).$$

Consider actions $a, b, c \in A$ such that

$$a \xrightarrow{\mathsf{vis}} b(\xrightarrow{\mathsf{soo}})^* c.$$

We should show that $a \xrightarrow{\mathsf{ar}} c$. That is,

$$\mathsf{id}(a) < \mathsf{id}(c) \wedge \mathsf{obj}(a) = \mathsf{obj}(c).$$

By the definition of vis and Lemma 18,

$$\mathsf{obj}(a) = \mathsf{obj}(b).$$

Also, by the definition of soo,

$$\mathsf{obj}(b) = \mathsf{obj}(c).$$

Hence, $\mathsf{obj}(a) = \mathsf{obj}(c)$. It remains to show that

$$\mathsf{id}(a) < \mathsf{id}(c).$$

Let k, l be indices of the trace τ such that

$$k \leq l \wedge (\_, b) = \iota_k \wedge (\_, c) = \iota_l.$$

Also, let

$$(\_, (R, \_, \_)) = \theta_l \in \mathcal{S}_{l-1} \;\text{ and }\; (\_, \_, \rho, \_, \_) = \delta_l \in \mathcal{D}_{l-1}$$

be the session and the replica that get transformed by the l-th step of the trace τ. Since the OpS rule is applied on these θ_l and δ_l with respect to the object $\mathsf{obj}(c)$, we should have

$$\mathsf{enable}(\mathsf{obj}(c), \theta_l, \delta_l).$$

Because of our supposition on the enable predicate, this implies that

$$R(\mathsf{obj}(c)) \subseteq \mathsf{id}_{\mathsf{type}(\mathsf{obj}(c))}(\rho_l(\mathsf{obj}(c))).$$

Note that since all configurations in τ respect time (Lemma 19) and $\mathsf{id}(c)$ is the timestamp of δ_l, the above subset relationship entails that

$$\forall a' \in R(\mathsf{obj}(c)).\, \mathsf{id}(a') < \mathsf{id}(c). \tag{10}$$

Since b and c are related by soo, they should have the same session id. This means that $R(\mathsf{obj}(c))$ includes the ids of all actions read in the k-th step, which has produced the action b. Since $a \xrightarrow{\mathsf{vis}} b$, the action a is one of those read actions, so its id should be in $R(\mathsf{obj}(c))$. Now from (10), we can infer that $\mathsf{id}(a) < \mathsf{id}(c)$, as desired.

Finally, we prove the item on the MWA axiom. Suppose that

$$\forall x, U, \rho.\, \mathsf{enable}(x, (\_, (\_, U, \_)), (\_, \_, \rho, \_, \_)) \implies U(x) \subseteq \mathsf{id}_{\mathsf{type}(x)}(\rho(x)).$$

Consider $a, b \in A$ such that

$$a \xrightarrow{\mathsf{soo}} b.$$

By the definition of soo, there exist indices k, l of the trace τ such that

$$k < l \wedge (\_, a) = \iota_k \wedge (\_, b) = \iota_l \wedge \mathsf{ses}(a) = \mathsf{ses}(b) \wedge \mathsf{obj}(a) = \mathsf{obj}(b).$$

Let

$$(\_, (\_, U, \_)) = \theta_l \in \mathcal{S}_{l-1} \text{ and } (\_, \_, \rho, \_, \_) = \delta_l \in \mathcal{D}_{l-1}$$

be the session and the replica that are transformed by the l-th step of the trace τ. Since the OPS rule applies to these session and replica, we should have that

$$\mathsf{enable}(\mathsf{obj}(b), \theta_l, \delta_l).$$

By the supposition on the enable predicate, this implies that

$$U(\mathsf{obj}(b)) \subseteq \rho(\mathsf{obj}(b)).$$

Furthermore, since all configurations in the trace τ respect time (Lemma 19) and the timestamp of δ_l is $\mathsf{id}(b)$, we have that

$$\forall a' \in U(\mathsf{obj}(b)).\, \mathsf{id}(a') < \mathsf{id}(b).$$

Recall that all the rules in our implementation only increase the sets stored in the U part of each session, and they store all actions performed by the session in its U component. These properties and the fact that $\mathsf{ses}(a) = \mathsf{ses}(b)$ imply that

$$a \in U(\mathsf{obj}(a)).$$

Since $\mathsf{obj}(a) = \mathsf{obj}(b)$, this in turn entails $a \in U(\mathsf{obj}(b))$. Hence,

$$\mathsf{id}(a) < \mathsf{id}(b),$$

from which follows the desired $a \xrightarrow{\mathsf{ar}} b$. □

PROPOSITION 24. *The following conditions on the* enable *predicate imply corresponding session-guarantee axioms as described:*

1. *For every fair trace τ, the* WFRV *axiom holds for the execution generated from τ if*

$$(\forall x, e, f, \sigma, \rho, \kappa.\, \mathsf{id}_{\mathsf{type}(x)}(\sigma) \subseteq \mathsf{depend}_{\mathsf{type}(x)}(e, f, \sigma, \rho, \kappa)) \wedge (\forall x, R, \rho.\, \mathsf{enable}(x, (\_, (R, \_, \_)), (\_, \_, \rho, \_, \_)) \implies R(x) \subseteq \mathsf{id}_{\mathsf{type}(x)}(\rho(x))).$$

2. *For every fair trace τ, the* MWV *axiom holds for the execution generated from τ if*

$$(\forall x, e, f, \sigma, \rho, \kappa.\, \mathsf{id}_{\mathsf{type}(x)}(\sigma) \subseteq \mathsf{depend}_{\mathsf{type}(x)}(e, f, \sigma, \rho, \kappa)) \wedge (\forall x, U, \rho.\, \mathsf{enable}(x, (\_, (\_, U, \_)), (\_, \_, \rho, \_, \_)) \implies U(x) \subseteq \mathsf{id}_{\mathsf{type}(x)}(\rho(x))).$$

Proof: Consider a fair trace

$$\tau = \mathcal{S}_0|\mathcal{D}_0 \xrightarrow{\iota_1} \mathcal{S}_1|\mathcal{D}_1 \xrightarrow{\iota_2} \ldots \xrightarrow{\iota_n} \mathcal{S}_n|\mathcal{D}_n.$$

Let $(A, \mathsf{so}, \mathsf{vis}, \mathsf{ar})$ be an execution generated by this trace τ. We go through both items in the proposition, and show that if the condition in the item holds, this generated execution satisfies the axiom mentioned in the item.

First, we prove the case of the WFRV axiom. Suppose that

$$(\forall x, e, f, \sigma, \rho, \kappa.\, \mathsf{id}_{\tau}(\sigma) \subseteq \mathsf{depend}_{\mathsf{type}(x)}(e, f, \sigma, \rho, \kappa)) \wedge (\forall x, R, \rho.\, \mathsf{enable}(x, (\_, (R, \_, \_)), (\_, \_, \rho, \_, \_)) \implies R(x) \subseteq \mathsf{id}_{\mathsf{type}(x)}(\rho(x))).$$

Consider actions $a, b, c, d \in A$ such that

$$a \xrightarrow{\mathsf{vis}} b (\xrightarrow{\mathsf{soo}})^* c \xrightarrow{\mathsf{vis}} d$$

Let i, j, k, l be indices of τ and W_i, W_j, W_k, W_l sets of actions such that

$$(W_i, a) = \iota_i \wedge (W_j, b) = \iota_j \wedge (W_k, c) = \iota_k \wedge (W_l, d) = \iota_l.$$

We should show $a \xrightarrow{\mathsf{vis}} d$, equivalently,

$$\mathsf{id}(a) \in W_l.$$

Since $a \xrightarrow{\mathsf{vis}} b$ and $c \xrightarrow{\mathsf{vis}} d$,

$$(\mathsf{id}(a) \in W_j) \wedge (\mathsf{id}(c) \in W_l).$$

Hence, it suffices to show that

$$W_j \subseteq W_k \wedge W_k \subseteq W_l. \tag{11}$$

Let

$$(\_, (R_k, \_, \_)) = \theta_k \in \mathcal{S}_{k-1} \text{ and } (\_, \_, \rho_k, \_, \_) = \delta_k \in \mathcal{D}_{k-1}$$

be the session and the replica that are updated by the OPS rule at the k-th step of τ. By the definition of the rule, we should have

$$\mathsf{enable}(\mathsf{obj}(c), \theta_k, \delta_k)$$

which by our supposition implies that

$$R_k(\mathsf{obj}(c)) \subseteq \mathsf{id}_{\mathsf{type}(\mathsf{obj}(c))}(\rho_k(\mathsf{obj}(c))). \tag{12}$$

The set $R_k(\mathsf{obj}(c))$ includes the ids of all the actions on $\mathsf{obj}(c)$ that have been read up to the $(k-1)$-th step. Since $\mathsf{ses}(b) = \mathsf{ses}(c)$ and $\mathsf{obj}(b) = \mathsf{obj}(c)$, this implies that

$$W_j \subseteq R_k(\mathsf{obj}(c)).$$

Furthermore, $\rho_k(\mathsf{obj}(c)) = W_k$, so we have

$$W_j \subseteq W_k,$$

which is the first conjunct of our proof obligation (11). It remains to show the second conjunct in (11). Let

$$(\_, \_, \rho_l, \_, \_) = \delta_l \in \mathcal{D}_{l-1}$$

be the replica that was used to create the action d in the l-th step. Then,

$$W_l = \mathsf{id}_{\mathsf{type}(\mathsf{obj}(d))}(\rho_l(\mathsf{obj}(d))).$$

Since $c \xrightarrow{\mathsf{vis}} d$, we have that $k < l$. Also, by the VISWF axiom (Proposition 21),

$$\mathsf{obj}(c) = \mathsf{obj}(d).$$

If δ_l and δ_k are about the same replica,

$$\begin{aligned} W_k &= \mathsf{id}_{\mathsf{type}(\mathsf{obj}(c))}(\rho_k(\mathsf{obj}(c))) \\ &= \mathsf{id}_{\mathsf{type}(\mathsf{obj}(d))}(\rho_k(\mathsf{obj}(d))) \\ &\subseteq \mathsf{id}_{\mathsf{type}(\mathsf{obj}(d))}(\rho_l(\mathsf{obj}(d))) = W_l, \end{aligned}$$

because the ids associated with object tables only grow by the rules in our abstract implementation. Suppose that δ_l and δ_k are configurations of different replicas. Recall that $\mathsf{id}(c) \in W_l$ because $c \xrightarrow{\mathsf{vis}} d$. In order for this to happen, a message

$$(x', \mathsf{id}(c), f', W', E')$$

for some x', f', W', E' must have been incorporated into the replica ρ_l by the application of the OPD rule before the l-th step. But the only message with the action id $\mathsf{id}(c)$ in the trace is the one generated by the k-th step in the trace τ. Hence,

$$\begin{aligned} E' &= \mathsf{depend}_{\mathsf{type}(\mathsf{obj}(c))}(\mathsf{id}(c), \_, \rho_k(\mathsf{obj}(c)), \_, \_) \\ &\supseteq \mathsf{id}_{\mathsf{type}(\mathsf{obj}(c))}(\rho_k(\mathsf{obj}(c))) \\ &= W_k. \end{aligned}$$

When the OPD rule was applied, it must have shown that every action id in W_k was incorporated into the $\mathsf{obj}(c)$ entry of the object table in an updated replica. Since the set of actions associated with the object table of a replica only grows and $\mathsf{obj}(c) = \mathsf{obj}(d)$, we can conclude that

$$W_k \subseteq \mathsf{id}_{\mathsf{type}(\mathsf{obj}(d))}(\rho_l(\mathsf{obj}(d))) = W_l,$$

as desired.

Next, we prove the item on the WMV axiom. Suppose that

$$\begin{aligned} &(\forall x, e, f, \sigma, \rho, \kappa.\, \mathsf{id}_\tau(\sigma) \subseteq \mathsf{depend}_{\mathsf{type}(x)}(e, f, \sigma, \rho, \kappa)) \\ &\wedge (\forall x, U, \rho.\, \mathsf{enable}(x, (\_, (\_, U, \_)), (\_, \_, \rho, \_, \_)) \\ &\qquad \Longrightarrow U(x) \subseteq \mathsf{id}_{\mathsf{type}(x)}(\rho(x))). \end{aligned}$$

Consider $a, b, c \in A$ such that

$$a \xrightarrow{\mathsf{soo}} b \xrightarrow{\mathsf{vis}} c.$$

Then,

$$\mathsf{ses}(a) = \mathsf{ses}(b) \wedge \mathsf{obj}(a) = \mathsf{obj}(b).$$

Let i, j, k be indices of the trace τ and W_i, W_j, W_k sets of action ids such that

$$(W_i, a) = \iota_i \wedge (W_j, b) = \iota_j \wedge (W_k, c) = \iota_k.$$

We should show that

$$\mathsf{id}(a) \in W_k.$$

Let

$$(\_, (\_, U_j, \_)) = \theta_j \in \mathcal{S}_{j-1} \text{ and } (\_, \_, \rho_j, \_, \_) = \delta_j \in \mathcal{D}_{j-1}$$

be the session and the replica involved in the application of the OPS rule in the j-th step. By our supposition,

$$U_j(\mathsf{obj}(b)) \subseteq \mathsf{id}_{\mathsf{type}(\mathsf{obj}(b))}(\rho_j(\mathsf{obj}(b))) = W_j.$$

Since $a \xrightarrow{\mathsf{soo}} b$ and U_j stores all the updates on the session θ_j,

$$\mathsf{id}(a) \in U_j(\mathsf{obj}(b)) \subseteq W_j.$$

Hence, we can discharge our proof obligation simply by showing that

$$W_j \subseteq W_k.$$

Let

$$(\_, (\_, U_k, \_)) = \theta_j \in \mathcal{S}_{k-1} \text{ and } (\_, \_, \rho_k, \_, \_) = \delta_k \in \mathcal{D}_{k-1}$$

be the session and the replica involved in the application of the OPS rule in the k-th step. Since $b \xrightarrow{\mathsf{vis}} c$,

$$(k < l) \wedge (\mathsf{obj}(b) = \mathsf{obj}(c)).$$

If δ_l and δ_k are configurations of the same replica, we have

$$\begin{aligned} W_l &= \mathsf{id}_{\mathsf{type}(\mathsf{obj}(b))}(\rho_l(\mathsf{obj}(b))) \\ &\subseteq \mathsf{id}_{\mathsf{type}(\mathsf{obj}(c))}(\rho_k(\mathsf{obj}(c))) = W_k, \end{aligned}$$

because the set of action ids associated with the object table of a replica only grows by the rules of our abstract implementation. Suppose that δ_l and δ_k are configurations of different replicas. In order for $\mathsf{id}(b)$ to appear in W_k, a message of the form

$$(x', \mathsf{id}(b), f', W', E')$$

for some x', f', W', E' must have been incorporated into the replica δ_k before the k-th step. But if a message stores the

action id $\mathsf{id}(b)$, it should be the one generated by the l-th step. Hence,

$$\begin{aligned} E' &= \mathsf{depend}_{\mathsf{type}(\mathsf{obj}(b))}(\mathsf{id}(b), \_, \rho_l(\mathsf{obj}(b)), \_, \kappa) \\ &\supseteq \mathsf{id}_{\mathsf{type}(\mathsf{obj}(b))}(\rho_l(\mathsf{obj}(b))). \\ &= W_l. \end{aligned}$$

Recall that whenever a message gets incorporated by our OPD rule, we check that every action ids in the message's E part appear in the corresponding entry of the object table of a replica updated by the rule. Furthermore, the action ids associated with any entry in the object table of a replica only increase by the rules of our abstract implementation. Hence,

$$\begin{aligned} W_l &\subseteq E' \\ &\subseteq \mathsf{id}_{\mathsf{type}(\mathsf{obj}(b))}(\rho_k(\mathsf{obj}(b))) \\ &= \mathsf{id}_{\mathsf{type}(\mathsf{obj}(c))}(\rho_k(\mathsf{obj}(c))) = W_k, \end{aligned}$$

which is the conclusion that we are looking for. □

B.6 Correspondence: Causality Axioms

PROPOSITION 25. *For every fair trace τ, the* COCV *and the* COCA *axioms hold for the execution generated from the trace τ if*

$$\begin{aligned} &(\forall x, e, f, \sigma, \rho, R, U, K. \\ &\qquad \mathsf{id}_{\mathsf{type}(x)}(\sigma) \cup \bigcup_{y \in \mathsf{Obj}} U(y) \\ &\qquad\qquad \subseteq \mathsf{depend}_{\mathsf{type}(x)}(e, f, \sigma, \rho, (R, U, K))) \wedge \\ &(\forall U, \rho.\ \mathsf{enable}(\_, (\_, (\_, U, \_)), (\_, \_, \rho, \_, \_)) \\ &\qquad \implies \forall x. U(x) \subseteq \mathsf{id}_{\mathsf{type}(x)}(\rho(x))) \end{aligned}$$

Proof: Consider a fair trace

$$\tau = \mathcal{S}_0|\mathcal{D}_0 \xrightarrow{\iota_1} \mathcal{S}_1|\mathcal{D}_1 \xrightarrow{\iota_2} \ldots \xrightarrow{\iota_n} \mathcal{S}_n|\mathcal{D}_n.$$

Let $(A, \mathsf{so}, \mathsf{vis}, \mathsf{ar})$ be an execution generated by this trace τ. We need to show that the execution satisfies the COCV and the COCA axioms.

For each binary relation r on A, we say that a set X of action ids is ***r*-closed** if

$$\forall a, b \in A.\, (\mathsf{id}(b) \in X \wedge (a, b) \in r \implies \mathsf{id}(a) \in X).$$

The key observation behind our proof is that every configuration $\mathcal{S}_i|\mathcal{D}_i$ in the trace τ satisfies the following property:

$$\begin{aligned} &(\forall (\_, \_, \rho, \_, \_) \in \mathcal{D}_i.\ \bigcup_{x \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(x)}(\rho(x)) \text{ is hb-closed}) \wedge \\ &(\forall (\_, \_, \_, M, N) \in \mathcal{D}_i. \forall a, a' \in A. \forall (\_, \mathsf{id}(a), \_, \_, E) \in M \cup N. \\ &\qquad (a', a) \in (\mathsf{vis} \cup \mathsf{so}) \implies \mathsf{id}(a') \in E). \end{aligned}$$

This property can be proven by induction on i. When $i = 0$, the property holds because $\mathcal{S}_0|\mathcal{D}_0$ is empty. Now consider the case that $i > 0$, and suppose that the property holds for all $j < i$. We do the case analysis on the rule used in the i-th step. If the rule is PROP, no new messages are generated, and the object tables of replicas remain unchanged by the rule. Hence, the property holds by induction hypothesis. If the rule is OPD, the situation is similar to the previous case. The only difference lies in one replica that gets affected by the rule. Let

$$\delta = (\_, \_, \rho, \_, \_) \text{ and } m = (x, e, \_, \_, E)$$

be a replica's configuration and the message that get used by the OPD rule in the i-th step. Also, let σ' be the new state of a data type computed by OPD, and a an action in A that has e as its id. By induction hypothesis, the set

$$\bigcup_{y \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(y)}(\rho(y))$$

is hb-closed, and

$$\forall a' \in A.\, (a', a) \in (\mathsf{vis} \cup \mathsf{so}) \implies \mathsf{id}(a') \in E.$$

Furthermore, by the definition of OPD,

$$E \subseteq \bigcup_{y \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(y)}(\rho(y)).$$

By the condition on eval and the definition of OPD,

$$\mathsf{id}_{\mathsf{type}(x)}(\sigma') = \mathsf{id}_{\mathsf{type}(x)}(\rho(x)) \cup \{e\}.$$

Recall that $\mathsf{hb} = (\mathsf{so} \cup \mathsf{vis})^+$. From what we have proved follows that the below set is hb-closed as well:

$$\bigcup_{y \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(y)}(\rho[x \mapsto \sigma'](y)).$$

The remaining case is OPS. Let

$$\theta = (\_, (\_, U, \_)), \text{ and } \delta = (d, t, \rho, \_, \_)$$

be the configurations of the session and the replica affected by the OPS rule. Let

$$m = (x, e, f, W', E') \text{ and } a$$

be the message and the action generated by the rule. Note that $\mathsf{id}(a) = e$. By the definition of the rule,

$$\exists y.\, \mathsf{enable}(y, \theta, \delta).$$

But because of the assumption in the proposition, this implies that

$$\forall y.\, U(y) \subseteq \mathsf{id}_{\mathsf{type}(y)}(\rho(y)). \tag{13}$$

Also, because of the assumption on depend in the proposition,

$$\mathsf{id}_{\mathsf{type}(x)}(\rho(x)) \cup \bigcup_{y \in \mathsf{Obj}} U(y) \subseteq E'.$$

This implies that

$$\forall a' \in A.\, (a', a) \in (\mathsf{vis} \cup \mathsf{so}) \implies \mathsf{id}(a') \in E'.$$

Furthermore, from what we have just shown, the condition on eval and the definition of OPS, it follows that the set of action ids associated with the updated replica is the following set:

$$Y = \{(t, d)\} \cup \bigcup_{y \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(y)}(\rho(y)).$$

By the definition of vis, the ids of actions in $\mathsf{vis}^{-1}(a)$ are included in the second argument of the above union. Also, by (13), those of actions in $\mathsf{so}^{-1}(a)$ are also included in the second argument. These two facts, the definition of hb and the hb-closedness of the second argument imply that the set Y is also hb-closed, as desired.

Let us go back to the proof that the execution $(A, \mathsf{so}, \mathsf{vis}, \mathsf{ar})$ satisfies the COCV and the COCA axioms. We consider the COCV axiom first. Consider $a, c \in A$ such that

$$(a \xrightarrow{\mathsf{hb}} c) \wedge \mathsf{obj}(a) = \mathsf{obj}(c).$$

By the definition of hb, there exists $b \in A$ such that

$$(a(\xrightarrow{\mathsf{hb}})^* b \xrightarrow{\mathsf{so}} c) \vee (a(\xrightarrow{\mathsf{hb}})^* b \xrightarrow{\mathsf{vis}} c). \tag{14}$$

Let i, j, k, W_i, W_j, W_k be indices and sets of action ids such that

$$(W_i, a) = \iota_i \wedge (W_j, b) = \iota_j \wedge (W_k, c) = \iota_k.$$

Let

$$\theta_k = (\_, (\_, U, \_)) \text{ and } \delta_k = (d, t, \rho, \_, \_)$$

be the configurations of the session and the replica used by the k-th step of the trace τ. Assume that the first disjunct of (14) holds. Then,

$$(j < k) \wedge (\mathsf{id}(b) \in \bigcup_{x \in \mathsf{Obj}} U(x)).$$

By the assumption in the proposition,

$$\bigcup_{x \in \mathsf{Obj}} U(x) \subseteq \bigcup_{x \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(x)}(\rho(x)).$$

Hence,

$$\mathsf{id}(b) \in \bigcup_{x \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(x)}(\rho(x)).$$

By what we proved about the hb closedness in the first half of this proof, the set $\bigcup_{x \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(x)}(\rho(x))$ is hb-closed. Hence,

$$\mathsf{id}(a) \in \bigcup_{x \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(x)}(\rho(x)).$$

By Lemma 18,

$$\mathsf{id}(a) \in \mathsf{id}_{\mathsf{type}(\mathsf{obj}(a))}(\rho(\mathsf{obj}(a))) = W_k.$$

Hence, $a \xrightarrow{\mathsf{vis}} c$ as desired. Now assume that the second disjunct of (14) holds. Then,

$$\mathsf{id}(b) \in \mathsf{id}_{\mathsf{type}(\mathsf{obj}(c))}(\rho(\mathsf{obj}(c))) = W_k.$$

By what we proved about the hb closedness in the first half of this proof, the set

$$\bigcup_{x \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(x)}(\rho(x))$$

is hb-closed. Hence,

$$\mathsf{id}(a) \in \bigcup_{x \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(x)}(\rho(x)).$$

But $\mathsf{obj}(a) = \mathsf{obj}(c)$, so by Lemma 18,

$$\mathsf{id}(a) \in \mathsf{id}_{\mathsf{type}(\mathsf{obj}(c))}(\rho(\mathsf{obj}(c))) = W_k,$$

from which follows $a \xrightarrow{\mathsf{vis}} c$ as desired.

Next, we handle the COCA axiom. Define a total order $<_\tau$ on all actions in A as follows:

$$a <_\tau b \iff \mathsf{id}(a) < \mathsf{id}(b).$$

This is an irreflexive transitive total order because so is $<$ on action ids and different actions in A have different action ids (Lemma 16). We will show that

$$\forall a, b \in A.\, (a \xrightarrow{\mathsf{ar}} b \vee a \xrightarrow{\mathsf{vis}} b \vee a \xrightarrow{\mathsf{so}} b) \implies a <_\tau b. \tag{15}$$

Then, from the transitivity and irreflexivity of $<_\tau$ follows that

$$(\mathsf{ar} \cup \mathsf{vis} \cup \mathsf{so})^+ = (\mathsf{ar} \cup \mathsf{hb})^+$$

is irreflexive, as desired. Pick $a, b \in A$ such that

$$a \xrightarrow{\mathsf{ar}} b \vee a \xrightarrow{\mathsf{vis}} b \vee a \xrightarrow{\mathsf{so}} b.$$

Let i, j, W_i, W_j be indices and sets of action ids such that

$$(W_i, a) = \iota_i \wedge (W_j, b) = \iota_j.$$

If $a \xrightarrow{\mathsf{ar}} b$, then $a <_\tau b$ by the definition of ar. If $a \xrightarrow{\mathsf{vis}} b$,

$$\mathsf{id}(a) \in W_j.$$

Since the configuration $\mathcal{S}_{i-1}|\mathcal{D}_{i-1}$ of the trace right before the i-th step respects time (Lemma 19),

$$\mathsf{id}(a) < \mathsf{id}(b).$$

Hence, $a <_\tau b$. The remaining case is that $a \xrightarrow{\mathsf{so}} b$. Let

$$\theta_j = (\_, (\_, U, \_)) \text{ and } \delta_j = (d, t, \rho, \_, \_)$$

be the configurations of the session and the replica used in the j-th step of the trace τ. Then, since $\mathsf{ses}(a) = \mathsf{ses}(b)$,

$$\mathsf{id}(a) \in \bigcup_{x \in \mathsf{Obj}} U(x).$$

By the assumption of the proposition,

$$\bigcup_{x \in \mathsf{Obj}} U(x) \subseteq \bigcup_{x \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(x)}(\rho(x))$$

Since the configuration $\mathcal{S}_{i-1}|\mathcal{D}_{i-1}$ of the trace right before the i-th step respects time (Lemma 19),

$$\forall a' \in \bigcup_{x \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(x)}(\rho(x)).\, \mathsf{id}(a') < (t, d) = \mathsf{id}(b).$$

From what we have proven so far, it follows that

$$\mathsf{id}(a) < \mathsf{id}(b),$$

which gives $a <_\tau b$ as desired. □

B.7 Correspondence: On-demand Cross Object Causality

We assume that each operation f is annotated with $\mu \in \{\mathsf{ORD}, \mathsf{CSL}\}$, as explained in the main text of the paper. Also, we assume the following two operations for each data type τ:

$$\mathsf{idc}_\tau : \mathsf{St}_\tau \to \mathcal{P}_{\mathsf{fin}}(\mathsf{AId}), \qquad \mathsf{ido}_\tau : \mathsf{St}_\tau \to \mathcal{P}_{\mathsf{fin}}(\mathsf{AId})$$

such that for every τ and all $\sigma, \sigma' \in \mathsf{St}_\tau$,

$$\begin{aligned}&(\mathsf{idc}_\tau(\sigma) \cap \mathsf{ido}_\tau(\sigma) = \emptyset \wedge \mathsf{idc}_\tau(\sigma) \cup \mathsf{ido}_\tau(\sigma) = \mathsf{id}_\tau(\sigma)) \wedge \\ &(\forall e, f_\mu, W.\, \mathsf{eval}_\tau(e, f_\mu, W, \sigma) = (\_, \sigma') \implies \\ &\quad ((\mu = \mathsf{ORD} \implies \mathsf{ido}_\tau(\sigma') = \mathsf{ido}_\tau(\sigma) \cup \{e\}) \wedge \\ &\quad (\mu = \mathsf{CSL} \implies \mathsf{ido}_\tau(\sigma') = \mathsf{ido}_\tau(\sigma) \cup \{e\})).\end{aligned}$$

PROPOSITION 26. *For every fair trace τ, the on-demand-causality versions of the* COCV *and the* COCA *axioms hold for the execution generated from the trace τ if*

$$\begin{aligned}&(\forall x, e, f_\mu, \sigma, \rho, R, U, K. \\ &\qquad \mathsf{idc}_{\mathsf{type}(x)}(\sigma) \cup \bigcup_{y \in \mathsf{Obj}} U(y) \\ &\qquad\qquad \subseteq \mathsf{depend}_{\mathsf{type}(x)}(e, f, \sigma, \rho, (R, U, K))) \wedge \\ &(\forall U, \rho.\, \mathsf{enable}(\_, (\_, (\_, U, \_)), (\_, \_, \rho, \_, \_)) \\ &\qquad \implies \forall x.\, U(x) \subseteq \mathsf{id}_{\mathsf{type}(x)}(\rho(x))).\end{aligned}$$

Proof: The proof is almost identical to that of Proposition 25, except a few changes due to a new definition of hb. We repeat the proof of Proposition 25 here but with these required changes. Consider a fair trace

$$\tau = \mathcal{S}_0|\mathcal{D}_0 \xrightarrow{\iota_1} \mathcal{S}_1|\mathcal{D}_1 \xrightarrow{\iota_2} \dots \xrightarrow{\iota_n} \mathcal{S}_n|\mathcal{D}_n.$$

Let $(A, \mathsf{so}, \mathsf{vis}, \mathsf{ar})$ be an execution generated by this trace τ. We need to show that the execution satisfies the versions of the COCV and the COCA axioms for the new definition of hb.

We remind the reader that for each binary relation r on A, a set X of action ids is said to be ***r*-closed** if

$$\forall a, b \in A.\, (\mathsf{id}(b) \in X \wedge (a, b) \in r \implies \mathsf{id}(a) \in X).$$

The key observation behind our proof is that every configuration $\mathcal{S}_i|\mathcal{D}_i$ in the trace τ satisfies the following property:

$$\begin{aligned}&(\forall (\_, \_, \rho, \_, \_) \in \mathcal{D}_i.\ \bigcup_{x \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(x)}(\rho(x)) \text{ is } \mathsf{hb}\text{-closed}) \wedge \\ &(\forall (\_, \_, \_, M, N) \in \mathcal{D}_i. \forall a, a' \in A. \forall (\_, \mathsf{id}(a), f_\mu, \_, E) \in M \cup N. \\ &\qquad (a', a) \in (\mathsf{so} \cup \mathsf{cvis}) \implies \mathsf{id}(a') \in E)\end{aligned}$$

Note that hb here is $(\mathsf{so} \cup \mathsf{cvis})^+$, not $(\mathsf{so} \cup \mathsf{vis})^+$ in Proposition 25. This property can be proven by induction on i. When $i = 0$, the property holds because $\mathcal{S}_0|\mathcal{D}_0$ is empty. Now consider the case that $i > 0$, and suppose that the property holds for all $j < i$. We do the case analysis on the rule used in the i-th step. If the rule is PROP, no new messages are generated, and the object tables of replicas remain unchanged by the rule. Hence, the property holds by induction hypothesis. If the rule is OPD, the situation is similar to the previous case. The only difference lies in one replica that gets affected by the rule. Let

$$\delta = (\_, \_, \rho, \_, \_) \text{ and } m = (x, e, f_\mu, \_, E)$$

be a replica's configuration and the message that get used by the OPD rule in the i-th step. Also, let σ' be the new state of a data type computed by OPD, and a an action in A that has e as its id. By induction hypothesis, the set

$$\bigcup_{y \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(y)}(\rho(y))$$

is hb-closed, and

$$\forall a' \in A.\, (a', a) \in (\mathsf{cvis} \cup \mathsf{so}) \implies \mathsf{id}(a') \in E.$$

Furthermore, by the definition of OPD,

$$E \subseteq \bigcup_{y \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(y)}(\rho(y)).$$

By the condition on eval and the definition of OPD,

$$\mathsf{id}_{\mathsf{type}(x)}(\sigma') = \mathsf{id}_{\mathsf{type}(x)}(\rho(x)) \cup \{e\}.$$

Recall that $\mathsf{hb} = (\mathsf{so} \cup \mathsf{cvis})^+$. From what we have proved follows that the below set is hb-closed as well:

$$\bigcup_{y \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(y)}(\rho[x \mapsto \sigma'](y)).$$

The remaining case is OPS. Let

$$\theta = (\_, (\_, U, \_)), \text{ and } \delta = (d, t, \rho, \_, \_)$$

be the configurations of the session and the replica affected by the OPS rule. Let

$$m = (x, e, f_\mu, W', E') \text{ and } a$$

be the message and the action generated by the rule. Note that $\mathsf{id}(a) = e$. By the definition of the rule,

$$\exists y.\, \mathsf{enable}(y, \theta, \delta).$$

But because of the assumption in the proposition, this implies that

$$\forall y.\, U(y) \subseteq \mathsf{id}_{\mathsf{type}(y)}(\rho(y)). \tag{16}$$

Also, because of the assumption on depend in the proposition,

$$\big(\mathsf{idc}_{\mathsf{type}(x)}(\rho(x)) \cup \bigcup_{y \in \mathsf{Obj}} U(y)\big) \subseteq E'.$$

Since $\mathsf{idc}_{\mathsf{type}(x)}(\rho(x))$ contains the ids of all actions that are cvis-related to a, the above subset relationship implies that

$$\forall a' \in A.\, (a', a) \in (\mathsf{cvis} \cup \mathsf{so}) \implies \mathsf{id}(a') \in E'.$$

Furthermore, from what we have just shown, the condition on eval and the definition of OPS, it follows that the set of action ids associated with the updated replica is the following set:

$$Y = \{(t, d)\} \cup \bigcup_{y \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(y)}(\rho(y)).$$

By the definition of vis, the ids of actions in $\mathsf{vis}^{-1}(a)$ are included in the second argument of the above union. This means that those of actions in $\mathsf{cvis}^{-1}(a)$ should also be included, because $\mathsf{cvis}^{-1}(a) \subseteq \mathsf{vis}^{-1}(a)$. Also, by (16), the ids of actions in $\mathsf{so}^{-1}(a)$ are also included in the second argument. These two facts, the definition of hb and the hb-closedness of the second argument imply that the set Y is also hb-closed, as desired.

Let us go back to the proof that the execution $(A, \mathsf{so}, \mathsf{vis}, \mathsf{ar})$ satisfies the COCV and the COCA axioms. We consider the COCV axiom first. Consider $a, c \in A$ such that

$$(a \xrightarrow{\mathsf{hb}} c) \wedge \mathsf{obj}(a) = \mathsf{obj}(c).$$

By the definition of hb, there exists $b \in A$ such that

$$(a(\xrightarrow{\mathsf{hb}})^* b \xrightarrow{\mathsf{so}} c) \vee (a(\xrightarrow{\mathsf{hb}})^* b \xrightarrow{\mathsf{cvis}} c). \tag{17}$$

Let i, j, k, W_i, W_j, W_k be indices and sets of action ids such that

$$(W_i, a) = \iota_i \wedge (W_j, b) = \iota_j \wedge (W_k, c) = \iota_k.$$

Let

$$\theta_k = (\_, (\_, U, \_)) \text{ and } \delta_k = (d, t, \rho, \_, \_)$$

be the configurations of the session and the replica used by the k-th step of the trace τ. Assume that the first disjunct of (17) holds. Then,

$$(j < k) \wedge (\mathsf{id}(b) \in \bigcup_{x \in \mathsf{Obj}} U(x)).$$

By the assumption in the proposition,

$$\bigcup_{x \in \mathsf{Obj}} U(x) \subseteq \bigcup_{x \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(x)}(\rho(x)).$$

Hence,

$$\mathsf{id}(b) \in \bigcup_{x \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(x)}(\rho(x)).$$

By what we proved about the hb closedness in the first half of this proof, the set $\bigcup_{x \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(x)}(\rho(x))$ is hb-closed. Hence,

$$\mathsf{id}(a) \in \bigcup_{x \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(x)}(\rho(x)).$$

By Lemma 18,

$$\mathsf{id}(a) \in \mathsf{id}_{\mathsf{type}(\mathsf{obj}(a))}(\rho(\mathsf{obj}(a))) = W_k.$$

Hence, $a \xrightarrow{\mathsf{vis}} c$ as desired. Now assume that the second disjunct of (17) holds. Then,

$$\mathsf{id}(b) \in \mathsf{id}_{\mathsf{type}(\mathsf{obj}(c))}(\rho(\mathsf{obj}(c))) = W_k.$$

By what we proved about the hb closedness in the first half of this proof, the set

$$\bigcup_{x \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(x)}(\rho(x))$$

is hb-closed. Hence,

$$\mathsf{id}(a) \in \bigcup_{x \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(x)}(\rho(x)).$$

But $\mathsf{obj}(a) = \mathsf{obj}(c)$, so by Lemma 18,

$$\mathsf{id}(a) \in \mathsf{id}_{\mathsf{type}(\mathsf{obj}(c))}(\rho(\mathsf{obj}(c))) = W_k,$$

from which follows $a \xrightarrow{\mathsf{vis}} c$ as desired.

