Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Proving consistency of concurrent data structures and transactional memory systems

Ph.D Thesis

Artem Khyzha

2018

Departamento de Lenguajes y Sistemas Informáticos e Ingenieria de Software

Escuela Técnica Superior de Ingenieros Informáticos

Proving consistency of concurrent data structures and transactional memory systems

Submitted in partial fulfillment of the requirements for the degree of:
Doctor of Philosophy in Software, Systems and Computing

Author: **Artem Khyzha**
Master in Software and Systems
Universidad Politécnica de Madrid

Advised by Prof. Alexey Gotsman
IMDEA Software Institute

Thesis Committee:

Manuel Carro Liñares	Universidad Politécnica de Madrid
Aleksandar Nanevski	IMDEA Software Institute
Matthew Parkinson	Microsoft Research
Ana Sokolova	University of Salzburg
Viktor Vafeiadis	Max Planck Institute for Software Systems

2018

Resumen

Los programadores pueden afrontar la complejidad de escribir software concurrente con la ayuda de librerías de estructuras de datos concurrentes y, más recientemente, memoria transaccional. Ambos enfoques facilitan el desarrollo de software proporcionando al programador garantías de corrección, que abstraen los detalles de la implementación de librerías y memoria transaccional: el programador puede asumir que los métodos y transacciones se ejecutan atómicamente, aún cuando sus implementaciones en realidad sean concurrentes. Habitualmente, los investigadores, para justificar estas garantías de correción, demuestran ciertos requisitos de consistencia: linearizabilidad en el caso de estructuras de datos concurrentes, y opacidad en memoria transaccional.

Esta tesis se centra en demostrar formalmente requisitos de consistencia. Proponemos una lógica modular y técnicas de demostración capaces de razonar sobre implementaciones de estructuras de datos concurrentes de grano fino, y sobre memoria transaccional. También proporcionamos fundamentos formales para un modelo de programación en el que se puede acceder a la memoria tanto desde dentro como desde fuera de una transacción.

Nuestra primera aportación es una lógica general para demostrar linearizabilidad, basada en la lógica "Views framework" y puede ser instanciada con diferentes métodos de razonamiento composicional sobre la concurrencia, tales como la lógica de separación o la lógica "rely-guarantee". Independientemente de la elección que se haga acerca del método de razonamiento modular con respecto a un hilo, la logica genérica aquí propuesta explica los principios de demostración de la linearizabilidad, mediante el método de punto de linearizabilidad, habitualmente utilizado. También demostramos la solidez de nuestra logica general.

En nuestra segunda aportación, proponemos un método de demostración de la linearizabilidad de ciertas estructuras de datos sobre las que es complicado razonar usando el método de puntos de linearizabilidad (por ejemplo, la Herlihy-Wing queue y la Time-Stamped queue). La idea clave de nuestro método es la construcción incremental, no de un único orden lineal de operaciones, sino de un orden parcial que describe múltiples linearizaciones que satisfacen la especificación secuencial. Esto permite retrasar las decisiones acerca del orden de las operaciones, imitando el comportamiento de la implementación de las estructuras de datos. Formalizamos nuestro método como una lógica basada en "rely-guarantee", y demostramos su efectividad verificando diferentes estructuras de datos retadoras: las colas Herlihy-Wing y Time-Stamped, así como el Optimistic set.

Por último, proponemos fundamentos para un modelo de programación, donde el programador puede acceder los mismos datos tanto desde dentro como

desde fuera de una transacción. Idealmente, éstos accesos se ejecutan con garantías atómicas fuertes. Desafortunadamente, muchas implementaciones de memoria transaccional que satisfacen el criterio de opacidad no proporcionan estas garantías, dado que en la práctica son prohibitivamente caras. Diversos investigadores han propuesto garantizar atomicidad fuerte sólo para ciertos programas libres de condiciones de carrera, y, en particular, para programas que realizan accesos no transaccionales siguiendo el estilo de privatización. Nuestra contribución es proponer una definición de condición de carrera que dé cabida a la privatización. Demostramos que si tenemos una memoria transaccional que satisface ciertos requisitos que generalizan la opacidad y un programa libres de condiciones de carrera asumiendo atomicidad fuerte, entonces dicho programa, en efecto, tiene una semántica de atomicidad fuerte. Mostramos que nuestra definición de condición de carrera permite al programador seguir los estilos de privatización. También proponemos un método para demostrar nuestra generalización de opacidad y aplicarlo a la memoria transaccional TL2.

Abstract

Programmers can manage the complexity of constructing concurrent software with the help of libraries of concurrent data structures and, more recently, transactional memory. Both approaches facilitate software development by giving a programmer correctness guarantees that abstract from the implementation details of libraries and transactional memory: the programmer can expect that each library method or transaction execute atomically, even if in reality the implementation executes them concurrently. To justify these correctness guarantees, researchers typically prove certain consistency conditions: linearizability for concurrent data structures and opacity for transactional memory.

This dissertation is dedicated to proving consistency conditions formally. We propose modular program logics and proof techniques capable of reasoning about sophisticated fine-grained concurrent implementations of data structures and transactional memory. We also provide formal foundations for the programming model in which memory can be accesses both inside and outside of transactions.

Our first contribution is a generic logic for proving linearizability, which builds on the Views framework and can be instantiated with different means of compositional reasoning about concurrency, such as separation logic or rely-guarantee. The proposed generic logic explicates principles of proving linearizability with commonly-used linearization-point method independently from a particular choice of an approach to thread-modular reasoning. We also prove our generic logic sound.

In our second contribution, we propose a proof method for linearizability of data structures that are challenging to reason about using linearization points (e.g., the Herlihy-Wing queue and the Time-Stamped queue). The key idea of our method is to incrementally construct not a single linear order of operations, but a partial order that describes multiple linearizations satisfying the sequential specification. This allows decisions about the ordering of operations to be delayed, mirroring the behavior of data structure implementations. We formalize our method as a program logic based on rely-guarantee reasoning, and demonstrate its effectiveness by verifying several challenging data structures: the Herlihy-Wing queue, the Time-Stamped queue and the Optimistic set.

Finally, we provide foundations for a programming model where a programmer can access the same data both inside and outside of transactions. In this model programmers would like to have strong atomicity guarantees from TM. Unfortunately, many implementations satisfying opacity do not provide them, since it is prohibitively expensive in practice. Researchers have suggested guaranteeing strong atomicity only for certain data-race free (DRF) programs and, in particular, for programs performing non-transactional accesses according to the privatization idiom. Our contribution is to propose a notion of DRF that

supports privatization. We prove that, if a TM satisfies a certain condition generalizing opacity and a program using it is DRF *assuming* strong atomicity, then the program indeed has strongly atomic semantics. We show that our DRF notion allows the programmer to use privatization idioms. We also propose a method for proving our generalization of opacity and apply it to the TL2 TM.

Acknowledgments

First and foremost, I want to thank my advisor, Alexey Gotsman, for his support and guidance during this research. He has provided incredibly helpful advice, patient mentoring and a generous amount of his time for discussions of countless research opportunities. Taking some of them has led to this dissertation.

I have been very fortunate to work with wonderful collaborators—Hagit Attiya, Mike Dodds, Matthew Parkinson and Noam Rinetzky—whom I thank for their guidance and assistance. Working with them was really inspiring and rewarding.

I am grateful to IMDEA Software Institute, where this research was undertaken. I am also grateful to Microsoft Research for providing me with Microsoft Research European PhD Scholarship.

I would like to give special thanks to Ilya Sergey and Giovanni Bernardi for many discussions about scientific ideas and research life, as well as an important dose of encouragement in the time when it was most necessary.

I am grateful to the following people for useful discussions and comments about this dissertation and the papers it is based on: Simon Doherty, Brijesh Dongol, Xinyu Feng, Ori Lahav, Mohsen Lesani, Adam Morrison, Azalea Raad, Ana Sokolova, Michael Spear, Tingzhe Zhou and everybody that I have forgotten to mention, as well as the anonymous referees who were exposed to early drafts of parts of my work.

I would like to thank the members of my thesis committee for taking time to read my thesis and for providing valuable feedback.

I was also fortunate to make great friends, with whom we shared great moments in Madrid: Alvaro García, Bogdan Kulynych, Borja de Régil, Damir Valput, German Delbianco, Goran Doychev, Julián Samborski, Mariam Nayem, Miguel Ambrona, Miriam García, Platon Kotzias, Srdjan Matic, Vincent Laporte, Yuri Meshman.

Finally, I am incredibly grateful to my family and my partner, Natalia, for their love and support.

Contents

Chapter 1

Introduction

The wide-spread use of multicore processors has significantly influenced the way we develop software. Modern software systems are often parallelized so that multiple threads of control could work on a single task together and benefit from parallelism of multicore architecture. To coordinate their work, threads interact by performing activities on data they share in memory, which requires synchronization to keep the data consistent.

Efficiency of synchronization of shared-memory accesses is critical for the performance of concurrent programs. In principle, threads could synchronize their accesses to data by protecting it with a single global lock so that only one thread at any time could access it. Such implementation strategy would be easy to reason about, but of limited use in practice, since it does not take advantage of the parallelism enabled by modern multiprocessors. To maximize performance, implementations of concurrent algorithms may use more elaborate locking schemes (such as hand-over-hand locking) or non-blocking techniques (based on using compare-and-swap and fetch-and-add instructions), allowing multiple threads to operate on the data structure with minimum synchronization. Unsurprisingly, efficient concurrent algorithms and data structures are notoriously challenging to implement and reason about, since that requires considering all possible interleavings between concurrently executing threads. To manage the complexity of constructing concurrent software, programmers can use libraries of *concurrent data structures* or *transactional memory*.

Libraries of *concurrent data structures* contain often-used functionality designed and implemented by expert programmers. These libraries (e.g., java.util.concurrent and Intel Threading Building Blocks) encapsulate highly-optimized implementations of data structures, such as queues and lists, and provide clients with a set of methods that can be called concurrently to operate on these. Even though library implementations typically allow different threads to modify data structures concurrently, they usually provide a guarantee to the programmer that each method behaves as if it executes atomically. In principle, the programmer using a concurrent data structure library can develop a program and reason about its correctness without knowing the library implementation details by relying on the atomicity guarantee.

Transactional memory (TM) [1, 2] offers an alternative programming model that facilitates the development of concurrent software by letting the programmer designate blocks of code as *transactions*. The programmer can then expect

that each transaction executes atomically and without interleaving with other transactions, even if a TM implementation actually executes transactions concurrently. A TM can be implemented in hardware [3, 4], software [2] or a combination of both [5, 6]. Analogously to concurrent data structure libraries, the programmer is not required to know the details of a TM implementation, as TM allows developing and reasoning about transactions as if they are atomic.

The guarantees of libraries and TM implementations can be formalized as *observational refinement*: every behavior a user can observe of any client program using a library implementation (a TM implementation) can also be observed when the program uses an abstract library (an abstract TM) that executes each method (each transaction) atomically. Since proving observational refinement directly requires considering every possible program using the implementation, it is typically proven indirectly via *consistency conditions*, which can be verified independently from client programs.

A commonly accepted consistency condition for concurrent data structures is *linearizability* [7]. It implies observational refinement under an assumption that client programs only interact with a data structure through its interface operations [8]. This assumption, also known as *interference freedom* [9], is common in concurrent data structure design. In Chapters 2 and 3 we present our contributions towards developing techniques for proving linearizability.

A commonly accepted consistency condition for transactional memory is *opacity* [10]. Similarly to linearizability, it implies observational refinement [11] under an assumption that memory accessed by transactions is never used outside of transactions in programs. However, programmers using a TM often would like to access the same data both inside and outside transactions, e.g., to improve performance or to support legacy code. One typical pattern for that is *privatization* [12]: from some point on, threads agree that a given object can be accessed only non-transactionally. A possibility of such non-transactional accesses necessitates a new consistency condition for TMs that could be used to guarantee atomicity of transactions in client programs.

When both transactional and non-transactional accesses to data are possible, programmers would ideally like the TM to guarantee *strong atomicity* [13, 14], which allows viewing transactions as executing atomically with respect to non-transactional accesses as well as other transactions. This is equivalent to considering every non-transactional access as a single-instruction transaction. Unfortunately, providing strong atomicity in software TMs requires instrumenting non-transactional accesses with additional instructions for maintaining TM metadata. This undermines scalability and makes it difficult to reuse legacy code. Since most existing TMs are either software-based or rely on a software fall-back, they do not perform such instrumentation and, hence, provide weaker atomicity guarantees. In Chapter 4, we provide foundations for the programming model mixing transactional and non-transactional accesses to data. To this end, we propose a restriction on client programs and a consistency condition for TMs, under which the programs get strong atomicity guarantees.

In the following, we give a more detailed overview of the contributions.

1.1 Proving Linearizability of Concurrent Data Structures

Linearizability requires that any execution of the data structure is justified by a *linearization* — a linear order on operations in the execution such that:

- the linearization respects the order of non-overlapping operations (the *real-time order*); and
- operations in the linearization behave atomically according to the data structure's sequential specification.

A common approach to proving linearizability is to find a *linearization point* for every method of the concrete library at which it can be thought of as taking effect. This must occur somewhere between the start and end of the operation, to ensure that the linearization preserves the real-time order. Given an execution of a data structure implementation, the matching linearization is constructed by executing an operation's specification at the linearization point of the operation's implementation.

Recent years have seen a number of proposals of program logics for proving linearizability. Although these logics differ in technical details, they embody similar reasoning principles. To explicate these principles, we propose a logic for proving linearizability that is generic: it can be instantiated with different means of compositional reasoning about concurrency, such as separation logic [15] or rely-guarantee [16]. To this end, we generalize the Views framework [17] for reasoning about concurrency to handle relations between programs, required for proving linearizability. We also consider sample instantiations of our generic logic and show that it is powerful enough to handle concurrent algorithms with challenging features, such as *helping* [9]. We present our generic logic for proving linearizability in Chapter 2 of this dissertation.

The linearization-point method of establishing linearizability is very popular, to the extent that most papers proposing new concurrent data structures include a placement of linearization points. For such algorithms, linearizability can be proved by incrementally constructing a linearization as the program executes. However, there are algorithms that cannot be proved linearizable using the linearization point method, because an operation's position in the linearization order may depend on future operations. This makes it very difficult to incrementally construct the linearization in a proof, because an operation's position in the linearization order may depend on future operations.

We propose a new proof method that can handle data structures with such future-dependent linearizations. Our key idea is to incrementally construct not a single linear order of operations, but a partial order that represents multiple linearizations satisfying the sequential specification. This allows decisions about the ordering of operations to be delayed, mirroring the behavior of data structure implementations. We formalize our method as a program logic based on rely-guarantee reasoning, and demonstrate its effectiveness by verifying several challenging data structures: the Herlihy-Wing queue [7], the Time-Stamped queue [18] and the Optimistic set [19]. We present our proof method for linearizability using partial orders in Chapter 3 of this dissertation.

1.2 Safe Privatization in Transactional Memory

Many TM implementations in practice only guarantee *weak atomicity* [13, 14], in which transactions are atomic only with respect to other transactions, but their execution may be interleaved with non-transactional code. Since guaranteeing strong atomicity for arbitrary programs is prohibitively expensive in software TMs, researchers have suggested guaranteeing strong atomicity only for certain *data-race free (DRF)* programs [20, 21], which do not access the same data concurrently inside and outside of transactions. For instance, in two typical patterns of non-transactional accesses, *privatization* and *publication*, at any point of time objects are accessed either only transactionally or only non-transactionally. Hence, programs using privatization and publication should be guaranteed strong atomicity.

Many TM implementations, when used out-of-the-box, do not guarantee strong atomicity for seemingly well-behaved programs using privatization. Supporting privatization safely in a TM with weaker atomicity guarantees is nontrivial: it often requires correctly inserting *transactional fences* [12], which wait until all active transactions complete. Thus, providing strong atomicity to DRF programs using privatization necessitates taking into account transactional fences.

In Chapter 4, we make the first proposal of DRF for transactional memory that considers a flexible programming model (with transactional fences) and comes with a formal justification of its appropriateness. We prove that, if a TM satisfies a certain consistency condition generalizing opacity and a program using it is DRF *assuming* strong atomicity, then the program indeed has strongly atomic semantics. The key feature of our proposal is that the programmer writing code without races according to our notion never needs to reason about weakly atomic semantics. Finally, we also show that our notion of DRF allows the programmer to use privatization idioms.

The new consistency condition that we require from TM implementations is *strong opacity*. It extends the standard notion of opacity [10], which, similarly to linearizability, requires that any execution of the TM is justified by a linearization — a linear order on transactions in the execution such that:

- the linearization respects the order of non-overlapping transactions (the *real-time order*); and
- transactions in the linearization behave as if they are executed atomically without interleaving with each other.

Strong opacity imposes additional requirements on non-transactional operations present in executions, because these can affect the behavior of the TM (e.g., via the idiom of "privatize, modify non-transactionally, publish").

To justify the appropriateness of requiring strong opacity from TMs, we develop a proof method for it inspired by our technique for linearizability using partial orders. We demonstrate the effectiveness of the method by applying it to prove the strong opacity of a realistic TM, TL2 [22], enhanced with transactional fences implemented using RCU. Our proof shows that this TM indeed guarantees strong atomicity to programs satisfying our notion of DRF.

1.3 List of Publications

This thesis is based on the following three papers published in proceedings of peer-reviewed academic conferences:

- Artem Khyzha, Alexey Gotsman, and Matthew Parkinson
  A Generic Logic for Proving Linearizability
  In Proceedings of International Symposium on Formal Methods (FM), Limassol, Cyprus, 2016.

- Artem Khyzha, Mike Dodds, Alexey Gotsman, and Matthew Parkinson
  Proving Linearizability Using Partial Orders
  In Proceedings of European Symposium on Programming (ESOP), Uppsala, Sweden, 2017.

- Artem Khyzha, Hagit Attiya, Alexey Gotsman, and Noam Rinetzky
  Safe Privatization for Transactional Memory
  In Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Vienna, Austria, 2018.

Chapter 2

A Generic Logic for Proving Linearizability

To manage the complexity of constructing concurrent software, programmers package often-used functionality into *libraries* of concurrent algorithms. These encapsulate data structures, such as queues and lists, and provide clients with a set of methods that can be called concurrently to operate on these (e.g., java.util.concurrent). To maximize performance, concurrent libraries may use sophisticated non-blocking techniques, allowing multiple threads to operate on the data structure with minimum synchronization. Despite this, each library method is usually expected to behave as though it executes atomically. This requirement is formalized by the standard notion of correctness for concurrent libraries, *linearizability* [7], which establishes a form of a simulation between the original *concrete* library and another *abstract* library, where each method is implemented atomically.

A common approach to proving linearizability is to find a *linearization point* for every method of the concrete library at which it can be thought of taking effect. [1] Given an execution of a concrete library, the matching execution of the abstract library, required to show the simulation, is constructed by executing the atomic abstract method at the linearization point of the concrete method. A difficulty in this approach is that linearization points are often not determined by a statically chosen point in the method code. For example, in concurrent algorithms with *helping* [9], a method may execute an operation originally requested by another method, called in a different thread; then the linearization point of the latter method is determined by an action of the former.

Recent years have seen a number of program logics for proving linearizability (see [23] for a survey). To avoid reasoning about the high number of possible interleavings between concurrently executing threads, these logics often use *thread-modular* reasoning. They establish protocols that threads should follow when operating on the shared data structure and reason separately about every thread, assuming that the rest follow the protocols. The logics for proving linearizability, such as [24, 25], usually borrow thread-modular reasoning rules from logics originally designed for proving non-relational properties of concur-

[1] Some algorithms cannot be reasoned about using linearization points, which we discuss in Section 2.6 and follow up on in Chapter 3.

rent programs, such as rely-guarantee [16], separation logic [15] or combinations thereof [24, 26]. Although this leads the logics to differ in technical details, they use similar methods for reasoning about linearizability, usually based on linearization points. Despite this similarity, designing a logic for proving linearizability that uses a particular thread-modular reasoning method currently requires finding the proof rules and proving their soundness afresh.

To consolidate this design space of linearization-point-based reasoning, we propose a logic for linearizability that is *generic*, i.e., can be instantiated with different means of thread-modular reasoning about concurrency, such as separation logic [15] or rely-guarantee [16]. To this end, we build on the recently-proposed Views framework [17], which unifies thread-modular logics for concurrency, such as the above-mentioned ones. Our contribution is to generalize the framework to reason about relations between programs, required for proving linearizability. In more detail, assertions in our logic are interpreted over a monoid of *relational views*, which describe relationships between the states of the concrete and the abstract libraries and the protocol that threads should follow in operating on these. The operation of the monoid, similar to the separating conjunction in separation logic [15], combines the assertions in different threads while ensuring that they agree on the protocols of access to the state. The choice of a particular relational view monoid thus determines the thread-modular reasoning method used by our logic.

To reason about linearization points, relational views additionally describe a set of special *tokens* (as in [24, 25, 27]), each denoting a one-time permission to execute a given atomic command on the state of the abstract library. The place where this permission is used in the proof of a concrete library method determines its linearization point, with the abstract command recorded by the token giving its specification. Crucially, reasoning about the tokens is subject to the protocols established by the underlying thread-modular reasoning method; in particular, their *ownership* can be transferred between different threads, which allows us to deal with helping.

We prove the soundness of our generic logic under certain conditions on its instantiations (Definition 2.2, §2.2). These conditions represent our key technical contribution, as they capture the essential requirements for soundly combining a given thread-modular method for reasoning about concurrency with the linearization-point method for reasoning about linearizability.

To illustrate the use of our logic, we present its example instantiations where thread-modular reasoning is done using disjoint concurrent separation logic [28] and a combination of separation logic and rely-guarantee [24]. We then apply the latter instantiation to prove the correctness of a sample concurrent algorithm with helping. We expect that our results will make it possible to systematically design logics using the plethora of other methods for thread-modular reasoning that have been shown to be expressible in the Views framework [28, 29, 30].

2.1 Methods Syntax and Sequential Semantics

We consider concurrent programs that consist of two components, which we call *libraries* and *clients*. Libraries provide clients with a set of methods, and clients call them concurrently. We distinguish *concrete* and *abstract* libraries, as the latter serve as specification for the former due to its methods being executed

$$\longrightarrow \subseteq \mathsf{Com} \times \mathsf{PCom} \times \mathsf{Com}$$

$$\frac{C_1 \rightarrowtail_\alpha C_1'}{C_1 \ ; \ C_2 \rightarrowtail_\alpha C_1' \ ; \ C_2} \quad \overline{C^\star \rightarrowtail_{\mathsf{id}} C; C^\star} \quad \overline{\alpha \rightarrowtail_\alpha \mathsf{skip}}$$

$$\frac{i \in \{1,2\}}{C_1 + C_2 \rightarrowtail_{\mathsf{id}} C_i} \qquad \overline{\mathsf{skip} \ ; \ C \rightarrowtail_{\mathsf{id}} C} \quad \overline{C^\star \rightarrowtail_{\mathsf{id}} \mathsf{skip}}$$

$$\longrightarrow\!\!\!\!\!\rightarrow \subseteq (\mathsf{Com} \times \mathsf{State}) \times (\mathsf{ThreadID} \times \mathsf{PCom}) \times (\mathsf{Com} \times \mathsf{State})$$

$$\frac{\sigma' \in [\![\alpha]\!]_t(\sigma) \quad C \rightarrowtail_\alpha C'}{\langle C, \sigma\rangle \xrightarrow{t,\alpha} \!\!\!\!\!\rightarrow \langle C', \sigma'\rangle}$$

Figure 2.1: The operational semantics of sequential commands

atomically.

Syntax. Concrete methods are implemented as *sequential commands* having the syntax:

$$C \in \mathsf{Com} ::= \alpha \mid C \ ; \ C \mid C + C \mid C^\star \mid \mathsf{skip}, \quad \text{where } \alpha \in \mathsf{PCom}$$

The grammar includes *primitive commands* α from a set PCom, sequential composition $C \ ; \ C$, non-deterministic choice $C + C$ and a finite iteration $C^\star$ (we are interested only in terminating executions) and a termination marker skip. We use $+$ and $(\cdot)^\star$ instead of conditionals and while loops for theoretical simplicity: as we show at the end of this section, given appropriate primitive commands the conditionals and loops can be encoded. We also assume a set APCom of *abstract primitive commands*, ranged over by A, with which we represent methods of an abstract library.

Semantics. We assume a set State of concrete states of the memory, ranged over by σ, and abstract states AState, ranged over by Σ. The memory is shared among N threads with thread identifiers $\mathsf{ThreadID} = \{1, 2, \ldots, N\}$, ranged over by t.

We assume that semantics of each primitive command α is given by a non-deterministic *state transformer* $[\![\alpha]\!]_t : \mathsf{State} \to \mathcal{P}(\mathsf{State})$, where $t \in \mathsf{ThreadID}$. For a state σ, the set of states $[\![\alpha]\!]_t(\sigma)$ is the set of possible resulting states for α executed atomically in a state σ and a thread t. State transformers may have different semantics depending on a thread identifier, which we use to introduce thread-local memory cells later in the technical development. Analogously, we assume semantics of abstract primitive commands with state transformers $[\![\mathrm{A}]\!]_t : \mathsf{AState} \to \mathcal{P}(\mathsf{AState})$, all of which update abstract states atomically. We also assume a primitive command $\mathsf{id} \in \mathsf{PCom}$ with the interpretation $[\![\mathsf{id}]\!]_t(\sigma) \triangleq \{\sigma\}$, and its abstract counterpart $\mathsf{id} \in \mathsf{APCom}$.

The sets of primitive commands PCom and APCom as well as corresponding state transformers are parameters of our framework. In Figure 2.1 we give rules of operational semantics of sequential commands, which are parametrized by semantics of primitive commands. That is, we define a transition relation $\longrightarrow\!\!\!\!\!\rightarrow \subseteq (\mathsf{Com} \times \mathsf{State}) \times (\mathsf{ThreadID} \times \mathsf{PCom}) \times (\mathsf{Com} \times \mathsf{State})$, so that $\langle C, \sigma\rangle \xrightarrow{t,\alpha} \!\!\!\!\!\rightarrow \langle C', \sigma'\rangle$ indicates a transition from C to C' updating the state from σ to σ' with a primitive command α in a thread t. The rules of the operational semantics are standard.

Let us show how to define traditional control flow primitives, such as an if-statement and a while-loop, in our programming language. Assuming a language for arithmetic expressions, ranged over by E, and a function $[\![E]\!]_\sigma$ that evaluates expressions in a given state σ, we define a primitive command $\texttt{assume}(E)$ that acts as a filter on states, choosing only those where E evaluates to non-zero values.

$$[\![\texttt{assume}(E)]\!]_t(\sigma) \triangleq \text{ if } [\![E]\!]_\sigma \neq 0 \text{ then } \{\sigma\} \text{ else } \emptyset.$$

Using $\texttt{assume}(E)$ and the C-style negation $!E$ in expressions, a conditional and a while-loop can be implemented as the following commands:

$$\texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2 \triangleq (\texttt{assume}(E); C_1) + (\texttt{assume}(!E); C_2)$$
$$\texttt{while } E \texttt{ do } C \triangleq (\texttt{assume}(E); C)^\star; \texttt{assume}(!E)$$

2.2 The Generic Logic

In this section, we present our framework for designing program logics for linearizability proofs. Given a concrete method and a corresponding abstract method, we aim to demonstrate that the former has a linearization point either within its code or in the code of another thread. The idea behind such proofs is to establish simulation between concrete and abstract methods using linearization points to determine when the abstract method has to make a transition to match a given execution of the concrete method. To facilitate such simulation-based proofs, we design our relational logic so that formulas in it denote relations between concrete states, abstract states and special *tokens*.

Tokens are our tool for reasoning about linearization points. At the beginning of its execution in a thread t, each concrete method m is given a token $\mathsf{todo}(\mathrm{A}_m)$ of the corresponding abstract primitive command A_m. The token represents a one-time permission for the method to take effect, i.e. to perform a primitive command A_m on an abstract machine. When the permission is used, a token $\mathsf{todo}(\mathrm{A}_m)$ in a thread t is irreversibly replaced with $\mathsf{done}(\mathrm{A}_m)$. Thus, by requiring that a method start its execution in a thread t with a token $\mathsf{todo}(\mathrm{A}_m)$ and ends with $\mathsf{done}(\mathrm{A}_m)$, we ensure that it has in its code a linearization point. The tokens of all threads are described by $\Delta \in \mathsf{Tokens}$:

$$\mathsf{Tokens} = \mathsf{ThreadID} \rightharpoonup (\{\mathsf{todo}(\mathrm{A}) \mid \mathrm{A} \in \mathsf{APCom}\} \cup \{\mathsf{done}(\mathrm{A}) \mid \mathrm{A} \in \mathsf{APCom}\})$$

Reasoning about states and tokens in the framework is done with the help of relational *views*. We assume a set Views, ranged over by p, q and r, as well as a reification function $\lfloor\ \rfloor : \mathsf{Views} \to \mathcal{P}(\mathsf{State} \times \mathsf{AState} \times \mathsf{Tokens})$ that interprets views as ternary relations on concrete states, abstract states and indexed sets of tokens.

Definition 2.1. *A relational view monoid is a commutative monoid* $(\mathsf{Views}, *, u)$*, where* Views *is an underlying set of relational views,* $*$ *is a monoid operation and* u *is a unit.*

The monoid structure of relational views allows treating them as restrictions on the environment of threads. Intuitively, each thread uses views to declare a protocol that other threads should follow while operating with concrete states,

abstract states and tokens. Similarly to the separating conjunction from separation logic, the monoid operation $*$ (view composition) applied to a pair of views combines protocols of access to the state and ensures that they do not contradict each other.

Disjoint Concurrent Separation logic. To give an example of a view monoid, we demonstrate the structure inspired by Disjoint Concurrent Separation logic (DCSL). A distinctive feature of DCSL is that its assertions enforce a protocol, according to which threads operate on disjoint pieces of memory. We assume a set of values Val, of which a subset $\mathsf{Loc} \subseteq \mathsf{Val}$ represents heap addresses. By letting $\mathsf{State} = \mathsf{AState} = (\mathsf{Loc} \rightharpoonup_{\mathrm{fin}} \mathsf{Val}) \cup \{\lightning\}$ we represent a state as either a finite partial function from locations to values or an exceptional faulting state $\lightning$, which denotes the result of an invalid memory access. We define an operation $\bullet$ on states, which results in $\lightning$ if either of the operands is $\lightning$, or the union of partial functions if their domains are disjoint. Finally, we assume that the set PCom consists of standard heap-manipulating commands with usual semantics [15, 17].

We consider the following view monoid:

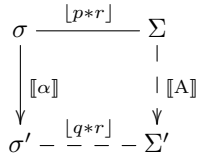
$$(\mathcal{P}((\mathsf{State} \setminus \{\lightning\}) \times (\mathsf{AState} \setminus \{\lightning\}) \times \mathsf{Tokens}), *_{\mathsf{SL}}, ([\,], [\,], [\,])),$$

where the unit is a triple of nowhere defined functions $[\,]$, and the view composition defined as follows:

$$p *_{\mathsf{SL}} p' \triangleq \{(\sigma \bullet \sigma', \Sigma \bullet \Sigma', \Delta \uplus \Delta') \mid (\sigma, \Sigma, \Delta) \in p \wedge (\sigma', \Sigma', \Delta') \in p'\}.$$

In this monoid, the composition enforces a protocol of exclusive ownership of parts of the heap: a pair of views can be composed only if they do not simultaneously describe the content of the same heap cell or a token. Since tokens are exclusively owned in DCSL, they cannot be accessed by other threads, which makes it impossible to express a helping mechanism with the DCSL views. In §2.4, we present another instance of our framework and reason about helping in it.

Reasoning about linearization points. We now introduce *action judgments*, which formalize linearization-points-based approach to proving linearizability within our framework.



Let us assume that α is executed in a concrete state σ with an abstract state Σ and a set of tokens Δ satisfying a precondition p. According to the action judgment $\alpha \Vdash_t \{p\}\{q\}$, for every update $\sigma' \in [\![\alpha]\!]_t(\sigma)$ of the concrete state, the abstract state may be changed to $\Sigma' \in [\![\mathrm{A}]\!]_{t'}(\Sigma)$ in order to satisfy the postcondition q, provided that there is a token $\mathsf{todo}(\mathrm{A})$ in a thread t'. When the abstract state Σ is changed and the token $\mathsf{todo}(\mathrm{A})$ of a thread t' is used, the concrete state update corresponds to a linearization point, or to a regular transition otherwise.

Definition 2.2. *The action judgment $\alpha \Vdash_t \{p\}\{q\}$ holds, iff the following is true:*

$$\forall r, \sigma, \sigma', \Sigma, \Delta.\, (\sigma, \Sigma, \Delta) \in \lfloor p * r \rfloor \wedge \sigma' \in [\![\alpha]\!]_t(\sigma) \implies \exists \Sigma', \Delta'.\, \mathsf{LP}^*(\Sigma, \Delta, \Sigma', \Delta') \wedge (\sigma', \Sigma', \Delta') \in \lfloor q * r \rfloor,$$

where LP^* *is the transitive closure of the following relation:*

$$\mathsf{LP}(\Sigma, \Delta, \Sigma', \Delta') \triangleq \exists t', \mathrm{A}.\, \Sigma' \in [\![\mathrm{A}]\!]_{t'}(\Sigma) \wedge \Delta(t') = \mathsf{todo}(\mathrm{A}) \wedge \Delta' = \Delta[t' : \mathsf{done}(\mathrm{A})],$$

and $f[x : a]$ *denotes the function such that* $f[x : a](x) = a$ *and for any* $y \neq x$, $f[x : a](y) = f(y)$.

Note that depending on pre- and postconditions p and q, $\alpha \Vdash_t \{p\}\{q\}$ may encode a regular transition, a conditional or a standard linearization point. It is easy to see that the latter is the case only when in all sets of tokens Δ from $\lfloor p \rfloor$ some thread t' has a todo-token, and in all Δ' from $\lfloor q \rfloor$ it has a done-token. Additionally, the action judgment may represent a conditional linearization point of another thread, as the LP relation allows using tokens of other threads.

Action judgments have a closure property that is important for thread-modular reasoning: when $\alpha \Vdash_t \{p\}\{q\}$ holds, so does $\alpha \Vdash_t \{p * r\}\{q * r\}$ for every view r. That is, execution of α and a corresponding linearization point preserves every view r that p can be composed with. Consequently, when in every thread action judgments hold of primitive commands and thread's views, all threads together mutually agree on each other's protocols of the access to the shared memory encoded in their views. This enables reasoning about every thread in isolation with the assumption that its environment follows its protocol. Thus, the action judgments formalize the requirements that instances of our framework need to satisfy in order to be sound. In this regard action judgments are inspired by semantic judgments of the Views Framework [17]. Our technical contribution is in formulating the essential requirements for thread-modular reasoning about linearizability of concurrent libraries with the linearization-point method and in extending the semantic judgment with them.

We let a *repartitioning implication* of views p and q, written $p \Rrightarrow q$, denote the following:

$$p \Rrightarrow q \triangleq \forall r.\, \lfloor p * r \rfloor \subseteq \lfloor q * r \rfloor. \tag{2.3}$$

A repartitioning implication $p \Rrightarrow q$ ensures that states satisfying p also satisfy q and additionally requires this property to preserve any view r.

Program logic. We are now in a position to present our generic logic for linearizability proofs via the linearization-point method. Assuming a view monoid and reification function as parameters, we define a minimal language Assn for assertions $\mathcal{P}$ and $\mathcal{Q}$ denoting sets of views:

$$\mathcal{P}, \mathcal{Q} \in \mathsf{Assn} ::= \rho \mid \mathcal{P} * \mathcal{Q} \mid \mathcal{P} \vee \mathcal{Q} \mid \mathcal{P} \Rrightarrow \mathcal{Q} \mid \exists X.\, \mathcal{P} \mid \ldots$$

The grammar includes view assertions ρ, a syntax VAssn of which is a parameter of the framework. Formulas of Assn may contain the standard connectives from separation logic, the repartitioning implication and the existential quantification over logical variables X, ranging over a set LVar.

Let us assume an interpretation of logical variables $\ell \in \mathsf{Int} = \mathsf{LVar} \to \mathsf{Val}$ that maps logical variables from LVar to values from a finite set Val. In Figure 2.2, we define a function $[\![\cdot]\!]. : \mathsf{Assn} \times \mathsf{Int} \to \mathsf{Views}$ that we use to interpret assertions. Interpretation of assertions is parametrized by $[\![\cdot]\!]. : \mathsf{VAssn} \times \mathsf{Int} \to \mathsf{Views}$. In

$$[\![\mathcal{P} * \mathcal{Q}]\!]_\ell = [\![\mathcal{P}]\!]_\ell * [\![\mathcal{Q}]\!]_\ell \qquad [\![\mathcal{P} \Rrightarrow \mathcal{Q}]\!]_\ell = [\![\mathcal{P}]\!]_\ell \Rrightarrow [\![\mathcal{Q}]\!]_\ell$$
$$[\![\mathcal{P} \vee \mathcal{Q}]\!]_\ell = [\![\mathcal{P}]\!]_\ell \vee [\![\mathcal{Q}]\!]_\ell \qquad [\![\exists X.\, \mathcal{P}]\!]_\ell = \bigvee_{n \in \mathsf{Val}} [\![\mathcal{P}]\!]_{\ell[X:n]}$$

Figure 2.2: Satisfaction relation for the assertion language Assn

$$\text{(Prim)}\quad \frac{\forall \ell.\ \alpha \Vdash_t \{[\![\mathcal{P}]\!]_\ell\}\{[\![\mathcal{Q}]\!]_\ell\}}{\vdash_t \{\mathcal{P}\}\ \alpha\ \{\mathcal{Q}\}}$$

$$\text{(Seq)}\quad \frac{\vdash_t \{\mathcal{P}\}\ C_1\ \{\mathcal{P}'\} \quad \vdash_t \{\mathcal{P}'\}\ C_2\ \{\mathcal{Q}\}}{\vdash_t \{\mathcal{P}\}\ C_1\ ;\ C_2\ \{\mathcal{Q}\}}$$

$$\text{(Frame)}\quad \frac{\vdash_t \{\mathcal{P}\}\ C\ \{\mathcal{Q}\}}{\vdash_t \{\mathcal{P} * \mathcal{R}\}\ C\ \{\mathcal{Q} * \mathcal{R}\}}$$

$$\text{(Disj)}\quad \frac{\vdash_t \{\mathcal{P}_1\}\ C\ \{\mathcal{Q}_1\} \quad \vdash_t \{\mathcal{P}_2\}\ C\ \{\mathcal{Q}_2\}}{\vdash_t \{\mathcal{P}_1 \vee \mathcal{P}_2\}\ C\ \{\mathcal{Q}_1 \vee \mathcal{Q}_2\}}$$

$$\text{(Ex)}\quad \frac{\vdash_t \{\mathcal{P}\}\ C\ \{\mathcal{Q}\}}{\vdash_t \{\exists X.\, \mathcal{P}\}\ C\ \{\exists X.\, \mathcal{Q}\}}$$

$$\text{(Choice)}\quad \frac{\vdash_t \{\mathcal{P}\}\ C_1\ \{\mathcal{Q}\} \quad \vdash_t \{\mathcal{P}\}\ C_2\ \{\mathcal{Q}\}}{\vdash_t \{\mathcal{P}\}\ C_1 + C_2\ \{\mathcal{Q}\}}$$

$$\text{(Iter)}\quad \frac{\vdash_t \{\mathcal{P}\}\ C\ \{\mathcal{P}\}}{\vdash_t \{\mathcal{P}\}\ C^\star\ \{\mathcal{P}\}}$$

$$\text{(Conseq)}\quad \frac{\mathcal{P}' \Rrightarrow \mathcal{P} \quad \vdash_t \{\mathcal{P}\}\ C\ \{\mathcal{Q}\} \quad \mathcal{Q} \Rrightarrow \mathcal{Q}'}{\vdash_t \{\mathcal{P}'\}\ C\ \{\mathcal{Q}'\}}$$

Figure 2.3: Proof rules

order to interpret disjunction, we introduce a corresponding operation on views and require the following properties from it:

$$\lfloor p \vee q \rfloor = \lfloor p \rfloor \cup \lfloor q \rfloor \qquad (p \vee q) * r = (p * r) \vee (q * r) \tag{2.4}$$

The judgments of the program logic take the form $\vdash_t \{\mathcal{P}\}\ C\ \{\mathcal{Q}\}$. In Figure 2.3, we present the proof rules, which are mostly standard. Among them, the Prim rule is noteworthy, since it incorporates the simulation-based approach to reasoning about linearization points introduced by action judgments. The Frame rule applies the idea of local reasoning from separation logic [15] to views. The Conseq enables weakening a precondition or a postcondition in a proof judgment and uses repartitioning implications to ensure the thread-modularity of the weakened proof judgment.

Semantics of proof judgments. We give semantics to judgments of the program logic by lifting the requirements of action judgments to sequential commands.

Definition 2.5 (Safety judgment)**.** *We define* safe_t *as the greatest relation such that the following holds whenever* $\mathsf{safe}_t(p, C, q)$ *does:*

- *if* $C \neq \mathsf{skip}$*, then* $\forall C', \alpha.\, C \rightarrowtail_\alpha C' \implies \exists p'.\ \alpha \Vdash_t \{p\}\{p'\} \wedge \mathsf{safe}_t(p', C', q)$,
- *if* $C = \mathsf{skip}$*, then* $p \Rrightarrow q$.

Lemma 2.6. $\forall t, \mathcal{P}, C, \mathcal{Q}. \vdash_t \{\mathcal{P}\}\ C\ \{\mathcal{Q}\} \implies \forall \ell.\, \mathsf{safe}_t([\![\mathcal{P}]\!]_\ell, C, [\![\mathcal{Q}]\!]_\ell)$.

We can understand the safety judgment $\mathsf{safe}_t(\llbracket\mathcal{P}\rrbracket_\ell, C, \llbracket\mathcal{Q}\rrbracket_\ell)$ as an obligation to create a sequence of views $\llbracket\mathcal{P}\rrbracket_\ell = p_1, p_2, \ldots, p_{n+1} = \llbracket\mathcal{Q}\rrbracket_\ell$ for each finite trace $\alpha_1, \alpha_2, \ldots, \alpha_n$ of C to justify each transition with action judgments $\alpha_1 \Vdash_t \{p_1\}\{p_2\}$, $\ldots$, $\alpha_n \Vdash_t \{p_n\}\{p_{n+1}\}$. Thus, when $\mathsf{safe}_t(\llbracket\mathcal{P}\rrbracket_\ell, C, \llbracket\mathcal{Q}\rrbracket_\ell)$ holds, it ensures that every step of the machine correctly preserves a correspondence between a concrete and abstract execution. Intuitively, the safety judgment lifts the simulation between concrete and abstract primitive commands established with action judgments to the implementation and specification of a method.

In Lemma 2.6, we establish that the proof judgments of the logic imply the safety judgments. As a part of the proof, we show that each of the proof rules of the logic holds of safety judgments. We summarize this observation in the following auxiliary lemma.

Lemma 2.7. *The safety relation* safe_t *has the following closure properties:*

$$\textsc{Seq}\text{:}\ \forall t, C_1, C_2, p, p', q.\ \mathsf{safe}_t(p, C_1, p') \land \mathsf{safe}_t(p', C_2, q) \implies \mathsf{safe}_t(p, C_1\ ;\ C_2, q);$$

$$\textsc{Frame}\text{:}\ \forall t, C, p, q, r.\ \mathsf{safe}_t(p, C, q) \implies \mathsf{safe}_t(p * r, C, q * r);$$

$$\textsc{Disj}\text{:}\ \forall t, C, p_1, p_2, q_1, q_2.\ \mathsf{safe}_t(p_1, C, q_1) \land \mathsf{safe}_t(p_2, C, q_2) \implies \mathsf{safe}_t(p_1 \lor p_2, C, q_1 \lor q_2).$$

$$\textsc{Choice}\text{:}\ \forall t, C_1, C_2, p, q.\ \mathsf{safe}_t(p, C_1, q) \land \mathsf{safe}_t(p, C_2, q) \implies \mathsf{safe}_t(p, C_1 + C_2, q);$$

$$\textsc{Iter}\text{:}\ \forall t, C, p.\ \mathsf{safe}_t(p, C, p) \implies \mathsf{safe}_t(p, C^\star, p);$$

$$\textsc{Conseq}\text{:}\ \forall t, C, p, p', q, q'.\ p' \Rrightarrow p \land \mathsf{safe}_t(p, C, q) \land q \Rrightarrow q' \implies \mathsf{safe}_t(p', C, q');$$

For convenience of presentation, we first show how Lemma 2.7 is used in the the proof of Lemma 2.6, and then prove Lemma 2.7.

Proof of Lemma 2.6. We prove the lemma by rule induction. For that we choose arbitrary thread identifier t and demonstrate that $\forall \ell.\, \mathsf{safe}_t(\llbracket\mathcal{P}\rrbracket_\ell, C, \llbracket\mathcal{Q}\rrbracket_\ell)$ is closed under the proof rules from Figure 2.3. The cases of Choice, Iter, Seq, Conseq, Frame and Disj rules are straightforward: they trivially follow from Lemma 2.7 after using the properties of $\llbracket - \rrbracket_\ell$ from Figure 2.2. The Ex rule uses the fact that Val, which is the range of i, is finite, which makes possible proving it just like the Disj rule.

It remains to consider the Prim rule to conclude Lemma 2.6. Let us assume that $\forall \ell'.\ \alpha \Vdash_t \{\llbracket\mathcal{P}\rrbracket_{\ell'}\}\{\llbracket\mathcal{Q}\rrbracket_{\ell'}\}$ holds. We need to demonstrate that so does $\forall \ell.\, \mathsf{safe}_t(\llbracket\mathcal{P}\rrbracket_\ell, \alpha, \llbracket\mathcal{Q}\rrbracket_\ell)$. To conclude that the latter holds, according Definition 2.5 we need to prove the following for every ℓ:

$$\forall C', \alpha'.\, \alpha \rightarrowtail_{\alpha'} C' \implies \exists p'.\ \alpha' \Vdash_t \{\llbracket\mathcal{P}\rrbracket_\ell\}\{p'\} \land \mathsf{safe}_t(p', C', \llbracket\mathcal{Q}\rrbracket_\ell) \tag{2.8}$$

According to the operational semantics from Figure 2.1, the only transition from a command α is $\alpha \rightarrowtail_\alpha \mathsf{skip}$. Thus, in the formula above $\alpha' = \alpha$ and $C' = \mathsf{skip}$. Note that $\mathsf{safe}_t(\llbracket\mathcal{Q}\rrbracket_\ell, \mathsf{skip}, \llbracket\mathcal{Q}\rrbracket_\ell)$ holds trivially. Additionally, by our assumption, $\alpha \Vdash_t \{\llbracket\mathcal{P}\rrbracket_{\ell'}\}\{\llbracket\mathcal{Q}\rrbracket_{\ell'}\}$ holds for any ℓ'. Consequently, it holds for $\ell' = \ell$. We conclude that by letting $p' = \llbracket\mathcal{Q}\rrbracket_\ell$ we satisfy (2.8). □

2.2.1 Proof of Lemma 2.7

We now prove the compositionality properties of the safety relation. We prove all of them by coinduction. To this end, we restate the definition of safe_t using the fixed-point notation. We consider a function

$$F_t : \mathcal{P}(\mathsf{Views} \times \mathsf{Com} \times \mathsf{Views}) \to \mathcal{P}(\mathsf{Views} \times \mathsf{Com} \times \mathsf{Views})$$

defined as follows:

$$\begin{aligned} F_t(X) \triangleq \{(p, \mathsf{skip}, q) \mid p \Rrightarrow q\} \cup \\ \{(p, C, q) \mid \forall C', \alpha.\, C \rightarrowtail_\alpha C' \implies \exists p'.\, \alpha \Vdash_t \{p\}\{p'\} \wedge (p', C', q) \in X\} \end{aligned}$$

Note that a powerset domain ordered by inclusion is a complete lattice and F is a mapping on it, which means that F is monotone. Consequently, by Knaster-Tarski fixed-point theorem F has the greatest fixed-point. It is easy to see that $\mathsf{safe}_t \triangleq \mathsf{gfp}\, F_t$ in Definition 2.5.

In the proof of Lemma 2.7 we use the following properties of the action judgment and the $\Rrightarrow$ relation.

Proposition 2.9 (Locality)**.**

$$\begin{aligned} &\forall p, q, r.\, p \Rrightarrow q \implies p * r \Rrightarrow q * r; \\ &\forall t, \alpha, p, q, r.\, \alpha \Vdash_t \{p\}\{q\} \implies \alpha \Vdash_t \{p * r\}\{q * r\}. \end{aligned}$$

Proposition 2.10 (Consequence)**.**

$$\forall t, \alpha, p, q, p', q'.\, p' \Rrightarrow p \wedge \alpha \Vdash_t \{p\}\{q\} \wedge q \Rrightarrow q' \implies \alpha \Vdash_t \{p'\}\{q'\}.$$

The proofs of Propositions 2.9 and 2.10 are straightforward: both properties can be easily checked after unfolding definitions of action judgments.

Proposition 2.11 (Distributivity)**.**

$$\forall t, \alpha, p_1, p_2, q_1, q_2.\, \alpha \Vdash_t \{p_1\}\{q_1\} \wedge \alpha \Vdash_t \{p_2\}\{q_2\} \implies \alpha \Vdash_t \{p_1 \vee p_2\}\{q_1 \vee q_2\}.$$

Proof. According to the Definition 2.2 of the action judgment $\alpha \Vdash_t \{p_1 \vee p_2\}\{q_1 \vee q_2\}$, in order to prove the latter we need to demonstrate the following:

$$\begin{aligned} \forall r, \sigma, \sigma', \Sigma, \Delta.\, \sigma' \in [\![\alpha]\!]_t(\sigma) \wedge (\sigma, \Sigma, \Delta) \in \lfloor (p_1 \vee p_2) * r \rfloor \implies \\ \exists \Sigma', \Delta'.\, \mathsf{LP}^*(\Sigma, \Delta, \Sigma', \Delta') \wedge (\sigma', \Sigma', \Delta') \in \lfloor (q_1 \vee q_2) * r \rfloor. \end{aligned} \quad (2.12)$$

Let us consider any view r, states σ, σ', Σ and tokens Δ such that both $\sigma' \in [\![\alpha]\!]_t(\sigma)$ and $(\sigma, \Sigma, \Delta) \in \lfloor (p_1 \vee p_2) * r \rfloor$ hold. According to the properties of disjunction stated in equalities (2.4),

$$\lfloor (p_1 \vee p_2) * r \rfloor = \lfloor (p_1 * r) \vee (p_2 * r) \rfloor = \lfloor p_1 * r \rfloor \cup \lfloor p_2 * r \rfloor.$$

Consequently, $(\sigma, \Sigma, \Delta) \in \lfloor p_1 * r \rfloor \cup \lfloor p_2 * r \rfloor$.

Let us assume that $(\sigma, \Sigma, \Delta) \in \lfloor p_1 * r \rfloor$ (the other case is analogous). Then according to the action judgment $\alpha \Vdash_t \{p_1\}\{q_1\}$, there exist Σ' and Δ' such that:

$$\mathsf{LP}^*(\Sigma, \Delta, \Sigma', \Delta') \wedge (\sigma', \Sigma', \Delta') \in \lfloor q_1 * r \rfloor. \quad (2.13)$$

Once again, according to the properties (2.4) of disjunction:

$$\lfloor q_1 * r \rfloor \subseteq \lfloor q_1 * r \rfloor \cup \lfloor q_2 * r \rfloor = \lfloor (q_1 * r) \vee (q_2 * r) \rfloor = \lfloor (q_1 \vee q_2) * r \rfloor,$$

which together with (2.13) means that $(\sigma', \Sigma', \Delta') \in \lfloor (q_1 \vee q_2) * r \rfloor$. Overall we have shown that there exist Σ' and Δ' such that $\mathsf{LP}^*(\Sigma, \Delta, \Sigma', \Delta')$ and $(\sigma', \Sigma', \Delta') \in \lfloor (q_1 \vee q_2) * r \rfloor$, which concludes the proof of (2.12). □

We now prove the closure properties from Lemma 2.7.

Proof of SEQ**.** Let ϕ be defined as follows:

$$\phi(X) \triangleq \{(p, C_1 \ ; \ C_2, q) \mid \exists q'.\, (p, C_1, q') \in X \wedge (q', C_2, q) \in X\}.$$

Then our goal is to prove that $\phi(\mathsf{safe}_t) \subseteq \mathsf{safe}_t$. For convenience, we prove an equivalent inequality $\phi(\mathsf{safe}_t) \cup \mathsf{safe}_t \subseteq \mathsf{safe}_t$ instead.

Since $\mathsf{safe}_t = \mathsf{gfp}\, F_t$, we can do a proof by coinduction: to conclude that $\phi(\mathsf{safe}_t) \cup \mathsf{safe}_t \subseteq \mathsf{safe}_t$ holds, we demonstrate:

$$\phi(\mathsf{safe}_t) \cup \mathsf{safe}_t \subseteq F_t(\phi(\mathsf{safe}_t) \cup \mathsf{safe}_t).$$

Let us consider any $(p, C, q) \in \mathsf{safe}_t$. Since $\mathsf{safe}_t = \mathsf{gfp}\, F_t$, we know that $\mathsf{safe}_t = F_t(\mathsf{safe}_t)$ holds. Then by monotonicity of F_t, $(p, C, q) \in F_t(\mathsf{safe}_t) \subseteq F_t(\phi(\mathsf{safe}_t) \cup \mathsf{safe}_t)$.

Now let us consider any $(p, C, q) \in \phi(\mathsf{safe}_t)$. There necessarily are C_1, C_2 and q' such that:

$$C = C_1 \ ; \ C_2 \wedge (p, C_1, q') \in \mathsf{safe}_t \wedge (q', C_2, q) \in \mathsf{safe}_t. \tag{2.14}$$

For $(p, C_1 \ ; \ C_2, q)$ to belong to $F_t(\phi(\mathsf{safe}_t) \cup \mathsf{safe}_t)$, the following has to be the case for every transition $C_1 \ ; \ C_2 \rightarrowtail_\alpha C'$:

$$\exists p'.\, \alpha \Vdash_t \{p\}\{p'\} \wedge (p', C', q) \in \phi(\mathsf{safe}_t) \cup \mathsf{safe}_t. \tag{2.15}$$

According to the rules of the operational semantics (Figure 2.1), when there is a transition $C_1 \ ; \ C_2 \rightarrowtail_\alpha C'$, either of the following is true:

- there exists C_1' such that $C' = C_1' \ ; \ C_2$ and $C_1 \rightarrowtail_\alpha C_1'$; or
- $C_1 = \mathsf{skip}$, $\alpha = \mathsf{id}$ and $C' = C_2$.

Let us assume that the former is the case. From (2.14) we know that $(p, C_1, q') \in \mathsf{safe}_t$, which means that the following holds of $C_1 \rightarrowtail_\alpha C_1'$:

$$\exists p''.\, \alpha \Vdash_t \{p\}\{p''\} \wedge (p'', \alpha_1', q') \in \mathsf{safe}_t.$$

When $(p'', \alpha_1', q') \in \mathsf{safe}_t$ and $(q', \alpha_2, q) \in \mathsf{safe}_t$, it is the case that $(p'', \alpha_1' \ ; \ \alpha_2, q) \in \phi(\mathsf{safe}_t)$. Thus, by letting $p' = p''$ we satisfy (2.15).

We now consider the case when $C_1 = \mathsf{skip}$, $\alpha = \mathsf{id}$ and $C' = C_2$. From (2.14) we know that $(p, \mathsf{skip}, q') \in \mathsf{safe}_t$, meaning that $p \Rrightarrow q'$, or, equivalently, $\mathsf{id} \Vdash_t \{p\}\{q'\}$. We also know from (2.14) that $(q', C_2, q) \in \mathsf{safe}_t$. Thus, (2.15) can be satisfied by letting $p' = q'$. □

Proof of FRAME**.** Let us define an auxiliary function:

$$\phi(X, r) \triangleq \{(p * r, C, q * r) \mid (p, C, q) \in X\}.$$

Then our goal is to prove that $\phi(\mathsf{safe}_t, r) \subseteq \mathsf{safe}_t$. Since $\mathsf{safe}_t = \mathsf{gfp}\, F_t$, we can do a proof by coinduction: to conclude that $\phi(\mathsf{safe}_t, r) \subseteq \mathsf{safe}_t$ holds, we demonstrate $\phi(\mathsf{safe}_t, r) \subseteq F_t(\phi(\mathsf{safe}_t, r))$.

Consider any $(p', C, q') \in \phi(\mathsf{safe}_t, r)$. There necessarily are p, q, r such that $p' = p * r$, $q' = q * r$ and $(p, C, q) \in \mathsf{safe}_t$. Let us assume that $C = \mathsf{skip}$. Then $(p, C, q) \in \mathsf{safe}_t$ implies that $p \Rrightarrow q$. By Proposition 2.9, $p * r \Rrightarrow q * r$, which implies $(p * r, \mathsf{skip}, q * r) \in \mathsf{safe}_t$ to hold.