Next, we handle the COCA axiom. Define a total order $<_\tau$ on all actions in A as follows:

$$a <_\tau b \iff \mathsf{id}(a) < \mathsf{id}(b).$$

This is an irreflexive transitive total order because so is $<$ on action ids and different actions in A have different action ids (Lemma 16). We will show that

$$\forall a, b \in A.\, (a \xrightarrow{\mathsf{ar}} b \vee a \xrightarrow{\mathsf{cvis}} b \vee a \xrightarrow{\mathsf{so}} b) \implies a <_\tau b. \tag{18}$$

Then, from the transitivity and irreflexivity of $<_\tau$ follows that

$$(\mathsf{ar} \cup \mathsf{cvis} \cup \mathsf{so})^+ = (\mathsf{ar} \cup \mathsf{hb})^+$$

is irreflexive, as desired. Pick $a, b \in A$ such that

$$a \xrightarrow{\mathsf{ar}} b \vee a \xrightarrow{\mathsf{cvis}} b \vee a \xrightarrow{\mathsf{so}} b.$$

Let i, j, W_i, W_j be indices and sets of action ids such that

$$(W_i, a) = \iota_i \wedge (W_j, b) = \iota_j.$$

If $a \xrightarrow{\mathsf{ar}} b$, then $a <_\tau b$ by the definition of ar. If $a \xrightarrow{\mathsf{cvis}} b$,

$$\mathsf{id}(a) \in W_j.$$

Since the configuration $\mathcal{S}_{i-1}|\mathcal{D}_{i-1}$ of the trace right before the i-th step respects time (Lemma 19),

$$\mathsf{id}(a) < \mathsf{id}(b).$$

Hence, $a <_\tau b$. The remaining case is that $a \xrightarrow{\mathsf{so}} b$. Let

$$\theta_j = (\_, (\_, U, \_)) \text{ and } \delta_j = (d, t, \rho, \_, \_)$$

be the configurations of the session and the replica used in the j-th step of the trace τ. Then, since $\mathsf{ses}(a) = \mathsf{ses}(b)$,

$$\mathsf{id}(a) \in \bigcup_{x \in \mathsf{Obj}} U(x).$$

By the assumption of the proposition,

$$\bigcup_{x \in \mathsf{Obj}} U(x) \subseteq \bigcup_{x \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(x)}(\rho(x))$$

Since the configuration $\mathcal{S}_{i-1}|\mathcal{D}_{i-1}$ of the trace right before the i-th step respects time (Lemma 19),

$$\forall e' \in \bigcup_{x \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(x)}(\rho(x)).\, e' < (t, d) = \mathsf{id}(b).$$

From what we have proven so far, it follows that

$$\mathsf{id}(a) < \mathsf{id}(b),$$

which gives $a <_\tau b$ as desired. □

B.8 Correspondence: Fence

To accommodate fences in our abstract implementation, we need to allow sessions to issue fence instructions. This amounts to four changes. The first is that the instr function can return not just operations on objects, but also the fence instruction:

$$\mathsf{instr} : \mathsf{Session} \to (\mathsf{Obj} \times \mathsf{Op}) \cup \{\mathsf{fence}\}.$$

The second change is the addition of a new function on sessions:

$$\mathsf{nextCode} : \mathsf{Code} \to \mathsf{Code}.$$

Given a code part K of a session, this function updates the part so that it is ready to execute the next command of K. The third change is the introduction of a new rule for fence:

$$\frac{\begin{array}{l}\delta = (d, t, \rho, M, N) \qquad \delta_i = (d_i, t_i, \rho_i, M_i, N_i) \\ \theta = (s, (R, U, K)) \qquad \mathsf{instr}(\theta) = \mathsf{fence} \\ \forall y \in \mathsf{Obj}.\, U(y) \subseteq \mathsf{id}_{\mathsf{type}(y)}\rho(y) \\ \forall i.\, \forall y \in \mathsf{Obj}.\, \mathsf{id}_{\mathsf{type}(y)}\rho(y) \subseteq \mathsf{id}_{\mathsf{type}(y)}\rho_i(y) \\ a = ((t, d), s, \mathsf{fence}) \qquad \theta' = (s, (R, U, \mathsf{nextCode}(K))) \\ \delta' = (d, \mathsf{next}(t), \rho, M, N) \\ \delta'_i = (d_i, \mathsf{max}(t_i, \mathsf{next}(t)), \rho_i, M_i, N_i)\end{array}}{\mathcal{S}, \theta \mid \delta, \delta_1, \ldots, \delta_n \xrightarrow{a} \mathcal{S}, \theta' \mid \delta', \delta'_1, \ldots, \delta'_n} \textsc{Opf}$$

The fourth change lies in the construction of an execution from a trace. Consider a fair trace:

$$\tau = (\mathcal{S}_0, \mathcal{D}_0, \{(\iota_k, \mathcal{S}_k, \mathcal{D}_k)\}_k).$$

We generate an execution $(A, \mathsf{so}, \mathsf{vis}, \mathsf{ar}, \mathsf{sc})$ from this trace τ as follows:

$$\begin{array}{rl}
A = \{a \mid \exists k.\, (_, a) = \iota_k \vee a = \iota_k\}, & \\
\mathsf{vis} = \{(a, b) \in A \times A \mid \exists k, W.\, (W, b) = \iota_k \wedge \mathsf{id}(a) \in W\}, & \\
\mathsf{so} = \{(a, b) \in A \times A \mid \exists k, l.\, k < l \wedge \mathsf{ses}(a) = \mathsf{ses}(b) & \\
\qquad \wedge ((_, a) = \iota_k \vee a = \iota_k) & \\
\qquad \wedge ((_, b) = \iota_l \vee b = \iota_l)\}, & \\
\mathsf{ar} = \{(a, b) \in A \times A \mid \exists k, l.\, (_, a) = \iota_k \wedge (_, b) = \iota_l & \\
\qquad \wedge \mathsf{id}(a) < \mathsf{id}(b) \wedge \mathsf{obj}(a) = \mathsf{obj}(b)\}, & \\
\mathsf{sc} = \{(a, b) \in A \times A \mid \exists k, l.\, a = \iota_k \wedge b = \iota_l \wedge k < l\}. &
\end{array}$$

The soundness of most axioms can be proved by arguments similar to what we presented so far. Hence, instead of going through all the axioms, we focus on the four most important ones: SCWF, THINAIR, COCV and COCA.

PROPOSITION 27. *Every fair trace generates executions satisfying* SCWF *and* THINAIR.

Proof: Pick a fair trace τ:

$$\tau = \mathcal{S}_0|\mathcal{D}_0 \xrightarrow{\iota_1} \mathcal{S}_1|\mathcal{D}_1 \xrightarrow{\iota_2} \ldots \xrightarrow{\iota_n} \mathcal{S}_n|\mathcal{D}_n$$

where $n \in \mathbb{N} \cup \{\omega\}$. Let $(A, \mathsf{vis}, \mathsf{so}, \mathsf{ar}, \mathsf{sc})$ be an execution constructed from this trace. We will show that this execution satisfies SCWF and THINAIR.

Firstly, we prove that the SCWF axiom holds for the execution. For all k, l, a, b, if

$$a = \iota_k \wedge b = \iota_l \wedge k \neq l$$

then

$$\mathsf{op}(a) = \mathsf{op}(b) = \mathsf{fence} \wedge a \neq b,$$

because of the definition of the rules in our abstract implementation. This immediately implies that sc is irreflexive. It also implies the transitivity of sc. To see this, consider a, b, c such that

$$a \xrightarrow{\mathsf{sc}} b \wedge b \xrightarrow{\mathsf{sc}} c.$$

By the definition of sc, there exist i, j, k, l such that

$$i < j \wedge k < l \wedge a = \iota_i \wedge b = \iota_j \wedge b = \iota_k \wedge c = \iota_l.$$

By what we have shown above, $j = k$. Hence, $a \xrightarrow{\mathsf{sc}} c$, as desired. Now it remains to show that sc is total. Consider $a, b \in A$ such that $\mathsf{op}(a) = \mathsf{op}(b) = \mathsf{fence}$ and $a \neq b$. By the definition of the rules in our abstract implementation, there must exist i, j such that

$$a = \iota_i \wedge b = \iota_j.$$

Since $a \neq b$, i must be different from j. Then, $i < j$ or $j < i$. Hence,

$$(a \xrightarrow{\mathsf{sc}} b) \vee (b \xrightarrow{\mathsf{sc}} a),$$

as the totality requires.

Secondly, we prove that the execution satisfies the THINAIR axiom. We note that for every action $a \in A$, there exists a unique index k such that

$$(\_, a) = \iota_k \vee a = \iota_k.$$

This is because the timestamps of replicas are only increasing by our rules, and every action in A uses the timestamp of the accessed replica, which is increased right after the generation of the action. We write $\mathsf{index}(a)$ for the unique index k, and using this notation, we define a binary relation $<_\tau$ on A:

$$a <_\tau b \iff \mathsf{index}(a) < \mathsf{index}(b).$$

By definition, $<_\tau$ is a total order on A. Hence, we can prove the required acyclicity of $\mathsf{so} \cup \mathsf{vis} \cup \mathsf{sc}$ by showing that

$$\forall a, b \in A.\, (a \xrightarrow{\mathsf{so}} b \vee a \xrightarrow{\mathsf{vis}} b \vee a \xrightarrow{\mathsf{sc}} b) \implies a <_\tau b.$$

Consider actions $a, b \in A$. Assume that $a \xrightarrow{\mathsf{so}} b$ or $a \xrightarrow{\mathsf{sc}} b$. In this case, by the definition of so and sc, we have that $a <_\tau b$, as required. The remaining case is that $a \xrightarrow{\mathsf{vis}} b$. By the definition of vis, there exists a set of actions W such that

$$((W, b) = \iota_{\mathsf{index}(b)}) \wedge (\mathsf{id}(a) \in W). \tag{19}$$

Recall that the initial configuration $\mathcal{S}_0|\mathcal{D}_0$ has to be empty by the definition of trace. One consequence of this and the definition of our rules is that

$$\begin{array}{l}\forall i.\, \forall (\_, \_, \rho, \_, \_) \in \mathcal{D}_i.\, \forall x \in \mathsf{Obj}.\, \forall a \in A. \\ \qquad \mathsf{id}(a) \in \mathsf{id}_{\mathsf{type}(x)}(\rho(x)) \implies \mathsf{index}(a) \leq i.\end{array}$$

Since W in (19) comes from ρ in $\mathcal{D}_{\mathsf{index}(b)-1}$ and it contains $\mathsf{id}(a)$, it follows from the above implication that $\mathsf{index}(a) < \mathsf{index}(b)$. This means that $a <_\tau b$, as desired. □

PROPOSITION 28. *For every fair trace τ, the fence versions of the* COCV *and the* COCA *axioms hold for the execution generated from the trace τ if*

$$\begin{aligned}&(\forall x, e, f_\mu, \sigma, \rho, R, U, K.\\ &\quad \mathsf{idc}_{\mathsf{type}(x)}(\sigma) \cup \bigcup_{y \in \mathsf{Obj}} U(y)\\ &\qquad \subseteq \mathsf{depend}_{\mathsf{type}(x)}(e, f, \sigma, \rho, (R, U, K))) \wedge\\ &(\forall U, \rho.\ \mathsf{enable}(\_, (\_, (\_, U, \_)), (\_, \_, \rho, \_, \_))\\ &\quad \implies \forall x.\, U(x) \subseteq \mathsf{id}_{\mathsf{type}(x)}(\rho(x))).\end{aligned}$$

Proof: The proof is similar to that of Proposition 26. Consider a fair trace

$$\tau = \mathcal{S}_0|\mathcal{D}_0 \xrightarrow{\iota_1} \mathcal{S}_1|\mathcal{D}_1 \xrightarrow{\iota_2} \ldots \xrightarrow{\iota_n} \mathcal{S}_n|\mathcal{D}_n.$$

Let $(A, \mathsf{so}, \mathsf{vis}, \mathsf{ar}, \mathsf{sc})$ be an execution generated by this trace τ. We need to show that the execution satisfies the fence versions of the COCV and the COCA axioms, which use the notion of hb that accommodates both the fence operator and the on-demand causal operations.

We remind the reader that for each binary relation r on A, a set X of action ids is said to be ***r*-closed** if

$$\forall a, b \in A.\, (\mathsf{id}(b) \in X \wedge (a, b) \in r \implies \mathsf{id}(a) \in X).$$

For every $a \in A$, we write $\mathsf{index}(a)$ for the unique index k such that

$$(a = \iota_k) \vee ((\_, a) = \iota_k).$$

Also, for each index i of the trace τ, we let

$$F_i = \{\mathsf{id}(a) \mid a \in A \wedge \mathsf{op}(a) = \mathsf{fence} \wedge \mathsf{index}(a) \leq i\}.$$

The key observation behind our proof is that every configuration $\mathcal{S}_i|\mathcal{D}_i$ in the trace τ satisfies the following property:

$$\begin{aligned}&(\forall (\_, \_, \rho, \_, \_) \in \mathcal{D}_i.\, F_i \cup \bigcup_{x \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(x)}(\rho(x)) \text{ is } \mathsf{hb}\text{-closed}) \wedge\\ &(\forall (\_, \_, \_, M, N) \in \mathcal{D}_i. \forall a, a' \in A. \forall (\_, \mathsf{id}(a), f_\mu, \_, E) \in M \cup N.\\ &\quad (a', a) \in (\mathsf{so} \cup \mathsf{cvis}) \implies \mathsf{id}(a') \in E \cup F_i).\end{aligned}$$

Note that hb here is $(\mathsf{so} \cup \mathsf{cvis} \cup \mathsf{sc})^+$. This property can be proven by induction on i. When $i = 0$, the property holds because $\mathcal{S}_0|\mathcal{D}_0$ is empty and for every $a \in A$, $\mathsf{index}(a) \geq 1$. We move on to the case that $i > 0$. Suppose that the property holds for all $j < i$. We do the case analysis on the rule used in the i-th step.

If the rule is PROP, no new messages nor fence actions are generated, and the object tables of replicas remain unchanged by the rule. Hence, the property holds by induction hypothesis and the fact that F_k only increases as k gets larger.

If the rule is OPD, the situation is similar to the previous case. The only difference lies in one replica that gets affected by the rule. Let

$$\delta = (\_, \_, \rho, \_, \_) \text{ and } m = (x, e, f_\mu, \_, E)$$

be a replica's configuration and the message that get used by the OPD rule in the i-th step. Also, let σ' be the new state of a data type computed by OPD, and a the action in A with $\mathsf{id}(a) = e$; such an action exists because all messages in a trace are generated together with corresponding actions in our abstract implementation. Since OPD does not generate a fence action,

$$F_{i-1} = F_i.$$

By induction hypothesis, the set

$$F_{i-1} \cup \bigcup_{y \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(y)}(\rho(y))$$

is hb-closed, and

$$\forall a' \in A.\, (a', a) \in (\mathsf{cvis} \cup \mathsf{so}) \implies \mathsf{id}(a') \in E \cup F_{i-1}$$

Furthermore, by the definition of OPD,

$$E \subseteq \bigcup_{y \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(y)}(\rho(y)).$$

By the condition on eval and the definition of OPD,

$$\mathsf{id}_{\mathsf{type}(x)}(\sigma') = \mathsf{id}_{\mathsf{type}(x)}(\rho(x)) \cup \{e\}.$$

Recall that $\mathsf{hb} = (\mathsf{so} \cup \mathsf{cvis} \cup \mathsf{sc})^+$, and that $\mathsf{op}(a) \neq \mathsf{fence}$ by the definition of the rules in our abstract implementation. From what we have proved follows that for every $b \in A$, if $(b, a) \in (\mathsf{so} \cup \mathsf{cvis} \cup \mathsf{sc})$, then $\mathsf{id}(b)$ belongs to the following set:

$$F_i \cup \bigcup_{y \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(y)}(\rho[x \mapsto \sigma'](y)). \tag{20}$$

This fact, $F_i = F_{i-1}$ and induction hypothesis imply that the set above is hb-closed.

The next case is OPS. Let

$$\theta = (\_, (\_, U, \_)), \text{ and } \delta = (d, t, \rho, \_, \_)$$

be the configurations of the session and the replica affected by the OPS rule. Let

$$m = (x, e, f_\mu, W', E') \text{ and } a$$

be the message and the action generated by the rule. By the definition of the rule,

$$\exists y.\, \mathsf{enable}(y, \theta, \delta).$$

But because of the assumption in the proposition, this implies that

$$\forall y.\, U(y) \subseteq \mathsf{id}_{\mathsf{type}(y)}(\rho(y)).$$

Also, because of the assumption on depend in the proposition,

$$\left(\mathsf{idc}_{\mathsf{type}(x)}(\rho(x)) \cup \bigcup_{y \in \mathsf{Obj}} U(y)\right) \subseteq E'$$

Since $\mathsf{idc}_{\mathsf{type}(x)}(\rho(x))$ contains the ids of all actions that are causal and vis-related to a, the above subset relationship implies that

$$\begin{aligned}&\forall a' \in A.\\ &\quad \mathsf{level}(a') = \mathsf{CSL} \wedge (a', a) \in (\mathsf{vis} \cup \mathsf{so}) \implies \mathsf{id}(a') \in E'.\end{aligned}$$

Also, for every $a' \in A$, if

$$\exists c.\, \mathsf{op}(a') = \mathsf{fence} \wedge a' \xrightarrow{\mathsf{so}} c \xrightarrow{\mathsf{hbo}} a$$

then

$$\mathsf{index}(a') < \mathsf{index}(a).$$

The details of this consequence are given in the second part of the proof of Proposition 27. Hence, every such a' should be in F_i. From this membership and the fact on E' above follows that

$$\forall a' \in A.\ (a', a) \in (\mathsf{so} \cup \mathsf{cvis}) \implies \mathsf{id}(a') \in E' \cup F_i.$$

This means that the generated message by the rule satisfies the property that we try to show. All the other messages satisfy the property as well because of induction hypothesis and the fact that $F_i = F_{i-1}$. By a similar reason, the object tables in $\mathcal{D}_i$ other than ρ also satisfy the desired property. It remains to show that the object table ρ satisfies the property. By the condition on eval and the definition of OPS, the set of action ids associated with the updated replica is the following set:

$$\{\mathsf{id}(a)\} \cup \bigcup_{y \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(y)}(\rho(y)).$$

Also,

$$F_{i-1} = F_i.$$

Now consider the set:

$$F_i \cup \{\mathsf{id}(a)\} \cup \bigcup_{y \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(y)}(\rho(y)). \tag{21}$$

By the definition of vis, the ids of actions in $\mathsf{vis}^{-1}(a)$ are included in the above union. Also, for every $a' \in A$, if

$$\exists c.\, \mathsf{op}(a') = \mathsf{fence} \wedge a' \xrightarrow{\mathsf{so}} c \xrightarrow{\mathsf{hbo}} a$$

then

$$a' \in F_{i-1} = F_i.$$

From what we just argued follows that the ids of actions in $\mathsf{cvis}^{-1}(a)$ are included in the set in (21). By the assumption of the proposition, the ids of actions in $\mathsf{so}^{-1}(a)$ are also included in the same set in (21). Furthermore, sc does not relate non-fence actions and $\mathsf{op}(a) \neq \mathsf{fence}$. Hence, the ids of the actions in $(\mathsf{so} \cup \mathsf{cvis} \cup \mathsf{sc})^{-1}(a)$ are contained in the set in (21). This membership and the induction hypothesis imply that the set in (21) is hb-closed, as desired.

The remaining case is OPF. This rule does not generate any messages, and $F_{i-1} \subseteq F_i$. So, the desired property on messages follows from the induction hypothesis. Also, the rule does not alter any object tables. Thus, it suffices to show that the fence action generated by the rule does not lead to the violation of the property on object tables. Let a be the fence action generated by the rule in the i-th step of the trace. Consider $a' \in A$ such that

$$(a', a) \in (\mathsf{so} \cup \mathsf{cvis} \cup \mathsf{sc}).$$

This implies that

$$\mathsf{index}(a') < \mathsf{index}(a). \tag{22}$$

It suffices to show that

$$\forall (\_, \_, \rho_i, \_, \_) \in \mathcal{D}_i.\, \mathsf{id}(a') \in F_i \cup \bigcup_{y \in \mathsf{Obj}} (\mathsf{id}_{\mathsf{type}(y)}(\rho_i(y))).$$

This follows from (22), the assumption of the proposition on the enable predicate, and the definitions of so, cvis and sc.

Let us go back to the proof that the execution

$$(A, \mathsf{so}, \mathsf{vis}, \mathsf{ar}, \mathsf{sc})$$

satisfies the COCV and the COCA axioms. We consider the COCV axiom first. Consider $a, c \in A$ such that

$$(a \xrightarrow{\mathsf{hb}} c) \wedge \mathsf{obj}(a) = \mathsf{obj}(c).$$

Let i, k, W_i, W_k be indices and sets of action ids such that

$$(W_i, a) = \iota_i \wedge (W_k, c) = \iota_k.$$

Then,

$$i < k$$

because $b \xrightarrow{\mathsf{hb}} b' \implies \mathsf{index}(b) < \mathsf{index}(b')$; the details of this implication are given in the second part of the proof of Proposition 27. Let

$$\theta_k = (\_, (\_, U, \_)) \ \text{ and } \ \delta_k = (d, t, \rho, \_, \_)$$

be the configurations of the session and the replica used by the k-th step of the trace τ. By what we proved about the hb closedness in the first half of this proof, the set

$$F_k \cup \bigcup_{x \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(x)}(\rho(x))$$

is hb-closed. Furthermore, a is not a fence action, but $a \xrightarrow{\mathsf{hb}} c$. Hence,

$$\mathsf{id}(a) \in \bigcup_{x \in \mathsf{Obj}} \mathsf{id}_{\mathsf{type}(x)}(\rho(x)).$$

This implies

$$\mathsf{id}(a) \in \mathsf{id}_{\mathsf{type}(\mathsf{obj}(a))}(\rho(\mathsf{obj}(a))) = W_k,$$

where we used the fact that $\mathsf{obj}(a) = \mathsf{obj}(c)$. This gives $a \xrightarrow{\mathsf{vis}} c$, as desired.

Next, we handle the COCA axiom. Define a total order $<_\tau$ on all actions in A as follows:

$$a <_\tau b \iff \mathsf{id}(a) < \mathsf{id}(b).$$

This is an irreflexive transitive total order because so is $<$ on action ids and different actions in A have different action ids. We will show that

$$\forall a, b \in A.\ (a \xrightarrow{\mathsf{ar}} b \vee a \xrightarrow{\mathsf{cvis}} b \vee a \xrightarrow{\mathsf{so}} b \vee a \xrightarrow{\mathsf{sc}} b) \implies a <_\tau b. \tag{23}$$

Then, from the transitivity and irreflexivity of $<_\tau$ follows that

$$(\mathsf{ar} \cup \mathsf{cvis} \cup \mathsf{so} \cup \mathsf{sc})^+ = (\mathsf{ar} \cup \mathsf{hb})^+$$

is irreflexive, as desired. Pick $a, b \in A$ such that

$$a \xrightarrow{\mathsf{ar}} b \vee a \xrightarrow{\mathsf{cvis}} b \vee a \xrightarrow{\mathsf{so}} b \vee a \xrightarrow{\mathsf{sc}} b.$$

If $a \xrightarrow{\mathsf{ar}} b$, then $a <_\tau b$ by the definition of ar. If $a \xrightarrow{\mathsf{sc}} b$, then a and b must be fence actions and we should have that

$$\mathsf{index}(a) < \mathsf{index}(b).$$

The OPF rule ensures that the timestamp of the generated fence action is smaller than the new timestamps of all replicas, and no rules in our implementation decrease the timestamps of replicas. Hence, the above relationship on the indices of a and b implies that

$$\mathsf{id}(a) < \mathsf{id}(b),$$

which gives $a <_\tau b$ as desired. By similar reasoning and the assumption of the proposition on the enable predicate, we can also conclude that if $a \xrightarrow{\mathsf{so}} b$, then $a <_\tau b$. The only remaining case is that

$$a \xrightarrow{\mathsf{cvis}} b.$$

This case gets split into two subcases:

$$(\exists f.\, a \xrightarrow{\mathsf{vis}} b \wedge \mathsf{op}(a) = f_{\mathsf{CSL}}) \vee$$
$$(\exists c.\, \mathsf{op}(a) = \mathsf{fence} \wedge a \xrightarrow{\mathsf{so}} c \xrightarrow{\mathsf{hbo}} b).$$

If the second disjunct above holds, we can again use the similar reasoning on the timestamps of replicas and the OPF rule. Since $\mathsf{index}(a) < \mathsf{index}(b)$ in this subcase, this reasoning shows that $\mathsf{id}(a) < \mathsf{id}(b)$. Hence, $a <_\tau b$. Now let us assume that the first disjunct holds. Then, neither a nor b is a fence action. Let i, j, W_i, W_j be indices and sets of action ids such that

$$(W_i, a) = \iota_i \wedge (W_j, b) = \iota_j.$$

Since $a \xrightarrow{\mathsf{vis}} b$, the set W_j should contain $\mathsf{id}(a)$. The rules of our implementation ensure that if an action is generated from a replica in the trace, it gets an id that is larger than the ids of any actions applied to the replica so far. Hence, $\mathsf{id}(a) < \mathsf{id}(b)$, which gives the desired $a <_\tau b$. □

C. Comparison with the C/C++ Memory Model

Our formalisation allows us to compare different forms of eventual consistency with memory consistency models implemented by common hardware and programming languages. To this end, we specialise the axioms in Figure 1 to the case when all objects are of the type intreg (§2), which corresponds to the setting of the latter models (Figure 6). The notion of an execution can be simplified in this case: we define a ***shared-memory execution*** as a tuple $(A, \mathsf{so}, \mathsf{vis}, \mathsf{ar})$, where (A, so) is a history, and vis and ar satisfy VISWF and ARWF from Figure 6. Now write actions do not have incoming vis edges, and the ar relation totally orders writes to the same object. Read actions have a single incoming vis edge from the write action it fetches its value from. According to the RVAL axiom in Figure 6, a read returns the value written by the vis-related write, or the default value 0 in its absence. The EVENTUAL axiom is adjusted to account for the changed type of vis: it prohibits infinitely many reads to read from writes preceding any given one in ar.

The other axioms in Figure 6 are formulated in the 'negative' form, prohibiting certain configurations in an execution, rather than in the 'positive' form, like in Figure 1, since here this is more concise. For example, RYW says that a read r that happened after a write w_2 in a session cannot read from a write w_1 earlier in ar. This follows from the RYW axiom in Figure 1: in an execution in the sense of Definition 5 satisfying that axiom, the read r has to see the write w_2; then according to RVAL in Figure 1 and the definition of $\mathcal{F}_{\mathsf{intreg}}$, r cannot take its value from w_1. Similarly, the COCV axiom says that, if a read r causally happens after a write w_2, then it cannot read from a write w_1 earlier in ar. As before, this follows from COCV in Figure 1, which in this case guarantees that, r sees both writes. There are no axioms corresponding to the WFRV and MWV session guarantees, because for integer registers, the session guarantee axioms in Figure 6 are enough to imply per-object causality: the conjunction of the axioms RYW–MWA in Figure 6 is equivalent to the conjunction of POCV and POCA.

The resulting consistency model maps to that of C/C++ [8] as explained in §3.3. The only major difference is that C/C++ has the following weaker analogue of COCA:

$$\neg\exists w_1, w_2.\quad w_1 \underset{\mathsf{ar}}{\overset{\mathsf{hb}}{\rightleftarrows}} w_2$$

This does not allow ar to contradict hb, but allows ar edges for different objects to form cycles with it. As we explained in §3.3, the stronger version of COCA is validated by common implementations of causal consistency.

PROPOSITION 29. *The system specifications for the axioms of basic eventual consistency, per-object causal consistency and causal consistency as defined in Figure 6 and Figure 1 for* intreg *are identical.*

D. Comparison with Parallel Snapshot Isolation

Figure 7 gives the pseudocode of the abstract implementation of parallel snapshot isolation for replicated data types, similar to the one given by Sovran et al. [30]. We have removed write-write conflict detection and introduced sessions. See [30] for the explanation of the implementation. We assume that each session always accesses the same site. To validate EVENTUAL, we also assume that the **upon** statement is subject to a fairness constraint that every update is eventually delivered to every site.

THEOREM 30. *The set of histories produced by the implementation in Figure 7 is equal to the one produced by the specification in Figure 4, assuming that all objects are of type* intreg *and all actions are causal.*

PROOF SKETCH. We first show that the set of histories produced by the implementation in Figure 7 is included into that specified in Figure 4. Consider a run of the implementation in Figure 7. We now construct a corresponding execution X and show that it satisfies the axioms in 4. To construct the arbitration relation ar, we associate every action a with a timestamp

Figure 6. Eventual consistency axioms specialised to read-write registers. Variables r, r_1, r_2 and w, w_1, w_2 range over read and write actions, respectively; arg selects the argument of a write.

WELL-FORMEDNESS AXIOMS

SOWF: so is the union of transitive, irreflexive and total orders on actions by each session

VISWF: $\forall w, r.\ w \xrightarrow{\mathsf{vis}} r \implies \mathsf{obj}(w) = \mathsf{obj}(r) \wedge \mathsf{op}(w) = \mathsf{wr} \wedge \mathsf{op}(r) = \mathsf{rd} \wedge (\forall w_1, w_2.\ w_1 \xrightarrow{\mathsf{vis}} r \wedge w_2 \xrightarrow{\mathsf{vis}} r \implies w_1 = w_2)$

ARWF: ar is the union of transitive, irreflexive and total orders on writes to each object

DATA TYPE AXIOM

RVAL: $\forall r \in A.\ (\exists w.\ w \xrightarrow{\mathsf{vis}} r \wedge \mathsf{rval}(r) = \mathsf{arg}(w)) \vee ((\neg\exists w.\ w \xrightarrow{\mathsf{vis}} r) \wedge \mathsf{rval}(r) = 0)$

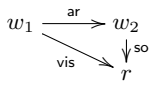
BASIC EVENTUAL CONSISTENCY AXIOMS

THINAIR: so ∪ vis is acyclic

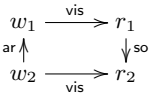
EVENTUAL: $\forall w \in A.\ \neg(\exists \text{ infinitely many } r \in A.\ \exists w'.\ w' \xrightarrow{\mathsf{vis}} r \wedge w' \xrightarrow{\mathsf{ar}} w)$

SESSION GUARANTEES (A.K.A. COHERENCE AXIOMS)

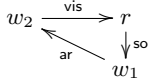
RYW: $\neg\exists r, w_1, w_2.$



MR: $\neg\exists r_1, r_2, w_1, w_2.$



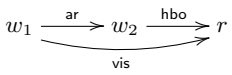
WFRA: $\neg\exists r, w_1, w_2.$



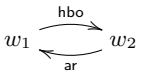
MWA: $\neg\exists w_1, w_2.$

CAUSALITY AXIOMS

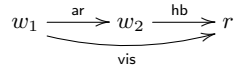
POCV: $\neg\exists w_1, w_2, r.$



POCA: $\neg\exists w_1, w_2.$



COCV: $\neg\exists w_1, w_2, r.$



COCA:

hb ∪ ar is acyclic

Figure 7. Abstract implementation of parallel snapshot isolation by Sovran et al. [30]

```
Sites: 1..N
Log[N]: log of operations per site
Transaction attributes: site, sessionId, startTs, commitTs[N]

operation startTx(x)
  x.startTs := new monotomic timestamp

operation write(x, obj, data)
  append (obj, data) to x.updates

operation read(x, obj)
  return state of obj from x.updates and Log[site(x)]
    up to timestamp x.startTs

operation commitTx(x, obj, data)
  x.commitTs[site(x)] := new monotonic timestamp
  append x.updates to Log[site(x)] with
    timestamp x.commitTs[site(x)]

upon [∃x,s: x.commitTs[site(x)] != ⊥ and
    x.commitTs[s] = ⊥ and ∀y such that
    y.commitTs[site(x)] < x.startTs : y.commitTs[s] != ⊥]
  x.commitTs[s] := new monotonic timestamp
  append x.updates to Log[s] with timestamp x.commitTs[s]
```

$t(a)$, which is a pair of the time when its transaction was commited on its original site and the order of the operation in the transaction. Then ar is constructed by lexicographically ordering the resulting timestamps. Then ARWF and VISWF (for any definition of vis) hold. To construct the visibility relation vis, we first construct vis′ relation, relating writes to reads that took them into account when computing their result in read(). We then let $\mathsf{vis} = ((\mathsf{so} \cup \mathsf{vis}')/\sim)^+$, so that COCV is satisfied automatically. Due to the causality clause in the **upon** statement, it is easy to check that the projection of vis to pairs of writes followed by reads is equal to vis′. Hence, RVAL is satisfied.

For all actions a and b, if $a \xrightarrow{\mathsf{so}} b$ or $a \xrightarrow{\mathsf{vis}} b$, then $t(a) < t(b)$, and thus, THINAIR holds as well. EVENTUAL is satisfied by the fairness condition associated with the **upon** statement. Since all actions in a transaction receive the same timestamp, the clause for ar in TRANSACT holds; since transactions are appended to logs atomically, so does the clause for vis. ISOLATION holds, since, in the implementation, operations performed by an uncommitted transactions are only visible to that transactions.

Assume that COCA does not hold, and thus, there is a cycle in $(\mathsf{so} \cup \mathsf{vis} \cup \mathsf{ar})/\sim$. The cycle cannot contain actions from a single transaction only, as this would contradict the way we assign timestamps $t(a)$ to actions a. Hence, the cycle contains actions from multiple transactions. Then every time we move from one transaction to another one on the cycle, the first component of the timestamp of the corresponding actions strictly increases, which yields a contradiction. Hence, COCA holds, which completes the proof.

We now show that the set of histories produced by specification in Figure 4 is included into that of the implementation in Figure 7. To this end, consider an execution $X = (A, \mathsf{so}, \mathsf{vis}, \mathsf{ar})$ satisfying the axioms in 4. We construct the corresponding run of the implementation as follows. Let ar′ be any total order on all transactions generated by $(\mathsf{ar} \cup \mathsf{vis} \cup \mathsf{so})/\sim$. Such an order exists due to COCA. We assume that every session has its own dedicated site. The action set A and the session order so determine the sequences of operations performed by every session. Every transaction executes atomically and commits straight away in the order given by ar′. Then so, vis and ar edges are consistent with the timestamps assigned to transactions. Propagation in the **upon** statement is driven by the vis edges: when an action a reads an object, we propagate all the writes that it is supposed to see, as well as their causal predecessors determined by the **upon** statement, to the site the action executes on (except the writes that have already been propagated to it). Since so, vis and ar edges are consistent with the timestamps, all such actions have already been executed. Furthermore, it is easy to check that this in fact propagates only the

actions visible to a. It remains to show that the return value of read() in the implementation is consistent with the one specified by the reference execution. Suppose this were not the case. Then at the time of execution of a read a in the implementation, its site would contain a write b such that $\neg(b \xrightarrow{\mathsf{vis}} a)$. The write b could only be propagated to the site as a consequence of an hb dependency from b to a. But then COCV implies that b has to be visible to a, yielding a contradiction. Hence, the run we have constructed is indeed a run of the implementation. $\square$

JFP **23** (4): 452–514, 2013.
doi:10.1017/S0956796813000075

Modular verification of preemptive OS kernels

ALEXEY GOTSMAN
IMDEA Software Institute
(*e-mail:* alexey.gotsman@imdea.org)

HONGSEOK YANG
University of Oxford
(*e-mail:* Hongseok.Yang@cs.ox.ac.uk)

Abstract

Most major OS kernels today run on multiprocessor systems and are preemptive: it is possible for a process running in the kernel mode to get descheduled. Existing modular techniques for verifying concurrent code are not directly applicable in this setting: they rely on scheduling being implemented correctly, and in a preemptive kernel, the correctness of the scheduler is interdependent with the correctness of the code it schedules. This interdependency is even stronger in mainstream kernels, such as those of Linux, FreeBSD or Mac OS X, where the scheduler and processes interact in complex ways. We propose the first logic that is able to decompose the verification of preemptive multiprocessor kernel code into verifying the scheduler and the rest of the kernel separately, even in the presence of complex interdependencies between the two components. The logic hides the manipulation of control by the scheduler when reasoning about preemptable code and soundly inherits proof rules from concurrent separation logic to verify it thread-modularly. We illustrate the power of our logic by verifying an example scheduler, which includes some of the key features of the scheduler from Linux 2.6.11 challenging for verification.

1 Introduction

Developments in formal verification now allow us to consider the full verification of an operating system (OS) kernel, one of the most crucial components in any system today. Several recent projects have demonstrated that a formal verification of can tackle realistic OS kernels, such as a variant of the L4 microkernel (Klein *et al.*, 2009) and Microsoft's Hyper-V hypervisor (Cohen *et al.*, 2010). However, these projects only dealt with relatively small microkernels; tackling today's mainstream operating systems, such as Windows and Linux, remains a daunting task. A way to approach this problem is to verify OS kernels *modularly*, i.e., by considering each of their components in isolation. In this paper, we tackle a major challenge OS kernels present for modular reasoning—handling kernel preemption in a multiprocessor system. Most major OS kernels are designed to run with multiple CPUs and are *preemptive*: it is possible for a process running in the kernel mode to get descheduled. Reasoning about such kernels is difficult for the following reasons.

First of all, in a multiprocessor system several invocations of a system call may be running concurrently in a shared address space, so reasoning about the call needs to consider all possible interactions among them. This is a notoriously difficult problem; however,

we now have a number of logics that can reason about concurrent code (O'Hearn, 2007; Gotsman *et al.*, 2007; Feng *et al.*, 2007a; Vafeiadis & Parkinson, 2007; Dinsdale-Young *et al.*, 2010; Cohen *et al.*, 2010; Dinsdale-Young *et al.*, 2013). The way the logics make verification tractable is by using *thread-modular* reasoning principles that consider every thread of computation in isolation under some assumptions about its environment and thus avoid direct reasoning about all possible interactions (Jones, 1983; Pnueli, 1985).

The problem is that all these logics can verify code only under so-called interleaving semantics, expressed by the well-known operational semantics rule:

$$\frac{C_k \longrightarrow C_k'}{C_1 \parallel \ldots \parallel C_k \parallel \ldots \parallel C_n \longrightarrow C_1 \parallel \ldots \parallel C_k' \parallel \ldots \parallel C_n}$$

This rule effectively assumes an abstract machine where every process C_k has its own CPU, whereas in an OS, the processes are multiplexed onto available CPUs by a scheduler, which is part of the OS kernel. Furthermore, in a preemptive kernel, the correctness of the scheduler is *interdependent* with the correctness of the rest of the kernel (which, in the following, we refer to as just the kernel). This is because, when reasoning about a system call implementation in a preemptive kernel, we have to consider the possibility of context-switch code getting executed at almost every program point. Upon a context switch, the state of the system call will be stored in kernel data structures and subsequently loaded for execution again, possibly on a different CPU. A bug in the context switch code can load an incorrect state of the system call implementation upon a context switch, and a bug in the system call can corrupt the scheduler's data structures. It is, of course, possible to reason about the kernel together with the scheduler as a whole, using one of the available logics. However, in a mainstream kernel, where kernel preemption is enabled most of the time, such reasoning would quickly become intractable.

Contributions. In this paper, we propose a logic that is able to decompose the verification of safety properties of preemptive OS code into verifying preemptable code and the scheduler (including the context-switch code) separately. This is the first logic that can achieve this in the presence of interdependencies between the scheduler and the kernel typical for mainstream OS kernels, such as those of Linux, FreeBSD and Mac OS X. The modularity of the logic is reflected in the structure of its proof system, which is partitioned into high-level and low-level parts. The high-level proof system verifies preemptable code assuming that the scheduler is implemented correctly (Section 5.2). It hides the complex manipulation of control by the context-switch code, which stores program counters of processes, describing their continuations, and jumps to one of them. In this way, the high-level proof system provides the illusion of an abstract machine where every process has its own virtual CPU—the control moves from one program point in the process code to the next without changing its state. This illusion is justified by verifying the scheduler code separately from the kernel in the low-level proof system (Section 5.3). Achieving this level of modularity requires us to cope with several technical challenges.

First, the setting of a preemptive OS kernel introduces an obligation to prove that the scheduler and the kernel do not corrupt each other's data structures. A common way to achieve this is by introducing the notion of *ownership* of memory areas: only the component owning an area of memory has the right to access it (Clarke *et al.*, 2001; Reynolds,

2002). A major difficulty of decomposing the verification of the mainstream OS kernels mentioned above lies in the fact that, in such kernels, there is no static address space separation between data structures owned by the scheduler and the rest of the kernel: the boundary between these changes according to a protocol for transferring the ownership of memory cells and permissions to access them in a certain way. For example, when an implementation of the `fork` system call asks the scheduler to make a new process runnable, the scheduler usually gains the ownership of the process descriptor provided by the system call implementation.

To deal with this, we base our proof systems on concurrent separation logic (O'Hearn, 2007), which we recap in Section 4. The logic allows us to track the dynamic memory partitioning between the scheduler and the rest of the kernel and prohibit memory accesses that cross the partitioning boundary. For example, assertions in the high-level proof system talk only about the memory belonging to the kernel and completely hide the memory belonging to the scheduler. A *frame property*, validated by concurrent separation logic, implies that in this case any memory not mentioned in the assertions, e.g., the memory belonging to the scheduler, is guaranteed not to be changed by the kernel. A realistic interface between the scheduler and the kernel is supported by proof rules for ownership transfer of logical assertions between the two components, describing permissions to access memory cells.

Using an off-the-shelf logic, however, is not enough to prove the correctness of a scheduler, as this requires domain-specific reasoning: e.g., we need to be able to check that a scheduler restores the state of a preempted process correctly when it resumes the process. To this end, the low-level proof system for reasoning about schedulers includes special Process assertions about the continuation of every OS process the scheduler manages, describing the states from which it can be safely resumed. A novelty of these assertions is the semantics of the separating conjunction connective on them, which treats the assertions affinely and allows us to interpret them as exclusive permissions to schedule the corresponding processes. This enables reasoning about scheduling on multiprocessors, as it allows checking that the scheduler invocations on different CPUs coordinate decisions about process scheduling correctly. Another interesting feature of Process assertions is that they describe only the part of the process state the scheduler is supposed to access and hide the rest; this ensures that the scheduler indeed cannot corrupt the latter. In this our Process assertions are similar to abstract predicates (Parkinson & Bierman, 2005).

Even though a scheduler is supposed to provide an illusion of running on a dedicated *virtual* CPU to every process, in practice, some features available to the kernel code can break through this abstraction: e.g., a process can disable preemption and become aware of the *physical* CPU on which it is currently executing. For example, some OS kernels use this to implement *per-CPU data structures* (Bovet & Cesati, 2005)—arrays indexed by CPU identifiers such that a process can only access an entry in an array when it runs on the corresponding CPU. We demonstrate that our approach can deal with such implementation exposures by extending the high-level proof system for the kernel code with axioms that allow reasoning about per-CPU data structures (Section 7).

In reasoning about mainstream operating systems, assertions describing the state transferred between the scheduler and the kernel can be complicated. The resulting ownership transfers make even formalising the notion of scheduler correctness non-trivial, as they are difficult to accommodate in an operational semantics of the abstract machine with

one CPU per process the scheduler is supposed to implement (Gotsman *et al.*, 2011). We resolve this problem in the following way. In our logic, the desired properties of OS code are proved with respect to the abstract machine using the high-level proof system; the low-level system then relates the concrete and the abstract machines. However, proofs in neither of the two systems are interpreted with respect to any semantics alone. Instead, our soundness statement (Section 8) interprets a proof of the kernel in the high-level system and a proof of the scheduler in the low-level one together with respect to the semantics of the concrete machine. To this end, the statement has to construct a global property of the machine from local assertions about OS components in a non-trivial way.

Even though all of the OS verification projects carried out so far had to deal with a scheduler (see Section 9 for a discussion), to our knowledge they have not produced methods for handling practical multiprocessor schedulers with a complicated scheduler/kernel interface. We illustrate the power of our logic by verifying an example scheduler (Sections 2.2 and 6), which includes some of the key features of the scheduler from Linux 2.6.11 exhibiting the issues mentioned above.

Limitations. Our goal here is not to verify an industrial-strength preemptive OS kernel—such an endeavour is beyond the scope of a single paper. Rather, we develop *principles* of how a given logic for verifying concurrent programs can be extended to verify preemptive kernel code with *real-world features*. These principles can then be used in verification projects that target different operating systems. To communicate the proposed principles cleanly and understandably, we present our results in a simplified setting:

- Instead of a realistic processor, such as x86, we consider an idealised machine (Sections 2.1 and 3).
- Since we are primarily interested in interactions of components within an OS kernel, our machine does not make a distinction between the user mode and the kernel mode.
- We base our logic for verifying OS kernels on one of the simplest logics for concurrent code—concurrent separation logic (O'Hearn, 2007). This logic would not be able to handle complicated concurrency mechanisms employed in modern OS kernels (Bovet & Cesati, 2005). However, as we argue in Section 8.2, our development can be adapted to its more advanced derivatives (Feng *et al.*, 2007a; Gotsman *et al.*, 2007; Vafeiadis & Parkinson, 2007; Dinsdale-Young *et al.*, 2010; Dinsdale-Young *et al.*, 2013).
- Since we concentrate on modular reasoning about concurrency and preemptive scheduling, our logic provides only rudimentary means of modular reasoning about sequential code and, in particular, procedures. We discuss ways of addressing this problem in Section 10.
- We consider scheduling interfaces providing only the basic services—context switch and process creation. We discuss how our logic can be extended to schedulers with more elaborate interfaces in Sections 5.5 and 10.
- Due to our focus on scheduling, we ignore many other aspects of an OS kernel, such as virtual memory and interrupts not related to scheduling.
- Our logic is designed for proving safety properties only. Proof methods for liveness properties usually rely on modular methods for safety properties. Thus, our logic is a prerequisite for attacking liveness in the future.

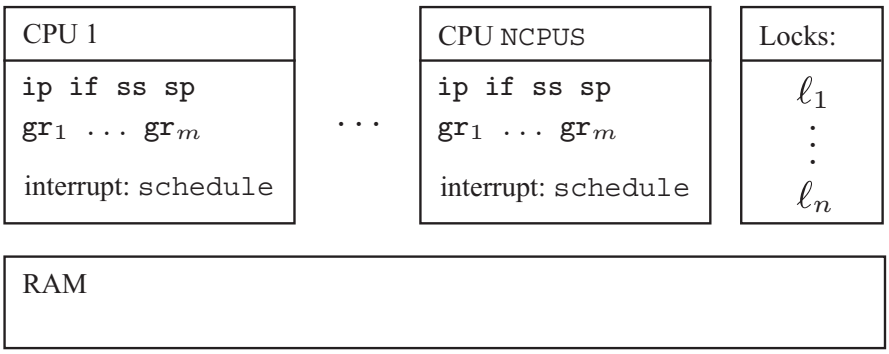


Fig. 1. The target machine.

Even though we develop our logic in this simplified setting, we hope that its modular nature makes it more likely that it will compose with logics for features and abstractions that we currently do not handle. A preliminary indication of this is our ability to deal with per-CPU data structures, despite the fact that these break through the abstraction implemented by a scheduler.

2 Informal development

We first explain our results informally, sketching the machine we use for formalising them (Section 2.1), illustrating the challenges of reasoning about schedulers by an example (Section 2.2) and describing the approach we take in our program logic (Section 2.3).

2.1 Example machine

We formalise our results for a simple machine, defined in Section 3. Here, we present it informally to the extent needed for understanding the rest of this section. We summarise its features in Figure 1.

The machine has multiple CPUs, identified by integers from 1 to `NCPUS`, communicating via the shared memory. We assume that the program the machine executes is stored separately from the heap and may not be modified; its commands are identified by labels. For simplicity, we also assume that programs can synchronise using a set of built-in locks (in a real system they would be implemented as spin-locks). Every CPU has a single interrupt, with its handler located at a distinguished label `schedule` (the same for all CPUs). A scheduler can use the interrupt to trigger a context switch. There are four special-purpose registers, `ip`, `if`, `ss` and `sp`, and m general-purpose ones, $\mathtt{gr}_1, \ldots, \mathtt{gr}_m$. The `ip` register is the instruction pointer. The `if` register controls interrupts: they are disabled on the corresponding CPU when `if` stores zero, and enabled otherwise. As `if` affects only one CPU, we might have several instances of the scheduler code executing in parallel on different CPUs. Upon an interrupt, the CPU sets `if` to 0, which prevents nested interrupts. The `ss` register keeps the starting address of the stack, and `sp` points to the top of the stack, i.e., its first free slot. The stack grows upwards, so we always have $\mathtt{ss} \leqslant \mathtt{sp}$. As we noted before, our machine does not make a distinction between the user mode and the

kernel mode—all processes can potentially access all available memory and execute all commands.

The machine executes programs in a minimalistic assembly-like programming language. It is described in full in Section 3; for now it suffices to say that the language includes standard commands for accessing registers and memory, and the following special ones:

- `lock`(ℓ) and `unlock`(ℓ) acquire and release the lock ℓ.
- `savecpuid`(e) stores the identifier of the CPU executing it at the address e.
- `call`(l) is a call to the function that starts at the label l. It pushes the label of the next instruction in the program and the values of the general-purpose registers onto the stack, and jumps to the label l.
- `icall`(l) behaves the same as `call`(l), except that it also disables interrupts by modifying the `if` register.
- `ret` is the return command. It pops the return label and the saved general-purpose registers off the stack, updates the registers with the new values, and jumps to the return label.
- `iret` is a variant of `ret` that additionally enables interrupts.

When the `if` register is set to a non-zero value, an interrupt can fire at any time. This has the same effect as executing `icall`(`schedule`).

2.2 Motivating example

The challenge. Figures 2–3 present an implementation of the scheduler that we use as a running example. Our goal is to be able to verify safety properties of OS processes managed by this scheduler using off-the-shelf concurrency logics, i.e., as though every process has its own virtual CPU. To show what this entails, consider the piece of code in Figure 4, which could be a part of a system call implementation in the kernel (we explain its proof later). The code removes some of the nodes from a cyclic doubly-linked list with a sentinel head node pointed to by `request_queue`. Here and in the following, we use some library functions: e.g., `remove_node` deletes a node from the doubly-linked list it belongs to. We assume that the code can be called concurrently by multiple processes, and thus protect the list with a lock `request_lock`: the list can only be accessed by the process that holds this lock.

There are a number of properties we might want to prove about this code: e.g., that pointer manipulations done by the invocation of `remove_node` in line 25 in Figure 4 do not overwrite a memory cell storing critical information elsewhere in the kernel, or that the doubly-linked list shape of `request_queue` is preserved. Modern concurrency logics can prove such properties by considering every process in isolation. Namely, every assertion in Figure 4 describes information about the state of the program relevant to the process executing the code. These assertions are justified using essentially sequential reasoning, and in particular, using the classical proof rule for sequential composition:

$$\frac{\{P_1\}\ C_1\ \{P_2\} \qquad \{P_2\}\ C_2\ \{P_3\}}{\{P_1\}\ C_1; C_2\ \{P_3\}} \tag{1}$$

We would like such reasoning (and the proof in Figure 4) to be sound even when the code is managed by the scheduler in Figures 2–3. Hence, we need to be able to ignore the fact that

the control flow of the code in Figure 4 can jump to the `schedule` function in Figure 2 at any time, with the state of the former stored in kernel data structures and loaded from them again later. Furthermore, we need to achieve this even though the scheduler and the system call implementation execute in a shared address space and can thus potentially access each other's data structures.

Example scheduler. The example scheduler in Figures 2–3 includes some of the key features of the scheduler from Linux 2.6.11 (Bovet & Cesati, 2005) that are challenging for verification, as detailed below.[1]

The scheduler's interface consists of two functions: `schedule` and `create`. The former is called as the interrupt handler or directly by a process and is responsible for switching the process running on the CPU and migrating processes between CPUs. The latter can be called by the kernel implementation of the `fork` system call and is responsible for inserting a newly created process into the scheduler's data structures, thereby making it runnable. Both functions are called by processes using the `icall` command that disables interrupts, so that the scheduler routines always execute with interrupts disabled.

Programming language. Even though we formalise our results for a machine executing a minimalistic programming language, we present the example in C. We now explain how a C program, such as the one in Figures 2–3, is mapped to our machine.

We assume that every global variable `x` is allocated at a fixed address `&x` in memory. Local variable declarations allocate local variables on the stack in the activation records of the corresponding procedures; these variables are then addressed via the `sp` register. When the variables go out of scope, they are removed from the stack by decrementing the `sp` register. The general-purpose registers are used to store intermediate values while computing complex expressions. In our C programs, we allow the `ss` and `sp` registers to be accessed directly as `_ss` and `_sp`. Function calls and returns are implemented using the `call` and `ret` commands of the machine. By default, parameters and return values are passed via the stack; in particular, a zero-filled slot for a return value is allocated on the stack before calling a function. Parameters of functions annotated with `_regparam` (such as `create`, line 64 in Figure 3) are passed via registers. We assume macros `lock`, `unlock`, `savecpuid` and `iret` for the corresponding machine commands.

Data structures. Every process is associated with a process descriptor of type `Process`. Its `prev` and `next` fields are used by the scheduler to connect descriptors into doubly-linked lists of processes it manages (*runqueues*). The scheduler uses per-CPU runqueues with dummy head nodes pointed to by the entries in the `runqueue` array. These are protected by the locks in the `runqueue_lock` array. The entries in the `current` array point to the descriptors of the processes running on the corresponding CPUs; these descriptors are not members of any runqueue. Thus, every process descriptor is either in the `current` array or in some runqueue. Note that every CPU always has at least one process to run—the

[1] We took an older version of the Linux kernel (from 2005) as a reference because its scheduler uses simpler data structures. Newer versions use more efficient data structures (Love, 2010) that would only complicate our running example without adding anything interesting.

```
#define StackSize ... // size of the stack
#define FORK_FRAME sizeof(Process*)
#define SCHED_FRAME sizeof(Process*)+sizeof(int)

struct Process {
  Process *prev, *next;
  word kernel_stack[StackSize];
  word *saved_sp;
  int timeslice;
};

Lock *runqueue_lock[NCPUS];
Process *runqueue[NCPUS];
Process *current[NCPUS];

void init() {
  for (int cpu = 0; cpu < NCPUS; cpu++) {
    Process* dummy = alloc(sizeof(Process));
    Process* process0 = alloc(sizeof(Process));
    dummy->prev = dummy->next = dummy;
    process0->timeslice = SCHED_QUANTUM;
    ... // initialise the stack of process0
    runqueue[cpu] = dummy;
    current[cpu] = process0;
  }
}

void schedule() {
  int cpu;
  Process *old_process;
  savecpuid(&cpu);
  load_balance(cpu);
  old_process = current[cpu];
  ... // update the timeslice of old_process
  if (old_process->timeslice) iret();
  old_process->timeslice = SCHED_QUANTUM;
  lock(runqueue_lock[cpu]);
  insert_node_after(runqueue[cpu]->prev, old_process);
  current[cpu] = runqueue[cpu]->next;
  remove_node(current[cpu]);
  old_process->saved_sp = _sp;
  _sp = current[cpu]->saved_sp;
  savecpuid(&cpu);
  _ss = current[cpu]->kernel_stack;
  unlock(runqueue_lock[cpu]);
  iret();
}
```

Fig. 2. The example scheduler (continued in Figure 3).

```
47 void load_balance(int cpu) {
48   int cpu2;
49   Process *proc;
50   if (random(0, 1)) return;
51   do { cpu2 = random(0, NCPUS-1); } while (cpu == cpu2);
52   if (cpu < cpu2) {
53     lock(runqueue_lock[cpu]); lock(runqueue_lock[cpu2]); }
54   else { lock(runqueue_lock[cpu2]); lock(runqueue_lock[cpu]); }
55   if (runqueue[cpu2]->next != runqueue[cpu2]) {
56     proc = runqueue[cpu2]->next;
57     remove_node(proc);
58     insert_node_after(runqueue[cpu], proc);
59   }
60   unlock(runqueue_lock[cpu]);
61   unlock(runqueue_lock[cpu2]);
62 }
63
64 _regparam void create(Process *new_process) {
65   int cpu;
66   savecpuid(&cpu);
67   new_process->timeslice = SCHED_QUANTUM;
68   lock(runqueue_lock[cpu]);
69   insert_node_after(runqueue[cpu], new_process);
70   unlock(runqueue_lock[cpu]);
71   iret();
72 }
73
74 int fork() {
75   Process *new_process;
76   new_process = alloc(sizeof(Process));
77   memcpy(new_process->kernel_stack, _ss, StackSize);
78   new_process->saved_sp = new_process->kernel_stack+
79       _sp-_ss-FORK_FRAME+SCHED_FRAME;
80   _icall create(new_process);
81   return 1;
82 }
```

Fig. 3. The example scheduler (continued).

one in the corresponding slot of the `current` array. Every process has its own kernel stack of a fixed size `StackSize`, represented by the `kernel_stack` field of its descriptor. When a process is preempted, the `saved_sp` field is used to save the value of the stack pointer register `sp`; the other registers are saved on the stack. Finally, while a process is running, the `timeslice` field gives the remaining time from its scheduling time quantum and is periodically updated by the scheduler. The `init` function in Figure 2 sketches code that could be used to initialise the scheduler data structures.

Apart from the data structures described above, a realistic kernel would contain many others not related to scheduling, including additional fields in process descriptors. The kernel data structures reside in the same address space as the ones belonging to the scheduler;

```
1  #define StackSize ...  // size of the stack
2  struct Request {
3    Request *prev, *next;
4    int data;
5  };
6  Request *request_queue;  // a cyclic doubly−linked list with a sentinel node
7  Lock *request_lock;  // protects the list
8
9  ...
10 Request *req, *tmp;
11 {req ⊩ sp..(ss + StackSize − 1) ↦ _ ∧ sp = ss + 2 · sizeof(Request∗)}
12 lock(request_lock);
13 {req ⊩ ∃x,y,z. &request_queue ↦ z ∗ z.prev ↦ y ∗ z.next ↦ x ∗ z.data ↦ _ ∗
14  dll_Λ(x,z,z,y) ∗ locked(request_lock) ∗
15  sp..(ss + StackSize − 1) ↦ _ ∧ sp = ss + 2 · sizeof(Request∗)}
16 req = request_queue->next;
17 {req ⊩ ∃y,z. &request_queue ↦ z ∗ z.prev ↦ y ∗ z.next ↦ req ∗ z.data ↦ _ ∗
18  dll_Λ(req,z,z,y) ∗ locked(request_lock) ∗
19  sp..(ss + StackSize − 1) ↦ _ ∧ sp = ss + 2 · sizeof(Request∗)}
20 while (req != request_queue) {
21   {req ⊩ ∃x,y,z,u,v. &request_queue ↦ z ∗ z.prev ↦ v ∗ z.next ↦ x ∗ z.data ↦ _ ∗
22    dll_Λ(x,z,req,y) ∗ req.prev ↦ y ∗ req.next ↦ u ∗ req.data ↦ _ ∗ dll_Λ(u,req,z,v) ∗
23    locked(request_lock) ∗ sp..(ss + StackSize − 1) ↦ _ ∧
24    sp = ss + 2 · sizeof(Request∗)}
25   if (stale_data(req->data)) remove_node(req);
26   {req ⊩ ∃x,y,z,u,v. &request_queue ↦ z ∗ z.prev ↦ v ∗ z.next ↦ x ∗ z.data ↦ _ ∗
27    dll_Λ(x,z,u,y) ∗ req.prev ↦ y ∗ req.next ↦ u ∗ req.data ↦ _ ∗ dll_Λ(u,y,z,v) ∗
28    locked(request_lock) ∗ sp..(ss + StackSize − 1) ↦ _ ∧
29    sp = ss + 2 · sizeof(Request∗)}
30   tmp = req;
31   req = req->next;
32   free(tmp);
33   {req ⊩ ∃x,y,z,v. &request_queue ↦ z ∗ z.prev ↦ v ∗ z.next ↦ x ∗ z.data ↦ _ ∗
34    dll_Λ(x,z,req,y) ∗ dll_Λ(req,y,z,v) ∗ locked(request_lock) ∗
35    sp..(ss + StackSize − 1) ↦ _ ∧ sp = ss + 2 · sizeof(Request∗)}
36 }
37 {req ⊩ ∃x,y,z. &request_queue ↦ z ∗ z.prev ↦ y ∗ z.next ↦ x ∗ z.data ↦ _ ∗
38  dll_Λ(x,z,z,y) ∗ locked(request_lock) ∗
39  sp..(ss + StackSize − 1) ↦ _ ∧ sp = ss + 2 · sizeof(Request∗)}
40 unlock(request_lock);
41 {req ⊩ sp..(ss + StackSize − 1) ↦ _ ∧ sp = ss + 2 · sizeof(Request∗)}
```

Fig. 4. An example system call part. The assertions are explained in Section 4.

thus, while verifying the OS, we have to prove that the two components do not corrupt each other's data structures.

The `schedule` function. According to the semantics of our machine, when `schedule` starts executing, interrupts are disabled and the previous values of `ip` and the general-purpose registers are saved on the top of the stack. The scheduler uses the empty slots on the stack of the process it has preempted to store activation records of its procedures

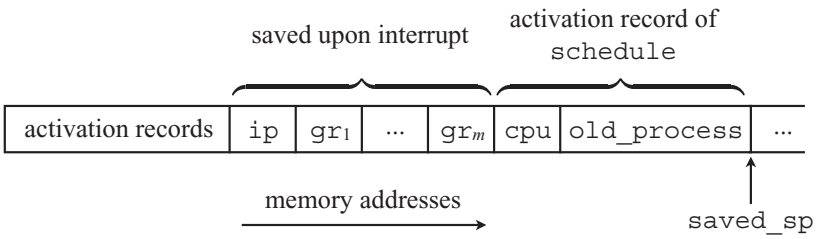


Fig. 5. The invariant of the stack of a preempted process.

and thus expects the kernel to leave enough of these. Intuitively, while a process is running, only this process has the right to access its stack, i.e., *owns* it. When the scheduler preempts the process, the right to access the empty slots on the stack (their *ownership*) is transferred to the scheduler. When the scheduler returns the control to this process, it transfers the ownership of the stack slots back. This is one example of ownership transfer we have to reason about.

The `schedule` function first calls `load_balance` (line 32 in Figure 2), which migrates processes between CPUs to balance the load; we describe it below. It then updates the timeslice of the currently running process, and if it becomes zero, proceeds to schedule another one (line 35); here we abstract from the particular way the timeslice is updated. The processes are scheduled in a round-robin fashion; thus, the function inserts the current process at the end of the local runqueue (line 38) and dequeues the process at the front of the runqueue, making it current (line 40).[2] The former is done using a library function `insert_node_after`, which inserts the node given as its second argument after the list node given as its first argument. The `schedule` function also refills the scheduling quantum of the process being descheduled (line 36). The runqueue manipulations are done with the corresponding lock held (lines 37 and 45). Note that in a realistic OS choosing a process to run would be more complicated, but still based on scheduler-private data structures protected by runqueue locks.

To save the state of the process being preempted, `schedule` copies `sp` into the `saved_sp` field of the process descriptor (line 41 in Figure 2). This field, together with the `kernel_stack` of the process, forms its saved state. The stack of a preempted process contains the activation records of functions called before the process was preempted, the label of the instruction to resume the process from and the values of general-purpose registers, saved upon the interrupt, as well as the activation record of `schedule`, as shown in Figure 5. This invariant holds for descriptors of all preempted processes.

The actual context switch is performed by the assignment to `sp` (line 42), which switches the current stack to another one satisfying the invariant in Figure 5. Since this changes the activation record of `schedule`, the function has to update the `cpu` variable (line 43), which lets it then retrieve and load the new value of `ss` (line 44). The `iret` command at the end of `schedule` (line 46) loads the values of the registers stored on the stack and enables interrupts, thus completing the context switch.

[2] Actually, in Linux 2.6.11 the descriptor of the current process stays in the runqueue. We dequeue it because this simplifies the following formal treatment of the example.

The `load_balance` function checks if the CPU given as its parameter is underloaded and, if it is the case, tries to migrate a process from another CPU to this one. The particular way the function performs the check and chooses the process is irrelevant for our purposes, and is thus abstracted by a random choice (line 50 in Figure 3). To migrate a process, the function chooses a runqueue to steal a process from (line 51) and locks it together with the current runqueue in the order determined by the corresponding CPU identifiers, to avoid deadlocks (lines 52–54). The function then removes one process from the victim runqueue (line 57), if it is non-empty (line 55), and inserts the process into the runqueue of the CPU it runs on (line 58). Note that two concurrent scheduler invocations executing `load_balance` on different CPUs may try to access the same runqueue. While verifying the scheduler, we have to ensure that they synchronise their accesses correctly. We also need to deal with the fact that, due to `load_balance`, processes cannot be tied to a CPU statically.

The `create` function inserts the descriptor of a newly created process with the address given as its parameter into the runqueue of the current CPU. We pass the parameter via a register, as this simplifies the following treatment of the example. The descriptor must be initialised like that of a preempted process, and hence its stack must satisfy the invariant in Figure 5. Upon a call to `create`, the ownership of the descriptor is transferred from the kernel to the scheduler. The `create` function must be called using `icall`, which disables interrupts; if interrupts were enabled, `schedule` could be called while `create` holds the lock for the current runqueue, resulting in a deadlock.

The `fork` function is formally not part of the scheduler. It illustrates how the rest of the kernel can use `create` to implement a common system call that creates a clone of the current process. This function allocates a new descriptor (line 76), copies the stack of the current process to it (line 77) and initialises the stack as expected by `create` (Figure 5). This amounts to discarding the topmost activation record of `fork` and pushing a fake activation record of `schedule` (line 78). We do not initialise the latter record, since `schedule` refreshes the values of the variables (line 43) when it receives control. Note that the values of registers in the initial state of the new process have been saved on the stack upon the call to `fork`. Since stack slots for return values are initialised with zeros, this is what `fork` in the child process will return; we return 1 in the parent process.

The need for modularity. We could try to verify the scheduler and the rest of the kernel (including, say, the system call in Figure 4) as a whole, modelling every CPU as a process in one of the existing program logics for concurrency (O'Hearn, 2007; Gotsman *et al.*, 2007; Feng *et al.*, 2007a; Vafeiadis & Parkinson, 2007; Dinsdale-Young *et al.*, 2010; Cohen *et al.*, 2010; Dinsdale-Young *et al.*, 2013). However, in this case our proofs would have to consider the possibility of the control flow going from any statement in a process to the `schedule` function, and from there to any other process. Thus, in reasoning about the system call implementation in Figure 4 we would end up having to reason explicitly about invariants and actions of both `schedule` and all other processes, making the reasoning non-modular and, most likely, intractable. In the rest of the paper, we propose a logic that avoids this pitfall.

2.3 Approach

Before presenting our logic for preemptive kernels in detail, we give an informal overview of the reasoning principles behind it. The goal of the logic is to reason about the kernel and the scheduler separately. Following previous work on OS verification (Feng *et al.*, 2008b; Yang & Hawblitzel, 2010), our logic uses different proof systems for this purpose: the high-level one for the kernel and the low-level one for the scheduler.

Modular reasoning via memory partitioning. The first challenge we have to deal with in separating the reasoning about the kernel and the scheduler is the fact that they share the same address space. To this end, our logic partitions the memory into two disjoint parts. The memory cells in each of the parts are *owned* by the corresponding component, meaning that only this component can access them. In our running example, the runqueues from Figure 2 will belong to the scheduler, and the request queue from Figure 4 to the kernel. It is important to note that this partitioning does not exist in the semantics, but is enforced by proofs in the logic to enable modular reasoning about the system. Modular reasoning becomes possible because, while reasoning about one component, one does not have to consider the memory partition owned by the other, since it cannot influence the behaviour of the component. An important feature of our logic, required for handling schedulers from mainstream kernels, is that the memory partitioning is not required to be static: the logic permits ownership transfer of memory cells between the areas owned by the scheduler and the kernel according to an axiomatically defined interface. For example, in reasoning about the scheduler of Section 2.2, the logic permits the transfer of the descriptor for a new process from the kernel to the scheduler at a call to `create`; this descriptor then becomes part of a runqueue owned by the scheduler. Of course, assigning the ownership of parts of memory to OS components requires checking that a component does not access the memory it does not own. To this end, our logic implements a form of rely-guarantee reasoning between the scheduler and the kernel, where one component assumes that the other does not touch its memory partition and provides well-formed pieces of memory at ownership transfer points.

Concurrent separation logic. Our logic is based on concurrent separation logic (O'Hearn, 2007), which we recap in Section 4. In particular, this logic provides us with means for modular reasoning within a given component, i.e., either among concurrent OS processes or concurrent scheduler invocations on different CPUs. The choice of concurrent separation logic was guided by the convenience of presentation; see Section 8 for a discussion of how more advanced logics can be integrated. However, the use of a version of separation logic is crucial, because we inherently rely on the *frame property* validated by the logic: the memory that is not mentioned by assertions in a proof of a command is guaranteed not to be changed by it. As we have noted, while reasoning about a component, we consider only the memory partition belonging to it. Hence, by the frame property we automatically know that the component cannot modify the others. This makes it easy to carry out the above-mentioned rely-guarantee reasoning between the scheduler and the kernel: one does not need to state assumptions about one component not

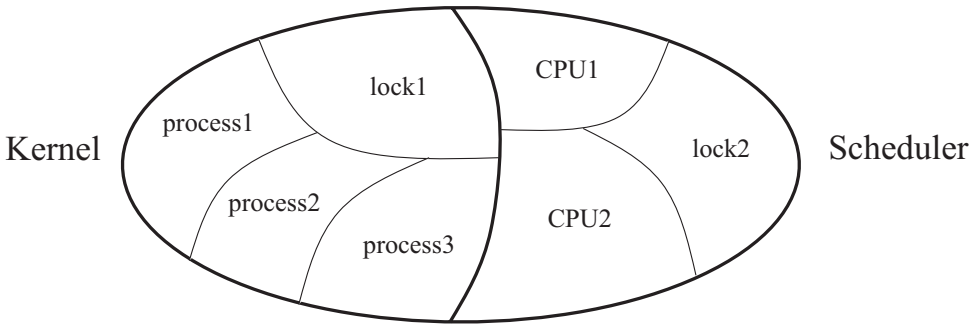


Fig. 6. The partitioning of the system state enforced by the logic. The memory is partitioned into two parts, owned by the scheduler and the kernel, respectively. The memory of each component is further partitioned into parts local to processes or scheduler invocations on a given CPU, and parts protected by locks.

modifying the memory of the other explicitly, as they will be automatically validated by the logic.

Concurrent separation logic lets us achieve modular reasoning within a given component by further partitioning the memory owned by it into disjoint process-local parts (one for each process or scheduler invocation on a given CPU) and protected parts (one for each free lock). A process-local part can only be accessed by the corresponding process or scheduler invocation, and a lock-protected part only when the process holds the lock. The resulting partitioning of the system state is illustrated in Figure 6. The frame property guarantees that a process cannot access the partition of the heap belonging to another one. To reason modularly about parts of the state protected by locks, the logic associates with every lock an assertion—its *lock invariant*—that describes the part of the state it protects. Lock invariants restrict how processes can change the protected state and, hence, allow reasoning about them in isolation. For example, in the program from Figure 4, the invariant of `request_lock` can state that it protects the `request_queue` variable and the doubly-linked list it identifies.

Scheduler-agnostic verification of kernel code. The high-level proof system (Section 5.2) reasons about preemptable code assuming an abstract machine where every process has its own virtual CPU with a dedicated set of registers. It relies on the partitioned view of memory described above to hide the state of the scheduler, with all the remaining state split among processes and locks accessible to them, as illustrated in Figure 7. We have primed process identifiers in the figure to emphasise that the state of the process can be represented differently in the abstract and physical machines: for example, if a process is not running, the values of its local registers can be stored in scheduler-private data structures, rather than in CPU registers.

Apart from hiding the state of the scheduler, the high-level system also hides the complex manipulation of the control flow performed by its context-switch code: the proof system assumes that the control moves from one point in the process code to the next without changing its state, ignoring the possibility of the context-switch code getting executed upon an interrupt. This is expressed by handling sequential composition in the proof system essentially using the standard rule (1) from Hoare logic. Explicit calls to the scheduler are treated as if they were executed atomically.

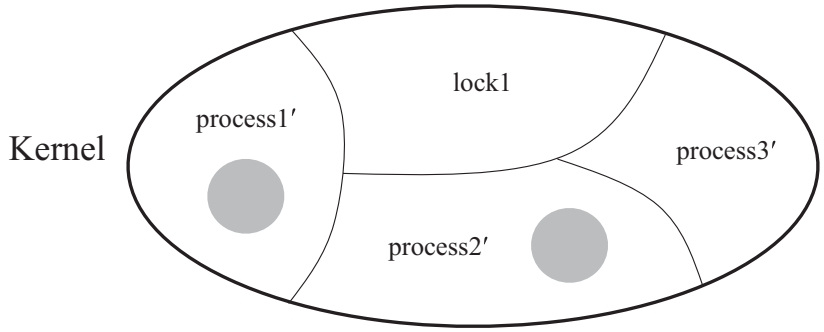


Fig. 7. The state of the abstract system with one virtual CPU per process. Process identifiers are primed to emphasise that the state of the process can be represented differently in the abstract and physical machines (cf. Figure 6). Dark regions illustrate the parts of process state that are tracked by a scheduler invocation running on a particular physical CPU.

Technically, the proof system is a straightforward adaptation of concurrent separation logic, which is augmented with proof rules axiomatising the effect of scheduler routines explicitly called by processes. The novelty here is that we can use such a scheduler-agnostic logic in this context at all. This is made possible by verifying the scheduler implementation using the low-level proof system (Section 5.3).

Proving schedulers correct. Intuitively, a correct scheduler provides an illusion of the above-mentioned abstract machine with one CPU per process, but what formal obligations does this entail? First, a process should not notice any effects of being preempted and then scheduled again: whenever the scheduler sets up a process to run on a CPU, it has to restore the state of the CPU registers to the one the process had last time it was preempted. The low-level proof system ensures this by recording these values in a special predicate Process, which can be viewed as an assertion about the continuation of a process describing the states from which it can be safely resumed.

In more detail, when a process is preempted and the control is given to the context-switch routine of the scheduler (`schedule` in the example from Section 2.2), a Process predicate recording the current values of the CPU registers appears in its precondition. When the context-switch routine terminates and the control is given to the process it resumes, the proof system requires the postcondition of the routine to exhibit a Process predicate with register values equal to the ones loaded onto the CPU. Roughly speaking, we thus require the following judgement to hold of the context-switch routine `schedule`:

$$\{\exists \vec{r}.\, \mathsf{Process}(\vec{r}) \wedge \vec{\mathtt{r}} = \vec{r} \ldots\}\ \mathtt{schedule}\ \{\exists \vec{r'}.\, \mathsf{Process}(\vec{r'}) \wedge \vec{\mathtt{r}} = \vec{r'} \ldots\}, \tag{2}$$

where $\vec{\mathtt{r}}$ is the vector of CPU register names. The Process predicate in the postcondition may correspond to a different process than the predicate in the precondition. For example, when the scheduler from Section 2.2 preempts a process and links its descriptor into a runqueue, assertions about the runqueue in the proof system can record the corresponding Process predicate with register values equal to the ones stored in the descriptor. This predicate can then be used to establish the postcondition of the context-switch routine when it decides to schedule the process again.

In the case of multiprocessors, ensuring that the scheduler preserves the state of a preempted process is not enough for it to be correct. The scheduler should also be prevented from duplicating processes at will: it would be incorrect to preempt one process and then schedule two copies of it on two CPUs at the same time. To check that this does not happen, the proof system interprets a Process predicate as not merely recording a process state, but serving as an exclusive permission for the scheduler invocation owning it to schedule the corresponding process. Technically, it treats Process predicates affinely, prohibiting their duplication. Judgement (2) is then interpreted as stating that the scheduler gets the ownership of a Process predicate when it preempts a process and gives it up when scheduling that process again. This ensures that, at any time, only a scheduler invocation on a single CPU can own a Process predicate for a given process and, hence, can schedule it. In terms of Figure 6, a Process predicate can only belong to one partition in the scheduler-owned memory at a time.

We note that the problem of interdependence between the correctness of the scheduler and the rest of the kernel that we address in this paper also arises in preemptive *uniprocessor* kernels. The Process predicate also allows us to verify schedulers on uniprocessors; however, its affine treatment described above is not relevant in this case.

Assertions of the low-level proof system can be thought of as relating the states of the concrete machine and the abstract one the scheduler is supposed to implement. An important feature of our logic is that this relation is *local*, in the sense that it does not describe the whole state of the two machines. Namely, since we use concurrent separation logic to reason about concurrent execution of scheduler invocations on different CPUs, an assertion in the low-level proof system describes only the state owned by a scheduler invocation on a particular CPU (e.g., the region marked CPU1 in Figure 6). Similarly, the Process predicates describe only the registers of the processes a scheduler invocation has permission to schedule (shown by the dark regions in Figure 7), but not the memory they own. Since the assertions about the scheduler cannot talk about the memory owned by kernel processes, the frame property automatically ensures that the scheduler cannot corrupt it.

Soundness. We establish the soundness of our logic using an approach atypical for the kind of setting we consider. Since a scheduler is supposed to provide an illusion of an abstract machine with one CPU per process, to formalise its correctness, we could define an operational semantics of such an abstract machine and prove that it reproduces any behaviour of the concrete machine with the scheduler, thus establishing a *refinement* between the two machines. However, for realistic OS schedulers defining a semantics for the abstract machine that a scheduler implements is difficult. This is because, in reasoning about mainstream operating systems, the state transferred between the scheduler and the kernel can be described by complicated assertions; in such cases, defining ownership transfer operationally is difficult (we discuss this further in Section 8).