Now let $C \neq \mathsf{skip}$. Since $(p, C, q) \in \mathsf{safe}_t$, by definition of the safety relation the following holds of every α, C' and any transition $C \rightarrowtail_\alpha C'$:

$$\exists p'.\ \alpha \Vdash_t \{p\}\{p'\} \wedge (p', C', q) \in \mathsf{safe}_t.$$

By Proposition 2.9, $\alpha \Vdash_t \{p\}\{p'\}$ implies $\alpha \Vdash_t \{p * r\}\{p' * r\}$. Also, when $(p', C', q) \in \mathsf{safe}_t$, it is the case that $(p' * r, C', q * r) \in \phi(\mathsf{safe}_t, r)$. Thus, we have shown for every transition $C \rightarrowtail_\alpha C'$ that there exists $p'' = p' * r$ such that $\alpha \Vdash_t \{p * r\}\{p''\}$ and $(p'', C', q * r) \in \phi(\mathsf{safe}_t, r)$:

$$\forall \alpha, C.\, C \rightarrowtail_\alpha C' \implies \exists p''.\ \alpha \Vdash_t \{p * r\}\{p''\} \wedge (p'', C', q * r) \in \phi(\mathsf{safe}_t, r),$$

which is sufficient to conclude that $(p * r, C, q * r) \in F_t(\phi(\mathsf{safe}_t, r))$. □

Proof of DISJ**.** Let ϕ be defined as follows:

$$\phi(X) \triangleq \{(p_1 \vee p_2, C, q_1 \vee q_2) \mid (p_1, C, q_1) \in X \wedge (p_2, C, q_2) \in X\}.$$

Then our goal is to prove that $\phi(\mathsf{safe}_t) \subseteq \mathsf{safe}_t$. Since $\mathsf{safe}_t = \mathsf{gfp}\, F_t$, we can do a proof by coinduction: to conclude that $\phi(\mathsf{safe}_t) \subseteq \mathsf{safe}_t$ holds, we demonstrate $\phi(\mathsf{safe}_t) \subseteq F_t(\phi(\mathsf{safe}_t))$.

Let us consider $(p, C, q) \in \phi(\mathsf{safe}_t)$. Then there necessarily are p_1, q_1, p_2 and q_2 such that $p = p_1 \vee p_2$, $q = q_1 \vee q_2$, and $(p_1, C, q_1), (p_2, C, q_2) \in \mathsf{safe}_t$. From the latter we get that for any α, C' and a transition $C \rightarrowtail_\alpha C'$ the following holds:

$$\exists p_1'.\ \alpha \Vdash_t \{p_1\}\{p_1'\} \wedge (p_1', C', q_1) \in \mathsf{safe}_t;$$
$$\exists p_2'.\ \alpha \Vdash_t \{p_2\}\{p_2'\} \wedge (p_2', C', q_2) \in \mathsf{safe}_t.$$

Then it is the case that $(p_1' \vee p_2', C', q_1 \vee q_2) \in \phi(\mathsf{safe}_t)$. Moreover, $\alpha \Vdash_t \{p_1 \vee p_2\}\{p_1' \vee p_2'\}$ holds by Proposition 2.11. Thus, we have shown for every transition $C \rightarrowtail_\alpha C'$ that there exists $p' = p_1' \vee p_2'$ such that $\alpha \Vdash_t \{p_1 \vee p_2\}\{p'\}$ and $(p', C', q_1 \vee q_2) \in \phi(\mathsf{safe}_t)$:

$$\forall \alpha, C.\, C \rightarrowtail_\alpha C' \implies \exists p'.\ \alpha \Vdash_t \{p_1 \vee p_2\}\{p'\} \wedge (p', C', q_1 \vee q_2) \in \phi(\mathsf{safe}_t).$$

which is sufficient to conclude that $(p_1 \vee p_2, C, q_1 \vee q_2) \in F_t(\phi(\mathsf{safe}_t))$. □

Proof of CHOICE**.** Let us define an auxiliary function:

$$\phi(X, Y) \triangleq \{(p, C_1 + C_2, q) \mid (p, C_1, q) \in X \wedge (p, C_2, q) \in Y\}.$$

Then our goal is to prove that $\phi(\mathsf{safe}_t, \mathsf{safe}_t) \subseteq \mathsf{safe}_t$. For convenience, we prove an equivalent inequality $\phi(\mathsf{safe}_t, \mathsf{safe}_t) \cup \mathsf{safe}_t \subseteq \mathsf{safe}_t$ instead.

Since $\mathsf{safe}_t = \mathsf{gfp}\, F_t$, we can do a proof by coinduction: to conclude that $\phi(\mathsf{safe}_t, \mathsf{safe}_t) \cup \mathsf{safe}_t \subseteq \mathsf{safe}_t$ holds, we demonstrate $\phi(\mathsf{safe}_t, \mathsf{safe}_t) \cup \mathsf{safe}_t \subseteq F_t(\phi(\mathsf{safe}_t, \mathsf{safe}_t) \cup \mathsf{safe}_t)$.

Let us consider $(p, C, q) \in \mathsf{safe}_t$. Since $\mathsf{safe}_t = \mathsf{gfp}\, F_t$, we know that $\mathsf{safe}_t = F_t(\mathsf{safe}_t)$ holds. Then by monotonicity of F_t, $(p, C, q) \in F_t(\mathsf{safe}_t) \subseteq F_t(\phi(\mathsf{safe}_t, \mathsf{safe}_t) \cup \mathsf{safe}_t)$.

Now let us consider $(p, C, q) \in \phi(\mathsf{safe}_t, \mathsf{safe}_t)$. There necessarily are C_1 and C_2 such that $C = C_1 + C_2$, $(p, C_1, q) \in \mathsf{safe}_t$, and $(p, C_2, q) \in \mathsf{safe}_t$. For $(p, C_1 + C_2, q)$ to belong to $F_t(\phi(\mathsf{safe}_t, \mathsf{safe}_t) \cup \mathsf{safe}_t)$, the following has to be proven for every transition $C_1 + C_2 \rightarrowtail_\alpha C'$:

$$\exists p'.\ \alpha \Vdash_t \{p\}\{p'\} \wedge (p', C', q) \in \phi(\mathsf{safe}_t, \mathsf{safe}_t) \cup \mathsf{safe}_t. \tag{2.16}$$

According to the rules of the operational semantics (Figure 2.1), whenever $C_1 + C_2 \rightarrowtail_\alpha C'$, necessarily $\alpha = \mathsf{id}$ and either $C' = C_1$ or $C' = C_2$. Let us assume that $C' = C_1$ (the other case is analogous). The action judgment $\mathsf{id} \Vdash_t \{p\}\{p\}$ holds trivially. Knowing that $(p, C_1, q) \in \mathsf{safe}_t$, it is easy to see that (2.16) can be satisfied by letting $p' = p$. Consequently, $(p, C_1 + C_2, q) \in F_t(\phi(\mathsf{safe}_t, \mathsf{safe}_t) \cup \mathsf{safe}_t)$, which concludes the proof. □

Proof of ITER**.** To do a proof by coinduction, we strengthen ITER property as follows:

$$\begin{aligned}&\forall p, C.\, ((p, C, p) \in \mathsf{safe}_t \implies (p, C^\star, p) \in \mathsf{safe}_t) \wedge \\ &(\forall p_1, C_1.\, (p_1, C_1, p) \in \mathsf{safe}_t \wedge (p, C, p) \in \mathsf{safe}_t \implies (p_1, C_1\ ;\ C^\star, p) \in \mathsf{safe}_t).\end{aligned} \tag{2.17}$$

Let us define auxiliary functions:

$$\begin{aligned}\phi(X) &\triangleq \{(p, C^\star, p) \mid (p, C, p) \in X\} \\ \psi(X) &\triangleq \{(p_1, C_1\ ;\ C_2^\star, p_2) \mid (p_1, C_1, p_2), (p_2, C_2, p_2) \in X\}.\end{aligned}$$

Using them, we rewrite (2.17) as $\phi(\mathsf{safe}_t) \cup \psi(\mathsf{safe}_t) \subseteq \mathsf{safe}_t$. Let $\xi = \{(p, \mathsf{skip}, p)\}$. It is easy to see that $\phi(\mathsf{safe}_t) \cup \psi(\mathsf{safe}_t) \cup \xi \subseteq \mathsf{safe}_t$ is also an equivalent reformulation of (2.17), since $\xi \subseteq \mathsf{safe}_t$ always holds.

Since $\mathsf{safe}_t = \mathsf{gfp}\, F_t$, we can do a proof by coinduction: to conclude that $\phi(\mathsf{safe}_t) \cup \psi(\mathsf{safe}_t) \cup \xi \subseteq \mathsf{safe}_t$, we demonstrate $\phi(\mathsf{safe}_t) \cup \psi(\mathsf{safe}_t) \cup \xi \subseteq F_t(\phi(\mathsf{safe}_t) \cup \psi(\mathsf{safe}_t) \cup \xi)$.

Consider any $(p, C, q) \in \xi$. Necessarily, $C = \mathsf{skip}$ and $q = p$. Note that $p \Rrightarrow p$ always holds, which by definition of F_t is sufficient for $(p, \mathsf{skip}, p) \in F_t(\phi(\mathsf{safe}_t) \cup \psi(\mathsf{safe}_t) \cup \xi)$. Thus, $(p, C, q) \in F_t(\phi(\mathsf{safe}_t) \cup \psi(\mathsf{safe}_t) \cup \xi)$.

Consider any $(p, C', q) \in \phi(\mathsf{safe}_t)$. Necessarily, $p = q$ and there exists a sequential command C such that $C' = C^\star$ and $(p, C, p) \in \mathsf{safe}_t$. We need to show that $(p, C^\star, p) \in F_t(\phi(\mathsf{safe}_t) \cup \psi(\mathsf{safe}_t) \cup \xi)$. For the latter to hold, by definition of F_t it is sufficient that for every α, C'' and a transition $C \rightarrowtail_\alpha C''$ the following be true:

$$\exists p''.\ \alpha \Vdash_t \{p\}\{p''\} \wedge (p'', C'', p) \in \phi(\mathsf{safe}_t) \cup \psi(\mathsf{safe}_t) \cup \xi. \tag{2.18}$$

According to the operational semantics in Figure 2.1, when there is a transition $C \rightarrowtail_\alpha C''$, necessarily $\alpha = \mathsf{id}$ and either $C'' = \mathsf{skip}$ or $C'' = C \ ; C^\star$. Let us assume that $C'' = \mathsf{skip}$. Since both $(p, \mathsf{skip}, p) \in \xi$ and $\alpha \Vdash_t \{p\}\{p\}$ always hold, it is easy to see that letting $p'' = p$ satisfies (2.18). Now let us turn to the case when $C'' = C \ ; C^\star$. Note that $(p, C \ ; C^\star, p) \in \psi(\mathsf{safe}_t)$ holds by definition of ψ. Thus, by letting $p'' = p$ we satisfy (2.18).

Consider $(p_1, C_0, p_2) \in \psi(\mathsf{safe}_t)$. Necessarily, there exist C_1 and C_2 such that:

$$C_0 = C_1 \ ; C_2^\star \wedge (p_1, C_1, p_2) \in \mathsf{safe}_t \wedge (p_2, C_2, p_2) \in \mathsf{safe}_t. \tag{2.19}$$

We need to show that $(p_1, C_1 \ ; C_2^\star, p_2) \in F_t(\phi(\mathsf{safe}_t) \cup \psi(\mathsf{safe}_t) \cup \xi)$. For the latter to hold, we need to prove the following for every α, C' and a transition $C_1 \ ; C_2^\star \rightarrowtail_\alpha C'$:

$$\exists p'.\ \alpha \Vdash_t \{p_1\}\{p'\} \wedge (p', C', p_2) \in \phi(\mathsf{safe}_t) \cup \psi(\mathsf{safe}_t) \cup \xi. \tag{2.20}$$

According to the operational semantics in Figure 2.1, when there is a transition $C_1 \ ; C_2^\star \rightarrowtail_\alpha C'$, either of the following is true:

- there are C_1' and a transition $C_1 \rightarrowtail_C C_1'$ such that $C' = C_1' \ ; C_2^\star$;
- $C_1 = \mathsf{skip}$, $C' = C_2$ and $\alpha = \mathsf{id}$.

Let us assume that the former is the case. From (2.19) we know that $(p_1, C_1, p_2) \in \mathsf{safe}_t$, so by definition of the safety relation we get that:

$$\exists p_1'.\ \alpha \Vdash_t \{p_1\}\{p_1'\} \wedge (p_1', C_1', p_2) \in \mathsf{safe}_t.$$

Consequently, $(p_1', C_1' \ ; C_2^\star, p_2) \in \psi(\mathsf{safe}_t)$. Thus, by letting $p' = p_1'$ we can satisfy (2.20).

Now let $C_1 = \mathsf{skip}$ and $\alpha = \mathsf{id}$. From (2.19) we know that $(p_1, \mathsf{skip}, p_2) \in \mathsf{safe}_t$, meaning that necessarily $p_1 \Rrightarrow p_2$. It is easy to see that $p_1 \Rrightarrow p_2$ holds if and only if so does $\mathsf{id} \Vdash_t \{p_1\}\{p_2\}$. Knowing that $(p_2, C_2, p_2) \in \mathsf{safe}_t$, we can satisfy (2.20) by letting $p' = p_2$. □

Proof of Conseq**.** Let us first show that $\mathsf{safe}_t(p', C, q)$ holds, when so do $p' \Rrightarrow p$ and $\mathsf{safe}_t(p, C, q)$. When $C = \mathsf{skip}$, $\mathsf{safe}_t(p, C, q)$ gives us that $p \Rrightarrow q$. It is easy to see that $p' \Rrightarrow p$ and $p \Rrightarrow q$ together imply $p' \Rrightarrow q$, which is sufficient to conclude that $\mathsf{safe}_t(p', C, q)$ holds. Let us assume that $C \neq \mathsf{skip}$. From $\mathsf{safe}_t(p, C, q)$ we get that the following holds of every transition $C \rightarrowtail_\alpha C'$:

$$\exists p''.\ \alpha \Vdash_t \{p\}\{p''\} \wedge \mathsf{safe}_t(p'', C', q)$$

However, by applying Proposition 2.10 about the Consequence property of axiom judgments to $p' \Rrightarrow p$ and $\alpha \Vdash_t \{p\}\{p''\}$ we get that $\alpha \Vdash_t \{p'\}\{p''\}$. Together with the formula above, it allows us to conclude that $\mathsf{safe}_t(p', C, q)$ holds.

Now let us prove that $\mathsf{safe}_t(p, C, q')$ holds, when so do $q \Rrightarrow q'$ and $\mathsf{safe}_t(p, C, q)$. We define an auxiliary function:

$$\phi(X) \triangleq \{(p, C, q') \mid \exists q.\ (p, C, q) \in X \wedge q \Rrightarrow q'\}.$$

Our goal is to prove that $\phi(\mathsf{safe}_t) \subseteq \mathsf{safe}_t$. Since $\mathsf{safe}_t = \mathsf{gfp}\, F_t$, we can do a proof by coinduction: to conclude that $\phi(\mathsf{safe}_t) \subseteq \mathsf{safe}_t$ holds, we demonstrate $\phi(\mathsf{safe}_t) \subseteq F_t(\phi(\mathsf{safe}_t))$.

Let us consider any $(p, C, q') \in \phi(\mathsf{safe}_t)$. Necessarily, there exists q such that $(p, C, q) \in \mathsf{safe}_t$ and $q \Rrightarrow q'$. When $C = \mathsf{skip}$, we need to show that $p \Rrightarrow q'$. Since $(p, \mathsf{skip}, q) \in \mathsf{safe}_t$, it is the case that $p \Rrightarrow q$. It is easy to see that $p \Rrightarrow q$ and $q \Rrightarrow q'$ together imply $p \Rrightarrow q'$, which is sufficient to conclude that $(p, C, q') \in F_t(\phi(\mathsf{safe}_t))$.

Now consider the case when $C \neq \mathsf{skip}$. Since $(p, \alpha, q) \in \mathsf{safe}_t$, by definition of the safety relation the following holds of every α, C' and a transition $C \rightarrowtail_\alpha C'$:

$$\exists p''.\ \alpha \Vdash_t \{p\}\{p''\} \wedge (p'', C', q) \in \mathsf{safe}_t$$

Knowing that $q \Rrightarrow q'$ and $(p'', C', q) \in \mathsf{safe}_t$, it is easy to see that $(p'', C', q') \in \phi(\mathsf{safe}_t)$. Thus, we have shown that:

$$\forall \alpha, C'.\, C \rightarrowtail_\alpha C' \implies \exists p''.\ \alpha \Vdash_t \{p\}\{p''\} \wedge (p'', C', q') \in \phi(\mathsf{safe}_t),$$

which is sufficient for concluding that $(p, C, q') \in F_t(\phi(\mathsf{safe}_t))$ holds. □

2.3 Soundness

In this section, we formulate linearizability for libraries. We also formulate and prove the soundness theorem, in which we state proof obligations that are necessary to conclude linearizability.

Libraries. We assume a set of method names Method, ranged over by m, and consider *a concrete library* $\ell : \mathsf{Method} \rightharpoonup ((\mathsf{Val} \times \mathsf{Val}) \to \mathsf{Com})$ that maps method names to commands from C, which are parametrized by a pair of values from Val. For a given method name $m \in \mathsf{dom}(\ell)$ and values $a, v \in \mathsf{Val}$, a command $\ell(m, a, v)$ is an implementation of m, which accepts a as a method argument and either returns v or does not terminate. Such an unusual way of specifying method's arguments and return values significantly simplifies further development, since it does not require modeling a call stack.

Along with the library ℓ we consider its specification in the form of an abstract library $\mathcal{L} \in \mathsf{Method} \rightharpoonup ((\mathsf{Val} \times \mathsf{Val}) \to \mathsf{APCom})$ implementing a set of methods $\mathsf{dom}(\mathcal{L})$ atomically as abstract primitive commands $\{\mathcal{L}(m, a, v) \mid m \in \mathsf{dom}(\mathcal{L})\}$ parametrized by an argument a and a return value v. Given a method $m \in \mathsf{Method}$, we assume that a parametrized abstract primitive command $\mathcal{L}(m)$ is intended as a specification for $\ell(m)$.

Linearizability. The 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. Consequently, we are not interested in internal steps recorded in library computations, but only in the interactions of the library with its client. We record such interactions using *histories*, which are traces including only events $\mathsf{call}\ m(a)$ and $\mathsf{ret}\ m(v)$ that indicate an invocation of a method m with a parameter a and returning from m with a return value v, or formally:

$$h ::= \varepsilon \mid (t, \mathsf{call}\ m(a)) :: h \mid (t, \mathsf{ret}\ m(v)) :: h.$$

Given a library ℓ, we generate all finite histories of ℓ by considering N threads repeatedly invoking library methods in any order and with any possible arguments. The execution of methods is described by semantics of commands from § 2.1.

We define a *thread pool* $\tau : \mathsf{ThreadID} \to (\mathsf{idle} \uplus (\mathsf{Com} \times \mathsf{Val}))$ to characterize progress of methods execution in each thread. The case of $\tau(t) = \mathsf{idle}$ corresponds to no method running in a thread t. When $\tau(t) = (C, v)$, to finish some method returning v it remains to execute C.

Definition 2.21. *We let* $\mathcal{H}[\![\ell, \sigma]\!] = \bigcup_{n \geq 0} \mathcal{H}_n[\![\ell, (\lambda t.\, \mathsf{idle}), \sigma]\!]$ *denote the set of all possible histories of a library* ℓ *that start from a state* σ*, where for a given thread pool* τ*,* $\mathcal{H}_n[\![\ell, \tau, \sigma]\!]$ *is defined as a set of histories such that* $\mathcal{H}_0[\![\ell, \tau, \sigma]\!] \triangleq \{\varepsilon\}$ *and:*

$$\begin{aligned}
\mathcal{H}_n[\![\ell, \tau, \sigma]\!] \quad \triangleq \quad & \{((t, \mathsf{call}\ m(a)) :: h) \mid a \in \mathsf{Val} \wedge m \in \mathsf{dom}(\ell) \wedge \tau(t) = \mathsf{idle} \wedge \\
& \qquad \exists v.\, h \in \mathcal{H}_{n-1}[\![\ell, \tau[t : (\ell(m, a, v), v)], \sigma]\!]\} \\
& \cup \{h \mid \exists t, \alpha, C, C', \sigma', v.\, \tau(t) = (C, v) \wedge \langle C, \sigma\rangle \xrightarrow{t,\alpha}\!\!\!\!\rightarrow \langle C', \sigma'\rangle \wedge \\
& \qquad h \in \mathcal{H}_{n-1}[\![\ell, \tau[t : (C', v)], \sigma']\!]\} \\
& \cup \{((t, \mathsf{ret}\ m(v)) :: h) \mid m \in \mathsf{dom}(\ell) \wedge \tau(t) = (\mathsf{skip}, v) \wedge \\
& \qquad h \in \mathcal{H}_{n-1}[\![\ell, \tau[t : \mathsf{idle}], \sigma]\!]\}
\end{aligned}$$

Thus, we construct the set of all finite histories inductively with all threads initially idling. At each step of generation, in any idling thread t any method $m \in \mathsf{dom}(\ell)$ may be called with any argument a and an expected return value v, which leads to adding a command $\ell(m, a, v)$ to the thread pool of a thread t. Also, any thread t, in which $\tau(t) = (C, v)$, may do a transition $\langle C, \sigma\rangle \xrightarrow{t,\alpha}\!\!\!\!\rightarrow \langle C', \sigma'\rangle$ changing a command in the thread pool and the concrete state. Finally, any thread that has finished execution of a method's command ($\tau(t) = (\mathsf{skip}, v)$) may become idle by letting $\tau(t) = \mathsf{idle}$.

In the following, we define $\mathcal{H}_n[\![\mathcal{L}, \mathcal{T}, \Sigma]\!]$ analogously with the help of an abstract thread pool $\mathcal{T} : \mathsf{ThreadID} \to (\mathsf{idle} \uplus (\mathsf{APCom} \times \mathsf{Val}))$.

Definition 2.22. *We let* $\mathcal{H}[\![\mathcal{L}, \Sigma]\!] \triangleq \bigcup_{n \geq 0} \mathcal{H}_n[\![\mathcal{L}, (\lambda t.\, \mathsf{idle}), \Sigma]\!]$ *denote the set of all possible histories of a library* $\mathcal{L}$ *that start from a state* Σ*, where for a given thread pool* $\mathcal{T}$*,* $\mathcal{H}_n[\![\mathcal{L}, \mathcal{T}, \Sigma]\!]$ *is defined as a set of histories such that* $\mathcal{H}_0[\![\mathcal{L}, \mathcal{T}, \Sigma]\!] \triangleq \{\varepsilon\}$ *and:*

$$\begin{aligned}
\mathcal{H}_n[\![\mathcal{L}, \mathcal{T}, \Sigma]\!] \quad \triangleq \quad & \{((t, \mathsf{call}\ m(a)) :: h) \mid m \in \mathsf{dom}(\mathcal{L}) \wedge \mathcal{T}(t) = \mathsf{idle} \wedge \\
& \qquad \exists r.\, h \in \mathcal{H}_{n-1}[\![\mathcal{L}, \mathcal{T}[t : (\mathcal{L}(m, a, r), r)], \Sigma]\!]\} \\
& \cup \{h \mid \exists t, \mathrm{A}, \Sigma', r.\, \mathcal{T}(t) = (\mathrm{A}, r) \wedge \Sigma' \in [\![\mathrm{A}]\!]_t(\Sigma) \wedge \\
& \qquad h \in \mathcal{H}_{n-1}[\![\mathcal{L}, \mathcal{T}[t : (\mathsf{skip}, r)], \Sigma']\!]\} \\
& \cup \{((t, \mathsf{ret}\ m(r)) :: h) \mid m \in \mathsf{dom}(\mathcal{L}) \wedge \mathcal{T}(t) = (\mathsf{skip}, r) \wedge \\
& \qquad h \in \mathcal{H}_{n-1}[\![\mathcal{L}, \mathcal{T}[t : \mathsf{idle}], \Sigma]\!]\}
\end{aligned}$$

Definition 2.23. *For libraries* ℓ *and* $\mathcal{L}$ *such that* $\mathsf{dom}(\ell) = \mathsf{dom}(\mathcal{L})$*, we say that* $\mathcal{L}$ ***linearizes*** ℓ *in the states* σ *and* Σ*, written* $(\ell, \sigma) \sqsubseteq (\mathcal{L}, \Sigma)$*, if* $\mathcal{H}[\![\ell, \sigma]\!] \subseteq \mathcal{H}[\![\mathcal{L}, \Sigma]\!]$.

That is, an abstract library $\mathcal{L}$ linearizes ℓ in the states σ and Σ, if every history of ℓ can be reproduced by $\mathcal{L}$. The definition is different from the standard one [7]: we use the result obtained by Gotsman and Yang [31] stating that the plain subset inclusion on the sets of histories produced by concrete and abstract libraries is equivalent to the original definition of linearizability.

Soundness w.r.t. linearizability. We now explain proof obligations that we need to show for every method m of a concrete library ℓ to conclude its linearizability. Particularly, for every thread t, argument a, return value v, and a command $\ell(m,a,v)$ we require that there exist assertions $\mathcal{P}(t,\mathcal{L}(m,a,v))$ and $\mathcal{Q}(t,\mathcal{L}(m,a,v))$, for which the following Hoare-style specification holds:

$$\vdash_t \{\mathcal{P}(t,\mathcal{L}(m,a,v))\}\ \ell(m,a,v)\ \{\mathcal{Q}(t,\mathcal{L}(m,a,v))\} \tag{2.24}$$

In the specification of $\ell(m,a,v)$, $\mathcal{P}(t,\mathcal{L}(m,a,v))$ and $\mathcal{Q}(t,\mathcal{L}(m,a,v))$ are assertions parametrized by a thread t and an abstract command $\mathcal{L}(m,a,v)$. We require that in a thread t of all states satisfying $\mathcal{P}(t,\mathcal{L}(m,a,v))$ and $\mathcal{Q}(t,\mathcal{L}(m,a,v))$ there be only tokens $\mathsf{todo}(\mathcal{L}(m,a,v))$ and $\mathsf{done}(\mathcal{L}(m,a,v))$ respectively:

$$\begin{array}{l}\forall \ell, t, \sigma, \Sigma, \Delta, r.\\ ((\sigma,\Sigma,\Delta) \in \lfloor [\![\mathcal{P}(t,\mathcal{L}(m,a,v))]\!]_\ell * r \rfloor \implies \Delta(t) = \mathsf{todo}(\mathcal{L}(m,a,v)))\\ \wedge\ ((\sigma,\Sigma,\Delta) \in \lfloor [\![\mathcal{Q}(t,\mathcal{L}(m,a,v))]\!]_\ell * r \rfloor \implies \Delta(t) = \mathsf{done}(\mathcal{L}(m,a,v)))\end{array} \tag{2.25}$$

Together, (2.24) and (2.25) impose a requirement that a concrete and an abstract method return the same return value v. We also require that the states satisfying the assertions only differ by a token of a thread t:

$$\begin{array}{l}\forall \ell, t, \mathrm{A}, \mathrm{A}', r, \Delta.\, (\sigma,\Sigma,\Delta[t:\mathsf{done}(\mathrm{A})]) \in \lfloor [\![\mathcal{Q}(t,\mathrm{A})]\!]_\ell * r \rfloor \iff\\ \qquad\qquad (\sigma,\Sigma,\Delta[t:\mathsf{todo}(\mathrm{A}')]) \in \lfloor [\![\mathcal{P}(t,\mathrm{A}')]\!]_\ell * r \rfloor.\end{array} \tag{2.26}$$

Theorem 2.27. *For given libraries ℓ and $\mathcal{L}$ together with states σ and Σ, $(\ell,\sigma) \sqsubseteq (\mathcal{L},\Sigma)$ holds, if $\mathsf{dom}(\ell) = \mathsf{dom}(\mathcal{L})$ and (2.24), (2.25) and (2.26) hold for every method m, thread t and values a and v.*

Proof of Theorem 2.27. We further refer to the assumptions of Theorem 2.27 as a relation $\mathsf{safelib}(\ell,\mathcal{L},\mathcal{P},\mathcal{Q})$ defined as follows.

Definition 2.28. *Given a concrete library ℓ, an abstract library $\mathcal{L}$ and $\mathcal{P},\mathcal{Q} : \mathsf{ThreadID} \times \mathsf{APCom} \to \mathsf{VAssn}$, we say that a relation $\mathsf{safelib}(\ell,\mathcal{L},\mathcal{P},\mathcal{Q})$ holds if and only if the following requirements are met:*

1. $\mathsf{dom}(\ell) = \mathsf{dom}(\mathcal{L})$;
2. $\forall \ell, t, \mathrm{A}, \sigma, \Sigma, \Delta, r.\, (\sigma,\Sigma,\Delta) \in \lfloor [\![\mathcal{P}(t,\mathrm{A})]\!]_\ell * r \rfloor \implies \Delta(t) = \mathsf{todo}(\mathrm{A})$;
3. $\forall \ell, t, \mathrm{A}, \sigma, \Sigma, \Delta, r.\, (\sigma,\Sigma,\Delta) \in \lfloor [\![\mathcal{Q}(t,\mathrm{A})]\!]_\ell * r \rfloor \implies \Delta(t) = \mathsf{done}(\mathrm{A})$;
4. $\forall \ell, t, \mathrm{A}, \mathrm{A}', r, \Delta.\, ((\sigma,\Sigma,\Delta[t:\mathsf{todo}(\mathrm{A})]) \in \lfloor [\![\mathcal{P}(t,\mathrm{A})]\!]_\ell * r \rfloor \iff (\sigma,\Sigma,\Delta[t:\mathsf{done}(\mathrm{A}')]) \in \lfloor [\![\mathcal{Q}(t,\mathrm{A}')]\!]_\ell * r \rfloor)$.
5. $\forall m, a, r, t.\, m \in \mathsf{dom}(\ell) \wedge a, r \in \mathsf{Val} \wedge t \in \mathsf{ThreadID} \implies \vdash_t \{\mathcal{P}(t,\mathcal{L}(m,a,r))\}\ \ell(m,a,r)\ \{\mathcal{Q}(t,\mathcal{L}(m,a,r))\}$;

To strengthen the statement of Theorem 2.27 as necessary for its proof, we define an auxiliary relation, a *thread pool invariant*. With this relation we establish a correspondence between the information about LP in a thread t from a given view v_t and sequential commands in a thread t of a concrete thread pool τ and abstract thread pool $\mathcal{T}$.

Definition 2.29. *Given a concrete library ℓ, an abstract library $\mathcal{L}$, predicates $\mathcal{P}, \mathcal{Q} : \mathsf{ThreadID} \times \mathsf{APCom} \to \mathsf{VAssn}$, a concrete thread pool τ, an abstract thread pool $\mathcal{T}$, a view v_t and an interpretation of logical variables ℓ, we say that a thread pool invariant $\mathsf{cinv}_t(\ell, \tau, \mathcal{T}, v_t, \Delta)$ holds in a thread t if and only if the following requirements are met:*

- *if $\tau(t) = \mathsf{idle}$, then $\mathcal{T}(t) = \mathsf{idle}$ and $v_t \Rrightarrow [\![\mathcal{Q}(t, \_)]\!]_\ell$, or*
- *there exist C, r, m, a such that $\tau(t) = (C, r)$ and the following holds:*

$$\begin{aligned}&\mathsf{safe}_t(v_t, C, \mathcal{Q}(t, \mathcal{L}(m, a, r))) \wedge \\ &\qquad ((\Delta(t) = \mathsf{todo}(\mathcal{L}(m, a, r)) \wedge \mathcal{T}(t) = (\mathcal{L}(m, a, r), r)) \vee \\ &\qquad\qquad (\Delta(t) = \mathsf{done}(\mathcal{L}(m, a, r)) \wedge \mathcal{T}(t) = (\mathsf{skip}, r))).\end{aligned}$$

We are now ready to prove Theorem 2.27.

Proof of Theorem 2.27. Let us consider any $\ell, \mathcal{L}, \mathcal{P}, \mathcal{Q}$ such that $\mathsf{safelib}(\ell, \mathcal{L}, \mathcal{P}, \mathcal{Q})$ holds. Let us explain how we strengthen the statement of the theorem in this proof. We prove that $\forall n.\, \phi(n)$ holds with $\phi(n)$ formulated as follows:

$$\begin{aligned}\phi(n) = \forall \ell, \sigma, \Sigma, \Delta, \tau, \mathcal{T}.\, (\exists v_1, \ldots, v_N.\, (\forall k.\, \mathsf{cinv}_k(\ell, \tau, \mathcal{T}, v_k, \Delta)) \wedge \\ (\sigma, \Sigma, \Delta) \in \lfloor \circledast_{t \in \mathsf{ThreadID}} v_t \rfloor) \implies \mathcal{H}_n[\![\ell, \tau, \sigma]\!] \subseteq \mathcal{H}[\![\mathcal{L}, \mathcal{T}, \Sigma]\!]. \quad (2.30)\end{aligned}$$

Note that according to the semantics of the assertion language Assn (Figure 2.2):

$$[\![\circledast_{k \in \mathsf{ThreadID}} (\exists \mathrm{A}.\, \mathcal{Q}(k, \mathrm{A}))]\!]_\ell = \circledast_{k \in \mathsf{ThreadID}} [\![(\exists \mathrm{A}.\, \mathcal{Q}(k, \mathrm{A}))]\!]_\ell.$$

With that in mind, it is easy to see that letting $v_k = [\![(\exists \mathrm{A}.\, \mathcal{Q}(k, \mathrm{A}))]\!]_\ell$ for all $k \in \mathsf{ThreadID}$, $\tau = (\lambda t.\, \mathsf{idle})$ and $\mathcal{T} = (\lambda t.\, \mathsf{idle})$ in (2.30) yields the formula:

$$\begin{aligned}(\forall \ell, \sigma, \Sigma, \Delta.\, (\sigma, \Sigma, \Delta) \in \lfloor [\![\circledast_{k \in \mathsf{ThreadID}} (\exists \mathrm{A}.\, \mathcal{Q}(k, \mathrm{A}))]\!]_\ell \rfloor \implies \\ \bigcup_{n \geq 0} \mathcal{H}_n[\![\ell, \lambda t.\, \mathsf{idle}, \sigma]\!] \subseteq \mathcal{H}[\![\mathcal{L}, \lambda t.\, \mathsf{idle}, \Sigma]\!]),\end{aligned}$$

which coincides with the statement of the theorem.

We prove $\forall n.\, \phi(n)$ by induction on n. Let us take any $\ell, \sigma, \Sigma, \Delta, \tau$ and $\mathcal{T}$, and consider $v_1, \ldots, v_N$ such that the premisses of $\phi(n)$ hold:

$$(\forall k.\, \mathsf{cinv}_k(\ell, \tau, \mathcal{T}, v_k, \Delta)) \wedge (\sigma, \Sigma, \Delta) \in \lfloor \circledast_{k \in \mathsf{ThreadID}} v_k \rfloor \quad (2.31)$$

We need to demonstrate that every history h of the concrete library ℓ from the set $\mathcal{H}_n[\![\ell, \tau, \sigma]\!]$ is also a history of the abstract library $\mathcal{L}$: $h \in \mathcal{H}[\![\mathcal{L}, \mathcal{T}, \Sigma]\!]$.

By Definition 2.21 of $\mathcal{H}_n[\![\ell, \tau, \sigma]\!]$, if $n = 0$, then h is an empty history that is trivially present in $\mathcal{H}[\![\mathcal{L}, \mathcal{T}, \Sigma]\!]$. Let us now consider $n > 0$ and assume that $\phi(n-1)$ holds. By definition of $\mathcal{H}_n[\![\ell, \tau, \sigma]\!]$, h corresponds to one of the three events in a thread t: a call of an arbitrary method m with an argument a in a thread t, a return from a method m with a return value r or a transition in a thread t. We consider each case separately.

Case #1. There is a history h', a thread t, a method $m \in \mathsf{dom}(\ell)$, its argument a and a return value r such that $h = (t, \mathsf{call}\ m(a)) :: h'$, $\tau(t) = \mathsf{idle}$ and $h' \in \mathcal{H}_{n-1}[\![\ell, \tau[t : (\ell(m,a,r),r), \sigma]\!]$. By Definition 2.22, to conclude that $h = (t, \mathsf{call}\ m(a)) :: h' \in \mathcal{H}[\![\mathcal{L}, \mathcal{T}, \Sigma]\!]$ it is necessary to show that $\mathcal{T}(t) = \mathsf{idle}$ and $h' \in \mathcal{H}[\![\mathcal{L}, \mathcal{T}[t : (\mathcal{L}(m,a,r),r)], \Sigma]\!]$, which we further do in the proof of Case #1.

According to (2.31), $\mathsf{cinv}_t(\ell, \tau, \mathcal{T}, v_t, \Delta)$ and $(\sigma, \Sigma, \Delta) \in \lfloor \circledast_{k \in \mathsf{ThreadID}} v_k \rfloor$ hold. Then necessarily $\mathcal{T}(t) = \mathsf{idle}$ and $v_t \Rrightarrow [\![\mathcal{Q}(t, \_)]\!]_\ell$, which corresponds to the only case when $\tau(t) = \mathsf{idle}$ in the thread pool invariant. By Definition 2.3 of repartitioning implication, $v_t \Rrightarrow [\![\mathcal{Q}(t, \_)]\!]_\ell$, $(\sigma, \Sigma, \Delta) \in \lfloor \circledast_{k \in \mathsf{ThreadID}} v_k \rfloor$ implies $(\sigma, \Sigma, \Delta) \in \lfloor [\![\mathcal{Q}(t, \_)]\!]_\ell * \circledast_{k \in \mathsf{ThreadID} \setminus \{t\}} v_k \rfloor$. From requirements to predicates $\mathcal{P}$ and $\mathcal{Q}$ in $\mathsf{safelib}(\ell, \mathcal{L}, \mathcal{P}, \mathcal{Q})$ we obtain that the following holds of (σ, Σ, Δ):

- $\Delta(t) = \mathsf{done}(\_)$, and
- $(\sigma, \Sigma, \Delta[t : \mathsf{todo}(\mathcal{L}(m,a,r))]) \in \lfloor [\![\mathcal{P}(t, \mathcal{L}(m,a,r))]\!]_\ell * \circledast_{k \in \mathsf{ThreadID} \setminus \{t\}} v_k \rfloor$.

Let $v'_t = [\![\mathcal{P}(t, \mathcal{L}(m,a,r))]\!]_\ell$ and $v'_k = v_k$ for all $k \neq t$. Obviously, $(\sigma, \Sigma, \Delta[t : \mathsf{todo}(\mathcal{L}(m,a,r))]) \in \lfloor \circledast_k v'_k \rfloor$.

Also, by Lemma 2.6, $\mathsf{safe}_t([\![\mathcal{P}(t, \mathcal{L}(m,a,r))]\!]_\ell, \ell(m,a,r), [\![\mathcal{Q}(t, \mathcal{L}(m,a,r))]\!]_\ell)$ holds. This allows us to conclude that in a thread t a thread pool invariant $\mathsf{cinv}_t(\ell, \tau[t : (\ell(m,a,r),r)], \mathcal{T}[t : (\mathcal{L}(m,a,r),r)], v'_t, \Delta[t : \mathcal{L}(m,a,r)])$ holds. Moreover, according to (2.31), thread pool invariants hold in all other threads as well.

We have shown that there exist $v'_1, \ldots, v'_N$ such that:

$$(\forall k.\, \mathsf{cinv}_k(\ell, \tau[t : (\ell(m,a,r),r)], \mathcal{T}[t : (\mathcal{L}(m,a,r),r)], v'_k, \Delta[t : \mathcal{L}(m,a,r)])) \wedge (\sigma, \Sigma, \Delta[t : \mathsf{todo}(\mathcal{L}(m,a,r))]) \in \lfloor \circledast_{k \in \mathsf{ThreadID}} v'_k \rfloor,$$

which by the induction hypothesis $\phi(n-1)$ implies that $h' \in \mathcal{H}_{n-1}[\![\ell, \tau[t : (\ell(m,a,r),r)], \sigma]\!] \subseteq \mathcal{H}[\![\mathcal{L}, \mathcal{T}[t : (\mathcal{L}(m,a,r),r)], \Sigma]\!]$. We have also show that $\mathcal{T}(t) = \mathsf{idle}$. By Definition 2.22, $h = (t, \mathsf{call}\ m(a)) :: h' \in \mathcal{H}[\![\mathcal{L}, \mathcal{T}, \Sigma]\!]$, which concludes the proof of Case #1.

Case #2. There is a history h', a thread t, a method $m \in \mathsf{dom}(\ell)$, its argument a and a return value r such that $h = (t, \mathsf{ret}\ m(r)) :: h'$, $\tau(t) = (\mathsf{skip}, r)$ and $h' \in \mathcal{H}_{n-1}[\![\ell, \tau[t : \mathsf{idle}], \sigma]\!]$. By Definition 2.22, to conclude that $h = (t, \mathsf{ret}\ m(r)) :: h' \in \mathcal{H}[\![\mathcal{L}, \mathcal{T}, \Sigma]\!]$ it is necessary to show that $\mathcal{T}(t) = (\mathsf{skip}, r)$ and $h' \in \mathcal{H}[\![\mathcal{L}, \mathcal{T}[t : \mathsf{idle}], \Sigma]\!]$, which we further do in this proof of Case #2.

According to (2.31), a thread invariant $\mathsf{cinv}_t(\ell, \tau, \mathcal{T}, v_t, \Delta)$ holds. Then the following is true:

$$\mathsf{safe}_t(v_t, \mathsf{skip}, [\![\mathcal{Q}(t, \mathcal{L}(m,a,r))]\!]_\ell) \wedge ((\Delta(t) = \mathsf{todo}(\mathcal{L}(m,a,r)) \wedge \mathcal{T}(t) = (\mathcal{L}(m,a,r),r)) \vee (\Delta(t) = \mathsf{done}(\mathcal{L}(m,a,r)) \wedge \mathcal{T}(t) = (\mathsf{skip}, r))). \quad (2.32)$$

By Definition 2.5 of $\mathsf{safe}_t(v_t, \mathsf{skip}, [\![\mathcal{Q}(t, \mathcal{L}(m,a,r))]\!]_\ell)$, $v_t \Rrightarrow [\![\mathcal{Q}(t, \mathcal{L}(m,a,r))]\!]_\ell$ holds. Consequently, by Definition 2.3:

$$(\sigma, \Sigma, \Delta) \in \lfloor v_t * \circledast_{k \in \mathsf{ThreadID} \setminus \{t\}} v_k \rfloor \subseteq \lfloor [\![\mathcal{Q}(t, \mathcal{L}(m,a,r))]\!]_\ell * \circledast_{k \in \mathsf{ThreadID} \setminus \{t\}} v_k \rfloor.$$

From the third requirement to $\mathcal{Q}$ in Definition 2.28:

$$(\sigma, \Sigma, \Delta) \in \lfloor [\![\mathcal{Q}(t, \mathcal{L}(m,a,r))]\!]_\ell * \circledast_k v_k \rfloor \implies \Delta(t) = \mathsf{done}(\mathcal{L}(m,a,r)).$$

Consequently, from (2.32) we get that $\Delta(t) = \mathsf{done}(\mathcal{L}(m,a,r))$ and $\mathcal{T}(t) = (\mathsf{skip}, r)$.

Let $v_t' = [\![\mathcal{Q}(t, \mathcal{L}(m,a,r))]\!]_\ell$ and $v_k' = v_k$ for $k \neq t$. It is easy to see that $\mathsf{cinv}_t(\ell, \tau[t : \mathsf{idle}], \mathcal{T}[t : \mathsf{idle}], v_t', \Delta)$ holds trivially by Definition 2.29. Moreover, according to (2.31), thread pool invariants hold in other threads as well.

We have shown that there exist $v_1', \ldots, v_N'$ such that:

$$(\forall k.\, \mathsf{cinv}_k(\ell, \tau[t : \mathsf{idle}], \mathcal{T}[t : \mathsf{idle}], v_k', \Delta)) \wedge (\sigma, \Sigma, \Delta) \in \lfloor \circledast_{k \in \mathsf{ThreadID}} v_k' \rfloor,$$

which by the induction hypothesis $\phi(n-1)$ implies that $h' \in \mathcal{H}_{n-1}[\![\ell, \tau[t : \mathsf{idle}], \sigma]\!] \subseteq \mathcal{H}[\![\mathcal{L}, \mathcal{T}[t : \mathsf{idle}], \Sigma]\!]$. We have also show that $\mathcal{T}(t) = (\mathsf{skip}, r)$. By Definition 2.22, $h = (t, \mathsf{ret}\ m(r)) :: h' \in \mathcal{H}[\![\mathcal{L}, \mathcal{T}, \Sigma]\!]$, which concludes the proof of Case #2.

Case #3. There is a thread t, sequential commands C and C', a primitive command α, concrete states σ and σ' and a return value r such that $\tau(t) = (C, r)$, $\langle C, \sigma \rangle \xrightarrow{t,\alpha}\!\!\!\!\rightarrow \langle C', \sigma' \rangle$ and $h \in \mathcal{H}_{n-1}[\![\ell, \tau[t : (C', r)], \sigma']\!]$.

According to (2.31), $\mathsf{cinv}_t(\ell, \tau, \mathcal{T}, v_t, \Delta)$ holds. Consequently, there exist a method m with its argument a such that $C = \mathcal{L}(m,a,r)$ and:

$$\begin{aligned}\mathsf{safe}_t(v_t, C, [\![\mathcal{Q}(t, \mathcal{L}(m,a,r))]\!]_\ell) \wedge \\ ((\Delta(t) = \mathsf{todo}(\mathcal{L}(m,a,r)) \wedge \mathcal{T}(t) = (\mathcal{L}(m,a,r), r)) \vee \\ (\Delta(t) = \mathsf{done}(\mathcal{L}(m,a,r)) \wedge \mathcal{T}(t) = (\mathsf{skip}, r))).\end{aligned} \quad (2.33)$$

It is easy to see that whenever there is a transition $\langle C, \sigma \rangle \xrightarrow{t,\alpha}\!\!\!\!\rightarrow \langle C', \sigma' \rangle$, there also is a stateless transition $C \rightarrowtail_\alpha C'$. By Definition 2.5 of the safety judgment $\mathsf{safe}_t(v_t, C, [\![\mathcal{Q}(t, \mathcal{L}(m,a,r))]\!]_\ell)$, if $C \rightarrowtail_\alpha C'$, then there exists a view v_t' such that $\alpha \Vdash_t \{v_t\}\{v_t'\}$ and $\mathsf{safe}_t(v_t', C', [\![\mathcal{Q}(t, \mathcal{L}(m,a,r))]\!]_\ell)$.

Let $v_k' = v_k$ for any $k \neq t$. By Definition 2.2 of the action judgment $\alpha \Vdash_t \{v_t\}\{v_t'\}$, for $(\sigma, \Sigma, \Delta) \in \lfloor v_t * \circledast_{k \neq t} v_k \rfloor$ and any $\sigma' \in [\![\alpha]\!]_t(\sigma)$, there exist Σ', Δ' such that:

$$\mathsf{LP}^*(\Sigma, \Delta, \Sigma', \Delta') \wedge (\sigma', \Sigma', \Delta') \in \lfloor v_t' * \circledast_{k \neq t} v_k \rfloor. \quad (2.34)$$

Let us assume that $\Delta = \Delta'$. Note that $\mathsf{safe}_t(v_t', C', [\![\mathcal{Q}(t, \mathcal{L}(m,a,r))]\!]_\ell)$ holds, and according to (2.33) the following holds too:

$$\begin{aligned}(\Delta(t) = \mathsf{todo}(\mathcal{L}(m,a,r)) \wedge \mathcal{T}(t) = (\mathcal{L}(m,a,r), r)) \vee \\ (\Delta(t) = \mathsf{done}(\mathcal{L}(m,a,r)) \wedge \mathcal{T}(t) = (\mathsf{skip}, r))\end{aligned}$$

Thus, it is easy to see that $\mathsf{cinv}_t(\ell, \tau[t : (C', r)], \mathcal{T}, v_t', \Delta)$ holds. Combining this observation with (2.34), we conclude that we have demonstrated existence of $v_1', \ldots, v_N'$ such that:

$$(\forall k.\, \mathsf{cinv}_k(\ell, \tau[t : (C', r)], \mathcal{T}, v_k', \Delta)) \wedge (\sigma', \Sigma', \Delta) \in \lfloor \circledast_{k \in \mathsf{ThreadID}} v_k' \rfloor,$$

which by the induction hypothesis $\phi(n-1)$ implies that $h \in \mathcal{H}_{n-1}[\![\ell, \tau[t : (C', r)], \sigma']\!] \subseteq \mathcal{H}[\![\mathcal{L}, \mathcal{T}, \Sigma]\!]$. This concludes the proof of the case when $\Delta = \Delta'$.

We now return to the case when $\Delta \neq \Delta'$. According to (2.34), $\mathsf{LP}^*(\Sigma, \Delta, \Sigma', \Delta')$ holds, meaning that linearization points of one or more threads have been passed. Without loss of generality, we assume the case of exactly one linearization point, i.e. that $\mathsf{LP}(\Sigma, \Delta, \Sigma', \Delta')$ holds. Consequently, according to Definition 2.2 there exist t' and A' such that:

$$\Sigma' \in [\![\mathrm{A}']\!]_{t'}(\Sigma) \wedge \Delta(t') = \mathsf{todo}(\mathrm{A}') \wedge \Delta' = \Delta[t' : \mathsf{done}(\mathrm{A}') \tag{2.35}$$

Let us consider the thread pool invariant $\mathsf{cinv}_{t'}(\ell, \tau, \mathcal{T}, v_{t'}, \Delta)$, which holds according to (2.31). We show that $\tau(t') \neq \mathsf{idle}$. From (2.35) we know that $\Delta(t') = \mathsf{todo}(\mathrm{A}')$. Since the third requirement to $\mathcal{Q}$ in Definition 2.28 requires that $\Delta(t) = \mathsf{done}(\mathcal{L}(m, a, r))$ hold, by Definition 2.29 it can only be the case that there exist C'', m', a', r' such that $\tau(t') = (C'', r')$ and the following is true:

$$\begin{aligned}&\mathsf{safe}_{t'}(v_t, C'', [\![\mathcal{Q}(t', \mathcal{L}(m', a', r'))]\!]_\ell) \wedge \\ &\quad ((\Delta(t') = \mathsf{todo}(\mathcal{L}(m', a', r')) \wedge \mathcal{T}(t') = (\mathcal{L}(m', a', r'), r')) \vee \\ &\quad\quad (\Delta(t') = \mathsf{done}(\mathcal{L}(m', a', r')) \wedge \mathcal{T}(t') = (\mathsf{skip}, r'))).\end{aligned} \tag{2.36}$$

From formula (2.35) we know that $\mathrm{A}' = \mathcal{L}(m', a', r')$. Consequently, $\Delta'(t') = \mathsf{done}(\mathcal{L}(m', a', r')) \wedge \mathcal{T}[t' : (\mathsf{skip}, r')](t') = (\mathsf{skip}, r')$ holds, which allows us to conclude the thread pool invariant $\mathsf{cinv}_{t'}(\ell, \tau[t : (C', r)], \mathcal{T}[t' : (\mathsf{skip}, r')], v'_{t'}, \Delta')$ in case of $t' \neq t$.

We now show that $\mathsf{cinv}_t(\ell, \tau[t : (C', r)], \mathcal{T}[t' : (\mathsf{skip}, r')], v'_t, \Delta')$ hold, both when $t = t'$ and $t \neq t'$. Let us first assume $t = t'$ ($r = r'$). Then from (2.33) we get that $\mathrm{A} = \mathcal{L}(m, a, r)$ and $\Delta'(t) = \mathsf{done}(\mathcal{L}(m, a, r))$ hold. When $t \neq t'$, no abstract transition is made in t, so $\Delta(t) = \Delta'(t)$ and $\mathcal{T}(t) = \mathcal{T}[t' : (\mathsf{skip}, r')](t)$. Consequently, the following is true in both cases:

$$\begin{aligned}&((\Delta'(t) = \mathsf{todo}(\mathcal{L}(m, a, r)) \wedge \mathcal{T}[t' : (\mathsf{skip}, r')](t) = (\mathcal{L}(m, a, r), r)) \vee \\ &\quad (\Delta'(t) = \mathsf{done}(\mathcal{L}(m, a, r)) \wedge \mathcal{T}[t' : (\mathsf{skip}, r')](t) = (\mathsf{skip}, r'))).\end{aligned} \tag{2.37}$$

Together with $\mathsf{safe}_t(v'_t, C', [\![\mathcal{Q}(t, \mathcal{L}(m, a, r))]\!]_\ell)$, those observations imply $\mathsf{cinv}_t(\ell, \tau[t : (C', r)], \mathcal{T}[t' : (\mathsf{skip}, r')], v'_t, \Delta')$.

Combining these observations with $(\sigma', \Sigma', \Delta') \in \lfloor v'_t * \circledast_{k \neq t} v_k \rfloor$ following from (2.34), we conclude that we have demonstrated existence of $v'_1, \ldots, v'_N$ such that:

$$(\forall k.\, \mathsf{cinv}_k(\ell, \tau[t : (C', r)], \mathcal{T}[t' : (\mathsf{skip}, r')], v'_k, \Delta')) \wedge (\sigma', \Sigma', \Delta') \in \lfloor \circledast_{k \in \mathsf{ThreadID}} v'_k \rfloor,$$

which by the induction hypothesis $\phi(n-1)$ implies that $h \in \mathcal{H}_{n-1}[\![\ell, \tau[t : (C', r)], \sigma']\!] \subseteq \mathcal{H}[\![\mathcal{L}, \mathcal{T}[t : (\mathsf{skip}, r')], \Sigma']\!]$. Now that we demonstrated that $\mathcal{T}(t') = (\mathcal{L}(m', a', r'), r')$, $\Sigma' \in [\![\mathcal{L}(m', a', r')]\!]_t(\Sigma)$ and $h \in \mathcal{H}[\![\mathcal{L}, \mathcal{T}[t : (\mathsf{skip}, r')], \Sigma']\!]$ all hold, by Definition 2.22 we can conclude that $h \in \mathcal{H}[\![\mathcal{L}, \mathcal{T}, \Sigma]\!]$. □

2.4 The RGSep-based Logic

In this section, we demonstrate an instance of the generic proof system that is capable of handling algorithms with helping. This instance is based on RGSep [24], which combines rely-guarantee reasoning [16] with separation logic [15].

The main idea of the logic is to partition the state into several thread-local parts (which can only be accessed by corresponding threads) and the shared part (which can be accessed by all threads). The partitioning is defined by proofs in the logic: an assertion in the code of a thread restricts its local state and the shared state. In addition, the partitioning is dynamic, meaning that resources, such as a part of a heap or a token, can be moved from the local state of a thread into the shared state and vice versa. By transferring a token to the shared state, a thread gives to its environment a permission to change the abstract state. This allows us to reason about environment helping that thread.

The RGSep-based view monoid. Similarly to DCSL, we assume that states represent heaps, i.e. that $\mathsf{State} = \mathsf{AState} = \mathsf{Loc} \rightharpoonup_{\mathrm{fin}} \mathsf{Val} \uplus \{\lightning\}$, and we denote all states but a faulting one with $\mathsf{State_H} = \mathsf{AState_H} = \mathsf{Loc} \rightharpoonup_{\mathrm{fin}} \mathsf{Val}$. We also assume a standard set of heap-manipulating primitive commands with usual semantics.

We define views as triples consisting of three components: a predicate P and binary relations R and G. A predicate $P \in \mathcal{P}((\mathsf{State_H} \times \mathsf{AState_H} \times \mathsf{Tokens})^2)$ is a set of pairs (l, s) of local and shared parts of the state, where each part consists of concrete state, abstract state and tokens. *Guarantee* G and *rely* R are relations from $\mathcal{P}((\mathsf{State} \times \mathsf{AState} \times \mathsf{Tokens})^2)$, which summarize how individual primitive commands executed by the method's thread (in case of G) and the environment (in case of R) may change the shared state. Together guarantee and rely establish a protocol that views of the method and its environment respectively must agree on each other's transitions, which allows us to reason about every thread separately without considering local state of other threads, assuming that they follow the protocol. The agreement is expressed with the help of a well-formedness condition on views of the RGSep-based monoid that their predicates must be *stable* under rely, meaning that their predicates take into account whatever changes their environment can make:

$$\mathsf{stable}(P, R) \triangleq \forall l, s, s'.\, (l, s) \in P \land (s, s') \in R \implies (l, s') \in P,$$

where we use l as a shorthand for $(\sigma_l, \Sigma_l, \Delta_l)$, and s as a shorthand for $(\sigma_s, \Sigma_s, \Delta_s)$.