To resolve this problem with stating soundness, we do not define the semantics of the abstract machine operationally; instead, we describe its behaviour only by the high-level proof system, thus giving it an axiomatic semantics. As expected, the low-level proof system is used to reason about the correspondence between the concrete and the abstract machines, with its assertions relating their states. However, proofs in neither of the two systems are interpreted with respect to any semantics alone: our soundness statement

(Section 8) interprets a proof of the kernel in the high-level system and a proof of the scheduler in the low-level one together with respect to the semantics of the concrete machine. Thus, instead of relating sets of executions of the two machines, like in the classical refinement, the soundness statement relates logical statements about the abstract machine (given by high-level proofs) to logical statements about the concrete one (given by a constraint on concrete states). Note that in this case the soundness statement for the logic does not yield a semantic statement of correctness for the scheduler being considered in isolation. Rather, its correctness is established indirectly by the fact that reasoning in the high-level proof system, which assumes the abstract one-CPU-per-process machine, is sound with respect to the concrete machine.

To formulate the soundness statement in the above way, we need to construct a global property about the whole system state shown in Figure 6 from local assertions about its components, corresponding to partitions in Figure 6. This does not just boil down to combining the assertions about the partitions using the separating conjunction, since the high-level and low-level proof systems work on different levels of abstraction. In particular, when conjoining assertions about the scheduler and the kernel, we need to make sure that the view of the parts of kernel state in the assertions about the scheduler (dark regions in Figure 7) is consistent with those about the kernel. This requires a delicate construction, combining relational composition and separating conjunction. Similar constructions can potentially be used for justifying other program logics working on several levels of abstraction at the same time.

3 Machine semantics

In this section, we give a formal semantics to the example machine informally presented in Section 2.1.

3.1 Storage model

Figure 8 gives a model for the set of configurations Config that can arise during an execution of the machine. A machine configuration is a triple with the components describing the values of registers of the CPUs in the machine (its *global context*), the state of the heap and the set of locks taken by some CPU (the *lockset* of the machine). Note that we allow the global context or the heap to be a partial function. However, the corresponding configurations are not encountered in the semantics we define in this section. They come in handy in Sections 5 and 8 to give a semantics to the assertion language and express the soundness of our logic.

In this paper, we use the following notation for partial functions: $f[x : y]$ is the function that has the same value as f everywhere, except for x, where it has the value y; $[]$ is a nowhere-defined function; $f \uplus g$ is the union of the disjoint partial functions f and g.

3.2 Programming language

We consider a low-level language where programs are represented by structures similar to control-flow graphs. The programs are constructed from primitive commands c, whose syntax we define in Figure 9. In addition to the primitive commands listed in Section 2,

$$
\begin{array}{lll}
k \in \mathsf{CPUid} & = \{1,\ldots,\mathtt{NCPUS}\} & \mathtt{r} \in \mathsf{Reg} = \{\mathtt{ip},\mathtt{if},\mathtt{ss},\mathtt{sp},\mathtt{gr}_1,\ldots,\mathtt{gr}_m\} \\
r \in \mathsf{Context} & = \mathsf{Reg} \to \mathsf{Val} & R \in \mathsf{GContext} = \mathsf{CPUid} \rightharpoonup \mathsf{Context} \\
\ell \in \mathsf{Lock} & = \{\ell_1,\ell_2,\ldots,\ell_n\} & L \in \mathsf{Lockset} = \mathscr{P}(\mathsf{Lock}) \\
h \in \mathsf{Heap} & = \mathsf{Loc} \rightharpoonup \mathsf{Val} & (R,h,L) \in \mathsf{Config} = \mathsf{GContext} \times \mathsf{Heap} \times \mathsf{Lockset}
\end{array}
$$

Fig. 8. The set of machine configurations Config. We assume sets Loc of valid memory addresses and Val of values such that $\mathsf{Loc} \subseteq \mathsf{Val}$.

$$
\begin{array}{lcl}
\mathtt{r} & \in & \mathsf{Reg} - \{\mathtt{ip},\mathtt{if}\} \\
\ell & \in & \mathsf{Lock} \\
l & \in & \mathsf{Label} = \mathbb{N} \\
e & ::= & \mathtt{r} \mid 0 \mid 1 \mid 2 \mid \ldots \mid e+e \mid e-e \\
b & ::= & e=e \mid e \leqslant e \mid b \wedge b \mid b \vee b \mid \neg b \\
c & ::= & \mathtt{skip} \mid \mathtt{r} := e \mid \mathtt{r} := [e] \mid [e] := e \mid \mathtt{assume}(b) \mid \mathtt{lock}(\ell) \mid \mathtt{unlock}(\ell) \\
 & \mid & \mathtt{savecpuid}(e) \mid \mathtt{call}(l) \mid \mathtt{icall}(l) \mid \mathtt{ret} \mid \mathtt{iret}
\end{array}
$$

Fig. 9. Primitive commands.

we have the following ones: `skip` and $\mathtt{r} := e$ have the standard meaning; $\mathtt{r} := [e]$ reads the contents of a heap cell e and assigns the value read to `r`; $[e] := e'$ updates the contents of cell e by e'; $\mathtt{assume}(b)$ acts as a filter on the state space of programs, choosing states satisfying b. The `assume` command is used to treat branches in conditionals and loops uniformly with the other primitive commands, as we explain below. We write PComm for the set of primitive commands. Note that primitive commands cannot access the `ip` register directly. Also, only `icall` and `iret` can affect the `if` register, a restriction that we lift in Section 7.

Commands C are partial maps from Label to $\mathsf{PComm} \times \mathscr{P}(\mathsf{Label})$. Intuitively, if $C(l) = (c,X)$, then c is labelled by l in C and can be followed by any command with a label in X. In this case, we let $\mathsf{comm}(C,l) = c$ and $\mathsf{next}(C,l) = X$. We denote the domain of C by $\mathsf{labels}(C)$ and the set of all commands by Comm.

The language constructs used in the example scheduler of Section 2, such as sequential composition, loops and conditionals, can be expressed as commands in a standard way, with conditions translated using `assume`. We illustrate this in Figure 10, where we represent the mapping C by a graph, with nodes annotated by labels and primitive commands and edges defining the next function.

3.3 Operational semantics

We now give a standard operational semantics to our programming language. We interpret primitive commands c using a transition relation $\rightsquigarrow_c$ of the following type:

$$
\begin{array}{c}
\mathsf{State} \;=\; \mathsf{Context} \times \mathsf{Heap} \times \mathsf{Lockset}; \qquad (3) \\
\rightsquigarrow_c \;\subseteq\; \big(\mathsf{CPUid} \times \mathsf{State} \times \mathsf{Label} \times \mathsf{Label}\big) \times \big((\mathsf{State} \times \mathsf{Label}) \cup \{\top\}\big).
\end{array}
$$

The input $(k,(r,h,L),l,l')$ to $\rightsquigarrow_c$ consists of the following components:

- $k \in \mathsf{CPUid}$ is the identifier of the CPU executing the command.

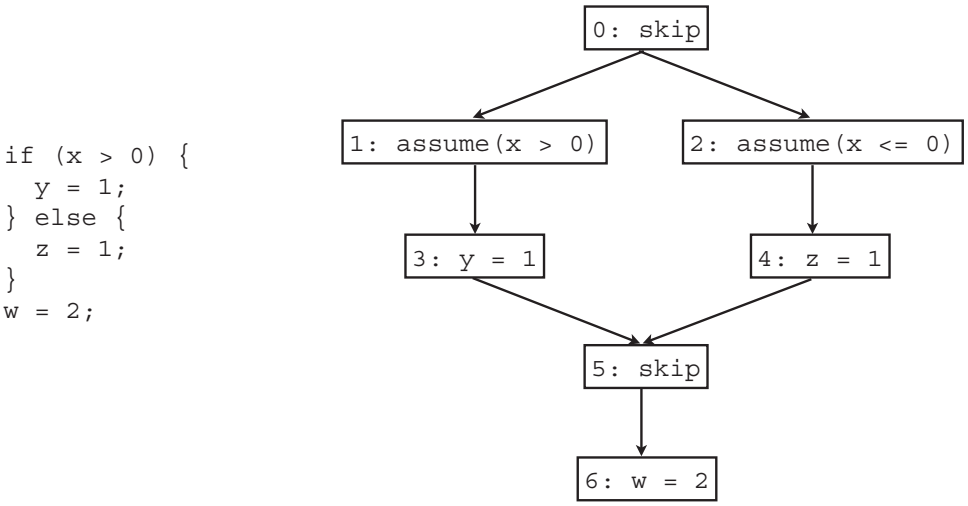


Fig. 10. Representing sequential composition and conditionals in our low-level language.

- (r,h,L) is the configuration of the system projected to this CPU, which we call a *state*. It includes the context of the CPU and the information about the shared resources—the heap and locks.
- $l \in \mathsf{Label}$ is the label of the command c.
- $l' \in \mathsf{Label}$ is the label of a primitive command following c in the program.

Given this input, the transition relation $\rightsquigarrow_c$ for c computes the next state of the CPU after running c, together with the label of the primitive command to run next. The former may be a special $\top$ state signalling a machine crash. The latter may be different from l' when c is a call or a return.

The relation $\rightsquigarrow_c$ is defined in Figure 11. In the figure and in the rest of the paper, we write $\_$ for an expression whose value is irrelevant and implicitly existentially quantified and $\vec{\mathtt{gr}}$ for the vector of general-purpose registers. The relation follows the informal meaning of primitive commands given in Sections 2.1 and 3.2. We have omitted standard definitions for `skip` and most of assignments (Reynolds, 2002). We have also omitted them for `icall` and `iret`: the definitions are the same as for `call` and `ret`, but additionally modify `if`. Note that $\rightsquigarrow_c$ may yield no poststate for a given prestate. Unlike a transition to the $\top$ state, this represents the command getting stuck. For example, according to Figure 11, acquiring the same lock twice leads to a deadlock, and releasing a lock that is not held crashes the system.

The program our machine executes is given by a command C that includes a primitive command labelled `schedule`, serving as the entry point of the interrupt handler. For such a command C, we give its meaning using a small-step operational semantics, formalised by the transition relation $\rightarrow_C \subseteq \mathsf{Config} \times (\mathsf{Config} \cup \{\top\})$ in Figure 12. The first rule in the figure describes a normal execution, where the value l of the `ip` register of CPU k is used to choose the primitive command c to run. After choosing c, the machine nondeterministically picks a label $l' \in \mathsf{next}(C,l)$ identifying the command to follow c, runs c according to the semantics $\rightsquigarrow_c$, and uses the result of this run to update the registers of CPU k and the heap and the lockset of the machine. For example, when a CPU executes the program in Figure 10 from label `0`, both labels `1` and `2` following it will be explored; however, only the branch where the `assume` condition evaluates to true will proceed further.

$$
\begin{array}{lll}
(k,(r,h[[[e]]r:u],L),l,l') & \rightsquigarrow_{\mathtt{r}:=[e]} & ((r[\mathtt{r}:u],h[[[e]]r:u],L),l') \\
(k,(r,h,L),l,l') & \rightsquigarrow_{\mathtt{assume}(b)} & ((r,h,L),l'),\ \text{if } [[b]]r=\mathsf{true} \\
(k,(r,h,L),l,l') & \not\rightsquigarrow_{\mathtt{assume}(b)} & \text{if } [[b]]r=\mathsf{false} \\
(k,(r,h,L),l,l') & \rightsquigarrow_{\mathtt{lock}(\ell)} & ((r,h,L\cup\{\ell\}),l'),\ \text{if } \ell\notin L \\
(k,(r,h,L),l,l') & \not\rightsquigarrow_{\mathtt{lock}(\ell)} & \text{if } \ell\in L \\
(k,(r,h,L),l,l') & \rightsquigarrow_{\mathtt{unlock}(\ell)} & ((r,h,L-\{\ell\}),l'),\ \text{if } \ell\in L \\
(k,(r,h[[[e]]r:\_],L),l,l') & \rightsquigarrow_{\mathtt{savecpuid}(e)} & ((r,h[[[e]]r:k],L),l') \\
(k,(r,h[r(\mathtt{sp})..(r(\mathtt{sp})+m):\_],L),l,l') & \rightsquigarrow_{\mathtt{call}(l'')} & \\
\multicolumn{3}{r}{((r[\mathtt{sp}:r(\mathtt{sp})+m+1],h[r(\mathtt{sp}):l',(r(\mathtt{sp})+1)..(r(\mathtt{sp})+m):r(\vec{\mathtt{gr}})],L),l'')} \\
\multicolumn{3}{l}{(k,(r,h[r(\mathtt{sp})-m-1:l'',(r(\mathtt{sp})-m)..(r(\mathtt{sp})-1):\vec{g}],L),l,l') \rightsquigarrow_{\mathtt{ret}}} \\
\multicolumn{3}{r}{((r[\mathtt{sp}:r(\mathtt{sp})-m-1,\vec{\mathtt{gr}}:\vec{g}],h[r(\mathtt{sp})-m-1:l'',(r(\mathtt{sp})-m)..(r(\mathtt{sp})-1):\vec{g}],L),l'')} \\
(k,(r,h,L),l,l') & \rightsquigarrow_{c} & \top,\ \text{otherwise}
\end{array}
$$

Fig. 11. Semantics of primitive commands. The notation $\rightsquigarrow_c \top$ indicates that the command c crashes, and $\not\rightsquigarrow_c$ that it does not crash, but gets stuck. The function $[[\cdot]]r$ evaluates expressions with respect to the context r.

$$
\frac{r(\mathtt{ip})=l\in\mathsf{labels}(C)\qquad l'\in\mathsf{next}(C,l)\qquad (k,(r,h,L),l,l')\rightsquigarrow_{\mathsf{comm}(C,l)}((r',h',L'),l'')}{(R[k:r],h,L)\rightarrow_C(R[k:r'[\mathtt{ip}:l'']],h',L')}
$$

$$
\frac{r(\mathtt{ip})=l\in\mathsf{labels}(C)\qquad r(\mathtt{if})=1\qquad (k,(r,h,L),l,l)\rightsquigarrow_{\mathtt{icall}(\mathtt{schedule})}((r',h',L'),l'')}{(R[k:r],h,L)\rightarrow_C(R[k:r'[\mathtt{ip}:l'']],h',L')}
$$

$$
\frac{r(\mathtt{ip})=l\in\mathsf{labels}(C)\qquad l'\in\mathsf{next}(C,l)\qquad (k,(r,h,L),l,l')\rightsquigarrow_{\mathsf{comm}(C,l)}\top}{(R[k:r],h,L)\rightarrow_C\top}
$$

$$
\frac{r(\mathtt{ip})\notin\mathsf{labels}(C)}{(R[k:r],h,L)\rightarrow_C\top}
$$

$$
\frac{r(\mathtt{if})=1\qquad \{r(\mathtt{sp}),\ldots,r(\mathtt{sp})+m\}\not\subseteq\mathsf{dom}(h)}{(R[k:r],h,L)\rightarrow_C\top}
$$

Fig. 12. Operational semantics of the machine.

The second rule in Figure 12 concerns interrupts. Upon an interrupt, the interrupt handler label `schedule` is loaded into `ip`, and the label of the command to execute after the handler returns is pushed onto the stack together with the values of the general-purpose registers. The remaining rules deal with crashes arising from erroneous execution of primitive commands, undefined command labels and a stack overflow upon an interrupt.

4 Baseline concurrency logic

We start by presenting the variation of concurrent separation logic on which our logic for verifying preemptive kernels is based (Section 5). The logic we present in this section assumes that interrupts are disabled on all CPUs. Thus, we assume that all `if` registers are initially set to zero and only consider programs that do not use `icall`, `iret` and `savecpuid` commands. One can thus think of a single process having been pinned to every

$$
\begin{array}{rcl}
x,y & \in & \mathsf{NVar} \\
\gamma & \in & \mathsf{CVar} \\
\mathtt{r} & \in & \mathsf{Reg} - \{\mathtt{ip},\mathtt{if}\} \\
\mathbf{r} & \in & \{\mathbf{ip},\mathbf{if},\mathbf{ss},\mathbf{sp},\mathbf{gr}_1,\ldots,\mathbf{gr}_m\} \\
E & ::= & x \mid \mathtt{r} \mid 0 \mid 1 \mid \ldots \mid E+E \mid E-E \mid G(\mathbf{r}) \\
G & ::= & \gamma \mid [\mathbf{ip}:E,\mathbf{if}:E,\mathbf{ss}:E,\mathbf{sp}:E,\vec{\mathbf{gr}}:\vec{E}] \\
\Sigma & ::= & \varepsilon \mid E \mid \Sigma\Sigma \\
B & ::= & E=E \mid \Sigma=\Sigma \mid G=G \mid E \leqslant E \mid B \wedge B \mid B \vee B \mid \neg B \\
P & ::= & B \mid \mathsf{true} \mid P \wedge P \mid \neg P \mid \exists x.P \mid \exists \gamma.P \\
 & \mid & \mathsf{emp} \mid E \mapsto E \mid E..E \mapsto \Sigma \mid P * P \mid \mathsf{dll}_\Lambda(E,E,E,E) \mid \mathsf{locked}(\ell)
\end{array}
$$

$$
\begin{array}{lll}
(r,h,L) \models_\eta B & \text{iff} & [\![B]\!]_\eta r = \mathit{true} \\
(r,h,L) \models_\eta P_1 \wedge P_2 & \text{iff} & (r,h,L) \models_\eta P_1 \text{ and } (r,h,L) \models_\eta P_2 \\
(r,h,L) \models_\eta \mathsf{emp} & \text{iff} & h = [\,] \text{ and } L = \emptyset \\
(r,h,L) \models_\eta E_0 \mapsto E_1 & \text{iff} & h = [[\![E_0]\!]_\eta r : [\![E_1]\!]_\eta r] \text{ and } L = \emptyset \\
(r,h,L) \models_\eta E_0..E_1 \mapsto \Sigma & \text{iff} & \exists j \geqslant 0.\, \exists u_1,\ldots,u_j \in \mathsf{Val}.\ L=\emptyset,\ j = [\![E_1]\!]_\eta r - [\![E_0]\!]_\eta r + 1, \\
 & & u_1u_2\ldots u_j = [\![\Sigma]\!]_\eta r \text{ and } h = [[\![E_0]\!]_\eta r : u_1,\ldots,[\![E_1]\!]_\eta r : u_j] \\
(r,h,L) \models_\eta \mathsf{locked}(\ell) & \text{iff} & h = [\,] \text{ and } L = \{\ell\} \\
(r,h,L) \models_\eta P_1 * P_2 & \text{iff} & \exists h_1,h_2,L_1,L_2.\ h = h_1 \uplus h_2,\ L = L_1 \uplus L_2, \\
 & & (r,h_1,L_1) \models_\eta P_1 \text{ and } (r,h_2,L_2) \models_\eta P_2
\end{array}
$$

Predicate dll_Λ is the least one satisfying the equivalence below:

$$
\mathsf{dll}_\Lambda(E_h,E_p,E_n,E_t) \iff \exists x.\,(E_h = E_n \wedge E_p = E_t \wedge \mathsf{emp}) \vee E_h.\mathtt{prev} \mapsto E_p * E_h.\mathtt{next} \mapsto x * \Lambda(E_h) * \mathsf{dll}_\Lambda(x,E_h,E_n,E_t)
$$

Fig. 13. Syntax and semantics of assertions of the baseline logic. We have omitted the standard clauses for most of the first-order connectives. The function $[\![\cdot]\!]_\eta r$ evaluates expressions with respect to the context r and the logical variable environment η.

CPU, so that we do not have to consider scheduling. Our logic for preemptive kernels in Section 5 lifts this restriction.

4.1 Assertion language

Mathematically, assertions denote sets of states as defined by (3). However, they describe properties of a single process, rather than the whole machine. Hence, unlike in Section 3.3, here a heap can be a partial function, with its domain defining the part of the heap owned by the process. Similarly, a lockset is now meant to contain only the set of locks that the process has permission to release.

We use a minor extension of the assertion language of separation logic (Reynolds, 2002), whose syntax and semantics are defined in Figure 13. We denote the set of assertions by $\mathsf{Assert}_\mathsf{H}$. We assume disjoint sets NVar and CVar containing logical variables for values and contexts, respectively. The latter is needed for the extension of the current logic to reasoning about preemptive kernels (Section 5). A context G is either a logical variable or

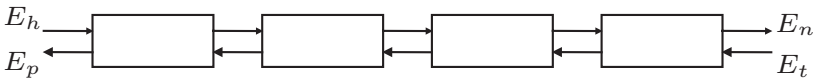


Fig. 14. An illustration of the $\mathsf{dll}_\Lambda(E_h, E_p, E_n, E_t)$ predicate.

a finite map from register labels $\mathbf{r}$ to expressions. Note that we use $\mathbf{r}$ to range over register names in context expressions, but `r` elsewhere in assertions and in programs. Expressions E and Booleans B are similar to those in programs, except that they allow logical variables to appear and include the lookup $G(\mathbf{r})$ of the value of the register $\mathbf{r}$ in the context G. Let a logical variable environment η be a mapping from $\mathsf{NVar} \cup \mathsf{CVar}$ to $\mathsf{Val} \cup \mathsf{Context}$ that respects the types of variables. Assertions denote sets of states from State as defined by the satisfaction relation $\models_\eta$ in Figure 13. For an environment η and an assertion P, we denote the set of states satisfying P by $[\![P]\!]_\eta$.

The assertions in the first line of the definition of P are connectives from the first-order logic with the standard semantics. We can define the missing connectives from the given ones. The assertions in the second line up to dll_Λ are standard assertions of separation logic (Reynolds, 2002). Informally, emp describes the empty heap, and $E \mapsto E'$ the heap with only one cell at the address E containing E'. The assertion $E..E' \mapsto \Sigma$ is the generalisation of the latter to several consecutive cells at the addresses from E to E' inclusive containing the sequence of values Σ. For a value u of a C type `t` taking several cells, we shorten $E..(E + \texttt{sizeof(t)} - 1) \mapsto u$ to just $E \mapsto u$. For a field `f` of a C structure, we use $E.\texttt{f} \mapsto E'$ as a shorthand for $(E + \mathit{off}) \mapsto E'$, where *off* is the offset of `f` in the structure. The separating conjunction $P_1 * P_2$ talks about the splitting of the local state, which consists of the heap and the lockset of the process. It says that a pair (h, L) can be split into two disjoint parts, such that one part (h_1, L_1) satisfies P_1 and the other (h_2, L_2) satisfies P_2.

The assertion $\mathsf{dll}_\Lambda(E_h, E_p, E_n, E_t)$ is an inductive predicate describing a segment of a doubly-linked list (Figure 14). We included it to describe the runqueues of the scheduler in our example; predicates for other data structures can be added straightforwardly (Reynolds, 2002). The predicate assumes a C structure definition with fields `prev` and `next`. Here, E_h is the address of the head of the list, E_t is the address of its tail, E_p is the pointer in the `prev` field of the head node, and E_n is the pointer in the `next` field of the tail node. The Λ parameter is a formula with one free logical variable describing the shape of each node in the list, excluding the `prev` and `next` fields; the logical variable defines the address of the node. For instance, the request queue from Figure 4 can be described by the following assertion:

$$\exists x, y, z.\, \&\texttt{request\_queue} \mapsto z * z.\texttt{prev} \mapsto y * z.\texttt{next} \mapsto x * z.\texttt{data} \mapsto \_ * \mathsf{dll}_\Lambda(x, z, z, y), \quad (4)$$

where $\Lambda(x) = x.\texttt{data} \mapsto \_$.

Finally, the assertion $\mathsf{locked}(\ell)$ is specific to reasoning about concurrent programs and denotes states with an empty local heap and the lockset consisting of ℓ, i.e., it denotes a permission to release the lock ℓ. Note that $\mathsf{locked}(\ell) * \mathsf{locked}(\ell)$ is inconsistent: acquiring the same lock twice leads to a deadlock.

In the following, we write $\mathit{var} \Vdash P$ for a local C variable or procedure parameter `var` instead of $\exists \mathit{var}.\, (\texttt{sp} - \mathit{var\_off}) \mapsto \mathit{var} * P$, where *var_off* is the offset of `var` with respect

to the top of the stack in the activation record of the function where it is declared (note that here `var` is a program variable, whereas *var* is a logical one). Thus, the assertion at line 13 of Figure 4 states that the local state of a process executing the system call consists of the local variables `req` and `tmp`, stored on its stack, the free part of the stack, the doubly-linked list `request_queue` and a permission to release `request_lock`.

To summarise, our assertion language extends that of concurrent separation logic with expressions to denote contexts and locked assertions to keep track of permissions to release locks.

4.2 Proof system

The proof system for the baseline logic is obtained by adapting concurrent separation logic to our low-level language. The judgements of the proof system are of the form $I, \Delta \vdash C$. Here, C specifies the code executed by processes on all CPUs; note that even though C is the same for all of them, the processes can still execute different programs if they start from different program points in C. We explain I below. The parameter $\Delta : \mathsf{Label} \to \mathsf{Assert}_{\mathsf{H}}$ in our judgement specifies local states of a process given the program point it is at; these states correspond to process partitions in Figure 7. It thus induces pre- and postconditions for all primitive commands in C. The top-level rule PROG of the proof system requires us to prove $I, \Delta \rhd_{l'} \{\Delta(l)\}\ c\ \{\Delta(l')\}$ for every primitive command c in C and the label l' of a command following c. This informally means that if c is run from an initial state satisfying $\Delta(l)$, then it accesses only the memory specified by $\Delta(l)$ and either terminates normally and ends up in a state satisfying $\Delta(l')$, or jumps to a label l'' whose assertion $\Delta(l'')$ holds in the current state. The proof rules for the above kind of judgements are also given in Figure 15. They include the standard separation logic axioms for primitive commands, such as ASSUME and STORE; see Reynolds (2002) for the others. Note that PROG treats sequential composition, represented by Δ as illustrated in Figure 10, in the same way as the classical proof rule (1).

The fact that $I, \Delta \rhd_{l'} \{\Delta(l)\}\ c\ \{\Delta(l')\}$ guarantees that c accesses only the memory specified by $\Delta(l)$ validates the frame property (Section 2.3): a process will not step out of the boundaries of its partition in Figure 7. This also allows us to include the FRAME rule of separation logic, which states that executing a command in a bigger local state does not change its behaviour. The rule is useful to restrict the reasoning about primitive commands to the memory they actually access. The rules CONSEQ, DISJ and EXISTS are standard rules of Hoare logic. To keep the logic sound we have to forbid applying EXISTS and FRAME to calls or returns.

The LOCK and UNLOCK axioms are inherited from concurrent separation logic and provide tools for modular reasoning about concurrent processes. They use the mapping $I : \mathsf{Lock} \rightharpoonup \mathsf{Assert}_{\mathsf{H}}$, which specifies the invariants of locks that can be used in C (see Section 2.3). An example of a lock invariant is the assertion (4), which states that the lock `request_lock` from Figure 4 protects a non-empty cyclic doubly-linked list of `Request` nodes with the head node at address `request_queue`. We do not allow lock invariants to contain registers or free occurrences of logical variables and require them to have an empty lockset: $\forall \ell, \eta, (r,h,L) \in [\![I(\ell)]\!]_\eta . L = \emptyset$. The latter does not allow us to prove programs where a lock is released by a CPU other than the one that acquired it, which our machine semantics allows. We put this restriction to simplify the explanation of soundness

$$\frac{\forall l \in \mathsf{labels}(C).\, \forall l' \in \mathsf{next}(C,l).\, (I, \Delta \rhd_{l'} \{\Delta(l)\}\ \mathsf{comm}(C,l)\ \{\Delta(l')\})}{I, \Delta \vdash C}\ \textsc{Prog}$$

$$\frac{}{I, \Delta \rhd_l \{P\}\ \mathtt{assume}(b)\ \{P \wedge b\}}\ \textsc{Assume}$$

$$\frac{}{I, \Delta \rhd_l \{e \mapsto \_\}[e] := e' \{e \mapsto e'\}}\ \textsc{Store}$$

$$\frac{I, \Delta \rhd_l \{P\} c \{Q\} \quad \mathsf{mod}(c) \cap \mathsf{free}(F) = \emptyset \quad c \text{ is not one of } \mathtt{call}, \mathtt{icall}, \mathtt{ret} \text{ and } \mathtt{iret}}{I, \Delta \rhd_l \{P * F\} c \{Q * F\}}\ \textsc{Frame}$$

$$\frac{P \implies P' \quad I, \Delta \rhd_l \{P'\}\ c\ \{Q'\} \quad Q' \implies Q}{I, \Delta \rhd_l \{P\}\ c\ \{Q\}}\ \textsc{Conseq}$$

$$\frac{I, \Delta \rhd_l \{P\}\ c\ \{Q\} \quad c \text{ is not one of } \mathtt{call}, \mathtt{icall}, \mathtt{ret} \text{ and } \mathtt{iret}}{I, \Delta \rhd_l \{\exists x.\, P\}\ c\ \{\exists x.\, Q\}}\ \textsc{Exists}$$

$$\frac{I, \Delta \rhd_l \{P_1\}\ c\ \{Q_1\} \quad I, \Delta \rhd_l \{P_2\}\ c\ \{Q_2\}}{I, \Delta \rhd_l \{P_1 \vee P_2\}\ c\ \{Q_1 \vee Q_2\}}\ \textsc{Disj}$$

$$\frac{}{I, \Delta \rhd_l \{\mathsf{emp}\}\ \mathtt{lock}(\ell)\ \{I(\ell) * \mathsf{locked}(\ell)\}}\ \textsc{Lock}$$

$$\frac{}{I, \Delta \rhd_l \{I(\ell) * \mathsf{locked}(\ell)\}\ \mathtt{unlock}(\ell)\ \{\mathsf{emp}\}}\ \textsc{Unlock}$$

$$\frac{(P * (\mathtt{sp}..(\mathtt{sp}+m) \mapsto l\, \mathtt{gr}_1 \ldots \mathtt{gr}_m)) \implies (\Delta(l')[(\mathtt{sp}+m+1)/\mathtt{sp}])}{I, \Delta \rhd_l \{P * (\mathtt{sp}..(\mathtt{sp}+m) \mapsto \_)\}\ \mathtt{call}(l')\ \{Q\}}\ \textsc{Call}$$

$$\frac{\forall l' \in \mathsf{Label}.(P * ((\mathtt{sp}-m-1)..(\mathtt{sp}-1) \mapsto E'\vec{E}) \wedge E' = l') \implies (\Delta(l')[(\mathtt{sp}-m-1)/\mathtt{sp}][\vec{E}/\vec{\mathtt{gr}}])}{I, \Delta \rhd_l \{P * ((\mathtt{sp}-m-1)..(\mathtt{sp}-1) \mapsto E'\vec{E})\}\mathtt{ret}\{Q\}}\ \textsc{Ret}$$

Fig. 15. Proof system of the baseline logic. Here, $\mathsf{mod}(c)$ is the set of registers modified by c, and $\mathsf{free}(F)$ is the set of registers appearing in F.

in Section 8. We consider a version of concurrent separation logic where lock invariants are allowed to be imprecise (O'Hearn, 2007) at the expense of excluding the conjunction rule from the proof system (Gotsman *et al.*, 2011).

The LOCK axiom says that upon acquiring a lock, the process gets the ownership of its invariant and a permission to release it. In terms of Figure 7, we can think of the corresponding lock partition becoming part of the process-local one, allowing the process to modify it at will. According to UNLOCK, before releasing the lock, the process must have the corresponding permission and must re-establish the lock invariant. When the lock is released, the process gives up the ownership of the permission and the invariant. In terms of Figure 7, the lock partition gets split off the process-local one.

The CALL and RET axioms mirror the operational semantics of `call` and `ret` (see Section 2.1 and Figure 11). CALL requires us to provide enough space on the stack to store the values of registers before a call. The precondition together with the modified stack then has to establish the assertion given by Δ at the target label. The premiss of RET requires us to make a case-split on all possible labels l' we could return to; the precondition has to establish the assertion at every such label after the values of general-purpose registers and `ip` (denoted by $\vec{E}$ and E') have been loaded from the stack.

The axioms CALL and RET provide only a very rudimentary treatment of procedures. In particular, our logic does not have analogues of the usual modular Hoare proof rules for procedures and does not allow applying the EXISTS and FRAME rules over a procedure call. This is because soundly formulating such proof rules in the setting where the stack is visible to procedure code and can potentially be modified by it is non-trivial. Since we are concerned with scheduler verification, in this paper we opted for a simplistic solution and did not include more powerful proof rules for procedures (Feng *et al.*, 2006). As we discuss in Section 10, this issue also represents a promising direction of future work.

The soundness statement of the logic (presented in Section 8.1) constrains the states obtained by running the machine with interrupts disabled: when CPUs are at given program points $l_1, \ldots, l_{\texttt{NCPUS}}$, the state of the machine can be obtained by combining the assertions $\Delta(l_1), \ldots, \Delta(l_{\texttt{NCPUS}})$, describing their local states, and the lock invariants of free locks, as per Figure 7. The set of free locks can be determined based on occurrences of locked predicates in the assertions $\Delta(l_1), \ldots, \Delta(l_{\texttt{NCPUS}})$.

Figure 4 gives an example proof of a concurrent program in the baseline logic, where every CPU executes the code shown in the figure. The assertions shown define the local state of a process and thus the required mapping Δ; the lock invariant of `request_lock` is (4). Note that the assertions describe the stack of a process explicitly. We introduced the $\Vdash$ notation in Section 4.1. When a process acquires `request_lock`, it gets the ownership of the doubly-linked list it protects, together with the corresponding locked predicate. After performing manipulations on the list, the process gives up its ownership upon releasing the lock. Our proof ensures that the code in Figure 4 preserves the doubly-linked list shape of `request_queue` and does not access memory cells other than those specified by the assertions.

5 Logic for preemptive kernels

In this paper we consider schedulers whose interface consists of two routines: `create` and `schedule`. Like in our example scheduler (Section 2.2), `create` makes a new process runnable, and `schedule` performs a context switch. We discuss how our results can be extended when new scheduler routines are introduced in Section 5.5 below. Our logic thus reasons about programs of the form:

$$\mathsf{C} \uplus [l_c : (\texttt{iret}, \{l_c + 1\})] \uplus \mathsf{S} \uplus [l_s : (\texttt{iret}, \{l_s + 1\})] \uplus \mathsf{K}. \qquad \text{(OS)}$$

where C and S are pieces of code implementing the `create` and `schedule` routines of the scheduler, l_c and l_s are their exit points, and K is the rest of the kernel code. Our high-level proof system is designed for proving K, and the low-level system for proving C and S.

We make several assumptions about programs:

- We require that C and S define primitive commands labelled `create` and `schedule`, which are meant to be the entry points for the corresponding scheduler routines. The `create` routine expects the address of the descriptor of the new process to be stored in the register $\texttt{gr}_1$. By our convention `schedule` also marks the entry point of the

interrupt handler. Thus, `schedule` may be called both directly by a process or by an interrupt.

- For now, we ensure that the kernel may not affect the status of interrupts, become aware of the particular CPU it is executing on, or change the stack address. Thus, K may not contain `savecpuid`, `icall` and `iret` (except calls to the scheduler routines `schedule` and `create`), and assignments writing to `ss`. In reality, a kernel might need to disable interrupts, and we generalise our results to handle this in Section 7.
- To ensure that the scheduler routines execute with interrupts disabled, we require that C and S may not contain `icall` and `iret`.
- We require that the kernel K and the scheduler C and S access disjoint sets of locks. This condition simplifies the soundness statement in Section 8 and can be lifted.
- For simplicity, we assume that the scheduler data structures are properly initialised when the program starts executing.

5.1 Interface parameters

As we noted in Section 2.3, our logic can be viewed as implementing a form of rely-guarantee reasoning between the scheduler and the kernel. In particular, interactions between them involve ownership transfer of memory cells at points where the control crosses the boundary between the two components. Hence, the high- and low-level proof systems have to agree on the description of the memory areas being transferred and the properties they have to satisfy. These descriptions form the specification of the interface between the scheduler and the kernel and, correspondingly, between the two proof systems. Here we describe parameters used to formulate it.

When the kernel calls the `create` routine of the scheduler, the latter might need to get the ownership of the process descriptor supplied as the parameter. In the two proof systems, we specify this descriptor using an assertion $\mathsf{desc}(d, \gamma) \in \mathsf{Assert}_{\mathsf{H}}$ with two free logical variables and no register occurrences. Our intention is that it describes the descriptor of a process with the context γ, allocated at the address d. However, the user of our logic is free to choose any assertion, depending on a particular scheduler implementation being verified. This flexibility has an impact on the soundness statement of the logic, as we discuss in Section 8. Since the scheduler and the kernel access disjoint sets of locks, we require that $[\![\mathsf{desc}(d, \gamma)]\!]$ have an empty lockset (Section 4.2). We assume that `create` does not transfer anything back to the kernel at its return, and thus do not introduce an interface parameter for this case.

The `schedule` routine can be called by the kernel explicitly, or as a consequence of an interrupt at any time. Due to the latter case, `schedule` cannot make that many assumptions about the state in which it is called. Therefore, rather than specifying the state to be transferred from the kernel to `schedule` upon an interrupt abstractly, like in the case of `create`, we fix it to be the free part of the stack of the process being preempted—the minimum `schedule` needs to execute. The scheduler returns the ownership of this memory to the process when it schedules the process again. The corresponding interface parameters determine the size of the part of the stack being transferred: the size of the stack $\texttt{StackSize} \in \mathbb{N}$ and the upper bound $\texttt{StackBound} \in \mathbb{N}$ on the stack usage by the kernel (excluding the scheduler). To ensure that the stack does not overflow while calling

$$\frac{\begin{array}{l}\forall l \in \mathsf{labels}(C).\, \forall l' \in \mathsf{next}(C,l).\, (I, \Delta \rhd_{l'} \{\Delta(l)\}\ \mathsf{comm}(C,l)\ \{\Delta(l')\}) \\ \forall l \in \mathsf{Label}(C).\, \exists P \in \mathsf{Assert}_{\mathsf{H}}.\, \Delta(l) \iff \\ \qquad ((0 \leqslant \mathtt{sp} - \mathtt{ss} \leqslant \mathtt{StackBound}) \wedge (\mathtt{sp}..(\mathtt{ss} + \mathtt{StackSize} - 1) \mapsto \_) * P)\end{array}}{I, \Delta \vdash C}\ \textsc{Prog-h}$$

$$\frac{}{I, \Delta \rhd_l \{P\}\ \mathtt{icall}\ (\mathtt{schedule})\ \{P\}}\ \textsc{Sched}$$

$$\frac{\begin{array}{l}\mathsf{free}(P) \cap \mathsf{Reg} = \emptyset \\ P \text{ has an empty lockset} \\ \forall l' \in \mathsf{Label}.\, (\exists \gamma.\, \mathsf{id} = \gamma \wedge \gamma(\mathbf{ip}) = l' \wedge \mathtt{sp}..(\mathtt{ss} + \mathtt{StackSize} - 1) \mapsto \_ * P) \implies \Delta(l')\end{array}}{I, \Delta \rhd_l \{\exists \gamma.\, \gamma(\mathbf{if}) = 1 \wedge \mathsf{desc}(\mathtt{gr}_1, \gamma) * P * Q\}\ \mathtt{icall}(\mathtt{create})\ \{\exists \gamma.\, Q\}}\ \textsc{Create}$$

Fig. 16. The rules specific to the high-level proof system. Here, $\mathsf{id} = [\mathbf{ip} : \_, \mathbf{if} : 1, \mathbf{ss} : \mathtt{ss}, \mathbf{sp} : \mathtt{sp}, \vec{\mathbf{gr}} : \vec{\mathtt{gr}}]$.

an interrupt hander, we require that $\mathtt{StackSize} - \mathtt{StackBound} \geqslant m + 1$, where m is the number of general-purpose registers.

In the following, we first present the high-level proof system used for verifying kernel code, which adapts the baseline concurrency logic from Section 4. We then present the low-level proof system for verifying scheduler code, which extends the high-level system.

5.2 High-level proof system

The high-level proof system is obtained from that of Section 4 with minimal changes. The proof system reasons under an illusion that every process runs on a separate virtual CPU with its own registers (but not memory), and its assertions now describe properties of processes under this assumption, as illustrated in Figure 7. Whereas in Section 4 we justified such reasoning by requiring processes to be pinned to physical CPUs by disabling interrupts, here we do not make this assumption.

The judgements of the proof system are of the same form as before: $I, \Delta \vdash C$, where C specifies all the code executed by kernel processes. As before, different processes can execute different programs by starting at different program points in C. The high-level proof system is meant for verifying the K part of the OS program and, hence, Δ in $I, \Delta \vdash C$ is now meant to give assertions only for the kernel code. When combining proofs in the high-level and low-level proof systems in Section 5.4, we enforce this by restricting Δ so that it is false everywhere except at labels in the kernel code. Similarly, I describes invariants for locks accessible in the kernel code only.

The changes to the logic from Section 4 are as follows. The Prog rule from Figure 15 gets replaced by a similar rule Prog-h, shown in Figure 16, and the rest of the rules in Figure 15 are left without changes. Prog-h inherits the premiss of Prog, and thus subsumes the usual sequential composition rule of Hoare logic: it assumes that the control follows the structure of the process code, even though the scheduler code can get executed due to an interrupt at any time. This possibility is accounted for by the second premiss of Prog-h. Recall from Section 5.1 that the kernel is supposed to transfer the ownership of the free part of the stack to the scheduler at an interrupt, and get it back when it is scheduled again. The second premiss of Prog-h ensures this by requiring all assertions in Δ to

satisfy some restrictions regarding stack usage, formulated using parameters `StackSize` and `StackBound` from Section 5.1:

- the free part of the stack of the process must always be in its local state so that it can be transferred to the interrupt handler at any time;
- this part must always be large enough for the handler to run without a stack overflow; and
- the assertions should be independent of any changes to the empty slots of the stack, which may be modified by the handler.

The latter condition is similar to that of stability in logics based on rely-guarantee (Feng *et al.*, 2007a; Vafeiadis & Parkinson, 2007).

To complete the illusion of uninterrupted control flow in a process, the high-level proof system treats explicit calls to the `create` and `schedule` routines of the scheduler as primitive commands, axiomatising their effect using SCHED and CREATE. These axioms are formulated as if after the corresponding `icall` commands the control just proceeded to the next statement in the program, instead of jumping to the implementation of the routines. This is despite the fact that, after a call to `schedule`, the process may be preempted and the control given to any other process in the system. In this way, the axioms abstract from the scheduler implementation.

The SCHED axiom states that invoking `schedule` has no effect from the point of view of the process—if it is preempted, the scheduler resumes it in the same context, and no other process can touch its local state. The axiom does not place any requirements on the process, as the preconditions necessary for the execution of `schedule`, which can anyway be invoked at any time as the interrupt handler, are established by the second premiss of PROG-H.

The CREATE axiom is more complicated. First, it requires the caller of `create` to provide a new descriptor $\mathsf{desc}(\mathtt{gr}_1, \gamma)$ for the process being created with the context γ. We pass the parameter via the register $\mathtt{gr}_1$ and not via the stack, as this simplifies the following technical presentation. The context is required to have `if` set, since after the context switch is finished, the process starts executing with interrupts enabled. Note that the descriptor is not present in the postcondition: it gets transferred to the scheduler and reappears in the precondition of the implementation of `create` (Section 5.4). The axiom also allows us to transfer the ownership of the part of the heap given by P to the newly created process, thus providing it with an initial local state. This is a typical idiom for high-level reasoning about processes in separation logics (Gotsman *et al.*, 2007). The premiss of the rule correspondingly requires that, after the registers and the stack are properly initialised, the state P we are transferring should establish the assertion at the label the process starts executing from. The effect of loading registers from γ is formulated using the context id.

For the example scheduler in Section 2.2, $\mathsf{desc}(d, \gamma)$ should describe a process descriptor with the stack initialised according to the invariant of a preempted process pictured in Figure 5:

$$\mathsf{desc}(d, \gamma) = d.\mathtt{prev} \mapsto \_ * d.\mathtt{next} \mapsto \_ * \mathsf{desc}_0(d, \gamma),$$

where

$$\begin{array}{rcl}\mathsf{desc}_0(d,\gamma) & = & (\gamma(\mathbf{if}) = 1) \wedge (\gamma(\mathbf{ss}) = d.\mathtt{kernel\_stack}) \wedge \\ & & (0 \leqslant \gamma(\mathbf{sp}) - \gamma(\mathbf{ss}) \leqslant \mathtt{StackBound}) \wedge d.\mathtt{timeslice} \mapsto \_ * \\ & & d.\mathtt{saved\_sp} \mapsto (\gamma(\mathbf{sp}) + m + 1 + \mathtt{SCHED\_FRAME}) * \\ & & \gamma(\mathbf{sp})..(\gamma(\mathbf{sp}) + m) \mapsto \gamma(\mathbf{ip})\gamma(\vec{\mathbf{gr}}) * \\ & & (\gamma(\mathbf{sp}) + m + 1)..(\gamma(\mathbf{ss}) + \mathtt{StackSize} - 1) \mapsto \_ \end{array}$$

and SCHED_FRAME is the size of the activation record of schedule (Figure 2). The descriptor does not include filled stack slots; they can be passed to the process directly in the precondition P.

Now assume that we want to create a process that will start executing from a label l_0 with an empty stack and the ownership of a cell at the address stored in the register $\mathtt{gr}_1$, so that

$$\Delta(l_0) = (\mathtt{ss} = \mathtt{sp} \wedge \mathtt{ss}..(\mathtt{ss} + \mathtt{StackSize} - 1) \mapsto \_ * \mathtt{gr}_1 \mapsto \_).$$

To apply CREATE, we let

$$P = (\gamma(\mathbf{ip}) = l_0 \wedge \gamma(\mathbf{ss}) = \gamma(\mathbf{sp}) \wedge \gamma(\mathbf{gr}_1) \mapsto \_).$$

Then the left-hand side of the implication in the last premiss of CREATE is false for all $l' \neq l_0$, and in this case the implication holds trivially; it is easy to check that the implication also holds for $l' = l_0$.

The proof in Figure 4, previously done using the baseline concurrency logic, is also a valid proof in the high-level logic for $\mathtt{StackBound} = 2 \cdot \mathtt{sizeof}(\mathtt{Request}*)$.

To summarise, the high-level proof system provides modern tools for modular reasoning about concurrent processes using proof rules of concurrent separation logic and hides the control flow of the scheduler by treating its routines as primitive commands. The soundness of such an illusion is established by verifying the scheduler code using a low-level proof system, which we describe next.

5.3 Low-level proof system

The low-level proof system is used for proving that the commands C and S of the OS program implement scheduling correctly. This boils down to checking the two obligations explained in Section 2.3:

1. A scheduler resumes a process with the state of the CPU registers it had the last time it was preempted.
2. A scheduler does not duplicate processes arbitrarily.

To reason about these, we extend the assertion language of Section 4.1 with an additional predicate:

$$P \ ::= \ \ldots \mid \mathsf{Process}(G),$$

where G ranges over context expressions. The predicate records the reference values G of registers in between the time a process is preempted and scheduled again. In Section 5.4 below, we use it to formulate the proof obligation on the context-switch routine of the

$$
\begin{array}{lll}
(r,h,L),M \models_\eta \mathsf{Process}(G) & \text{iff} & h=[],\ L=\emptyset,\ M=\{[\![G]\!]_\eta r\} \\
(r,h,L),M \models_\eta P*Q & \text{iff} & \exists h_1,h_2,L_1,L_2,M_1,M_2.h{=}h_1 \uplus h_2, L=L_1 \uplus L_2, M=M_1 \uplus M_2, \\
 & & (r,h_1,L_1),M_1 \models_\eta P \text{ and } (r,h_2,L_2),M_2 \models_\eta Q \\
(r,h,L),M \models_\eta \mathsf{emp} & \text{iff} & h=[],\ L=\emptyset \text{ and } M=\emptyset \\
(r,h,L),M \models_\eta P \wedge Q & \text{iff} & (r,h,L),M \models_\eta P \text{ and } (r,h,L),M \models_\eta Q
\end{array}
$$

Fig. 17. Semantics of low-level assertions. The $\uplus$ operation on multisets adds up the number of occurrences of each element in its operands.

scheduler formalising (2) from Section 2.3 and thus ensuring property 1 above. We denote the extended set of assertions by $\mathsf{Assert}_\mathsf{L}$.

The addition of the Process predicate changes objects described by assertions: they now denote relations defined by subsets of

$$\mathsf{SchedState} = \mathsf{State} \times \mathscr{M}(\mathsf{Context}),$$

where $\mathscr{M}(\mathsf{Context})$ is the set of all finite multisets of contexts. Here, an element of State describes a state local to a scheduler invocation on a CPU (Figure 6) and that of $\mathscr{M}(\mathsf{Context})$ interprets Process predicates. We give the formal semantics of assertions using the satisfaction relation $\models_\eta$ in Figure 17, parameterised by environments η. The first two cases in the figure are the most interesting ones. The assertion $\mathsf{Process}(G)$ describes a scheduler invocation having the empty heap and lockset and a permission to schedule a single process with the register values G. The separating conjunction $P*Q$ splits all parts of the state-multiset pair except the current scheduler context such that the first part satisfies P and the second Q. This definition of $*$ prohibits duplicating Process and thus ensures property 2 above:

$$\neg(\mathsf{Process}(G) \implies \mathsf{Process}(G) * \mathsf{Process}(G)).$$

The semantic definitions for the remaining assertions are obtained from the corresponding cases in our high-level proof system (Figure 13) either by requiring the multiset component M to be empty, like in the case of emp, or by propagating M to their sub-assertions, like in the case of $P \wedge Q$. For example, the assertion $\exists\gamma.\,\mathsf{desc}(d,\gamma) * \mathsf{Process}(\gamma)$ denotes a descriptor of a preempted process with a Process predicate matching the state stored in it and thus certifying its validity. We denote the set of states satisfying P by $[\![P]\!]_\eta$.

Relations in SchedState can be thought of as connecting the states of the concrete machine and the abstract machine with one CPU per process. As we have noted in Section 2.3, these relations do not describe the full state of the machines. The first component in a relation describes the local state of a scheduler invocation running on a CPU, including its context and the heap and the lockset local to it (e.g., the region marked CPU1 in Figure 6). The multiset in the second part records information about the states of processes described by Process predicates in the assertion (cf. the dark regions in Figure 7), which includes their contexts, but excludes local heaps and locksets. The low-level logic we present in this section is based on separation logic and, hence, the invisibility of these parts of process state to the scheduler automatically guarantees that it cannot access them.

The judgements of the low-level proof system have the form $I,\Delta \vdash_k C$, where $k \in \mathsf{CPUid}$, $I : \mathsf{Lock} \rightharpoonup \mathsf{Assert}_\mathsf{L}$ is a vector of invariants for locks accessible to the scheduler, and

$\Delta : \mathsf{Label} \to \mathsf{Assert_L}$ is a mapping from program positions to low-level assertions. When considering a complete system in Section 5.4, we restrict Δ so that it is false everywhere except at labels in the scheduler code. Since we allow the scheduler to use the `savecpuid` command, the judgement includes the identifier k of the CPU executing the code C.

The intuitive meaning of the judgements is the same as in the high-level system (Section 5.2), with the component describing process states unchanged during the execution of scheduler commands. The judgements thus express how the scheduler code changes the relationship between the state of the scheduler on the CPU k and those of processes running on the machine. The proof rule for deriving our judgements is identical to PROG from Figure 15, modulo the addition of k:

$$\frac{\forall l \in \mathsf{labels}(C). \forall l' \in \mathsf{next}(C,l). I, \Delta \rhd^k_{l'} \{\Delta(l)\}\ \mathsf{comm}(C,l)\ \{\Delta(l')\}}{I, \Delta \vdash_k C}\ \textsc{Prog-l}$$

Note that the syntactic structure of the OS program (see the beginning of Section 5) ensures that the scheduler always executes with interrupts disabled. Thus, in the rule we are able to follow the control flow of C. The low-level system inherits the proof rules for deriving judgements for primitive commands $I, \Delta \rhd_l \{P\}\ c\ \{Q\}$ in Figure 15, adding the superscript k to $\rhd_l$ and ignoring the rules for `icall(schedule)` and `icall(create)`. It also has a rule for `savecpuid`, which makes use of the index k:

$$\frac{}{I, \Delta \rhd^k_l \{e \mapsto \_\}\ \texttt{savecpuid}(e)\ \{e \mapsto k\}}\ \textsc{CPUid}$$

5.4 Putting the two proof systems together

The proof systems presented in Sections 5.2 and 5.3 allow us to reason about the kernel and the scheduler code. We now describe a rule for combining judgements from the two systems, which defines proof obligations for the OS components. This allows us to prove the OS program defined at the beginning of Section 5.

As can be seen from the example of Section 2.2, a scheduler might need to maintain some data structures related to every CPU, which can be accessed by a scheduler invocation running on it. A data structure of this kind in our example scheduler is the element of the `current` array corresponding to the current CPU. Let J_k be an invariant of such data structures for CPU k, which is meant to hold when the scheduler is not running on it. Similar to lock invariants, we do not allow J_k to contain free logical variables or registers, except `ss`. In this case, we can allow `ss` because we have previously required that the kernel cannot modify it. We denote the vector of invariants J_k by J.

Consider assertions $I_\mathsf{K}, \Delta_\mathsf{K}$ and $I_\mathsf{S}, \Delta^k_\mathsf{S}$ for all $k \in \mathsf{CPUid}$, corresponding to the kernel and the scheduler code, respectively:

- $\mathsf{dom}(I_\mathsf{K}) \cap \mathsf{dom}(I_\mathsf{S}) = \emptyset$;
- $\forall l. l \notin \mathsf{dom}(\mathsf{K}) \implies \Delta_\mathsf{K}(l) = \mathsf{false}$;
- $\forall l. l \notin \mathsf{dom}(\mathsf{S}) \uplus \mathsf{dom}(\mathsf{C}) \uplus \{l_s, l_c\} \implies \Delta^k_\mathsf{S}(l) = \mathsf{false}$.

The proof rule for the program OS is as follows:

$$\frac{\begin{array}{l} I_{\mathsf{K}}, \Delta_{\mathsf{K}} \vdash \mathsf{K} \\ \forall k \in \mathsf{CPUid}.\ I_{\mathsf{S}}, \Delta_{\mathsf{S}}^k \vdash_k \mathsf{S},\ \ I_{\mathsf{S}}, \Delta_{\mathsf{S}}^k \vdash_k \mathsf{C} \\ \forall k \in \mathsf{CPUid}.\ \Delta_{\mathsf{S}}^k(\texttt{schedule}) = \Delta_{\mathsf{S}}^k(l_s) = \Delta_{\mathsf{S}}^k(l_c) = \mathsf{SchedState}_k \\ \forall k \in \mathsf{CPUid}.\ \Delta_{\mathsf{S}}^k(\texttt{create}) = (\exists \gamma.\ \gamma(\mathbf{if}) = 1 \wedge \mathsf{SchedState}_k * \mathsf{desc}(\texttt{gr}_1, \gamma) * \mathsf{Process}(\gamma)) \end{array}}{I_{\mathsf{K}}, \Delta_{\mathsf{K}} \mid I_{\mathsf{S}}, \{\Delta_{\mathsf{S}}^k\}_{k \in \mathsf{CPUid}} \mid J \vdash (\mathsf{S}, \mathsf{C}, \mathsf{K})}\ \text{OS}$$

where

$$\begin{array}{rl} \mathsf{SchedState}_k\ =\ & \exists l, \vec{g}.\, 0 \leqslant \texttt{sp} - \texttt{ss} - m - 1 \leqslant \texttt{StackBound} \wedge \\ & (\texttt{sp} - m - 1)..(\texttt{sp} - 1) \mapsto l\vec{g} * \texttt{sp}..(\texttt{ss} + \texttt{StackSize} - 1) \mapsto \_ * \\ & J_k * \mathsf{Process}([\mathbf{ip} : l, \mathbf{if} : 1, \mathbf{ss} : \texttt{ss}, \mathbf{sp} : \texttt{sp} - m - 1, \vec{\mathbf{gr}} : \vec{g}]). \end{array}$$

The first two premisses require us to prove the kernel and the scheduler code in their respective proof systems. The rest define pre- and postconditions for `schedule` and `create` by fixing the assertions at the corresponding labels. This is done using the predicate $\mathsf{SchedState}_k$, which describes the state of a scheduler invocation at CPU k just after it is called using `icall` or before it returns by executing `iret`.

According to the penultimate premiss of the proof rule, when `schedule` is called the stack satisfies the bound on stack usage. The scheduler gets the ownership of the per-CPU data structure J_k, a part of the stack of the process being preempted (which contains the values of registers saved upon the call together with the empty slots), and a Process predicate consistent with the registers saved on the stack. The predicate certifies that, when the scheduler starts executing, the state of the preempted process in the machine corresponds to its state in the abstract machine. The `schedule` routine has to re-establish the same assertion before returning. In the case when it schedules a different process, this will be done using a different Process predicate. However, since the scheduler can only get a Process predicate in the precondition of `schedule` (and when a new process is created; see below), its postcondition guarantees that the process being scheduled has the same register values it had last time it was preempted. Note that the pre- and postconditions of `schedule` mirror the second premiss of the PROG-H rule. Thus, the assumptions it makes about the kernel are justified by the proof of the latter in the high-level system. Also, the per-CPU state of the scheduler J_k is treated similarly to a piece of state protected by a lock: a scheduler invocation gets its ownership when the scheduler starts executing, and gives it up after giving the control back to a process.

The precondition of `create` is similar to that of `schedule`, but additionally assumes a process descriptor for a new process with the address in $\texttt{gr}_1$, and a corresponding Process assertion initialised according to the information in the descriptor. This descriptor is guaranteed to be provided by the kernel by the precondition of the CREATE rule. Adding the new Process assertion can be understood intuitively as creating a fresh virtual CPU for the new process in the abstract machine.

5.5 Extending the logic

Our logic considers scheduling interfaces providing only the fundamental routines for context switch and process creation. However, our simple treatment of scheduler routines allows extending the logic when routines are added to its interface by:

- adding axioms, similar to SCHED and CREATE, for the new routines to the high-level proof system, specifying the pieces of state transferred between the scheduler and the kernel at calls to and returns from the routines; and
- adding new obligations for the routines to the proof rule from Section 5.4, to be discharged using the low-level proof system.

In this case, the pre- and postconditions of the routines in the low-level proof system should mirror those of the axioms in the high-level proof system, similarly to how this is the case for `schedule` and `create`. In Section 7 we demonstrate how a similar approach can be used to deal with features that break through the virtual CPU abstraction implemented by a scheduler, such as access to the interrupt status flag by the kernel.

6 Verifying the example scheduler

We have used the logic to manually construct a proof of the example scheduler of Section 2.2, establishing the judgements about `schedule` and `create` required by the OS proof rule from Section 5.4.[3] By the soundness theorem for our logic (presented in Section 8), this implies that any property of a piece of high-level code proved in concurrent separation logic, including memory safety and functional correctness, holds of the code when it is managed by the example scheduler. In particular, this is true of the properties of the code in Figure 4 described in Section 4. The full proof is given in a Supplementary Appendix available at http://dx.doi.org/10.1017/S0956796813000075 (Gotsman & Yang, 2013). Here, we present only lock and per-CPU scheduler invariants, together with a sketch of the proof of `schedule`.

The invariants of runqueue locks are as follows:

$$I(\texttt{runqueue\_lock}[k]) = \exists x, y, z.\, \texttt{runqueue}[k] \mapsto z * \\ \mathsf{desc}_0(z, \_) * z.\texttt{prev} \mapsto y * z.\texttt{next} \mapsto x * \mathsf{dll}_\Lambda(x, z, z, y),$$

where $\Lambda(d) = \exists \gamma.\, \mathsf{desc}_0(d, \gamma) * \mathsf{Process}(\gamma)$ and desc_0 is defined in Section 5.2. Thus, a runqueue for a CPU k contains a list of descriptors of preempted processes together with Process predicates matching the state stored in them. The per-CPU scheduler invariants are:

$$J_k = \exists d.\, (d.\texttt{kernel\_stack} = \texttt{ss}) \wedge \texttt{current}[k] \mapsto d * \\ d.\texttt{prev} \mapsto \_ * d.\texttt{next} \mapsto \_ * d.\texttt{timeslice} \mapsto \_ * d.\texttt{saved\_sp} \mapsto \_.$$

Thus, the invariant for CPU k includes the descriptor of the process currently running on the CPU. We also know that the current stack is the one identified by its descriptor (recall that the kernel cannot modify the `ss` register).

[3] Since we do not support modular reasoning about procedures, we constructed a proof *schema* for the body of `fork` in Figure 3, which is meant to be instantiated and inlined at every use.

Below we give a sketch of the proof of the context-switch routine `schedule` with the following main idea. When an invocation of `schedule` acquires the runqueue lock and removes a descriptor from the runqueue, it gets the ownership of the corresponding Process predicate, which lets it schedule the process by establishing the postcondition $\mathsf{SchedState}_k$ of `schedule`. When the process is preempted again, `schedule` receives the Process predicate in its precondition $\mathsf{SchedState}_k$. This predicate and the state in J_k let the scheduler insert the descriptor back into the runqueue while maintaining its invariant.

To make sure that the kernel leaves enough space on the stack for the activation records of `schedule` and `load_balance` or `create`, we assume that

$$\texttt{StackSize} - \texttt{StackBound} \geqslant 2 \cdot m + 2 + 4 \cdot \texttt{sizeof(int)} + 2 \cdot \texttt{sizeof(Process*)}.$$

We abbreviate `SCHED_FRAME` to s. Below k is the identifier of the CPU for which the proof is done.

$\{\mathsf{SchedState}_k\}$	
`int cpu;`	
`Process *old_process;`	
$\{cpu, old\_process \Vdash \exists l, \vec{g}, d.\ \texttt{if} = 0 \wedge$	Unpack $\mathsf{SchedState}_k$.
$d.\texttt{kernel\_stack} = \texttt{ss} \wedge$	Local variables `cpu` and `old_process`
$0 \leqslant \texttt{sp} - \texttt{ss} - m - s - 1 \leqslant \texttt{StackBound} \wedge$	are allocated on the stack.
$\texttt{current}[k] \mapsto d * d.\texttt{prev} \mapsto \_ * d.\texttt{next} \mapsto \_ *$	The assertion $d.\texttt{timeslice} \mapsto \_$ allows
$d.\texttt{timeslice} \mapsto \_ * d.\texttt{saved\_sp} \mapsto \_ *$	us to prove the following commands
$(\texttt{sp} - s - m - 1)..(\texttt{sp} - s - 1) \mapsto l\vec{g} *$	manipulating the `timeslice` field of
$\texttt{sp}..(\texttt{ss} + \texttt{StackSize} - 1) \mapsto \_ *$	the current descriptor.
$\mathsf{Process}([\mathbf{ip}: l, \mathbf{if}: 1, \mathbf{ss}: \texttt{ss}, \mathbf{sp}: \texttt{sp} - s - m - 1, \vec{\mathbf{gr}}: \vec{g}])\}$	
`savecpuid(&cpu);`	
`load_balance(cpu);`	
`old_process = current[cpu];`	
... // update the timeslice of old_process	
`if (old_process->timeslice) iret();`	
`old_process->timeslice = SCHED_QUANTUM;`	
$\{cpu, old\_process \Vdash \exists l, \vec{g}.\ \texttt{if} = 0 \wedge$	`cpu` is now equal to k, and
$old\_process.\texttt{kernel\_stack} = \texttt{ss} \wedge$	`old_process` points to the descriptor
$cpu = k \wedge 0 \leqslant \texttt{sp} - \texttt{ss} - m - s - 1 \leqslant \texttt{StackBound} \wedge$	of the current process.
$\texttt{current}[k] \mapsto old\_process *$	
$old\_process.\texttt{prev} \mapsto \_ * old\_process.\texttt{next} \mapsto \_ *$	
$old\_process.\texttt{timeslice} \mapsto \_ *$	
$old\_process.\texttt{saved\_sp} \mapsto \_ *$	
$(\texttt{sp} - s - m - 1)..(\texttt{sp} - s - 1) \mapsto l\vec{g} *$	
$\texttt{sp}..(\texttt{ss} + \texttt{StackSize} - 1) \mapsto \_ *$	
$\mathsf{Process}([\mathbf{ip}: l, \mathbf{if}: 1, \mathbf{ss}: \texttt{ss}, \mathbf{sp}: \texttt{sp} - s - m - 1, \vec{\mathbf{gr}}: \vec{g}])\}$	
`lock(runqueue_lock[cpu]);`	
$\{cpu, old\_process \Vdash \mathsf{locked}(\texttt{runqueue\_lock}[k]) *$	Acquiring the lock gets us ownership
$\exists l, \vec{g}.\ \texttt{if} = 0 \wedge old\_process.\texttt{kernel\_stack} = \texttt{ss} \wedge$	of the locked predicate and the lock
$cpu = k \wedge 0 \leqslant \texttt{sp} - \texttt{ss} - m - s - 1 \leqslant \texttt{StackBound} \wedge$	invariant.
$\texttt{current}[k] \mapsto old\_process *$	This allows us to prove the commands
$old\_process.\texttt{prev} \mapsto \_ * old\_process.\texttt{next} \mapsto \_ *$	below that manipulate the runqueue.
$old\_process.\texttt{timeslice} \mapsto \_ *$	
$old\_process.\texttt{saved\_sp} \mapsto \_ *$	
$(\texttt{sp} - s - m - 1)..(\texttt{sp} - s - 1) \mapsto l\vec{g} *$	
$\texttt{sp}..(\texttt{ss} + \texttt{StackSize} - 1) \mapsto \_ *$	
$\mathsf{Process}([\mathbf{ip}: l, \mathbf{if}: 1, \mathbf{ss}: \texttt{ss}, \mathbf{sp}: \texttt{sp} - s - m - 1, \vec{\mathbf{gr}}: \vec{g}]) *$	

$\exists x,y,z.\,\texttt{runqueue}[k] \mapsto z *$
$\mathsf{desc}_0(z,\_) * z.\texttt{prev} \mapsto y * z.\texttt{next} \mapsto x * \mathsf{dll}_\Lambda(x,z,z,y)\}$

```
insert_node_after(runqueue[cpu]->prev,
    old_process);
current[cpu] = runqueue[cpu]->next;
remove_node(current[cpu]);
old_process->saved_sp = _sp;
```

$\{(cpu, old\_process \Vdash \mathsf{locked}(\texttt{runqueue\_lock}[k]) *$
$\exists l,\vec{g}.\,\texttt{if} = 0 \wedge old\_process.\texttt{kernel\_stack} = \texttt{ss} \wedge$
$cpu = k \wedge 0 \leqslant \texttt{sp} - \texttt{ss} - m - s - 1 \leqslant \texttt{StackBound} \wedge$
$\texttt{current}[k] \mapsto x * old\_process.\texttt{prev} \mapsto y *$
$old\_process.\texttt{next} \mapsto z * old\_process.\texttt{timeslice} \mapsto \_ *$
$old\_process.\texttt{saved\_sp} \mapsto \texttt{sp} *$
$(\texttt{sp} - s - m - 1)..(\texttt{sp} - s - 1) \mapsto l\vec{g} *$
$\texttt{sp}..(\texttt{ss} + \texttt{StackSize} - 1) \mapsto \_ *$
$\mathsf{Process}([\mathbf{ip}: l, \mathbf{if}: 1, \mathbf{ss}: \texttt{ss}, \mathbf{sp}: \texttt{sp} - s - m - 1, \vec{\mathbf{gr}}: \vec{g}]) *$
$\exists x,y,z,w,\gamma.\,\texttt{runqueue}[k] \mapsto z * \mathsf{desc}_0(z,\_) *$
$z.\texttt{prev} \mapsto old\_process * z.\texttt{next} \mapsto w *$
$\mathsf{desc}_0(x,\gamma) * \mathsf{Process}(\gamma) * x.\texttt{prev} \mapsto \_ * x.\texttt{next} \mapsto \_ *$
$\mathsf{dll}_\Lambda(w,z,old\_process,y)) \vee \ldots\}$

`old_process` is now at the end of the runqueue, and process that was at the front is now the current one. We are still using the stack of the old process.

We have omitted the case corresponding to the runqueue being originally empty.

```
_sp = current[cpu]->saved_sp;
savecpuid(&cpu);
_ss = current[cpu]->kernel_stack;
```

$\{(cpu, old\_process \Vdash \mathsf{locked}(\texttt{runqueue\_lock}[k]) *$
$\exists l,\vec{g},d.\,\texttt{if} = 0 \wedge d.\texttt{kernel\_stack} = \texttt{ss} \wedge$
$cpu = k \wedge 0 \leqslant \texttt{sp} - \texttt{ss} - m - s - 1 \leqslant \texttt{StackBound} \wedge$
$\texttt{current}[k] \mapsto d * d.\texttt{prev} \mapsto \_ * d.\texttt{next} \mapsto \_ *$
$d.\texttt{timeslice} \mapsto \_ * d.\texttt{saved\_sp} \mapsto \texttt{sp} *$
$(\texttt{sp} - s - m - 1)..(\texttt{sp} - s - 1) \mapsto l\vec{g} *$
$\texttt{sp}..(\texttt{ss} + \texttt{StackSize} - 1) \mapsto \_ *$
$\mathsf{Process}([\mathbf{ip}: l, \mathbf{if}: 1, \mathbf{ss}: \texttt{ss}, \mathbf{sp}: \texttt{sp} - s - m - 1, \vec{\mathbf{gr}}: \vec{g}]) *$
$\exists x,y,z.\,\texttt{runqueue}[k] \mapsto z *$
$\mathsf{desc}_0(z,\_) * z.\texttt{prev} \mapsto y * z.\texttt{next} \mapsto x * \mathsf{dll}_\Lambda(x,z,z,y)\}$

We are now using the stack of the new process.
The descriptor of the old process has been merged into the dll predicate representing the runqueue.

```
unlock(runqueue_lock[cpu]);
```

$\{cpu, old\_process \Vdash \exists l,\vec{g},d.\,\texttt{if} = 0 \wedge$
$d.\texttt{kernel\_stack} = \texttt{ss} \wedge$
$0 \leqslant \texttt{sp} - \texttt{ss} - m - s - 1 \leqslant \texttt{StackBound} \wedge$
$\texttt{current}[k] \mapsto d * d.\texttt{prev} \mapsto \_ * d.\texttt{next} \mapsto \_ *$
$d.\texttt{timeslice} \mapsto \_ * d.\texttt{saved\_sp} \mapsto \_ *$
$(\texttt{sp} - s - m - 1)..(\texttt{sp} - s - 1) \mapsto l\vec{g} *$
$\texttt{sp}..(\texttt{ss} + \texttt{StackSize} - 1) \mapsto \_ *$
$\mathsf{Process}([\mathbf{ip}: l, \mathbf{if}: 1, \mathbf{ss}: \texttt{ss}, \mathbf{sp}: \texttt{sp} - s - m - 1, \vec{\mathbf{gr}}: \vec{g}])\}$

After releasing the lock, we give up the ownership of the runqueue and the locked predicate.

```
// We deallocate local variables here
```

$\{\mathsf{SchedState}_k\}$

```
iret();
```

Note that the above proof would not go through if we forgot to acquire the runqueue lock before accessing it in `schedule`. This is because, according to the STORE axiom from Figure 15, we need to have ownership of a memory cell in order to access it. Without acquiring the lock, we would not get the ownership of the doubly-linked list representing the runqueue and, hence, would not be able to justify the correctness of runqueue manipulations.

Our logic is not tied to the particular scheduler implementation we consider here. For example, if we represented the runqueue using a red-black tree sorted by process priorities, like in the newer versions of the Linux kernel (Love, 2010), then we would be able to verify the resulting scheduler as above, but with a new runqueue lock invariant.

7 Breaking through the scheduler abstraction: per-CPU data structures

Even though a scheduler is supposed to provide an illusion of running on a dedicated *virtual* CPU to every process, in practice, some features available to the kernel code can break through this abstraction: e.g., a process can disable preemption (which for our machine corresponds to disabling interrupts) and become aware of the *physical* CPU on which it is currently executing. So far we have ignored this possibility by not allowing the kernel to access the `if` register or execute the `savecpuid` command (Section 5). One way in which OS kernels, such as Linux, use preemption disabling is for implementing so-called *per-CPU data structures* (Bovet & Cesati, 2005)—arrays indexed by CPU identifiers such that a process can only access an entry in an array when it runs on the corresponding CPU. This is widely used to implement CPU-local caches of data, which can be accessed without synchronisation with processes running on other CPUs.

The code in Figure 18, whose proof we explain below, illustrates this by the example of a memory allocator, whose routines can be called concurrently by multiple processes. The allocator manages nodes of type `Node`, which it stores in a doubly-linked list `free_list`. Since multiple invocations of the allocator routines can try to access the free list concurrently, such accesses have to be protected by `list_lock`. To avoid this synchronisation in most cases, the deallocation routine shown in Figure 18 first stores nodes in a CPU-local cache; only when this cache overflows does the routine acquire `list_lock` and move the nodes from the cache to the free list. An allocation routine, which we have omitted, could benefit from a similar optimisation, by trying to allocate a node from the CPU-local cache first and accessing the shared free list only when this fails. For the manipulations of a CPU-local cache to be safe, we need to make sure that at most one allocator invocation can access it at a time. We achieve this by disabling interrupts, and hence, preemption, for the duration of the access, using a pair of new commands `cli` (for disabling interrupts) and `sti` (for enabling them). We also use the `savecpuid` command to index into the array of per-CPU caches.

The use of per-CPU data structures makes it more challenging to separate the verification of the kernel from that of the scheduler, as this exposes the notion of a physical CPU that a scheduler is meant to hide. We now show that we can deal with such implementation exposures while preserving the level of abstraction our logic has enabled so far. Instead of exposing the low-level meaning of concepts such as interrupts and physical CPUs in the logic, our approach is to hide them behind an axiomatic interface that allows only reasoning about their intended uses in the kernel, such as per-CPU data structures.

We extend the set of primitive commands from Figure 9 with the above-mentioned commands for disabling and enabling interrupts, meant for the use by the kernel code only:

$$c \;::=\; \ldots \mid \texttt{cli} \mid \texttt{sti}$$

```
1  struct Node {
2    Node *prev, *next;
3    int data;
4  };
5
6  Node *free_list; // a cyclic doubly−linked list with a sentinel node
7  Lock *list_lock; // protects the list
8  Node *free_cache[NCPUS]; // CPU−local caches of free nodes
9                           // (cyclic doubly−linked lists with sentinel nodes)
10 int count[NCPUS]; // number of nodes in each CPU−local cache
11
12 void free(Node *n) {
13   int cpu;
14   {n, cpu ⊩ n.prev ↦ _ ∗ n.next ↦ _ ∗ n.data ↦ _ ∗
15    sp..(ss + StackSize − 1) ↦ _ ∧ sp = ss + 2 · sizeof(Request∗)}
16   cli();
17   {n, cpu ⊩ ∃x, y, z, i, k. &count[k] ↦ i ∗ &free_cache[k] ↦ z ∗ z.prev ↦ y ∗ z.next ↦ x ∗
18    z.data ↦ _ ∗ dll^i_Λ(x, z, z, y) ∗ CPU(k) ∗ n.prev ↦ _ ∗ n.next ↦ _ ∗ n.data ↦ _ ∗
19    sp..(ss + StackSize − 1) ↦ _ ∧ sp = ss + 2 · sizeof(Request∗)}
20   savecpuid(&cpu);
21   {n, cpu ⊩ ∃x, y, z, i. &count[cpu] ↦ i ∗ &free_cache[cpu] ↦ z ∗ z.prev ↦ y ∗ z.next ↦ x ∗
22    z.data ↦ _ ∗ dll^i_Λ(x, z, z, y) ∗ CPU(cpu) ∗ n.prev ↦ _ ∗ n.next ↦ _ ∗ n.data ↦ _ ∗
23    sp..(ss + StackSize − 1) ↦ _ ∧ sp = ss + 2 · sizeof(Request∗)}
24   insert_node_after(free_cache[cpu], n);
25   count[cpu]++;
26   {n, cpu ⊩ ∃x, y, z, i. &count[cpu] ↦ i ∗ &free_cache[cpu] ↦ z ∗ z.prev ↦ y ∗ z.next ↦ x ∗
27    z.data ↦ _ ∗ dll^i_Λ(x, z, z, y) ∗ CPU(cpu) ∗
28    sp..(ss + StackSize − 1) ↦ _ ∧ sp = ss + 2 · sizeof(Request∗)}
29   if (count[cpu] > LIMIT) {
30     lock(list_lock);
31     {n, cpu ⊩ ∃x, y, z, i. &count[cpu] ↦ i ∗ &free_cache[cpu] ↦ z ∗ z.prev ↦ y ∗ z.next ↦ x ∗
32      z.data ↦ _ ∗ dll^i_Λ(x, z, z, y) ∗ CPU(cpu) ∗
33      ∃x′, y′, z′. &free_list ↦ z′ ∗ z′.prev ↦ y′ ∗ z′.next ↦ x′ ∗ z′.data ↦ _ ∗ dll_Λ(x′, z′, z′, y′) ∗
34      sp..(ss + StackSize − 1) ↦ _ ∧ sp = ss + 2 · sizeof(Request∗)}
35     move_contents(free_cache[cpu], free_list);
36     {n, cpu ⊩ ∃z. &count[cpu] ↦ _ ∗ &free_cache[cpu] ↦ z ∗ z.prev ↦ z ∗ z.next ↦ z ∗
37      z.data ↦ _ ∗ CPU(cpu) ∗
38      ∃x′, y′, z′. &free_list ↦ z′ ∗ z′.prev ↦ y′ ∗ z′.next ↦ x′ ∗ z′.data ↦ _ ∗ dll_Λ(x′, z′, z′, y′) ∗
39      sp..(ss + StackSize − 1) ↦ _ ∧ sp = ss + 2 · sizeof(Request∗)}
40     unlock(list_lock);
41     count[cpu] = 0;
42   }
43   {n, cpu ⊩ ∃x, y, z, i. &count[cpu] ↦ i ∗ &free_cache[cpu] ↦ z ∗ z.prev ↦ y ∗ z.next ↦ x ∗
44    z.data ↦ _ ∗ dll^i_Λ(x, z, z, y) ∗ CPU(cpu) ∗
45    sp..(ss + StackSize − 1) ↦ _ ∧ sp = ss + 2 · sizeof(Request∗)}
46   sti();
47   {n, cpu ⊩ sp..(ss + StackSize − 1) ↦ _ ∧ sp = ss + 2 · sizeof(Request∗)}
48 }
```

Fig. 18. A memory deallocation routine using per-CPU caches of free nodes.

$$\text{For } v_1, v_2 \in \mathsf{CPUid} \cup \{\bot\} \text{ let } v_1 \circ v_2 = v \iff (v_1 = v \wedge v_2 = \bot) \vee (v_1 = \bot \wedge v_2 = v)$$

$$\begin{array}{lll}
(r,h,L,v) \models_\eta \mathsf{CPU}(e) & \text{iff} & h = [],\ L = \emptyset,\ [\![e]\!]_r = v \text{ and } v \in \mathsf{CPUid} \\
(r,h,L,v) \models_\eta P_1 * P_2 & \text{iff} & \exists h_1, h_2, L_1, L_2, v_1, v_2.\ h = h_1 \uplus h_2,\ L = L_1 \uplus L_2,\ v = v_1 \circ v_2, \\
 & & (r,h_1,L_1,v_1) \models_\eta P_1 \text{ and } (r,h_2,L_2,v_2) \models_\eta P_2 \\
(r,h,L,v) \models_\eta \mathsf{emp} & \text{iff} & h = [],\ L = \emptyset \text{ and } v = \bot \\
(r,h,L,v) \models_\eta P \wedge Q & \text{iff} & (r,h,L,v) \models_\eta P \text{ and } (r,h,L,v) \models_\eta Q
\end{array}$$

Fig. 19. Semantics of high-level assertions adjusted for handling per-CPU data structures. The semantics of assertions not shown is adjusted similarly.