A predicate that is stable under rely cannot be invalidated by any state transition from rely. Stable predicates with rely and guarantee relations form the view monoid with the underlying set of views:

$$\mathsf{Views_{RGsep}} \triangleq \{(P, R, G) \mid \mathsf{stable}(P, R)\} \cup \{\bot\},$$

where $\bot$ denotes a special *inconsistent* view with the empty reification. The reification of other views simply joins shared and local parts of the state:

$$\lfloor (P, R, G) \rfloor = \{(\sigma_l \bullet \sigma_s, \Sigma_l \bullet \Sigma_s, \Delta_l \uplus \Delta_s) \mid ((\sigma_l, \Sigma_l, \Delta_l), (\sigma_s, \Sigma_s, \Delta_s)) \in P\}.$$

Let an operation $\cdot$ be defined on states analogously to DCSL. Given predicates P and P', we let $P * P'$ be a predicate denoting the pairs of local and shared states in which the local state can be divided into two substates such that one of them together with the shared state satisfies P and the other together with the shared state satisfies P':

$$P * P' \triangleq \{((\sigma_l \bullet \sigma'_l, \Sigma_l \bullet \Sigma'_l, \Delta_l \uplus \Delta'_l), s) \mid ((\sigma_l, \Sigma_l, \Delta_l), s) \in P \land ((\sigma'_l, \Sigma'_l, \Delta'_l), s) \in P'\}$$

$\models$: $(\mathsf{State} \times \mathsf{AState} \times \mathsf{Tokens}) \times (\mathsf{State} \times \mathsf{AState} \times \mathsf{Tokens}) \times \mathsf{Int} \times \mathsf{Assn}$

$$
\begin{array}{ll}
(l,s,\ell) \models E \mapsto F, & \text{iff } \sigma_l = [[\![E]\!]_\ell : [\![F]\!]_\ell], \Sigma_l = [\,], \text{ and } \Delta_l = [\,] \\
(l,s,\ell) \models E \Mapsto F, & \text{iff } \sigma_l = [\,], \Sigma_l = [[\![E]\!]_\ell : [\![F]\!]_\ell], \text{ and } \Delta_l = [\,] \\
(l,s,\ell) \models [\mathsf{todo}(\mathrm{A})]_t\,, & \text{iff } \sigma_l = [\,], \Sigma_l = [\,], \text{ and } \Delta_l = [t : \mathsf{todo}(\mathrm{A})] \\
(l,s,\ell) \models [\mathsf{done}(\mathrm{A})]_t\,, & \text{iff } \sigma_l = [\,], \Sigma_l = [\,], \text{ and } \Delta_l = [t : \mathsf{done}(\mathrm{A})] \\
(l,s,\ell) \models \boxed{\pi}, & \text{iff } \sigma_l = [\,], \Sigma_l = [\,], \Delta_l = [\,], \text{ and} \\
 & \qquad (s, ([\,],[\,],[\,]), \ell) \models \pi \\
(l,s,\ell) \models \pi * \pi', & \text{iff there exist } \sigma_l', \sigma_l'', \Sigma_l', \Sigma_l'', \Delta_l', \Delta_l'' \text{ such that} \\
 & l = (\sigma_l' \bullet \sigma_l'', \Sigma_l' \bullet \Sigma_l'', \Delta_l' \uplus \Delta_l''), \\
 & ((\sigma_l', \Sigma_l', \Delta_l'), s, \ell) \models \pi, \text{ and} \\
 & ((\sigma_l'', \Sigma_l'', \Delta_l''), s, \ell) \models \pi'
\end{array}
$$

Figure 2.4: Satisfaction relation for a fragment of the assertion language VAssn (for brevity, we let $l = (\sigma_l, \Sigma_l, \Delta_l)$ and $s = (\sigma_s, \Sigma_s, \Delta_s)$ in each case)

We now define the monoid operation $*$, which we use to compose views of different threads. When composing views (P, R, G) and (P', R', G') of the parallel threads, we require predicates of both to be immune to interference by all other threads and each other. Otherwise, the result is inconsistent:

$$(P,R,G) * (P',R',G') \triangleq \text{if } G \subseteq R' \wedge G' \subseteq R \text{ then } (P * P', R \cap R', G \cup G') \text{ else } \bot.$$

That is, we let the composition of views be consistently defined when the state transitions allowed in a guarantee of one thread are treated as environment transitions in the other thread, i.e. $G \subseteq R'$ and $G' \subseteq R$. The rely of the composition is $R \cap R'$, since the predicate $P * P'$ is guaranteed to be stable only under environment transitions described by both R and R'. The guarantee of the composition is $G \cup G'$, since other views need to take into account all state transitions either from G or from G'.

Finally, the unit u_{RGsep} is a view that does not restrict states and the allowed state transitions of the environment, while disallowing any action in the current thread:

$$u_{\mathsf{RGsep}} = (\{([\,],[\,],[\,])\} \times (\mathsf{State} \times \mathsf{AState} \times \mathsf{Tokens}), (\mathsf{State} \times \mathsf{AState} \times \mathsf{Tokens})^2, \emptyset)$$

The RGSep-based program logic. We define the view assertion language VAssn that is a parameter of the proof system. Each view assertion ρ takes form of a triple $(\pi, \mathcal{R}, \mathcal{G})$, and the syntax for π is:

$$
\begin{array}{lll}
E & ::= & a \mid X \mid E + E \mid \ldots, \quad \text{where } X \in \mathsf{LVar}, a \in \mathsf{Val} \\
\pi & ::= & E = E \mid E \mapsto E \mid E \Mapsto E \mid [\mathsf{todo}(\mathrm{A})]_t \mid [\mathsf{done}(\mathrm{A})]_t \mid \boxed{\pi} \mid \pi * \pi \mid \neg\pi \mid \ldots
\end{array}
$$

Formula π denotes a predicate of a view as defined by a satisfaction relation $\models$ in Figure 2.4. There $E \mapsto E$ and $E \Mapsto E$ denote a concrete and an abstract state describing singleton heaps. A non-boxed formula π denotes the view with the local state satisfying π and shared state unrestricted; $\boxed{\pi}$ denotes the view with the empty local state and the shared state satisfying π; $\pi * \pi'$ the composition of predicates corresponding to π and π'. The semantics of the rest of connectives is standard. Additionally, for simplicity of presentation of the syntax, we require

that boxed assertions $\boxed{\pi}$ be not nested (as opposed to preventing that in the definition).

The other components $\mathcal{R}$ and $\mathcal{G}$ of a view assertion are sets of *rely/guarantee actions* $\mathcal{A}$ with the syntax: $\mathcal{A} ::= \pi \rightsquigarrow \pi'$. An action $\pi \rightsquigarrow \pi'$ denotes a change of a part of the shared state that satisfies π into one that satisfies π', while leaving the rest of the shared state unchanged. We associate with an action $\pi \rightsquigarrow \pi'$ all state transitions from the following set:

$$\begin{aligned}[\![\pi \rightsquigarrow \pi']\!] = \{((\sigma_s \bullet \sigma''_s, \Sigma_s \bullet \Sigma''_s, \Delta_s \uplus \Delta''_s), (\sigma'_s \bullet \sigma''_s, \Sigma'_s \bullet \Sigma''_s, \Delta'_s \uplus \Delta''_s)) \mid \\ \exists \ell.\, (([\,], [\,], [\,]), (\sigma_s, \Sigma_s, \Delta_s), \ell) \models \boxed{\pi} \wedge (([\,], [\,], [\,]), (\sigma'_s, \Sigma'_s, \Delta'_s), \ell) \models \boxed{\pi'}\}\end{aligned}$$

We give semantics to view assertions with the function $[\![\cdot]\!]$. that is defined as follows:

$$[\![(\pi, \mathcal{R}, \mathcal{G})]\!]_\ell \triangleq (\{(l, s) \mid (l, s, \ell) \models \pi\}, \bigcup\nolimits_{\mathcal{A} \in \mathcal{R}} [\![\mathcal{A}]\!], \bigcup\nolimits_{\mathcal{A} \in \mathcal{G}} [\![\mathcal{A}]\!]).$$

Refined action judgments for RGSep-based logic. Since action judgments are essential for reasoning about primitive commands in our logic, we further refine conditions under which it holds of views from the RGSep-based view monoid.

Proposition 2.38. *The action judgment* $\alpha \Vdash_t \{(P, R, G)\}\{(Q, R, G)\}$ *holds, if it is true that:*

- $\begin{aligned}[t]&\forall \sigma_l, \sigma_s, \sigma'_l, \sigma'_s, \Sigma_l, \Sigma_s, \Delta_l, \Delta_s.\\ &((\sigma_l, \Sigma_l, \Delta_l), (\sigma_s, \Sigma_s, \Delta_s)) \in P \wedge \sigma'_l \bullet \sigma'_s \in [\![\alpha]\!]_t(\sigma_l \bullet \sigma_s) \implies\\ &\quad \exists \Sigma'_l, \Sigma'_s, \Delta'_l, \Delta'_s.\, ((\sigma'_l, \Sigma'_l, \Delta'_l), (\sigma'_s, \Sigma'_s, \Delta'_s)) \in Q \wedge\\ &\quad\quad ((\sigma_s, \Sigma_s, \Delta_s), (\sigma'_s, \Sigma'_s, \Delta'_s)) \in G \wedge\\ &\quad\quad\quad \mathsf{LP}^*(\Sigma_l \bullet \Sigma_s, \Delta_l \uplus \Delta_s, \Sigma'_l \bullet \Sigma'_s, \Delta'_l \uplus \Delta'_s);\end{aligned}$
- $[\![\alpha]\!]_t(\sigma) \neq \lightning \implies \forall \sigma'.\, [\![\alpha]\!]_t(\sigma \bullet \sigma') = \{\sigma'' \bullet \sigma' \mid \sigma'' \in [\![\alpha]\!]_t(\sigma)\}.$

The requirement to primitive commands in Proposition 2.38 is similar to that of the action judgments. The difference is that in the RG-based proof system it is not necessary to require α to preserve any view r of the environment: since a predicate P_r of any view (P_r, R_r, G_r) in another thread is stable under R_r, it is also stable under $G \subseteq R_r$ whenever $(P, R, G) * (P_r, R_r, G_r)$ is defined. Consequently, views of the environment are never invalidated by local transitions. Using the premise of Proposition 2.38 in Prim rule makes it closer to the standard proof rule for the atomic step in Rely/Guarantee.

2.5 Case Study: Flat Combining

In this section, we demonstrate how to reason about algorithms with helping using relational views. We choose a simple library ℓ implementing a concurrent increment and prove its linearizability with the RGSep-based logic.

The concrete library ℓ has one method inc, which increments the value of a shared counter k by the argument of the method. The specification of ℓ is given by an abstract library $\mathcal{L}$. The abstract command, provided by $\mathcal{L}$ as an

implementation of inc, operates with an abstract counter K as follows (assuming that K is initialized by zero):

```
void ℒ(inc, a, r) {
  atomic {
    K := K + a;
    assume(K == r);
  }
}
```

That is, $\mathcal{L}(\mathsf{inc}, a, r)$ atomically increments a counter and a command assume(K == r), which terminates only if the return value r chosen at the invocation equals to the resulting value of K. This corresponds to how we specify methods' return values in §2.3.

In Figure 2.5, we show the pseudo-code of the implementation of a method inc in a C-style language along with a proof outline. The method $\ell(\mathsf{inc}, a, r)$ takes one argument, increments a shared counter k by it and returns the increased value of the counter. Since k is shared among threads, they follow a protocol regulating the access to the counter. This protocol is based on flat combining [32], which is a synchronization technique enabling a parallel execution of sequential operations.

The protocol is the following. When a thread t executes $\ell(\mathsf{inc}, a, r)$, it first makes the argument of the method visible to other threads by storing it in an array arg, and lets res[t] = nil to signal to other threads its intention to execute an increment with that argument. It then spins in the loop on line 8, trying to write its thread identifier into a variable L with a compare-and-swap (CAS). Out of all threads spinning in the loop, the one that succeeds in writing into L becomes a *combiner*: it performs the increments requested by all threads with arguments stored in arg and writes the results into corresponding cells of the array res. The other threads keep spinning and periodically checking the value of their cells in res until a non-nil value appears in it, meaning that a combiner has performed the operation requested and marked it as finished. The protocol relies on the assumption that nil is a value that is never returned by the method. Similarly to the specification of the increment method, the implementation in Figure 2.5 ends with a command assume(res[mytid()] = r).

The proof outline features auxiliary assertions defined in Figure 2.6. In the assertions we let _ denote a value or a logical variable whose name is irrelevant. We assume that each program variable var has a unique location in the heap and denote it with &var. Values a, r and t are used in the formulas and the code as constants.

We prove the following specification for $\ell(\mathsf{inc}, a, r)$:

$$\mathcal{R}_t, \mathcal{G}_t \vdash_t \left\{ \begin{array}{c} \mathsf{global} * M(t) * \\ [\mathsf{todo}(\mathcal{L}(\mathsf{inc}, a, r))]_t \end{array} \right\} \ell(\mathsf{inc}, a, r) \left\{ \begin{array}{c} \mathsf{global} * M(t) * \\ [\mathsf{done}(\mathcal{L}(\mathsf{inc}, a, r))]_t \end{array} \right\}$$

In the specification, $M(t)$ asserts the presence of arg[t] and res[t] in the shared state, and global is an assertion describing the shared state of all the threads. Thus, the pre- and postcondition of the specification differ only by the kind of token given to t.

The main idea of the proof is in allowing a thread t to share the ownership of its token $[\mathsf{todo}(\mathcal{L}(\mathsf{inc}, a, r))]_t$ with the other threads. This enables two possibilities for t. Firstly, t may become a combiner. Then t has a linearization point on

line 17 (when the loop index i equals to t). In this case t also *helps* other concurrent threads by performing their linearization points on line 17 (when $i \neq t$). The alternative possibility is that some other thread becomes a combiner and does a linearization point of t. Thus, the method has a non-fixed linearization point, as it may occur in the code of a different thread.

We further explain how the tokens are transferred. On line 6 the method performs the assignment `res[mytid()] := nil`, signaling to other threads about a task this thread is performing. At this step, the method transfers its token $[\mathsf{todo}(\mathcal{L}(\mathsf{inc}, a, r))]_t$ to the shared state, as represented by the assertion $\boxed{\mathsf{true} * \mathsf{task}_{\mathsf{todo}}(t, a, r)}$. In order to take into consideration other threads interfering with t and possibly helping it, here and further we stabilize the assertion by adding a disjunct $\mathsf{task}_{\mathsf{done}}(t, a, r)$.

If a thread t gets help from other threads, then $\mathsf{task}_{\mathsf{done}}(t, a, r)$ holds, which implies that $\mathtt{res}[t] \neq \mathtt{nil}$ and t cannot enter the loop on line 8. Otherwise, if t becomes a combiner, it transfers $\mathsf{INV}(\_)$ from the shared state to the local state of t to take over the ownership of the counters `k` and `K` and thus ensure that the access to the counter is governed by the mutual exclusion protocol. At each iteration i of the forall loop, `res[i] = nil` implies that $\mathsf{task}_{\mathsf{todo}}(i, \_, \_)$ holds, meaning that there is a token of a thread i in the shared state. Consequently, on line 17 a thread t may use it to perform a linearization point of i.

The actions defining the guarantee relation $\mathcal{G}_t$ of a thread t' are the following:

1. $\&\mathtt{arg}[t] \mapsto \_ * \&\mathtt{res}[t] \not\mapsto \mathtt{nil} \rightsquigarrow \&\mathtt{arg}[t] \mapsto a * \&\mathtt{res}[t] \not\mapsto \mathtt{nil}$;
2. $\&\mathtt{arg}[t] \mapsto a * \&\mathtt{res}[t] \not\mapsto \mathtt{nil} \rightsquigarrow \mathsf{task}_{\mathsf{todo}}(t, a, r)$;
3. $\&\mathtt{L} \mapsto 0 * \mathsf{INV}(\_) \rightsquigarrow \&\mathtt{L} \mapsto t$;
4. $\&\mathtt{L} \mapsto t * \mathsf{task}_{\mathsf{todo}}(T, A, R) \rightsquigarrow \&\mathtt{L} \mapsto t * \mathsf{task}_{\mathsf{done}}(T, A, R)$
5. $\&\mathtt{L} \mapsto t \rightsquigarrow \&\mathtt{L} \mapsto 0 * \mathsf{INV}(\_)$
6. $\mathsf{task}_{\mathsf{done}}(t, a, r) \rightsquigarrow \&\mathtt{arg}[t] \mapsto a * \&\mathtt{res}[t] \mapsto r$

Out of them, conditions 2 and 6 specify transferring the token of a thread t to and from the shared state, and condition 4 describes using the shared token of a thread T. The rely relation of a thread t is then defined as the union of all actions from guarantee relations of other threads and an additional action for each thread $t' \in \mathsf{ThreadID} \setminus \{t\}$ allowing the client to prepare a thread t' for a new method call by giving it a new token: $[\mathsf{done}(\mathcal{L}(\mathsf{inc}, A, R))]_{t'} \rightsquigarrow [\mathsf{todo}(\mathcal{L}(\mathsf{inc}, A', R'))]_{t'}$.

```
1  int L = 0, k = 0, arg[N], res[N]; \\ initially all res[i] ≠ nil
2
3  void ℓ(inc, a, r) {
4    { global ∗ M(t) ∗ [todo(L(inc, a, r))]_t }
5    arg[mytid()] := a;
6    res[mytid()] := nil;
7    { global ∗ [true ∗ (task_todo(t, a, r) ∨ task_done(t, a, r))] }
8    while (res[mytid()] = nil) {
9      if (CAS(&L, 0, mytid())) {
10       { [&L ↦ t ∗ ⊛_{j∈ThreadID} tinv(j)] ∗
           [true ∗ (task_todo(t, a, r) ∨ task_done(t, a, r))] ∗ INV(_) }
11       for (i := 1; i ≤ N; ++i) {
12         { [&L ↦ t ∗ ⊛_{j∈ThreadID} tinv(j)] ∗ INV(_) ∗ LI(i, t, a, r) }
13         if (res[i] = nil) {
14           { ∃V, A, R. INV(V) ∗ LI(i, t, a, r) ∗
               [&L ↦ t ∗ ⊛_{j∈ThreadID} tinv(j)] ∗ [true ∗ task_todo(i, A, R)] }
15           k := k + arg[i];
16           { ∃V, A, R. &k ↦ V + A ∗ &K ⤇ V ∗ LI(i, t, a, r) ∗
               [&L ↦ t ∗ ⊛_{j∈ThreadID} tinv(j)] ∗ [true ∗ task_todo(i, A, R)] }
17           res[i]  := k;
18           { ∃V, A, R. INV(V + A) ∗ LI(i + 1, t, a, r) ∗
               [&L ↦ t ∗ ⊛_{j∈ThreadID} tinv(j)] ∗ [true ∗ task_done(i, A, R)] }
19         }
20       { [&L ↦ t ∗ ⊛_{j∈ThreadID} tinv(j)] ∗ [true ∗ task_done(t, a, r)] ∗ INV(_) }
21       }
22       L = 0;
23     }
24   }
25   assume(res[mytid()] = r);
26   { global ∗ M(t) ∗ [done(L(inc, a, r))]_t }
27 }
```

Figure 2.5: Proof outline for a flat combiner of a concurrent increment.

$$
\begin{array}{lcl}
X \not\mapsto Y & \triangleq & \exists Y'.\, X \mapsto Y' * Y \neq Y' \\
M(t) & \triangleq & \boxed{\mathsf{true} * (\&\mathtt{arg}[t] \mapsto \_ * \&\mathtt{res}[t] \not\mapsto \mathtt{nil})} \\
\mathsf{task}_{\mathsf{todo}}(t,a,r) & \triangleq & \&\mathtt{arg}[t] \mapsto a * \&\mathtt{res}[t] \mapsto \mathtt{nil} * [\mathsf{todo}(\mathcal{L}(\mathsf{inc},a,r))]_t\,; \\
\mathsf{task}_{\mathsf{done}}(t,a,r) & \triangleq & \&\mathtt{arg}[t] \mapsto a * \&\mathtt{res}[t] \mapsto r * r \neq \mathtt{nil} * [\mathsf{done}(\mathcal{L}(\mathsf{inc},a,r))]_t\,; \\
\mathsf{INV}(V) & \triangleq & \&\mathtt{k} \mapsto V * \&\mathtt{K} \rightrightarrows V \\
\mathsf{LI}(i,t,a,r) & \triangleq & \boxed{\begin{array}{l}\mathsf{true} * ((t < i \wedge \mathsf{task}_{\mathsf{done}}(t,a,r)) \vee \\ \quad (t \geq i \wedge (\mathsf{task}_{\mathsf{todo}}(t,a,r) \vee \mathsf{task}_{\mathsf{done}}(t,a,r))))\end{array}} \\
\mathsf{tinv}(i) & \triangleq & \&\mathtt{arg}[i] \mapsto \_ * \&\mathtt{res}[i] \rightrightarrows \_ \vee \mathsf{task}_{\mathsf{todo}}(i,\_,\_) \vee \mathsf{task}_{\mathsf{done}}(i,\_,\_) \\
\mathsf{global} & \triangleq & \boxed{(\&\mathtt{L} \mapsto 0 * \mathsf{INV}(\_) \vee \&\mathtt{L} \not\mapsto 0) * \circledast_{j\in\mathsf{ThreadID}}\, \mathsf{tinv}(j)},
\end{array}
$$

Figure 2.6: Auxiliary predicates. $\circledast_{j\in\mathsf{ThreadID}}\, \mathsf{tinv}(j)$ denotes $\mathsf{tinv}(1) * \mathsf{tinv}(2) * \cdot *$ $\mathsf{tinv}(N)$

2.6 Summary and Related Work

There has been a significant amount of research on methods for proving linearizability. We do not attempt a comprehensive survey here (see [23]) and only describe the most closely related work.

The existing logics for linearizability that use linearization points differ in the thread-modular reasoning method used and, hence, in the range of concurrent algorithms that they can handle. Our goal in this chapter was to propose a uniform basis for designing such logics and to formalize the method they use for reasoning about linearizability in a way independent of the particular thread-modular reasoning method used. We have only shown instantiations of our logic based on disjoint concurrent separation logic [28] and RGSep [24]. However, we expect that our logic can also be instantiated with more complex thread-modular reasoning methods, such as those based on concurrent abstract predicates [29] or islands and protocols [33].

Our notion of tokens is based on the idea of treating method specifications as resources when proving atomicity, which has appeared in various guises in several logics [24, 25, 27]. Our contribution is to formalize this method of handling linearization points independently from the underlying thread-modular reasoning method and to formulate the conditions for soundly combining the two (Definition 2.2, §2.2).

We have presented a logic that unifies the various logics based on linearization points with helping. However, much work still remains as this reasoning method cannot handle all algorithms. Some logics have introduced *speculative* linearization points to increase their applicability [33, 25]; our approach to helping is closely related to this, and we hope could be extended to speculation. But there are still examples beyond this form of reasoning: for instance there are no proofs of the Herlihy-Wing queue [7] using linearization points (with helping and/or speculation). This algorithm can be shown linearizable using forwards/backwards simulation [7] and more recently has been shown to only require a backwards simulation [34]. But integrating this form of simulation with the more intricate notions of interference expressible in the Views framework remains an open problem. In Chapter 3, we propose a method of using partial orders in forward simulation proofs of linearizability that is applicable to the Herlihy-Wing queue. We conjecture that the Views framework can be extended to support the latter style of reasoning.

Another approach to proving linearizability is the aspect-oriented method. This gives a series of properties of a queue [35] (or a stack [36]) implementation which imply that the implementation is linearizable. This method been applied to algorithms that cannot be handled with standard linearization-point-based methods. However, the aspect-oriented approach requires a custom theorem per data structure, which limits its applicability.

In this chapter we concentrated on linearizability in its original form [7], which considers only finite computations and, hence, specifies only safety properties of the library. Linearizability has since been generalized to also specify liveness properties [37]. Another direction of future work is to generalize our logic to handle liveness, possibly building on ideas from [38].

When a library is linearizable, one can use its atomic specification instead of the actual implementation to reason about its clients [8]. Some logics achieve the same effect without using linearizability, by expressing library specifications as

judgments in the logic rather than as the code of an abstract library [39, 40, 41]. It is an interesting direction of future work to determine a precise relationship between this method of specification and linearizability, and to propose a generic logic unifying the two.

2.6.1 Summary

We have presented a logic for proving the linearizability of concurrent libraries that can be instantiated with different methods for thread-modular reasoning. To this end, we have extended the Views framework [17] to reason about relations between programs. Our main technical contribution in this regard was to propose the requirement for axiom soundness (Definition 2.2, §2.2) that ensures a correct interaction between the treatment of linearization points and the underlying thread-modular reasoning. We have shown that our logic is powerful enough to handle concurrent algorithms with challenging features, such as helping. More generally, our work marks the first step towards unifying the logics for proving relational properties of concurrent programs.

Chapter 3

Proving Linearizability Using Partial Orders

Linearizability is a commonly accepted notion of correctness of concurrent data structures. It matters for programmers using such data structures because it implies *contextual refinement*: any behavior of a program using a concurrent data structure can be reproduced if the program uses its sequential implementation where all operations are executed atomically [8]. This allows the programmer to soundly reason about the behavior of the program assuming a simple sequential specification of the data structure. Linearizability requires that for any execution of operations on the data structure there exists a linear order of these operations, called a *linearization*, such that: *(i)* the linearization respects the order of non-overlapping operations (the *real-time order*); and *(ii)* the behavior of operations in the linearization matches the sequential specification of the data structure. To illustrate this, consider an execution in Figure 3.1, where three threads are accessing a queue. Linearizability determines which values x the dequeue operation is allowed to return by considering the possible linearizations of this execution. Given *(i)*, we know that in any linearization the enqueues must be ordered before the dequeue, and Enq(1) must be ordered before Enq(3). Given *(ii)*, a linearization must satisfy the sequential specification of a queue, so the dequeue must return the oldest enqueued value. Hence, the execution in Figure 3.1 has three possible linearizations: [Enq(1); Enq(2); Enq(3); Deq():1], [Enq(1); Enq(3); Enq(2); Deq():1] and [Enq(2); Enq(1); Enq(3); Deq():2]. This means that the dequeue is allowed to return 1 or 2, but not 3.

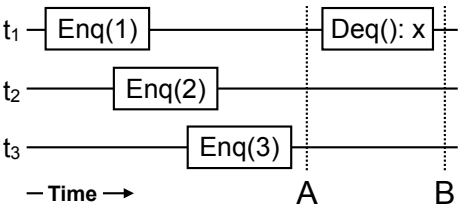


Figure 3.1: Example execution.

For a large class of algorithms, linearizability can be proved by incrementally constructing a linearization as the program executes. Effectively, one shows that the program execution and its linearization stay in correspondence under each program step (this is formally known as a *forward simulation*). The point in the execution of an operation at which it is appended to the linearization is called

its *linearization point*. This must occur somewhere between the start and end of the operation, to ensure that the linearization preserves the real-time order. For example, when applying the linearization point method to the execution in Figure 3.1, by point (A) we must have decided if Enq(1) occurs before or after Enq(2) in the linearization. Thus, by this point, we know which of the three possible linearizations matches the execution. This method of establishing linearizability is very popular, to the extent that most papers proposing new concurrent data structures include a placement of linearization points. However, there are algorithms that cannot be proved linearizable using the linearization point method.

In this chapter we consider several examples of such algorithms, including the *time-stamped (TS) queue* [18, 36]—a recent high-performance data structure with an extremely subtle correctness argument. Its key idea is for enqueues to attach timestamps to values, and for these to determine the order in which values are dequeued. As illustrated by the above analysis of Figure 3.1, linearizability allows concurrent operations, such as Enq(1) and Enq(2), to take effect in any order. The TS queue exploits this by allowing values from concurrent enqueues to receive incomparable timestamps; only pairs of timestamps for non-overlapping enqueue operations must be ordered. Hence, a dequeue can potentially have a choice of the "earliest" enqueue to take values from. This allows concurrent dequeues to go after different values, thus reducing contention and improving performance.

The linearization point method simply does not apply to the TS queue. In the execution in Figure 3.1, values 1 and 2 could receive incomparable timestamps. Thus, at point (A) we do not know which of them will be dequeued first and, hence, in which order their enqueues should go in the linearization: this is only determined by the behavior of dequeues later in the execution. Similar challenges exist for other queue algorithms such as the baskets queue [42], LCR queue [43] and Herlihy-Wing queue [7]. In all of these algorithms, when an enqueue operation returns, the precise linearization of earlier enqueue operations is not necessarily known.

In this chapter, we propose a new proof method that can handle algorithms where incremental construction of linearizations is not possible. We formalize it as a program logic, based on Rely-Guarantee [16], and apply it to give simple proofs to the TS queue [36], the Herlihy-Wing queue [7] and the Optimistic Set [19]. The key idea of our method is to incrementally construct not a single linearization of an algorithm execution, but an *abstract history*—a partially ordered history of operations such that it contains the real-time order of the original execution and *all* its linearizations satisfy the sequential specification. By embracing partiality, we enable decisions about order to be delayed, mirroring the behavior of the algorithms. At the same time, we maintain the simple inductive style of the standard linearization-point method: the proof of linearizability of an algorithm establishes a simulation between its execution and a growing abstract history. By analogy with linearization points, we call the points in the execution where the abstract history is extended *commitment points*.

Consider again the TS queue execution in Figure 3.1. By point (A) we construct the abstract history in Figure 3.2(a). The edge in the figure is mandated by the real-time order in the original execution; Enq(1) and Enq(2) are left unordered, and so are Enq(2) and Enq(3). At the start of the execution of the

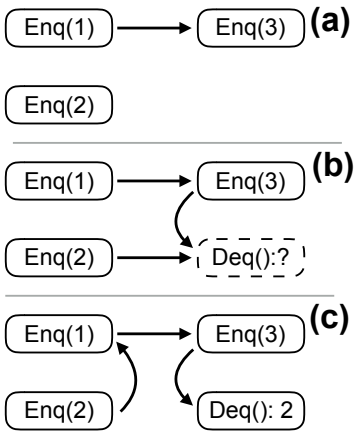


Figure 3.2: Abstract histories constructed for prefixes of the execution in Figure 3.1: (a) is at point (A); (b) is at the start of the dequeue operation; and (c) is at point (B). We omit the transitive consequences of the edges shown.

dequeue, we update the history to the one in Figure 3.2(b). A dashed ellipse represents an operation that is not yet completed, but we have committed to performing it (case 1 above). When the dequeue successfully removes a value, e.g., 2, we update the history to the one in Figure 3.2(c). To this end, we complete the dequeue by recording its result (case 3). We also commit to an order between the Enq(1) and Enq(2) operations (case 2). This is needed to ensure that all linearizations of the resulting history satisfy the sequential queue specification, which requires a dequeue to remove the oldest value in the queue.

We demonstrate the simplicity of our method by giving proofs to challenging algorithms that match the intuition for why they work. Our method is also similar in spirit to the standard linearization point method. Thus, even though in this chapter we formulate the method as a program logic, we believe that algorithm designers can also benefit from it in informal reasoning, using abstract histories and commitment points instead of single linearizations and linearization points.

3.1 Linearizability, Abstract Histories and Commitment Points

Preliminaries. We consider a data structure that can be accessed concurrently via operations $\mathsf{op} \in \mathsf{Op}$ in several threads, identified by $t \in \mathsf{ThreadID}$. Each operation takes one argument and returns one value, both from a set Val; we use a special value $\bot \in \mathsf{Val}$ to model operations that take no argument or return no value. Linearizability relates the observable behavior of an implementation of such a concurrent data structure to its sequential specification [7]. We formalize both of these by sets of *histories*, which are partially ordered sets of *events*, recording operations invoked on the data structure. Formally, an event is of the form $e = [i : (t, \mathsf{op}, a, r)]$. It includes a unique identifier $i \in \mathsf{EventID}$ and records an operation $\mathsf{op} \in \mathsf{Op}$ called by a thread $t \in \mathsf{ThreadID}$ with an argument $a \in \mathsf{Val}$,

which returns a value $r \in \mathsf{Val} \uplus \{\mathsf{todo}\}$. We use the special return value todo for events describing operations that have not yet terminated, and call such events *uncompleted*. We denote the set of all events by Event. Given a set $E \subseteq \mathsf{Event}$, we write $E(i) = (t, \mathsf{op}, a, r)$ if $[i : (t, \mathsf{op}, a, r)] \in E$ and let $\lfloor E \rfloor$ consist of all completed events from E. We let $\mathsf{id}(E)$ denote the set of all identifiers of events from E. Given an event identifier i, we also use $E(i).\mathsf{tid}$, $E(i).\mathsf{op}$, $E(i).\mathsf{arg}$ and $E(i).\mathsf{rval}$ to refer to the corresponding components of the tuple $E(i)$.

Definition 3.1. *A* history[1] *is a pair $H = (E, R)$, where $E \subseteq \mathsf{Event}$ is a finite set of events with distinct identifiers and $R \subseteq \mathsf{id}(E) \times \mathsf{id}(E)$ is a strict partial order (i.e., transitive and irreflexive), called the* real-time order*. We require that for each $t \in \mathsf{ThreadID}$:*

- *events in t are totally ordered by R:*
  $\forall i, j \in \mathsf{id}(E).\, i \neq j \wedge E(i).\mathsf{tid} = E(j).\mathsf{tid} = t \implies (i \xrightarrow{R} j \vee j \xrightarrow{R} i)$;
- *only maximal events in R can be uncompleted:*
  $\forall i \in \mathsf{id}(E).\, \forall t \in \mathsf{ThreadID}.\, E(i).\mathsf{rval} = \mathsf{todo} \implies \neg \exists j \in \mathsf{id}(E).\, i \xrightarrow{R} j$;
- *R is an interval order:*
  $\forall i_1, i_2, i_3, i_4.\, i_1 \xrightarrow{R} i_2 \wedge i_3 \xrightarrow{R} i_4 \implies i_1 \xrightarrow{R} i_4 \vee i_2 \xrightarrow{R} i_3$.

We let $\mathsf{History}$ be the set of all histories. A history (E, R) is sequential*, written $\mathsf{seq}(E, R)$, if $\mathsf{id}(E) = \lfloor E \rfloor$ and R is total on E.*

Informally, $i \xrightarrow{R} j$ means that the operation recorded by $E(i)$ completed before the one recorded by $E(j)$ started. The real-time order in histories produced by concurrent data structure implementations may be partial, since in this case the execution of operations may overlap in time; in contrast, specifications are defined using sequential histories, where the real-time order is total.

Linearizability. Assume we are given a set of histories that can be produced by a given data structure implementation (we introduce a programming language for implementations and formally define the set of histories an implementation produces in §3.4). Linearizability requires all of these histories to be matched by a similar history of the data structure specification (its *linearization*) that, in particular, preserves the real-time order between events in the following sense: the real-time order of a history $H = (E, R)$ is *preserved* in a history $H' = (E', R')$, written $H \sqsubseteq H'$, if $E = E'$ and $R \subseteq R'$.

The full definition of linearizability is slightly more complicated due to the need to handle uncompleted events: since operations they denote have not terminated, we do not know whether they have made a change to the data structure or not. To account for this, the definition makes all events in the implementation history complete by discarding some uncompleted events and completing the remaining ones with an arbitrary return value. Formally, an event $e = [i : (t, \mathsf{op}, a, r)]$ *can be completed* to an event $e' = [i' : (t', \mathsf{op}', a', r')]$, written $e \trianglelefteq e'$, if $i = i'$, $t = t'$, $\mathsf{op} = \mathsf{op}'$, $a = a'$ and either $r = r' \neq \mathsf{todo}$ or

[1] For technical convenience, our notion of a history is different from the one in the classical linearizability definition [7], which uses separate events to denote the start and the end of an operation. We require that R be an interval order, we ensure that our notion is consistent with an interpretation of events as segments of time during which the corresponding operations are executed, with R ordering i_1 before i_2 if i_1 finishes before i_2 starts [44].

$r' = \mathsf{todo}$. A history $H = (E, R)$ *can be completed* to a history $H' = (E', R')$, written $H \trianglelefteq H'$, if $\mathsf{id}(E') \subseteq \mathsf{id}(E)$, $\lfloor E \rfloor \subseteq \lfloor E' \rfloor$, $R \cap (\mathsf{id}(E') \times \mathsf{id}(E')) = R'$ and $\forall i \in \mathsf{id}(E').\, [i : E(i)] \trianglelefteq [i : E'(i)]$.

Definition 3.2. *A set of histories $\mathcal{H}_1$ (defining the data structure implementation) is* linearized *by a set of sequential histories $\mathcal{H}_2$ (defining its specification), written $\mathcal{H}_1 \sqsubseteq \mathcal{H}_2$, if $\forall H_1 \in \mathcal{H}_1.\, \exists H_2 \in \mathcal{H}_2.\, \exists H_1'.\, H_1 \trianglelefteq H_1' \wedge H_1' \sqsubseteq H_2$.*

Let $\mathcal{H}_{\mathsf{queue}}$ be the set of sequential histories defining the behavior of a queue with $\mathsf{Op} = \{\mathrm{Enq}, \mathrm{Deq}\}$. We defer its formal definition until §3.4, but for example, [Enq(2); Enq(1); Enq(3); Deq():2] $\in \mathcal{H}_{\mathsf{queue}}$ and [Enq(1); Enq(2); Enq(3); Deq():2] $\notin \mathcal{H}_{\mathsf{queue}}$.

Proof method. In general, a history of a data structure (H_1 in Definition 3.2) may have multiple linearizations (H_2) satisfying a given specification $\mathcal{H}$. In our proof method, we use this observation and construct a partially ordered history, an *abstract history*, all linearizations of which belong to $\mathcal{H}$.

Definition 3.3. *A history H is an* abstract history *of a specification given by the set of sequential histories $\mathcal{H}$ if $\{H' \mid \lfloor H \rfloor \sqsubseteq H' \wedge \mathsf{seq}(H')\} \subseteq \mathcal{H}$, where $\lfloor (E, R) \rfloor = (\lfloor E \rfloor, R \cap (\mathsf{id}(\lfloor E \rfloor) \times \mathsf{id}(\lfloor E \rfloor)))$. We denote this by $\mathsf{abs}(H, \mathcal{H})$.*

We define the construction of an abstract history $H = (E, R)$ by instrumenting the data structure operations with auxiliary code that updates the history at certain *commitment points* during operation execution. There are three kinds of commitment points:

1. When an operation op with an argument a starts executing in a thread t, we extend E by a fresh event $[i : (t, \mathsf{op}, a, \mathsf{todo})]$, which we order in R after all events in $\lfloor E \rfloor$.

2. At any time, we can add more edges to R.

3. By the time an operation finishes, we have to assign its return value to its event in E.

Note that, unlike Definition 3.2, Definition 3.3 uses a particular way of completing an abstract history H, which just discards all uncompleted events using $\lfloor - \rfloor$. This does not limit generality because, when constructing an abstract history, we can complete an event (item 3) right after the corresponding operation makes a change to the data structure, without waiting for the operation to finish.

In §3.5 we formalize our proof method as a program logic and show that it indeed establishes linearizability. Before this, we demonstrate informally how the obligations of our proof method are discharged on an example.

3.2 Running Example: the Time-Stamped Queue

We use the TS queue [18] as our running example. Values in the queue are stored in per-thread single-producer (SP) multi-consumer pools, and we begin by describing this auxiliary data structure.

```
1  PoolID insert(ThreadID t, Val v) {
2    p := new PoolID();
3    pools(t) := pools(t) · (p, v, ⊤);
4    return p;
5  }
6
7  setTimestamp(ThreadID t, PoolID p, TS τ) {
8    if (∃Σ, Σ′, v. pools(t) = Σ · (p, v, _) · Σ′)
9      pools(t) := Σ · (p, v, τ) · Σ′;
10 }
11
12 Val remove(ThreadID t, PoolID p) {
13   if (∃Σ, Σ′, v, τ. pools(t) = Σ · (p, v, τ) · Σ′) {
14     pools(t) := Σ · Σ′;
15     return v;
16   } else return NULL;
17 }
18
19 (PoolID × TS) getOldest(ThreadID t) {
20   if (∃p, τ. pools(t) = (p, _, τ) · _)
21     return (p, τ);
22   else
23     return (NULL, NULL);
24 }
```

Figure 3.3: Operations on abstract SP pools `pools` : ThreadID → Pool. All operations are atomic.

SP pools. SP pools have well-known linearizable implementations [18], so we simplify our presentation by using abstract pools with the atomic operations given in Figure 3.3. This does not limit generality: since linearizability implies contextual refinement [8] properties proved using the abstract pools will stay valid for their linearizable implementations. In the figure and in the following we denote irrelevant expressions by _.

The SP pool of a thread contains a sequence of triples (p, v, τ), each consisting of a unique identifier $p \in \mathsf{PoolID}$, a value $v \in \mathsf{Val}$ enqueued into the TS queue by the thread and the associated timestamp $\tau \in \mathsf{TS}$. The set of timestamps TS is partially ordered by $<_{\mathsf{TS}}$, with a distinguished timestamp $\top$ that is greater than all others. We let `pool` be the set of states of an abstract SP pool. Initially all pools are empty. The operations on SP pools are as follows:

- `insert(t,v)` appends a value `v` to the back of the pool of thread `t` and associates it with the special timestamp $\top$; it returns an identifier for the added element.
- `setTimestamp(t,p,`τ`)` sets to τ the timestamp of the element identified by `p` in the pool of thread `t`.
- `getOldest(t)` returns the identifier and timestamp of the value from the front of the pool of thread `t`, or (NULL, NULL) if the pool is empty.
- `remove(t,p)` tries to remove a value identified by `p` from the pool of thread `t`. Note this can fail if some other thread removes the value first.

```
25 enqueue(Val v) {
26   atomic {
27     PoolID node := insert(myTid(), v);
28     G_ts[myEid()] := ⊤;
29   }
30   TS timestamp := newTimestamp();
31   atomic {
32     setTimestamp(myTid(), node, timestamp);
33     G_ts[myEid()] := timestamp;
34     E(myEid()).rval := ⊥;
35   }
36   return ⊥;
37 }
```

Figure 3.4: The TS queue: enqueue. Shaded portions are auxiliary code used in the proof.

Separating `insert` from `setTimestamp` and `getOldest` from `remove` in the SP pool interface reduces the atomicity granularity, and permits more efficient implementations.

Core TS queue algorithm. Figures 3.4 and 3.5 give the code for our version of the TS queue. Shaded portions are auxiliary code needed in the linearizability proof to update the abstract history at commitment points; it can be ignored for now. In the overall TS queue, enqueuing means adding a value with a certain timestamp to the pool of the current thread, while dequeuing means searching for the value with the minimal timestamp across per-thread pools and removing it.

In more detail, the `enqueue`(v) operation first inserts the value v into the pool of the current thread, defined by `myTid` (line 27). At this point the value v has the default, maximal timestamp $\top$. The code then generates a new timestamp using `newTimestamp` and sets the timestamp of the new value to it (lines 30-32). We describe an implementation of `newTimestamp` later in this section. The key property that it ensures is that out of two non-overlapping calls to this function, the latter returns a higher timestamp than the former; only concurrent calls may generate incomparable timestamps. Hence, timestamps in each pool appear in the ascending order.

The `dequeue` operation first generates a timestamp `start_ts` at line 42, which it further uses to determine a consistent snapshot of the data structure. After generating `start_ts`, the operation iterates through per-thread pools, searching for a value with a minimal timestamp (lines 46–56). The search starts from a random pool, to make different threads more likely to pick different elements for removal and thus reduce contention. The pool identifier of the current candidate for removal is stored in `cand_pid`, its timestamp in `cand_ts` and the thread that inserted it in `cand_tid`. On each iteration of the loop, the code fetches the earliest value enqueued by thread `k` (line 48) and checks whether its timestamp is smaller than the current candidate's `cand_ts` (line 52). If the timestamps are incomparable, the algorithm keeps the first one (either would be legitimate). Additionally, the algorithm never chooses a value as a candidate if its timestamp is greater than `start_ts`, because such values are not guaranteed to be read in a consistent manner.

```
38 Val dequeue() {
39   Val ret := NULL;
40   EventID CAND;
41   do {
42     TS start_ts := newTimestamp();
43     PoolID pid, cand_pid := NULL;
44     TS ts, cand_ts := ⊤;
45     ThreadID cand_tid;
46     for each k in 1..NThreads do {
47       atomic {
48         (pid, ts) := getOldest(k);
49         R := (R ∪ {(e, myEid()) | e ∈ id(⌊E⌋) ∩ inQ(pools, E, G_ts)
50                                   ∧ ¬(start_ts <_TS G_ts(e))})^+;
51       }
52       if (pid ≠ NULL && ts <_TS cand_ts && ¬(start_ts <_TS ts)) {
53         (cand_pid, cand_ts, cand_tid) := (pid, ts, k);
54         CAND := enqOf(E, G_ts, cand_tid, cand_ts);
55       }
56     }
57     if (cand_pid ≠ NULL)
58       atomic {
59         ret := remove(cand_tid, cand_pid);
60         if (ret ≠ NULL) {
61           E(myEid()).rval := ret;
62           R := (R ∪ {(CAND, e) | e ∈ inQ(pools, E, G_ts)}
63                   ∪ {(myEid(), d) | E(d).op = Deq ∧ d ∈ id(E \ ⌊E⌋)})^+;
64         }
65       }
66   } while (ret = NULL);
67   return ret;
68 }
```

Figure 3.5: The TS queue: dequeue. Shaded portions are auxiliary code used in the proof.

If a candidate has been chosen once the iteration has completed, the code tries to remove it (line 58). This may fail if some other thread got there first, in which case the operation restarts. Likewise, the algorithm restarts if no candidate was identified (the full algorithm in [18] includes an emptiness check, which we omit for simplicity).

Timestamp generation. The TS queue requires that sequential calls to `newTimestamp` generate ordered timestamps. This ensures that the two sequentially enqueued values cannot be dequeued out of order. However, concurrent calls to `newTimestamp` may generate incomparable timestamps. This is desirable because it increases flexibility in choosing which value to dequeue, reducing contention.

There are a number of implementations of `newTimestamp` satisfying the above requirements [36]. For concreteness, we consider the implementation given in Figure 3.6. Here a timestamp is either $\top$ or a pair of integers (b, e), representing a time interval. In every timestamp (b, e), $b \leq e$. Two timestamps are considered ordered $(b_1, e_1) <_{\mathsf{TS}} (b_2, e_2)$ if $e_1 < b_2$, i.e., if the time intervals do

```
69 int counter = 1;
70
71 TS newTimestamp() {
72   int ts = counter;
73   TS result;
74   if (CAS(counter, ts, ts+1))
75     result = (ts, ts);
76   else
77     result = (ts, counter-1);
78   return result;
79 }
```

Figure 3.6: Timestamp generation algorithm.

not overlap. Intervals are generated with the help of a shared `counter`. The algorithm reads the counter as the start of the interval and attempts to atomically increment it with a CAS (lines 72-74), which is a well-known atomic compare-and-swap operation. It atomically reads the counter and, if it still contains the previously read value `ts`, updates it with the new timestamp $\texttt{ts}+1$ and returns true; otherwise, it does nothing and returns false. If CAS succeeds, then the algorithm takes the interval start and end values as equal (line 75). If not, some other thread(s) increased the counter. The algorithm reads the counter again and subtracts 1 to give the end of the interval (line 77). Thus, either the current call to `newTimestamp` increases the counter, or some other thread does so. In either case, subsequent calls will generate timestamps greater than the current one.

This timestamping algorithm allows concurrent enqueue operations in Figure 3.1 to get incomparable timestamps. Then the dequeue may remove either 1 or 2 depending on where it starts traversing the pools[2] (line 46). As we explained in the beginning of the chapter, this makes the standard method of linearization point inapplicable for verifying the TS queue.

3.3 The TS Queue: Informal Development

In this section we explain how the abstract history is updated at the commitment points of the TS Queue and justify informally why these updates preserve the key property of this history—that all its linearizations satisfy the sequential queue specification. We present the details of the proof of the TS queue in §3.6.

Ghost state and auxiliary definitions. To aid in constructing the abstract history (E, R), we instrument the code of the algorithm to maintain a piece of ghost state—a partial function $G_{\mathsf{ts}} : \mathsf{EventID} \rightharpoonup \mathsf{TS}$. Given the identifier i of an event $E(i)$ denoting an `enqueue` that has inserted its value into a pool, $G_{\mathsf{ts}}(i)$ gives the timestamp currently associated with the value. The statements in lines 28 and 33 in Figure 3.4 update G_{ts} accordingly. These statements use a special command `myEid`() that returns the identifier of the event associated with the current operation.

As explained in §3.2, the timestamps of values in each pool appear in strictly ascending order. As a consequence, all timestamps assigned by G_{ts} to events of

[2]Recall that the randomness is required to reduce contention

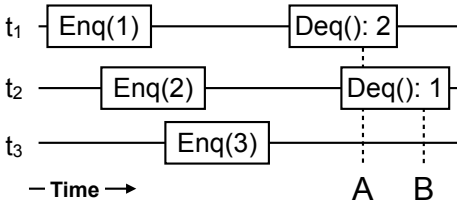


Figure 3.7: Example execution extending Figure 3.1. Dotted lines indicate commitment points at lines 58–65 of the dequeues.

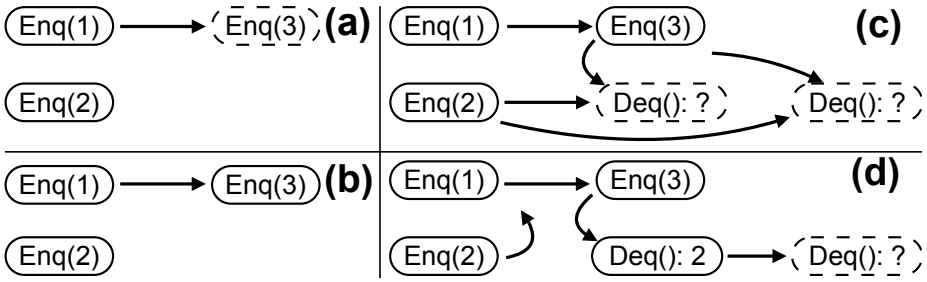


Figure 3.8: Changes to the abstract history of the execution in Figure 3.7.

a given thread t are distinct, which is formalized by the following property:

$$\forall i, j.\, i \neq j \wedge E(i).\mathsf{tid} = E(j).\mathsf{tid} \wedge i, j \in \mathsf{dom}(G_{\mathsf{ts}}) \implies G_{\mathsf{ts}}(i) \neq G_{\mathsf{ts}}(j)$$

Hence, for a given thread t and a timestamp τ, there is at most one enqueue event in E that inserted a value with the timestamp τ in the pool of a thread t. In the following, we denote the identifier of this event by $\mathtt{enqOf}(E, G_{\mathsf{ts}}, t, \tau)$ and let the set of the identifiers of such events for all values currently in the pools be $\mathsf{inQ}(\mathtt{pools}, E, G_{\mathsf{ts}})$:

$$\mathsf{inQ}(\mathtt{pools}, E, G_{\mathsf{ts}}) \triangleq \{\mathtt{enqOf}(E, G_{\mathsf{ts}}, t, \tau) \mid \exists p.\, \mathtt{pools}(t) = \_ \cdot (p, \_, \tau) \cdot \_\}$$

Commitment points and history updates. We further instrument the code with statements that update the abstract history at commitment points, which we now explain. As a running example, we use the execution in Figure 3.7, extending that in Figure 3.1. As we noted in §3.1, when an operations starts, we automatically add a new uncompleted event to E to represent this operation and order it after all completed events in R. For example, before the start of Enq(3) in the execution of Figure 3.7, the abstract history contains two events Enq(1) and Enq(2) and no edges in the real-time order. At the start of Enq(3) the history gets transformed to that in Figure 3.8(a). The commitment point at line 32 in Figure 3.4 completes the enqueue by giving it a return value $\perp$, which results in the abstract history in Figure 3.8(b).

Upon a dequeue's start, we similarly add an event representing it. Thus, by point (A) in Figure 3.7, the abstract history is as shown in Figure 3.8(c). At every iteration `k` of the loop, the dequeue performs a commitment point at lines 49–50, where we order enqueue events of values currently present in the pool of a thread `k` before the current dequeue event. Specifically, we add an edge $(e, \mathtt{myEid}())$ for each identifier e of an enqueue event whose value is in the `k`'s pool and whose timestamp is not greater than the dequeue's own timestamp `start_ts`. Such ordering ensures that in all linearizations of the abstract

history, the values that the current dequeue observes in the pool according to the algorithm are also enqueued in the sequential queue prior to the dequeue. In particular, this also ensures that in all linearizations, the dequeue returns a value that has already been inserted.

The key commitment point in dequeue occurs in lines 58–65, where the abstract history is updated if the dequeue successfully removes a value from a pool. The ghost code at line 54 stores the event identifier for the enqueue that inserted this value in `CAND`. At the commitment point we first complete the current dequeue event by assigning the value removed from a pool as its return value. This ensures that the dequeue returns the same value in the concrete execution and the abstract history. Finally, we order events in the abstract history to ensure that all linearizations of the abstract history satisfy the sequential queue specification. To this end, we add the following edges to R and then transitively close it:

1. $(\texttt{CAND}, e)$ for each identifier e of an enqueue event whose value is still in the pools. This ensures that the dequeue removes the oldest value in the queue.

2. $(\texttt{myEid}(), d)$ for each identifier d of an uncompleted dequeue event. This ensures that dequeues occur in the same order as they remove values from the queue.

At the commitment point (A) in Figure 3.7 the abstract history gets transformed from the one in Figure 3.8(c) to the one in Figure 3.8(d).

3.4 Programming Language

To formalize our proof method, we first introduce a programming language for data structure implementations. This defines such implementations by functions $D : \mathsf{Op} \to \mathsf{Com}$ mapping operations to *commands* from a set Com. The commands, ranged over by C, are written in a simple while-language, which we further define.

Operation syntax. Data structures implement every operation $\mathsf{op} \in \mathsf{Op}$ as *sequential commands* with the following syntax:

$$C \in \mathsf{Com} ::= \alpha \mid C\,;C \mid C + C \mid C^* \mid \mathsf{skip}, \quad \text{where } \alpha \in \mathsf{PCom}$$

The grammar includes *primitive commands* α from a set PCom (assignment, CAS, etc.) and standard control-flow constructs: sequential composition $C\,;C$, non-deterministic choice $C + C$, finite iteration C^* (we are interested only in terminating executions) and a termination marker skip. We use $+$ and $(\ )^*$ instead of conditionals and while loops for theoretical simplicity: as we show further, given appropriate primitive commands conditionals and loops can be encoded.

Operations semantics. Let $\mathsf{Loc} \subseteq \mathsf{Val}$ be the set of all memory locations. We let $\mathsf{State} = \mathsf{Loc} \to \mathsf{Val}$ be the set of all states of the data structure implementation, ranged over by s. Recall from §3.1 that operations of a data structure

$$\rightarrowtail \subseteq \mathsf{Com} \times \mathsf{PCom} \times \mathsf{Com} :$$

$$\frac{C_1 \rightarrowtail_\alpha C_1'}{C_1' ; C_2 \rightarrowtail_\alpha C_1' ; C_2} \qquad \frac{}{C^* \rightarrowtail_{\mathsf{id}} C; C^*} \qquad \frac{}{\alpha \rightarrowtail_\alpha \mathsf{skip}}$$

$$\frac{}{\mathsf{skip} ; C \rightarrowtail_{\mathsf{id}} C} \qquad \frac{}{C^* \rightarrowtail_{\mathsf{id}} \mathsf{skip}} \qquad \frac{i \in \{1,2\}}{C_1 + C_2 \rightarrowtail_{\mathsf{id}} C_i}$$

$$\longrightarrow \subseteq (\mathsf{Com} \times \mathsf{State}) \times \mathsf{ThreadID} \times (\mathsf{Com} \times \mathsf{State}) :$$

$$\frac{s' \in [\![\alpha]\!]_t(s) \quad C \rightarrowtail_\alpha C'}{\langle C, s\rangle \longrightarrow_t \langle C', s'\rangle}$$

Figure 3.9: The operational semantics of sequential commands

can be called concurrently in multiple threads from ThreadID, and thats tates are shared among threads from ThreadID. For every thread t, we use distinguished locations $\mathtt{arg}[t], \mathtt{res}[t] \in \mathsf{Loc}$ to store an argument, respectively, the return value of an operation called in this thread.

We assume the semantics of each atomic command $\alpha \in \mathsf{PCom}$ given by a non-deterministic state transformers $[\![\alpha]\!]_t : \mathsf{State} \to \mathcal{P}(\mathsf{State})$, $t \in \mathsf{ThreadID}$. For a state s, $[\![\alpha]\!]_t(s)$ is the set of states resulting from thread t executing α atomically in s. We also assume a primitive command $\mathsf{id} \in \mathsf{PCom}$ with the interpretation $[\![\mathsf{id}]\!]_t(s) \triangleq \{s\}$.

State transformers may have different semantics depending on a thread identifier, which we use to restrict access to thread-local memory locations such as $\mathtt{arg}[t]$ and $\mathtt{res}[t]$ for each thread t.

We lift the semantics of state transformers to a sequential small-step operational semantics of arbitrary commands from Com. Figure 3.9 gives selected rules of operational semantics; $\langle C, s\rangle \longrightarrow_t \langle C', s'\rangle$ indicates a transition from C to C' by performing a primitive command α in a thread t that updates the state from s to s'. The rules of the operational semantics are standard.

Let us show how to define traditional control flow primitives, such as an if-statement and a while-loop, in our programming language. Assuming a language for arithmetic expressions, ranged over by $\mathcal{E}$, and a function $[\![\mathcal{E}]\!]_s$ that evaluates expressions in a given state s, we define a primitive command $\mathtt{assume}(\mathcal{E})$ that acts as a filter on states, choosing only those where $\mathcal{E}$ evaluates to non-zero values.

$$[\![\mathtt{assume}(\mathcal{E})]\!]_t(s) \triangleq (\text{if } [\![\mathcal{E}]\!]_s \neq 0 \text{ then } \{s\} \text{ else } \emptyset).$$

Using $\mathtt{assume}(\mathcal{E})$ and the C-style negation $!\mathcal{E}$ in expressions, a conditional and a while-loop can be implemented as the following commands:

$$\mathtt{if}\ \mathcal{E}\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2 \triangleq (\mathtt{assume}(\mathcal{E}); C_1) + (\mathtt{assume}(!\mathcal{E}); C_2)$$

$$\mathtt{while}\ \mathcal{E}\ \mathtt{do}\ C \triangleq (\mathtt{assume}(\mathcal{E}); C)^*; \mathtt{assume}(!\mathcal{E})$$

Data structure histories. We now define the set of histories produced by a data structure implementation D, which is required by the definition of linearizability (Definition 3.2, §3.1). Informally, these are the histories produced by threads repeatedly invoking data structure operations in any order and with any possible arguments (this can be thought of as running the data structure implementation under its *most general client* [31]). We define this formally using a concurrent small-step semantics of the data structure D that also constructs corresponding histories: $\twoheadrightarrow_D \subseteq (\mathsf{Cont} \times \mathsf{State} \times \mathsf{History})^2$, where

$\mathsf{Cont} = \mathsf{ThreadID} \to (\mathsf{Com} \uplus \{\mathsf{idle}\})$. Here a function $\tau \in \mathsf{Cont}$ characterises the progress of an operation execution in each thread t: $\tau(t)$ gives the continuation of the code of the operation executing in thread t, or idle if no operation is executing. The relation $\twoheadrightarrow_D$ defines how a step of an operation in some thread transforms the data structure state and the history:

$$\frac{i \notin \mathsf{id}(E) \quad a \in \mathsf{Val} \quad E' = E[i : (t, \mathsf{op}, a, \mathsf{todo})] \quad R' = R \cup \{(j, i) \mid j \in \lfloor E \rfloor\}}{\langle \tau[t : \mathsf{idle}], s, (E, R)\rangle \twoheadrightarrow_D \langle \tau[t : D(\mathsf{op})], s[\mathtt{arg}[t] : a], (E', R')\rangle}$$

$$\frac{\langle C, s\rangle \longrightarrow_t \langle C', s'\rangle}{\langle \tau[t : C], s, (E, R)\rangle \twoheadrightarrow_D \langle \tau[t : C'], s', (E, R)\rangle}$$

$$\frac{i = \mathsf{last}(t, (E, R)) \quad E(i) = (t, \mathsf{op}, a, \mathsf{todo}) \quad E' = E[i : (t, \mathsf{op}, a, s(\mathtt{res}[t]))]}{\langle \tau[t : \mathsf{skip}], s, (E, R)\rangle \twoheadrightarrow_D \langle \tau[t : \mathsf{idle}], s, (E', R)\rangle}$$

First, an idle thread t may call any operation $\mathsf{op} \in \mathsf{Op}$ with any argument a. This sets the continuation of thread t to $D(\mathsf{op})$, stores a into $\mathtt{arg}[t]$ and adds a new event i to the history, ordered after all completed events.