We furthermore lift the restriction we made in Section 5 that prohibits the kernel code from using the `savecpuid` command. The semantics of our programming language is adjusted by adding the following clauses for the new commands to the transition relation from Figure 11:

$$\begin{array}{l}
(k,(r[\mathtt{if}:1],h,L),l,l') \rightsquigarrow_{\mathtt{cli}} ((r[\mathtt{if}:0],h,L),l'); \quad (k,(r[\mathtt{if}:0],h,L),l,l') \not\rightsquigarrow_{\mathtt{cli}} ; \\
(k,(r[\mathtt{if}:0],h,L),l,l') \rightsquigarrow_{\mathtt{sti}} ((r[\mathtt{if}:1],h,L),l'); \quad (k,(r[\mathtt{if}:1],h,L),l,l') \rightsquigarrow_{\mathtt{sti}} \top.
\end{array}$$

Note that according to this semantics, calling `cli` twice on a CPU freezes it and calling `sti` twice crashes the system.

To handle the new commands in the high-level proof system, we extend its assertion language from Figure 13 with the predicate $\mathsf{CPU}(e)$, which certifies that the process owning it is running on the CPU with the identifier e:

$$P \ ::= \ \ldots \mid \mathsf{CPU}(e)$$

This requires us to adjust the domain over which assertions of the high-level proof system are interpreted, replacing State defined in (3) by

$$\mathsf{State}_{\mathsf{l}} = \mathsf{Context} \times \mathsf{Heap} \times \mathsf{Lockset} \times (\mathsf{CPUid} \cup \{\bot\}).$$

The last component records the CPU that the current process is executing on, or $\bot$ if the assertion does not carry such information. The assertion semantics from Figure 13 is adjusted as shown in Figure 19. Note that, according to this semantics, the assertion $\mathsf{CPU}(k_1) * \mathsf{CPU}(k_2)$ is inconsistent for all $k_1, k_2 \in \mathsf{CPUid}$. This ensures that an assertion can denote at most one $\mathsf{CPU}(e)$ predicate: a process cannot be at two CPUs at the same time. We denote the extended set of assertions by $\mathsf{Assert}_{\mathsf{l}}$.

Judgements of the high-level proof system now have the forms $I,H,\Delta \vdash C$ or $I,H,\Delta \rhd_l \{P\}\ c\ \{Q\}$, where the additional component H is a vector of invariants describing the kernel data structures local to every CPU in the system. We do not allow invariants in H to contain registers or free occurrences of logical variables and require them to have an empty lockset (Section 4.2). We also require that invariants in I and H do not own CPU predicates: $\forall \eta, (r,h,L,v) \in [\![I(\ell)]\!]_\eta . v = \bot$ and the same for H. To preserve the soundness of the CREATE proof rule from Figure 16, we have to impose the same requirement on the $\mathsf{desc}(d,\gamma)$ predicate and the assertion P used in the rule. All these restrictions ensure that a CPU predicate never gets transferred between processes. This is necessary for soundness,

$$\frac{}{I,H,\Delta \rhd_l \{\mathsf{emp}\}\ \texttt{cli}\ \{\exists k.\mathsf{CPU}(k) * H_k\}}\ \textsc{Sti}$$

$$\frac{}{I,H,\Delta \rhd_l \{\exists k.\mathsf{CPU}(k) * H_k\}\ \texttt{sti}\ \{\mathsf{emp}\}}\ \textsc{Cli}$$

$$\frac{}{I,H,\Delta \rhd_l \{e \mapsto \_ * \mathsf{CPU}(\_)\}\ \texttt{savecpuid}(e)\ \{\exists k.\, e \mapsto k * \mathsf{CPU}(k)\}}\ \textsc{CPUid-fixed}$$

$$\frac{}{I,H,\Delta \rhd_l \{e \mapsto \_\}\ \texttt{savecpuid}(e)\ \{e \mapsto \_\}}\ \textsc{CPUid-any}$$

Fig. 20. Proof rules for per-CPU data structures.

since such a predicate makes a statement about the physical CPU that only a particular process is executing on.

The proof rules for the new commands accessible to kernel code are given in Figure 20; these extend the rules in Figures 15 and 16. The rules express a simple reasoning method similar to that used for lock invariants (Section 4): a process executing `cli` gets the ownership of a CPU predicate for some CPU identifier and the corresponding per-CPU data structure (CLI); it gives both up when it executes `sti` (STI). In between calling `cli` and `sti`, the process may modify the CPU-local data structure in any way. The last two axioms are similar to CPUID from Section 5.3. CPUID-FIXED ensures that the `savecpuid` command returns the value consistent with the CPU predicate owned by the process. According to CPUID-ANY, we cannot make any constraints on the value returned by `savecpuid` without such a predicate. We note that our proof rules for interrupts are essentially identical to those in Feng *et al.* (2008a). Our goal here is not to propose a new logic for interrupts, but to demonstrate how natural reasoning methods for such low-level features can be integrated into our logic for preemptive kernels.

Figure 18 gives a proof of the example deallocation routine using our proof rules. We assume the following lock and per-CPU data structure invariants:

$$\begin{array}{ll} I(\texttt{list\_lock}) = & \exists x,y,z.\, \&\texttt{free\_list} \mapsto z * \\ & \quad z.\texttt{prev} \mapsto y * z.\texttt{next} \mapsto x * z.\texttt{data} \mapsto \_ * \mathsf{dll}_\Lambda(x,z,z,y); \\ H_k = & \exists x,y,z,i.\, \&\texttt{count}[k] \mapsto i * \&\texttt{free\_cache}[k] \mapsto z * \\ & \quad z.\texttt{prev} \mapsto y * z.\texttt{next} \mapsto x * z.\texttt{data} \mapsto \_ * \mathsf{dll}^i_\Lambda(x,z,z,y), \end{array}$$

where $\Lambda(x) = x.\texttt{data} \mapsto \_$ and dll^i_Λ is the straightforward generalisation of the dll_Λ predicate from Figure 13 that specifies the number i of nodes in the list.

The soundness statement for our logic, which we discuss next, includes the extension presented in this section.

8 Soundness

A typical approach to proving the soundness of a logic such as ours would be to define an operational semantics of the abstract machine with one CPU per process the scheduler is supposed to implement. Then, the soundness statement of the high-level proof system could restrict the behaviour of this machine, and that of the low-level proof system would establish that any behaviour of the concrete machine with the scheduler is reproducible in

the abstract one. As a corollary, the statements about the behaviour of the kernel proved in the high-level proof system would be carried over to the concrete machine.

Following this approach is difficult in our case for the following reason. In our logic, the pieces of state whose ownership is transferred between the scheduler and the kernel can be described by arbitrary logical assertions, e.g., $\mathsf{desc}(d, \gamma)$ from Section 5.1. In some cases, e.g., when these assertions are imprecise (O'Hearn, 2007), their transfer from the kernel to the scheduler is hard to express operationally when defining a semantics of the abstract machine; see Gotsman *et al*. (2011) for a discussion. The situation would be worse had we based our logic on a more advanced modular concurrency logic, such as deny-guarantee (Dinsdale-Young *et al.*, 2010), which would be needed to handle real OS code. This is because proofs of soundness for such logics do not give an operational semantics to separate components of a program.

For this reason, we do not define an operational semantics of the abstract machine, and neither of the two proof systems of our logic is proved sound with respect to any semantics alone. Instead, our soundness statement interprets a proof of the kernel in the high-level system and a proof of the scheduler in the low-level one together with respect to the semantics of the concrete machine. This makes it convenient for us to prove the soundness of the fragment of our logic inherited from concurrent separation logic in a way (Gotsman *et al.*, 2011) other than its original proof (Brookes, 2007), as the latter relied on giving a semantics to separate processes of the program in isolation. We therefore start by formulating the soundness statement of the baseline concurrency logic from Section 4, which allows us to introduce the basic techniques we use in stating soundness.

8.1 Soundness of the baseline logic

Assume a proof $I, \Delta \vdash C$ in the logic of Section 4 and an environment η giving the values of the logical variables used in this proof. Consider a point in an execution of the machine of Section 3 when the CPUs are at program positions $l_1, \ldots, l_{\mathtt{NCPUS}} \in \mathsf{labels}(C)$. We formulate the soundness of the logic by extracting the set of configurations that the machine can be in at this point from the proof of C. We achieve this by combining the process-local states, defined by Δ, and the states protected by the free locks, defined by I, as per Figure 7. This formalises the intuitive explanations of the reasoning approach of concurrent separation logic given in Section 2.3.

The mapping Δ describes the states that can be owned by processes at the program positions $l_1, \ldots, l_{\mathtt{NCPUS}}$: $[\![\Delta(l_1)]\!]\eta, \ldots, [\![\Delta(l_{\mathtt{NCPUS}})]\!]\eta \in \mathscr{P}(\mathsf{State})$. We now combine these states to get a configuration from Config (Figure 8) that describes the contexts of all CPUs and the part of the heap and the lockset of the machine belonging to the local state of any process. To this end, we lift states to configurations using the operation $\lfloor \cdot \rfloor_k^{\mathsf{BA}}$ in Figure 22 below, which tags their contexts with a CPU identifier $k \in \mathsf{CPUid}$. We then combine the resulting configurations using the operation $\star_{\mathsf{B}}$ in Figure 23 below, which merges the contexts for different CPUs and takes the union of the heaps and locksets. In Figures 22 and 23 we also summarise all other operations for lifting states to configurations and combining the latter that we use in formulating soundness.

Using the above operations, the set of configurations describing the process-local part of the machine state is thus given by the following predicate over Config:

$$\mathsf{IProc}_\eta(l_1,\ldots,l_{\mathtt{NCPUS}}) = \circledast_{k\in\mathsf{CPUid}}^{\mathsf{B}} \lfloor [\![\Delta(l_k)]\!]_\eta \rfloor_k^{\mathsf{BA}},$$

where $\circledast_{\mathsf{B}}$ is the iterated version of $\star_{\mathsf{B}}$. This predicate, however, does not describe the whole heap and lockset of the machine, as we have not taken into account their parts belonging to the invariants of free locks, which are not included into the local state of any process (Figure 7). In any configuration $(R,h,L) \in \mathsf{IProc}_\eta(l_1,\ldots,l_{\mathtt{NCPUS}})$, L gives the set of locks held by any process, so that the set of free locks is $\mathsf{Lock} - L$. To combine their invariants, we use the operation $\lfloor\cdot\rfloor^{\mathsf{BL}}$ in Figure 22, which lifts states, meant to come from lock invariants, to configurations by discarding the context and assuming an empty lockset (recall that lock invariants cannot have free register occurrences and are required to have an empty lockset; see Section 4). The following predicate on configurations describes the part of the machine state belonging to all locks from a set L':

$$\mathsf{ILock}_{L'} = \circledast_{\ell\in L'}^{\mathsf{B}} \lfloor [\![I(\ell)]\!] \rfloor^{\mathsf{BL}}.$$

Here, we omit an environment defining the values of logical variables from $[\![I(\ell)]\!]$, since lock invariants are insensitive to these variables. The set of all configurations the machine can be in when CPUs are at program positions $l_1,\ldots,l_{\mathtt{NCPUS}}$ is obtained by combining the above predicate with $\mathsf{IProc}_\eta(l_1,\ldots,l_{\mathtt{NCPUS}})$:

$$\begin{aligned}\mathsf{IProg}_\eta(l_1,\ldots,l_{\mathtt{NCPUS}}) = \{(R,h_1 \uplus h_2,L) \mid \\ (R,h_1,L) \in \mathsf{IProc}_\eta(l_1,\ldots,l_{\mathtt{NCPUS}}) \wedge ([],h_2,\emptyset) \in \mathsf{ILock}_{\mathsf{Lock}-L}\}.\end{aligned}$$

Theorem 1
Assume $I,\Delta \vdash C$ in the logic of Section 4, $(R,h,L) \to_C (R',h',L')$ and $R(k,\mathtt{if}) = 0$ for $k = 1..\mathtt{NCPUS}$. Then for all environments η,

$$(R,h,L) \in \mathsf{IProg}_\eta(R(1,\mathtt{ip}),\ldots,R(\mathtt{NCPUS},\mathtt{ip})),$$

entails

$$(R',h',L') \in \mathsf{IProg}_\eta(R'(1,\mathtt{ip}),\ldots,R'(\mathtt{NCPUS},\mathtt{ip})).$$

Theorem 1 shows that IProg_η defines an inductive invariant of the system. Since it excludes the error configuration $\top$, the provability of a program in our logic implies its safety.

To summarise, we obtain an overapproximation of the set of machine configurations in two stages: first, we look up the local states at the program positions given in Δ; second, we look up the lock-protected states using the lockset information extracted from the local states. Although this formulation using lookups is easy to understand, it gets unwieldy for more complicated logics, such as our logic for preemptive kernels. We therefore show how to reformulate Theorem 1 in a somewhat less intuitive, but more compact way. First, we replace the map IProc_η from program positions to process-local parts of configurations by a relation. Let

$$\mathsf{IProc}'_\eta = \circledast_{k\in\mathsf{CPUid}}^{\mathsf{B}} \bigcup_{l\in\mathsf{labels}(C)} \lfloor [\![\Delta(l)]\!]_\eta \cap \mathsf{at}_{\mathsf{B}}(l) \rfloor_k^{\mathsf{BA}},$$

where $\mathsf{at}_\mathsf{B}(l) = \{(r,h,L) \in \mathsf{State} \mid r(\mathtt{ip}) = l\}$. The predicate $\mathsf{IProc}'_\eta \subseteq \mathsf{Config}$ describes the part of the machine state belonging to processes for all possible program positions; it can thus be viewed as an invariant of the process-local state. Since assertions in Δ do not restrict the value of the `ip` register (Section 4.1), we have to do this explicitly using at_B. Then the set of configurations that the machine can be in at any time is now given by the following predicate:

$$\mathsf{IProg}'_\eta = \bigcup_{L \uplus L' = \mathsf{Lock}} ((\mathsf{IProc}_\eta \star_\mathsf{B} \mathsf{ILock}_{L'}) \cap \mathsf{held}_\mathsf{B}(L)),$$

where $\mathsf{held}_\mathsf{B}(L) = \{(R,h,L) \in \mathsf{Config}\}$. Here we branch over all sets of locks L that could be held by processes and compute the lock-protected state for its complement L'. We then ensure that L is indeed the set of all held locks by intersecting the result with $\mathsf{held}_\mathsf{B}(L)$. The following theorem is equivalent to Theorem 1.

Theorem 2
If $I, \Delta \vdash C$ in the logic of Section 4, then for all environments η, the set of configurations $\mathsf{IProg}'_\eta \cap \{(R,h,L) \mid \forall k = 1..\mathsf{CPUid}.\, R(k, \mathtt{if}) = 0\}$ is preserved by $\rightarrow_C$.

Theorems 1 and 2 follow from the proof of the soundness statement of our logic for preemptive kernels, which we now formulate using the approach just presented.

8.2 Soundness of the logic for preemptive kernels

Consider a program OS of the form introduced in Section 5 and assume its proof

$$I_\mathsf{K}, H, \Delta_\mathsf{K} \mid I_\mathsf{S}, \{\Delta^k_\mathsf{S}\}_{k \in \mathsf{CPUid}} \mid J \vdash (\mathsf{S}, \mathsf{C}, \mathsf{K})$$

in our logic for preemptive kernels, including the extension to per-CPU data structures from Section 7. We also assume an environment η giving the values of the logical variables used in the proof. To explain the soundness statement informally, let us fix a point in a machine execution and assume for simplicity that every CPU is executing the scheduler code. We construct an inductive system invariant by conjoining the descriptions of pieces of the machine state owned by different OS components, as per Figure 6. We extract these descriptions from the proof as follows:

- If a CPU k is at a program position l in the scheduler code, then $[\![\Delta^k_\mathsf{S}(l)]\!]_\eta \in \mathscr{P}(\mathsf{SchedState})$ describes the state local to the scheduler invocation running on the CPU, including its context, heap and lockset, and the contexts of the processes it has a permission to schedule.
- The combined lockset of all these states tells us which of the locks accessible to the scheduler are free. As in Section 8.1, this allows us to obtain a description of the whole scheduler state by combining the local states with the invariants of all free scheduler locks given by I_S.
- The combined scheduler state contains not only the part of the heap belonging to it, but also the contexts of all the processes that exist in the machine, including their program positions. By looking up the assertions at these positions in Δ_K, we obtain a description of the local states of the processes, including their heaps and locksets.

$$\begin{aligned}
\mathsf{State} &= \mathsf{Context} \times \mathsf{Heap} \times \mathsf{Lockset} \\
\mathsf{SchedState} &= \mathsf{State} \times \mathscr{M}(\mathsf{Context}) \\
\mathsf{Config} &= (\mathsf{CPUid} \to \mathsf{Context}) \times \mathsf{Heap} \times \mathsf{Lockset} \\
\mathsf{SchedConfig} &= \mathsf{Config} \times \mathscr{M}(\mathsf{Context}) \times \mathscr{P}(\mathsf{CPUid}) \\
\mathsf{KernelConfig} &= \mathscr{M}(\mathsf{Context}) \times \mathsf{Heap} \times \mathsf{Lockset} \times \mathscr{P}(\mathsf{CPUid})
\end{aligned}$$

Fig. 21. A summary of the semantic domains used in formulating soundness.

$$\begin{aligned}
\lfloor p_{\mathsf{B}} \rfloor^{\mathsf{BA}}_{k} &= \{([k:r],h,L) \in \mathsf{Config} \mid (r,h,L) \in p_{\mathsf{B}}\} \\
\lfloor p_{\mathsf{B}} \rfloor^{\mathsf{BL}} &= \{([\,],h,\emptyset) \in \mathsf{Config} \mid (r,h,\emptyset) \in p_{\mathsf{B}}\} \\
\lfloor p_{\mathsf{S}} \rfloor^{\mathsf{SA}}_{k,V} &= \{(([k:r],h,L),M,V) \in \mathsf{SchedConfig} \mid ((r,h,L),M) \in p_{\mathsf{S}}\} \\
\lfloor p_{\mathsf{S}} \rfloor^{\mathsf{SL}} &= \{(([\,],h,\emptyset),M,\emptyset) \in \mathsf{SchedConfig} \mid ((r,h,\emptyset),M) \in p_{\mathsf{S}}\} \\
\lfloor p_{\mathsf{K}} \rfloor^{\mathsf{KA}}_{r} &= \{(\{r\},h,L,\{v\} - \{\bot\}) \in \mathsf{KernelConfig} \mid \\
&\qquad (r, h \uplus [r(\mathtt{sp})..(r(\mathtt{ss}) + \mathtt{StackSize} - 1) : _], L, v) \in p_{\mathsf{K}} \wedge (r(\mathtt{if}) = 0 \iff v \neq \bot)\} \\
\lfloor p_{\mathsf{K}} \rfloor^{\mathsf{KL}} &= \{(\emptyset,h,\emptyset,\emptyset) \in \mathsf{KernelConfig} \mid (r,h,\emptyset,\bot) \in p_{\mathsf{K}}\}
\end{aligned}$$

Fig. 22. Operations lifting states to configurations. Here, $p_{\mathsf{B}} \in \mathscr{P}(\mathsf{State})$, $p_{\mathsf{S}} \in \mathscr{P}(\mathsf{SchedState})$, $p_{\mathsf{K}} \in \mathscr{P}(\mathsf{State}_{\mathsf{I}})$, $k \in \mathsf{CPUid}$, $V \in \mathscr{P}(\mathsf{CPUid})$ and $r \in \mathsf{Context}$. We have a pair of operations for every domain of states, one for states coming from assertions in the code (marked by A) and another for states coming from lock or per-CPU invariants (marked by L).

- Again, their combined lockset tells us which of the locks accessible to the kernel are free, allowing us to obtain a description of all lock-protected kernel state from the invariants I_{K}.

We now define this construction formally and for the general case. As we have noted in Section 8.1, we do this by packaging the results of all the lookups mentioned in the above explanation into relations and then performing a relational composition on them.

We start by defining an invariant of the part of the machine state owned by the scheduler. Let us again consider a point in a machine execution when every CPU is executing the scheduler code. To combine the scheduler-local states given by Δ^k_{S} for all CPUs $k \in \mathsf{CPUid}$, we lift them to configurations in the set SchedConfig defined in Figure 21 (in the figure we also summarise all the other domains used in formulating soundness). A configuration $((R,h,L),M,V)$ describes the combined state of multiple scheduler invocations: R defines the contexts on the corresponding CPUs, h and L the combined heap and lockset, and M the contexts of the processes that the invocations have a permission to schedule. To handle per-CPU data structures, we also add a component V describing the set of CPUs on which processes have disabled interrupts using `cli`. When every CPU is executing the scheduler code, this set is empty; we use the general case below. We lift states in SchedState to configurations in SchedConfig using the operation $\lfloor \cdot \rfloor^{\mathsf{SA}}_{k,V}$, defined in Figure 22 for $k \in \mathsf{CPUid}$ and $V \in \mathscr{P}(\mathsf{CPUid})$. We combine the resulting configurations using the operation $\star_{\mathsf{S}}$ in Figure 23. This is similar to $\star_{\mathsf{B}}$, but additionally combines the information about the processes the scheduler invocations know about.

Thus, at those points in the machine execution when all CPUs are executing scheduler invocations, the part of the machine state local to these invocations is described by the

$$
\begin{array}{rl}
\bullet_{\mathsf{B}} & : \mathsf{Config} \times \mathsf{Config} \rightharpoonup \mathsf{Config} \\
\bullet_{\mathsf{S}} & : \mathsf{SchedConfig} \times \mathsf{SchedConfig} \rightharpoonup \mathsf{SchedConfig} \\
\bullet_{\mathsf{K}} & : \mathsf{KernelConfig} \times \mathsf{KernelConfig} \rightharpoonup \mathsf{KernelConfig} \\
\bullet_{\mathsf{SK}} & : \mathsf{SchedConfig} \times \mathsf{KernelConfig} \rightharpoonup \mathsf{Config}
\end{array}
$$

$$
\begin{array}{rcl}
(R_1,h_1,L_1) \bullet_{\mathsf{B}} (R_2,h_2,L_2) & = & (R_1 \uplus R_2, h_1 \uplus h_2, L_1 \uplus L_2) \\
((R_1,h_1,L_1),M_1,V_1) \bullet_{\mathsf{S}} ((R_2,h_2,L_2),M_2,V_2) & = & ((R_1 \uplus R_2, h_1 \uplus h_2, L_1 \uplus L_2), M_1 \uplus M_2, V_1 \uplus V_2) \\
(M_1,h_1,L_1,V_1) \bullet_{\mathsf{K}} (M_2,h_2,L_2,V_2) & = & (M_1 \uplus M_2, h_1 \uplus h_2, L_1 \uplus L_2, V_1 \uplus V_2) \\
((R,h_1,L_1),M_1,V_1) \bullet_{\mathsf{SK}} (M_2,h_2,L_2,V_2) & = & (R, h_1 \uplus h_2, L_1 \uplus L_2), \text{ if } M_1 = M_2 \text{ and } V_1 = V_2 \\
((R,h_1,L_1),M_1,V_1) \bullet_{\mathsf{SK}} (M_2,h_2,L_2,V_2) & & \text{undefined, otherwise}
\end{array}
$$

Let $\star_{\mathsf{B}}$, $\star_{\mathsf{S}}$, $\star_{\mathsf{K}}$, $\star_{\mathsf{SK}}$ be the pointwise liftings of $\bullet_{\mathsf{B}}$, $\bullet_{\mathsf{S}}$, $\bullet_{\mathsf{K}}$, $\bullet_{\mathsf{SK}}$ to sets of configurations. For example, for $p_1, p_2 \in \mathscr{P}(\mathsf{Config})$, we define $\star_{\mathsf{B}} : \mathscr{P}(\mathsf{Config}) \times \mathscr{P}(\mathsf{Config}) \to \mathscr{P}(\mathsf{Config})$ as follows: $p_1 \star_{\mathsf{B}} p_2 = \{(R_1,h_1,L_1) \bullet_{\mathsf{B}} (R_2,h_2,L_2) \mid (R_1,h_1,L_1) \in p_1 \wedge (R_2,h_2,L_2) \in p_2\}$.

Fig. 23. Operations for combining configurations. Recall that the $\uplus$ operation on multisets adds up the number of occurrences of each element in its operands.

following predicate over SchedConfig:

$$
\bigcircledast_{k \in \mathsf{CPUid}}^{\star_{\mathsf{S}}} \bigcup_{l \in (\mathsf{labels}(\mathsf{S} \uplus \mathsf{C}) \uplus \{l_s, l_c\})} \lfloor [\![\Delta_{\mathsf{S}}^k(l)]\!]_\eta \cap \mathsf{at}_{\mathsf{S}}(l) \cap \mathsf{if}_{\mathsf{S}}(0) \rfloor_{k,\emptyset}^{\mathsf{SA}},
$$

where $\mathsf{at}_{\mathsf{S}}(l) = \{((r,h,L),M) \in \mathsf{SchedState} \mid r(\mathtt{ip}) = l\}$ and $\mathsf{if}_{\mathsf{S}}(v) = \{((r,h,L),M) \in \mathsf{SchedState} \mid r(\mathtt{if}) = v\}$. Like in the definition of IProc'_η in Section 8.1, we branch over all program positions in the scheduler code and combine the local states at these positions given by Δ_{S}^k; we also restrict the value of the `ip` and `if` registers explicitly.

We now need to consider the case when a process is running on some CPU k. Let l be its program position. In this case, the scheduler still owns some state associated with the CPU, e.g., the per-CPU scheduler invariant J_k. We describe this state by the following predicate over SchedState:

$$
\begin{array}{l}
\mathsf{SchedSleep}_k(l) = J_k * \mathtt{sp}..(\mathtt{ss} + \mathtt{StackSize} - 1) \mapsto \_ * \\
\qquad\qquad\qquad\qquad \mathsf{Process}([\mathbf{ip} : l, \mathbf{if} : \mathtt{if}, \mathbf{ss} : \mathtt{ss}, \mathbf{sp} : \mathtt{sp}, \vec{\mathbf{gr}} : \vec{\mathtt{gr}}]).
\end{array}
$$

Note that, when a scheduler invocation starts executing on a CPU, the invariant J_k is added to its local state, which is why previously we did not have to take it into account when defining the state local to active scheduler invocations. Although assertions in the high-level proof system mention the empty slots of the process stack, the slots in fact belong to the scheduler when the process is preempted. For the sake of uniformity, we choose to count them in the scheduler state even when the process is running and, hence, add them to $\mathsf{SchedSleep}_k(l)$. The Process predicate in $\mathsf{SchedSleep}_k(l)$ describes the currently running process; it corresponds to the Process predicate that the scheduler lost when it transferred the control to the process (see the postcondition of `schedule` in the OS proof rule from Section 5.4). We took the liberty of using `if` in $\mathsf{SchedSleep}_k(l)$, even though this is prohibited in our logic, since this assertion is used only for formulating soundness.

The following predicate over SchedConfig describes the scheduler state excluding that protected by free locks:

$$\mathsf{ISched}_\eta = \bigcup_{\substack{V_\mathsf{I}\subseteq V_\mathsf{K},\\ V_\mathsf{S}\uplus V_\mathsf{K}=\mathsf{CPUid}}} \left(\left(\circledast_{k\in V_\mathsf{S}} \bigcup_{l\in(\mathsf{labels}(\mathsf{S}\uplus\mathsf{C})\uplus\{l_s,l_c\})} \lfloor[\![\Delta^k_\mathsf{S}(l)]\!]_\eta \cap \mathsf{at}_\mathsf{S}(l)\cap \mathsf{if}_\mathsf{S}(0)\rfloor^{\mathsf{SA}}_{k,\emptyset}\right) \star_\mathsf{S} \left(\circledast_{k\in V_\mathsf{I}} \bigcup_{l\in\mathsf{labels}(\mathsf{K})} \lfloor[\![\mathsf{SchedSleep}_k(l)]\!]_\eta \cap \mathsf{at}_\mathsf{S}(l)\cap \mathsf{if}_\mathsf{S}(0)\rfloor^{\mathsf{SA}}_{k,\{k\}}\right) \star_\mathsf{S} \left(\circledast_{k\in V_\mathsf{K}-V_\mathsf{I}} \bigcup_{l\in\mathsf{labels}(\mathsf{K})} \lfloor[\![\mathsf{SchedSleep}_k(l)]\!]_\eta \cap \mathsf{at}_\mathsf{S}(l)\cap \mathsf{if}_\mathsf{S}(1)\rfloor^{\mathsf{SA}}_{k,\emptyset}\right)\right).$$

Here, we branch over all splittings of CPUs into those executing the scheduler and the kernel code, given by V_S and V_K. We also branch over all sets $V_\mathsf{I} \subseteq V_\mathsf{K}$ of CPUs where processes have disabled interrupts. For every CPU k, we then branch over all possible program positions l. Depending on whether l is in the scheduler or the kernel code, we use either the assertion in the scheduler proof or the invariant $\mathsf{SchedSleep}_k$. In the latter case, V_I determines the last component of the resulting configurations.

To obtain the whole state owned by the scheduler, we take into account the invariants of free locks accessible to it, similarly to how it was done in the definition of IProg'_η in Section 8.1. To this end, we use the operation $\lfloor\cdot\rfloor^{\mathsf{SL}}$ in Figure 22 that converts states in SchedState, meant to come from lock invariants, to configurations in SchedConfig. Then the part of the machine state belonging to the scheduler locks from a set L' is defined by the following predicate:

$$\mathsf{ISchedLock}_{L'} = \circledast_{\ell\in L'} \lfloor[\![I_\mathsf{S}(\ell)]\!]\rfloor^{\mathsf{SL}}.$$

Hence, the invariant of the whole scheduler state is

$$\bigcup_{L\uplus L'=\mathsf{dom}(I_\mathsf{S})} ((\mathsf{ISched}_\eta \star_\mathsf{S} \mathsf{ISchedLock}_{L'}) \cap \mathsf{held}_\mathsf{S}(L)), \tag{5}$$

where $\mathsf{held}_\mathsf{S}(L) = \{((R,h,L),M,V) \in \mathsf{SchedConfig}\}$. In a configuration $((R,h,L), M,V)$ from the set (5), the components M and V give the information about all processes that exist in the system, whether running or preempted, with the former taken into account due to the inclusion of the corresponding Process predicate into $\mathsf{SchedSleep}_k$. In the following, we use this fact to connect the invariant of the scheduler with that of the kernel. We now proceed to define the latter.

Consider a process with a context r, which could come from a configuration in the scheduler invariant (5). Then its local state is given by $[\![\Delta_\mathsf{K}(r(\mathtt{ip}))]\!]_\eta \in \mathscr{P}(\mathsf{State}_\mathsf{I})$. We now combine such states for different processes in a form appropriate for composing with the scheduler invariant (5). We start by lifting them to configurations in the set KernelConfig defined in Figure 21. A configuration $(M,h,L,V) \in \mathsf{KernelConfig}$ describes the combined state of multiple processes, with the contexts M, the combined heap h and lockset L and the set V of CPUs on which they have disabled interrupts. We perform the lifting using the operation $\lfloor\cdot\rfloor^{\mathsf{KA}}_r$ in Figure 22 that selects the states with the context $r \in \mathsf{Context}$ and removes the empty slots of the process stack, accounted for in the scheduler state (in $\mathsf{SchedSleep}_k$, if

the process is running, and in Δ_{S}^{k} and I_{S}, if it is preempted). We then combine the resulting configurations using the operation $\star_{\mathsf{K}}$ in Figure 23. The invariant of the process-local part of the machine state is thus given by the following predicate over KernelConfig:

$$\mathsf{IKernel}_{\eta} = \bigcup_{M \in \mathscr{M}(\mathsf{Context})} \circledast_{r \in M}^{\mathsf{K}} \lfloor [\![\Delta_{\mathsf{K}}(r(\mathtt{ip}))]\!]_{\eta} \rfloor_{r}^{\mathsf{KA}}.$$

Here we branch over all possible finite multisets M of contexts of processes that may run on the machine. For every context r in M, the local state of the corresponding process is then determined by the assertion in the proof of the kernel at the program point $r(\mathtt{ip})$, restricted to the states with the context r. Note that the comprehension $r \in M$ over a multiset M considers every instance of an element in the multiset separately.

As before, to obtain the invariant of the whole kernel state, we need to take into account the invariants of free locks accessible to the kernel. Additionally, we need to include kernel per-CPU invariants H_k for all CPUs k where processes have not disabled interrupts (for those where they have, the per-CPU data structures have been merged into their local states). We use the operation $\lfloor \cdot \rfloor^{\mathsf{KL}}$ in Figure 22 to convert states in $\mathsf{State}_{\mathsf{l}}$, meant to come from kernel lock invariants or kernel per-CPU data structure invariants, to configurations in KernelConfig. Since the invariants cannot contain CPU predicates (Section 7), the operation assumes that the last component of the states it is given is always $\bot$. Then the part of the machine state belonging to kernel locks from a set L' or per-CPU invariants for the set of CPUs V are respectively given by

$$\mathsf{IKernelLock}_{L'} = \circledast_{\ell \in L'}^{\mathsf{K}} \lfloor [\![I_{\mathsf{K}}(\ell)]\!] \rfloor^{\mathsf{KL}}; \qquad \mathsf{IPercpu}_{V} = \circledast_{k \in V}^{\mathsf{K}} \lfloor [\![H_k]\!] \rfloor^{\mathsf{KL}}.$$

Hence, the invariant of the whole kernel state is

$$\bigcup_{\substack{L \uplus L' = \mathsf{dom}(I_{\mathsf{K}}) \\ V \uplus V' = \mathsf{CPUid}}} ((\mathsf{IKernel}_{\eta} \star_{\mathsf{K}} \mathsf{IKernelLock}_{L'} \star_{\mathsf{K}} \mathsf{IPercpu}_{V'}) \cap \mathsf{held}_{\mathsf{K}}(L) \cap \mathsf{disabled}(V)),$$

where

$$\mathsf{held}_{\mathsf{K}}(L) = \{(M,h,L,V) \in \mathsf{KernelConfig}\};$$
$$\mathsf{disabled}(V) = \{(M,h,L,V) \in \mathsf{KernelConfig}\}.$$

Finally, we compose the above invariant of the kernel with that of the scheduler (5) to obtain an invariant of the whole system. To this end, we use the operation $\star_{\mathsf{SK}}$ in Figure 23, which combines heaps and locksets, provided that the contexts of processes and sets of CPUs on which they disabled interrupts match in both arguments. Then the system invariant is given by the following predicate over Config:

$$\mathsf{IOs}_{\eta} = \left(\bigcup_{L \uplus L' = \mathsf{dom}(I_{\mathsf{S}})} ((\mathsf{ISched}_{\eta} \star_{\mathsf{S}} \mathsf{ISchedLock}_{L'}) \cap \mathsf{held}_{\mathsf{S}}(L)) \right) \star_{\mathsf{SK}}$$

$$\left(\bigcup_{\substack{L \uplus L' = \mathsf{dom}(I_{\mathsf{K}}) \\ V \uplus V' = \mathsf{CPUid}}} ((\mathsf{IKernel}_{\eta} \star_{\mathsf{K}} \mathsf{IKernelLock}_{L'} \star_{\mathsf{K}} \mathsf{IPercpu}_{V'}) \cap \mathsf{held}_{\mathsf{K}}(L) \cap \mathsf{disabled}(V)) \right).$$

The following theorem, proved in Appendix A, states the soundness of our logic for preemptive kernels.