Second, a thread t executing an operation may do a transition allowed by the sequential semantics of the operation's implementation. Finally, when a thread t finishes executing an operation, as denoted by a continuation skip, the corresponding event is completed with the return value in $\mathtt{res}[t]$. The identifier $\mathsf{last}(t, (E, R))$ of this event is determined as the last one in E by thread t according to R: as per Definition 3.1, events by each thread are totally ordered in a history, ensuring that $\mathsf{last}(t, H)$ is well-defined.

$$\mathsf{last}(t, (E, R)) \triangleq \begin{cases} i, & \text{such that } E(i).\mathsf{tid} = t \\ & \text{and } (\forall j.\, j \neq i \wedge E(j).\mathsf{tid} = t \implies j \xrightarrow{R} i) \\ \bot, & \text{if } \forall i.\, E(i).\mathsf{tid} \neq t \end{cases}$$

Now given an initial state $s_0 \in \mathsf{State}$, we define the set of histories of a data structure D as $\mathcal{H}[\![D, s_0]\!] = \{H \mid \langle (\lambda t.\, \mathsf{idle}), s_0, (\emptyset, \emptyset)\rangle \twoheadrightarrow_D^* \langle \_, H\rangle\}$. We say that a data structure (D, s_0) is *linearizable* with respect to a set of sequential histories $\mathcal{H}$ if $\mathcal{H}[\![D, s_0]\!] \sqsubseteq \mathcal{H}$ (Definition 3.2).

Specification histories. We assume a set of specification states SState, ranged over by σ, and a specification $\mathbb{D}$ of a data structure that interprets every operation $\mathsf{op} \in \mathsf{Op}$ as a sequential state transformer $(\!|\mathsf{op}|\!)_{\mathbb{D}} : (\mathsf{SState} \times \mathsf{Val}) \to \mathcal{P}(\mathsf{SState} \times \mathsf{Val})$. When $(\sigma', r) \in (\!|\mathsf{op}|\!)_{\mathbb{D}}(\sigma, a)$, we say that sequential execution of op with an argument a leads to a state σ' with a return value r.

We generate all sets of histories of a specification $\mathbb{D}$ starting from an initial state σ_0 as follows:

$$\mathcal{H}[\![\mathbb{D}, \sigma_0]\!] \triangleq \{H \mid \mathsf{seq}(H) \wedge \langle (\sigma_0, (\emptyset, \emptyset)\rangle \twoheadrightarrow_{\mathbb{D}}^* \langle \_, H\rangle\},$$

where $\twoheadrightarrow_{\mathbb{D}}$ is a relation constraining a single step in the generation of sequential histories (similarly to $\twoheadrightarrow_D$):

$$\frac{i \notin \mathsf{id}(E) \quad a \in \mathsf{Val} \quad t \in \mathsf{ThreadID} \quad (\sigma', r) \in (\!|\mathsf{op}|\!)_{\mathbb{D}}(\sigma, a) \quad E' = E[i : (t, \mathsf{op}, a, r)] \quad R' = R \cup \{(j, i) \mid j \in \lfloor E \rfloor\}}{\langle \sigma, (E, R)\rangle \twoheadrightarrow_{\mathbb{D}} \langle \sigma', (E', R')\rangle}$$

Having generated all histories $\mathcal{H} = \mathcal{H}[\![\mathbb{D}, \sigma_0]\!]$ of a data structure specification $\mathbb{D}$, we can use Theorem 3.6 to conclude that $\mathcal{H}[\![D, s_0]\!] \sqsubseteq \mathcal{H}[\![\mathbb{D}, \sigma_0]\!]$.

$$
\begin{array}{ll}
(s,H,G,\ell) \models E \mapsto F, & \text{iff } s(\llbracket E \rrbracket_\ell) = \llbracket F \rrbracket_\ell \\
(s,H,G,\ell) \models E = F, & \text{iff } \llbracket E \rrbracket_\ell = \llbracket F \rrbracket_\ell \\
(s,H,G,\ell) \models P \vee Q, & \text{iff either } (s,H,G,\ell) \models P \text{ or } (s,H,G,\ell) \models Q \text{ holds} \\
(s,H,G,\ell) \models P \wedge Q, & \text{iff both } (s,H,G,\ell) \models P \text{ and } (s,H,G,\ell) \models Q \text{ hold} \\
(s,H,G,\ell) \models P \Rightarrow Q, & \text{iff } (s,H,G,\ell) \models Q \text{ holds, when so does } (s,H,G,\ell) \models P \\
(s,H,G,\ell) \models \exists X.\, P, & \text{iff there is } a \in \mathsf{Val} \text{ such that } (s,H,G,\ell[X:a]) \models P \text{ holds} \\
(s,H,G,\ell) \models \forall X.\, P, & \text{iff for any } a \in \mathsf{Val}, (s,H,G,\ell[X:a]) \models P \text{ holds}
\end{array}
$$

Figure 3.10: Semantics of the assertion language Assn. Here we assume a function $\llbracket - \rrbracket_\ell : \mathsf{Expr} \times (\mathsf{LVars} \to \mathsf{Val}) \to \mathsf{Val}$ substitutes logical variables in expressions with their interpretation and evaluates the result.

3.5 Logic

We now formalize our proof method as a Hoare logic based on rely-guarantee [16]. We make this choice to keep presentation simple; our method is general and can be combined with more advanced methods for reasoning about concurrency [24, 29, 45].

Assertions $P, Q \in \mathsf{Assn}$ in our logic denote sets of *configurations* $\kappa \in \mathsf{Config} = \mathsf{State} \times \mathsf{History} \times \mathsf{Ghost}$, relating the data structure state, the abstract history and the ghost state from a set Ghost. The latter can be chosen separately for each proof; e.g., in the proof of the TS queue in §3.3 we used $\mathsf{Ghost} = \mathsf{EventID} \to \mathsf{TS}$. We do not prescribe a particular syntax for assertions, but assume that it includes at least the first-order logic, with a set LVars of special *logical variables* used in specifications and not in programs.

Formally, assertions $P, Q \in \mathsf{Assn}$ are described with the following grammar:

$$
\begin{array}{lll}
E, F & ::= & a \mid X \mid E + F \mid \ldots, \quad \text{where } X \in \mathsf{LVars}, a \in \mathsf{Val} \\
p \in \mathsf{Pred} & ::= & E = F \mid E \mapsto F \mid ... \\
P, Q \in \mathsf{Assn} & ::= & p \mid P \vee Q \mid P \wedge Q \mid P \Rightarrow Q \mid \exists X.\, P \mid \forall X.\, P
\end{array}
$$

Thus, assertions from Assn contain the standard logical connectives, and among them the existential and universal quantification over logical variables X, ranging over a set LVars. We assume a set Pred of predicates, which includes a predicate $E \mapsto F$ denoting a concrete state that describes a singleton heap. We also assume a function $\ell : \mathsf{LVars} \rightharpoonup \mathsf{Val}$ denoting an interpretation of logical variables.

In Figure 3.10, we define a satisfaction relation $\models$ that defines sets of configurations denoted by assertions from Assn. Additionally, we define a function $\llbracket - \rrbracket_- : \mathsf{Assn} \times (\mathsf{LVars} \to \mathsf{Val}) \to \mathcal{P}(\mathsf{Config})$ such that $\llbracket P \rrbracket_\ell$ gives the denotation of an assertion P with respect to an interpretation $\ell : \mathsf{LVars} \to \mathsf{Val}$ of logical variables:

$$\llbracket P \rrbracket_\ell = \{(s,H,G) \mid s,H,G,\ell \models P\}$$

Rely-guarantee is a *compositional* verification method: it allows reasoning about the code executing in each thread separately under some assumption on its environment, specified by a *rely*. In exchange, the thread has to ensure that its behavior conforms to a *guarantee*. Accordingly, judgments of our logic take the form $R, G \vdash_t \{P\}\ C\ \{Q\}$, where C is a command executing in thread t, P and Q are Hoare pre- and post-conditions from Assn, and $R, G \subseteq \mathsf{Config}^2$ are relations defining the rely and the guarantee. Informally, the judgment states

$$\text{(RG-Weaken)}\quad \frac{R, G \vdash_t \{P\}\ C\ \{Q\} \quad P' \Rightarrow P \quad R' \subseteq R \quad G \subseteq G' \quad Q \Rightarrow Q'}{R', G' \vdash_t \{P'\}\ C\ \{Q'\}}$$

$$\text{(Skip)}\quad \frac{}{R, G \vdash_t \{P\}\ \mathsf{skip}\ \{P\}}$$

$$\text{(Seq)}\quad \frac{R, G \vdash_t \{P\}\ C\ \{P'\} \quad R, G \vdash_t \{P'\}\ C'\ \{Q\}}{R, G \vdash_t \{P\}\ C\,;C'\ \{Q\}}$$

$$\text{(Choice)}\quad \frac{R, G \vdash_t \{P\}\ C\ \{Q\} \quad R, G \vdash_t \{P\}\ C'\ \{Q\}}{R, G \vdash_t \{P\}\ C + C'\ \{Q\}}$$

$$\text{(Iter)}\quad \frac{R, G \vdash_t \{P\}\ C\ \{P\}}{R, G \vdash_t \{P\}\ C^*\ \{P\}}$$

Figure 3.11: Proof rules of Rely/Guarantee

$$\frac{\forall \ell.\, G \vDash_t \{[\![P]\!]_\ell\}\ \alpha\ \{[\![Q]\!]_\ell\} \wedge \mathsf{stable}([\![P]\!]_\ell, R) \wedge \mathsf{stable}([\![Q]\!]_\ell, R)}{R, G \vdash_t \{P\}\ \alpha\ \{Q\}}$$

where for $p, q \in \mathcal{P}(\mathsf{Config})$:

$$\mathsf{stable}(p, R) \triangleq \forall \kappa, \kappa'.\, \kappa \in p \wedge (\kappa, \kappa') \in R \implies \kappa' \in p$$

$$\begin{aligned} G \vDash_t \{p\}\ \alpha\ \{q\} \triangleq \forall s, s', H, G.\, (s, H, G) \in p \wedge s' \in [\![\alpha]\!]_t(s) \implies \\ \exists H', G'.\, (s', H', G') \in q \wedge H \rightsquigarrow^* H' \wedge ((s, H, G), (s', H', G')) \in G \end{aligned}$$

and for $(E, R), (E', R') \in \mathsf{History}$:

$$\begin{aligned} (E, R) \rightsquigarrow (E', R') \triangleq (E = E' \wedge R \subseteq R') \vee \\ (\exists i, t, \mathsf{op}, a, r.\, (\forall j.\, j \neq i \implies E(j) = E'(j)) \wedge \\ E(i) = (t, \mathsf{op}, a, \mathsf{todo}) \wedge E'(i) = (t, \mathsf{op}, a, r)) \end{aligned}$$

Figure 3.12: Proof rule for primitive commands.

that C satisfies the Hoare specification $\{P\}\_\{Q\}$ and changes program configurations according to G, assuming that concurrent threads change program configurations according to R.

Our logic includes the standard Hoare proof rules for reasoning about sequential control-flow constructs, which we list in Figure 3.11 and Figure 3.12. We focus on explaining the rule for atomic commands in Figure 3.12, which plays a crucial role in formalizing our proof method. The proof rule derives judgments of the form $R, G \vdash_t \{P\}\ \alpha\ \{Q\}$. The rule takes into account possible interference from concurrent threads by requiring the denotations of P and Q to be *stable* under the rely R, meaning that they are preserved under transitions the latter allows. The rest of the requirements are expressed by the judgment $G \vDash_t \{p\}\ \alpha\ \{q\}$. This requires that for any configuration (s, H, G) from the precondition denotation p and any data structure state s' resulting from thread t executing α in s, we can find a history H' and a ghost state G' such that the new configuration (s', H', G') belongs to the postcondition denotation q. This allows updating the history and the ghost state (almost) arbitrarily, since these are only part of the proof and not of the actual data structure implementation;

the shaded code in Figures 3.4 and 3.5 indicates how we perform these updates in the proof of the TS queue. Updates to the history, performed when α is a commitment point, are constrained by a relation $\rightsquigarrow \subseteq \mathsf{History}^2$, which only allows adding new edges to the real-time order or completing events with a return value. This corresponds to commitment points of kinds 2 and 3 from §3.1. Finally, as is usual in rely-guarantee, the judgment $G \vDash_t \{p\}\ \alpha\ \{q\}$ requires that the change to the program configuration be allowed by the guarantee G.

Note that $\rightsquigarrow$ does not allow adding new events into histories (commitment point of kind 1): this happens automatically when an operation is invoked. In the following, we use a relation $\dashrightarrow_t \subseteq \mathsf{Config}^2$ to constrain the change to the program configuration upon an operation invocation in thread t:

$$\begin{aligned}\langle s, (E, R), G\rangle \dashrightarrow_t \langle s', (E', R'), G'\rangle \iff (\forall l \in \mathsf{Loc}.\, l \neq \mathtt{arg}[t] \implies s(l) = s'(l))\\ \wedge \exists i \notin \mathsf{id}(E).\, E' = E \uplus \{[i : t, \_, \_, \mathsf{todo}]\}\\ \wedge R' = (R \cup \{(j, i) \mid j \in \lfloor E \rfloor\}) \wedge G = G'\end{aligned}$$

Thus, when an operation is invoked in thread t, $\mathtt{arg}[t]$ is overwritten by the operation argument and an uncompleted event associated with thread t and a new identifier i is added to the history; this event is ordered after all completed events, as required by our proof method (§3.1).

Semantics of Hoare specifications. For every set of configurations $p \in \mathcal{P}(\mathsf{Config})$ and each rely R, let $\mathsf{ssw}(p, R)$ be the *strongest stable weaker* set of configurations:

$$\mathsf{ssw}(p, R) = p \cup \{\kappa' \mid \kappa \in p \wedge (\kappa, \kappa') \in R\}$$

Definition 3.4 (Safety Judgment)**.** *We define* safe_t *as the greatest relation such that the following holds whenever* $\mathsf{safe}_t(R, G, p, C, q)$ *does:*

- *if* $C \neq \mathsf{skip}$*, then* $\forall C', \alpha.\, C \rightarrowtail_\alpha C' \implies \exists p'.\, \mathsf{stable}(p', R)$
  $\wedge\, G \vDash_t \{\mathsf{ssw}(p, R)\}\ \alpha\ \{p'\} \wedge \mathsf{safe}_t(R, G, p', C', q)$,
- *if* $C = \mathsf{skip}$*, then* $\mathsf{ssw}(p, R) \subseteq q$.

Lemma 3.5. *For any* t, P, C, op, a *and* Q*, if* $R, G \vdash_t \{P\}\ C\ \{Q\}$ *holds then* $\forall \ell.\, \mathsf{safe}_t(R, G, [\![P]\!]_\ell, C, [\![lQ]\!]_\ell)$

Proof method for linearizability. The rule for primitive commands and the standard Hoare logic proof rules allow deriving judgments about the implementations $D(\mathsf{op})$ of every operation op in a data structure D. The following theorem formalizes the requirements on these judgments sufficient to conclude the linearizability of D with respect to a given set of sequential histories $\mathcal{H}$. The theorem uses the following auxiliary assertions, describing the event corresponding to the current operation op in a thread t at the start and end of its execution (last is defined in §3.4):

$$\begin{aligned}[\![\mathrm{started}_{\mathcal{I}}(t, \mathsf{op})]\!]_\ell = \{(s, (E, R), G) \mid E(\mathsf{last}(t, (E, R))) = (t, \mathsf{op}, s(\mathtt{arg}[t]), \mathsf{todo})\\ \wedge \exists \kappa \in [\![\mathcal{I}]\!]_\ell.\, \langle\kappa\rangle \dashrightarrow_t \langle s, (E, R), G\rangle\};\\ [\![\mathrm{ended}(t, \mathsf{op})]\!]_\ell = \{(s, (E, R), G) \mid E(\mathsf{last}(t, (E, R))) = (t, \mathsf{op}, \_, s(\mathtt{res}[t]))\}.\end{aligned}$$

The assertion $\mathrm{started}_{\mathcal{I}}(t, \mathsf{op})$ is parametrized by a global invariant $\mathcal{I}$ used in the proof. With the help of it, $\mathrm{started}_{\mathcal{I}}(t, \mathsf{op})$ requires that configurations in its denotation be results of adding a new event into histories satisfying $\mathcal{I}$.

Theorem 3.6. *Given a data structure D, its initial state $s_0 \in \mathsf{State}$ and a set of sequential histories $\mathcal{H}$, we have (D, s_0)* linearizable *with respect to $\mathcal{H}$ if there exists an assertion $\mathcal{I}$ and relations $R_t, G_t \subseteq \mathsf{Config}^2$ for each $t \in \mathsf{ThreadID}$ such that:*

1. $\exists G_0.\, \forall \ell.\, (s_0, (\emptyset, \emptyset), G_0) \in [\![\mathcal{I}]\!]_\ell$;
2. $\forall t, \ell.\, \mathsf{stable}([\![\mathcal{I}]\!]_\ell, R_t)$;
3. $\forall H, \ell.\, (\_, H, \_) \in [\![\mathcal{I}]\!]_\ell \implies \mathsf{abs}(H, \mathcal{H})$;
4. $\forall t, \mathsf{op}.\, (R_t, G_t \vdash_t \{\mathcal{I} \wedge \mathrm{started}_{\mathcal{I}}(t, \mathsf{op})\}\ D(\mathsf{op})\ \{\mathcal{I} \wedge \mathrm{ended}(t, \mathsf{op})\})$;
5. $\forall t, t'.\, t \neq t' \implies G_t \cup \dashrightarrow_t \subseteq R_{t'}$.

Here $\mathcal{I}$ is the invariant used in the proof, which item 1 requires to hold of the initial data structure state s_0, the empty history and some some initial ghost state G_0. Item 2 then ensures that the invariant holds at all times. Item 3 requires any history satisfying the invariant to be an abstract history of the given specification $\mathcal{H}$ (Definition 3.3, §3.1). Item 4 constraints the judgment about an operation op executed in a thread t: the operation is executed from a configuration satisfying the invariant and with a corresponding event added to the history; by the end of the operation's execution, we need to complete the event with the return value matching the one produced by the code. Finally, item 5 formalizes a usual requirement in rely-guarantee reasoning: actions allowed by the guarantee of a thread t have to be included into the rely of any other thread t'. We also include the relation $\dashrightarrow_t$, describing the automatic creation of a new event upon an operation invocation in thread t.

Proof of Theorem 3.6

For convenience, we further refer to the assumptions of Theorem 3.6 as a relation $\mathsf{safelib}$ defined as follows.

Definition 3.7. *Given a data structure D, its initial state $s_0 \in \mathsf{State}$ and a set of sequential histories $\mathcal{H}$, we say that $\mathsf{safelib}(D, \mathbb{D}, \mathcal{H})$ holds, if there exists an assertion $\mathcal{I}$ and relations $R_t, G_t \subseteq \mathsf{Config}^2$ for each $t \in \mathsf{ThreadID}$ such that:*

1. $\exists G_0.\, \forall \ell.\, (s_0, (\emptyset, \emptyset), G_0) \in [\![I]\!]_\ell$;
2. $\forall t, \ell.\, \mathsf{stable}([\![I]\!]_\ell, R_t)$;
3. $\forall H, \ell.\, (\_, H, \_) \in [\![I]\!]_\ell \implies \mathsf{abs}(H, \mathcal{H})$;
4. $\forall t, \mathsf{op}.\, (R_t, G_t \vdash_t \{I \wedge \mathrm{started}_{\mathcal{I}}(t, \mathsf{op})\}\ D(\mathsf{op})\ \{I \wedge \mathrm{ended}(t, \mathsf{op})\})$;
5. $\forall t, t'.\, t \neq t' \implies G_t \cup \dashrightarrow_t \subseteq R_{t'}$.

Proposition 3.8. *If $H \rightsquigarrow^* H'$, then there exists H'' such that $H \trianglelefteq H'' \wedge H'' \sqsubseteq H'$*

The proof is straightforward: when $(E, R) \rightsquigarrow^* (E', R')$ holds, $(E, R) \trianglelefteq (E', R)$ and $(E', R) \sqsubseteq (E', R')$.

In the following theorem, we prove that the $\rightsquigarrow$ relation is the correspondence established between a concrete history of a data structure and a matching abstract history under conditions of Theorem 3.6.

Theorem 3.9. *Given a data structure D, its initial state $s_0 \in \mathsf{State}$ and a set of sequential histories $\mathcal{H}$, if* $\mathsf{safelib}(D, \mathbb{D}, \mathcal{H})$ *holds, then the following is true:*

$$\forall h.\, h \in \mathcal{H}[\![D, s_0]\!] \implies \exists H.\, h \rightsquigarrow^* H \wedge \mathsf{abs}(H, \mathcal{H})$$

Intuitively, Theorem 3.9 describes the main idea of our method: for every concrete history $h \in \mathcal{H}[\![D, s_0]\!]$, we build a matching abstract history H such that all of its linearizations are sequential histories from $\mathcal{H}$.

Proof. Proof of Theorem 3.6 We show that Theorem 3.9 is a corollary of Theorem 3.6: given a data structure D, its initial state $s_0 \in \mathsf{State}$ and a set of sequential histories $\mathcal{H}$, we have (D, s_0) *linearizable* with respect to $\mathcal{H}$ if the following holds:

$$\forall H_1.\, H_1 \in \mathcal{H}[\![D, s_0]\!] \implies \exists H.\, H_1 \rightsquigarrow^* H \wedge \mathsf{abs}(H, \mathcal{H}) \tag{3.10}$$

Let us consider every history $H_1 \in \mathcal{H}[\![D, s_0]\!]$. To conclude linearizability w.r.t. $\mathcal{H}$, we need to show the following:

$$\exists H_2 \in \mathcal{H}.\, \exists H_1'.\, H_1 \trianglelefteq H_1' \wedge H_1' \sqsubseteq H_2 \tag{3.11}$$

According to (3.10), there exists H such that $\mathsf{abs}(H, \mathcal{H})$ and $H_1 \rightsquigarrow^* H$ both hold. By Definition 3.3, the former gives the followings:

$$\{H' \mid \lfloor H \rfloor \sqsubseteq H' \wedge \mathsf{seq}(H')\} \subseteq \mathcal{H} \tag{3.12}$$

Note that the set above is not empty, since H is acyclic and thus has at least one linearization H'. Thus, $H' \in \mathcal{H}$ holds.

By Proposition 3.8, there exists a history H'' such that $\mathit{history}_1 \trianglelefteq H''$ and $H'' \sqsubseteq H$. It is easy to see that the following two observations can be made for H_1, H and H'':

- since $H'' \sqsubseteq H$ holds, so does $\lfloor H'' \rfloor \sqsubseteq \lfloor H \rfloor$, and
- since $H_1 \trianglelefteq H''$ holds, so does $H_1 \trianglelefteq \lfloor H'' \rfloor$.

By combining these two observations with (3.12), we conclude that there exist $H_2 = H'$ and $H_1' = \lfloor H'' \rfloor$ such that $H_1 \trianglelefteq H_1'$ and $H_1' \sqsubseteq H_2$. This concludes the proof of (3.11). □

Thus, we reduced the proof of Theorem 3.6 to proving Theorem 3.9. We now introduce auxiliary definitions necessary for the proof of Theorem 3.9, and then present the proof itself.

Definition 3.13. *We let* $\mathrm{isIdle}(t) \in \mathsf{Assn}$ *be an assertion satisfying the following:*

$$[\![\mathrm{isIdle}(t)]\!]_\ell = \{(s, H, G) \mid \mathsf{last}(t, H) = \bot \vee \mathsf{last}(t, H) \in \lfloor H \rfloor\}.$$

The assertion $\mathrm{isIdle}(t)$ represents the set of configurations, in which abstract histories do not have uncompleted events in a thread t.

Definition 3.14. *We let* cinv *be a relation such that* $\mathsf{cinv}(t, \tau, p_t, R_t, G_t, \mathcal{I})$ *holds whenever the following is true:*

- $\mathsf{stable}(p_t, R_t)$ *and* $p_t \subseteq [\![\mathcal{I}]\!]_\ell$ *holds;*

- *if* $\tau(t) = \mathsf{idle}$ *then* $p_t \subseteq [\![\mathrm{isIdle}(t)]\!]_\ell$ *holds;*
- *if* $\tau(t) \neq \mathsf{idle}$ *then there is* op *such that* $\mathsf{safe}_t(R_t, G_t, p_t, \tau(t), [\![\mathcal{I} \wedge \mathrm{ended}(t, \mathsf{op})]\!]_\ell)$.

Proof of Theorem 3.9. Let us consider a data structure D, its initial state $s_0 \in \mathsf{State}$ and the set of specification histories $\mathcal{H}$. Let us assume that $\mathsf{safelib}(D, \mathbb{D}, \mathcal{H})$ holds. In particular, there exist R_t, G_t and $\mathcal{I}$ satisfying the constraints in $\mathsf{safelib}(D, \mathbb{D}, \mathcal{H})$. We prove that (D, s_0) is linearizable with respect to $\mathcal{H}$, i.e., $\mathcal{H}[\![D, s_0]\!] \sqsubseteq \mathcal{H}$. To this end, we strengthen the statement of the theorem as follows:

$$\begin{aligned}\forall \tau, s, h.\, \langle (\lambda t.\, \mathsf{idle}), s_0, (\emptyset, \emptyset)\rangle \twoheadrightarrow_D^* \langle \tau, s, h\rangle \implies \\ \exists H, G.\, h \rightsquigarrow^* H \wedge \mathsf{abs}(H, \mathcal{H}) \wedge \\ (\forall \ell, t.\, \exists p_t.\, (s, H, G) \in [\![p_t]\!]_\ell \wedge \mathsf{cinv}(\ell, t, \tau, p_t, R_t, G_t, \mathcal{I}))\end{aligned}$$

The proof is done by induction on the length n of executions in $\mathcal{H}[\![D, s_0]\!]$. We define the following formula:

$$\begin{aligned}\phi(n) \triangleq \forall \tau, s, h.\, \langle (\lambda t.\, \mathsf{idle}), s_0, (\emptyset, \emptyset)\rangle \twoheadrightarrow_D^n \langle \tau, s, h\rangle \implies \\ \exists H, G.\, h \rightsquigarrow^* H \wedge \\ (\forall \ell, t.\, \exists p_t.\, (s, H, G) \in p_t \wedge \mathsf{cinv}(\ell, t, \tau, p_t, R_t, G_t, \mathcal{I}))\end{aligned}$$

and prove that $\forall n \geq 0.\, \phi(n)$ holds. Note that in $\phi(n)$ we omit the requirement $\mathsf{abs}(H, \mathcal{H})$, since it is implied by the other requirements. Specifically, when cinv holds of each thread t, the configuration (s, H, G) satisfies the invariant $\mathcal{I}$. Consequently, by $\mathsf{safelib}(D, s_0, \mathcal{H})$, $\mathsf{abs}(H, \mathcal{H})$ holds.

Base of the induction. We need to show that $\phi(n)$ holds when $n = 0$. In this case, $\tau = (\lambda t.\, \mathsf{idle})$, $s = s_0$ and $h = (\emptyset, \emptyset)$. According to Definition 3.7.1 of $\mathsf{safelib}(D, \mathbb{D}, \mathcal{H})$, there exists a ghost state G_0 such that:

$$\forall \ell.\, (s_0, (\emptyset, \emptyset), G_0) \in [\![I]\!]_\ell$$

It is easy to see that $\phi(0)$ holds for $H = (\emptyset, \emptyset)$ and $G = G_0$ when $p_t = \mathcal{I}$ for each thread t.

Induction step. We need to show that $\forall n.\, \phi(n) \implies \phi(n+1)$. Let us choose any n and assume that $\phi(n)$ holds. We need to prove that so does $\phi(n+1)$, i.e., for every τ', s' and h' such that $\langle (\lambda t.\, \mathsf{idle}), s_0, (\emptyset, \emptyset)\rangle \twoheadrightarrow_D^{n+1} \langle \tau', s', h'\rangle$ holds, the following is true:

$$\begin{aligned}\exists H', G'.\, h' \rightsquigarrow^* H' \wedge \mathsf{abs}(H', \mathcal{H}) \wedge \\ (\forall \ell, t.\, \exists p'_t.\, (s', H', G') \in p'_t \wedge \mathsf{cinv}(\ell, t, \tau', p'_t, R_t, G_t, \mathcal{I}))\end{aligned} \quad (3.15)$$

When $\langle (\lambda t.\, \mathsf{idle}), s_0, (\emptyset, \emptyset)\rangle \twoheadrightarrow_D^{n+1} \langle \tau', s', h'\rangle$, there exist τ, s and H such that:

$$\langle (\lambda t.\, \mathsf{idle}), s_0, (\emptyset, \emptyset)\rangle \twoheadrightarrow_D^n \langle \tau, s, h\rangle \wedge \langle \tau, s, h\rangle \twoheadrightarrow_D \langle \tau', s', h'\rangle$$

According to the induction hypothesis $\phi(n)$, for τ, s and h there exist H and G such that:

$$\begin{aligned}h \rightsquigarrow^* H \wedge \mathsf{abs}(H, \mathcal{H}) \wedge \\ (\forall \ell, t.\, \exists p_t.\, (s, H, G) \in [\![p_t]\!]_\ell \wedge \mathsf{cinv}(\ell, t, \tau, p_t, R_t, G_t, \mathcal{I}))\end{aligned} \quad (3.16)$$

By definition of the transition relation $\twoheadrightarrow_D$, a transition $\langle \tau, s, h\rangle \twoheadrightarrow_D \langle \tau', s', h'\rangle$ corresponds to one of the three cases for a continuation $\tau(T)$ of some thread T: an invocation of an arbitrary new operation in T, a return from the current operation of T, or a transition in T. We consider each case separately.

Let E' and R' be such that $h' = (E', R')$, and let E and R be such that $h = (E, R)$.

Case #1. There exists a thread T such that $\tau(T) = \mathsf{idle}$, an operation $\mathsf{op} \in \mathsf{Op}$, its arguments a, its event identifier $i \notin \mathsf{id}(E)$ such that $\langle \tau, s, (E, R)\rangle \twoheadrightarrow_D \langle \tau', s', (E', R')\rangle$ holds and the following is true:

- $\tau' = \tau[T : D(\mathsf{op})]$,
- $s' = s[\mathtt{arg}[T] : a]$,
- $E' = E[i : (T, \mathsf{op}, a, \mathsf{todo})]$,
- $R' = R \cup \{(j, i) \mid j \in \lfloor E \rfloor\}$.

Let us consider any ℓ. According to (3.16), for a thread T there exists p_T such that $\mathsf{cinv}(\ell, T, \tau, p_T, R_T, G_T, \mathcal{I})$ and $(s, H, G) \in p_T$ both hold.

From the former we learn that $\kappa \in p_T \subseteq [\![\mathcal{I} \wedge \mathrm{isIdle}(T)]\!]_\ell$, since $\tau(T) = \mathsf{idle}$. It is easy to see that under this condition, for every abstract history H' such that $\langle s, H, G\rangle \dashrightarrow_T \langle s', H', G\rangle$ holds, the following is true:

- $h' \rightsquigarrow^* H'$ holds, since the new event $[i : (T, \mathsf{op}, a, \mathsf{todo})]$ and the new edges added into $h = (E, R)$ can also be added into H,
- $(s', H', G) \in [\![\mathcal{I} \wedge \mathrm{started}(T, \mathsf{op})]\!]_\ell$ holds.

According to $\mathsf{safelib}(D, s_0, \mathcal{H})$, the command $D(\mathsf{op})$ fulfills the following specification:

$$R_T, G_T \vdash_T \left\{\mathcal{I} \wedge \mathrm{started}_{\mathcal{I}}(T, \mathsf{op})\right\}\ D(\mathsf{op})\ \left\{\mathcal{I} \wedge \mathrm{ended}(T, \mathsf{op})\right\})$$

Hence, $\mathsf{safe}_T(R_T, G_T, [\![\mathcal{I} \wedge \mathrm{started}_{\mathcal{I}}(T, \mathsf{op})]\!]_\ell, D(\mathsf{op}), [\![\mathcal{I} \wedge \mathrm{ended}(T, \mathsf{op})]\!]_\ell)$ holds by Lemma 3.5. Let $p'_T = \mathsf{ssw}([\![\mathcal{I} \wedge \mathrm{started}_{\mathcal{I}}(T, \mathsf{op})]\!]_\ell, R_T)$. It is easy to see that $\mathsf{safe}_T(R_T, G_T, p'_T, D(\mathsf{op}), [\![\mathcal{I} \wedge \mathrm{ended}(T, \mathsf{op})]\!]_\ell)$ holds as well. Thus, $\mathsf{cinv}(\ell, T, \tau', p'_T, R_T, G_T, \mathcal{I})$ holds. For a thread T, we have found p'_T such that:

$$(s', H', G) \in p'_T \wedge \mathsf{cinv}(\ell, t, \tau, p'_T, R_T, G_T, \mathcal{I})$$

Let us consider every thread $t \neq T$. By the hypothesis (3.16), there exists p_t such that $(s, H, G) \in p_t$ and $\mathsf{cinv}(\ell, t, \tau, p_t, R_t, G_t, \mathcal{I})$ hold. According to the latter, p_t is stable under R_t. By $\mathsf{safelib}(D, s_0, \mathcal{H})$, R_t includes $\dashrightarrow_T$, so p_t is stable under $\dashrightarrow_T$ as well. Consequently, $(s', H', G) \in p_t$ holds.

Thus, we have found H', $G' = G$ and a new set of configurations p_T for a thread T such that (3.15) holds.

Case #2. There exists a thread T such that $\tau(T) = \mathsf{skip}$, an operation $\mathsf{op} \in \mathsf{Op}$, its arguments a, its event identifier $i = \mathsf{last}(t, (E, R))$ such that $\langle \tau, s, (E, R)\rangle \twoheadrightarrow_D \langle \tau', s', (E', R')\rangle$ holds and the following is true:

- $\tau' = \tau[T : \mathsf{idle}]$,
- $E(i) = (t, \mathsf{op}, a, \mathsf{todo})$

- $E' = E[i : (t, \mathsf{op}, a, s(\mathtt{res}[t]))]$,
- $s' = s$ and $R = R'$

Let us consider any ℓ. According to (3.16), for a thread T there exists p_T such that $\mathsf{cinv}(\ell, T, \tau, p_T, R_T, G_T, \mathcal{I})$ and $(s, H, G) \in p_T$ both hold. From the former we learn that $\mathsf{safe}_T(R_T, G_T, p_T, \mathsf{skip}, [\![\mathcal{I} \wedge \mathrm{ended}(T, \mathsf{op})]\!]_\ell)$ holds. Then the following is true:

$$(s, H, G) \in p_T \subseteq [\![\mathcal{I} \wedge \mathrm{ended}(T, \mathsf{op})]\!]_\ell \subseteq [\![\mathcal{I} \wedge \mathrm{isIdle}(T)]\!]_\ell$$

Hence, the abstract history H does not have uncompleted events in a thread T. By the induction hypothesis, $h \rightsquigarrow^* H$ holds. Since h' completes the event that is already completed in H, we can conclude that $h' \rightsquigarrow^* H$ holds too.

Also, when $p_T \subseteq [\![\mathcal{I} \wedge \mathrm{isIdle}(T)]\!]_\ell$ holds, so does $\mathsf{cinv}(\ell, T, \tau', p_T, R_T, G_T, \mathcal{I})$. Thus, for a thread T, we have found $p'_T = p_T$ such that:

$$(s', H, G) \in p'_T \wedge \mathsf{cinv}(\ell, t, \tau', p'_T, R_T, G_T, \mathcal{I})$$

Overall, we have found $H' = H$ and $G' = G$ such that (3.11) holds.
Case #3. There exists a thread T, a primitive command α, commands C and C' such that $C \rightarrowtail_\alpha C'$, $s' \in [\![\alpha]\!]_T(s)$ and the following is true:

- $\tau(T) = C$ and $\tau' = \tau[T : C']$,
- $E' = E$ and $R = R'$

Let us consider any ℓ. According to (3.16), for a thread T there exists p_T such that $\mathsf{cinv}(\ell, T, \tau, p_T, R_T, G_T, \mathcal{I})$ and $(s, H, G) \in p_T$ both hold. According to the former, $\mathsf{safe}_T(R_T, G_T, p_T, C, [\![\mathcal{I} \wedge \mathrm{ended}(T, \_)]\!]_\ell)$. By Definition 3.4, there exists a stable p'_T such that $G_T \vDash_T \{p_T\}\ \alpha\ \{p'_T\}$ and $\mathsf{safe}_T(R_T, G_T, p'_T, C', [\![\mathcal{I} \wedge \mathrm{ended}(T, \_)]\!]_\ell)$. Also, the following holds:

- By definition of $G_T \vDash_T \{p_T\}\ \alpha\ \{p'_T\}$, there exist H' and G' such that $(s', H', G') \in p'_T$. Also, $G_T \vDash_T \{p_T\}\ \alpha\ \{p'_T\}$ implies that $h' = h \rightsquigarrow^* H \rightsquigarrow^* H'$.
- By $\mathsf{safelib}(D, s_0, \mathcal{H})$, $[\![\mathcal{I}]\!]_\ell$ is stable under G_T, meaning that $p'_T \subseteq [\![\mathcal{I}]\!]_\ell$ holds.

It is easy to see that $\mathsf{cinv}(\ell, T, \tau', p'_T, R_T, G_T, \mathcal{I})$ holds. Thus, for a thread T, we have found p'_T such that:

$$(s', H', G') \in p'_T \wedge \mathsf{cinv}(\ell, t, \tau', p'_T, R_T, G_T, \mathcal{I})$$

Let us consider every thread $t \neq T$. By the hypothesis (3.16), there exists p_t such that $(s, H, G) \in p_t$ and $\mathsf{cinv}(\ell, t, \tau, p_t, R_t, G_t, \mathcal{I})$ hold. According to the latter, p_t is stable under $G_T \subseteq R_t$. Consequently, $(s', H', G') \in p_t$ holds.

Thus, we have found H', G' and a new set of configurations p'_T for a thread T such that (3.15) holds. □

($\mathsf{INV}_{\mathsf{LIN}}$) all linearizations of completed events of the abstract history satisfy the queue specification:

$$\forall H'.\, \lfloor H \rfloor \sqsubseteq H' \wedge \mathsf{seq}(H') \implies H' \in \mathcal{H}_{\mathsf{queue}} \wedge \mathsf{same\_data}(s, H, G_{\mathsf{ts}}, H')$$

($\mathsf{INV}_{\mathsf{ORD}}$) properties of the partial order of the abstract history:

(i) completed dequeues precede uncompleted ones:

$$\forall i \in \mathsf{id}(\lfloor E \rfloor).\, \forall j \in \mathsf{id}(E \setminus \lfloor E \rfloor).\, E(i).\mathsf{op} = E(j).\mathsf{op} = \mathrm{Deq} \implies i \xrightarrow{R} j$$

(ii) enqueues of already dequeued values precede enqueues of values in the pools:

$$\forall i \in \mathsf{id}(\lfloor E \rfloor) \setminus \mathsf{inQ}(s(\mathtt{pools}), E, G_{\mathsf{ts}}).\, \forall j \in \mathsf{inQ}(s(\mathtt{pools}), E, G_{\mathsf{ts}}).\, i \xrightarrow{R} j$$

($\mathsf{INV}_{\mathsf{ALG}}$) properties of the algorithm used to build the loop invariant:

(i) enqueues of values in the pools are ordered only if so are their timestamps:

$$\forall i, j \in \mathsf{inQ}(s(\mathtt{pools}), E, G_{\mathsf{ts}}).\, i \xrightarrow{R} j \implies G_{\mathsf{ts}}(i) <_{\mathsf{TS}} G_{\mathsf{ts}}(j)$$

(ii) values in each pool appear in the order of enqueues that inserted them:

$$\forall t, \tau_1, \tau_2.\, \mathtt{pools}(t) = \_ \cdot (\_, \_, \tau_1) \cdot \_ \cdot (\_, \_, \tau_2) \cdot \_ \implies \mathtt{enqOf}(E, G_{\mathsf{ts}}, t, \tau_1) \xrightarrow{R} \mathtt{enqOf}(E, G_{\mathsf{ts}}, t, \tau_2)$$

(iii) the timestamps of values are smaller than the global counter:

$$\forall i, a, b.\, G_{\mathsf{ts}}(i) = (a, b) \implies b < s(\mathtt{counter})$$

(iv) the pools contain the values of uncompleted enqueue events:

$$\forall i \in \mathsf{id}(E \setminus \lfloor E \rfloor).\, E(i).\mathsf{op} = \mathrm{Enq} \implies i \in \mathsf{inQ}(s(\mathtt{pools}), E, G_{\mathsf{ts}})$$

($\mathsf{INV}_{\mathsf{WF}}$) properties of ghost state:

(i) G_{ts} associates timestamps with enqueue events:

$$\forall i.\, i \in \mathsf{dom}(G_{\mathsf{ts}}) \implies E(i).\mathsf{op} = \mathrm{Enq}$$

(ii) each value in a pool has a matching event for the enqueue that inserted it:

$$\forall t, v, \tau.\, \mathtt{pools}(t) = \_ \cdot (\_, v, \tau) \cdot \_ \implies \exists i.\, E(i) = (t, \mathrm{Enq}, v, \_) \wedge G_{\mathsf{ts}}(i) = \tau$$

(iii) all timestamps assigned by G_{ts} to events of a given thread are distinct:

$$\forall i, j.\, i \neq j \wedge E(i).\mathsf{tid} = E(j).\mathsf{tid} \wedge i, j \in \mathsf{dom}(G_{\mathsf{ts}}) \implies G_{\mathsf{ts}}(i) \neq G_{\mathsf{ts}}(j)$$

(iv) G_{ts} associates uncompleted enqueue events with the timestamp $\top$:

$$\forall i.\, E(i).\mathsf{op} = \mathrm{Enq} \implies (i \notin \mathsf{id}(\lfloor E \rfloor) \iff i \notin \mathsf{dom}(G_{\mathsf{ts}}) \vee G_{\mathsf{ts}}(i) = \top)$$

Figure 3.13: The invariant $\mathsf{INV} = \mathsf{INV}_{\mathsf{LIN}} \wedge \mathsf{INV}_{\mathsf{ORD}} \wedge \mathsf{INV}_{\mathsf{ALG}} \wedge \mathsf{INV}_{\mathsf{WF}}$

3.6 The TS Queue: Proof Details

In this section, we present some of the details of the proof of the TS Queue. We provide additional details in Appendix A.1.

Invariant. We satisfy the obligation 4 from Theorem 3.6 by proving the invariant INV defined in Figure 3.13. The invariant is an assertion consisting of four parts: $\mathsf{INV}_{\mathsf{LIN}}$, $\mathsf{INV}_{\mathsf{ORD}}$, $\mathsf{INV}_{\mathsf{ALG}}$ and $\mathsf{INV}_{\mathsf{WF}}$. Each of them denotes a set of configurations satisfying the listed constraints for a given interpretation of logical variables ℓ. The first part of the invariant, $\mathsf{INV}_{\mathsf{LIN}}$, ensures that every history satisfying the invariant is an abstract history of the queue, which discharges the obligation 3 from Theorem 3.6. In addition to that, $\mathsf{INV}_{\mathsf{LIN}}$ requires that a relation $\mathsf{same\_data}$ hold of a configuration (s, H, G_{ts}) and every linearization H'. In this way, we ensure that the pools and the final state of the sequential queue after H' contain values inserted by the same enqueue events (we formalize $\mathsf{same\_data}$ in Appendix A.1). The second part, $\mathsf{INV}_{\mathsf{ORD}}$, asserts ordering properties of events in the partial order that hold by construction. The third part, $\mathsf{INV}_{\mathsf{ALG}}$, is a collection of properties relating the order on timestamps to the partial order in abstract history. Finally, $\mathsf{INV}_{\mathsf{WF}}$ is a collection of well-formedness properties of the ghost state.

Loop invariant. We now present the key verification condition that arises in the `dequeue` operation: demonstrating that the ordering enforced at the commitment points at lines 49–50 and 58–65 does not invalidate acyclicity of the abstract history. To this end, for the `foreach` loop (lines 46–56) we build a loop invariant based on distinguishing certain values in the pools as *seen* by the `dequeue` operation. With the help of the loop invariant we establish that acyclicity is preserved at the commitment points.

Recall from §3.2, that the `foreach` loop starts iterating from a random pool. In the proof, we assume that the loop uses a thread-local variable `A` for storing a set of identifiers of threads that have been iterated over in the loop. We also assume that at the end of each iteration the set `A` is extended with the current loop index `k`.

Note also that for each thread `k`, the commitment point of a dequeue d at lines 49–50 ensures that enqueue events of values the operation sees in `k`'s pool precede d in the abstract history. Based on that, during the `foreach` loop we can we distinguish enqueue events with values in the pools that a dequeue d has seen after looking into pools of threads from `A`. We define the set of all such enqueue events as follows:

$$\begin{aligned}\mathsf{seen}((s,(E,R),G_{\mathsf{ts}}),d) \triangleq \{e \mid e \in \mathsf{id}(\lfloor E \rfloor) \cap \mathsf{inQ}(s(\mathtt{pools}),E,G_{\mathsf{ts}}) \\ \wedge\, e \xrightarrow{R} d \wedge \neg(s(\mathtt{start\_ts}) <_{\mathsf{TS}} G_{\mathsf{ts}}(e)) \wedge E(e).\mathsf{tid} \in \mathtt{A}\}\end{aligned} \quad (3.17)$$

A loop invariant LI is simply a disjunction of two auxiliary assertions, isCand and noCand, which are defined in Figure 3.14 (given an interpretation of logical variables ℓ, each of assertions denotes a set of configurations satisfying the listed constraints). The assertion noCand denotes a set of configurations $\kappa = (s, (E, R), G_{\mathsf{ts}})$, in which the `dequeue` operation has not chosen a candidate for removal after having iterated over the pools of threads from `A`. In this case, $s(\mathtt{cand\_pid}) = \text{NULL}$, and the current dequeue has not seen any enqueue event in the pools of threads from `A`.

$$\text{(noCand):}\ \mathsf{seen}((s, H, G_{\mathsf{ts}}), \mathtt{myEid}()) = \emptyset \wedge s(\mathtt{cand\_pid}) = \text{NULL}$$

$$\text{(minTS}(e)\text{):}\ \forall e' \in \mathsf{seen}((s, H, G_{\mathsf{ts}}), \mathtt{myEid}()).\, \neg(G_{\mathsf{ts}}(e') <_{\mathsf{TS}} G_{\mathsf{ts}}(e))$$

$$\begin{aligned}\text{(isCand):}\ \exists \mathtt{CAND}.\, \mathtt{CAND} = \mathtt{enqOf}(E, G_{\mathsf{ts}}, s(\mathtt{cand\_tid}), s(\mathtt{cand\_ts}))\\ \wedge \text{minTS}(\mathtt{CAND}) \wedge (\mathtt{CAND} \in \mathsf{inQ}(s(\mathtt{pools}), E, G_{\mathsf{ts}}) \implies\\ \mathtt{CAND} \in \mathsf{seen}((s, H, G_{\mathsf{ts}}), \mathtt{myEid}())) \wedge s(\mathtt{cand\_pid}) \neq \text{NULL}\end{aligned}$$

Figure 3.14: Auxiliary assertions for the loop invariant

The assertion isCand denotes a set of configurations $\kappa = (s, (E, R), G_{\mathsf{ts}})$, in which an enqueue event $\mathtt{CAND} = \mathtt{enqOf}(E, G_{\mathsf{ts}}, \mathtt{cand\_tid}, \mathtt{cand\_ts})$ has been chosen as a candidate for removal out of the enqueues seen in the pools of threads from `A`. As `CAND` may be removed by a concurrent dequeue, isCand requires that `CAND` remain in the set $\mathsf{seen}(\kappa, \mathtt{myEid}())$ as long as `CAND`'s value remains in the pools. Additionally, by requiring minTS(`CAND`), isCand asserts that the timestamp of `CAND` is minimal among other enqueues seen by `myEid`().

In the following lemma, we prove that the assertion isCand implies minimality of `CAND` in the abstract history among enqueue events with values in the pools of threads from `A`. The proof is based on the observation that enqueues of values seen in the pools by a dequeue are never preceded by unseen enqueues.

Lemma 3.18. *For every* $\ell : \mathsf{LVars} \to \mathsf{Val}$ *and configuration* $(s, (E, R), G_{\mathsf{ts}}) \in [\![\text{isCand}]\!]_\ell$, *if* $\mathtt{CAND} = \mathtt{enqOf}(E, G_{\mathsf{ts}}, \mathtt{cand\_tid}, \mathtt{cand\_ts})$ *and* $\mathtt{CAND} \in \mathsf{inQ}(s(\mathtt{pools}), E, G_{\mathsf{ts}})$ *both hold, then the following is true:*

$$\forall e \in \mathsf{inQ}(s(\mathtt{pools}), \mathtt{E}, \mathtt{G_{ts}}).\, E(e).\mathsf{tid} \in \mathtt{A} \implies \neg(e \xrightarrow{R} \mathtt{CAND})$$

Acyclicity. At the commitment points extending the order of the abstract history, we need to show that the extended order is acyclic as required by Definition 3.1 of the abstract history. To this end, we argue that the commitment points at lines 49–50 and lines 58–65 preserve acyclicity of the abstract history.

The commitment point at lines 49–50 orders certain completed enqueue events before the current uncompleted dequeue event `myEid`(). By Definition 3.1 of the abstract history, the partial order on its events is transitive, and uncompleted events do not precede other events. Since `myEid`() does not precede any other event, ordering any completed enqueue event before `myEid`() cannot create a cycle in the abstract history.

We now consider the commitment point at lines 58–65 in the current dequeue `myEid`(). Prior to the commitment point, the loop invariant `LI` has been established in all threads, and the check `cand_pid` $\neq$ NULL at line 57 has ruled out the case when noCand holds. Thus, the candidate for removal `CAND` has the properties described by isCand. If `CAND`'s value has already been dequeued concurrently, the removal fails, and the abstract history remains intact (and acyclic). When the removal succeeds, we consider separately the two kind of edges added into the abstract history (E, R):

1. **The case of** $(\mathtt{CAND}, e)$ **for each** $e \in \mathsf{inQ}(\mathtt{pools}, E, G_{\mathsf{ts}})$. By Lemma 3.18, an edge $(e, \mathtt{CAND})$ is not in the partial order R of the abstract history.

There is also no sequence of edges $e \xrightarrow{R} ... \xrightarrow{R} \texttt{CAND}$, since R is transitive by Definition 3.1. Hence, cycles do not arise from ordering `CAND` before e.

2. **The case of** $(\texttt{myEid}(), d)$ **for each identifier** d **of an uncompleted dequeue event**. By Definition 3.1 of the abstract history, uncompleted events do not precede other events. Since d is uncompleted event, it does not precede `myEid()`. Hence, ordering `myEid()` in front of all such dequeue events does not create cycles.

Rely and guarantee relations. We now explain how we generate rely and guarantee relations for the proof. Instead of constructing the relations with the help of abstracted intermediate assertions of a proof outline for the `enqueue` and `dequeue` operations, we use the non-deterministic state transformers of primitive commands together with the ghost code in Figure 3.4 and Figure 3.5. To this end, the semantics of state transformers is extended to account for changes to abstract histories and ghost state. We found that generating rely and guarantee relations in such non-standard way results in cleaner stability proofs for the TS Queue, and makes them similar in style to checking non-interference in the Owicki-Gries method [46].

Let us refer to atomic blocks with corresponding ghost code at line 27, line 32, line 49 and line 58 as atomic steps `insert`, `setTS`, `scan(k)` ($\texttt{k} \in \mathsf{ThreadID}$) and `remove` respectively, and let us also refer to the CAS operation at line 74 as `genTS`. For each thread t and atomic step $\hat{\alpha}$, we assume a non-deterministic configuration transformer $[\![\hat{\alpha}]\!]_t : \mathsf{Config} \to \mathcal{P}(\mathsf{Config})$ that updates state according to the semantics of a corresponding primitive command, and history with ghost state as specified by ghost code.

Given an assertion P, an atomic step $\hat{\alpha}$ and a thread t, we associate them with the following relation $G_{t,\hat{\alpha},P} \subseteq \mathsf{Config}^2$:

$$G_{t,\hat{\alpha},P} \triangleq \{(\kappa, \kappa') \mid \exists \ell.\, \kappa \in [\![P]\!]_\ell \wedge \kappa' \in [\![\hat{\alpha}]\!]_t(\kappa)\}$$

Additionally, we assume a relation $G_{t,\texttt{local}}$, which describes arbitrary changes to certain program variables and no changes to the abstract history and the ghost state. That is, we say that `pools` and `counter` are *shared* program variables in the algorithm, and all others are *thread-local*, in the sense that every thread has its own copy of them. We let $G_{t,\texttt{local}}$ denote every possible change to thread-local variables of a thread t only.

For each thread t, relations G_t and R_t are defined as follows:

$$\begin{array}{rcl} P_{\mathsf{op}} & \triangleq & \mathsf{INV} \wedge \mathrm{started}(t, \mathsf{op}) \\ G_t & \triangleq & (\bigcup_{t' \in \mathsf{ThreadID}} G_{t,\texttt{scan(t')},P_{\mathrm{Deq}}}) \cup G_{t,\texttt{remove},P_{\mathrm{Deq}}} \\ & & \cup\, G_{t,\texttt{insert},P_{\mathrm{Enq}}} \cup G_{t,\texttt{setTS},P_{\mathrm{Enq}}} \cup G_{t,\texttt{genTS},\mathsf{INV}} \cup G_{t,\texttt{local}}, \\ R_t & \triangleq & \cup_{t' \in \mathsf{ThreadID} \setminus \{t\}} (G_{t'} \cup \dashrightarrow_{t'}) \end{array}$$

As required by Theorem 3.6, the rely relation of a thread t accounts for addition of new events in every other thread t' by including $\dashrightarrow'_t$. Also, R_t takes into consideration every atomic step by the other threads. Thus, the rely and guarantee relations satisfy all the requirement 5 of the proof method from Theorem 3.6. It is easy to see that the requirement 2 is also fulfilled: the global invariant INV is simply preserved by each atomic step, so it is indeed stable under rely relations of each thread.

The key observation implying stability of the loop invariant in every thread t is presented in the following lemma, which states that environment transitions in the rely relation never extend the set of enqueues seen by a given dequeue.

Lemma 3.19. *If a dequeue event* `DEQ` *generated its timestamp* `start_ts`*, then:*

$$\forall \kappa, \kappa'.\, (\kappa, \kappa') \in R_t \implies \mathsf{seen}(\kappa', \texttt{DEQ}) \subseteq \mathsf{seen}(\kappa, \texttt{DEQ})$$

3.7 The Optimistic Set: Informal Development

The algorithm. We now present another example, the Optimistic Set [19], which is a variant of a classic algorithm by Heller et al. [47], rewritten to use atomic sections instead of locks. However, this is a highly-concurrent algorithm: every atomic section accesses a small bounded number of memory locations. In this section we only give an informal explanation of the proof and commitment points; more details are provided in Appendix A.2.

The set is implemented as a sorted singly-linked list. Each node in the list has three fields: an integer `val` storing the key of the node, a pointer `next` to the subsequent node in the list, and a boolean flag `marked` that is set true when the node gets removed. The list also has sentinel nodes `head` and `tail` that store $-\infty$ and $+\infty$ as keys accordingly. The set defines three operations: `insert`, `remove` and `contains`. Each of them uses an internal operation `locate` to traverse the list. Given a value `v`, `locate` traverses the list nodes and returns a pair of nodes `(p, c)`, out of which `c` has a key greater or equal to `v`, and `p` is the node preceding `c`.

The `insert` (`remove`) operation spins in a loop locating a place after which a new node should be inserted (after which a candidate for removal should be) and attempting to atomically modify the data structure. The attempt may fail if either `p.next = c` or `!p.marked` do not hold: the former condition ensures that concurrent operations have not removed or inserted new nodes immediately after `p.next`, and the latter checks that `p` has not been removed from the set. When either check fails, the operation restarts. Both conditions are necessary for preserving integrity of the data structure.