Theorem 3
If $I_{\mathsf{K}}, H, \Delta_{\mathsf{K}} \mid I_{\mathsf{S}}, \{\Delta_{\mathsf{S}}^k\}_{k \in \mathsf{CPUid}} \mid J \vdash (\mathsf{S}, \mathsf{C}, \mathsf{K})$, then for all environments η, the set of configurations IOs_η is preserved by $\rightarrow_{\mathsf{OS}}$.

Consequences. Theorem 3 allows carrying over statements proved in the high-level proof system about the abstract machine with one virtual CPU per process to the concrete machine. For example, it implies that the properties of the code in Figure 4 described in Section 4 hold when it is managed by the example scheduler. To demonstrate this formally, assume that the initial machine configuration satisfies IOs_η. Then the soundness statement ensures that the machine cannot reach an error label l_e on any CPU, provided that the assertion at this program point in all high-level proofs is false. Indeed, in this case the invariant IOs_η does not contain any states where one of the CPUs is at l_e. Note that the functional correctness of an OS kernel is usually formulated as a refinement between the kernel and its specification. As an OS kernel does not usually make any assumptions about user processes, complications with formulating this refinement that necessitate the unusual soundness statement of our logic do not arise, and thus proving that it can be reduced to proving an invariance property relating the kernel and its specification (Gargano *et al.*, 2005; Klein *et al.*, 2009). Thus, Theorem 3 can also be used to justify such proofs.

Ownership transfer. It is instructive to analyse how the ownership transfer between the scheduler and the kernel is handled by our soundness statement. For example, consider a transfer of a new process descriptor $\mathsf{desc}(d, \gamma)$ from the kernel to the scheduler at a call to `create`. Since the CREATE axiom requires the descriptor in its precondition, before the kernel calls `create`, the state partitioning defined by IOs_η counts the descriptor as part of $\mathsf{IKernel}_\eta$. Since, by the OS proof rule, the implementation of `create` receives the descriptor in its precondition, in the configuration immediately after the call to `create`, IOs_η defines it to be part of ISched_η. Thus, ownership transfer repartitions program state among the parts defined above.

Proof idea and other concurrency logics. The proof of Theorem 3 is relatively straightforward, if technical. When the transition in $\rightarrow_{\mathsf{OS}}$ considered in the theorem corresponds to a command associated with an ownership transfer in our logic, we prove that the target configuration belongs to IOs_η by redistributing the state among components used to construct this invariant, following the above explanation of ownership transfers. When the transition in $\rightarrow_{\mathsf{OS}}$ corresponds to a command that is not associated with an ownership transfer between components, such as an assignment, we first 'unpack' the IOs_η invariant to get the local state of the process or the scheduler invocation executing the command. We then replace this local state with the new one specified by the proof and show that we can 'pack' the invariant back to obtain the target state of the machine.

We based our logic for preemptable code on concurrent separation logic, which would not be able to handle complicated concurrency mechanisms employed in modern OS kernels (Bovet & Cesati, 2005). The approach we take in stating and proving the soundness of our logic has been applied extensively to various concurrent derivatives of separation

logic (Gotsman, 2009; Gotsman *et al.*, 2011). This leads us to believe that we can integrate more advanced logics from this class (Feng *et al.*, 2007a; Vafeiadis & Parkinson, 2007; Dinsdale-Young *et al.*, 2010; Dinsdale-Young *et al.*, 2013) without problems.

9 Related work

There have been a number of OS verification projects; see Klein (2009) for a survey. To our knowledge, none of these has included the verification of a scheduler in a preemptive kernel with the realistic features we consider. A representative example is the seL4 project (Klein *et al.*, 2009), which verified a variant of the L4 microkernel as a whole, together with the scheduler. There, proofs about kernel components other than the scheduler had to ensure the preservation of its invariants, e.g., the well-formedness of its runqueue. The proof was still tractable because the kernel was running on a uniprocessor and used an event-based execution model, so that preemption was disabled most of the time. However, such architecture is not used by mainstream operating systems. If fact, as noted by Klein *et al*. (2009), the absence of verification technology dealing with preemption was one of the reasons for the choice of this architecture in seL4.

The closest work to ours is the one by Feng *et al*. (2007b; 2008a; 2008b), who proposed a logic for verifying OS kernels, also based on separation logic. Like us, they structure the logic into separate proof systems for the scheduler and preemptable code. We thus share their vision (Feng *et al.*, 2008b; Shao, 2010) of verifying different components of systems software using specialised logics that allow reasoning on an appropriate level of abstraction. However, there are differences between our work and theirs in the OS features handled and in the general approach to formulating the logic and proving it sound.

As far as OS features are concerned, Feng *et al*. consider a uniprocessor and verify an idealised scheduler without dynamic process creation or ownership transfer between the scheduler and processes. As a consequence, they do not have an analogue of our affine Process predicate needed to handle multiprocessing. On the other hand, Feng *et al*. support modular reasoning about procedures, which we do not. As for the general approach, Feng *et al*. formulate the logics for the scheduler and preemptive code and justify their soundness by embedding them into OCAP (Feng *et al.*, 2007b), a logic supporting first-class code pointers. This support is then used to handle transfers of control between the scheduler and the kernel and to reason modularly about procedures. In contrast, we establish the soundness of our proof systems by a direct correspondence to an operational semantics, without going through an intermediate logic.

The verification of realistic kernels requires supporting modular reasoning about procedures as well as multiprocessing and ownership transfer. Thus, both the logic of Feng *et al*. and ours would need to be extended before they are up to the task. In the case of Feng *et al*.'s logic, this would require extending OCAP to accommodate multiprocessing. This is not completely trivial, since on a multiprocessor, the scheduler and the kernel can run at the same time on different CPUs, and OCAP currently requires the control to be within a single component at any point of time. One would also need to extend OCAP to treat assertions about code pointers affinely, as our Process predicates. Conversely, to provide support for modular reasoning about procedures in our logic, we would have to borrow the corresponding proof rules from one of the available logics

for storable code, possibly a relative of OCAP (Feng *et al.*, 2006; Feng *et al.*, 2007b; Ni & Shao, 2006; Schwinghammer *et al.*, 2009; Charlton, 2011). The soundness could still be proved by a correspondence to an operational semantics, but our current proof would have to be adjusted. Thus, both our approach and the one of Feng *et al*. can potentially be extended to cover a wider range of OS features, possibly by exploiting techniques from the other. It remains to be seen in which settings a given approach works better.

Even though in this paper we focus on a low-level programming language, the reasoning principles we propose are high level and analogous to those developed for control flow in functional programs. For example, the Process predicate in our low-level proof system can be viewed as an assertion about an affine continuation, providing a clean model for capturing and resuming process state. Our use of separating conjunction over such predicates is analogous to the use of linear typing in the study of continuations in functional programs (Berdine *et al.*, 2002; Hasegawa, 2002; Thielecke, 2003; Hasegawa, 2004; Laird, 2005). One can thus think of our work as layering clean functional reasoning on top of low-level OS code.

Maeda and Yonezawa have proved a simple context-switch routine using an extension of alias types (Maeda & Yonezawa, 2009). Their proof expresses the disjointness of data structures belonging to the scheduler and the rest of the kernel using the tensor operator of alias types, which corresponds to our separating conjunction. However, their type system does not hide the internal data structures of the scheduler while proving the rest of the kernel, and is thus non-modular.

Yang and Hawblitzel have recently developed a kernel where most of the codebase is typechecked and therefore cannot directly access data structures belonging to the core part of the kernel, including the scheduler (Yang & Hawblitzel, 2010). However, the guarantees established by the type system do not take into account the contents of data structures, so the kernel can still subvert the scheduler by leaving them in an inconsistent state. The OS resorts to runtime checks in such cases, introducing a performance penalty. The relationship to this work is that of a trade-off: type safety guarantees are easier to get, but are not as strong as those provided by a program logic.

Refinement is a well-known approach in the verification of both operating systems and general concurrent programs (Back, 1981; Gargano *et al.*, 2005; Jones, 2007; Klein *et al.*, 2009; Turon & Wand, 2011). Our logic can be viewed as implementing a form of refinement where the semantics of the abstract system is defined axiomatically by the high-level proof system and refinement relations, defined by the low-level proof system, focus only on the relevant state of the systems related. We thus advance the refinement theory to systems with complex ownership transfers.

10 Conclusion

In this paper, we have neither verified a complete operating system nor built an automatic tool. Instead, we have proposed a proof rule that allows decomposing the verification of a preemptive OS kernel into two simpler tasks—verifying the scheduler and preemptable code separately. Furthermore, we have, for the first time, achieved this for the patterns of interaction between the scheduler and the kernel present in mainstream operating systems.

Such a result is relevant no matter what type of formal analysis of OS code one is performing: manual or automatic verification, or even bug-finding. Moreover, as we argued in Section 2.2, the straightforward approach of verifying the scheduler together with the rest of the kernel makes reasoning intractable; thus, a result such as ours is in fact indispensable for verifying realistic OS kernels.

Despite our development being carried out for a particular baseline concurrency logic and a class of scheduling interfaces, the key technical methods we proposed in this paper are transferable and can be reused in OS verification projects. These include:

- exploiting a logic validating the frame property to hide the state of the scheduler while verifying the kernel and vice versa;
- using the Process assertions to reason about the correct treatment of process states by the scheduler and the affine semantics of $*$ on them to reason about scheduling on multiprocessors;
- dealing with features breaking through the scheduler abstraction, such as interrupt disabling, by axiomatising their intended uses when reasoning about the kernel; and
- formulating soundness by constructing a global property from local assertions on different levels of abstraction using a combination of the separating conjunction and relational composition.

Acknowledgments

We thank Anindya Banerjee, Xinyu Feng, Boris Köpf, Mark Marron, Peter O'Hearn, Matthew Parkinson, Noam Rinetzky, Zhong Shao, Viktor Vafeiadis and Jules Villard for comments and discussions that helped improve the paper. Gotsman was supported by the EU FET ADVENT project. Yang was supported by EPSRC.

Supplementary materials

For supplementary material for this article, please visit dx.doi.org/10.1017/S0956796813000075.

A Appendix. Proof of soundness

Auxiliary definitions. In the following, we write $\{E(\_)\}$, where E is an expression with occurrences of $\_$, to mean the set of values arising from evaluating E with $\_$ substituted for any values from the corresponding domains.

For a set Σ let $\mathscr{P}(\Sigma)^\top$ be the domain of subsets of Σ with a special element $\top$. The order $\sqsubseteq$ in the domain $\mathscr{P}(\Sigma)^\top$ is subset inclusion with $\top$ being the greatest element. We define two partial operations interpreting the $*$ connectives in the high- and low-level proof systems, respectively:

$$*_{\mathsf{K}} \ : \ \mathscr{P}(\mathsf{State})^\top \times \mathscr{P}(\mathsf{State})^\top \to \mathscr{P}(\mathsf{State})^\top;$$
$$*_{\mathsf{S}} \ : \ \mathscr{P}(\mathsf{SchedState})^\top \times \mathscr{P}(\mathsf{SchedState})^\top \to \mathscr{P}(\mathsf{SchedState})^\top.$$

For $p,q \in \mathscr{P}(\mathsf{State})$ we let

$$p *_{\mathsf{K}} q = \{(r, h_1 \uplus h_2, L_1 \uplus L_2) \mid (r,h_1,L_1) \in p \wedge (r,h_2,L_2) \in q\}; \quad \top * p = p * \top = \top.$$

For $p,q \in \mathscr{P}(\mathsf{SchedState})$ we let

$$p *_{\mathsf{S}} q = \{((r, h_1 \uplus h_2, L_1 \uplus L_2), M_1 \uplus M_2) \mid ((r,h_1,L_1),M_1) \in p \wedge ((r,h_2,L_2),M_2) \in q\};$$
$$\top * p = p * \top = \top.$$

We use the following definitions: for $p \subseteq \mathsf{State}_\mathsf{l}$, $q \subseteq \mathsf{SchedState}$, $l \in \mathsf{Label}$, $k \in \mathsf{CPUid}$, $\ell \in \mathsf{Lock}$ let

$$\begin{aligned}
\mathsf{at}_{\mathsf{K}}(l) &= \{(r,h,L,v) \in \mathsf{State}_\mathsf{l} \mid r(\mathtt{ip}) = l\};\\
\mathsf{lk}_{\mathsf{K}}(\ell) &= \{(r,[],\{\ell\},\bot) \in \mathsf{State}_\mathsf{l}\};\\
\mathsf{lk}_{\mathsf{S}}(\ell) &= \{((r,[],\{\ell\}),\emptyset) \in \mathsf{SchedState}\};\\
\mathsf{int}(k) &= \{(r,[],\emptyset,k) \in \mathsf{State}_\mathsf{l}\};\\
\mathsf{to}_{\mathsf{K}}(l,p) &= \{(r[\mathtt{ip}:l],h,L,v) \in \mathsf{State}_\mathsf{l} \mid (r,h,L,v) \in p\};\\
\mathsf{to}_{\mathsf{S}}(l,q) &= \{((r[\mathtt{ip}:l],h,L),M) \in \mathsf{SchedState} \mid ((r,h,L),M) \in q\}.
\end{aligned}$$

Finally, consider a process descriptor predicate $\mathsf{desc}(d,\gamma)$ with free logical variables d and γ and an environment η. We define $\boldsymbol{desc}_\eta : \mathsf{Val} \times \mathsf{Context} \to \mathsf{State}$ as follows: for $u \in \mathsf{Val}$ and $r \in \mathsf{Context}$ we let $\boldsymbol{desc}_\eta(u,r) = [\![\mathsf{desc}(d,\gamma)]\!]_{\eta[d:u,\gamma:r]}$.

Transformers for primitive commands. It is convenient for us to reformulate the semantics of primitive commands c in Figure 11 and Section 7 in terms of transformers

$$f_c^k : \mathsf{Label} \times \mathsf{Label} \times \mathsf{State} \to \mathscr{P}(\mathsf{State})^\top, \quad k \in \mathsf{CPUid}$$

for $c \in \mathsf{PComm}$, defined as follows: $f_c^k(l,l',(r,h,L)) = \top$, if $k,(r,h,L),l,l' \rightsquigarrow_c \top$; otherwise,

$$f_c^k(l,l',(r,h,L)) = \bigcup \big\{(r'[\mathtt{ip}:l''],h',L') \mid (k,(r,h,L),l,l') \rightsquigarrow_c ((r',h',L'),l'')\big\}.$$

We extend the transformers to operate on states in $\mathsf{SchedState}$ and $\mathsf{State}_\mathsf{l}$:

$$\begin{aligned}
f_c^k &: \mathsf{Label} \times \mathsf{Label} \times \mathsf{SchedState} \to \mathscr{P}(\mathsf{SchedState})^\top, && k \in \mathsf{CPUid};\\
f_c^k &: \mathsf{Label} \times \mathsf{Label} \times \mathsf{State}_\mathsf{l} \to \mathscr{P}(\mathsf{State}_\mathsf{l})^\top, && k \in \mathsf{CPUid}.
\end{aligned}$$

We let

$$f_c^k(l,l',((r,h,L),M)) = \{((r',h',L'),M) \mid (r',h',L') \in f_c^k(l,l',(r,h,L))\}, \tag{A 1}$$

if $f_c^k(l,l',(r,h,L)) \neq \top$, and $f_c^k(l,l',((r,h,L),M)) = \top$, otherwise. We let

$$f_c^k(l,l',(r,h,L,v)) = \{(r',h',L',v) \mid (r',h',L') \in f_c^k(l,l',(r,h,L))\}, \tag{A 2}$$

if $f_c^k(l,l',(r,h,L)) \neq \top$, and $f_c^k(l,l',(r,h,L,v)) = \top$, otherwise. We then lift these transformers to the corresponding domains pointwise. For example, for $p \in \mathscr{P}(\mathsf{State}_\mathsf{l})^\top$ we let

$$f_c^k(l,l',p) = \begin{cases} \bigsqcup\{f_c^k(l,l',(r,h,L,v)) \mid (r,h,L,v) \in p\}, & \text{if } p \neq \top;\\ \top, & \text{if } p = \top. \end{cases}$$

The transformers thus defined satisfy the property of *locality* (Calcagno *et al.*, 2007) with respect to the operations $*_{\mathsf{S}}$ and $*_{\mathsf{K}}$:

$$\forall p,q \in \mathscr{P}(\mathsf{SchedState}).\, f_c^k(p *_{\mathsf{S}} q) \sqsubseteq f_c^k(p) *_{\mathsf{S}} q; \tag{A 3}$$

$$\forall p,q \in \mathscr{P}(\mathsf{State_I}).\, f_c^k(p *_{\mathsf{K}} q) \sqsubseteq f_c^k(p) *_{\mathsf{K}} q. \tag{A 4}$$

Semantic proofs. To prove Theorem 3, we translate a syntactic proof in our logic into a semantic form, which annotates every program point in the OS code with a description of the state local to the process or the scheduler invocation executing the code. Namely, given an environment η, a *semantic proof* (Gotsman *et al.*, 2011) of the OS program is defined as a tuple $(G_{\mathsf{S}}, G_{\mathsf{K}}, \mathscr{I}_{\mathsf{S}}, \mathscr{I}_{\mathsf{K}}, \mathscr{H}, \mathscr{J})$, where

- $G_{\mathsf{S}}^k : \mathsf{Label} \to \mathscr{P}(\mathsf{SchedState})$, $k \in \mathsf{CPUid}$;
- $G_{\mathsf{K}} : \mathsf{Label} \to \mathscr{P}(\mathsf{State_I})$;
- $\mathscr{I}_{\mathsf{S}} \in \mathsf{Lock} \rightharpoonup \mathscr{P}(\mathsf{SchedState})$;
- $\mathscr{I}_{\mathsf{K}} \in \mathsf{Lock} \rightharpoonup \mathscr{P}(\mathsf{State_I})$;
- $\mathscr{H}_k \in \mathscr{P}(\mathsf{State_I})$, $k \in \mathsf{CPUid}$,
- $\mathscr{J}_k \in \mathscr{P}(\mathsf{SchedState})$, $k \in \mathsf{CPUid}$,

such that $\mathscr{I}_{\mathsf{K}}$, $\mathscr{I}_{\mathsf{S}}$, $\mathscr{H}$, $\mathscr{J}$ satisfy the analogues of the well-formedness restrictions previously imposed on I_{K}, I_{S} , H, J, and the conditions in Figure A 1 hold. The latter conditions are semantic counterparts of the axioms in the high- and low-level proof systems. The following lemma shows that a syntactic proof can be converted into a semantic one.

Lemma 1
Given a proof $I_{\mathsf{K}}, H, \Delta_{\mathsf{K}} \mid I_{\mathsf{S}}, \{\Delta_{\mathsf{S}}^k\}_{k \in \mathsf{CPUid}} \mid J \vdash (\mathsf{S}, \mathsf{C}, \mathsf{K})$ and an environment η, there exists a semantic proof $(G_{\mathsf{S}}, G_{\mathsf{K}}, [\![I_{\mathsf{S}}]\!]_\eta, [\![I_{\mathsf{K}}]\!]_\eta, [\![H]\!]_\eta, [\![J]\!]_\eta)$ such that for all $l \in \mathsf{Label}$ and $k \in \mathsf{CPUid}$ we have $G_{\mathsf{K}}(l) = [\![\Delta_{\mathsf{K}}(l)]\!]_\eta \cap \mathsf{at}_{\mathsf{K}}(l)$ and $G_{\mathsf{S}}^k(l) = [\![\Delta_{\mathsf{S}}^k(l)]\!]_\eta \cap \mathsf{at}_{\mathsf{S}}(l)$.

We omit the straightforward proof of the lemma and proceed to prove the main soundness theorem.

Proof of Theorem 3. Let us fix an environment η. We first apply Lemma 1 to construct a semantic proof $(G_{\mathsf{S}}, G_{\mathsf{K}}, \mathscr{I}_{\mathsf{S}}, \mathscr{I}_{\mathsf{K}}, \mathscr{H}, \mathscr{J})$ from the given syntactic one. Assume now that $\sigma \in \mathsf{IOs}_\eta$ and $\sigma \to_{\mathsf{OS}} \sigma'$ for some $\sigma' \in \mathsf{Config} \cup \{\top\}$. We need to show that $\sigma' \in \mathsf{IOs}_\eta$. Let the command in $\sigma \to_{\mathsf{OS}} \sigma'$ be executed by CPU k. We can thus assume

$$\sigma = (R[k:r], h, L),\ \ R(k) \text{ is undefined},\ \ r(\mathtt{ip}) = l,\ \ c = \mathsf{comm}(\mathsf{OS}, l),\ \ l' \in \mathsf{next}(\mathsf{OS}, l).$$

By the definition of IOs_η, there exist

$$h_1, h_2 \in \mathsf{Heap},\ \ L_1 \subseteq \mathsf{dom}(I_{\mathsf{S}}),\ \ L_2 \subseteq \mathsf{dom}(I_{\mathsf{K}}),\ \ M \in \mathscr{M}(\mathsf{Context}),\ \ V \in \mathscr{P}(\mathsf{CPUid})$$

such that

$$((R[k:r], h_1, L_1), M, V) \in \mathsf{ISched}_\eta \star_{\mathsf{S}} \mathsf{ISchedLock}_{\mathsf{dom}(I_{\mathsf{S}}) - L_1}; \tag{A 24}$$

$$(M, h_2, L_2, V) \in \mathsf{IKernel}_\eta \star_{\mathsf{K}} \mathsf{IKernelLock}_{\mathsf{dom}(I_{\mathsf{K}}) - L_2} \star_{\mathsf{K}} \mathsf{IPercpu}_{\mathsf{CPUid} - V} \tag{A 25}$$

$$h = h_1 \uplus h_2, \quad L = L_1 \uplus L_2. \tag{A 26}$$

We now consider several cases of how σ' may be obtained.

$$\forall l \in \mathsf{Label}.\, l \notin \mathsf{dom}(\mathsf{K}) \implies G_{\mathsf{K}}(l) = \emptyset; \tag{A 5}$$

$$\forall l \in \mathsf{Label}, k \in \mathsf{CPUid}.\, l \notin \mathsf{dom}(\mathsf{S}) \uplus \mathsf{dom}(\mathsf{C}) \uplus \{l_s, l_c\} \implies G^k_{\mathsf{S}}(l) = \emptyset; \tag{A 6}$$

$$\forall k \in \mathsf{CPUid}.\, G^k_{\mathsf{S}}(\mathtt{schedule}) = [\![\mathsf{SchedState}_k]\!]_\eta \cap \mathsf{at}_{\mathsf{S}}(\mathtt{schedule}); \tag{A 7}$$

$$\forall k \in \mathsf{CPUid}.\, G^k_{\mathsf{S}}(l_s) = [\![\mathsf{SchedState}_k]\!]_\eta \cap \mathsf{at}_{\mathsf{S}}(l_s); \tag{A 8}$$

$$\forall k \in \mathsf{CPUid}.\, G^k_{\mathsf{S}}(\mathtt{create}) = [\![\exists \gamma.\, \gamma(\mathbf{if}) = 1 \wedge \mathsf{SchedState}_k * \mathsf{desc}(\mathtt{gr}_1, \gamma) * \mathsf{Process}(\gamma)]\!]_\eta \cap \mathsf{at}_{\mathsf{S}}(\mathtt{create}); \tag{A 9}$$

$$\forall k \in \mathsf{CPUid}.\, G^k_{\mathsf{S}}(l_c) = [\![\mathsf{SchedState}_k]\!]_\eta \cap \mathsf{at}_{\mathsf{S}}(l_c); \tag{A 10}$$

$$\begin{array}{c}\forall l \in \mathsf{Label}, (r,h,L,v) \in G_{\mathsf{K}}(l).\, 0 \leqslant r(\mathtt{sp}) - r(\mathtt{ss}) \leqslant \mathtt{StackBound} \wedge \\ \mathsf{dom}(h) \supseteq \{r(\mathtt{sp}), \ldots, r(\mathtt{ss}) + \mathtt{StackSize} - 1\} \wedge \\ \forall h'.\, (\forall u \notin \{r(\mathtt{sp}), \ldots, r(\mathtt{ss}) + \mathtt{StackSize} - 1\}.\, h(u) = h'(u)) \implies (r,h',L,v) \in G_{\mathsf{K}}(l),\end{array} \tag{A 11}$$

and for all $l \in \mathsf{labels}(\mathsf{OS})$, $l' \in \mathsf{next}(\mathsf{OS}, l)$, $c = \mathsf{comm}(\mathsf{OS}, l)$ and $k \in \mathsf{CPUid}$, we have:

- if c is not `lock` or `unlock`, and $l \in \mathsf{labels}(\mathsf{S}) \uplus \mathsf{labels}(\mathsf{C})$, then

$$f^k_c(l, l', G^k_{\mathsf{S}}(l)) \neq \top \wedge \forall ((r,h,L),M) \in f^k_c(l, l', G^k_{\mathsf{S}}(l)).\, ((r,h,L),M) \in G^k_{\mathsf{S}}(r(\mathtt{ip})); \tag{A 12}$$

- if c is not `lock`, `unlock`, `icall`, `cli` or `sti` and $l \in \mathsf{labels}(\mathsf{K})$, then

$$f^k_c(l, l', G_{\mathsf{K}}(l)) \neq \top \wedge \forall (r,h,L,v) \in f^k_c(l, l', G_{\mathsf{K}}(l)).\, (r,h,L,v) \in G_{\mathsf{K}}(r(\mathtt{ip})); \tag{A 13}$$

- if c is $\mathtt{lock}(\ell)$ and $l \in \mathsf{labels}(\mathsf{S}) \uplus \mathsf{labels}(\mathsf{C})$, then

$$\mathsf{to}_{\mathsf{S}}(l', (G^k_{\mathsf{S}}(l) *_{\mathsf{S}} \mathscr{I}_{\mathsf{S}}(\ell) *_{\mathsf{S}} \mathsf{lk}_{\mathsf{S}}(\ell))) \subseteq G^k_{\mathsf{S}}(l'); \tag{A 14}$$

- if c is $\mathtt{lock}(\ell)$ and $l \in \mathsf{labels}(\mathsf{K})$, then

$$\mathsf{to}_{\mathsf{K}}(l', (G_{\mathsf{K}}(l) *_{\mathsf{K}} \mathscr{I}_{\mathsf{K}}(\ell) *_{\mathsf{K}} \mathsf{lk}_{\mathsf{K}}(\ell))) \subseteq G_{\mathsf{K}}(l'); \tag{A 15}$$

- if c is $\mathtt{unlock}(\ell)$ and $l \in \mathsf{labels}(\mathsf{S}) \uplus \mathsf{labels}(\mathsf{C})$, then

$$\mathsf{to}_{\mathsf{S}}(l', G^k_{\mathsf{S}}(l)) \subseteq G^k_{\mathsf{S}}(l') *_{\mathsf{S}} \mathscr{I}_{\mathsf{S}}(\ell) *_{\mathsf{S}} \mathsf{lk}_{\mathsf{S}}(\ell); \tag{A 16}$$

- if c is $\mathtt{unlock}(\ell)$ and $l \in \mathsf{labels}(\mathsf{K})$, then

$$\mathsf{to}_{\mathsf{K}}(l', G_{\mathsf{K}}(l)) \subseteq G_{\mathsf{K}}(l') *_{\mathsf{K}} \mathscr{I}_{\mathsf{K}}(\ell) *_{\mathsf{K}} \mathsf{lk}_{\mathsf{K}}(\ell); \tag{A 17}$$

- if c is `cli` and $l \in \mathsf{labels}(\mathsf{K})$, then

$$\mathsf{to}_{\mathsf{K}}(l', (G_{\mathsf{K}}(l) *_{\mathsf{K}} \mathscr{H}_k *_{\mathsf{K}} \mathsf{int}(k))) \subseteq G_{\mathsf{K}}(l'); \tag{A 18}$$

- if c is `sti` and $l \in \mathsf{labels}(\mathsf{K})$, then

$$\mathsf{to}_{\mathsf{K}}(l', G_{\mathsf{K}}(l)) \subseteq G_{\mathsf{K}}(l') *_{\mathsf{K}} \mathscr{H}_k *_{\mathsf{K}} \mathsf{int}(k); \tag{A 19}$$

- if c is $\mathtt{icall}(\mathtt{schedule})$, then

$$\mathsf{to}_{\mathsf{K}}(l', G_{\mathsf{K}}(l)) \subseteq G_{\mathsf{K}}(l'); \tag{A 20}$$

- if c is $\mathtt{icall}(\mathtt{create})$, then for some $P, Q \in \mathsf{Assert}_{\mathsf{I}}$ such that $\mathsf{free}(P) \cap \mathsf{Reg} = \emptyset$, P does not own CPU predicates and has an empty lockset, we have

$$G_{\mathsf{K}}(l) \subseteq [\![\exists \gamma.\, \gamma(\mathbf{if}) = 1 \wedge \mathsf{desc}(\mathtt{gr}_1, \gamma) * P * Q]\!]_\eta \cap \mathsf{at}_{\mathsf{K}}(l); \tag{A 21}$$

$$G_{\mathsf{K}}(l') \supseteq [\![\exists \gamma.\, Q]\!]_\eta \cap \mathsf{at}_{\mathsf{K}}(l'); \tag{A 22}$$

$$\forall r \in \mathsf{Context}.\, \{(r, [r(\mathtt{sp})..(r(\mathtt{ss}) + \mathtt{StackSize} - 1) : \_], \emptyset, \bot)\} *_{\mathsf{K}} [\![P]\!]_{\eta[\gamma:r]} \subseteq G_{\mathsf{K}}(r(\mathtt{ip})). \tag{A 23}$$

Fig. A 1. Conditions for a semantic proof $(G_{\mathsf{S}}, G_{\mathsf{K}}, \mathscr{I}_{\mathsf{S}}, \mathscr{I}_{\mathsf{K}}, \mathscr{H}, \mathscr{J})$.

Case 1. σ' is obtained by applying the fourth rule in Figure 12. This case is impossible, since by (A 24) and the definition of ISched_η we have $l = r(\mathtt{ip}) \in \mathsf{labels}(\mathsf{OS})$.

Case 2. σ' is obtained by applying the first or the third rule in Figure 12, with the command executed by the scheduler and different from `lock`, `unlock` or `iret`. In this case, $l \in \mathsf{labels}(\mathsf{S}) \uplus \mathsf{labels}(\mathsf{C})$ and

$$\begin{array}{l}(f_c^k(l,l',(r,h,L)) = \top \implies \sigma' = \top) \wedge \\ (f_c^k(l,l',(r,h,L)) \neq \top \implies \sigma' \in (\{(R,[],\emptyset)\} \star_{\mathsf{B}} \lfloor f_c^k(l,l',(r,h,L))\rfloor_k^{\mathsf{BA}})).\end{array} \tag{A 27}$$

From (A 24), for some $h_\mathsf{S} \in \mathsf{Heap}$, $L_\mathsf{S} \in \mathsf{Lockset}$, $M_\mathsf{S} \in \mathscr{M}(\mathsf{Context})$, $V_\mathsf{S}, V_\mathsf{K} \in \mathscr{P}(\mathsf{CPUid})$ we have $V_\mathsf{S} \uplus V_\mathsf{K} \uplus \{k\} = \mathsf{CPUid}$, $V \subseteq V_\mathsf{K}$, $r(\mathtt{if}) = 0$,

$$((r,h_1,L_1),M) \in G_\mathsf{S}^k(l) *_\mathsf{S} \{((\_,h_\mathsf{S},L_\mathsf{S}),M_\mathsf{S})\} \tag{A 28}$$

and for $W = V$ and $W_1 = L_1$ we have

$$\begin{array}{rl}((R,h_\mathsf{S},L_\mathsf{S}),M_\mathsf{S},W) \in & \left(\circledast_{j\in V_\mathsf{S}} \bigcup_{l\in(\mathsf{labels}(\mathsf{S}\uplus\mathsf{C})\uplus\{l_s,l_c\})} \lfloor[\![\Delta_\mathsf{S}^j(l)]\!]_\eta \cap \mathsf{at}_\mathsf{S}(l) \cap \mathsf{if}_\mathsf{S}(0)\rfloor_{j,\emptyset}^{\mathsf{SA}}\right) \star_\mathsf{S} \\ & \left(\circledast_{j\in V_\mathsf{I}} \bigcup_{l\in\mathsf{labels}(\mathsf{K})} \lfloor[\![\mathsf{SchedSleep}_j(l)]\!]_\eta \cap \mathsf{at}_\mathsf{S}(l) \cap \mathsf{if}_\mathsf{S}(0)\rfloor_{j,\{j\}}^{\mathsf{SA}}\right) \star_\mathsf{S} \\ & \left(\circledast_{j\in V_\mathsf{K}-V_\mathsf{I}} \bigcup_{l\in\mathsf{labels}(\mathsf{K})} \lfloor[\![\mathsf{SchedSleep}_j(l)]\!]_\eta \cap \mathsf{at}_\mathsf{S}(l) \cap \mathsf{if}_\mathsf{S}(1)\rfloor_{j,\emptyset}^{\mathsf{SA}}\right) \star_\mathsf{S} \\ & \mathsf{ISchedLock}_{\mathsf{dom}(I_\mathsf{S})-W_1}.\end{array} \tag{A 29}$$

We have:

$$\begin{array}{rll} & f_c^k(l,l',((r,h,L),M)) & \\ = & f_c^k(l,l',\{((r,h_1,L_1),M)\} *_\mathsf{S} \{((\_,h_2,L_2),\emptyset)\}) & \textit{by (A 26)} \\ \sqsubseteq & f_c^k(l,l',G_\mathsf{S}^k(l) *_\mathsf{S} \{((\_,h_2\uplus h_\mathsf{S},L_2\uplus L_\mathsf{S}),M_\mathsf{S})\}) & \textit{by (A 28)} \\ \sqsubseteq & f_c^k(l,l',G_\mathsf{S}^k(l)) *_\mathsf{S} \{((\_,h_2\uplus h_\mathsf{S},L_2\uplus L_\mathsf{S}),M_\mathsf{S})\} & \textit{by (A 3)}\end{array}$$

From this and (A 12), $f_c^k(l,l',G_\mathsf{S}^k(l)) \neq \top$, hence, by (A 1) and (A 27), $\sigma' \neq \top$. Let $\sigma' = (R[k:r'],h',L)$. Then, $r'(\mathtt{if}) = r(\mathtt{if}) = 0$ and from the above, (A 27) and (A 12), we have

$$\begin{array}{l}((R[k:r'],h',L),M,V) \in \{((R,[],\emptyset),\emptyset,\emptyset)\} \star_\mathsf{S} \\ \qquad\qquad \lfloor (G_\mathsf{S}^k(r'(\mathtt{ip})) \cap \mathsf{if}_\mathsf{S}(0)) *_\mathsf{S} \{((\_,h_2\uplus h_\mathsf{S},L_2\uplus L_\mathsf{S}),M_\mathsf{S})\}\rfloor_{k,V}^{\mathsf{SA}}.\end{array}$$

From this and (A 6) we get $r'(\mathtt{ip}) \in \mathsf{dom}(\mathsf{S}) \uplus \mathsf{dom}(\mathsf{C}) \uplus \{l_s,l_c\}$. Hence, by (A 29) we have

$$((R[k:r'],h',L),M,V) \in \mathsf{ISched}_\eta \star_\mathsf{S} \mathsf{ISchedLock}_{\mathsf{dom}(I_\mathsf{S})-L_1} \star_\mathsf{S} \{(([],h_2,L_2),\emptyset,\emptyset)\}.$$

Then, from (A 25) and the definition of $\star_{\mathsf{SK}}$ we get $\sigma' \in \mathsf{IOs}_\eta$.

Case 3. σ' is obtained by applying the first rule in Figure 12, with the scheduler executing `lock`. In this case

$$l \in \mathsf{labels}(\mathsf{S}) \uplus \mathsf{labels}(\mathsf{C}), \quad c = \mathtt{lock}(\ell), \quad \ell \notin L, \quad \sigma' = (R[k:r[\mathtt{ip}:l']],h,L\cup\{\ell\}).$$

From (A 24), for some h_S, L_S, M_S, V_S, V_K we have $V_S \uplus V_K \uplus \{k\} = \mathsf{CPUid}$, $V \subseteq V_K$, $r(\mathtt{if}) = 0$,

$$((r, h_1, L_1), M) \in G^k_S(l) *_S \mathscr{I}_S(\ell) *_S \{((\_, h_S, L_S), M_S)\}$$

and (A 29) holds for $W = V$ and $W_1 = L_1 \cup \{\ell\}$. Then,

$$((r[\mathtt{ip} : l'], h_1, L_1 \cup \{\ell\}), M) \in \mathsf{to}_S(l', (G^k_S(l) *_S \mathscr{I}_S(\ell) *_S \mathsf{lk}_S(\ell))) *_S \{((\_, h_S, L_S), M_S)\}.$$

By (A 14), this implies

$$((r[\mathtt{ip} : l'], h_1, L_1 \cup \{\ell\}), M) \in G^k_S(l') *_S \{((\_, h_S, L_S), M_S)\}.$$

From this and (A 6) we get $l' \in \mathsf{dom}(\mathsf{S}) \uplus \mathsf{dom}(\mathsf{C}) \uplus \{l_s, l_c\}$. Hence, by (A 29) for $W = V$ and $W_1 = L_1 \cup \{\ell\}$,

$$((R[k : r[\mathtt{ip} : l']], h_1, L_1 \cup \{\ell\}), M, V) \in \mathsf{ISched}_\eta \star_S \mathsf{ISchedLock}_{\mathsf{dom}(I_S) - (L_1 \cup \{\ell\})}.$$

Then, from (A 25) and the definition of $\star_{SK}$ we get $\sigma' \in \mathsf{IOs}_\eta$.

Case 4. σ' is obtained by applying the first or the third rule in Figure 12, with the scheduler executing `unlock`. In this case, $l \in \mathsf{labels}(\mathsf{S}) \uplus \mathsf{labels}(\mathsf{C})$, $c = \mathtt{unlock}(\ell)$ and (A 27) holds.