When the elements are removed from the set, their corresponding nodes have the `marked` flag set and get unlinked from the list. However, the `next` field of the removed node is not altered, so marked and unmarked nodes of the list form a tree such that each node points towards the root, and only nodes reachable from the head of the list are unmarked. In Figure 3.16, we have an example state of the data structure. The `insert` and `remove` operations determine the position of a node `p` in the tree by checking the flag `p.marked`. In `remove`, this check prevents removing the same node from the data structure twice. In `insert`, checking `!p.marked` ensures that the new node `n` is not inserted into a branch of removed nodes and is reachable from the head of the list.

In contrast to `insert` and `remove`, `contains` never modifies the shared state and never restarts. This leads to a subtle interaction that may happen due to interference by concurrent events: it may be correct for `contains` to return `true` even though the node may have been removed by the time `contains` finds it in the list.

In Figure 3.16, we illustrate the subtleties with the help of a state of the set, which is a result of executing the trace from Figure 3.17, assuming that

```
struct Node {
  Node *next;
  Int val;
  Bool marked;
}

Bool contains(v) {
  p, c := locate(v);
  return (c.val = v);
}

Bool insert(v) {
  Node×Node p, c;
  do {
    p, c := locate(v);
    atomic {
      if (p.next = c
          && !p.marked) {
        commit_insert();
        if (c.val ≠ v) {
          Node *n := new Node;
          n->next := c;
          n->val := v;
          n->marked := false;
          p.next := n;
          return true;
        } else
          return false;
      }
    }
  } while (true);
}
```

```
Node×Node locate(v) {
  Node prev := head;
  Node curr := prev.next;
  while (curr.val < v) {
    prev := curr;
    atomic {
      curr := curr.next;
      if (E(myEid()).op = contains
          && (curr.val ≥ v))
        commit_contains();
    }
  }
  return prev, curr;
}

Bool remove(v) {
  Node×Node p, c;
  do {
    p, c := locate(v);
    atomic {
      if (p.next = c
          && !p.marked) {
        commit_remove();
        if (c.val = v) {
          c.marked := true;
          p.next := c.next;
          return true;
        } else
          return false;
      }
    }
  } while (true);
}
```

Figure 3.15: The Optimistic Set. Shaded portions are auxiliary code used in the proof

values 1, 2 and 4 have been initially inserted in sequence by performing "Ins(1)", "Ins(2)" and "Ins(4)". We consider the following scenario. First, "Con(2)" and "Con(3)" start traversing through the list and get preempted when they reach the node containing 1, which we denote by n_1. Then the operations are finished in the order depicted in Figure 3.17. Note that "Con(2)" returns true even though the node containing 2 is removed from the data structure by the time the `contains` operation locates it. This surprising behavior occurs due to the values 1 and 2 being on the same branch of marked nodes in the list, which makes it possible for "Con(2)" to resume traversing from n_1 and find 2. On the other hand, "Con(3)" cannot find 3 by traversing the nodes from n_1: the `contains` operation will reach the node n_2 and return false, even though 3 has been concurrently inserted into the set by this time. Such behavior is correct, since it can be justified by a linearization ["Ins(1)", "Ins(2)", "Ins(4)", "Rem(1)", "Con(2): true", "Rem(2)", "Con(3): false", "Ins(3)"]. Intuitively, such linearization order is possible, because pairs of events ("Con(2): true", "Rem(2)") and ("Con(3):

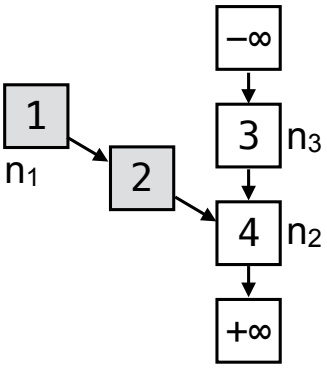


Figure 3.16: Example state of the optimistic set. Shaded nodes have their "marked" field set.

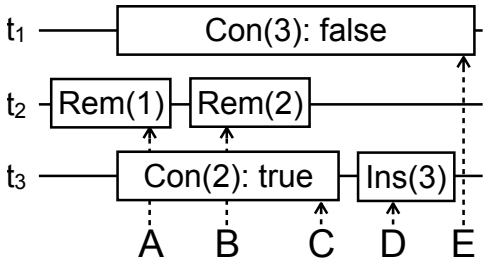


Figure 3.17: Example execution of the set. "Ins" and "Rem" denote successful `insert` and `remove` operations accordingly, and "Con" denotes `contains` operations. A–E correspond to commitment points of operations.

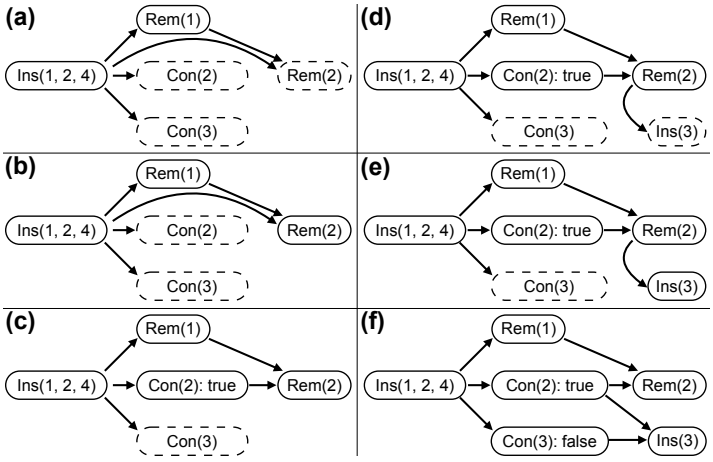


Figure 3.18: Changes to the abstract history of the execution in Figure 3.17. Edges implied by transitivity are omitted.

false", "Ins(3)") overlap in the execution.

Building a correct linearization order by identifying a linearization point of `contains` is complex, since it depends on presence of concurrent `insert` and `remove` operation as well as on current position in the traversal of the data structure. We demonstrate a different approach to the proof of the Optimistic Set based on the following insights. Firstly, we observe that only decisions about a relative order of operations with the same argument need to be committed into the abstract history, since linearizability w.r.t. the sequential specification of a set does not require enforcing any additional order on concurrent operations with different arguments. Secondly, we postpone decisions about ordering `contains` operations w.r.t. concurrent events till their return values are determined. Thus, in the abstract history for Figure 3.17, "Con(2): true" and "Rem(2)" remain unordered until the former encounters the node removed by the latter, and the

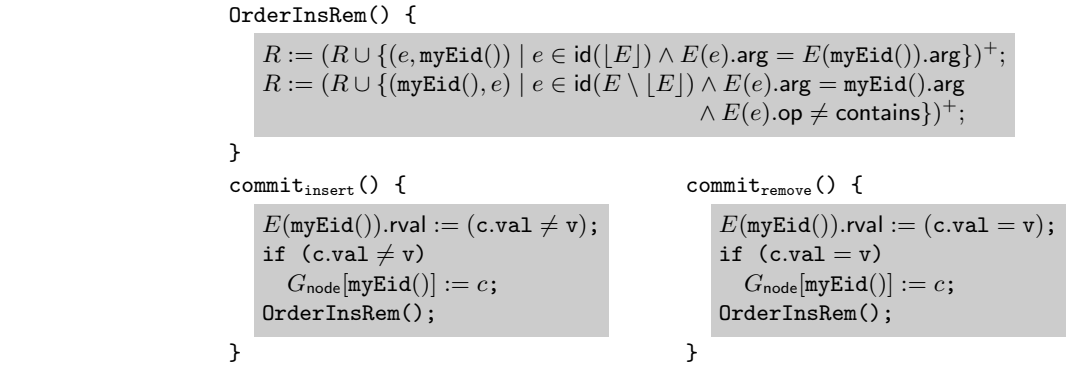


Figure 3.19: The auxiliary code executed at the commitment points of `insert` and `remove`

order between operations becomes clear. Intuitively, we construct a linear order on completed events with the same argument, and let `contains` operations be inserted in a certain place in that order rather than appended to it.

Preliminaries. We assume that a set NodeID is a set of pointers to nodes, and that the state of the linked list is represented by a partial map $\mathsf{NodeID} \rightharpoonup \mathsf{NodeID} \times \mathsf{Int} \times \mathsf{Bool}$. To aid in constructing the abstract history (E, R), the code maintains a piece of ghost state—a partial function $G_{\mathsf{node}} : \mathsf{EventID} \rightharpoonup \mathsf{NodeID}$. Given the identifier i of an event $E(i)$ denoting an `insert` that has inserted its value into the set, $G_{\mathsf{node}}(i)$ returns a node identifier (a pointer) of that value in the data structure. Similarly, for a successful remove event identifier i, $G_{\mathsf{node}}(i)$ returns a node identifier that the corresponding operation removed from the data structure.

Commitment points. The commitment points in the `insert` and `remove` operations are denoted by ghost code in Figure 3.19. They are similar in structure and update the order of events in the abstract history in the same way described by `OrderInsRem`. That is, these commitment points maintain a linear order on completed events of operations with the same argument: on the first line of `OrderInsRem`, the current `insert`/`remove` event identified by `myEid()` gets ordered after each operation e with the same argument as `myEid()`. On the second line of `OrderInsRem`, uncompleted `insert` and `remove` events with the same argument are ordered after `myEid()`. Note that uncompleted `contains` events remain unordered w.r.t. `myEid()`, so that later on at the commitment point of `contains` they could be ordered before the current `insert` or `remove` operation (depending on whether they return false or true accordingly), if it is necessary.

At the commitment point, the `remove` operation assigns a return value to the corresponding event. When the removal is successful, the commitment point associates the removed node with the event by updating G_{node}. Let us illustrate how commit$_{\texttt{remove}}$ changes abstract histories on the example. For the execution in Figure 3.17, after starting the operation "Rem(2)" we have the abstract history Figure 3.18(a), and then at point (B) "Rem(2)" changes the history to Figure 3.18(b). The uncompleted event "Con(2)" remains unordered w.r.t. "Rem(2)" until it determines its return value (true) later on in the execution, at

```
commit_contains() {
  E(myEid()).rval := (curr.val = v);
  EventID obs:= if (curr.val = v) then insOf(E, curr)
                                  else lastRemOf(E, R, v);
  if (obs ≠ ⊥)
    R := (R ∪ {(obs, myEid())})⁺;
  R := (R ∪ {(myEid(), i) | ¬(i →_R myEid()) ∧ E(i).arg = E(myEid()).arg})⁺;
}
```

where for an abstract history (E, R), a node identifier n and a value v:

$$\mathsf{insOf}(E, n) = \begin{cases} i, \text{if } G_{\mathsf{node}}(i) = n, E(i).\mathsf{op} = \mathsf{insert} \text{ and } E(i).\mathsf{rval} = \mathsf{true} \\ \text{undefined otherwise} \end{cases}$$

$$\mathsf{lastRemOf}(E, R, v) = \begin{cases} i, & \text{if } E(i) = (\_, \mathsf{remove}, v, \mathsf{true}) \\ & \quad \wedge (\forall i'.\, E(i).\mathsf{op} = \mathsf{remove} \wedge E(i').\mathsf{arg} = v \implies \\ & \qquad\qquad i' \xrightarrow{R} i) \\ \bot, & \text{if } \neg\exists i.\, E(i) = (\_, \mathsf{remove}, v, \mathsf{true}) \end{cases}$$

Figure 3.20: The auxiliary code executed at the commitment point of `contains`

which point it gets ordered before "Rem(2)".

At the commitment point, the `insert` operation assigns a return value to the event based on the check $\mathtt{c.val} \neq \mathtt{v}$ determining whether `v` is already in the set. In the execution Figure 3.17, prior to the start of "Ins(3)" we have the abstract history Figure 3.18(c). When the event starts, a new event is added into the history (commitment point of kind 1), which changes it to Figure 3.18(d). At point (D) in the execution, $\mathtt{commit}_{\mathtt{insert}}$ takes place, and the history is updated to Figure 3.18(e). Note that "Ins(3)" and "Con(3)" remain unordered until the latter determines its return value (false) and orders itself before "Ins(3)" in the abstract history.

The commitment point at lines 40–42 of the `contains` operation occurs at the last iteration of the sorted list traversal in the `locate` method. The last iteration takes place when $\mathtt{curr.val} \geq v$ holds. In Figure 3.20, we present the auxiliary code $\mathtt{commit}_{\mathtt{contains}}$ executed at line 42 in this case. Depending on whether a requested value is found or not, the abstract history is updated differently, so we further explain the two cases separately. In both cases, the `contains` operation determines which event in the history it should immediately follow in all linearizations.

Case (i). If `curr.val = v`, the requested value `v` is found, so the current event `myEid`() receives `true` as its return value. In this case, $\mathtt{commit}_{\mathtt{contains}}$ adds two kinds of edges in the abstract history.

- Firstly, $(\mathsf{insOf}(E, \mathtt{curr}), \mathtt{myEid}())$ is added to ensure that `myEid`() occurs in all linearizations of the abstract history after the `insert` event of the node `curr`.
- Secondly, $(\mathtt{myEid}(), i)$ is added for every other identifier i of an event that does not precede `myEid`() and has an argument v. The requirement not to

precede myEid() is explained by the following. Even though at commitment points of insert and remove operations we never order events w.r.t. contains events, there still may be events preceding myEid() in real-time order. Consequently, it may be impossible to order myEid() immediately after insOf(E, curr).

At point (C) in the example from Figure 3.17, $\texttt{commit}_{\texttt{contains}}$ in "Con(2)" changes the history from Figure 3.18(b) to Figure 3.18(c). To this end, "Con(2)" is completed with a return value true and gets ordered after "Ins(2)" (this edge happened to be already in the abstract history due to the real-time order), and also in front of events following "Ins(2)", but not preceding "Con(2)". This does not include "Ins(4)" due to the real-time ordering, but includes "Rem(2)", so the latter is ordered after the contains event, and all linearizations of the abstract history Figure 3.18(c) meet the sequential specification in this example. In general case, we also need to show that successful remove events do not occur between insOf(E, curr), myEid()) and myEid() in the resulting abstract history, which we establish formally in Appendix A.2. Intuitively, when myEid() returns true, all successful removes after insOf(E, curr) are concurrent with myEid(): if they preceded myEid() in the real-time order, it would be impossible for the contains operation to reach the removed node by starting from the head of the list in order return true.

Case (ii). Prior to executing $\texttt{commit}_{\texttt{contains}}$, at line 40 we check that curr.val $\geq$ v. Thus, if curr.val = v does not hold in $\texttt{commit}_{\texttt{contains}}$, the requested value v is not found in the sorted list, and false becomes the return value of the current event myEid(). In this case, $\texttt{commit}_{\texttt{contains}}$ adds two kinds of edges in the abstract history.

- Firstly, (lastRemOf($E, R,$ v), myEid()) is added, when there are successful remove events of value v (note that they are linearly ordered by construction of the abstract history, so we can choose the last of them). This ensures that myEid() occurs after a successful remove event in all linearizations of the abstract history.

- Secondly, (myEid(), i) is added for every other identifier i of an event that does not precede myEid() and has an argument v, which is analogous to the case (i).

Intuitively, if v has never been removed from the set, myEid() needs to happen in the beginning of the abstract history and does not need to be ordered after any event.

For example, at point (D) in the execution from Figure 3.17, $\texttt{commit}_{\texttt{contains}}$ changes the abstract history from Figure 3.18(e) to Figure 3.18(f). To this end, "Con(3)" is ordered in front of all events with argument 3 (specifically, "Ins(3)"), since there are no successful removes of 3 in the abstract history. Analogously to the case (i), in general to ensure that all linearizations of the resulting abstract history meet the sequential specification, we need to show that there cannot be any successful insert events of v between lastRemOf($E, R,$ v) (or the beginning of the abstract history, if it is undefined) and myEid(). We prove this formally in Appendix A.2. Intuitively, when myEid() returns false, all successful insert events after lastRemOf($E, R,$ v) (or the beginning of the history) are concurrent with myEid(): if they preceded myEid() in the real-time order, the inserted nodes

would be possible to reach by starting from the head of the list, in which case the `contains` operation could not possibly return false.

3.8 Summary and Related Work

There has been a great deal of work on proving algorithms linearizable; see [23] for a broad survey. However, despite a large number of techniques, often supported by novel mathematical theory, it remains the case that all but the simplest algorithms are difficult to verify. Our aim is to verify the most complex kind of linearizable algorithms, those where the linearization of a set of operations cannot be determined solely by examining the prefix of the program execution consisting of these operations. Furthermore, we aim to do this while maintaining a relatively simple proof argument.

Much work on proving linearizability is based on different kinds of *simulation proofs*. Loosely speaking, in this approach the linearization of an execution is built incrementally by considering either its prefixes or suffixes (respectively known as *forward* and *backward* simulations). This supports inductive proofs of linearizability: the proof involves showing that the execution and its linearization stay in correspondence under forward or backward program steps. The linearization point method is an instance of forward simulation: a syntactic point in the code of an operation is used to determine when to add it to the linearization.

As we explained in the beginning of the chapter, forward simulation alone is not sufficient in general to verify linearizability. However, Schellhorn et al. [48] prove that backward simulation alone *is* always sufficient.They also present a proof technique and use it to verify the Herlihy-Wing queue [7]. However, backwards simulation proofs are difficult to understand intuitively: programs execute forwards in time, and therefore it is much more natural to reason this way.

The queue originally proposed by Herlihy and Wing in their paper on linearizability [7] has proved very difficult to verify. Their proof sketch is based on reasoning about the possible linearizations arising from a given queue configuration. Our method could be seen as being midway between this approach and linearization points. We use partiality in the abstract history to represent sets of possible linearizations, which helps us simplify the proof by omitting irrelevant ordering (§3.1).

Another class of approach to proving linearizability is based on special-purpose program logics. These can be seen as a kind of forward simulation: assertions in the proof represent the connection between program execution and its linearization. To get around the incompleteness of forward simulation, several authors have introduced auxiliary notions that support limited reasoning about future behavior in the execution, and thus allow the proof to decide the order of operations in the linearization [24, 25, 33]. However, these new constructs have subtle semantics, which results in proofs that are difficult to understand intuitively.

Our approach is based on program logic, and therefore is a kind of forward simulation. The difference between us and previous program logics is that we do not explicitly construct a linear order on operations, but only a partial order. This removes the need for special constructs for reasoning about future behavior,

but creates the obligation to show that the partially ordered abstract history can always be linearized.

One related approach to ours is that of Hemed et al. [49], who generalize linearizability to data structures with concurrent specifications (such as barriers) and propose a proof method for establishing it. To this end, they also consider histories where some events are partially ordered—such events are meant to happen concurrently. However, the goal of Hemed et al.'s work is different from ours: their abstract histories are never linearized, to allow concurrent specifications; in contrast, we guarantee the existence of a linearization consistent with a sequential specification. It is likely that the two approaches can be naturally combined.

Linking-in-time [50] is another approach to reasoning about algorithms with future-dependent linearizations developed concurrently with ours by Delbianco et al. Their proof of logical atomicity of Jayanti's snapshot algorithm can be seen as an extension of the linearization-points method with means for *relinking* linearization order. That is, they show that the linearization order in the proof of Jayanti's algorithm can always be altered upon discovering that a speculative decision about the location of linearization points of operation has been wrong. In contrast, in our approach instead of making a speculative decision about the total ordering of events, we maintain a partial order and prove that it represents only correct linearizations.

Aspect proofs [35] are a non-simulation approach that is related to our work. An aspect proof imposes a set of forbidden shapes on the real-time order on methods; if an algorithm avoids these shapes, then it is necessarily linearizable. These shapes are specific to a particular data structure, and indeed the method as proposed in [35] is limited to queues (extended to stacks in [36]). In contrast, our proof method is generic, not tied to a particular kind of data structure. Furthermore, checking the absence of forbidden shapes in the aspect method requires global reasoning about the whole program execution, whereas our approach supports inductive proofs. The original proof of the TS stack used an extended version of the aspect approach [36]. However, without a way of reasoning inductively about programs, the proof of correctness reduced to a large case-split on possible executions. This made the proof involved and difficult. Our proof is based on an inductive argument, which makes it easier.

Another class of algorithms that are challenging to verify are those that use *helping*, where operations complete each others' work. In such algorithms, an operation's position in the linearization order may be fixed by a helper method. Our approach can also naturally reason about this pattern: the helper operation may modify the abstract history to mark the event of the operation being helped as completed.

The Optimistic set was also proven linearizable by O'Hearn et al. in [19]. The essence of the work is a collection of lemmas (including the Hindsight Lemma) proven outside of the logic to justify conclusions about properties of the past of executions based on the current state. Based on our case study of the Optimistic set algorithm, we conjecture that at commitment points we make a constructive decision about extending abstract history where the hindsight proof would use the Hindsight Lemma to non-constructively extend a linearization with the `contains` operation.

3.8.1 Summary

The popular approach to proving linearizability is to construct a total linearization order by appending new operations as the program executes. This approach is straightforward, but is limited in the range of algorithms it can handle. In this chapter, we present a new approach which lifts these limitations, while preserving the appealing incremental proof structure of traditional linearization points. As with linearization points, our fundamental idea can be explained simply: at commitment points, operations impose order between themselves and other operations, and all linearizations of the order must satisfy the sequential specification. Nonetheless, our technique generalizes to far more subtle algorithms than traditional linearization points.

We have applied our approach to two algorithms known to present particular problems for linearization points. Although, we have not presented it here, our approach scales naturally to *helping*, where an operation is completed by another thread. We can support this, by letting any thread complete the operation in an abstract history. In future work, we plan to apply our approach to the Time-Stamped stack [36], which poses verification challenges similar to the TS queue; a flat-combining style algorithm, which depends fundamentally on helping, as well as a range of other challenging algorithms.

Chapter 4

Safe Privatization in Transactional Memory

Transactional memory (TM) facilitates the development of concurrent applications by letting the programmer designate certain code blocks as *atomic* [1]. TM allows developing a program and reasoning about its correctness as if each atomic block executes as a *transaction*—atomically and without interleaving with other blocks—even though in reality the blocks can be executed concurrently. This guarantee can be formalized as *observational refinement* [51]: every behavior a user can observe of a program using a TM implementation can also be observed when the program uses an abstract TM that executes each block atomically. A TM can be implemented in hardware [3, 4], software [2] or a combination of both [5, 6].

Often programmers using a TM would like to access the same data both inside and outside transactions. This may be desirable to avoid performance overheads of transactional accesses, to support legacy code, or for explicit memory deallocation. One typical pattern is *privatization* [12], illustrated in Figure 4.1(a). There the `atomic` blocks return a value signifying whether the transaction committed or aborted. In the program, an object `x` is guarded by a flag `x_is_private`, showing whether the object should be accessed transactionally (false) or non-transactionally (true). The left-hand-side thread first tries to set the flag inside transaction T_1, thereby *privatizing* `x`. If successful, it then accesses `x` non-transactionally. A concurrent transaction T_2 in the right-hand-side thread checks the flag `x_is_private` prior to accessing `x`, to avoid simultaneous transactional and non-transactional access to the object. We expect the postcondition shown to hold: if privatization is successful, at the end of the program `x` should store 1, not 42. The opposite idiom is *publication*, illustrated in Figure 4.2. The left-hand-side thread writes to `x` non-transactionally and then clears the flag `x_is_private` inside transaction T_1, thereby *publishing* `x`. The right-hand-side thread tests the flag inside transaction T_2, and if it is cleared, reads `x`. Again, we expect the postcondition to hold: if the right-hand-side thread sees the write to the flag, it should also see the write to `x`. The two idioms can be combined: the programmer may privatize an object, then access it non-transactionally, and then publish it back for transactional access.

Ideally, programmers mixing transactional and non-transactional accesses to

(a) Delayed commit problem:

```
{ x_is_private = false ∧ x = 0 }
l := atomic {          ||  atomic { // T2
    // T1              ||    if (!
  x_is_private         ||      x_is_private
     = true;           ||      ) {
}                      ||      x = 42;
if (l ==               ||    }
    committed)         ||  }
  x = 1; // ν          ||
  { l = committed  ⟹  x = 1 }
```

(b) Doomed transaction problem:

```
{ x_is_private = false ∧ x = 0 }
l := atomic {          ||  atomic { // T2
    // T1              ||    if (!
  x_is_private         ||      x_is_private
     = true;           ||      ) {
}                      ||      while (x
if (l ==               ||      == 1) {}
    committed)         ||    }
  x = 1; // ν          ||  }
```

Figure 4.1: Privatization examples.

objects would like the TM to guarantee *strong atomicity* [14], where transactions can be viewed as executing atomically also with respect to non-transactional accesses, i.e., without interleaving with them. This is equivalent to considering every non-transactional access as a single-instruction transaction. For example, the program in Figure 4.3 under strongly atomic semantics can only produce executions where each of the non-transactional accesses ν_1 and ν_2 executes either before or after the transaction T, so that the postcondition in Figure 4.3 always holds.

Unfortunately, providing strong atomicity in software requires instrumenting non-transactional accesses with additional instructions for maintaining TM metadata. This undermines scalability and makes it difficult to reuse legacy code. Since most existing TMs are either software-based or rely on a software fall-back, they do not perform such instrumentation and, hence, provide weaker atomicity guarantees. For example, they may allow the program in Figure 4.3 to execute non-transactional accesses ν_1 and ν_2 between transactional writes to `x` and `y` and, thus, observe an intermediate state of the transaction, e.g., $\texttt{x} = 1$ and $\texttt{y} = 0$, which violates the postcondition in Figure 4.3.

Researchers have suggested resolving the tension between strong TM semantics and performance by taking inspiration from non-transactional shared-memory models, which are subject to the same problem: optimizations performed by processors and compilers weaken the guarantee of *sequential consistency* [52] ideal for this setting. The compromise taken is to guarantee sequential consistency for certain *data-race free (DRF)* programs, which do not access the same data concurrently without synchronization [53]. Racy programs either are allowed to produce non-sequentially-consistent behaviors [54], or are declared

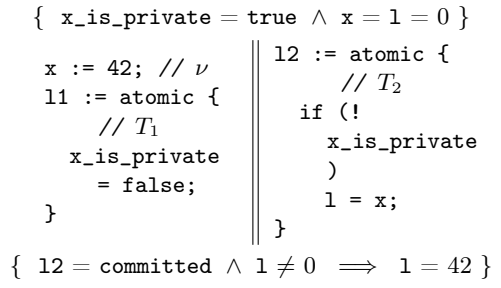


Figure 4.2: Publication example.

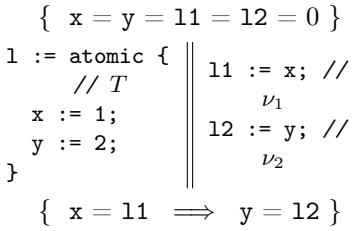


Figure 4.3: A racy example.

faulty and thus having no semantics at all [55]. DRF thus establishes a contract between the programmer and the run-time system, which can be formalized by the so-called *Fundamental Property*: if a program is DRF *assuming* the strong semantics (such as sequential consistency), then the program does have the strong semantics. The crucial feature of this property is that DRF is checked by considering only executions under the strong semantics, which relieves the programmer from having to reason about the weaker semantics of unrestricted programs. The DRF contract has formed the basis of the memory models of both Java [54] and C/C++ [55].

Applying the above approach to TM, strong atomicity would be guaranteed only for programs that do not have an analog of data races in this setting—informally, concurrent transactional and non-transactional accesses to the same data [56, 57, 12, 58, 20, 21]. For example, we do not want to guarantee strong atomicity for the program in Figure 4.3, which has such concurrent accesses to `x` and `y`. On the other hand, the programs in Figure 4.1 and Figure 4.2 should be guaranteed strong atomicity, since at any point of time, an object is accessed either only transactionally or only non-transactionally. Unfortunately, whereas the DRF contract in non-transactional memory models has been worked out in detail, the situation in transactional models remains unsettled. There is currently no consensus on a single definition of transactional DRF: there are multiple competing proposals [20, 21, 59, 60, 56], which often come without a formal justification similar to the Fundamental Property of non-transactional memory models.

This dissertation makes a step towards a definition of transactional DRF on a par with solutions in non-transactional memory models. A key technical challenge we tackle is that many TM implementations, when used out-of-the-box, do not guarantee strong atomicity for seemingly well-behaved programs using privatization, such as the ones in Figure 4.1 [22, 61, 62, 63]. For example, such

TMs may invalidate the postcondition of the program in Figure 4.1(a) due to the *delayed commit* problem [12]. In more detail, many software TM implementations execute transactions optimistically, buffering their writes, and flush them to memory only on commit. In this case, it is possible for the transaction T_1 to privatize `x` and for ν to modify it after T_2 started committing, but before its write to `x` reached the memory, so that T_2's write subsequently overwrites ν's write and violates the postcondition. TMs that make transactional updates in-place and undo them on abort are subject to a similar problem. In Figure 4.1(b) we give another privatization example that is prone to a different problem—the *doomed transaction* problem [12]. A TM may execute T_2's read from `x_is_private`, and then T_1 and ν. Because T_1 modifies `x_is_private`, at this point T_2 is "doomed", i.e., guaranteed to abort if it finishes executing. But if the non-transactional write ν is uninstrumented and ignores the metadata the TM maintains to ensure the consistency of reads, T_2 will read the value written to `x` by ν and enter an infinite loop. This would never happen under strong atomicity, where T_1 and ν may not execute while T_2 is running.

A possible solution to the above problems is for the compiler or the programmer to insert special *transactional fences* [12]. These have semantics similar to *read-copy-update (RCU)* [64]: a fence blocks until all the transactions that were active when it was invoked complete, by either committing or aborting. For example, assume we insert a fence between the transaction T_1 and the non-transactional access ν in Figure 4.1(a). Then the delayed commit problem does not arise: if T_2 enters the `if` body and writes to `x`, then it must begin before the fence does; thus the fence will wait until T_2 completes and flushes its write to memory, so that T_2 cannot incorrectly overwrite ν. Analogously, a fence between T_1 and ν in Figure 4.1(b) ensures that the doomed transaction problem does not arise: if T_2 reaches the while loop, then ν cannot execute before T_2 finishes, and thus the while loop immediately terminates.

Unfortunately, inserting transactional fences conservatively after every transaction, even when not required, undermines scalability. For example, Yoo et al. [65] showed that unnecessarily fencing a selection of transactional benchmarks leads to overheads of 32% on average and 107% in the worst case, the latter on one of the STAMP benchmarks [66]. For this reason, researchers have suggested placing transactional fences selectively, e.g., according to programmer annotations [65]. However, omitting fences without violating strong atomicity is nontrivial: for example, for several years the TM implementation in the GCC compiler had a buggy placement of transactional fences that omitted them after read-only transactions; this has recently been shown to violate strong atomicity [67]. To make sure such bugs are not habitual, we need a notion of transactional DRF that would take into account selective fence placements.

In this chapter we propose just such a notion and formalize its Fundamental Property using observational refinement: if a program is DRF under strong atomicity (formalized as *transactional sequential consistency* [20, 21]), then all its executions are observationally equivalent to strongly atomic ones. We furthermore prove that the Fundamental Property holds under a certain condition on the TM, generalizing opacity [10, 68], which we call *strong opacity*. Thus, similarly to non-transactional memory models, the programmer writing code that has no data races according to our notion never needs to reason about weakly atomic semantics.

Our results thus establish a contract between client programs and TM im-

plementations sufficient for strong atomicity. Of course, for this contract to be useful, it should not overconstrain either of its participants: programmers should be able to use the typical programming idioms, and common TM implementations should satisfy strong opacity that we require. In this chapter we argue that this is indeed the case.

On the client side, our DRF notion allows the programmer to use privatization and publication idioms—programs in Figure 4.2 and in Figure 4.1 with a fence between T_1 and ν are considered data-race free and thus guaranteed strong atomicity. We hence view privatization and publication idioms as just particular ways of ensuring data-race freedom.

To justify appropriateness of our requirements on TM systems, we develop a method for proving that a TM satisfies strong opacity for DRF programs. Our method is *modular*: it requires only a minimal adjustment to a proof of the usual opacity of the TM assuming no mixed transactional/non-transactional accesses. We demonstrate the effectiveness of the method by applying it to prove the strong opacity of a realistic TM, TL2 [22], enhanced with transactional fences implemented using RCU. Our proof shows that this TM will indeed guarantee strong atomicity to programs satisfying our notion of DRF.

Thus, this dissertation makes the first proposal of transactional DRF that considers a flexible programming model (with transactional fences) and comes with a formal justification of its appropriateness—the Fundamental Property and the notion of TM correctness required for it to hold.

4.1 Programming Language

We now introduce a simple programming language with mixed transactional/non-transactional accesses, for which we formalize our results. We also define the semantics of the language when using a given TM implementation. As a special case of this semantics, we get the notion of strong atomicity [14] (also called transactional sequential consistency [20]): this is obtained by instantiating our semantics with a special idealized *atomic* TM where the execution of transactions does not interleave with that of other transactions or with non-transactional accesses.

4.1.1 Programming Language Syntax

A *program* $P = C_1 \parallel \ldots \parallel C_N$ in our language is a parallel composition of *commands* C_t executed by different *threads* $t \in \mathsf{ThreadID} = \{1, \ldots, N\}$. Every thread $t \in \mathsf{ThreadID}$ has a set of *local variables* $l \in \mathsf{LVar}_t$, which only it can access; for simplicity, we assume that these are integer-valued. Threads have access to a *transactional memory* (*TM*), which manages a fixed collection of *shared register objects* $x \in \mathsf{Reg}$. The syntax of commands $C \in \mathsf{Com}$ is as follows:

$$\begin{array}{rcl} C & = & \alpha \mid C\,;C \mid \texttt{if}\ (b)\ \texttt{then}\ C\ \texttt{else}\ C \mid \texttt{while}\ (b)\ \texttt{do}\ C \\ & \mid & l := \texttt{atomic}\,\{C\} \mid l := x.\texttt{read}() \mid x.\texttt{write}(e) \mid \texttt{fence} \end{array}$$

where b and e denote Boolean, respectively, integer *expressions* over local variables and constants. The language includes *primitive commands* $\alpha \in \mathsf{PCom}$, which operate on local variables, and standard control-flow constructs.

An *atomic block* $l := \texttt{atomic}\,\{C\}$ executes C as a *transaction*, which the TM can *commit* or *abort*. The system's decision is returned in the local variable l, which gets assigned a distinguished value committed or aborted. We do not allow programs to abort a transaction explicitly, and forbid nested atomic blocks and, hence, nested transactions.

Commands can invoke two methods on a shared register x: $x.\texttt{read}()$ returns the current value of x, and $x.\texttt{write}(e)$ sets it to e. Threads may call these methods both *inside* and *outside* atomic blocks. We refer to the former as a *transactional* accesses and to the latter as a *non-transactional* accesses. To make our presentation more approachable, following [68] we assume that each write in a single program execution writes a distinct value. Finally, the language includes a *transactional fence* command `fence`, which acts as previously explained in the beginning of this chapter. It may only be used outside transactions.

The simplicity of the above language allows us to clearly explain our contributions. We leave handling advanced features, such as nested transactions [69, 70] and nested synchronization [71] as future work.

4.1.2 A Trace-based Model of Computations

To define the semantics of our programming language, we need a formal model for program computations. To this end, we introduce *traces*—certain finite sequences of *actions*, each describing a single computation step (for simplicity, in this chapter we consider only finite computations). Let ActionId be a set of *action identifiers*. Actions are of two kinds. A *primitive action* denotes the execution of a primitive command and is of the form (a, t, α), where $a \in \mathsf{ActionId}$, $t \in \mathsf{ThreadID}$ and $\alpha \in \mathsf{PCom}$. A *TM interface action* has one of the forms shown in Figure 4.4. We use ψ to range over actions.

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. The request actions $\mathsf{write}(x, v)$ and $\mathsf{read}(x)$ denote invocations of the `write`, respectively, `read` methods of register x; a `write` action is annotated with the value v written. The response actions $\mathsf{ret}(\bot)$ and $\mathsf{ret}(v)$ denote the return from invocations of `write`, respectively, `read` methods of a register; the latter is annotated with the value v read. For reasons explained below, we consider non-transactional accesses to registers as calling into the TM, and hence use the same actions for them as for transactional accesses. The TM may abort a transaction (but not a non-transactional access) at any point when it is in control; this is recorded by an aborted response action. The actions fbegin and fend denote the beginning, respectively, the end of the execution of a `fence` command. In the following _ denotes an irrelevant expression.

To formalize restrictions on accesses to variables by primitive commands, we partition the set PCom into m classes: $\mathsf{PCom} = \biguplus_{t=1}^{m} \mathsf{LPcomm}_t$. The intention is that commands from LPcomm_t can access only the local variables of thread t (LVar_t). To ensure that in our programming language 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 LPcomm_t.

Request actions	Matching response actions
$(a, t, \mathsf{txbegin})$	$(a, t, \mathsf{ok}) \mid (a, t, \mathsf{aborted})$
$(a, t, \mathsf{txcommit})$	$(a, t, \mathsf{committed}) \mid (a, t, \mathsf{aborted})$
$(a, t, \mathsf{write}(x, v))$	$(a, t, \mathsf{ret}(\bot)) \mid (a, t, \mathsf{aborted})$
$(a, t, \mathsf{read}(x))$	$(a, t, \mathsf{ret}(v)) \mid (a, t, \mathsf{aborted})$
(a, t, fbegin)	(a, t, fend)

Figure 4.4: TM interface actions. Here $a \in \mathsf{ActionId}$, $t \in \mathsf{ThreadID}$, $x \in \mathsf{Reg}$, and $v \in \mathbb{Z}$.

Definition 4.1. *A* trace τ *is a finite sequence of actions satisfying the following well-formedness conditions:*

1. *every action in* τ *has a unique identifier: if* $\tau = \tau_1\,(a_1, \_, \_)\,\tau_2\,(a_2, \_, \_)\,\tau_3$ *then* $a_1 \neq a_2$*;*

2. *commands in actions executed by a thread* t *do not access local variables of other threads* $t' \neq t$*: if* $\tau = \_\ (\_, t, \alpha)\ \_$ *then* $\alpha \in \mathsf{LPcomm}_t$*;*

3. *every* `write` *operation writes a unique value: if* $\tau = \_\ (\_, \_, \mathsf{write}(\_, v))\ \_\ (\_, \_, \mathsf{write}(\_, v'))\ \_$ *then* $v \neq v'$*;*

4. *for every thread* t*, the projection* $\tau|_t$ *of* τ *onto the actions by* t *cannot contain a request action immediately followed by a primitive action: if* $\tau|_t = \_\,\psi_1\psi_2\,\_$ *and* ψ_1 *is a request then* ψ_2 *is a response;*

5. *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;*

6. *actions denoting the beginning and end of transactions are properly matched: for every thread* t*, in the projection of* $\tau|_t$ *to* txbegin*,* committed *and* aborted *actions,* txbegin *alternates with* committed *or* aborted*, starting from* txbegin*;*

7. *non-transactional accesses execute atomically: if* $\tau = \tau_1\,\psi\,\tau_2$*, where* ψ *is a* read *or a* write *request action by thread* t*, and all the transactions of* t *in* τ_1 *completed, then* τ_2 *begins with a response to* ψ*.*

8. *non-transactional accesses never abort: if* $\tau = \_\,\psi_1\,\psi_2\,\tau_2$*, where* ψ_1 *is a non-transactional request action then* ψ_2 *is not an* aborted *action;*

9. *fence actions may not occur inside transactions; if* $\tau = \tau_1\,(t, \mathsf{fbegin})\,\_$ *the all the transactions* t *in* τ_1 *completed; and*

10. *fence blocks until all active transactions complete: if* $\tau = \tau_1\,(\_, t, \mathsf{txbegin})\,\tau_2\,(\_, t', \mathsf{fbegin})\,\tau_3\,(\_, t', \mathsf{fend})\,\tau_4$ *then either* τ_2 *or* τ_3 *contains an action of the form* $(\_, t, \mathsf{committed})$ *or* $(\_, t, \mathsf{aborted})$*.*

We denote the set of traces by Trace and the set of actions in a trace τ by $\mathsf{act}(\tau)$. For a trace $\tau = \tau_0\_$, where τ_0 is also a trace, we say that τ_0 is a prefix of τ.

A *transaction* T is a nonempty 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. 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.

A transaction T is in a trace τ if T is a subsequence of τ and no longer transaction is. We let $\mathsf{txns}(\tau)$ be the set of transactions in τ.

We refer to TM interface actions in a trace outside of a transaction as *non-transactional actions*. We call a matching request/response pair of a read or a write a *non-transactional access*. We denote by $\mathsf{nontxn}(\tau)$ the set of non-transactional accesses in τ and range over them by ν.

A *history* is a trace containing only TM interface actions; we use H, S to range over histories. Since histories fully capture the possible interactions between a TM and a client program, we often conflate the notion of a TM and the set of histories it produces. Hence, a *transactional memory* $\mathcal{H}$ is a set of histories that is prefix-closed and closed under renaming action identifiers. Note that histories include actions corresponding to non-transactional accesses, even though these may not be directly managed by the TM implementation. This is needed to account for changes to registers performed by such actions when defining the TM semantics: e.g., in the case when a register is privatized, modified non-transactionally and then published back for transactional access. Of course, a well-formed TM semantics should not impose restrictions on the placement of non-transactional actions, since these are under the control of the program.

4.1.3 Programming Language Semantics

The *semantics* of the programming language is the set of traces that computations of programs produce. A *state* of a program $P = C_1 \parallel \ldots \parallel C_N$ records the values of all its variables: $s \in \mathsf{State} = (\biguplus_{t=1}^{N} \mathsf{LVar}_t) \to \mathbb{Z}$. The semantics of a program P is given by the set of traces $[\![P, \mathcal{H}]\!](s) \subseteq \mathsf{Trace}$ it produces when executed with a TM $\mathcal{H}$ 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 executed from s with the behavior of the TM unrestricted, i.e., considering all possible values the TM can return on reads and allowing transactions to commit or abort arbitrarily. This definition follows the intuitive semantics of our programming language. We then restrict $[\![P]\!](s)$ to the set of traces produced by P when executed with $\mathcal{H}$ by selecting those traces that interact with the TM in a way consistent with $\mathcal{H}$: $[\![P, \mathcal{H}]\!](s) = \{\tau \mid \tau \in [\![P]\!](s) \wedge \mathsf{history}(\tau) \in \mathcal{H}\}$, where $\mathsf{history}(\cdot)$ projects to TM interface actions.

The set $[\![P]\!](s)$ is itself 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:

$$x := 1;\ \texttt{if}\,(x = 1)\ y := 1\ \texttt{else}\ y := 2$$

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 $\lightning$ 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 $\texttt{assume}(b)$, where b is a Boolean expression over local variables of thread t, defining the condition. We state their semantics formally below; informally, $\texttt{assume}(b)$ 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 4.5 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 $\lightning$ 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 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 which respect Definition 4.1, as denoted by the prefix operator. We take prefix closure here (while respecting the atomicity of non transactional access) 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 by considering all possible unfoldings of the loop body. Again, we record branching decisions using `assume` actions.

The set of action sequences of `read` and `write` accesses includes both sequences where the access executes successfully and where the current transaction is aborted. The former set is constructed by nondeterministically choosing an integers v to describe the the return and parameter for the `read` and `write` accesses, respectively. To ensure that e indeed evaluates to v, in the case of a `write`, Note that some of the choices here might not be feasible: the chosen v might not be the value of the parameter expression e when the method is invoked. infeasible choices are filtered out at the following stages of the seman-

$$
\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 & = & \{\tau_1\,\tau_2 \ldots \tau_{2n}\,(\_, t, \mathsf{assume}(\neg b)) \mid n \in \mathbb{N} \wedge \forall j \in \{1, \ldots, n\}.\, \tau_{2j-1} = (\_, t, \mathsf{assume}(b)) \\
 & & \qquad\qquad \wedge\, \tau_{2j} \in A'(C)t\} \cup \{(\_, t, \mathsf{assume}(\neg b))\} \\
A'(l := x.\texttt{read}())t & = & \{(\_, t, \mathsf{read}(x))\,(\_, t, \mathsf{ret}(v))\,(\_, t, l := v) \mid v \in \mathbb{Z}\} \cup \{(\_, t, \mathsf{read}(x))\,(\_, t, \mathsf{aborted})\} \\
A'(x.\texttt{write}(e))t & = & \{(\_, t, \mathsf{assume}(e = v))\,(\_, t, \mathsf{write}(x, v))\,(\_, t, \mathsf{ret}(\bot))) \mid v \in \mathbb{Z}\} \\
 & & \cup\, \{(\_, t, \mathsf{assume}(e = v))\,(\_, t, \mathsf{write}(x, v))\,(\_, t, \mathsf{aborted}) \mid v \in \mathbb{Z}\} \\
A'(\texttt{fence})t & = & \{(\_, t, \mathsf{fbegin})\,(\_, t, \mathsf{fend})\} \\
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 \\
 & & \qquad\qquad \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 \\
 & & \qquad\qquad \tau \in A'(C)t \wedge (\_, 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 \leq t \leq m \implies \tau_t \in A'(C_t)t\}) \\
A(P) & = & A'(P) \cap \mathsf{Trace}
\end{array}
$$

Figure 4.5: The definition of $A(P)$.

tics 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 $[\![P, \mathcal{H}]\!](s)$ by selecting the sequences from $[\![P]\!](s)$ that interact with the transactional memory correctly. The set of action sequences of a `fence` command is comprised of all traces comprised of a fbegin action followed by a fend action, indicating that once a `fence` command is invoked, the thread gets blocked until it ends. 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 $\alpha \in \mathsf{PCom}$, given by a function $[\![\alpha]\!]$ that defines how the program state is transformed by executing α. 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 $[\![\alpha]\!]$ as a function of only those variables that α is allowed to access. Namely, the semantics of $\alpha \in \mathsf{LPcomm}_t$ is given by

$$[\![c]\!] : (\mathsf{LVar}_t \to \mathbb{Z}) \to \mathcal{P}(\mathsf{LVar}_t \to \mathbb{Z}).$$

Note that we allow α to be non-deterministic.

For a valuation q of variables that α is allowed to access, $[\![\alpha]\!](q)$ 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 $l := l'$ has the following semantics:

$$[\![l := l']\!](q) = \{q[l \mapsto q(l')]\}\ .$$

We define the semantics of `assume` commands following the informal explanation given at the beginning of this section: for example,

$$[\![\texttt{assume}(l = v)]\!](q) = \begin{cases} \{q\}, & \text{if } q(l) = v; \\ \emptyset, & \text{otherwise.} \end{cases} \tag{4.2}$$

Thus, when the condition in `assume` does not hold of q, the command stops the computation by not producing any output.

We lift functions $[\![\alpha]\!]$ to full states by keeping the variables that α is not allowed to access unmodified and producing $\lightning$ if α 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. (For simplicity, we assume commands to not fault.)

Trace evaluation. Using the semantics of primitive commands, we first define the evaluation of a single action on a given state:

$$\begin{aligned} &\mathsf{eval} : \mathsf{State} \times \mathsf{Action} \to \mathcal{P}(\mathsf{State}) \\ &\mathsf{eval}(s, (\_, t, c)) \;=\; [\![c]\!](s); \\ &\qquad \mathsf{eval}(s, \psi) \;=\; \{s\}. \end{aligned}$$

Note that this does not change the state s as a result of TM interface or `fence` 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:

$$\mathsf{eval} : \mathsf{State} \times \mathsf{Trace} \to \mathcal{P}(\mathsf{State})$$
$$\mathsf{eval}(s,\tau) = \begin{cases} \emptyset, & \text{if } \tau = \tau'\varphi \wedge \mathsf{eval}(s,\tau') = \emptyset; \\ \mathsf{evalna}(s, \tau|_{\neg\mathsf{abortact}}), & \text{otherwise,} \end{cases}$$

where $\tau|_{\neg\mathsf{abortact}}$ denotes the trace obtained from τ by removing all actions inside aborted transactions, and

$$\mathsf{evalna}(s,\tau) = \begin{cases} \{s\}, & \text{if } \tau = \varepsilon; \\ \{s'' \in \mathsf{eval}(s',\varphi) \mid s' \in \mathsf{evalna}(s,\tau')\}, & \text{if } \tau = \tau'\varphi. \end{cases}$$

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.

Recall that 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 (4.2) to filter out traces that make branching decisions inconsistent with the program state. For example, consider again the program "$l := 1;$ `if` $(l = 1)$ $l' := 1$ `else` $l' := 2$". 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 4.5, only the trace executing $l' := 1$ will result in a nonempty set of final states after the evaluation and, therefore, only this trace will be included into $[\![P]\!](s)$.

4.1.4 Strong Atomicity

We now define an idealized *atomic* TM $\mathcal{H}_{\mathsf{atomic}}$ where the execution of transactions does not interleave with that of other transactions or with non-transactional accesses. By instantiating the semantics of §4.1.3 with this TM, we formalize the strongly atomic semantics [14] (transactional sequential consistency [20, 21]). $\mathcal{H}_{\mathsf{atomic}}$ contains only histories that are *non-interleaved*, i.e., where actions by one transaction do not overlap with the actions of another transaction or of non-transactional accesses. Note that by definition actions pertaining to different non-transactional accesses cannot interleave. Note also that transactions in a non-interleaved history do not have to be complete. For example,

$$H_0 = (\_, t_1, \mathsf{txbegin})\,(\_, t_1, \mathsf{ok})\,(\_, t_1, \mathsf{write}(x,1))\,(\_, t_1, \mathsf{ret}(\bot))$$
$$(\_, t_1, \mathsf{txcommit})\,(\_, t_2, \mathsf{txbegin})\,(\_, t_2, \mathsf{ok})(\_, t_2, \mathsf{write}(x,2))$$
$$(\_, t_3, \mathsf{read}(x))\,(\_, t_3, \mathsf{ret}(1))$$

is non-interleaved. We have to allow such histories in $\mathcal{H}_{\mathsf{atomic}}$, because they may be produced by programs in our language, e.g., due to a non-terminating loop inside an atomic block.

We define $\mathcal{H}_{\mathsf{atomic}}$ in such a way that the changes made by a live or aborted transaction are invisible to other transactions. However, there is no such certainty in the treatment of a commit-pending transaction: the TM implementation might have already reached a point at which it is decided that the transaction will commit. Then the transaction is effectively committed, and its operations may affect other transactions [68]. To account for this, when defining $\mathcal{H}_{\mathsf{atomic}}$ we consider every possible completion of each commit-pending transaction in a history to either committed or an aborted one. Formally, we say that a history H^c is a *completion* of a non-interleaved history H if:

1. H^c is non-interleaved;
2. H^c is has no commit-pending transactions;
3. H is a subsequence of H^c; and
4. any action in H^c which is not in H is either a committed or an aborted action.

For example, we can obtain a completion of history H_0 above by inserting $(\_, t_1, \mathsf{committed})$ after $(\_, t_1, \mathsf{txcommit})$.

We define $\mathcal{H}_{\mathsf{atomic}}$ as the set of all non-interleaved histories H that have a completion H^c where every response action of a $\mathsf{read}(x)$ returns the value v in the last preceding $\mathsf{write}(x, v)$ action that is not located in an aborted or live transaction different from the one of the read; if there is no such write, the read should return the initial value v_{init}. For example, $H_0 \in \mathcal{H}_{\mathsf{atomic}}$. Hence, $\mathcal{H}_{\mathsf{atomic}}$ defines the intuitive atomic semantics of transactions.

4.2 Data-Race Freedom

A data race happens between a pair of *conflicting* actions, as defined below.

Definition 4.3. *A non-transactional request action ψ and a transactional request action ψ'* conflict *if ψ and ψ' are executed by different threads and they are read or write actions on the same register, with at least one being a write.*

For such actions to form a data race, they should be *concurrent*. As is standard, we formalize this using a *happens-before* relation on actions in a history: $\mathsf{hb}(H) \subseteq \mathsf{act}(H) \times \mathsf{act}(H)$. To streamline explanations, we first define DRF in terms of happens-before, and only after this define the latter. For a history H and an index i, let $H(i)$ denote the i-th action in the sequence H.

Definition 4.4. *Actions $H(i)$ and $H(j)$ in a history H form a* data race*, if they conflict and are not related by $\mathsf{hb}(H)$ either way. A history H is* data-race free (DRF)*, written $\mathsf{DRF}(H)$, if it has no data races.*

Definition 4.5. *A program P is* data-race free (DRF) *when executed from a state s with a TM $\mathcal{H}$, written $\mathsf{DRF}(P, s, \mathcal{H})$, if $\forall \tau \in [\![P]\!](\mathcal{H}, s).\, \mathsf{DRF}(H(\tau))$.*

Our goal is to enable programmers to ensure strong atomicity of a program by checking its data-race freedom. However, the notion of DRF depends on the TM $\mathcal{H}$, and we do not want the programmer to have to reason about the actual TM implementation. In Section 4.4, we give conditions on TM $\mathcal{H}$ under

which strong atomicity of a program is guaranteed if it is DRF *assuming* strong atomicity, i.e., $\mathsf{DRF}(P, s, \mathcal{H}_{\mathsf{atomic}})$ for $\mathcal{H}_{\mathsf{atomic}}$ from §4.1.4. We next define $\mathsf{hb}(H)$ and show examples of programs that are racy and race-free under $\mathcal{H}_{\mathsf{atomic}}$.

For a history H, we define several relations over $\mathsf{act}(H)$, which we explain in the following:

- *execution order*:
  $\alpha <_H \alpha'$ iff for some i, j we have $\alpha = H(i)$, $\alpha' = H(j)$ and $i < j$.
- *per-thread order* $\mathsf{po}(H)$:
  $\psi <_{\mathsf{po}(H)} \psi'$ iff $\psi <_H \psi'$ and actions ψ and ψ' are by the same thread.
- *restricted per-thread order* $\mathsf{xpo}(H)$:
  $\psi <_{\mathsf{xpo}(H)} \psi'$ iff $\psi <_H \psi'$, actions ψ and ψ' are by the same thread t, and there is a $(\_, t, \mathsf{txbegin})$ action between ψ and ψ'.
- *client order* $\mathsf{cl}(H)$:
  $\psi <_{\mathsf{cl}(H)} \psi'$ iff $\psi <_H \psi'$ and ψ, ψ' are non-transactional in H.
- *after-fence order* $\mathsf{af}(H)$:
  $\psi <_{\mathsf{af}(H)} \psi'$ iff $\psi <_H \psi'$, $\psi = (\_, \_, \mathsf{fbegin})$ and $\psi' = (\_, \_, \mathsf{txbegin})$, i.e., the transaction begins after the fence does (Figure 4.6(a)).
- *before-fence order* $\mathsf{bf}(H)$:
  $\psi <_{\mathsf{bf}(H)} \psi'$ iff $\psi <_H \psi'$, $\psi \in \{(\_, \_, \mathsf{committed}), (\_, \_, \mathsf{aborted})\}$ and $\psi' = (\_, \_, \mathsf{fend})$, i.e., the transaction ends before the fence does (Figure 4.6(b)).
- *read-dependency* relation $\mathsf{wr}_x(H)$ for $x \in \mathsf{Reg}$[1]:
  $\psi <_{\mathsf{wr}_x(H)} \psi'$ iff $\psi = (\_, \_, \mathsf{write}(x, v))$, $\psi' = (\_, \_, \mathsf{ret}(v))$ and the matching request action for ψ' is $(\_, \_, \mathsf{read}(x))$.
- *transactional read-dependency* relation $\mathsf{txwr}_x(H)$:
  $\psi <_{\mathsf{txwr}_x(H)} \psi'$ iff $\psi <_{\mathsf{wr}_x(H)} \psi'$, and ψ and ψ' are transactional.

Definition 4.6. *For a history H we let the* happens-before *relation of H be*

$$\mathsf{hb}(H) = (\mathsf{po}(H) \cup \mathsf{cl}(H) \cup \mathsf{af}(H) \cup \mathsf{bf}(H) \cup \bigcup_{x \in \mathsf{Reg}} (\mathsf{xpo}(H) \,;\, \mathsf{txwr}_x(H)))^+.$$

Components of the happens-before describe various forms of synchronization available in our programming language, which we now explain one by one. First, actions by the same thread cannot be concurrent, and thus, we let $\mathsf{po}(H) \subseteq \mathsf{hb}(H)$. To concentrate on issues related to TM, in this chapter we do not consider the integration of transactions into a language with a weak memory model and assume that the underlying non-transactional memory is sequentially consistent. Hence, we do not consider pairs of concurrent non-transactional accesses as races and let $\mathsf{cl}(H) \subseteq \mathsf{hb}(H)$. This can be used to privatize an object by agreeing on its status outside transactions, as illustrated in Figure 4.7. There the left-hand-side thread writes to `x` inside a transaction and then sets the flag `x_is_ready` outside. The right-hand-side thread keeps reading the flag non-transactionally until it is set, and then reads `x` non-transactionally. This program is DRF under $\mathcal{H}_{\mathsf{atomic}}$ because, in any of its traces, the conflicting write

[1]The notation wr, standing for "write-to-read", is chosen to mirror other kinds of dependencies introduced in §4.5.

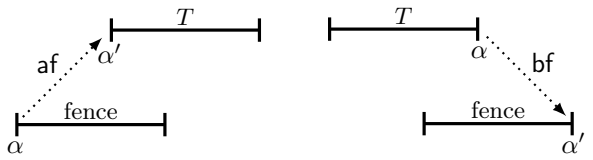


Figure 4.6: Illustration of the fence relations.

$\{$ `x_is_ready` $=$ `false` $\wedge$ `x` $= 0$ $\}$

```
l1 := atomic { // T          ||  do {
  x = 42;                   ||    l2 := x_is_ready; // ν′
}                           ||  } while (¬l2);
x_is_ready := true; // ν     ||  int l3 := x; // ν″
```

$\{$ `l1` $=$ `committed` $\implies$ `l3` $= 42$ $\}$

Figure 4.7: Privatization by agreement outside transactions.

in T and the non-transactional read ν'' are ordered in happens-before due to the client order between the write in ν and the read in ν' that causes the `do` loop to terminate.