From (A 24), there exist h_S, L_S, M_S, V_S, V_K such that $V_S \uplus V_K \uplus \{k\} = \mathsf{CPUid}$, $V \subseteq V_K$, $r(\mathtt{if}) = 0$ and (A 28) and (A 29) hold for $W = V$ and $W_1 = L_1$. From (A 28) we then get

$$((r[\mathtt{ip} : l'], h_1, L_1), M) \in \mathsf{to}_S(l', G^k_S(l)) *_S \{((\_, h_S, L_S), M_S)\}.$$

Then by (A 16)

$$((r[\mathtt{ip} : l'], h_1, L_1), M) \in G^k_S(l') *_S \mathscr{I}_S(\ell) *_S \mathsf{lk}_S(\ell) *_S \{((\_, h_S, L_S), M_S)\}.$$

Hence, $\ell \in L_1$, which by (A 27) implies $\sigma' \neq \top$. Then, $\sigma' = (R[k : r[\mathtt{ip} : l']], h, L - \{\ell\})$. The above also implies

$$((r[\mathtt{ip} : l'], h_1, L_1 - \{\ell\}), M) \in G^k_S(l') *_S \mathscr{I}_S(\ell) *_S \{((\_, h_S, L_S), M_S)\}.$$

From this and (A 6) we get $l' \in \mathsf{dom}(\mathsf{S}) \uplus \mathsf{dom}(\mathsf{C}) \uplus \{l_s, l_c\}$. Hence, by (A 29)

$$(R[k : r[\mathtt{ip} : l']], h_1, L_1 - \{\ell\}) \in \mathsf{ISched}_\eta \star_S \mathsf{ISchedLock}_{\mathsf{dom}(I_S) - (L_1 - \{\ell\})}.$$

Then, from (A 25) and the definition of $\star_{SK}$ we get $\sigma' \in \mathsf{IOs}_\eta$.

Case 5. σ' is obtained by applying the first or the third rule in Figure 12, with the command executed by the kernel and different from `lock`, `unlock`, `icall`, `sti` or `cli`. In this case, $l \in \mathsf{labels}(\mathsf{K})$ and (A 27) holds.

From (A 24), there exist h_S, L_S, M_S, V_S, V_K such that $V_S \uplus V_K \uplus \{k\} = \mathsf{CPUid}$, $V \subseteq V_K \cup \{k\}$, $r(\mathtt{if}) = 0 \iff k \in V$, (A 29) holds for $W = V - \{k\}$ and $W_1 = L_1$ and

$$((r, h_1, L_1), M) \in ([\![\mathsf{SchedSleep}_k(l)]\!]_\eta \cap \mathsf{at}_S(l)) *_S \{((\_, h_S, L_S), M_S)\}. \tag{A 30}$$

The latter implies

$$((r, h_1, L_1), M) \in \mathscr{J}_k *_S \{((\_, [r(\mathtt{sp})..(r(\mathtt{ss}) + \mathtt{StackSize} - 1 : \_)] \uplus h_S, L_S), \{r\} \uplus M_S)\}.$$

Then, for some $h_1' \in \mathsf{Heap}$ and

$$h_0 \in \{[r(\mathtt{sp})..(r(\mathtt{ss}) + \mathtt{StackSize} - 1) : \_]\}$$

we have $h_1 = h_1' \uplus h_0$ and

$$((r, h_1', L_1), M) \in \mathscr{J}_k *_{\mathsf{S}} \{((\_, h_{\mathsf{S}}, L_{\mathsf{S}}), \{r\} \uplus M_{\mathsf{S}})\}. \tag{A 31}$$

Let $h_2' = h_2 \uplus h_0$, then

$$h = h_1' \uplus h_2', \quad L = L_1 \uplus L_2. \tag{A 32}$$

Also, let $v = k$, if $k \in V$, and $v = \bot$, otherwise.

Note that from (A 30) it follows that $r \in M$. Then by (A 25) and (A 11) for some $h_{\mathsf{K}} \in \mathsf{Heap}$ and $L_{\mathsf{K}} \in \mathsf{Lockset}$ we have

$$(r, h_2', L_2, v) \in G_{\mathsf{K}}(l) *_{\mathsf{K}} \{(\_, h_{\mathsf{K}}, L_{\mathsf{K}}, \emptyset)\} \tag{A 33}$$

and[4]

$$(M - \{r\}, h_{\mathsf{K}}, L_{\mathsf{K}}, V - \{k\}) \in \circledast_{\mathsf{K}}{}_{r'' \in M - \{r\}} \lfloor [\![\Delta_{\mathsf{K}}(r''(\mathtt{ip}))]\!]_\eta \rfloor^{\mathsf{KA}}_{r''} \star_{\mathsf{K}} \mathsf{IKernelLock}_{\mathsf{dom}(I_{\mathsf{K}}) - L_2} \star_{\mathsf{K}} \mathsf{IPercpu}_{\mathsf{CPUid} - V}. \tag{A 34}$$

We have:

$$\begin{array}{lll} & f_c^k(l, l', (r, h, L, v)) & \\ = & f_c^k(l, l', \{(r, h_2', L_2, v)\} *_{\mathsf{K}} \{(\_, h_1', L_1, \bot)\}) & \textit{by (A 32)} \\ \sqsubseteq & f_c^k(l, l', G_{\mathsf{K}}(l) *_{\mathsf{K}} \{(\_, h_1' \uplus h_{\mathsf{K}}, L_1 \uplus L_{\mathsf{K}}, \bot)\}) & \textit{by (A 33)} \\ \sqsubseteq & f_c^k(l, l', G_{\mathsf{K}}(l)) *_{\mathsf{K}} \{(\_, h_1' \uplus h_{\mathsf{K}}, L_1 \uplus L_{\mathsf{K}}, \bot)\} & \textit{by (A 4)} \end{array}$$

By (A 13), $f_c^k(l, l', G_{\mathsf{K}}(l)) \neq \top$, hence, by (A 2) and (A 27), $\sigma' \neq \top$. Let $\sigma' = (R[k : r'], h', L)$. Then, by (A 27) and (A 13), for some $h_3 \in \mathsf{Heap}$ and $L_3 \in \mathsf{Lockset}$, we have $h' = h_3 \uplus h_1' \uplus h_{\mathsf{K}}$, $L = L_3 \uplus L_1 \uplus L_{\mathsf{K}}$ and $(r', h_3, L_3) \in G_{\mathsf{K}}(r'(\mathtt{ip}))$. Using (A 11), we conclude that for some $h_2'' \in \mathsf{Heap}$ and

$$h_0' \in [r'(\mathtt{sp})..(r'(\mathtt{ss}) + \mathtt{StackSize} - 1) : \_]$$

we have $h' = h_2'' \uplus h_0' \uplus h_1'$ and

$$(\{r'\}, h_2'', L_2, \{k\} - (\mathsf{CPUid} - V)) \in \lfloor [\![\Delta_{\mathsf{K}}(r'(\mathtt{ip}))]\!]_\eta \rfloor^{\mathsf{KA}}_{r'} \star_{\mathsf{K}} \{(\emptyset, h_{\mathsf{K}}, L_{\mathsf{K}}, \emptyset)\}.$$

From this and (A 5), we get $r'(\mathtt{ip}) \in \mathsf{dom}(\mathsf{K})$. Let $M' = (M - \{r\}) \uplus \{r'\}$. Then by (A 34) this implies

$$(M', h_2'', L_2, V) \in \mathsf{IKernel}_\eta \star_{\mathsf{K}} \mathsf{IKernelLock}_{\mathsf{dom}(I_{\mathsf{K}}) - L_2} \star_{\mathsf{K}} \mathsf{IPercpu}_{\mathsf{CPUid} - V}. \tag{A 35}$$

Let $h_1'' = h_1' \uplus h_0'$. Then from (A 31) we get

$$((r', h_1'', L_1), M') \in ([\![\mathsf{SchedSleep}_k(r'(\mathtt{ip}))]\!]_\eta \cap \mathsf{at}_{\mathsf{S}}(r'(\mathtt{ip}))) *_{\mathsf{S}} \{((\_, h_{\mathsf{S}}, L_{\mathsf{S}}), M_{\mathsf{S}})\}.$$

Since $r'(\mathtt{ip}) \in \mathsf{dom}(\mathsf{K})$, together with (A 29) for $W = V$ and $W_1 = L_1$, this implies

$$((R[k : r'], h_1'', L_1), M', V) \in \mathsf{ISched}_\eta \star_{\mathsf{S}} \mathsf{ISchedLock}_{\mathsf{dom}(I_{\mathsf{S}}) - L_1}.$$

[4] Note that if there are several occurrences of r in M, $M - \{r\}$ removes only one of them.

By the definition of $\star_{\mathsf{SK}}$, from this and (A 35) we get $\sigma' \in \mathsf{IOs}_\eta$.

Case 6. σ' is obtained by applying the first rule in Figure 12, with the kernel executing `lock`. In this case

$$l \in \mathsf{labels}(\mathsf{K}), \quad c = \mathtt{lock}(\ell), \quad \ell \notin L, \quad \sigma' = (R[k : r[\mathtt{ip} : l']], h, L \cup \{\ell\}).$$

As in Case 5, there exist $h_\mathsf{S}, L_\mathsf{S}, M_\mathsf{S}, V_\mathsf{S}, V_\mathsf{K}, h_1', h_0, h_2', v$ satisfying the conditions stated there. Additionally, from (A 25) for some $h_\mathsf{K}, L_\mathsf{K}$ we get

$$(r, h_2', L_2, v) \in G_\mathsf{K}(l) *_\mathsf{K} \mathscr{I}_\mathsf{K}(\ell) *_\mathsf{K} \{(\_, h_\mathsf{K}, L_\mathsf{K}, \bot)\}$$

and

$$(M - \{r\}, h_\mathsf{K}, L_\mathsf{K}, V - \{k\}) \in \circledast_{r'' \in M - \{r\}}^{\star_\mathsf{K}} \lfloor [\![\Delta_\mathsf{K}(r''(\mathtt{ip}))]\!]_\eta \rfloor_{r''}^{\mathsf{KA}} \star_\mathsf{K} \mathsf{IKernelLock}_{\mathsf{dom}(I_\mathsf{K}) - (L_2 \cup \{\ell\})} \star_\mathsf{K} \mathsf{IPercpu}_{\mathsf{CPUid} - V}. \quad \text{(A 36)}$$

This implies

$$(r[\mathtt{ip} : l'], h_2', L_2 \cup \{\ell\}, v) \in \mathsf{to}_\mathsf{K}(l', (G_\mathsf{K}(l) *_\mathsf{K} \mathscr{I}_\mathsf{K}(\ell) *_\mathsf{K} \mathsf{Ik}_\mathsf{K}(\ell))) *_\mathsf{K} \{(\_, h_\mathsf{K}, L_\mathsf{K}, \bot)\}.$$

Hence, by (A 15)

$$(r[\mathtt{ip} : l'], h_2', L_2 \cup \{\ell\}, v) \in G_\mathsf{K}(l') *_\mathsf{K} \{(\_, h_\mathsf{K}, L_\mathsf{K}, \bot)\}.$$

Then from (A 11) it follows that

$$(\{r[\mathtt{ip} : l']\}, h_2, L_2 \cup \{\ell\}, \{k\} - (\mathsf{CPUid} - V)) \in \lfloor [\![\Delta_\mathsf{K}(l')]\!]_\eta \rfloor_{r[\mathtt{ip}:l']}^{\mathsf{KA}} \star_\mathsf{K} \{(\emptyset, h_\mathsf{K}, L_\mathsf{K}, \emptyset)\}.$$

From this and (A 5), we get $l' \in \mathsf{dom}(\mathsf{K})$. Let $M' = (M - \{r\}) \uplus \{r[\mathtt{ip} : l']\}$. Then by (A 36) we get

$$(M', h_2, L_2 \cup \{\ell\}, V) \in \mathsf{IKernel}_\eta \star_\mathsf{K} \mathsf{IKernelLock}_{\mathsf{dom}(I_\mathsf{K}) - (L_2 \cup \{\ell\})} \star_\mathsf{K} \mathsf{IPercpu}_{\mathsf{CPUid} - V}. \quad \text{(A 37)}$$

From (A 30) we get

$$((r[\mathtt{ip} : l'], h_1, L_1), M') \in ([\![\mathsf{SchedSleep}_k(l')]\!]_\eta \cap \mathsf{at}_\mathsf{S}(l')) *_\mathsf{S} \{((\_, h_\mathsf{S}, L_\mathsf{S}), M_\mathsf{S})\}.$$

Since $l' \in \mathsf{dom}(\mathsf{K})$, together with (A 29) for $W = V - \{k\}$ and $W_1 = L_1$, this implies

$$((R[k : r[\mathtt{ip} : l']], h_1, L_1), M', V) \in \mathsf{ISched}_\eta \star_\mathsf{S} \mathsf{ISchedLock}_{\mathsf{dom}(I_\mathsf{S}) - L_1}. \quad \text{(A 38)}$$

By the definition of $\star_{\mathsf{SK}}$, from this and (A 37) we get $\sigma' \in \mathsf{IOs}_\eta$.

Case 7. σ' is obtained by applying the first or the third rule in Figure 12, with the kernel executing `unlock`. In this case, $l \in \mathsf{labels}(\mathsf{K})$, $c = \mathtt{unlock}(\ell)$ and (A 27) holds.

As in Case 5, there exist $h_\mathsf{S}, L_\mathsf{S}, M_\mathsf{S}, V_\mathsf{S}, V_\mathsf{K}, h_1', h_0, h_2', v, h_\mathsf{K}, L_\mathsf{K}$ satisfying the conditions stated there. Then using (A 33), we get

$$(r[\mathtt{ip} : l'], h_2', L_2, v) \in \mathsf{to}_\mathsf{K}(l', G_\mathsf{K}(l)) *_\mathsf{K} \{(\_, h_\mathsf{K}, L_\mathsf{K}, \bot)\}.$$

Hence, by (A 17)

$$(r[\mathtt{ip} : l'], h_2', L_2, v) \in G_\mathsf{K}(l') *_\mathsf{K} \mathscr{I}_\mathsf{K}(\ell) *_\mathsf{K} \mathsf{Ik}_\mathsf{K}(\ell) *_\mathsf{K} \{(\_, h_\mathsf{K}, L_\mathsf{K}, \bot)\}.$$

Hence, $\ell \in L_2$, which means that $\sigma' \neq \top$. Then $\sigma' = (R[k : r[\mathtt{ip} : l']], h, L - \{\ell\})$. The above also implies

$$(r[\mathtt{ip} : l'], h_2', L_2 - \{\ell\}, v) \in G_{\mathsf{K}}(l') *_{\mathsf{K}} \mathscr{I}_{\mathsf{K}}(\ell) *_{\mathsf{K}} \{(\_, h_{\mathsf{K}}, L_{\mathsf{K}}, \bot)\}.$$

Then from (A 11) it follows that

$$(\{r[\mathtt{ip} : l']\}, h_2, L_2 - \{\ell\}, \{k\} - (\mathsf{CPUid} - V)) \in \\ \lfloor [\![\Delta_{\mathsf{K}}(l')]\!]_\eta \rfloor^{\mathsf{KA}}_{r[\mathtt{ip}:l']} \star_{\mathsf{K}} \lfloor \mathscr{I}_{\mathsf{K}}(\ell) *_{\mathsf{K}} \{(\_, h_{\mathsf{K}}, L_{\mathsf{K}}, \bot)\} \rfloor^{\mathsf{KL}}.$$

From this and (A 5) we get $l' \in \mathsf{dom}(\mathsf{K})$. Let $M' = (M - \{r\}) \uplus \{r[\mathtt{ip} : l']\}$. Then by (A 34) we get

$$(M', h_2, L_2 - \{\ell\}, V) \in \mathsf{IKernel}_\eta \star_{\mathsf{K}} \mathsf{IKernelLock}_{\mathsf{dom}(I_{\mathsf{K}}) - (L_2 - \{\ell\})} \star_{\mathsf{K}} \mathsf{IPercpu}_{\mathsf{CPUid} - V}.$$

As in the previous case, from (A 30) and (A 29) for $W = V - \{k\}$ and $W_1 = L_1$, we can establish (A 38). Together with the last inclusion, this implies $\sigma' \in \mathsf{IOs}_\eta$.

Case 8. σ' is obtained by applying the first or the third rule in Figure 12, with the kernel executing `cli` or `sti`. These cases are similar to the previous two and are omitted. They rely on (A 18) and (A 19) instead of (A 15) and (A 17).

Case 9. σ' is obtained by applying the first or the third rule in Figure 12, with the kernel executing `icall(schedule)`. In this case, $l \in \mathsf{labels}(\mathsf{K})$, $c = \mathtt{icall(schedule)}$ and (A 27) holds.

As in Case 5, there exist $h_{\mathsf{S}}, L_{\mathsf{S}}, M_{\mathsf{S}}, V_{\mathsf{S}}, V_{\mathsf{K}}, h_1', h_0, h_2', v, h_{\mathsf{K}}, L_{\mathsf{K}}$ satisfying the conditions stated there. Additionally, $r(\mathtt{if}) = 1$ and hence, $k \notin V$ and $v = \bot$. From (A 33) we then get

$$(r[\mathtt{ip} : l'], h_2', L_2, \bot) \in \mathsf{to}_{\mathsf{K}}(l', G_{\mathsf{K}}(l)) *_{\mathsf{K}} \{(\_, h_{\mathsf{K}}, L_{\mathsf{K}}, \bot)\}.$$

By (A 20), this implies

$$(r[\mathtt{ip} : l'], h_2', L_2, \bot) \in G_{\mathsf{K}}(l') *_{\mathsf{K}} \{(\_, h_{\mathsf{K}}, L_{\mathsf{K}}, \bot)\}.$$

Then using (A 11) we get

$$(\{r[\mathtt{ip} : l']\}, h_2, L_2, \emptyset) \in \lfloor [\![\Delta_{\mathsf{K}}(l')]\!]_\eta \rfloor^{\mathsf{KA}}_{r[\mathtt{ip}:l']} \star_{\mathsf{K}} \{(\_, h_{\mathsf{K}}, L_{\mathsf{K}}, \emptyset)\}.$$

Let $M' = (M - \{r\}) \uplus \{r[\mathtt{ip} : l']\}$. Then by (A 34) we have

$$(M', h_2, L_2, V) \in \mathsf{IKernel}_\eta \star_{\mathsf{S}} \mathsf{IKernelLock}_{\mathsf{dom}(I_{\mathsf{K}}) - L_2} \star_{\mathsf{S}} \mathsf{IPercpu}_{\mathsf{CPUid} - V}. \tag{A 39}$$

From (A 30) we get $\mathsf{dom}(h) \supseteq \{r(\mathtt{sp}), \ldots, r(\mathtt{sp}) + m + 1\}$, which implies that $\sigma' \neq \top$. Then $\sigma' = (R[k : r''], h_1'' \uplus h_2, L)$, where

$$r'' = r[\mathtt{ip} : \mathtt{schedule}, \mathtt{sp} : (r(\mathtt{sp}) + m + 1), \mathtt{if} : 0]$$

and

$$h_1'' = h_1[r(\mathtt{sp}) : l', (r(\mathtt{sp}) + 1) : r(\mathtt{gr}_1), \ldots, (r(\mathtt{sp}) + m) : r(\mathtt{gr}_m)]. \tag{A 40}$$

From (A 30) we also get

$$((r, h_1, L_1), M') \in \mathscr{J}_k *_{\mathsf{S}} \\ \{((\_, [r(\mathtt{sp}) .. (r(\mathtt{ss}) + \mathtt{StackSize} - 1) : \_] \uplus h_{\mathsf{S}}, L_{\mathsf{S}}), \{r[\mathtt{ip} : l']\} \uplus M_{\mathsf{S}})\}.$$

Hence,

$$((r'', h_1'', L_1), M') \in \mathscr{J}_k *_{\mathsf{S}} \{((\_, [(r''(\mathtt{sp}) - m - 1)..(r''(\mathtt{sp}) - 1) : l' r(\mathtt{gr}_1) \dots r(\mathtt{gr}_m), \\ r''(\mathtt{sp})..(r''(\mathtt{ss}) + \mathtt{StackSize} - 1) : \_] \uplus h_{\mathsf{S}}, L_{\mathsf{S}}), \{r[\mathtt{ip} : l']\} \uplus M_{\mathsf{S}})\}.$$

From (A 33) and (A 11) we get $0 \leqslant r(\mathtt{sp}) - r(\mathtt{ss}) \leqslant \mathtt{StackBound}$, so that $0 \leqslant r''(\mathtt{sp}) - r''(\mathtt{ss}) - m - 1 \leqslant \mathtt{StackBound}$. Thus,

$$((r'', h_1'', L_1), M') \in ([\![\mathsf{SchedState}_k]\!]_\eta \cap \mathsf{at}_{\mathsf{S}}(\mathtt{schedule})) *_{\mathsf{S}} \{((\_, h_{\mathsf{S}}, L_{\mathsf{S}}), M_{\mathsf{S}})\}.$$

Together with (A 29) for $W = V - \{k\}$ and $W_1 = L_1$ and (A 7), this implies

$$((R[k : r''], h_1'', L_1), M') \in \mathsf{ISched}_\eta \star_{\mathsf{S}} \mathsf{ISchedLock}_{\mathsf{dom}(I_{\mathsf{S}}) - L_1}.$$

By the definition of $\star_{\mathsf{SK}}$, from this and (A 39) we get $\sigma' \in \mathsf{IOs}_\eta$.

Case 10. σ' is obtained by applying the second or the last rule in Figure 12, i.e., by executing an interrupt. This case is virtually identical to the previous one and is omitted.

Case 11. σ' is obtained by applying the first or the third rule in Figure 12, with the scheduler executing `iret` at l_s or l_c. In this case

$$l \in \{l_s, l_c\}, \quad l' \in \{l_s + 1, l_c + 1\}, \quad c = \mathtt{iret}$$

and (A 27) holds.

From (A 24), there exist $h_{\mathsf{S}}, L_{\mathsf{S}}, M_{\mathsf{S}}, V_{\mathsf{S}}, V_{\mathsf{K}}$ satisfying the conditions stated there, in particular, (A 28) and (A 29) for $W = V$ and $W_1 = L_1$. Then from (A 28), (A 8) and (A 10) we get

$$((r, h_1, L_1), M) \in ([\![\mathsf{SchedState}_k]\!]_\eta \cap (\mathsf{at}_{\mathsf{S}}(l_s) \cup \mathsf{at}_{\mathsf{S}}(l_c))) *_{\mathsf{S}} \{((\_, h_{\mathsf{S}}, L_{\mathsf{S}}), M_{\mathsf{S}})\}. \quad \text{(A 41)}$$

Hence, $\mathsf{dom}(h_1) \supseteq \{r(\mathtt{sp}) - m - 1, \dots, r(\mathtt{sp}) - 1\}$ and $\sigma' \neq \top$. Let

$$l'' = h_1(r(\mathtt{sp}) - m - 1), \quad g_1 = h_1(r(\mathtt{sp}) - m), \quad \dots, \quad g_m = h_1(r(\mathtt{sp}) - 1).$$

Then, $\sigma' = (R[k : r'], h, L)$, where

$$r' = r[\mathtt{ip} : l'', \mathtt{sp} : (r(\mathtt{sp}) - m - 1), \mathtt{gr}_1 : g_1, \dots, \mathtt{gr}_m : g_m, \mathtt{if} : 1].$$

From (A 41) we now obtain

$$((r[\mathtt{ip} : l''], h_1, L_1), M) \in ([\![\mathsf{SchedState}_k]\!]_\eta \cap \mathsf{at}_{\mathsf{S}}(l'')) *_{\mathsf{S}} \{((\_, h_{\mathsf{S}}, L_{\mathsf{S}}), M_{\mathsf{S}})\}.$$

Hence,

$$((r', h_1, L_1), M) \in \\ \mathsf{at}_{\mathsf{S}}(l'') \cap (\{((\_, [\mathtt{sp}..(\mathtt{ss} + \mathtt{StackSize} - 1) : \_] \uplus h_{\mathsf{S}}, L_{\mathsf{S}}), \{r'\} \uplus M_{\mathsf{S}})\} *_{\mathsf{S}} \mathscr{J}_k),$$

which is equivalent to

$$((r', h_1, L_1), M) \in ([\![\mathsf{SchedSleep}_k(l'')]\!]_\eta \cap \mathsf{at}_{\mathsf{S}}(l'')) *_{\mathsf{S}} \{((\_, h_{\mathsf{S}}, L_{\mathsf{S}}), M_{\mathsf{S}})\}.$$

Note that $r' \in M$. Hence, from (A 25) and (A 5) we get $l'' \in \mathsf{labels}(\mathsf{K})$. By (A 29) for $W = V$ and $W_1 = L_1$ we then have

$$((R[k : r'], h_1, L_1), M) \in \mathsf{ISched}_\eta \star_{\mathsf{S}} \mathsf{ISchedLock}_{\mathsf{dom}(I_{\mathsf{S}}) - L_1}.$$

From (A 25) and the definition of $\star_{\mathsf{SK}}$, we get $\sigma' \in \mathsf{IOs}_\eta$.

Case 12. σ' is obtained by applying the first or the third rule in Figure 12, with the kernel executing `icall(create)`. In this case, $l \in \mathsf{labels}(\mathsf{K})$, $c = \texttt{icall(create)}$ and (A 27) holds.

As in Case 5, there exist $h_\mathsf{S}, L_\mathsf{S}, M_\mathsf{S}, V_\mathsf{S}, V_\mathsf{K}, h'_1, h_0, h'_2, h_\mathsf{K}, L_\mathsf{K}$ satisfying the conditions stated there. Additionally, $r(\texttt{if}) = 1$ and hence, $k \notin V$ and $v = \bot$. From (A 33) and (A 21) we get

$$(r, h'_2, L_2, \bot) \in \{(\_, h_\mathsf{K}, L_\mathsf{K})\} *_\mathsf{K} [\![\exists \gamma.\, \gamma(\mathbf{if}) = 1 \wedge \mathsf{desc}(\texttt{gr}_1, \gamma) * P * Q]\!]_\eta.$$

Hence, there exists r' such that $r'(\texttt{if}) = 1$ and

$$(r, h'_2, L_2, \bot) \in \boldsymbol{desc}_\eta(u, r') *_\mathsf{K} [\![P]\!]_{\eta'} *_\mathsf{K} [\![Q]\!]_{\eta'} *_\mathsf{K} \{(\_, h_\mathsf{K}, L_\mathsf{K}, \bot)\},$$

where $u = r(\texttt{gr}_1)$ and $\eta' = \eta[\gamma : r']$. Since $\mathsf{free}(P) \cap \mathsf{Reg} = \emptyset$ and $\mathsf{free}(\mathsf{desc}(d, \gamma)) \cap \mathsf{Reg} = \emptyset$, we have

$$(r[\texttt{ip} : l'], h'_2, L_2, \bot) \in \boldsymbol{desc}_\eta(u, r') *_\mathsf{K} [\![P]\!]_{\eta'} *_\mathsf{K} \mathsf{to}_\mathsf{K}(l', [\![Q]\!]_{\eta'}) *_\mathsf{K} \{(\_, h_\mathsf{K}, L_\mathsf{K}, \bot)\}.$$

Using (A 22), we then get

$$(r[\texttt{ip} : l'], h'_2, L_2, \bot) \in \boldsymbol{desc}_\eta(u, r') *_\mathsf{K} [\![P]\!]_{\eta'} *_\mathsf{K} G_\mathsf{K}(l') *_\mathsf{K} \{(\_, h_\mathsf{K}, L_\mathsf{K}, \bot)\}.$$

According to (A 11), this implies

$$(\{r[\texttt{ip} : l']\}, h_2, L_2, \emptyset) \in \lfloor \boldsymbol{desc}_\eta(u, r') \rfloor^{\mathsf{KL}} \star_\mathsf{K} \lfloor [\![P]\!]_{\eta'} \rfloor^{\mathsf{KL}} \star_\mathsf{K} \lfloor [\![\Delta_\mathsf{K}(l')]\!]_\eta \rfloor^{\mathsf{KA}}_{r[\texttt{ip}:l']} \star_\mathsf{K} \{(\emptyset, h_\mathsf{K}, L_\mathsf{K}, \emptyset)\}.$$

Then for some $h''_2, h_d \in \mathsf{Heap}$ such that $h_2 = h''_2 \uplus h_d$ we have $\{(\_, h_d, \emptyset, \bot)\} \subseteq \boldsymbol{desc}_\eta(u, r')$ (recall that all states from $\boldsymbol{desc}_\eta(u, r')$ have an empty lockset) and

$$(\{r[\texttt{ip} : l']\}, h''_2, L_2, \emptyset) \in \lfloor [\![P]\!]_{\eta'} \rfloor^{\mathsf{KL}} \star_\mathsf{K} \lfloor [\![\Delta_\mathsf{K}(l')]\!]_\eta \rfloor^{\mathsf{KA}}_{r[\texttt{ip}:l']} \star_\mathsf{K} \{(\emptyset, h_\mathsf{K}, L_\mathsf{K}, \emptyset)\}.$$

Then from (A 23) and (A 11) we get

$$(\{r[\texttt{ip} : l'], r'\}, h''_2, L_2, \emptyset) \in \lfloor [\![\Delta_\mathsf{K}(r'(\texttt{ip}))]\!]_\eta \rfloor^{\mathsf{KA}}_{r'} \star_\mathsf{K} \lfloor [\![\Delta_\mathsf{K}(l')]\!]_\eta \rfloor^{\mathsf{KA}}_{r[\texttt{ip}:l']} \star_\mathsf{K} \{(\emptyset, h_\mathsf{K}, L_\mathsf{K}, \emptyset)\}.$$

Let $M' = (M - \{r\}) \uplus \{r[\texttt{ip} : l'], r'\}$. Then by (A 34) we have

$$(M', h''_2, L_2, V) \in \mathsf{IKernel}_\eta \star_\mathsf{S} \mathsf{IKernelLock}_{\mathsf{dom}(I_\mathsf{K}) - L_2} \star_\mathsf{S} \mathsf{IPercpu}_{\mathsf{CPUid} - V}. \tag{A 42}$$

As in Case 9, we can assume that

$$\sigma' = (R[k : r''], h''_1 \uplus h_2, L) = (R[k : r''], h''_1 \uplus h_d \uplus h''_2, L),$$

where

$$r'' = r[\texttt{ip} : \texttt{create}, \texttt{sp} : (r(\texttt{sp}) + m + 1), \texttt{if} : 0]$$

and h''_1 is defined by (A 40). Let $h'''_1 = h''_1 \uplus h_d$. Then from (A 30) we get

$$\begin{aligned}((r, h'''_1, L_1), M') \in \mathscr{J}_k *_\mathsf{S} (\boldsymbol{desc}_\eta(u, r') \times \{\emptyset\}) *_\mathsf{S} \\ \{((\_, [r(\texttt{sp})..(r(\texttt{sp}) + m) : l' r(\texttt{gr}_1) \ldots r(\texttt{gr}_m), \\ (r(\texttt{sp}) + m + 1)..(r(\texttt{ss}) + \texttt{StackSize} - 1) : \_] \uplus h_\mathsf{S}, L_\mathsf{S}), \{r[\texttt{ip} : l'], r'\} \uplus M_\mathsf{S})\}.\end{aligned}$$

Similar to how it was done in Case 9, using (A 9) we now establish

$$((r'', h_1''', L_1), M') \in G_{\mathsf{S}}^{k}(\texttt{create}) *_{\mathsf{S}} \{((\_, h_{\mathsf{S}}, L_{\mathsf{S}}), M_{\mathsf{S}})\}.$$

Together with (A 29) for $W = V$ and $W_1 = L_1$, this implies

$$((R[k : r''], h_1''', L_1), M', V) \in \mathsf{ISched}_{\eta} \star_{\mathsf{S}} \mathsf{ISchedLock}_{\mathsf{dom}(I_{\mathsf{S}}) - L_1}.$$

By the definition of $\star_{\mathsf{SK}}$, from this and (A 42) we get $\sigma' \in \mathsf{IOs}_{\eta}$. □

References

Back, R.-J. (1981) On correct refinement of programs. *J. Comput. Syst. Sci.* **23**, 49–68.

Berdine, J., O'Hearn, P. W., Reddy, U. S. & Thielecke, H. (2002) Linear continuation-passing. *Higher-order Symb. Comput.* **15**(2–3), 181–208.

Bovet, D. & Cesati, M. (2005) *Understanding the Linux Kernel*, 3rd ed. O'Reilly.

Brookes, S. D. (2007) A semantics of concurrent separation logic. *Theor. Comput. Sci.* **375**(1–3), 227–270.

Calcagno, C., O'Hearn, P. W. & Yang, H. (2007) Local action and abstract separation logic. In *Symposium on Logic in Computer Science (LICS'07)*. IEEE, pp. 366–378.

Charlton, N. (2011) Hoare logic for higher order store using simple semantics. In *Conference on Logic, Language, Information and Computation (WoLLIC'11)*. LNCS, vol. 6642. Springer, pp. 52–66.

Clarke, D. G., Noble, J. & Potter, J. (2001) Simple ownership types for object containment. In *European Conference on Object-Oriented Programming (ECOOP'01)*. LNCS, vol. 2072. Springer, pp. 53–76.

Cohen, E., Schulte, W. & Tobies, S. (2010) Local verification of global invariants in concurrent programs. In *Conference on Computer-Aided Verification (CAV'10)*. LNCS, vol. 6174. Springer, pp. 480–494.

Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M. & Yang, H. (2013) Views: Compositional reasoning for concurrent programs. In *Symposium on Principles of Programming Languages (POPL'13)*. ACM, pp. 287–300.

Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M. & Vafeiadis, V. (2010) Concurrent abstract predicates. In *European Conference on Object-Oriented Programming (ECOOP'10)*. LNCS, vol. 6183. Springer, pp. 504–528.

Feng, X., Ferreira, R. & Shao, Z. (2007a) On the relationship between concurrent separation logic and assume-guarantee reasoning. In *European Conference on Programming (ESOP'07)*. LNCS, vol. 4421. Springer, pp. 173–188.

Feng, X., Ni, Z., Shao, Z. & Guo, Y. (2007b) An open framework for foundational proof-carrying code. In *Workshop on Types in Language Design and Implementation (TLDI'07)*. ACM, pp. 67–78.

Feng, X., Shao, Z., Dong, Y. & Guo, Y. (2008a) Certifying low-level programs with hardware interrupts and preemptive threads. In *Conference on Programming Language Design and Implementation (PLDI'08)*. ACM, pp. 170–182.

Feng, X., Shao, Z., Guo, Y. & Dong, Y. (2008b) Combining domain-specific and foundational logics to verify complete software systems. In *Conference on Verified Software: Theories, Tools, Experiments (VSTTE'08)*. LNCS, vol. 5295. Springer, pp. 54–69.

Feng, X., Shao, Z., Vaynberg, A., Xiang, S. & Ni, Z. (2006) Modular verification of assembly code with stack-based control abstractions. In *Conference on Programming Language Design and Implementation (PLDI'06)*. ACM, pp. 401–414.

Gargano, M., Hillebrand, M., Leinenbach, D. & Paul, W. (2005) On the correctness of operating system kernels. In *Conference on Theorem Proving in Higher-Order Logics (TPHOLs'05)*. LNCS, vol. 3603. Springer, pp. 1–16.

Gotsman, A. (2009) *Logics and Analyses for Concurrent Heap-Manipulating Programs*. PhD Thesis, University of Cambridge.

Gotsman, A., Berdine, J. & Cook, B. (2011) Precision and the conjunction rule in concurrent separation logic. *ENTCS* **276**(1), 171–190. MFPS'11: Mathematical Foundations of Programming Semantics.

Gotsman, A., Berdine, J., Cook, B., Rinetzky, N. & Sagiv, M. (2007) Local reasoning for storable locks and threads. In *Asian Symposium on Programming Languages and Systems (APLAS'07)*. LNCS, vol. 4807. Springer, pp. 19–37.

Gotsman, A. & Yang, H. (2013) *Electronic Appendix for This Paper*. Available from http://dx.doi.org/10.1017/S0956796813000075.

Hasegawa, M. (2002) Linearly used effects: Monadic and CPS transformations into the linear lambda calculus. In *International Symposium on Functional and Logic Programming (FLOPS'02)*. LNCS, vol. 2441. Springer, pp. 167–182.

Hasegawa, M. (2004) Semantics of linear continuation-passing in call-by-name. In *International Symposium on Functional and Logic Programming (FLOPS'04)*. LNCS, vol. 2998. Springer, pp. 229–243.

Jones, C. (2007) Splitting atoms safely. *Theor. Comput. Sci.* **375**, 109–119.

Jones, C. B. (1983) Specification and design of (parallel) programs. In *IFIP Congress*, pp. 321–332.

Klein, G. (2009) Operating system verification–an overview. *Sādhanā* **34**, 26–69.

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H. & Winwood, S. (2009) seL4: Formal verification of an OS kernel. In *Symposium on Operating Systems Principles (SOSP'09)*. ACM, pp. 207–220.

Laird, J. (2005) Game semantics and linear CPS interpretation. *Theor. Comput. Sci.* **333**(1–2), 199–224.

Love, R. (2010) *Linux Kernel Development*, 3rd ed. Addison Wesley.

Maeda, T. & Yonezawa, A. (2009) Writing an OS kernel in a strictly and statically typed language. In *Formal to Practical Security*. LNCS, vol. 5458. Springer, pp. 181–197.

Ni, Z. & Shao, Z. (2006) Certified assembly programming with embedded code pointers. In *Symposium on Principles of Programming Languages (POPL'06)*. ACM, pp. 320–333.

O'Hearn, P. W. (2007) Resources, concurrency and local reasoning. *Theor. Comput. Sci.* **375**, 271–307.

Parkinson, M. & Bierman, G. (2005) Separation logic and abstraction. In *Symposium on Principles of Programming Languages (POPL'05)*. ACM, pp. 247–258.

Pnueli, A. (1985) In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems*. Springer, pp. 123–144.

Reynolds, J. C. (2002) Separation logic: A logic for shared mutable data structures. In *Symposium on Logic in Computer Science (LICS'02)*. IEEE, pp. 55–74.

Schwinghammer, J., Birkedal, L., Reus, B. & Yang, H. (2009) Nested Hoare triples and frame rules for higher-order store. In *Conference on Computer Science Logic (CSL'09)*. LNCS, vol. 5771. Springer, pp. 440–454.

Shao, Z. (2010) Certified software. *Commun. ACM* **53**(12), 56–66.

Thielecke, H. (2003) From control effects to typed continuation passing. In *Symposium on Principles of Programming Languages (POPL'03)*. ACM, pp. 139–149.

Turon, A. & Wand, M. (2011) A separation logic for refining concurrent objects. In *Symposium on Principles of Programming Languages (POPL'11)*. ACM, pp. 247–258.

Vafeiadis, V. & Parkinson, M. J. (2007) A marriage of rely/guarantee and separation logic. In *Conference on Concurrency Theory (CONCUR'07)*. LNCS, vol. 4703. Springer, pp. 256–271.

Yang, J. & Hawblitzel, C. (2010) Safe to the last instruction: Automated verification of a type-safe operating system. In *Conference on Programming Language Design and Implementation (PLDI'10)*. ACM, pp. 99–110.