We also have $\mathsf{xpo}(H) \,;\, \mathsf{txwr}_x(H) \subseteq \mathsf{hb}(H)$. Intuitively, this is because, if we have $(\alpha, \alpha') \in \mathsf{txwr}_x(H)$, then the commands by the thread of α preceding the transaction of α are guaranteed to have taken effect by the time α' executes.[2] This ensures that publication can be done safely, as we now illustrate by showing that the program in Figure 4.2 is DRF under $\mathcal{H}_{\mathsf{atomic}}$. Traces of the program may have only a single pair of conflicting actions—the accesses to `x` in ν and T_2. For both conflicting actions to occur, T_2 has to read `false` from `x_is_private`. Since under $\mathcal{H}_{\mathsf{atomic}}$ transactions do not interleave with other transactions or non-transactional accesses, for this T_1 has to execute before T_2, yielding a history of the form $\nu T_1 T_2$. In this history, we have a read-dependency between the write to `x_is_private` in T_1 and the read from `x_is_private` in T_2. But then the write to `x` in ν happens before the read from `x` in T_2, so that these actions cannot form a race.

Relations $\mathsf{af}(H)$ and $\mathsf{bf}(H)$ are used to formalize synchronization ensured by transactional fences. Recall that a fence blocks until all active transactions complete, by either committing or aborting. Hence, every transaction either begins after the fence does (and thus the fence does not need to wait for it; Figure 4.6(a)) or ends (including any required clean-up) before the fence does (Figure 4.6(b)). The relations $\mathsf{af}(H)$ and $\mathsf{bf}(H)$ capture the two respective cases. Note that, as required by Definition 4.1, every transaction has to be related to a fence at least by one of the two relations: a transaction may not span a fence.

Including after-fence and before-fence relations into happens-before ensures that privatization can be done safely given an appropriate placement of fences. To illustrate this, we show that the programs in Figure 4.1 are DRF under $\mathcal{H}_{\mathsf{atomic}}$ when we place a transactional fence between T_1 and ν. The possible

[2]Note, however, that the commands preceding α in its transaction may not have taken effect by this time: the TM may flush the writes of the transaction to the memory in any order. This is why we do not require $\mathsf{txwr}_x(H) \subseteq \mathsf{hb}(H)$.

conflicts are between the accesses to `x` in ν and T_2. For a conflict to occur, T_2 has to read `false` from `x_is_private`. Then T_2 has to execute before T_1, yielding a history H of the form $T_2 T_1 \psi_1 \psi_2 \nu$, where ψ_1 and ψ_2 denote the request and the response actions of the fence. Since T_2 occurs before ψ_2 in the history, they are related by the before-fence relation. But then the accesses to `x` in T_2 happen-before the write in ν and, therefore, the conflicting actions do not form a race. Finally, the program in Figure 4.3 is racy, since its traces contain pairs of conflicting actions unordered in happens-before. Inserting fences into this program will not make it DRF.

Our notion of DRF under $\mathcal{H}_{\mathsf{atomic}}$ establishes the condition that a program has to satisfy to be guaranteed strong atomicity. In the next section, we formulate the obligations of its TM counterpart in the DRF contract.

4.3 Strong Opacity

We state the requirements on a TM $\mathcal{H}$ by generalizing the notion of *opacity* [10, 68], yielding what we call *strong opacity*. As part of our definition, we require that a history H of a TM $\mathcal{H}$ can be matched by a history S of the atomic TM $\mathcal{H}_{\mathsf{atomic}}$ that "looks similar" to H from the perspective of the program. The similarity is formalized by the following relation $H \sqsubseteq S$, which requires S to be a permutation of H preserving the happens-before relation.

Definition 4.7. *A history H_1 is in the* strong opacity relation *with a history H_2, written $H_1 \sqsubseteq H_2$, if there is a bijection $\theta : \{1, \ldots, |H_1|\} \to \{1, \ldots, |H_2|\}$ such that:*

- $\forall i.\, H_1(i) = H_2(\theta(i))$, *and*
- $\forall i, j.\, i < j \wedge H_1(i) <_{\mathsf{hb}(H_1)} H_2(j) \implies \theta(i) < \theta(j)$.

The original definition of opacity requires any history of a TM $\mathcal{H}$ to have a matching history of the atomic TM $\mathcal{H}_{\mathsf{atomic}}$. However, such a requirement would be too strong for our setting: since the TM has no control over non-transactional actions of its clients, histories in $\mathcal{H}$ may be racy, and we do not want to require the TM to guarantee strong atomicity in such cases. Hence, our definition of strong opacity requires only DRF histories to have justifications in $\mathcal{H}_{\mathsf{atomic}}$. Let $\mathcal{H}|_{\mathsf{DRF}} = \{H \in \mathcal{H} \mid \mathsf{DRF}(H)\}$.

Definition 4.8. *A TM $\mathcal{H}$ is* strongly opaque, *written $\mathcal{H}|_{\mathsf{DRF}} \sqsubseteq \mathcal{H}_{\mathsf{atomic}}$, if*

$$\forall H.\, H \in \mathcal{H}|_{\mathsf{DRF}} \implies \exists S.\, S \in \mathcal{H}_{\mathsf{atomic}} \wedge H \sqsubseteq S.$$

Apart from the restriction to DRF histories, strong opacity and the usual opacity differ in several other ways. First, unlike in the usual opacity, our histories include non-transactional actions, because these can affect the behavior of the TM (e.g., via the idiom of "privatize, modify non-transactionally, publish", §4.1.2). Second, instead of preserving happens-before $\mathsf{hb}(H_1)$ in Definition 4.7, the usual opacity requires preserving the program order $\mathsf{po}(H_1)$ and the following *real-time order* $\mathsf{rt}(H_1)$ on actions:

$$\alpha <_{\mathsf{rt}(H)} \alpha \triangleq \psi \in \{(\_, \_, \mathsf{committed}), (\_, \_, \mathsf{aborted})\} \wedge \psi' = (\_, \_, \mathsf{txbegin}) \wedge \psi <_H \psi'.$$

This orders non-overlapping transactions, with the duration of a transaction determined 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). As shown in [8], preserving real-time order is unnecessary if program threads do not have means of communication not reflected in histories. Since we record the actions using both transactional and non-transactional accesses, preserving real-time order is unnecessary for our results. However, we use this order to prove strong opacity by adjusting the proofs of the usual one (§4.5). Finally, preserving happens-before in Definition 4.7 is required so that we could check DRF assuming strong atomicity, as we explain next.

4.4 The Fundamental Property

We now formalize the Fundamental Property of our DRF notion using *observational refinement* [51]: if a program is DRF under the atomic TM $\mathcal{H}_{\mathsf{atomic}}$, then any trace of the program under a strongly opaque TM $\mathcal{H}$ has an *observationally equivalent* trace under the atomic TM $\mathcal{H}_{\mathsf{atomic}}$.

Definition 4.9. *Traces τ and τ' are* observationally equivalent*, denoted by $\tau \sim \tau'$, if:*

$$(\forall t \in \mathsf{ThreadID}.\, \tau|_t = \tau'|_t) \wedge (\tau|_{\mathsf{nontx}} = \tau'|_{\mathsf{nontx}}),$$

where $\tau|_{\mathsf{nontx}}$ denotes the subsequence of τ containing all actions from non-transactional accesses.

Equivalent traces are considered indistinguishable to the user. In particular, the sequences of non-transactional accesses in equivalent traces (which usually include all input-output) satisfy the same linear-time temporal properties. We lift the equivalence to sets of traces as follows.

Definition 4.10. *A set of traces $\mathcal{T}$* observationally refines *a set of traces $\mathcal{T}'$, written $\mathcal{T} \preceq \mathcal{T}'$, if $\forall \tau \in \mathcal{T}.\, \exists \tau' \in \mathcal{T}'.\, \tau \sim \tau'$.*

Theorem 4.11 (Fundamental Property)**.** *If $\mathcal{H}$ is a TM such that $\mathcal{H}|_{\mathsf{DRF}} \sqsubseteq \mathcal{H}_{\mathsf{atomic}}$, then:*

$$\forall P, s.\, \mathsf{DRF}(P, s, \mathcal{H}_{\mathsf{atomic}}) \Longrightarrow [\![P]\!](\mathcal{H}, s) \preceq [\![P]\!](\mathcal{H}_{\mathsf{atomic}}, s).$$

Theorem 4.11 establishes a contract between the programmer and the TM implementors. The TM implementor has to ensure strong opacity of the TM assuming the program is DRF: $\mathcal{H}|_{\mathsf{DRF}} \sqsubseteq \mathcal{H}_{\mathsf{atomic}}$. The programmer has to ensure the DRF of the program assuming strong atomicity: $\mathsf{DRF}(P, s, \mathcal{H}_{\mathsf{atomic}})$. This contract lets the programmer to check properties of a program assuming strong atomicity ($[\![P]\!](\mathcal{H}_{\mathsf{atomic}}, s)$) and get the guarantee that the properties hold when the program uses the actual TM implementation ($[\![P]\!](\mathcal{H}, s)$). We have already shown that the expected privatization and publication idioms are DRF under strong atomicity (§4.2), so that the programmer can satisfy its part of the contract. In the following sections we develop a method for discharging the obligations of the TM.

The proof of Theorem 4.11 follows directly from the following lemma:

Lemma 4.12. *If $\mathcal{H}$ is a TM such that $\mathcal{H}|_{\mathsf{DRF}} \sqsubseteq \mathcal{H}_{\mathsf{atomic}}$, then:*

1. $\forall P, s.\, \mathsf{DRF}(P, s, \mathcal{H}) \implies [\![P]\!](\mathcal{H}, s) \preceq [\![P]\!](\mathcal{H}_{\mathsf{atomic}}, s)$.

2. $\forall P, s.\, \mathsf{DRF}(P, s, \mathcal{H}_{\mathsf{atomic}}) \implies \mathsf{DRF}(P, s, \mathcal{H})$.

Part 1 shows that if a program is DRF under the concrete TM $\mathcal{H}$, then it has the expected strongly atomic semantics. It is an adaptation of a result from [51]. Part 2 enables checking DRF using an atomic TM $\mathcal{H}_{\mathsf{atomic}}$ and is a contribution of this dissertation. Its proof relies on the fact that strong opacity preserves happens-before (Definition 4.7).

Proof of Lemma 4.12. The key step in the proof of Lemma 4.12(1) is the next lemma. 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 4.13 (Rearrangement)**.**

$$\forall H, S \in \mathsf{History}.\, H \sqsubseteq S \implies (\forall \tau_H.\, \mathsf{history}(\tau_H) = H \implies \exists \tau_S.\, \mathsf{history}(\tau_S) = S \wedge \tau_H \sim \tau_S).$$

The proof follows the proof of the corresponding lemma in [51] and is omitted.

We also rely on the following proposition that allows us to conclude that the trace τ_S resulting from the rearrangement in Lemma 4.13 can be produced by a program P if so can the original trace τ_H.

Proposition 4.14. *If $\tau_H \in [\![P]\!](s)$ and $\tau_H \sim \tau_S$, then $\tau_S \in [\![P]\!](s)$.*

This proposition follows immediately from Definition 4.9 and the definition of $A(P)$ in Figure 4.5.

Lemma 4.13 and Proposition 4.14 together yield the following corollary, from which Lemma 4.12(1) follows.

Corollary 4.15. *If $\mathcal{H}$ is a TM such that $\mathcal{H}|_{\mathsf{DRF}} \sqsubseteq \mathcal{H}_{\mathsf{atomic}}$, then:*

$$\forall P, s, \tau_H.\, \tau_H \in [\![P]\!](\mathcal{H}, s) \wedge \mathsf{DRF}(\tau_H) \implies \exists \tau_S.\, \tau_S \in [\![P]\!](\mathcal{H}_{\mathsf{atomic}}, s) \wedge \mathsf{history}(\tau_H) \sqsubseteq \mathsf{history}(\tau_S) \wedge \tau_H \sim \tau_S.$$

The following proposition states the prefix-closure property of the programming language semantics.

Proposition 4.16. *For every program P, TM $\mathcal{H}$ and state s, the set $[\![P]\!](\mathcal{H}, s)$ is prefix-closed.*

Proof of Lemma 4.12(2). Let us consider a TM $\mathcal{H}$ such that $\mathcal{H}|_{\mathsf{DRF}} \sqsubseteq \mathcal{H}_{\mathsf{atomic}}$, any client program P and an initial state s. We prove the theorem by contrapositive, i.e., by demonstrating:

$$\neg\mathsf{DRF}(P, s, \mathcal{H}) \implies \neg\mathsf{DRF}(P, s, \mathcal{H}_{\mathsf{atomic}})$$

Let us assume that $\mathsf{DRF}(P, s, \mathcal{H})$ does not hold. By Definition 4.5, there exists $\hat{\tau} \in [\![P]\!](\mathcal{H}, s)$ such that its history $\hat{H} = \mathsf{history}(\hat{\tau})$ is racy, i.e., $\mathsf{DRF}(\hat{H})$ does not hold. More specifically, $\hat{\tau}$ contains a data race (ψ', ψ). Note that there might be multiple races in $\hat{\tau}$, and we choose such race (ψ', ψ) that the later of the two

actions, ψ, is the earliest in the execution order of $\hat{H}$. We split the further proof depending on whether ψ is transactional or non-transactional.

Case 1: ψ is non-transactional. Let $\hat{\tau}$ take form of $\hat{\tau} = \tau\,\psi\,\beta\,\_$, where β is a matching response action for ψ and τ is a prefix of $\hat{\tau}$ containing ψ'. It is easy to see that our choice of the data race is such that $\tau\,\psi\,\beta$ is racy and τ is data-race free. Additionally, as stated in Proposition 4.16, $[\![P]\!](\mathcal{H}, s)$ is prefix-closed. Therefore, both $\tau\,\psi\,\beta \in [\![P]\!](\mathcal{H}, s)$ and $\tau \in [\![P]\!](\mathcal{H}, s)$ hold. By Corollary 4.15, there exists a trace $\tau' \in [\![P]\!](\mathcal{H}_{\mathsf{atomic}}, s)$ such that its history $S = \mathsf{history}(\tau') \in \mathcal{H}_{\mathsf{atomic}}$ is in the strong opacity relation with $H = \mathsf{history}(\tau)$ ($H \sqsubseteq S$) and $\tau \sim \tau'$.

Let us consider a trace $\tau'' = \tau'\,\psi\,\beta'$, which extends τ' with the request action ψ and a matching response β' in such a way that its history $S'' = S\,\psi\,\beta' \in \mathcal{H}_{\mathsf{atomic}}$. Such response β' and a trace τ'' exist, since $S\,\psi$ is a non-interleaved history, and it is easy to see that it can always be extended with a response to ψ (not necessarily returning the same result as β).

Since $\tau \sim \tau'$, we have $\tau\,\psi\,\beta \sim \tau'\,\psi\,\beta$ according to Definition 4.9. Therefore, by Proposition 4.14, $\tau'\,\psi\,\beta \in [\![P]\!](s)$. Also, as evident from the definition of $A(P)$ in Figure 4.5, $A(P)$ is closed under replacing a return value of a trailing read-response action, so $\tau'' \in [\![P]\!](s)$ holds. Knowing that $S'' \in \mathcal{H}_{\mathsf{atomic}}$, we conclude that $\tau'' \in [\![P]\!](\mathcal{H}_{\mathsf{atomic}}, s)$.

We argue that $\mathsf{hb}(H\,\psi\,\beta) = \mathsf{hb}(S'')$. First, since $\tau\,\psi\,\beta \sim \tau'\,\psi\,\beta$ and τ'' differs from $\tau'\,\psi\,\beta$ only in the return value of β', $\mathsf{cl}(H\,\psi\,\beta) = \mathsf{cl}(S'')$ and $\mathsf{po}(H\,\psi) = \mathsf{po}(S'')$ hold. Since ψ is neither a fence action nor a beginning or an end of a transaction, $\mathsf{af}(H\,\psi\,\beta) = \mathsf{af}(S'')$ and $\mathsf{bf}(H\,\psi\,\beta) = \mathsf{bf}(S'')$ hold too. Finally, since ψ is non-transactional, it does not contribute to $(\mathsf{xpo}(S'')\,;\,\mathsf{txwr}(S''))$. Overall, $\mathsf{hb}(H\,\psi\,\beta) = \mathsf{hb}(S'')$ holds.

Knowing that $\mathsf{hb}(\mathsf{history}(\tau\,\psi\,\beta)) = \mathsf{hb}(H\,\psi\,\beta) = \mathsf{hb}(S'')$ and that the conflict (ψ', ψ) is unordered in $\mathsf{hb}(\mathsf{history}(\tau\,\psi\,\beta))$, we conclude that (ψ', ψ) is a data race in τ''.

Case 2: ψ is transactional. Let $\hat{\tau}$ take form of $\hat{\tau} = \tau\,\psi\,\_$, where the prefix τ contains ψ'. It is easy to see that our choice of the data race is such that $\tau\,\psi$ is racy and τ is data-race free. Additionally, as stated in Proposition 4.16, $[\![P]\!](\mathcal{H}, s)$ is prefix-closed. Therefore, both $\tau\,\psi \in [\![P]\!](\mathcal{H}, s)$ and $\tau \in [\![P]\!](\mathcal{H}, s)$ hold. By Corollary 4.15, there exists a trace $\tau' \in [\![P]\!](\mathcal{H}_{\mathsf{atomic}}, s)$ such that its history $S = \mathsf{history}(\tau') \in \mathcal{H}_{\mathsf{atomic}}$ is in the strong opacity relation with $H = \mathsf{history}(\tau)$ ($H \sqsubseteq S$) and $\tau \sim \tau'$.

Let us consider a trace τ'' obtained from τ' by inserting ψ after the last action by the same thread. It is easy to see that τ'' is well-formed and that $S' = \mathsf{history}(\tau'')$ is non-interleaved, meaning that $S' \in \mathcal{H}_{\mathsf{atomic}}$ holds.

Since $\tau \sim \tau'$, the following holds:

$$(\forall t \in \mathsf{ThreadID}.\, \tau|_t = \tau'|_t) \wedge (\tau|_{\mathsf{nontx}} = \tau'|_{\mathsf{nontx}}).$$

Note that the following holds, because ψ is transactional:

$$\tau''|_{\mathsf{nontx}} = \tau'|_{\mathsf{nontx}} = \tau|_{\mathsf{nontx}} = (\tau\,\psi)|_{\mathsf{nontx}}.$$

Let t' be the thread of ψ. For all $t \in \mathsf{ThreadID} \setminus \{t'\}$, the following holds, because ψ is by a different thread:

$$(\tau\,\psi)|_t = \tau|_t = \tau'|_t = \tau''|_t,$$

and $(\tau\psi)|_{t'} = \tau''|_{t'}$ by construction. Overall, we have:

$$(\forall t \in \mathsf{ThreadID}.\, (\tau\psi)|_t = \tau''|_t) \wedge ((\tau\psi)|_{\mathsf{nontx}} = \tau''|_{\mathsf{nontx}}),$$

so $\tau\psi \sim \tau''$. Therefore, by Proposition 4.14, $\tau'' \in [\![P]\!](s)$. Knowing that $S' \in \mathcal{H}_{\mathsf{atomic}}$, we conclude that $\tau'' \in [\![P]\!](\mathcal{H}_{\mathsf{atomic}}, s)$.

We argue that $\mathsf{hb}(H\psi) = \mathsf{hb}(S')$. Similarly to Case 1, we observe that po, cl, af and bf components of $\mathsf{hb}(H\psi)$ and $\mathsf{hb}(S')$ are equal. Let us consider the edges from $(\mathsf{xpo}(H\psi)\,;\,\mathsf{txwr}(H\psi))$ that may appear in $\mathsf{hb}(H\psi) \setminus \mathsf{hb}(H)$:

- if ψ is a write request, then $\{(\beta, \beta') \mid \beta <_{\mathsf{xpo}(H\psi)} \psi <_{\mathsf{txwr}(H\psi)} \beta'\}$;
- if ψ is a read request, then $\{(\beta, \psi) \mid \exists \beta'.\, \beta <_{\mathsf{xpo}(H\psi)} \beta' <_{\mathsf{txwr}(H\psi)} \psi\}$.

Note that $\mathsf{txwr}(H\psi) = \mathsf{txwr}(S')$, because $H\psi$ and S' are histories with unique writes, so read-dependencies are uniquely determined. Also, $\mathsf{xpo}(H\psi) = \mathsf{xpo}(S')$, because ψ is executed within the same transaction in the two histories. Therefore, the aforementioned edges are present in $\mathsf{hb}(H\psi)$ if and only if they are present in $\mathsf{hb}(S')$. Overall, $\mathsf{hb}(H\psi) = \mathsf{hb}(S')$ holds.

Knowing that $\mathsf{hb}(\mathsf{history}(\tau\psi)) = \mathsf{hb}(S')$ and that the conflict (ψ', ψ) is unordered in $\mathsf{hb}(\mathsf{history}(\tau\psi))$, we conclude that (ψ', ψ) is a data race in $\tau'\psi$. □

4.5 Proving Strong Opacity

We now develop a method for proving $\mathcal{H}|_{\mathsf{DRF}} \sqsubseteq \mathcal{H}_{\mathsf{atomic}}$. The method builds on a *graph characterization* of opacity of Guerraoui and Kapalka [68], which was proposed for proving the usual opacity of TMs that do not allow mixed transactional/non-transactional accesses to the same data. The characterization allows checking opacity of a history by checking two properties: *consistency* of the history and the acyclicity of a certain *opacity graph*, which we define further in this section.

Consistency captures some very basic properties of read-dependency relation wr_x (for each register x) that have to be satisfied by every opaque TM history. Intuitively, in a consistent history every transaction T reading the value of a register x either reads the latest value T itself wrote to x before, or some value written non-transactionally or by a committed or commit-pending transaction.

Definition 4.17. *A pair of matching request and response actions (ψ, ψ') is said to be local to $T \in \mathsf{txns}(H)$, if:*

- $\psi = (\_, \_, \mathsf{read}(x)) \wedge$
  $\exists \beta \in T.\, \beta <_{\mathsf{po}(H)} \psi \wedge \beta = (\_, \_, \mathsf{write}(x, \_))$*; or*
- $\psi = (\_, \_, \mathsf{write}(x, \_)) \wedge$
  $\exists \beta \in T.\, \psi <_{\mathsf{po}(H)} \beta \wedge \beta = (\_, \_, \mathsf{write}(x, \_))$.

We let $\mathsf{local}(H)$ denote the set of all local actions in H.

Thus, local reads from x are preceded by a write to x in the same transaction; local writes to x are followed by a write to x in the same transaction.

Definition 4.18. *In a history H, a read request $\psi = (\_, \_, \mathsf{read}(x))$ and its matching response $\psi' = (\_, \_, \mathsf{ret}(v))$ are said to be consistent, if:*

- *when* $(\psi, \psi') \in \mathsf{local}(H)$ *and performed by a transaction* T*,* v *is the value written by the most recent write* $(\_, \_, \mathsf{write}(x, v))$ *preceding the read in* T*;*
- *when* $(\psi, \psi') \notin \mathsf{local}(H)$*, either there exists a non-local* β *not located in an aborted or live transaction such that* $\beta <_{\mathsf{wr}_x(H)} \psi'$*, or there is no such* β *and* $v = v_{\mathsf{init}}$*.*

We also say that a history H *is consistent, written* $\mathsf{cons}(H)$*, if all of its matching read requests and responses are.*

We now present the definition of an opacity graph of a history with mixed transactional/non-transactional accesses.

Definition 4.19. *The* opacity graph *of a history* H *is a tuple* $G = (N, \mathsf{vis}, \mathsf{HB}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$*, where:*

- $N = \mathsf{txns}(H) \cup \mathsf{nontxn}(H)$ *is the set of nodes.*
- $\mathsf{vis} \subseteq N$ *is a* visibility *predicate, such that it holds of all non-transactional accesses and committed transactions and does not hold of all aborted and live transactions.*
- $\mathsf{HB} \in \mathcal{P}(N \times N)$ *is such that*

$$n \xrightarrow{\mathsf{HB}} n' \iff \exists \alpha \in n, \alpha' \in n'.\, \psi <_{\mathsf{hb}(H)} \psi'.$$

- $\mathsf{WR} \in \mathsf{Reg} \to \mathcal{P}(N \times N)$ *specifies* read-dependency *relations on nodes: for each* $x \in \mathsf{Reg}$*,*

$$n \xrightarrow{\mathsf{WR}_x} n' \iff n \neq n' \wedge \exists \psi \in n, \psi' \in n'.\, \psi <_{\mathsf{wr}_x(H)} \psi',$$

  where the relation on actions $\mathsf{wr}_x(H)$ *is defined in §4.2. We require that each node that is read from be visible:*

$$\forall n, x.\, n \xrightarrow{\mathsf{WR}_x} \_ \implies \mathsf{vis}(n).$$

- $\mathsf{WW} \in \mathsf{Reg} \to \mathcal{P}(N \times N)$ *specifies* write-dependency *relations, such that for each* $x \in \mathsf{Reg}$*,* WW_x *is an irreflexive total order on* $\{n \in N \mid \mathsf{vis}(n) \wedge (\_, \_, \mathsf{write}(x, \_)) \in n\}$*.*
- $\mathsf{RW} \in \mathsf{Reg} \to \mathcal{P}(N \times N)$ *specifies* anti-dependency *relations, computed from* WR *and* WW *as follows:*

$$\begin{aligned} n \xrightarrow{\mathsf{RW}_x} n' \iff n \neq n' \wedge ((\exists n''.\, n'' \xrightarrow{\mathsf{WW}_x} n' \wedge n'' \xrightarrow{\mathsf{WR}_x} n) \\ \vee (\mathsf{vis}(n') \wedge (\_, \_, \mathsf{write}(x, \_)) \in n' \\ \wedge (\_, \_, \mathsf{ret}(x, v_{\mathsf{init}})) \in n)). \end{aligned}$$

We let $\mathsf{Graph}(H)$ denote the set of all opacity graphs of H. We say that a graph G is *acyclic*, written $\mathsf{acyclic}(G)$, if edges from HB, WR, WW and RW do not form a cycle.

The nodes in our opacity graph include transactions and non-transactional accesses in H. The intention of the vis predicate is to mark those nodes that

have taken effect, in particular, commit-pending transactions that should be considered committed (cf. history completions in §4.1.4). The other components, intuitively, constrain the order in which the nodes should go in a sequential history witnessing the strong opacity of H. The HB relation is the lifting of happens-before to the nodes of the graph. A read-dependency $n \xrightarrow{\mathsf{WR}_x} n'$ specifies when the node n' reads a value of x written by another node n. A write-dependency $n \xrightarrow{\mathsf{WW}_x} n'$ specifies when n' overwrites a value of x written by n; for the writes to take effect, both nodes should be visible. Finally, an anti-dependency $n \xrightarrow{\mathsf{RW}_x} n'$ specifies when n reads a value of x overwritten by n'; the initial value v_{init} of x is considered overwritten by any write to the register.

The following lemma (proved in §4.5 shows that we can check strong opacity of a history by checking its consistency and the acyclicity of its opacity graph. Then the theorem following from it gives a criterion for the strong opacity of a TM $\mathcal{H}$.

Lemma 4.20. $\forall H.\, (\mathsf{cons}(H) \wedge \exists G \in \mathsf{Graph}(H).\, \mathsf{acyclic}(\mathsf{G})) \implies (\exists H' \in \mathcal{H}_{\mathsf{atomic}}.\, H \sqsubseteq H')$.

Theorem 4.21. $\mathcal{H} \sqsubseteq \mathcal{H}_{\mathsf{atomic}}$ *holds, if the following is true:*

$$\forall H \in \mathcal{H}.\, \mathsf{cons}(H) \wedge \exists G \in \mathsf{Graph}(H).\, \mathsf{acyclic}(\mathsf{G}).$$

In comparison to the graph characterization of the usual opacity [68] for TMs without mixed transactional/non-transactional accesses, ours is more complex: the graph includes non-transactional accesses and the acyclicity check has to take into account the happens-before relation. We now show that, to prove the strong opacity of a TM using Theorem 4.21, we need to make only a minimal adjustment to a proof of its usual opacity using graph characterization. The latter characterization includes only transactions as nodes of the graph, and instead of happens-before, considers the lifting of the real-time order from §4.3 to transactions: for a history H, we let $\mathsf{RT}(H)$ be the relation between transactions in H such that $T <_{\mathsf{RT}(H)} T'$ iff for some $\alpha \in T$ and $\alpha' \in T$ we have $\alpha <_{\mathsf{rt}(H)} \alpha$.

In the following we abuse notation and denote by WR also the relation $\bigcup_{x \in \mathsf{Reg}} \mathsf{WR}_x$, and similarly for WW and RW.

Theorem 4.22. *Let a history* $H \in \mathcal{H}_{\mathbb{C}}|_{\mathsf{DRF}}$ *and an opacity graph* $G = (N, \mathsf{vis}, \mathsf{HB}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}) \in \mathsf{Graph}(H)$ *be such that the relation* $(\mathsf{HB}\ ;\ (\mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}))$ *is irreflexive. If* G *contains a cycle, then it also contains a cycle over transactions only with edges from* $\mathsf{RT} \cup \mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}$.

Thus, the theorem allows us to modularize the proof of the acyclicity of an opacity graph into: (i) checking the absence of "small" cycles with a single dependency edge; and (ii) checking the absence of cycles in the projection of the graph to transactions, with real-time order replacing happens-before. The latter acyclicity check is exactly the one required in the graph characterization of the usual opacity [68]. In the next section, we show how the theorem enables a simple proof of strong opacity of a realistic TM, TL2 [22].

Proof of Lemma 4.20

The key idea of the proof of Lemma 4.20 is

To prove Lemma 4.20, we introduce a notion of *fenced* opacity graphs, which extend opacity graphs with fence actions. We prove that fenced opacity graphs are acyclic when corresponding opacity graphs are.

Let $\mathsf{fact}(H)$ denote the set of all fence actions of a history H, or formally:

$$\mathsf{fact}(H) \triangleq \{(a,t,\mathsf{fbegin}) \mid (a,t,\mathsf{fbegin}) \in \mathsf{act}(H)\} \\ \cup \{(a,t,\mathsf{fbegin}) \mid (a,t,\mathsf{fend}) \in \mathsf{act}(H)\}.$$

Definition 4.23. *Given a history H and its opacity graph $G = (N, \mathsf{vis}, \mathsf{HB}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}) \in \mathsf{Graph}(H)$, we define a matching fenced graph $\overline{G}$ as a tuple $(\overline{N}, \mathsf{vis}, \overline{\mathsf{HB}}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$, where:*

- *$\overline{N} = N \cup \mathsf{fact}(H)$ extends the set of nodes of G with nodes denoting fence actions of the history H;*
- *$\overline{\mathsf{HB}} \in \mathcal{P}(\overline{N} \times \overline{N})$ is such that*

$$n \xrightarrow{\overline{\mathsf{HB}}} n' \iff \exists \psi, \psi'.\, (\psi \in n \vee \psi = n \in \mathsf{fact}(H)) \\ \wedge (\psi' \in n' \vee \psi' = n' \in \mathsf{fact}(H)) \wedge \psi <_{\mathsf{hb}(H)} \psi'.$$

That is, a fenced graph $\overline{G}$ corresponding to $G \in \mathsf{Graph}(H)$ simply extends G with fence actions and happens-before edges.

Proposition 4.24. $\forall H, G.\, G \in \mathsf{Graph}(H) \wedge \mathsf{acyclic}(G) \implies \mathsf{acyclic}(\overline{G})$.

Proof. Let $G = (N, \mathsf{vis}, \mathsf{HB}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$ and $\overline{G} = (\overline{N}, \mathsf{vis}, \overline{\mathsf{HB}}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$. It is easy to see that:

$$\overline{\mathsf{HB}} = \mathsf{HB} \uplus \{(\psi, \psi') \mid \psi <_{\mathsf{hb}(H)} \psi' \wedge (\psi \in \mathsf{fact}(H) \vee \psi' \in \mathsf{fact}(H))\}$$

That is, all edges between non-fence nodes implied by transitivity via fence nodes in $\overline{G}$ are already present in G. We know that G is acyclic. Therefore, if $\overline{G}$ contains a cycle, it necessarily involves fence actions.

Let us first consider a cycle over nodes of $\overline{G}$ consisting entirely of fence action nodes. The cycle takes the following form then:

$$\phi_1 \xrightarrow{\mathsf{HB}} \phi_2 \xrightarrow{\mathsf{HB}} \ldots \xrightarrow{\mathsf{HB}} \phi_k \xrightarrow{\mathsf{HB}} \phi_1,$$

where k is the length of the cycle and ϕ_i is a fence action for each i such that $1 \leq i \leq k$. For each consecutive ϕ_i and ϕ_{i+1} in the cycle, we assume that there is no non-fence action n such that $\phi_i \xrightarrow{\mathsf{HB}} n \xrightarrow{\mathsf{HB}} \phi_{i+1}$ (we consider cycles involving non-fence actions separately). By Definition 4.19, the following must be the case:

$$\phi_1 <_{\mathsf{hb}(H)} \phi_2 <_{\mathsf{hb}(H)} \cdots <_{\mathsf{hb}(H)} \phi_k <_{\mathsf{hb}(H)} \phi_1,$$

It is easy to see that such cycle is not possible: without intermediate non-fence actions, fence actions can only be related by $\mathsf{po}(H)$ in $\mathsf{hb}(H)$, and $\mathsf{po}(H)$ is not cyclic.

We now consider a cycle in $\overline{G}$ involving at least one node that does not correspond to a fence action. We consider each sequence of fence actions in the cycle surrounded by non-fence actions:

$$n \xrightarrow{\mathsf{HB}} \phi_1 \xrightarrow{\mathsf{HB}} \phi_2 \xrightarrow{\mathsf{HB}} \ldots \xrightarrow{\mathsf{HB}} \phi_k \xrightarrow{\mathsf{HB}} n'$$

where n and n' denote non-fence actions (or, possibly, the same non-fence action), and for each i ($1 \leq i \leq k$), ϕ_i is a fence action. Since HB is transitive, $n \xrightarrow{\mathsf{HB}} n'$ is present in the graph G. By replacing each segment of fence actions in the cycle with HB-edges between non-fence actions, we transform the cycle in $\overline{G}$ into a cycle in G. By the premise of the proposition, $\mathsf{acyclic}(G)$ holds. Thus, we arrive to a contradiction, meaning that $\mathsf{acyclic}(\overline{G})$ holds. □

Before proving Lemma 4.20, let us restate the definition of $\mathcal{H}_{\mathsf{atomic}}$ from §4.1.4.

Definition 4.25. *In a non-interleaved history H, a read request $(\_, \_, \mathsf{read}(x))$ and its matching response $(\_, \_, \mathsf{ret}(v))$ are said to be legal, if v is the value returned by the last preceding $\mathsf{write}(x, v)$ action that is not located in an aborted or live transaction different from the one of the read; if there is no such write, then $v = v_{\mathsf{init}}$.*

Thus, $\mathcal{H}_{\mathsf{atomic}}$ can be defined as the set of all non-interleaved histories H that have a completion H^c, in which every pair of a matching read request and response is legal.

Proof of Lemma 4.20. In the proof, we only consider finite histories of TM. Let H_1 be an consistent TM history. Assume that there exists an acyclic graph $G \in \mathsf{Graph}(H_1)$. We consider its matching fenced graph $\overline{G} = (N, \mathsf{vis}, \mathsf{HB}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$, which is acyclic by Proposition 4.24.

Let k be the number of nodes in N and a sequence $S = n_1 n_2 \ldots n_k$ be the result of a topological sort of graph $\overline{G}$. Let $\prec_S$ be a relation on nodes of S such that $n \prec_S n'$ if n occurs earlier than n' in S.

We define a relation $\mathsf{lin} \subseteq \mathsf{act}(H_1) \times \mathsf{act}(H_1)$ on actions of history H_1 so that for every actions $\psi <_{\mathsf{lin}} \psi'$ if for $n, n' \in S$ such that $\psi \in n$ and $\psi' \in n'$, either $n = n' \wedge \psi <_{\mathsf{po}(H_1)} \psi'$ or $n \neq n' \wedge n \prec_S n'$ holds. It is easy to see that lin is a linear order on $\mathsf{act}(H_1)$.

Let H_2 be a sequence obtained by putting the actions of the history H_2 in the order lin. Note that $\mathsf{hb}(H) \subseteq \mathsf{lin}$. Therefore, by Definition 4.7, $H_1 \sqsubseteq H_2$ holds. It is easy to see that H_2 is non-interleaved. To conclude that $H_2 \in \mathcal{H}_{\mathsf{atomic}}$, it remains to show that it has a non-interleaved completion, in which every matching read request and response are legal.

Let H_2^c be a non-interleaved completion of H_2 obtained by committing each transaction from vis and aborting all other commit-pending transaction. We argue that $\mathsf{cons}(H_2^c)$ holds. It is easy to see that local read actions are consistent in H_2^c. Consider any non-local read (ψ, ψ'). Since $\mathsf{cons}(H_1)$ holds, there are two possibilities:

- There exists a non-local β such that $\beta <_{\mathsf{wr}\_} \psi'$ and β is not located in an aborted or live transaction in H_1. Let $n \in N$ be the node of $\hat{G}$ containing β. By Definition 4.19, $n \in \mathsf{vis}$ holds. Therefore, n is not an aborted or live transaction in H_2^c either, meaning that (ψ, ψ') is consistent in H_2^c.
- There is no such write β, and ψ' returns v_{init}. It is easy to see that (ψ, ψ') is consistent in H_2^c.

We consider every read request $\psi = (\_, \_, \mathsf{read}(x))$ and its matching response $\psi' = (\_, \_, \mathsf{ret}(v))$ in H_2^c. When $(\psi, \psi') \in \mathsf{local}(H_2^c)$, the consistency of H_2^c immediately implies that (ψ, ψ') is legal.

Let us assume that $(\psi, \psi') \notin \mathsf{local}(H_2^c)$, and let n be its node in the graph $\hat{G}$. By Definition 4.18, there are two possibilities:

- there exists a non-local β_1 not located in an aborted or live transaction and such that $\beta_1 <_{\mathsf{wr}_x(H_2^c)} \psi'$; or
- $v = v_{\mathsf{init}}$, otherwise.

Let us assume that there is no β_1 satisfying the above and that $v = v_{\mathsf{init}}$. Consider any action $\beta_2 = (\_, \_, \mathsf{write}(x, \_))$ in H_2^c that is not located in a live or aborted transaction, and let n_2 denote its node in $\hat{G}$. Knowing that n_2 is not an aborted transaction in H_2^c, we conclude that $n \in \mathsf{vis}$ holds. By Definition 4.19, $n \xrightarrow{\mathsf{RW}_x} n_2$ holds. Since lin is a topological sort of the order including RW_x, all actions of the node n_2 occur after all actions of the node n, meaning that β_2 does not precede ψ in H_2^c. Therefore, It is easy to see that (ψ, ψ') is legal.

We now consider the case when in H_2^c there exists a non-local β_1 not located in an aborted or live transaction and such that $\beta_1 <_{\mathsf{wr}_x(H_2^c)} \psi'$. It is easy to see that in order to conclude that (ψ, ψ') is legal, we only need to prove that β_1 is the most recent write to x preceding ψ. Let n and n_1 be the nodes of ψ and β_1 accordingly in the graph $\overline{G}$. Firstly, we observe that $\beta_1 <_{\mathsf{wr}_x(H_1)} \psi$ holds and, hence, so does $n_1 \xrightarrow{\mathsf{WR}} n$. Since lin is a topological sort of the order including WR, actions of n_1 precede actions of n in H_2^c. Therefore, β_1 precedes ψ in H_2^c. Let us consider any other write $\beta_2 = (\_, \_, \mathsf{write}(x, \_))$ that occurs after β_1 in H_2^c and that is not located in a live or aborted transaction. Let n_2 be its node in the graph $\overline{G}$. Knowing that n_2 is not an aborted transaction in H_2^c, we conclude that $n \in \mathsf{vis}$ holds. By Definition 4.19, WW totally orders visible nodes writing to the same register, so either $n_2 \xrightarrow{\mathsf{WW}} n_1$ or $n_1 \xrightarrow{\mathsf{WW}} n_2$ holds. Recall that lin is a topological sort of the order including WW and RW. Consequently, it can only be the case that $n_1 \xrightarrow{\mathsf{WW}} n_2$ holds. By Definition 4.19, $n \xrightarrow{\mathsf{RW}} n_2$ holds too, and, hence, $\psi <_{\mathsf{lin}} \beta_2$. Thus, we have shown that β_1 is the last write to x that precedes ψ and that is not located in an aborted or live transaction, meaning that (ψ, ψ') is legal. □

Proof of Theorem 4.22

We now prove Theorem 4.22. The main idea of the proof lies in the observation that any edge in $\mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}$, where one of the endpoints is a transaction and one is a non-transactional access, yields a pair of conflicting actions in H. Since H is DRF, this means that the nodes are related by HB one way or another, and the irreflexivity of $(\mathsf{HB} \; ; (\mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}))$ means that the dependency edge has to be covered by HB. Using this, we can transform any cycle in the graph into one in $\mathsf{RT} \cup \mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}$ by replacing segments of edges involving non-transactional accesses by the real-time order.

Lemma 4.26. *Let a DRF history H and its graph $G = (N, \_, \mathsf{HB}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}) \in \mathsf{Graph}(H)$ be such that $(\mathsf{HB} \; ; (\mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}))$ is*

irreflexive. Then for all nodes $n, n' \in N$, such that at most one of them is a transaction:

$$n \xrightarrow{\mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}} n' \implies n \xrightarrow{\mathsf{HB}} n'.$$

Proof. Let us first consider two nodes $n = \nu$ and $n' = \nu'$ that are non-transactional accesses such that $\nu \xrightarrow{\mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}} \nu'$. All non-transactional actions are totally ordered by $\mathsf{cl}(H)$, and, therefore, by $\mathsf{hb}(H)$. Consequently, at least one of the two edges, $\nu \xrightarrow{\mathsf{HB}} \nu'$ and $\nu' \xrightarrow{\mathsf{HB}} \nu$, is present in the graph G. Knowing that $(\mathsf{HB} \,;\, (\mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}))$ is irreflexive, we conclude that only the former edge is possible, i.e., $\nu \xrightarrow{\mathsf{HB}} \nu'$ holds.

Let us now consider a node-transaction $n = T$ and a non-transactional access $n' = \nu$ such that $T \xrightarrow{\mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}} \nu$ (the case when n is a non-transactional access and n' is a transaction is analogous). We first assume that T and ν are by the same thread. Then either $T \xrightarrow{\mathsf{HB}} \nu$ or $\nu \xrightarrow{\mathsf{HB}} T$ holds, since actions of T and ν are related by program order accordingly. Knowing that $(\mathsf{HB} \,;\, (\mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}))$ is irreflexive, we conclude that it is only possible that $T \xrightarrow{\mathsf{HB}} \nu$.

We now consider T and ν that are by different threads. Let ψ' be the request action of ν. It is easy to see from Definition 4.19 of edges of the opacity graph that there exists an action $\psi \in T$ and a register x such that ψ and ψ' access x. Since ψ and ψ' are by different threads and access the same register, they form a conflict. However, H is DRF, so by Definition 4.4, either $\psi <_{\mathsf{hb}(H)} \psi'$ or $\psi' <_{\mathsf{hb}(H)} \psi$ holds. When lifted to nodes of the opacity graph, it is the case that either $T \xrightarrow{\mathsf{HB}} \nu$ or $\nu \xrightarrow{\mathsf{HB}} T$. Knowing that $(\mathsf{HB} \,;\, (\mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}))$ is irreflexive, we conclude that it is only possible that $T \xrightarrow{\mathsf{HB}} \nu$. □

Given a history H and a graph $G = (H, \_, \_, \mathsf{WR}, \_, \_)$, we write $n \xrightarrow{\mathsf{TXWR}} n'$ when $n, n' \in \mathsf{txns}(H)$ and $n \xrightarrow{\mathsf{WR}} n'$. We also use a shorthand $\mathsf{xpotxwr}(H) = \bigcup_{x \in \mathsf{Reg}} (\mathsf{xpo}(H) \,;\, \mathsf{txwr}_x(H))$.

Definition 4.27. *A* hb*-path π in a history H is a (possibly empty) sequence of triples:*

$$\pi = (\psi_1, R_1, \psi_2)(\psi_2, R_2, \psi_3) \dots (\psi_{m-1}, R_{m-1}, \psi_m),$$

such that:

- $\psi_1, \psi_2, \dots, \psi_m \in \mathsf{act}(H)$*;*
- *each R_i $(1 \le i \le m-1)$ is one of the following relations:* $\mathsf{po}(H)$*,* $\mathsf{cl}(H)$*,* $\mathsf{af}(H)$*,* $\mathsf{bf}(H)$ *and* $\mathsf{xpotxwr}(H)$*.*
- *for every i such that $1 \le i \le m-1$, $\psi_i <_{R_i} \psi_{i+1}$.*

We let $\mathsf{Paths}(H)$ *denote the set of all* hb*-paths in H.*

Definition 4.28. *Given a history H and transactions $T, T' \in \mathsf{txns}(H)$, we let* $\mathsf{TXPaths}_n(H, T, T') \subseteq \mathsf{Paths}(H)$ *denote the set of all* hb*-paths satisfying the following conditions:*

- *the* hb*-path starts with an action of T and ends with an action of T';*

- *n is the number of occurrences of triples of the form $(\_, \mathsf{xpotxwr}(H), \_)$ in the* hb*-path.*

Lemma 4.29. *For a history $H \in \mathsf{History}$, an opacity graph $G = (N, \mathsf{vis}, \mathsf{HB}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}) \in \mathsf{Graph}(H)$ and two distinct transactions $T, T' \in \mathsf{txns}(H)$, if $T \xrightarrow{\mathsf{HB}} T'$, then $T \xrightarrow{\mathsf{RT} \cup \mathsf{TXWR}}{}^{+} T'$.*

Proof. By Definition 4.19, $T \xrightarrow{\mathsf{HB}} T'$ if and only if there exist $\psi \in T$ and $\psi' \in T'$ such that $\psi <_{\mathsf{hb}(H)} \psi'$. It is easy to see that $T \xrightarrow{\mathsf{HB}} T'$ holds if and only if $\bigcup_{n \geq 0} \mathsf{TXPaths}_n(H, T, T') \neq \emptyset$. Thus, to prove the lemma, we demonstrate that for a given history H and its graph $G = (N, \mathsf{vis}, \mathsf{HB}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}) \in \mathsf{Graph}(H)$, the following holds:

$$\forall T, T' \in \mathsf{txns}(H).\, T \neq T' \wedge \bigcup_{n \geq 0} \mathsf{TXPaths}_n(H, T, T') \neq \emptyset \implies T \xrightarrow{\mathsf{RT} \cup \mathsf{TXWR}}{}^{+} T'. \quad (4.30)$$

We define $\Phi(n)$ as the following auxiliary statement:

$$\forall T, T' \in \mathsf{txns}(H).\, T \neq T' \wedge \mathsf{TXPaths}_n(H, T, T') \neq \emptyset \implies T \xrightarrow{\mathsf{RT} \cup \mathsf{TXWR}}{}^{+} T'.$$

To prove (4.30), we show that $\Phi(n)$ holds for all $n \geq 0$ by induction on n.

Base of the induction. Let us consider two distinct transactions T and T' and the hb-paths $\mathsf{TXPaths}_0(H, T, T')$ between them. Since these hb-paths begin and end in different transactions and only feature relations $\mathsf{po}(H)$, $\mathsf{cl}(H)$, $\mathsf{af}(H)$ and $\mathsf{bf}(H)$, every hb-path $\mathsf{TXPaths}_0(H, T, T')$ includes actions denoting the end of T and the beginning of T'. Hence, there exists a path $\pi = (\psi, \_, \_) \ldots (\_, \_, \psi') \in \mathsf{TXPaths}_0(H, T, T')$ such that ψ is the end of a transaction T and ψ' is the beginning of a transaction T'. Since relations $\mathsf{po}(H)$, $\mathsf{cl}(H)$, $\mathsf{af}(H)$ and $\mathsf{bf}(H)$ are all consistent with the history execution order $<_H$ (see §4.2), $\psi <_H \psi'$. Therefore, $T \xrightarrow{\mathsf{RT}} T'$ holds.

Induction step. Assuming that $\Phi(i)$ holds for $i \leq n$, we demonstrate that so does $\Phi(n+1)$. Let us consider two distinct transactions $T, T' \in \mathsf{txns}(H)$ and a hb-path $\pi \in \mathsf{TXPaths}_{n+1}(H, T, T')$.

Consider actions ψ_1 and ψ_2 such that the triple $(\psi_1, \mathsf{xpotxwr}(H), \psi_2)$ occurs the latest in π: there exist π' and π'' such that $\pi = \pi'(\psi_1, \mathsf{xpotxwr}(H), \psi_2)\pi''$ and there is no triple $(\_, \mathsf{xpotxwr}(H), \_)$ in π''. By definition of $\mathsf{xpo}(H)$ and $\mathsf{txwr}(H)$, there exists a transaction T_1 and actions ψ_{begin} and ψ_{write} such that:

- ψ_{begin} denotes a beginning of a transaction T_1;
- ψ_{write} denotes a write request action by T_1;
- $\psi_1 <_{\mathsf{po}(H)} \psi_{\mathsf{begin}} <_{\mathsf{po}(H)} \psi_{\mathsf{write}} <_{\mathsf{txwr\_}(H)} \psi_2$ holds.

To conclude the induction step, we make the following three observations. Firstly, note that the hb-path $\pi'(\psi_1, \mathsf{po}(H), \psi_{\mathsf{write}}) \in \mathsf{TXPaths}_n(H, T, T_1)$, and, therefore, $T \xrightarrow{\mathsf{RT} \cup \mathsf{TXWR}}{}^{+} T_1$ holds by the induction hypothesis $\Phi(n)$. Secondly, let T_2 be the transaction of ψ_2. Since $\psi_{\mathsf{write}} <_{\mathsf{txwr\_}(H)} \psi_2$ holds, so does $T_1 \xrightarrow{\mathsf{TXWR}} T_2$. Finally, $\pi'' \in \mathsf{TXPaths}_0(H, T_2, T')$. By the induction hypothesis $\Phi(0)$, $T_2 \xrightarrow{\mathsf{RT} \cup \mathsf{TXWR}}{}^{+} T'$ holds. Altogether, the three observations imply $T \xrightarrow{\mathsf{RT} \cup \mathsf{TXWR}}{}^{+} T'$. □

Proof of Theorem 4.22. Let us consider a DRF history H and its graph $G = (N, \mathsf{vis}, \mathsf{HB}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}) \in \mathsf{Graph}(H)$. Consider a cycle π in G. Without loss of generality, we can assume that all vertices on the cycle are distinct.

We first consider the case when all nodes in the cycle π are non-transactional accesses. Note that non-transactional accesses of H are totally ordered by $\mathsf{cl}(H)$. Knowing that $\mathsf{cl}(H)$ is consistent with the execution order $<_H$ of the history H, we conclude that there cannot be such cycle π.

In order to construct a cycle π' over transactions only with edges from $\mathsf{RT} \cup \mathsf{DEP}$, we consider any two adjacent transactions in the cycle π, i.e., such transactions T and T' that no other transaction appears between T and T'. We then demonstrate that in the opacity graph G, there is a path from T to T' in $\mathsf{RT} \cup \mathsf{DEP}$ over transactions only, and we add the path to π'.

There are two possibilities: either T' immediately follows T in the cycle, or they are separated by non-transactional accesses.

When T' immediately follows T in the cycle π, the two transactions are connected by an opacity graph edge $\mathsf{HB} \cup \mathsf{DEP}$. If $T \xrightarrow{\mathsf{HB}} T'$, then by Lemma 4.29, $T \xrightarrow{\mathsf{RT} \cup \mathsf{TXWR}}{}^{+} T'$, so we add the latter edges to π'. If $T \xrightarrow{\mathsf{DEP}} T'$, then we add the same edge to π'.

When T and T' are separated by other actions in the cycle π, we consider a path between the two transactions:

$$T \xrightarrow{\mathsf{HB} \cup \mathsf{DEP}} n_1 \xrightarrow{\mathsf{HB} \cup \mathsf{DEP}} n_2 \xrightarrow{\mathsf{HB} \cup \mathsf{DEP}} \ldots \xrightarrow{\mathsf{HB} \cup \mathsf{DEP}} n_K \xrightarrow{\mathsf{HB} \cup \mathsf{DEP}} T',$$

where each $n_i \in \mathsf{nontxn}(H)$ $(i = 1..K)$ is a node of the graph G denoting a non-transactional access. By Lemma 4.26, all non-HB edges on the path can be replaced by HB:

$$T \xrightarrow{\mathsf{HB}} n_1 \xrightarrow{\mathsf{HB}} n_2 \xrightarrow{\mathsf{HB}} \ldots \xrightarrow{\mathsf{HB}} n_K \xrightarrow{\mathsf{HB}} T'$$

Therefore, $T \xrightarrow{\mathsf{HB}} T'$ holds. By Lemma 4.29, there is a sequence of edges $T \xrightarrow{\mathsf{RT} \cup \mathsf{TXWR}}{}^{+} T'$. We add all of those edges to the cycle π'. □

Analogously to Theorem 4.22, we can prove the following lemma. Let txWR, txWW, txRW denote WR, WW and RW dependencies between transactions accordingly.

Lemma 4.31. *Let a data-race free history H and an opacity graph $G = (N, \mathsf{vis}, \mathsf{HB}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}) \in \mathsf{Graph}(H)$ be such that the relation $(\mathsf{HB} \,;\, (\mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}))$ is irreflexive. If $T \xrightarrow{\mathsf{HB} \cup \mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}}{}^{+} T'$ holds, then so does $T \xrightarrow{\mathsf{RT} \cup \mathsf{txWR} \cup \mathsf{txWW} \cup \mathsf{txRW}}{}^{+} T'$.*

That is, if there exists a path over edges $\mathsf{HB} \cup \mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}$ between two distinct transactions T and T' of the graph G, there also exists a path consisting only of edges from $\mathsf{RT} \cup \mathsf{txWR} \cup \mathsf{txWW} \cup \mathsf{txRW}$ (note that RT edges are not necessarily present in the graph G).

4.6 Case Study: TL2

The TL2 algorithm. The metadata maintained by the TL2 software TM are summarized in Figure 4.8. For each register x, TL2 maintains its value $\texttt{reg}[x]$,

version number $\texttt{ver}[x]$ and a write-lock $\texttt{lock}[x]$. New version numbers are generated with the help of a global counter `clock`, which transactions advance on commit. For every thread t, the TM maintains a flag $\texttt{active}[t]$, which indicates that the thread t is currently performing a transaction and is used to implement fences. TL2 also maintains metadata for each transaction T: a read-set $\texttt{rset}[T]$ of registers T has read from, a write-set $\texttt{wset}[T]$ of registers and values T intends to write to.

For brevity, we only provide pseudocode for transaction commits and fences, and describe the initialization, read, and write informally. When a transaction T starts in a thread t, it sets the flag $\texttt{active}[t]$ to true, and stores the value of `clock` into a local variable $\texttt{rver}[T]$, which determines T's *read timestamp*: TL2 allows T to read registers only with versions less than or equal to $\texttt{rver}[T]$. The write of a value v into a register x simply adds the pair (x, v) to the write-set $\texttt{wset}[T]$.

Each time T performs a read from a register x, it first checks if it has already performed a write to x, in which case it returns that the value for x from the write-set $\texttt{wset}[T]$. In other cases, T reads the current value $\texttt{reg}[x]$ and checks that its version is less than or equal to $\texttt{rver}[T]$; if not, TL2 aborts the transaction.

Upon a commit, the current transaction T executes the function `txcommit` in Figure 4.8. The commit starts by acquiring locks on each register in the write-set $\texttt{wset}[T]$ (lines 11–18). Next, T fetches-and-increments the value of `clock`, which it stores into $\texttt{wver}[T]$ and uses as the version for the new values T will write to registers—its *write timestamp* (line 21). Afterwards, T ensures that each register x in $\texttt{rset}[T]$ has not been modified during the execution of T by checking that x's version $\texttt{ver}[x]$ remains less than or equal to $\texttt{rver}[T]$ and that x is not currently locked (lines 22–28). The transaction then proceeds to write to the registers and release the locks one register at a time (lines 31–34). Finally, T commits. Upon aborting or committing at lines 18, 28 or 36, T executes a handler that clears the $\texttt{active}[t]$ flag (not shown in the code).

We consider a simple implementation of transactional fences in lines 39–45 (taken from [72]). The implementation works in two steps: it first determines which transactions the fence should wait for by checking and storing their `active` flags, and then blocks until the threads performing those transactions clear their `active` flags.

Proof overview. Due to space constraints, we only give an overview of the proof of the strong opacity of TL2. To generate the set $\mathcal{H}_{\mathsf{TL2}}$ of all histories of TL2, we consider the *most general client* of TL2: a program where every thread non-deterministically chooses the commands to execute. The well-formedness conditions on fences from Definition 4.1 can be established with a simple reasoning about the `fence` function in lines 39–45 independently from the rest of the proof. We prove strong opacity using Theorem 4.21: for every execution of the most general client of TL2 with a DRF history H, we show that H is consistent and build an opacity graph. To this end, we only need to define a visibility relation vis and write-dependencies WW, as the other components of the graph can be computed from these and H.

The consistency proof and the construction of the graph are inductive in the length of the execution of the most general client. We start with an empty

```
1  Value clock, reg[NRegs], ver[NRegs];
2  Lock lock[NRegs];
3  Bool active[NThreads];
4  Set<Register> rset; // for each transaction
5  Map<Register, Value> wset; // for each transaction
6  Value rver; // for each transaction, initially ⊥
7  Value wver; // for each transaction, initially ⊤
8
9  function txcommit(Transaction T) {
10   Set<Lock> lset := ∅;
11   foreach x in wset[T] {
12     Bool locked := lock[x].trylock();
13     if (¬locked)
14       lset.add(x);
15     else {
16       foreach y in lset[T]
17         lock[y].unlock();
18       return aborted(T);
19     }
20   }
21   wver[T] := fetch_and_increment(clock)+1;
22   foreach x in rset[T].keys() {
23     Bool locked := lock[x].test();
24     Value ts := ver[x];
25     if (locked ∨ rver[T] < ts) {
26       foreach y in lset[T]
27         lock[y].unlock();
28       return aborted(T);
29     }
30   }
31   foreach (x, v) in wset[T] {
32     reg[x] := v;
33     ver[x] := wver[T];
34     lock[x].unlock();
35   }
36   return committed(T);
37 }
38
39 function fence() {
40   Bool r[NThreads]; // initially all false
41   foreach t in ThreadID
42     r[t] := active[t];
43   foreach t in ThreadID
44     if (r[t])
45       while (active[t]);
46   return;
47 }
```

Figure 4.8: A fragment of the TL2 algorithm.

trace, an empty history and an empty graph, and extend them as the executions proceeds. Whenever a non-transactional access ν is executed, we add a new visible node to the graph. When ν is a write to a register x, we also append it to the total order WW_x. Whenever a new transaction T starts, we add a corresponding invisible node. When T executes the `txcommit` function, if it reaches line 31, then we are sure T is going to commit. At this point we therefore we make T visible and append it to the total order WW_x for each register $x \in \mathtt{wset}[T]$.

We need to show that, whenever the graph is extended with new edges, it stays acyclic. To this end, we use Theorem 4.22 to reduce the acyclicity check to the one required when proving the usual opacity, i.e., checking the absence of cycles over transactions in $\mathsf{RT} \cup \mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}$ (checking the absence of cycles with a single dependency is easier and we omit its description for brevity). In our proof, only graph updates of the read and commit operations of each transaction impose proof obligations.

At every step of the graph construction, we maintain an inductive invariant that helps us prove both consistency of the history and the acyclicity of $\mathsf{RT} \cup \mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}$. Its most important part associates a notion of time with the edges of the graph based on the read and write timestamps of transactions:

1. $\forall T, T'.\, T \xrightarrow{\mathsf{RT}} T' \implies \mathtt{rver}[T'] = \bot \vee$
   $((\mathsf{vis}(T) \implies \mathtt{wver}[T] \leq \mathtt{rver}[T']) \wedge (\neg\mathsf{vis}(T) \implies \mathtt{rver}[T] \leq \mathtt{rver}[T'])).$
2. $\forall T, T'.\, T \xrightarrow{\mathsf{WR}} T' \implies \mathtt{wver}[T] \leq \mathtt{rver}[T'].$
3. $\forall T, T'.\, T \xrightarrow{\mathsf{RW}} T' \implies \mathtt{rver}[T] < \mathtt{wver}[T'].$
4. $\forall T, T'.\, T \xrightarrow{\mathsf{WW}} T' \implies \mathtt{wver}[T] < \mathtt{wver}[T'].$

Property 1 asserts that, whenever a transaction T' occurs after a completed transaction T in the real time, it either has not yet generated a read timestamp $\mathtt{rver}[T']$, or it has and $\mathtt{rver}[T']$ is greater or equal to $\mathtt{wver}[T]$ (when T is visible and, therefore, committed) or $\mathtt{rver}[T]$ (otherwise). Property 2 asserts that, whenever a transaction T' reads a value of a register written by a transaction T, the version that T' assigned to the register may not be greater than the read timestamp of T. This is validated by the check TL2 performs when reading registers. Property 3 asserts that a transaction T' overwriting the value read by a transaction T has the write timestamp greater than the read timestamp of T. It holds because, if T' commits its write after T reads the previous value of the register, then T generates its read timestamp before T' generates its write timestamp. Property 4 follows from the mutual exclusion that TL2 ensures for committing transactions that write the same register x (using $\mathtt{lock}[x]$). Since writes in commit operations occur within a critical section, write dependencies are always consistent with the order on write timestamps.

With the help of the above invariant, we establish that for a path between any transactions T and T' in the graph, certain inequalities between their timestamps take place depending on visibility of the two transactions, such as the following:

$$\mathsf{vis}(T) \wedge \mathsf{vis}(T') \implies \mathtt{wver}[T] < \mathtt{wver}[T']. \tag{4.32}$$

Using this and other minor observations, we can demonstrate that graph updates preserve the acyclicity of $\mathsf{RT} \cup \mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}$, by showing that a cycle would

imply a contradiction involving the timestamps of transactions. As an example of such reasoning, consider a transaction T executing the `txcommit` operation. As T reaches line 31, we mark it as visible and add new write dependencies in the graph. Let us assume that adding $T' \xrightarrow{\mathsf{WW}} T$, where T' is some transaction, causes a cycle over transactions. Then there must exist a path from T to T'. Note that $\mathsf{vis}(T)$ and $\mathsf{vis}(T')$ both hold, since they are ordered by WW. By (4.32), $\mathtt{wver}[T] < \mathtt{wver}[T']$ holds, because there is a path from T to T'. On the other hand, Property 4 above gives us $\mathtt{wver}[T'] < \mathtt{wver}[T]$, since $T' \xrightarrow{\mathsf{WW}} T$ is in the graph. Thus, we have arrived to a contradiction.

4.7 Related Work

In this chapter we have concentrated on one technique for ensuring privatization safety—transactional fences. However, there have been several proposals of alternative techniques (see [14, §4.6.1] for a survey), and in the future, we plan to address these. In particular, some TMs do not require transactional fences for safe privatization [73, 74, 75, 76], even though the programmer still has to follow a certain DRF discipline. Such a discipline has been proposed by Dalessandro and Scott [20, 21], but it did not come with a formal justification, such as our proofs of the Fundamental Property and TM correctness.

Kestor et al. [59] proposed a notion of DRF for TMs that do not support safe privatization and a race-detection tool for this notion. Unlike us, they do not consider transactional fences, so that the only way to safely privatize an object is to agree on its status outside transactions (Figure 4.7). Our notion of DRF specializes to the one by Kestor et al. if we consider only histories without fences.

Lesani et al. [60] proposed a transactional DRF based on TMS [77], a TM consistency criterion. However, as they acknowledge, their proposal does not support privatization.

To the best of our knowledge, a line of work by Abadi et al. was the only one that proposed disciplines for privatization with a formal justification of their safety [56, 78]. However, they did not take into account transactional fences and considered programming disciplines more restrictive than ours. Their *static separation* [56] ensures strong atomicity by not mixing transactional and non-transactional accesses to the same register. *Dynamic separation* [78] relaxes this by introducing explicit commands to privatize and publish an object. We believe such disciplines are particular ways of achieving the more general notion of data-race freedom that we adopted.

Gotsman et al. have previously proposed a logic for reasoning about programs using RCU [72]. Since transactional fences are similar to RCU, we believe this logic can be adapted to guide programmers in inserting fences to satisfy our notion of DRF.

In this chapter we assumed sequential consistency as a baseline non-transactional memory model. However, transactions are being integrated into languages, such as C++, that have weaker memory models [79]. Our definition of a data race is given in the axiomatic style used in the C++ memory model [55]. For this reason, we believe that our results can in the future be adapted to the more complex setting of C++.

Guerraoui et al. [80] considered TMs that provide strong atomicity without making any assumptions about the client program. They formalized the requirement on such TMs as parameterized opacity and proved the impossibility of achieving it on many memory models without instrumenting non-transactional accesses. This result justifies our decision to provide strong atomicity only to DRF programs.

Chapter 5

Conclusion

This thesis presents advances in program logics and proof techniques for reasoning about fine-grained concurrent implementations of data structures and transactional memory. It also provides formal foundations for the programming model in which memory can be accesses both inside and outside of transactions.

Firstly, we have presented a generic logic for proving linearizability of concurrent data structures. It unifies the various logics based on linearization points with helping. As designing a new logic for reasoning about a new class of algorithms requires finding the proof rules and proving their soundness afresh, our goal in Chapter 2 was to propose a framework for designing such logics and to formalize the method they use for reasoning about linearizability in a way independent of the particular thread-modular reasoning technique. We have shown instantiations of our logic based on disjoint concurrent separation logic [28] and RGSep [24]. However, we expect that our logic can also be instantiated with more complex thread-modular reasoning methods, such as those based on concurrent abstract predicates [29] or islands and protocols [33].

The notable limitation of the approach we took in Chapter 2 is that of the linearization-point method. It is limited in the range of algorithms it can handle: in particular, algorithms requiring reasoning about future behavior of operations in the execution are notoriously difficult to prove using linearization points. In Chapter 3, we presented a new proof method which lifts these limitations, while preserving the inductive proof structure of traditional linearization points. As with linearization points, our key idea can be explained simply: at commitment points, operations impose order between themselves and other operations, and all linearizations of the order must satisfy the sequential specification. Nonetheless, our method generalizes to algorithms such as the Herlihy-Wing queue and the Time-stamped Queue, which are known to be challenging to reason about in terms of linearization points.

In Chapter 4, we studied the problem of safe privatization support in transactional memory (TM). We proposed a notion of data-race freedom (DRF) and showed that DRF programs get strong atomicity guarantees from a TM system and, therefore, privatization-safety, under assumption that the TM system satisfies strong opacity. The notion of data-race freedom supports the standard primitive for ensuring privatization safety, i.e., transactional fences. However, there have been several proposals of alternative approaches and in the future, we plan to address these.

We conclude by noting some directions of further research that our results suggest.

5.1 Future Directions

The Views Framework for liveness-preserving linearizability. In this dissertation we concentrated on linearizability in its original form [7], which considers only finite computations and, hence, specifies only safety properties of the library. Linearizability has since been generalized to also specify liveness properties [37]. A possible direction of future work is to generalize our generic logic from Chapter 2 to handle liveness, possibly building on ideas from [38].

Reasoning about future-dependent linearizations in the Views Framework. We formalized our proof method from Chapter 3 as a Hoare logic based on rely-guarantee [16]. Our method is general and can be combined with more advanced methods for reasoning about concurrency [24, 29, 45]. However, since modifying those logics accordingly requires re-proving their soundness, extending the Views framework to support the proof method of using partial orders could be a possible direction of future work.

Automation of proving linearizability using partial orders. In this dissertation we have concentrated on simplifying manual proofs. However, our approach in Chapter 3 also seems like a promising candidate for automation, as it requires no special meta-theory, just reasoning about partial orders. We are hopeful that we can automate such arguments using off-the-shelf solvers such as Z3, and we plan to experiment with this in future.

DRF for privatization-safe TMs. In Chapter 4 we have concentrated on one technique for ensuring privatization safety—transactional fences. However, there have been several proposals of alternative techniques (see [14, §4.6.1] for a survey), and in the future, we plan to address these. In particular, some TMs do not require transactional fences for safe privatization [73, 74, 75, 76], even though the programmer still has to follow a certain DRF discipline. Such a discipline has been proposed by Dalessandro and Scott [20, 21], but it did not come with a formal justification, such as our proofs of the Fundamental Property and TM correctness.

DRF for strong atomicity under weak memory models. Our work in Chapter 4 assumed sequential consistency as a baseline non-transactional memory model. However, transactions are being integrated into languages, such as C++, that have weaker memory models [79]. Our definition of a data race is given in the axiomatic style used in the C++ memory model [55]. For this reason, we believe that our results can in the future be adapted to the more complex setting of C++.

Transactional race-detection tools. Kestor et al. [59] proposed a notion of DRF for TMs that do not support safe privatization and a race-detection tool for this notion. Unlike us, they do not consider transactional fences, so

that the only way to safely privatize an object is to agree on its status outside transactions (Figure 4.7). Our notion of DRF specializes to the one by Kestor et al. if we consider only histories without fences. We hope that, in the future, race-detection tools like the one of Kestor et al. can be adapted to detect our notion of data races.

Bibliography

[1] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *International Symposium on Computer Architecture (ISCA)*, pp. 289–300, 1993.

[2] N. Shavit and D. Touitou, "Software transactional memory," *Distributed Computing*, vol. 10, no. 2, pp. 99–116, 1997.

[3] Intel Corporation, "Intel architecture instruction set extensions programming reference. Chapter 8: Intel transactional synchronization extensions," 2012.

[4] H. Q. Le, G. L. Guthrie, D. E. Williams, M. M. Michael, B. G. Frey, W. J. Starke, C. May, R. Odaira, and T. Nakaike, "Transactional memory support in the ibm power8 processor," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 8:1–8:14, 2015.

[5] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen, "Hybrid transactional memory," in *Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pp. 209–220, 2006.

[6] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid transactional memory," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 336–346, 2006.

[7] M. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM TOPLAS*, 1990.

[8] I. Filipovic, P. O'Hearn, N. Rinetzky, and H. Yang, "Abstraction for concurrent objects," *Theoretical Computer Science*, vol. 411, no. 51-52, pp. 4379 – 4398, 2010.

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

[10] R. Guerraoui and M. Kapalka, "On the correctness of transactional memory," in *Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pp. 175–184, 2008.

[11] H. Attiya, G. Ramalingam, and N. Rinetzky, "Sequential verification of serializability," in *Proceedings of the 37th ACM Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pp. 31–42, 2010.

[12] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott, "Privatization techniques for software transactional memory," Tech. Rep. 915, Computer Science Department, University of Rochester, 2007.

[13] C. Blundell, E. C. Lewis, and M. M. K. Martin, "Subtleties of transactional memory atomicity semantics," *IEEE Computer Architecture Letters*, vol. 5, no. 2, 2006.

[14] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory*. Morgan and Claypool Publishers, 2nd ed., 2010.

[15] P. W. O'Hearn, J. C. Reynolds, and H. Yang, "Local reasoning about programs that alter data structures," in *CSL*, 2001.

[16] C. B. Jones, "Specification and design of (parallel) programs," in *IFIP Congress*, 1983.

[17] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. J. Parkinson, and H. Yang, "Views: compositional reasoning for concurrent programs," in *POPL*, 2013.

[18] A. Haas, *Fast Concurrent Data Structures Through Timestamping*. PhD thesis, University of Salzburg, 2015.

[19] P. W. O'Hearn, N. Rinetzky, M. T. Vechev, E. Yahav, and G. Yorsh, "Verifying linearizability with hindsight," in *PODC*, 2010.

[20] L. Dalessandro and M. L. Scott, "Strong isolation is a weak idea," in *Workshop on Transactional Computing (TRANSACT)*, 2009.

[21] L. Dalessandro, M. L. Scott, and M. F. Spear, "Transactions as the foundation of a memory consistency model," in *International Symposium on Distributed Computing (DISC)*, pp. 20–34, 2010.

[22] D. Dice, O. Shalev, and N. Shavit, "Transactional locking II," in *International Symposium on Distributed Computing (DISC)*, pp. 194–208, 2006.

[23] B. Dongol and J. Derrick, "Verifying linearizability: A comparative survey," *arXiv CoRR*, vol. 1410.6268, 2014.

[24] V. Vafeiadis, *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, UK, 2008. Technical Report UCAM-CL-TR-726.

[25] H. Liang and X. Feng, "Modular verification of linearizability with non-fixed linearization points," in *PLDI*, 2013.

[26] X. Feng, "Local rely-guarantee reasoning," in *POPL*, 2009.

[27] P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner, "TaDA: A logic for time and data abstraction," in *ECOOP*, 2014.

[28] P. W. O'Hearn, "Resources, concurrency, and local reasoning," *Theoretical Computer Science*, 2007.

[29] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis, "Concurrent abstract predicates," in *European Conference on Object-Oriented Programming, ECOOP 2010*, 2010.

[30] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson, "Permission accounting in separation logic," in *POPL*, 2005.

[31] A. Gotsman and H. Yang, "Linearizability with ownership transfer," *LMCS*, 2013.

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

[33] A. J. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer, "Logical relations for fine-grained concurrency," in *POPL*, 2013.

[34] G. Schellhorn, H. Wehrheim, and J. Derrick, "How to prove algorithms linearisable," in *CAV*, 2012.

[35] T. A. Henzinger, A. Sezgin, and V. Vafeiadis, "Aspect-oriented linearizability proofs," in *CONCUR*, 2013.

[36] M. Dodds, A. Haas, and C. M. Kirsch, "A scalable, correct time-stamped stack," in *POPL*, 2015.

[37] A. Gotsman and H. Yang, "Liveness-preserving atomicity abstraction," in *ICALP*, 2011.

[38] H. Liang, X. Feng, and Z. Shao, "Compositional verification of termination-preserving refinement of concurrent programs," in *LICS*, 2014.

[39] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer, "Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning," in *POPL*, 2015.

[40] K. Svendsen and L. Birkedal, "Impredicative concurrent abstract predicates," in *ESOP*, 2014.

[41] I. Sergey, A. Nanevski, and A. Banerjee, "Specifying and verifying concurrent algorithms with histories and subjectivity," in *ESOP*, 2015.

[42] M. Hoffman, O. Shalev, and N. Shavit, "The baskets queue," in *OPODIS*, Springer, 2007.

[43] A. Morrison and Y. Afek, "Fast concurrent queues for x86 processors," in *PPoPP*, 2013.

[44] P. C. Fishburn, "Intransitive indifference with unequal indifference intervals," *Journal of Mathematical Psychology*, vol. 7, 1970.

[45] A. Turon, D. Dreyer, and L. Birkedal, "Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency," in *ICFP*, 2013.

[46] S. S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs I," *Acta Informatica*, vol. 6, 1976.

[47] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, and N. Shavit, "A lazy concurrent list-based set algorithm," in *OPODIS*, 2005.

[48] G. Schellhorn, J. Derrick, and H. Wehrheim, "A sound and complete proof technique for linearizability of concurrent data structures," *ACM TOCL*, vol. 15, 2014.

[49] N. Hemed, N. Rinetzky, and V. Vafeiadis, "Modular verification of concurrency-aware linearizability," in *DISC*, 2015.

[50] G. A. Delbianco, I. Sergey, A. Nanevski, and A. Banerjee, "Concurrent data structures linked in time," in *European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, pp. 8:1–8:30, 2017.

[51] H. Attiya, A. Gotsman, S. Hans, and N. Rinetzky, "A programming language perspective on transactional memory consistency," in *Symposium on Principles of Distributed Computing (PODC)*, pp. 309–318, 2013.

[52] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Computers*, vol. 28, no. 9, pp. 690–691, 1979.

[53] S. V. Adve and M. D. Hill, "Weak ordering - A new definition," in *International Symposium on Computer Architecture (ISCA)*, pp. 2–14, 1990.

[54] J. Manson, W. Pugh, and S. V. Adve, "The java memory model," in *Symposium on Principles of Programming Languages (POPL)*, pp. 378–391, 2005.

[55] *ISO/IEC. Programming Languages — C++, 14882:2017.* 2017.

[56] M. Abadi, A. Birrell, T. Harris, and M. Isard, "Semantics of transactional memory and automatic mutual exclusion," *ACM Trans. Program. Lang. Syst.*, vol. 33, pp. 2:1–2:50, 2011.

[57] M. Abadi, T. Harris, and K. F. Moore, "A model of dynamic separation for transactional memory," in *International Conference on Concurrency Theory (CONCUR)*, pp. 6–20, 2008.

[58] K. F. Moore and D. Grossman, "High-level small-step operational semantics for transactions," in *Symposium on Principles of Programming Languages (POPL)*, pp. 51–62, 2008.

[59] G. Kestor, O. S. Unsal, A. Cristal, and S. Tasiran, "T-rex: a dynamic race detection tool for C/C++ transactional memory applications," in *European Systems Conference (Eurosys)*, pp. 20:1–20:12, 2014.

[60] M. Lesani, V. Luchangco, and M. Moir, "Specifying transactional memories with nontransactional operations," in *Workshop on the Theory of Transactional Memory (WTTM)*, 2013.

[61] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg, "McRT-STM: a high performance software transactional memory system for a multi-core runtime," in *Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pp. 187–197, 2006.

[62] P. Felber, C. Fetzer, P. Marlier, and T. Riegel, "Time-based software transactional memory," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 12, pp. 1793–1807, 2010.

[63] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott., "Lowering the overhead of software transactional memory.," in *Workshop on Transactional Computing (TRANSACT)*, 2006.

[64] P. E. McKenney, *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.

[65] R. M. Yoo, Y. Ni, A. Welc, B. Saha, A. Adl-Tabatabai, and H. S. Lee, "Kicking the tires of software transactional memory: why the going gets tough," in *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 265–274, 2008.

[66] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *International Symposium on Workload Characterization (IISWC)*, pp. 35–46, 2008.

[67] T. Zhou, P. Zardoshti, and M. F. Spear, "Practical experience with transactional lock elision," in *International Conference on Parallel Processing (ICPP)*, pp. 81–90, 2017.

[68] R. Guerraoui and M. Kapalka, *Principles of Transactional Memory*. Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers, 2010.

[69] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman, "Open nesting in software transactional memory," in *Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pp. 68–78, 2007.

[70] J. E. B. Moss and A. L. Hosking, "Nested transactional memory: model and architecture sketches," *Science of Computer Programming*, vol. 63, no. 2, pp. 186–201, 2006.

[71] T. Shpeisman, A.-R. Adl-Tabatabai, R. Geva, Y. Ni, and A. Welc, "Towards transactional memory semantics for C++," in *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 49–58, 2009.

[72] A. Gotsman, N. Rinetzky, and H. Yang, "Verifying concurrent memory reclamation algorithms with grace.," in *European Symposium on Programming (ESOP)*, pp. 249–269, 2013.

[73] L. Dalessandro, M. F. Spear, and M. L. Scott, "Norec: streamlining STM by abolishing ownership records," in *Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pp. 67–78, 2010.

[74] M. F. Spear, M. M. Michael, and C. von Praun, "RingSTM: Scalable transactions with a single atomic instruction," in *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 275–284, 2008.

[75] D. Dice and N. Shavit, "TLRW: return of the read-write lock," in *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 284–293, 2010.

[76] M. Olszewski, J. Cutler, and J. G. Steffan, "JudoSTM: A dynamic binary-rewriting approach to software transactional memory," in *International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pp. 365–375, 2007.

[77] S. Doherty, L. Groves, V. Luchangco, and M. Moir, "Towards formally specifying and verifying Transactional Memory," *Formal Aspects of Computing*, vol. 25, no. 5, pp. 769–799, 2013.

[78] M. Abadi, A. Birrell, T. Harris, J. Hsieh, and M. Isard, "Implementation and use of transactional memory with dynamic separation," in *International Conference on Compiler Construction (CC)*, pp. 63–77, 2009.

[79] *ISO/IEC. Technical Specification for C++ Extensions for Transactional Memory, 19841:2015.* 2015.

[80] R. Guerraoui, T. A. Henzinger, M. Kapalka, and V. Singh, "Transactions in the jungle," in *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 263–272, 2010.

Appendix A

Detailed Case Studies

A.1 Linearizability of the Time-Stamped Queue

A.1.1 Proof outline

We prove the following specifications for Enq and Deq for each thread $t \in \mathsf{ThreadID}$ (proof outlines are provided in Figure A.1 and Figure A.2):

$$R_t, G_t \vdash_t \{\mathsf{INV} \wedge \mathrm{started}(t, \mathrm{Enq})\}\ \mathrm{Enq}\ \{\mathsf{INV} \wedge \mathrm{ended}(t, \mathrm{Enq})\}$$
$$R_t, G_t \vdash_t \{\mathsf{INV} \wedge \mathrm{started}(t, \mathrm{Deq})\}\ \mathrm{Deq}\ \{\mathsf{INV} \wedge \mathrm{ended}(t, \mathrm{Deq})\}$$

In the specifications, INV is the global invariant, and R_t and G_t are rely and guarantee relations defined in §3.6. Assertions $\mathrm{started}(t, \mathsf{op})$ and $\mathrm{ended}(t, \mathsf{op})$ are defined in §3.5.

We prove the following specifications for Enq and Deq for each thread $t \in \mathsf{ThreadID}$ (proof outlines are provided in Figure A.1 and Figure A.2):

$$R_t, G_t \vdash_t \{\mathsf{INV} \wedge \mathrm{started}(t, \mathrm{Enq})\}\ \mathrm{Enq}\ \{\mathsf{INV} \wedge \mathrm{ended}(t, \mathrm{Enq})\}$$
$$R_t, G_t \vdash_t \{\mathsf{INV} \wedge \mathrm{started}(t, \mathrm{Deq})\}\ \mathrm{Deq}\ \{\mathsf{INV} \wedge \mathrm{ended}(t, \mathrm{Deq})\}$$

In the specifications, INV is the global invariant, and R_t and G_t are rely and guarantee relations defined in §3.6. Assertions $\mathrm{started}(t, \mathsf{op})$ and $\mathrm{ended}(t, \mathsf{op})$ are defined in §3.5.

We introduce assertions noPotCand and isPotCand to denote the properties analogous to noCand and isCand that hold of an enqueue event in the front of the pool of thread $\mathtt{k}$. To this end, we let $\mathsf{seen}_{\mathtt{A}}(\kappa, d)$ denote the set of enqueue events observed d in the pools of threads from $\mathtt{A}$ (see Definition 3.17), and we also let $\mathrm{minTS}_{\mathtt{A}}(\mathtt{ENQ})$ be a predicate asserting that $\mathtt{ENQ}$'s timestamp is minimal among enqueues in $\mathtt{pools}(\mathtt{k})$.

$$\begin{array}{lcl}
[\![\mathrm{noPotCand}]\!]_\ell & = & \{(s, H, G_{\mathsf{ts}}) \mid \mathsf{seen}_{\mathtt{k}}((s, H, G_{\mathsf{ts}}), \mathtt{myEid}()) = \emptyset \wedge s(\mathtt{pid}) = \mathrm{NULL}\}; \\
[\![\mathrm{isPotCand}]\!]_\ell & = & \{(s, H, G_{\mathsf{ts}}) \mid \exists \mathtt{ENQ}.\, \mathtt{ENQ} = \mathtt{enqOf}(E, G, s(\mathtt{k}), s(\mathtt{ts})) \\
 & & \quad \wedge \mathrm{minTS}_{\mathtt{k}}(\mathtt{ENQ}) \wedge (\mathtt{ENQ} \in \mathsf{inQ}(s(\mathtt{pools}), E, G_{\mathsf{ts}}) \implies \\
 & & \quad\quad \mathtt{ENQ} \in \mathsf{seen}_{\mathtt{k}}((s, H, G_{\mathsf{ts}}), \mathtt{myEid}())) \wedge s(\mathtt{pid}) \neq \mathrm{NULL}\};
\end{array}$$

We further explain how they are used in Appendix A.1.2.

```
1  enqueue(Val v) {
2    { INV ∧ started(t, Enq) }
3    atomic { // insert
4      PoolID node := insert(myTid(), v);
5      G_ts[myEid()] := ⊤;
6    }
7    { INV ∧ started(t, Enq) }
8    TS timestamp := newTimestamp();
9    { ∃T. INV ∧ started(t, Enq) ∧ newTS(timestamp) }
10   atomic { // setTS
11     setTimestamp(myTid(), node, timestamp);
12     G_ts[myEid()] := timestamp;
13     E(myEid()).rval := ⊥;
14   }
15   { INV ∧ ended(t, Enq) }
16   return ⊥;
17 }
```

Figure A.1: The proof outline for the enqueue method of the TS queue.

```
18 Val dequeue() {
19   { INV ∧ started(t, Deq) }
20   Val ret := NULL;
21   EventID CAND;
22   do {
23     TS start_ts := newTimestamp();
24     PoolID pid, cand_pid := NULL;
25     TS ts, cand_ts := ⊤;
26     ThreadID cand_tid;
27     { INV ∧ started(t, Deq) ∧ LI ∧ res[t] = NULL }
28     for each k in 1..NThreads do {
29       { INV ∧ started(t, Deq) ∧ LI ∧ res[t] = NULL }
30       atomic {
31         (pid, ts) := getOldest(k);
32         R := (R ∪ {(e, myEid()) | e ∈ id(⌊E⌋) ∩ inQ(pools, E, G_ts)
33                                   ∧ ¬(start_ts <_TS G_ts(e))})^+;
34       }
35       { INV ∧ started(t, Deq) ∧ LI ∧ res[t] = NULL
          ∧ (noPotCand ∨ isPotCand) }
36       if (pid ≠ NULL && ts <_TS cand_ts && ¬(start_ts <_TS ts)) {
37         (cand_pid, cand_ts, cand_tid) := (pid, ts, k);
38         CAND:= enqOf(E, G_ts, cand_tid, cand_ts);
39       }
40       { INV ∧ started(t, Deq) ∧ LI ∧ res[t] = NULL }
41     }
42     { INV ∧ started(t, Deq) ∧ LI ∧ res[t] = NULL }
43     if (cand_pid ≠ NULL)
44       { INV ∧ started(t, Deq) ∧ isCand ∧ cand_pid ≠ NULL ∧ res[t] = NULL }
45       atomic {  // remove
46         ret := remove(cand_tid, cand_pid);
47         if (ret ≠ NULL) {
48           E(myEid()).rval := ret;
49           R := (R ∪ {(CAND, e) | e ∈ inQ(pools, E, G_ts)}
50                   ∪ {(myEid(), d) | E(d).op = Deq ∧ d ∈ id(E \ ⌊E⌋)})^+;
51         }
52       }
53       { INV ∧ ((started(t, Deq) ∧ res[t] = NULL) ∨ ended(t, Deq)) }
54   } while (ret = NULL);
55   { INV ∧ ended(t, Deq) }
56   return ret;
57 }
```

Figure A.2: The proof outline for the dequeue method of the TS queue.

A.1.2 Preservation of the loop invariant.

We consider the current dequeue operation myEid() in a thread t, which generates a timestamp start_ts and proceeds to execute for each loop. At k's iteration of the loop, the following two steps performed:

- **Step 1:** Using getOldest(k), we learn a pool identifier pid and a timestamp ts of the value in the front of the queue (if there is any). Additionally, every enqueue in pools(k) with a timestamp not greater than start_ts is ordered in front of myEid(). As a result, one of the two cases takes place:
  - noPotCand holds, in which case we say that there is no potential candidate in pools(k). This describes configurations, in which pools(k) is either empty or ts is greater than start_ts.
  - isPotCand holds, in which case we say that the enqueue event $\texttt{ENQ} = \texttt{enqOf}(E, G_{\texttt{ts}}, \texttt{k}, \texttt{ts})$ is the potential candidate in pools(k). This describes configurations, in which ts is not greater than start_ts. Additionally, this requires that ts be smaller than other timestamps currently present in pools(k). Note that the latter follows from $\mathsf{INV}_{\mathsf{ALG}}$(ii) and $\mathsf{INV}_{\mathsf{ALG}}$(i) after Step 1.
- **Step 2:** if there is a potential candidate, its timestamp is compared to the timestamp cand_ts of the current candidate for removal and the earliest of the two is kept as the candidate;

To show that the loop invariant LI is preserved by the k's iteration, we consider separately the cases when there is the potential candidate ENQ and when there is no such enqueue event.

Let us first assume that isPotCand holds, and $\texttt{ENQ} = \texttt{enqOf}(E, G_{\texttt{ts}}, \texttt{k}, \texttt{ts})$ is the potential candidate in pools(k). At step 2 (line 36), the current dequeue compares ts to cand_ts and decides whether to choose ENQ as the candidate for removal. According to the loop invariant, there are two possibilities: either no candidate has been chosen so far (noCand holds), or there is a candidate CAND (isCand holds). When the former is the case, $\mathsf{seen}_{\texttt{A}}((s, H, G_{\texttt{ts}}), \texttt{myEid}()) = \emptyset$. It is easy to see that isPotCand immediately implies isCand after k's iteration. Let us now consider the case when isCand holds and CAND has been selected as the candidate for removal out of enqueues in threads from A. Let us assume that $\texttt{ts} <_{\texttt{TS}} \texttt{cand\_ts}$ takes place (the other situation is justified analogously). To conclude isCand for the next iteration, we need to show that:

- $\texttt{ENQ} \in \mathsf{inQ}(\texttt{pools}, E, G_{\texttt{ts}}) \implies \texttt{ENQ} \in \mathsf{seen}_{\texttt{A}\uplus\{\texttt{k}\}}((s, H, G_{\texttt{ts}}), \texttt{myEid}())$, and
- $\mathrm{minTS}_{\texttt{A}\uplus\{\texttt{k}\}}(\texttt{ENQ})$.

The first requirement follows trivially from isPotCand: if $\texttt{ENQ} \in \mathsf{inQ}(\texttt{pools}, E, G_{\texttt{ts}})$ holds, then so does:

$$\texttt{ENQ} \in \mathsf{seen}_{\{\texttt{k}\}}((s, H, G_{\texttt{ts}}), \texttt{myEid}()) \subseteq \mathsf{seen}_{\texttt{A}\uplus\{\texttt{k}\}}((s, H, G_{\texttt{ts}}), \texttt{myEid}()).$$

It remains to show that $\mathrm{minTS}_{\texttt{A}\uplus\{\texttt{k}\}}(\texttt{ENQ})$ holds, i.e. that every other enqueue in $\mathsf{seen}_{\texttt{A}\uplus\{\texttt{k}\}}((s, H, G_{\texttt{ts}}), \texttt{myEid}())$ does not have a timestamp smaller than ENQ. According to isPotCand, ENQ is minimal among enqueues in thread k. Let

us assume that ENQ is not minimal among enqueues in A, i.e. that there is $\texttt{ENQ}' \in \mathsf{seen}_{\texttt{A}}((\texttt{s}, \texttt{H}, \texttt{G}_{\mathsf{ts}}), \texttt{myEid}())$ such that $G_{\mathsf{ts}}(\texttt{ENQ}') <_{\mathsf{TS}} G_{\mathsf{ts}}(\texttt{ENQ})$. Knowing that $G_{\mathsf{ts}}(\texttt{ENQ}) <_{\mathsf{TS}} \texttt{cand\_ts}$, we conclude that $G_{\mathsf{ts}}(\texttt{ENQ}') <_{\mathsf{TS}} \texttt{cand\_ts}$, which contradicts the loop invariant. Therefore, ENQ is minimal among enqueues in both A and k.

Now let us assume that noPotCand holds, i.e. that there is no potential candidate in pools(k). In this case, the candidate for removal remains unchanged. Intuitively, when there is no potential candidate in pools(k), all values occurring in the pool have timestamps greater than start_ts. According to the invariant $\mathsf{INV}_{\mathsf{ALG}}$(i), all successors of corresponding events will have even greater timestamps.

Prior to k's iteration, either noCand or isCand holds. Let us first assume the former. Then no candidate has been chosen after iterating over A. Together, noCand and noPotCand immediately imply noCand for the next iteration. Let us now consider the case when isCand holds. Then there is a candidate for removal CAND. It is easy to see that $\texttt{CAND} \in \mathsf{seen}_{\texttt{A}\uplus\{\texttt{k}\}}(\kappa, \texttt{myEid}())$ holds, so it remains to ensure that $\mathrm{minTS}_{\texttt{A}\uplus\{\texttt{k}\}}(\texttt{CAND})$ holds. To this end, we need to demonstrate that for every enqueue $e \in \mathsf{seen}_{\texttt{k}}(\kappa, \texttt{myEid}())$, $\neg(G_{\mathsf{ts}}(e) <_{\mathsf{TS}} G_{\mathsf{ts}}(\texttt{CAND}))$ holds. However, according to noPotCand, there are no such enqueues e, so isCand can be concluded for the next iteration.

A.1.3 Auxiliary proofs for the loop invariant (Lemma 3.18)

Lemma A.1. *Given any configuration $\kappa = (s, (E, R), G_{\mathsf{ts}})$ satisfying* INV *and an identifier d of a dequeue event that has generated its timestamp* start_ts*, an enqueue by a visited thread not seen by d does not precede any enqueue seen by d:*

$$\forall i, i'.\, i \in \mathsf{inQ}(s, E, G_{\mathsf{ts}}) \setminus \mathsf{seen}(\kappa, d) \\ \wedge\, i' \in \mathsf{seen}(\kappa, d) \wedge \{E(i).\mathsf{tid}, E(i').\mathsf{tid}\} \subseteq \texttt{A} \implies \neg(i \xrightarrow{R} i').$$

Proof. We do a proof by contradiction. Let us assume that there exist i and i' satisfying the premise of the implication above and such that $i \xrightarrow{R} i'$ holds. Since $E(i).\mathsf{tid} \in \texttt{A}$ and $i \in \mathsf{inQ}(s, E, G_{\mathsf{ts}}) \setminus \mathsf{seen}(\kappa, d)$, the following holds by definition of seen:

(a) $i \notin \lfloor \mathsf{id}(E) \rfloor$,

(b) $s(\texttt{start\_ts}) <_{\mathsf{TS}} G_{\mathsf{ts}}(i)$,

(c) $\neg(i \xrightarrow{R} d)$.

Note that (b) takes place whenever (a) does. Let us assume (a). Since i is not completed, $G_{\mathsf{ts}}(i) = \top$ holds (by $\mathsf{INV}_{\mathsf{WF}}$). On the other hand, by the assumption of the lemma, start_ts contains a non-maximal timestamp. Under such conditions, (b) holds.

Let us obtain a contradiction for (b). Since κ satisfies the invariant INV, from $\mathsf{INV}_{\mathsf{ALG}}$(i) and $i \xrightarrow{R} i'$ we learn that $G_{\mathsf{ts}}(i) <_{\mathsf{TS}} G_{\mathsf{ts}}(i')$. Consequently:

$$s(\texttt{start\_ts}) <_{\mathsf{TS}} G_{\mathsf{ts}}(i) <_{\mathsf{TS}} G_{\mathsf{ts}}(i')$$

On the other hand, since $i' \in \mathsf{seen}(\kappa, d)$ holds, so does $\neg(s(\texttt{start\_ts}) <_{\mathsf{TS}} G_{\mathsf{ts}}(i'))$ by definition of seen. Thus, we arrived to a contradiction.

Let us obtain a contradiction for (c). Since $i' \in \mathsf{seen}(\kappa, d)$, $i' \xrightarrow{R} d$ holds. By Definition 3.1 of a history, R is a transitive relation. Thus, $i \xrightarrow{R} i'$ and $i' \xrightarrow{R} d$ together imply $i \xrightarrow{R} d$, which contradicts (c). □

Lemma 3.18. Let us take any interpretation of logical variables ℓ and a configuration $\kappa = (s, (E, R), G_{\mathsf{ts}}) \in [\![\text{isCand}]\!]_\ell$ such that $\texttt{CAND} = \texttt{enqOf}(E, G_{\mathsf{ts}}, \texttt{cand\_tid}, \texttt{cand\_ts})$ and $\texttt{CAND} \in \mathsf{inQ}(s(\texttt{pools}), E, G_{\mathsf{ts}})$ both hold. We need to prove that:

$$\forall e \in \mathsf{inQ}(s(\texttt{pools}), \texttt{E}, \texttt{G}_{\mathsf{ts}}).\, E(e).\mathsf{tid} \in \texttt{A} \implies \neg(e \xrightarrow{R} \texttt{CAND})$$

Let $\texttt{DEQ} = \texttt{myEid}()$ be the current dequeue event. By definition, the set $\mathsf{seen}(\kappa, \texttt{DEQ})$ is a subset of $\mathsf{inQ}(\texttt{pools}, E, G_{\mathsf{ts}})$. In other words, every enqueue with a value in the data structure is either seen by DEQ or not.

According to isCand, $\texttt{CAND} \in \mathsf{seen}(\kappa, \texttt{DEQ})$. By Lemma A.1, no unseen enqueue can precede CAND in the abstract history. Additionally, since CAND's timestamp is minimal among enqueues seen by DEQ, CAND is necessary R-minimal among them according to $\mathsf{INV}_{\mathsf{ALG}}$(i). □

A.1.4 Preservation of $\mathsf{INV}_{\mathsf{LIN}}$

Sequential queue specification. We first introduce auxiliary notation related to the sequential queue specification in order to simplify the further technical development.

We call L a sequential queue history, if $\mathsf{seq}(L)$ holds and each event in L is either an enqueue or a dequeue operation. For simplicity, we assume that dequeue operations always return values and never return Empty[1].

Following the definition of specification histories in Section 3.4, given $L \in \mathcal{H}_{\mathsf{queue}}$, we consider the state σ of the sequential queue such that the following holds:

$$\langle(\varepsilon, (\emptyset, \emptyset)\rangle \twoheadrightarrow^*_{\mathsf{queue}} \langle\sigma, L\rangle\}$$

That is, σ is the state of the sequential queue after executing operations from L starting from the empty queue state ε. For convenience, we let $\mathsf{sq}\{L\}$ denote a sequence of enqueue operations that inserted each value in σ. In particular, when $\mathsf{sq}\{L\} = e \cdot \_$, the value $e.\mathsf{arg}$ is the front of the queue: $\sigma = e.\mathsf{arg} \cdot \_$.

When L is a sequential queue history, we also use a notation $L \cdot [i : (t, \mathsf{op}, a, r)]$ to denote a history resulting from appending the event $[i : (t, \mathsf{op}, a, r)]$ to the linear order of L.

The following definition establishes a mapping from dequeues to enqueues in a sequential queue history.

Definition A.2. *Given a sequential queue history L, we define $\mathsf{rf}\{L\}$ as a map from dequeue to enqueue events in L such that:*

$$\begin{aligned}
\mathsf{rf}\{L \cdot [_ : (_, \text{Enq}, _, _)]\} &= \mathsf{rf}\{L\} \\
\mathsf{rf}\{L \cdot [_ : (_, \text{Deq}, _, r)]\} &= \begin{cases} \mathsf{rf}\{L\}[d : e], & \textit{if } \mathsf{sq}\{L\} = e \cdot _ \textit{ and } e.\mathsf{arg} = r \\ \textit{undefined}, & \textit{otherwise} \end{cases}
\end{aligned}$$

[1] Instead of being a part of the queue specification, this assumption can be proven formally as an invariant of the TS Queue and HW Queue.

It is easy to see that $\mathsf{rf}\{L\}$ is always well-defined for each $L \in \mathcal{H}_{\mathsf{queue}}$ and that the FIFO properties hold of $\mathsf{rf}\{L\}$.

Proposition A.3. *For $L \in \mathcal{H}_{\mathsf{queue}}$, the map $\mathsf{rf}\{L\}$ satisfies the following:*

1. *$\mathsf{rf}\{L\}$ is well-defined;*
2. *$\mathsf{rf}\{L\}$ correctly maps dequeue events to enqueue events:*
   - *enqueues are uniquely matched with dequeues in rf (rf is injective):*
   $$\forall d, d' \in \mathsf{dom}(\mathsf{rf}\{L\}).\, \mathsf{rf}\{L\}(d) = \mathsf{rf}\{L\}(d') \implies d = d'$$
   - *values inserted by enqueues equal to values removed by matching dequeues:*
   $$\forall d \in \mathsf{dom}(\mathsf{rf}\{L\}).\, E(d).\mathsf{rval} = E(\mathsf{rf}\{L\}(d)).\mathsf{arg}$$
3. *every successful dequeue event is matched in $\mathsf{rf}\{L\}$ with an enqueue according to the FIFO policy:*

$$\forall d.\, E(d).\mathsf{op} = \mathrm{Deq} \wedge E(d).\mathsf{rval} \neq \mathrm{NULL} \implies \mathsf{rf}\{L\}(d) \xrightarrow{L} d \wedge$$
$$(\forall e'.\, E(e').\mathsf{op} = \mathrm{Enq} \wedge e' \xrightarrow{L} \mathsf{rf}\{L\}(d) \implies \exists d'.\, \mathsf{rf}\{L\}(d') = e \wedge d' \xrightarrow{L} d)$$

Proposition A.4. *Given $L \in \mathcal{H}_{\mathsf{queue}}$ and any enqueue event $e \in L$, $\mathsf{sq}\{L\}$ contains e if and only if there is no dequeue event $d \in L$ such that $\mathsf{rf}\{L\}(d) = e$.*

Finally, we define the relation $\mathsf{same\_data}$ (previously informally introduced in Section 3.6 in the definition of $\mathsf{INV}_{\mathsf{LIN}}$).

Definition A.5 ($\mathsf{same\_data}$)**.** *Given a configuration $\kappa = (s, H, G)$ and $L \in \mathcal{H}_{\mathsf{queue}}$, we say that $\mathsf{same\_data}((s, H, G), L)$ holds if any enqueue e occurs in $\mathsf{sq}\{L\}$ if and only if it occurs in $\mathsf{inQ}(s, H, G)$.*

Proving the preservation of $\mathsf{INV}_{\mathsf{LIN}}$. We now prove preservation of the invariant $\mathsf{INV}_{\mathsf{LIN}}$ by the commitment points. In the following proofs, we only consider two commitment points: `setTS` corresponding to the atomic block at lines 10-14 and `remove` corresponding to the atomic block at lines 45-52. Since we only need to consider commitment points adding new *completed* events to histories, we ignore the commitment point at lines 3-6. We also ignore the commitment point at lines 30-34, because it only refines the order in the history and cannot invalidate the sequential queue specification.

Lemma A.6. *When INV holds, the commitment point* `setTS` *at lines 10-14 preserves $\mathsf{INV}_{\mathsf{LIN}}$:*

$$\forall \ell, \kappa, \kappa'.\, \kappa \in [\![\mathsf{INV}]\!]_\ell \wedge \kappa' \in [\![\mathtt{setTS}]\!]_t(\kappa) \implies \kappa' \in [\![\mathsf{INV}_{\mathsf{LIN}}]\!]_\ell$$

Proof. Let us choose any interpretation of logical variables ℓ and let s, H and G be such that $(s, H, G) = \kappa \in [\![\mathsf{INV}]\!]_\ell$. Let $\kappa' = (s', H', G')$ be the results of the commitment point `setTS`. We prove that $(s', H', G') \in [\![\mathsf{INV}_{\mathsf{LIN}}]\!]_\ell$ holds.

Consider any linearization L' of H' (meaning that $\mathsf{seq}(L')$ and $\lfloor H' \rfloor \sqsubseteq L'$ hold). We need to prove that $L' \in \mathcal{H}_{\mathsf{queue}}$. Let e be the enqueue event completed

at the commitment point. Then L' takes form of $L_1 \cdot e \cdot L_2$, where L_1 and L_2 are sequences of events such that $\mathsf{seq}(L_1 \cdot L_2)$ and $\lfloor H \rfloor \sqsubseteq L_1 \cdot L_2$ hold. Thus, $L_1 \cdot L_2 \in \mathcal{H}_{\mathsf{queue}}$, since H satisfies $\mathsf{INV}_{\mathsf{LIN}}$ according to the premise of the lemma.

It is easy to see that all dequeue events in L_1 return values corresponding to the queue specification, since L_1 is a prefix of a linearization $L_1 \cdot L_2$ of a history satisfying $\mathsf{INV}_{\mathsf{LIN}}$. Consequently, to conclude that $L_1 \cdot e \cdot L_2$ meets the queue specification, we only need to show that all dequeues in L_2 return correct values.

Let $d_1, d_2, \ldots, d_k$ be all of the dequeues in L_2. We consider $\mathsf{rf}\{L'\}$ and enqueue events $e_1 = \mathsf{rf}\{L'\}(d_1)$, $e_2 = \mathsf{rf}\{L'\}(d_2)$, ..., $e_k = \mathsf{rf}\{L'\}(d_k)$. Since $(s, H, G) \in [\![\mathsf{INV}]\!]_\ell$, $\mathsf{same\_data}((s, H, G), L_1 \cdot L_2)$ holds. By Proposition A.4, when e_i (for each $1 \leq i \leq k$) is read by a dequeue in $L_1 \cdot L_2$, it must be the case that $e_i \notin \mathsf{sq}\{L_1 \cdot L_2\}$. By Definition A.5 of $\mathsf{same\_data}((s, H, G), L_1 \cdot L_2)$, we conclude that $e_i \notin \mathsf{inQ}(\kappa)$ holds, and, because `setTS` does not alter those enqueue events, $e_i \notin \mathsf{inQ}(\kappa')$ holds too. Since invariants $\mathsf{INV}_{\mathsf{ALG}}$(iv) and $\mathsf{INV}_{\mathsf{ORD}}$(ii) hold of (s, H, G), we learn that $e \in \mathsf{inQ}(\kappa)$ and that e_i precedes e in H for each $1 \leq i \leq k$. The latter observation also holds of H', since it extends H monotonously, and of its linearization L'.

According to the FIFO properties stated in Proposition A.3(3), L_1 and L_2 take the following form:

$$L_1 = \_ \cdot e_1 \cdot \_ \cdot \ldots \cdot \_ \cdot e_k \cdot \_$$
$$L_2 = \_ \cdot d_1 \cdot \_ \cdot \ldots \cdot \_ \cdot d_k \cdot \_$$

Let $\{L^i \mid 1 \leq i \leq k\}$ be the prefixes of L_2 such that each L^i ends right before d_i. Since $L_1 \cdot L_2 \in \mathcal{H}_{\mathsf{queue}}$ is a correct sequential queue history, it must be the case that:

- $\mathsf{sq}\{L_1 \cdot L^k\} = e_k \cdot \_$,
- ...
- $\mathsf{sq}\{L_1 \cdot L^2\} = e_2 \cdot \ldots \cdot e_k \cdot \_$, and
- $\mathsf{sq}\{L_1 \cdot L^1\} = e_1 \cdot e_2 \cdot \ldots \cdot e_k \cdot \_$.

It is easy to see that $\mathsf{sq}\{L_1 \cdot e \cdot L^1\}$ also starts with $e_1 \cdot e_2 \cdot \ldots \cdot e_k$, meaning that each dequeue d_i (for $1 \leq i \leq k$) returns a correct return value in L'.

□

Lemma A.7. *When* INV, isCand *and* `cand_pid` $\neq$ NULL *all hold, the commitment point* `remove` *at lines 45-52 preserves* $\mathsf{INV}_{\mathsf{LIN}}$.

$$\forall \ell, \kappa, \kappa'.\, \kappa \in [\![\mathsf{INV} \wedge \text{isCand} \wedge \texttt{cand\_pid} \neq \text{NULL}]\!]_\ell \\ \wedge\, \kappa' \in [\![\texttt{remove}]\!]_t(\kappa) \implies \kappa' \in [\![\mathsf{INV}_{\mathsf{LIN}}]\!]_\ell$$

Proof. Let us choose any interpretation of logical variables ℓ and let s, H and G be such that (s, H, G) satisfy INV, isCand and `cand_pid` $\neq$ NULL. Let s', H' and G' be the results of the atomic step `remove`. We prove that $(s', H', G') \in [\![\mathsf{INV}_{\mathsf{LIN}}]\!]_\ell$.

In this proof, we consider the case of a successful removal (when removal fails, the atomic step does nothing, so $\mathsf{INV}_{\mathsf{LIN}}$ is trivially preserved). In the

following, we consider a dequeue event d that is completed at the commitment point and e is the enqueue event that is dequeued. Upon the successful removal, $(\texttt{cand\_pid}, \_, \texttt{cand\_ts})$ occur in the pool of a thread $\texttt{cand\_tid}$. Consequently, $e = \texttt{enqOf}(E, G_{\mathsf{ts}}, \texttt{cand\_tid}, \texttt{cand\_ts}) \in \mathsf{inQ}(\texttt{pools}, H, G)$.

Consider any linearization L' of H' (meaning that $\mathsf{seq}(L')$ and $\lfloor H' \rfloor \sqsubseteq L'$ hold). We need to prove that $L' \in \mathcal{H}_{\mathsf{queue}}$. Note that e precedes d in H', since this is how they get ordered at the commitment point. Hence, L' takes form of $L_1 \cdot e \cdot L_2 \cdot d \cdot L_3$, where L_1, L_2 and L_3 are sequences of events such that $\mathsf{seq}(L_1 \cdot e \cdot L_2 \cdot L_3)$ and $\lfloor H \rfloor \sqsubseteq L_1 \cdot e \cdot L_2 \cdot L_3$ hold. Thus, $L_1 \cdot e \cdot L_2 \cdot L_3 \in \mathcal{H}_{\mathsf{queue}}$, since H satisfies $\mathsf{INV}_{\mathsf{LIN}}$ according to the premise of the lemma.

It is easy to see that all dequeue events in $L_1 \cdot e \cdot L_2$ return values corresponding to the queue specification, since $L_1 \cdot e \cdot L_2$ is a prefix of a correct linearization $L_1 \cdot e \cdot L_2 \cdot L_3$. Consequently, to conclude that $L_1 \cdot e \cdot L_2 \cdot d \cdot L_3$ meets the queue specification, we need to show that all dequeues in $d \cdot L_3$ return correct values. Note that d is an uncompleted dequeue in H, so by $\mathsf{INV}_{\mathsf{ORD}}$ it is later in the order than all completed dequeues in H, which remains true for H'. Thus, dequeues in L' precede d, meaning that L_3 does not contain any dequeue event. Consequently, we only need to prove that $E(e).\mathsf{arg}$ is a correct return value for d according to the sequential specification.

Let us consider $L = L_1 \cdot e \cdot L_2 \cdot L_3$. Since $e \in \mathsf{inQ}(s, H, G)$ and H satisfies $\mathsf{INV}_{\mathsf{LIN}}$, e is in $\mathsf{sq}\{L\}$ according to $\mathsf{same\_data}$. Note that the commitment point $\texttt{remove}$ orders e in front of every enqueue belonging to $\mathsf{inQ}(s, H, G)$. Knowing from $\mathsf{same\_data}$ that $\mathsf{inQ}(s, H, G)$ and $\mathsf{sq}\{L\}$ consist of the same enqueue events, and that out of all of them, e occurs the first in L, we conclude that $\mathsf{sq}\{L\} = e \cdot \_$. Additionally, we make an observation that L_3 consists entirely of unread enqueues: the invariant $\mathsf{INV}_{\mathsf{ORD}}$(ii) states that all enqueues that are not in $\mathsf{inQ}(s, H, G)$ must precede e in H (and, therefore, in its linearization L too). This allows us to conclude that $\mathsf{sq}\{L_1 \cdot e \cdot L_2\} = e \cdot \_$ holds. Consequently, it is correct for d to return the value of e in $L_1 \cdot e \cdot L_2 \cdot d \cdot L_3$. □

A.1.5 Preservation of $\mathsf{INV}_{\mathsf{ALG}}$(i)

Showing that the invariant INV is preserved by all primitive commands is mostly straightforward, except for the command assigning a timestamp to an enqueued value at line 10. When the latter happens, it is necessary to prove that the property of the timestamps $\mathsf{INV}_{\mathsf{ALG}}$(i) is not invalidated. To show that this is indeed the case, one has to observe a certain property of timestamps generated by the function $\texttt{newTimestamp}$: a timestamp generated for the current enqueue event $\texttt{myEid}()$ and stored in a memory cell $\texttt{timestamp}$ is greater than timestamps of all enqueues that precede $\texttt{myEid}()$ in the abstract history and still have their values in the data structure. Specifically, we define an assertion newTS($\texttt{ts}$) denoting configurations $(s, (E, R), G)$ that satisfy the following:

$$\forall i \in \mathsf{inQ}(s(\texttt{pools}), E, G).\, i \xrightarrow{R} \texttt{myEid}() \implies G_{\mathsf{ts}}(i) <_{\mathsf{TS}} \texttt{ts}$$

It is easy to see that newTS asserts the same property as $\mathsf{INV}_{\mathsf{ALG}}$(i), but only for the current event and a timestamp generated for it. When at line 10 the timestamp gets assigned, newTS enables concluding that $\mathsf{INV}_{\mathsf{ALG}}$(i) is preserved.

We prove the following Hoare specification for the timestamp generation algorithm and outline the proof in Figure A.3:

```
int counter = 1;

TS newTimestamp() {
{ INV ∧ inserted(t, Enq) }
  int oldCounter = counter;
{ INV ∧ inserted(t, Enq) ∧ cntProp(oldCounter)
  ∧ oldCounter ≤ counter }
  TS result;
  if (CAS(counter, oldCounter, oldCounter+1))
    { INV ∧ inserted(t, Enq)
    ∧ cntProp(oldCounter) ∧ oldCounter < counter }
    result = (oldCounter, oldCounter);
    { INV ∧ inserted(t, Enq) ∧ newTS(result) }
  else
    { INV ∧ inserted(t, Enq)
    ∧ cntProp(oldCounter) ∧ oldCounter < counter }
    result = (oldCounter, counter-1);
    { INV ∧ inserted(t, Enq) ∧ newTS(result) }
  { INV ∧ inserted(t, Enq) ∧ newTS(result) }
  return result;
}
```

Figure A.3: Proof outline for the timestamp generating algorithm

```
{ INV ∧ started(t, Enq) }
TS timestamp := newTimestamp();
{ INV ∧ started(t, Enq) ∧ newTS }
```

The assertion newTS is obtained with the help of the following auxiliary assertion, which connects the generated timestamp to the real-time order using $\mathsf{INV}_{\mathsf{ALG}}$(iii):

$$[\![\mathsf{cntProp}(C)]\!]_\ell \triangleq \{(s,H,G) \mid \forall a,b.\, \forall i \in \mathsf{inQ}(s(\texttt{pools}),H,G).\, i \xrightarrow{R} \texttt{myEid}() \\ \wedge\, G_{\mathsf{ts}}(i) = (a,b) \implies b < C\}$$

It is easy to see that cntProp(counter) is implied by the invariant property $\mathsf{INV}_{\mathsf{ALG}}$(iii). Thus, after the first line of newTimestamp, cntProp(oldCounter) holds. Later on, when the timestamp result is formed, cntProp(counter) yields us the fact that result is a timestamp greater than timestamps of all enqueues that have a value in the pools and precede myEid(), which concludes the proof of newTS(ts).

A.1.6 Stability of the loop invariant

Proof of Lemma 3.19. Since $(\kappa,\kappa') \in R_t$, there exists a thread t' such that one of the following situations takes place:

- $(\kappa,\kappa') \in \dashrightarrow_{t'}$,
- $(\kappa,\kappa') \in G_{t',\texttt{local}}$, or
- there exists $\hat{\alpha}$ and P such that $G_{t',\hat{\alpha},P} \subseteq R_t$ and $(\kappa,\kappa') \in G_{t',\hat{\alpha},P}$.

In further, we prove the lemma separately for each $\hat{\alpha}$ and P. In each case, we assume that $\kappa = (s, (E, R), G_{\mathsf{ts}})$ and $\kappa' = (s', (E', R'), G'_{\mathsf{ts}})$.
Case #1: $(\kappa, \kappa') \in \dashrightarrow_{t'}$. This environment transition only adds a new event e in a thread t' and orders it after completed events. As a result of this environment transition, e is uncompleted in κ'. By Definition 3.1, $e \xrightarrow{R'} \texttt{DEQ}$ does not hold. Consequently, $e \notin \mathsf{seen}(\kappa', \texttt{DEQ})$. It is easy to see that all other enqueues outside of $\mathsf{seen}(\kappa, \texttt{DEQ})$ are not affected by this environment transition, so we can conclude that $\mathsf{seen}(\kappa', \texttt{DEQ}) \subseteq \mathsf{seen}(\kappa, \texttt{DEQ})$.

Cases #2 and #3: $\hat{\alpha}$ is either `insert` or `setTS`, and $P = \mathsf{INV} \wedge \text{started}(t', \text{Enq})$. These environment transitions only update the abstract history, concrete and ghost state associated with an event e, which is uncompleted in κ (that is, $E(e) = (t', \text{Enq}, v, \mathsf{todo})$). Since e is uncompleted, by Definition 3.1, $e \xrightarrow{R} \texttt{DEQ}$ does not hold. Neither of these environment transitions add any edges into the abstract history, meaning that $e \xrightarrow{R'} \texttt{DEQ}$ does not hold either. Consequently, $e \notin \mathsf{seen}(\kappa', \texttt{DEQ})$. It is easy to see that all other enqueues outside of $\mathsf{seen}(\kappa, \texttt{DEQ})$ are not affected by this environment transition, so we can conclude that $\mathsf{seen}(\kappa', \texttt{DEQ}) \subseteq \mathsf{seen}(\kappa, \texttt{DEQ})$.

Case #4: $\hat{\alpha} = \texttt{scan}$ and $P = \mathsf{INV} \wedge \text{started}(t', \text{Deq})$. This environment transition orders some of the enqueue events in front of an uncompleted dequeue d ($E(d) = (t', \text{Deq}, \_, \mathsf{todo})$). Let e be an enqueue event in κ such that $e \notin \mathsf{seen}(\kappa, \texttt{DEQ})$. Out of the reasons why e is not visible by $\texttt{DEQ}$ in κ, only $e \xrightarrow{R} \texttt{DEQ}$ may be affected by this environment transition, as it simply adds edges in the abstract history. However, we argue that an edge $e \xrightarrow{R'} \texttt{DEQ}$ is not added by `scan`. Indeed, d is uncompleted, so by Definition 3.1 it cannot precede any other event in the abstract history. Consequently, $e \xrightarrow{R'} \texttt{DEQ}$ is not added as implied by transitivity.

Case #5: $\hat{\alpha} = \texttt{remove}$ and $P = \mathsf{INV} \wedge \text{started}(t', \text{Deq})$. Let d be the uncompleted event by a thread t', i.e. such that $E(d) = (t', \text{Deq}, \_, \mathsf{todo})$. Let $e \in \mathsf{inQ}(s(\texttt{pools}), E, G_{\mathsf{ts}})$ be the enqueue event removed by this environment transition. As a result, $e \in \mathsf{inQ}(s'(\texttt{pools}), E', G'_{\mathsf{ts}})$ does not hold in κ', so $e \notin \mathsf{seen}(\kappa', \texttt{DEQ})$. It is easy to see that this environment transition affects other enqueue events only by ordering them w.r.t. other events. Consequently, if some $\texttt{ENQ} \notin \mathsf{seen}(\kappa, \texttt{DEQ})$, the only reason it may become visible in κ' is an addition of the edge $\texttt{ENQ} \xrightarrow{R'} \texttt{DEQ}$. However `remove` does not introduce such edge, and it is not implied by transitivity.

Case #6: $\hat{\alpha} = \texttt{genTS}$ and $P = \mathsf{INV}$. This transition does not affect any concrete state, ghost state or the abstract history associated with any enqueue event.

Case #7: $(\kappa, \kappa') \in G_{t', \texttt{local}}$. This transition does not affect any concrete state, ghost state or the abstract history associated with any enqueue event. □

A.2 Linearizability of the Optimistic Set

A.2.1 Overview of proof details

We prove the following specifications for the set operations:

$$R_t, G_t \vdash_t \{\mathsf{INV} \wedge \mathrm{started}(t, \mathsf{insert})\}\ \mathsf{insert}\ \{\mathsf{INV} \wedge \mathrm{ended}(t, \mathsf{insert})\}$$

$$R_t, G_t \vdash_t \{\mathsf{INV} \wedge \mathrm{started}(t, \mathsf{remove})\}\ \mathsf{remove}\ \{\mathsf{INV} \wedge \mathrm{ended}(t, \mathsf{remove})\}$$

$$R_t, G_t \vdash_t \{\mathsf{INV} \wedge \mathrm{started}(t, \mathsf{contains})\}\ \mathsf{contains}\ \{\mathsf{INV} \wedge \mathrm{ended}(t, \mathsf{contains})\}$$

For each thread t, we generate rely and guarantee relations analogously to §3.6. To this end, we let `insert`, `remove`, `contains` denote atomic steps corresponding to atomic blocks extended with ghost code in Figure 3.16 (at lines 16-30, 54-64 and 38-45 accordingly). For each thread t, relations G_t and R_t are then defined as follows:

$$\begin{array}{rcl} G_t & \triangleq & G_{t,\texttt{insert},\mathsf{INV}} \cup G_{t,\texttt{remove},\mathsf{INV}} \cup G_{t,\texttt{contains},\mathsf{INV}} \cup G_{t,\texttt{local}}, \\ R_t & \triangleq & \cup_{t' \in \mathsf{ThreadID} \setminus \{t\}} (G_{t'} \cup \dashrightarrow_{t'}) \end{array}$$

In the above, we assume a relation $G_{t,\texttt{local}}$, which describes arbitrary changes to certain program variables and no changes to the abstract history and the ghost state. That is, we say that the nodes of the linked list (such as `head` are *shared* program variables in the algorithm, and all others are `thread-local`, in the sense that every thread has its own copy of them. We let $G_{t,\texttt{local}}$ denote every possible change to *thread-local* variables of a thread t only.

In Figure A.4 we present the invariant INV. To formulate the invariant, we characterize all of the nodes in the data structure as either *reachable* or *unreachable*.

Definition A.8 (Reachable nodes)**.** *For a set of nodes of the data structure in a state s, we let $\rightsquigarrow_s \subseteq \mathsf{NodeID} \times \mathsf{NodeID}$ to be a* reachability *relation on the nodes and let $n \rightsquigarrow_s n'$ hold whenever there exists a sequence of node identifiers $n_0, n_1, \ldots, n_k$ ($k \geq 0$) such that $n_{i+1} = (*n_i).\texttt{next}$, $n_0 = n$, $n_k = n'$.*

Additionally, we define a function $\mathsf{remOf} : \mathcal{P}(\mathsf{Event}) \times \mathsf{NodeID} \rightharpoonup \mathsf{EventID}$ which maps a node identifier n to a matching remove event identifier i (if it exists).

$$\mathsf{remOf}(E, n) = \begin{cases} i, \text{if } G_{\mathsf{node}}(i) = n \wedge E(i).\mathsf{op} = \mathsf{remove} \\ \text{undefined otherwise} \end{cases}$$

We also assume that Event consists of well-typed queue events $[i : (t, \mathsf{op}, a, r)]$ meeting the following constraints:

- $\mathsf{op} \in \mathsf{Op} = \{\mathsf{insert}, \mathsf{remove}, \mathsf{contains}\}$,
- $a \in \mathsf{Val}$, and
- $r \in \{\mathsf{false}, \mathsf{true}\}$.

A.2.2 Loop invariant and preservation of $\mathsf{INV}_{\mathsf{LIN}}$

The key technical details for constructing abstract histories of the Optimistic Set are related to the preservation of the acyclicity of the abstract history and the invariant $\mathsf{INV}_{\mathsf{LIN}}$ at the commitment points.

Preservation of acyclicity of abstract histories by $\texttt{commit}_{\texttt{insert}}$ and $\texttt{commit}_{\texttt{remove}}$ is justified straightforwardly as follows. These two commitment points simply order the current event after all other completed events with the same argument and in front of all uncompleted events with the same argument. Knowing that the acyclicity of abstract histories and the invariant $\mathsf{INV}_{\mathsf{ORD}}$ hold prior to these commitment points, it is easy to see that the new edges cannot create a cycle.

Preservation of $\mathsf{INV}_{\mathsf{LIN}}$ by $\texttt{commit}_{\texttt{insert}}$ and $\texttt{commit}_{\texttt{remove}}$ is also straightforward to conclude. Note that the sequential specification of the set is only concerned with the order of events with the same argument. By construction, the order of completed events with the same argument is linear, as asserted by $\mathsf{INV}_{\mathsf{ORD}}$. The two commitment points extend abstract histories by appending insert and remove events to the linear order corresponding to their arguments. It is trivial to consider all possible linearizations of an abstract history constructed in this way and conclude that it satisfies the sequential specification.

In the rest of this section, we focus on discharging proof obligations induced by the commitment point $\texttt{commit}_{\texttt{contains}}$. As we argue in §3.7, in order to conclude that the return value of the `contains` operation matches the sequential specification of the set (and also that the resulting abstract history is acyclic), it is necessary to demonstrate the following two properties of the current `contains` event `myEid`() upon the commitment point:

- if $\texttt{curr.val} = E(\texttt{myEid}()).\mathsf{arg}$, then all successful removes occurring after $\mathsf{insOf}(E, \texttt{curr})$ are concurrent with `myEid`();
- if $\texttt{curr.val} > E(\texttt{myEid}()).\mathsf{arg}$, then all successful inserts occurring after $\mathsf{lastRemOf}(E, E(\texttt{myEid}()).\mathsf{arg})$ are concurrent with `myEid`();

To discharge both obligations, we build a loop invariant LI for the loop in the `locate` operation invoked by the `contains` operation in a thread t. For a given interpretation of logical variables ℓ, the loop invariant LI denotes triples $(s, (E, R), G_{\mathsf{node}}) \in [\![\mathsf{LI}]\!]_\ell$ such that the following conditions hold of the current node `curr`in a thread t and every node $n' \in \mathsf{NodeID}$:

- when n' is reachable from n and stores the value sought by the `contains` operation, it is either in the data structure or a matching remove operation is concurrent with the current one:

$$\texttt{curr} \rightsquigarrow_s n' \wedge n'.\mathsf{val} = \mathsf{last}(t, E, R).\mathsf{arg} \implies$$
$$n'.\mathsf{marked} = \mathsf{false} \vee \neg(\mathsf{remOf}(n') \xrightarrow{R} \mathsf{last}(t, E, R))$$

- when n' is not reachable from n and stores the value sought by the `contains` operation, it is either removed from the data structure or it has been inserted concurrently:

$$\texttt{curr} \not\rightsquigarrow_s n' \wedge n'.\mathsf{val} = \mathsf{last}(t, E, R).\mathsf{arg} \implies$$
$$n'.\mathsf{marked} = \mathsf{true} \vee \neg(\mathsf{insOf}(n') \xrightarrow{R} \mathsf{last}(t, E, R))$$

In the following two lemmas, we establish the two properties required in §3.7 with the help of the loop invariant LI.

Lemma A.9. *For every* $\ell : \mathsf{LVars} \to \mathsf{Val}$ *and configuration* $(s, (E, R), G_{\mathsf{ts}}) \in [\![\mathsf{LI}]\!]_\ell$*, if* $\mathtt{curr.val} = E(\mathtt{myEid}()).\mathsf{arg}$ *then:*

$$\neg\exists r.\, E(r) = (\_, \mathsf{remove}, E(\mathtt{myEid}()).\mathsf{arg}, \mathsf{true}) \wedge \mathsf{insOf}(E, \mathtt{curr}) \xrightarrow{R} r \xrightarrow{R} \mathtt{myEid}()$$

Proof. The node $\mathtt{curr}$ is reachable from itself, i.e., $\mathtt{curr} \rightsquigarrow_s \mathtt{curr}$ holds. Also, $\mathtt{curr.val} = E(\mathtt{myEid}()).\mathsf{arg}$. The first case of the loop invariant applies then, and the following holds:

$$\mathtt{curr.marked} = \mathsf{false} \vee \neg(\mathsf{remOf}(\mathtt{curr}) \xrightarrow{R} \mathtt{myEid}()) \tag{A.10}$$

Let us first consider the case when $\mathtt{curr.marked} = \mathsf{false}$. According to $\mathsf{INV}_{\mathsf{ALG}}(\mathrm{i})$, the following is true:

$$(E, \mathtt{curr}) \in \mathsf{dom}(\mathsf{insOf}) \wedge (E, \mathtt{curr}) \notin \mathsf{dom}(\mathsf{remOf}) \wedge$$
$$(\forall i.\, \mathsf{insOf}(E, n) \xrightarrow{R} i \wedge E(i).\mathsf{arg} = \mathtt{curr.val} \wedge E(i).\mathsf{rval} = \mathsf{true} \implies E(i).\mathsf{op} = \mathsf{contains})$$

meaning that no successful $\mathtt{remove}$ event of $E(\mathtt{myEid}()).\mathsf{arg}$ follows $\mathsf{insOf}(E, \mathtt{curr})$ in the abstract history. This allows us to conclude the statement of the lemma in the case of $\mathtt{curr.marked} = \mathsf{false}$.

Let us now consider the case when $\mathtt{curr.marked} = \mathsf{true}$. From (A.10), we know that $\neg(\mathsf{remOf}(\mathtt{curr}) \xrightarrow{R} \mathtt{myEid}())$ also holds then. Let us assume that there exists a remove event r contradicting the lemma:

$$E(r).\mathsf{op} = \mathsf{remove} \wedge E(r).\mathsf{arg} = E(\mathtt{myEid}()).\mathsf{arg} \wedge \mathsf{insOf}(E, \mathtt{curr}) \xrightarrow{R} r \xrightarrow{R} \mathtt{myEid}() \tag{A.11}$$

Note that $r \neq \mathsf{remOf}(E, \mathtt{curr})$, since that together with (A.11) immediately contradicts $\neg(\mathsf{remOf}(\mathtt{curr}) \xrightarrow{R} \mathtt{myEid}())$. By $\mathsf{INV}_{\mathsf{ALG}}(\mathrm{ii})$, r cannot occur between $\mathsf{insOf}(E, \mathtt{curr})$ and $\mathsf{remOf}(E, \mathtt{curr})$. Knowing from $\mathsf{INV}_{\mathsf{ORD}}(\mathrm{i})$ that completed events with the same argument are linearly ordered, we note that only the following can be the case:

$$\mathsf{insOf}(E, \mathtt{curr}) \xrightarrow{R} \mathsf{remOf}(E, \mathtt{curr}) \xrightarrow{R} r.$$

However, together with (A.11), that implies $\mathsf{remOf}(\mathtt{curr}) \xrightarrow{R} \mathtt{myEid}()$. We arrived to a contradiction. □

Lemma A.12. *For every* $\ell : \mathsf{LVars} \to \mathsf{Val}$ *and configuration* $(s, (E, R), G_{\mathsf{ts}}) \in [\![\mathsf{LI}]\!]_\ell$*, if* $\mathtt{curr.val} > E(\mathtt{myEid}()).\mathsf{arg}$ *then:*

$$\neg\exists i.\, E(i) = (\_, \mathsf{insert}, E(\mathtt{myEid}()).\mathsf{arg}, \mathsf{true}) \wedge \mathsf{lastRemOf}(E(\mathtt{myEid}()).\mathsf{arg}) \xrightarrow{R} i \xrightarrow{R} \mathtt{myEid}()$$

Proof. We use $a = E(\texttt{myEid}()).\mathsf{arg}$ as a shorthand for the argument of the current `contains` operations. The proof is by contradiction. Let us assume that there exists an event identifier i such that:

$$E(i) = (\_, \mathsf{insert}, a, \mathsf{true}) \wedge \mathsf{lastRemOf}(a) \xrightarrow{R} i \xrightarrow{R} \texttt{myEid}() \tag{A.13}$$

That is, there exists an `insert` event identifier i that is after the last `remove` of the value a, but before the current contains event. By $\mathsf{INV}_{\mathsf{WF}}$(ii), the node $G_{\mathsf{node}}(i)$ stores the argument of the insert, i.e., $G_{\mathsf{node}}(i).\texttt{val} = a$ holds. The assumption of the lemma can then be rewritten as $\texttt{curr.val} > G_{\mathsf{node}}(i).\texttt{val}$. According to the contrapositive of $\mathsf{INV}_{\mathsf{ALG}}$(iii), $\texttt{curr.val} > G_{\mathsf{node}}(i).\texttt{val}$ implies that $\texttt{curr} \not\rightsquigarrow_s G_{\mathsf{node}}(i)$.

Knowing that $\texttt{curr} \not\rightsquigarrow_s G_{\mathsf{node}}(i)$ and that $G_{\mathsf{node}}(i).\texttt{val} = a$, from the loop invariant we learn that:

$$G_{\mathsf{node}}(i).\texttt{marked} = \mathsf{true} \vee \neg(i \xrightarrow{R} \texttt{myEid}()) \tag{A.14}$$

In the rest of the proof, we do a case split on the two possible values of $G_{\mathsf{node}}(i).\texttt{marked}$. Let us first consider the case when $G_{\mathsf{node}}(i).\texttt{marked} = \mathsf{true}$. According to $\mathsf{INV}_{\mathsf{ALG}}$(ii), there exists $\mathsf{remOf}(E, G_{\mathsf{node}}(i))$. By $\mathsf{INV}_{\mathsf{ALG}}$(iv), $i \xrightarrow{R} \mathsf{remOf}(E, G_{\mathsf{node}}(i))$. Note that the event $\mathsf{lastRemOf}(a)$ is the last `remove` event of the value a, so $i \xrightarrow{R} \mathsf{lastRemOf}(a)$ must hold. However, that contradicts (A.13).

Let us now consider the case when $G_{\mathsf{node}}(i).\texttt{marked} = \mathsf{false}$. By (A.14), $\neg(i \xrightarrow{R} \texttt{myEid}())$ also holds then. However, that contradicts (A.13). □

($\mathsf{INV}_{\mathsf{LIN}}$) all linearizations of completed events of the abstract history satisfy the queue specification:

$$\mathsf{abs}(H, \mathcal{H}_{\mathsf{set}})$$

($\mathsf{INV}_{\mathsf{ORD}}$) properties of the partial order of the abstract history:

(i) completed events with the same argument are linearly ordered:

$$\forall i, j \in \mathsf{id}(\lfloor E \rfloor).\, E(i).\mathsf{arg} = E(j).\mathsf{arg} \implies i \xrightarrow{R} j \vee j \xrightarrow{R} i$$

(ii) no edges originate from uncompleted events:

$$\forall i \in \mathsf{id}(E \setminus \lfloor E \rfloor), j \in \mathsf{id}(E).\, \neg(i \xrightarrow{R} j)$$

($\mathsf{INV}_{\mathsf{ALG}}$) for every node $n \in \mathsf{dom}(s)$, the following holds:

(i) $n.\mathtt{marked} = \mathsf{false} \implies (E, n) \in \mathsf{dom}(\mathsf{insOf}) \wedge (E, n) \notin \mathsf{dom}(\mathsf{remOf})$
$\wedge\, (\forall i.\, \mathsf{insOf}(E, n) \xrightarrow{R} i \wedge E(i).\mathsf{arg} = n.\mathtt{val} \wedge E(i).\mathsf{rval} = \mathsf{true} \implies E(i).\mathsf{op} = \mathsf{contains})$

(ii) $n.\mathtt{marked} = \mathsf{true} \implies (E, n) \in \mathsf{dom}(\mathsf{insOf}) \wedge (E, n) \in \mathsf{dom}(\mathsf{remOf})$
$\wedge\, (\forall i.\, \mathsf{insOf}(E, n) \xrightarrow{R} i \xrightarrow{R} \mathsf{remOf}(E, n) \wedge E(i).\mathsf{rval} = \mathsf{true} \implies E(i).\mathsf{op} = \mathsf{contains})$

(iii) $\forall n' \in \mathsf{dom}(s).\, n \rightsquigarrow_s n' \implies n.\mathtt{val} \leq n'.\mathtt{val}$

(iv) $\forall n.\, (E, n) \in \mathsf{dom}(\mathsf{remOf}) \implies \mathsf{insOf}(E, n) \xrightarrow{R} \mathsf{remOf}(E, n)$

(v) $\mathtt{head} \rightsquigarrow_s n \iff n.\mathtt{marked} = \mathsf{false}$

(vi) $n \rightsquigarrow_s \mathtt{tail}$

(vii) $n.\mathtt{marked} = \mathsf{true} \vee n.\mathtt{marked} = \mathsf{false}$

($\mathsf{INV}_{\mathsf{WF}}$) properties of ghost state:

(i) $\forall e.\, E(e).\mathsf{rval} = \mathsf{true} \iff e \in \mathsf{dom}(G_{\mathsf{node}})$

(ii) $\forall e.\, E(e).\mathsf{arg} = G_{\mathsf{node}}(e).\mathtt{val}$

Figure A.4: The Optimistic Set: The invariant $\mathsf{INV} = \mathsf{INV}_{\mathsf{LIN}} \wedge \mathsf{INV}_{\mathsf{ORD}} \wedge \mathsf{INV}_{\mathsf{ALG}} \wedge \mathsf{INV}_{\mathsf{WF}}$

A.3 Linearizability of the Herlihy-Wing Queue

A.3.1 The algorithm

We now present the Herlihy-Wing queue [7] as our next running example. Values in the queue are stored in an infinite array, `Array`, with unbounded index `back` pointing to the first unoccupied cell of the array. Initially, each cell of the array is considered empty and contains NULL. Accordingly, initially $\texttt{back} = 0$.

An `enqueue` operation performs two steps. First, it acquires an index `k` with the help of atomic command `inc` returning the value of `back` and then incrementing it. At the second step, the `enqueue` operations stores its argument in $\texttt{Array}[\texttt{k}]$.

A `dequeue` operation obtains the length of the currently used part of the array and stores it in `n`. Then the operation iterates over array cells from the beginning till `n` and looks at the values in them. If a non-NULL value is encountered, the cells gets overwritten with NULL to remove the value from the queue, and the value itself is returned as a result of the `dequeue` operation. Alternatively, if all cells of `Array` appeared to store NULL during the loop, the algorithm restarts.

A.3.2 Concrete and auxiliary state

We assume that Event consists of well-typed queue events $[i : (t, \mathsf{op}, a, r)]$ meeting the following constraints:

- $\mathsf{op} \in \mathsf{Op} = \{\mathrm{Enq}, \mathrm{Deq}\}$,
- $\mathsf{op} = \mathrm{Deq} \iff a = \bot$, and
- $r = \bot \implies \mathsf{op} = \mathrm{Enq}$.

We consider a set of states $\mathsf{State} = \mathsf{Loc} \to \mathsf{Val}$, ranged over by s, where $\mathsf{Loc} = \{\texttt{back}\} \cup \{\texttt{Array}[i] \mid i \in \mathbb{N}\} \cup \{\texttt{k}[t] \mid t \in \mathsf{ThreadID}\} \cup ...$ is the set of all memory locations including the global `back` and infinite array $\texttt{Array}[]$, as well as thread-local variables (`k`, `n` etc).

We use a function $G_{\mathsf{slot}} : \mathsf{EventID} \rightharpoonup \mathsf{Slot}$ as ghost state in the proof in order to map event identifiers to slots in the infinite array. The map is established with the help of auxiliary code in the atomic block at line 7 in Figure A.5.

For given G_{slot} and E, every enqueue event $e \in \mathsf{id}(E)$ can be one of the following:

- $e \notin \mathsf{dom}(G_{\mathsf{slot}})$ — the slot is not assigned to the enqueue yet,
- $e \in \mathsf{withSlot}(s, E, G_{\mathsf{slot}})$ — the enqueue has a slot, but has not written a value into it yet:

$$\mathsf{withSlot}(s, E, G_{\mathsf{slot}}) \triangleq \{e \mid e \in \mathsf{id}(E \setminus \lfloor E \rfloor) \wedge E(e).\mathsf{op} = \mathrm{Enq} \\ \wedge\, s(\texttt{Array}[G_{\mathsf{slot}}(e)]) = \mathrm{NULL}\}$$

- $e \in \mathsf{untaken}(s, E, G_{\mathsf{slot}})$ — the slot has writen a value into its slot:

$$\mathsf{untaken}(s, E, G_{\mathsf{slot}}) \triangleq \{e \mid e \in \lfloor E \rfloor \wedge E(e).\mathsf{op} = \mathrm{Enq} \\ \wedge\, s(\texttt{Array}[G_{\mathsf{slot}}(e)]) = E(e).\mathsf{arg}\}$$

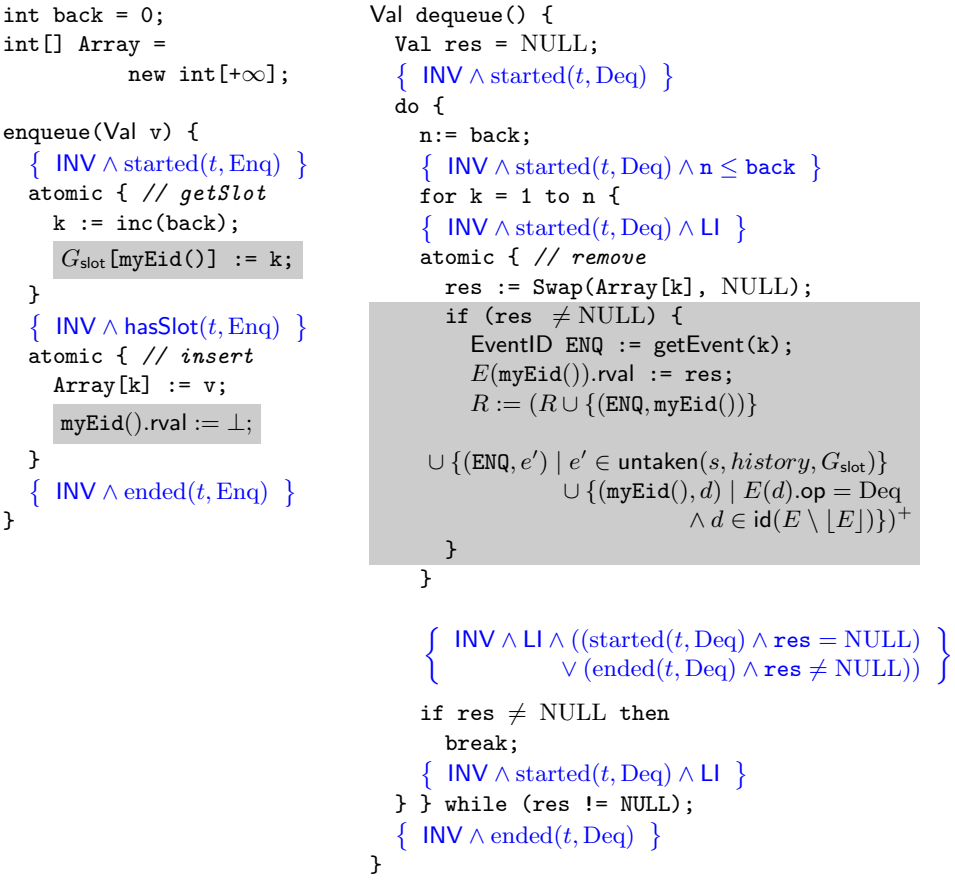


Figure A.5: The Herlihy-Wing queue

- $e \in \mathsf{taken}(s, E, G_{\mathsf{slot}})$ – the value written into the slot by the enqueue has been successfully taken by some dequeue event:

$$\mathsf{taken}(s, E, G_{\mathsf{slot}}) \triangleq \{e \mid e \in \lfloor E \rfloor \wedge E(e).\mathsf{op} = \mathrm{Enq} \\ \wedge\, s(\mathtt{Array}[G_{\mathsf{slot}}(e)]) = \mathrm{NULL}\}$$

A.3.3 Commitment points

To explain the construction of abstract histories for the Herlihy-Wing queue, we instrument the code in Figure A.5 with auxiliary code. When an operation starts, we automatically add a new uncompleted event into the set of events E to represent this operation and order it in R after all completed events. Aside from that, the `enqueue` operation has two more commitment points. For the first, the auxiliary code in the atomic block at line 7 maintains the ghost state G_{slot}. For the second, the auxiliary code at line 12 completes the enqueue event.

Upon a `dequeue`'s start, we similarly add an event representing it, and then the operation does one of the two commitment points. At line 22, the current `dequeue` operation encounters a non-NULL value in a slot `Array`[`k`], in which case it returns this value and removes it from the array. The auxiliary code

accompanying this change to the state completes the `dequeue` event and also adds three following kinds of edges to R and then transitively closes it:

1. $(\texttt{ENQ}, \texttt{myEid}())$, ensuring that in all linearizations of the abstract history, the current dequeue returns a value that has been already inserted by $\texttt{ENQ} = \texttt{enqOf}(E, G, \texttt{k})$.
2. $(\texttt{ENQ}, e)$ for each identifier e of an enqueue event whose value is still in the pools. This ensures that the `dequeue` removes the oldest value in the queue.
3. $(\texttt{myEid}(), d)$ for each identifier d of an uncompleted dequeue event. This ensures that dequeues occur in the same order as they remove values from the queue.

A.3.4 The overview of proof details

In Figure A.5, we provide the proof outlines for the enqueue and dequeue operations, in which we prove the following specifications:

$$R_t, G_t \vdash_t \{\mathsf{INV} \wedge \mathrm{started}(t, \mathrm{Deq})\}\ \mathrm{Deq}\ \{\mathsf{INV} \wedge \mathrm{ended}(t, \mathrm{Deq})\}$$

$$R_t, G_t \vdash_t \{\mathsf{INV} \wedge \mathrm{started}(t, \mathrm{Enq})\}\ \mathrm{Enq}\ \{\mathsf{INV} \wedge \mathrm{ended}(t, \mathrm{Enq})\}$$

In the proof outlines, we use an auxiliary assertion describing an enqueue event that has obtained a slot in the array, but has not written into it yet.

$$[\![\mathsf{hasSlot}(t, \mathsf{op})]\!]_\ell = \{(s, H, G_{\mathsf{slot}}) \mid E(\mathsf{last}(t, H)) = (t, \mathsf{op}, s(\texttt{arg}[t]), \mathsf{todo}) \\ \wedge\ \mathsf{last}(t, H) \in \mathsf{dom}(G)\};$$

For each thread t, we generate rely and guarantee relations analogously to §3.6. To this end, we let `getSlot`, `insert` and `remove` denote atomic steps corresponding to atomic blocks extended with ghost code in Figure A.5 (at lines 7, 12 and 22) accordingly). For each thread t, relations G_t and R_t are then defined as follows:

$$\begin{aligned} G_t &\triangleq G_{t,\texttt{getSlot},\mathsf{INV}\wedge\mathrm{started}(t,\mathrm{Enq})} \cup G_{t,\texttt{insert},\mathsf{INV}\wedge\mathsf{hasSlot}(t,\mathrm{Enq})} \cup G_{t,\texttt{remove},\mathsf{INV}} \\ &\qquad \cup G_{t,\texttt{local}}, \\ R_t &\triangleq \cup_{t' \in \mathsf{ThreadID}\setminus\{t\}} (G_{t'} \cup \dashrightarrow_{t'}) \end{aligned}$$

In the above, we assume a relation $G_{t,\texttt{local}}$, which describes arbitrary changes to certain program variables and no changes to the abstract history and the ghost state. That is, we say that `back` and `Array` are *shared* program variables in the algorithm, and all others are `thread-local`, in the sense that every thread has its own copy of them. We let $G_{t,\texttt{local}}$ denote every possible change to *thread-local* variables of a thread t only.

In Figure A.6 we present the invariant INV. It consists of several properties:

- $\mathsf{INV}_{\mathsf{LIN}}$ – the main correctness property;
- $\mathsf{INV}_{\mathsf{ORD}}$ – properties of uncompleted events that hold by construction of the partial order;

- $\mathsf{INV}_{\mathsf{ALG}}$ – a property of the array slots;
- $\mathsf{INV}_{\mathsf{WF}}$ – well-formedness of ghost state.

We introduce the following auxiliary predicate that we use in the invariant $\mathsf{INV}_{\mathsf{LIN}}$:

Definition A.15 (same_data)**.** *Given a configuration* $\kappa = (s, H, G_{\mathsf{slot}})$ *and* $L \in \mathcal{H}_{\mathsf{queue}}$*, we say that* $\mathsf{same\_data}((s, H, G_{\mathsf{slot}}), L)$ *holds if any enqueue* e *occurs in* $\mathsf{sq}\{L\}$ *if and only if it occurs in* $\mathsf{untaken}(s, H, G)$*.*

A.3.5 Loop invariant

We define a loop invariant LI, which we use to ensure that the uncompleted dequeue of a thread t returns a correct return value (the value inserted by the R-minimal enqueue).

Definition A.16. *Given interpretation of logical variables* ℓ*, we let* LI *be an assertion denoting the set of configurations* $[\![\mathsf{LI}]\!]_\ell$ *such that every configuration* $(s, (E, R), G_{\mathsf{slot}})$ *in it satisfies the following:*

1. $\forall e, e' \in \mathsf{untaken}(E, G).\, G_{\mathsf{slot}}(e) < \mathtt{k} \leq G_{\mathsf{slot}}(e') \leq \mathtt{n} \implies \neg(e \xrightarrow{R} e')$;
2. $\forall e \in \mathsf{untaken}(E, G).\, G_{\mathsf{slot}}(e) < s(\mathtt{k}) \implies \neg(e \xrightarrow{R} \mathtt{myEid}())$;
3. $s(\mathtt{n}) \leq s(\mathtt{back})$.

The loop invariant LI consists of three properties, which are formulated w.r.t. the thread-local memory cells `k` (contains the current loop index), `n` (contains the loop boundary), `back` and events of the abstract history. The first property states that an enqueue event e of a value in each slot preceding the current ($G_{\mathsf{slot}}(e) < k$) does not precede in R an enqueue event e' of a value from the subsequent part of the array. The second property similarly requires that an enqueue event e of a value in each slot that has already been visited ($G_{\mathsf{slot}}(e) < s(k)$) does not precede the current dequeue event `myEid()`. Finally, the third property simply asserts that the value in `n` is smaller than `back`.

The following lemma justifies the history update by the atomic step `remove`.

Lemma A.17. *For every* $\ell : \mathsf{LVars} \to \mathsf{Val}$ *and configuration* $(s, (E, R), G_{\mathsf{ts}}) \in [\![\mathsf{LI}]\!]_\ell$*, if* $s(\mathtt{Array}[\mathtt{k}]) \neq \mathrm{NULL}$*, then* $\mathtt{ENQ} = \mathtt{enqOf}(E, G, \mathtt{k})$ *is minimal among untaken enqueue events:*

$$\forall e \in \mathsf{untaken}(s, H, G).\, \neg(e \xrightarrow{R} \mathtt{ENQ})$$

Proof. The statement of the lemma follows from the first property of the loop invariant, $\mathsf{INV}_{\mathsf{ALG}}$ and $\mathsf{INV}_{\mathsf{WF}}$. According to the latter, every untaken enqueue e has a value in a slot $G_{\mathsf{slot}}(e) < s(\mathtt{back})$.

When $\mathtt{Array}[\mathtt{k}] \neq \mathrm{NULL}$, it is easy to see that all untaken enqueues with slots later than k in the array cannot precede `ENQ` according to $\mathsf{INV}_{\mathsf{ALG}}$, and the loop invariant asserts that all untaken enqueues before k in the array do not precede `ENQ` either. Thus, `ENQ` is a minimal untaken enqueue. □ □

With the help of Lemma A.17, we can conclude that the history update of the atomic step `remove` at line 22 in the dequeue operation does not invalidate acyclicity of the partial order. Let $\texttt{Array}[\texttt{k}] \neq \text{NULL}$ hold and let $\texttt{ENQ} = \texttt{enqOf}(E, G, \texttt{k})$ be an identifier of an enqueue event whose value is being removed. We consider separately each kind of edges added into the abstract history:

1. **The case of** $(\texttt{ENQ}, \texttt{myEid}())$. Note that prior to the commitment point, $\texttt{myEid}()$ is an uncompleted event. By Definition 3.1 of the abstract history, the partial order on its events is transitive, and uncompleted events do not precede other events. Thus, ordering `ENQ` before $\texttt{myEid}()$ does not create a cycle.

2. **The case of** $(\texttt{myEid}(), d)$ **for each identifier** d **of an uncompleted dequeue event**. Analogously to the previous case, if d is uncompleted event, it does not precede other events in the abstract history. Hence, ordering $\texttt{myEid}()$ in front of all such dequeue events does not create cycles.

3. **The case of** $(\texttt{ENQ}, e)$ **for each** $e \in \mathsf{untaken}(s, H, G)$. By Lemma A.17, from LI it follows that no $e \in \mathsf{untaken}(s, H, G)$ precedes `ENQ` in the abstract history. Consequently, ordering `ENQ` before all such enqueue events does not create cycles.

A.3.6 Preservation of $\mathsf{INV}_{\mathsf{LIN}}$

The proof of preservation of $\mathsf{INV}_{\mathsf{LIN}}$ is structured analogously to Appendix A.1.4. We reuse the notation for the sequential queue specification. In the proofs of preservation of the invariant $\mathsf{INV}_{\mathsf{LIN}}$ by the commitment points, we only consider two of them: `insert` corresponding to the atomic block at lines 12-15 and `remove` corresponding to the atomic block at at lines 22-31. That is because we only need to consider commitment points adding new *completed* events to histories.

Lemma A.18. *When* INV *holds, the commitment point* `insert` *at lines 12-15 preserves* $\mathsf{INV}_{\mathsf{LIN}}$*:*

$$\forall \ell, \kappa, \kappa'.\, \kappa \in [\![\mathsf{INV}]\!]_\ell \wedge \kappa' \in [\![\texttt{insert}]\!]_t(\kappa) \implies \kappa' \in [\![\mathsf{INV}_{\mathsf{LIN}}]\!]_\ell$$

Proof. Let us choose any interpretation of logical variables ℓ and let s, H and G be such that $(s, H, G_{\mathsf{slot}}) = \kappa \in [\![\mathsf{INV}]\!]_\ell$. Let $\kappa' = (s', H', G'_{\mathsf{slot}})$ be the results of the commitment point `setTS`. We prove that $(s', H', G'_{\mathsf{slot}}) \in [\![\mathsf{INV}_{\mathsf{LIN}}]\!]_\ell$ holds.

Consider any linearization L' of H' (meaning that $\mathsf{seq}(L')$ and $\lfloor H' \rfloor \sqsubseteq L'$ hold). We need to prove that $L' \in \mathcal{H}_{\mathsf{queue}}$. Let e be the enqueue event completed at the commitment point. Then L' takes form of $L_1 \cdot e \cdot L_2$, where L_1 and L_2 are sequences of events such that $\mathsf{seq}(L_1 \cdot L_2)$ and $\lfloor H \rfloor \sqsubseteq L_1 \cdot L_2$ hold. Thus, $L_1 \cdot L_2 \in \mathcal{H}_{\mathsf{queue}}$, since H satisfies $\mathsf{INV}_{\mathsf{LIN}}$ according to the premise of the lemma.

It is easy to see that all dequeue events in L_1 return values corresponding to the queue specification, since L_1 is a prefix of a linearization $L_1 \cdot L_2$ of a history satisfying $\mathsf{INV}_{\mathsf{LIN}}$. Consequently, to conclude that $L_1 \cdot e \cdot L_2$ meets the queue specification, we only need to show that all dequeues in L_2 return correct values.

Let $d_1, d_2, \ldots, d_k$ be all of the dequeues in L_2. We consider $\mathsf{rf}\{L'\}$ and enqueue events $e_1 = \mathsf{rf}\{L'\}(d_1)$, $e_2 = \mathsf{rf}\{L'\}(d_2)$, ..., $e_k = \mathsf{rf}\{L'\}(d_k)$. Since $(s, H, G_{\mathsf{slot}}) \in [\![\mathsf{INV}]\!]_\ell$, $\mathsf{same\_data}((s, H, G_{\mathsf{slot}}), L_1 \cdot L_2)$ holds. By Proposition A.4, when e_i (for each $1 \leq i \leq k$) is read by a dequeue in $L_1 \cdot L_2$, it must be the case that $e_i \notin \mathsf{sq}\{L_1 \cdot L_2\}$. By Definition A.15 of $\mathsf{same\_data}((s, H, G_{\mathsf{slot}}), L_1 \cdot L_2)$, we conclude that $e_i \notin \mathsf{untaken}(s, H, G)$ holds, and, because `insert` does not alter those enqueue events, $e_i \notin \mathsf{untaken}(s', H', G'_{\mathsf{slot}})$ holds too. Since invariants $\mathsf{INV}_{\mathsf{WF}}$(c) and $\mathsf{INV}_{\mathsf{ORD}}$(ii) hold of $(s', H', G'_{\mathsf{slot}})$, we learn that $e \in \mathsf{untaken}(s', H', G'_{\mathsf{slot}})$ and that e_i precedes e in H' for each $1 \leq i \leq k$. The latter observation also holds of the linearization L' of H'.

According to the FIFO properties stated in Proposition A.3(3), L_1 and L_2 take the following form:

$$L_1 = \_ \cdot e_1 \cdot \_ \cdot \ldots \cdot \_ \cdot e_k \cdot \_$$
$$L_2 = \_ \cdot d_1 \cdot \_ \cdot \ldots \cdot \_ \cdot d_k \cdot \_$$

Let $\{L^i \mid 1 \leq i \leq k\}$ be the prefixes of L_2 such that each L^i ends right before d_i. Since $L_1 \cdot L_2 \in \mathcal{H}_{\mathsf{queue}}$ is a correct sequential queue history, it must be the case that:

- $\mathsf{sq}\{L_1 \cdot L^k\} = e_k \cdot \_$,
- ...
- $\mathsf{sq}\{L_1 \cdot L^2\} = e_2 \cdot \ldots \cdot e_k \cdot \_$, and
- $\mathsf{sq}\{L_1 \cdot L^1\} = e_1 \cdot e_2 \cdot \ldots \cdot e_k \cdot \_$.

It is easy to see that $\mathsf{sq}\{L_1 \cdot e \cdot L^1\}$ also starts with $e_1 \cdot e_2 \cdot \ldots \cdot e_k$, meaning that each dequeue d_i (for $1 \leq i \leq k$) returns a correct return value in L'. □

Lemma A.19. *When* INV *and* LI *hold, the commitment point* `remove` *at lines 22-31 preserves* $\mathsf{INV}_{\mathsf{LIN}}$.

$$\forall \ell, \kappa, \kappa'. \kappa \in [\![\mathsf{INV} \wedge \mathsf{LI}]\!]_\ell \wedge \kappa' \in [\![\texttt{remove}]\!]_t(\kappa) \implies \kappa' \in [\![\mathsf{INV}_{\mathsf{LIN}}]\!]_\ell$$

Proof. Let us choose any interpretation of logical variables ℓ and let s, H and G be such that $(s, H, G_{\mathsf{slot}})$ satisfy INV, isCand and `cand_pid` $\neq$ NULL. Let s', H' and G'_{slot} be the results of the atomic step `remove`. We prove that $(s', H', G'_{\mathsf{slot}}) \in [\![\mathsf{INV}_{\mathsf{LIN}}]\!]_\ell$.

In this proof, we consider the case of a successful removal (when removal fails, the atomic step does nothing, so $\mathsf{INV}_{\mathsf{LIN}}$ is trivially preserved). In the following, we consider a dequeue event d that is completed at the commitment point and e is the enqueue event that is dequeued.

Consider any linearization L' of H' (meaning that $\mathsf{seq}(L')$ and $\lfloor H' \rfloor \sqsubseteq L'$ hold). We need to prove that $L' \in \mathcal{H}_{\mathsf{queue}}$. Note that e precedes d in H', since this is how they get ordered at the commitment point. Hence, L' takes form of $L_1 \cdot e \cdot L_2 \cdot d \cdot L_3$, where L_1, L_2 and L_3 are sequences of events such that $\mathsf{seq}(L_1 \cdot e \cdot L_2 \cdot L_3)$ and $\lfloor H \rfloor \sqsubseteq L_1 \cdot e \cdot L_2 \cdot L_3$ hold. Thus, $L_1 \cdot e \cdot L_2 \cdot L_3 \in \mathcal{H}_{\mathsf{queue}}$, since H satisfies $\mathsf{INV}_{\mathsf{LIN}}$ according to the premise of the lemma.

It is easy to see that all dequeue events in $L_1 \cdot e \cdot L_2$ return values corresponding to the queue specification, since $L_1 \cdot e \cdot L_2$ is a prefix of a correct linearization $L_1 \cdot e \cdot L_2 \cdot L_3$. Consequently, to conclude that $L_1 \cdot e \cdot L_2 \cdot d \cdot L_3$ meets the queue

specification, we need to show that all dequeues in $d \cdot L_3$ return correct values. Note that d is an uncompleted dequeue in H, so by $\mathsf{INV}_{\mathsf{ORD}}$(i) it is later in the order than all completed dequeues in H, which remains true for H'. Thus, dequeues in L' precede d, meaning that L_3 does not contain any dequeue event. Consequently, we only need to prove that $E(e).\mathsf{arg}$ is a correct return value for d according to the sequential specification.

Let us consider $L = L_1 \cdot e \cdot L_2 \cdot L_3$. Since $e \in \mathsf{untaken}(s, H, G_{\mathsf{slot}})$ and H satisfies $\mathsf{INV}_{\mathsf{LIN}}$, e is in $\mathsf{sq}\{L\}$ according to same_data. Note that the commitment point `remove` orders e in front of every enqueue belonging to $\mathsf{untaken}(s, H, G_{\mathsf{slot}})$. Knowing from same_data that $\mathsf{untaken}(s, H, G_{\mathsf{slot}})$ and $\mathsf{sq}\{L\}$ consist of the same enqueue events, and that out of all of them, e occurs the first in L, we conclude that $\mathsf{sq}\{L\} = e \cdot \_$. Additionally, we make an observation that L_3 consists entirely of unread enqueues: the invariant $\mathsf{INV}_{\mathsf{ORD}}$(ii) states that all enqueues that are not in $\mathsf{untaken}(s, H, G_{\mathsf{slot}})$ must precede e in H (and, therefore, in its linearization L too). This allows us to conclude that $\mathsf{sq}\{L_1 \cdot e \cdot L_2\} = e \cdot \_$ holds. Consequently, it is correct for d to return the value of e in $L_1 \cdot e \cdot L_2 \cdot d \cdot L_3$. □

($\mathsf{INV}_{\mathsf{LIN}}$) all linearizations of completed events of the abstract history satisfy the queue specification:

$$\mathsf{abs}(H, \mathcal{H}_{\mathsf{queue}})$$

($\mathsf{INV}_{\mathsf{ORD}}$) properties of the partial order of the abstract history:

(i) completed dequeues precede uncompleted ones:

$$\forall i \in \mathsf{id}(\lfloor E \rfloor).\, \forall j \in \mathsf{id}(E \setminus \lfloor E \rfloor).\, E(i).\mathsf{op} = E(j).\mathsf{op} = \mathrm{Deq} \implies i \xrightarrow{R} j$$

(ii) enqueues of already dequeued values precede enqueues of values in the array:

$$\forall i \in \mathsf{id}(\lfloor E \rfloor) \setminus \mathsf{untaken}(s, E, G_{\mathsf{slot}}).\, \forall j \in \mathsf{untaken}(s, E, G_{\mathsf{slot}}).\, i \xrightarrow{R} j$$

($\mathsf{INV}_{\mathsf{ALG}}$) the order on untaken enqueue events does not contradict the order in which they appear in the array:

$$\forall e_1, e_2 \in \mathsf{untaken}(s, E, G).\, e_1 \xrightarrow{R} e_2 \implies G_{\mathsf{slot}}(e_1) < G_{\mathsf{slot}}(e_2)$$

($\mathsf{INV}_{\mathsf{WF}}$) well-formedness properties of ghost state that enumerate all possible combinations of states, ghost states and events in a history:

(a) G_{slot} maps some of the events from E to slots preceding `back`:

$$\forall e \in \mathsf{dom}(G_{\mathsf{slot}}).\, e \in \mathsf{id}(E) \wedge G_{\mathsf{slot}}(e) < s(\mathtt{back})$$

(b) G_{slot} is injective;

(c) if $s(\mathtt{Array}[k]) \neq \mathrm{NULL}$ then k is a slot corresponding to an untaken completed enqueue event:

$$\forall k, v.\, s(\mathtt{Array})[k] \neq \mathrm{NULL} \iff \exists e \in \mathsf{untaken}(s, E, G_{\mathsf{slot}}).\, G_{\mathsf{slot}}(e) = k$$

(d) if $s(\mathtt{Array}[k]) = \mathrm{NULL}$ then k is a slot behind the `back` of the array, or it has not been used yet, or it is assigned to an uncompleted enqueue, or it has been inserted into and taken already:

$$\begin{aligned} \forall k.\, s(\mathtt{Array})[k] = \mathrm{NULL} \iff\ & s(\mathtt{back}) \leq k \vee k \notin \mathsf{dom}(G_{\mathsf{slot}}^{-1}) \vee \\ & (\exists e \in \mathsf{taken}(s, E, G_{\mathsf{slot}}).\, G_{\mathsf{slot}}(e) = k) \vee \\ & (\exists e \in \mathsf{withSlot}(E, G_{\mathsf{slot}}).\, G_{\mathsf{slot}}(e) = k) \end{aligned}$$

Figure A.6: The invariant $\mathsf{INV} = \mathsf{INV}_{\mathsf{LIN}} \wedge \mathsf{INV}_{\mathsf{ORD}} \wedge \mathsf{INV}_{\mathsf{WF}} \wedge \mathsf{INV}_{\mathsf{ALG}}$

```
1  Value clock, reg[NRegs], ver[NRegs];
2  Lock lock[NRegs];
3  Bool active[NThreads];
4  Set<Register> rset; // for each transaction
5  Map<Register, Value> wset; // for each transaction
6  Value rver; // for each transaction, initially ⊥
7  Value wver; // for each transaction, initially ⊤
8
9  function txbegin(Transaction T) {
10   active[threadOf(T)] := true;
11   rver[T] := clock;
12   return;
13 }
14
15 function read(Transaction T, Register x) {
16   if (wset[T].contains(x))
17     return wset[T].get(x);
18   ts1 := ver[x];
19   value := reg[x];
20   locked := lock[x].test();
21   ts2 := ver[x];
22   if (locked ∨ ts1 ≠ ts2 ∨ rver[T] < ts2)
23     return abort(T);
24   rset[T].put(x);
25   return value;
26 }
27
28 function write(Transaction T, Register x, Value v) {
29   wset[T].put(x, v);
30   return;
31 }
32
33 function fence() {
34   Bool r[NThreads]; // initially all false
35   foreach t in ThreadID
36     r[t] := active[t];
37   foreach t in ThreadID
38     if (r[t])
39       while (active[t]);
40   return;
41 }
```

Figure A.7: TL2 pseudocode (part 1). The code continues in Figure A.8.

```
41 function txcommit(Transaction T) {
42   Set<Lock> lset := ∅;
43   foreach x in wset[T] {
44     Bool locked := lock[x].trylock();
45     if (¬locked)
46       lset.add(x);
47     else {
48       foreach y in lset[T]
49         lock[y].unlock();
50       return abort(T);
51     }
52   }
53   wver[T] := fetch_and_increment(clock)+1;
54   foreach x in rset[T] {
55     Bool locked := lock[x].test();
56     atomic {
57         Value ts := ver[x];
58         pv[T][x] := ¬(locked ∨ rver[T] < ts);
59     }
60     if (locked ∨ rver[T] < ts) {
61       foreach y in lset[T]
62         lock[y].unlock();
63       return abort(T);
64     }
65   }
66   foreach (x, v) in wset[T] {
67     reg[x] := v;
68     ver[x] := wver[T];
69     lock[x].unlock();
70   }
71   return commit(T);
72 }
73
74 function abort(Transaction T) {
75     return aborted;
76     active[threadOf(T)] := false;
77 }
78
79 function commit(Transaction T) {
80     return committed;
81     active[threadOf(T)] := false;
82 }
```

Figure A.8: TL2 pseudocode (part 2).

A.4 Strong Opacity of TL2

In this section, we provide details for the proof of strong opacity of TL2. To this end, we first argue that histories of TL2 are well-formed, and in the rest of the section we discharge the proof obligations arising from graph characterization of TL2 histories by means of Lemma 4.20 and Theorem 4.22.

A.4.1 Preliminaries

In Figure A.7 we present the full pseudo-code of the TL2 software TM implementation. In the code, we use `NThreads` to denote the number of threads and `threadOf`(T) to denote the thread executing a given transaction T. We also let $\bot$ and $\top$ denote special minimal and maximal values, which `rver`$[T]$ and `wver`$[T]$ are initialized to (for every transaction T). The value $\bot$ is also used to describe a state of a lock as follows. We assume that the lock `lock`$[x]$ of each register x is either unlocked or stores an identifier of a transaction holding the lock, i.e., `Lock` $= \{\bot\} \uplus$ `Transaction`. Thus, when `lock`$[x] = \bot$, the lock is not acquired by any transaction, and when `lock`$[x] = T$, we know that T is holding a lock on x.

Functions `txbegin`, `txcommit`, `write`, `read` and `fence` of the pseudocode generate corresponding request and response actions from Figure 4.4 simultaneously with an invocation and return from the function accordingly. Additionally, we assume that each transaction T upon aborting or committing executes a handler `aborted`(T) or `committed`(T) accordingly, both of which simply unset the `active`$[t]$ flag (at line 76 or line 81) after appending an abort-response or a commit-response.

In the proof of strong opacity of TL2, we consider every execution of the most general client of TL2. At each step of it, we maintain a triple (s, H, G) consisting of a current state s, a current history H and its matching opacity graph $G \in \mathsf{Graph}(H)$. In addition to these three components, for every transaction T we also maintain a *ghost* variable `pv`$[T]$: `Register` $\to$ `Bool` that maps a register to `true` only if T post-validated a read from it. Ghost variables are different from usual variables in that they do not describe the concrete state of the execution, and are means to representing information about the past of executions in proofs. For simplicity of presentation, in the following we treat the `pv` variables as if they were a part of concrete state.

A.4.2 Well-formedness of TL2 histories

We argue that the histories of TL2 are well-formed. The most non-trivial well-formedness property of histories is that of the fences, i.e., we need to show that:

- Fence blocks until all active transactions complete:
  if $\tau = \tau_1\,(\_, t, \mathsf{txbegin})\,\tau_2\,(\_, t', \mathsf{fbegin})\,\tau_3\,(\_, t', \mathsf{fend})\,\tau_4$ then either τ_2 or τ_3 contains an action of the form $(\_, t, \mathsf{committed})$ or $(\_, t, \mathsf{aborted})$.

Let us consider the fence implementation at lines 33–39 of Figure A.7. The fence function consists of two loops: first, it iterates over all threads and records whether each thread t has an active transaction in a local variable `r`$[t]$; and then for each thread t, the fence waits until `active`$[t]$ becomes false, if `r`$[t]$ is `true`.

Let us consider execution of a fence in a thread t' and let A be the set of all active transactions at the beginning of the execution of the fence. For each transaction $T \in A$ in a thread t, there are two possibilities:

- The transaction T commits (aborts) before the fence checks $\texttt{active}[t]$ for the first time. Therefore, its commit-response $(\_, t, \mathsf{committed})$ (abort-response $(\_, t, \mathsf{aborted})$) occurs before $(\_, t', \mathsf{fend})$ in a history of the execution.

- The transaction T commits (aborts) after the fence checks $\texttt{active}[t]$ for the first time, meaning that $\texttt{active}[t] = \texttt{true}$ at that point. When that is the case, the fence sets $\texttt{r}[t]$ to $\texttt{true}$. Later on at line 39, the fence repeatedly re-reads the value of $\texttt{active}[t]$ until it observes $\texttt{false}$. Note that T will append a commit-response $(\_, t, \mathsf{committed})$ (abort-response $(\_, t, \mathsf{aborted})$) to the history before setting $\texttt{active}[t]$ to $\texttt{false}$. Therefore, by the time the fence reads $\texttt{false}$ from $\texttt{active}[t]$, the T's commit-response $(\_, t, \mathsf{committed})$ (abort-response $(\_, t, \mathsf{aborted})$) is already in the history, meaning that it occurs before $(\_, t', \mathsf{fend})$.

Based on these observations, we conclude that fences block until all active transactions complete, and that histories of TL2 are well-formed.

A.4.3 Opacity graph construction

The construction of the graph is inductive in the length of the execution of the most general client. At each step of the construction, we maintain a triple (s, H, G) consisting of the concrete state s, a history H and its opacity graph $G \in \mathsf{Graph}(H)$. We start from the initial state[2] empty history and an empty graph, and modify them as the execution proceeds. To this end, simultaneously with certain primitive commands of the TL2 algorithm in Figure A.7 we execute *graph updates* modifying the opacity graph.

In Figure A.9, we specify all the changes each graph update performs to an opacity graph $G = (N, \mathsf{vis}, \mathsf{HB}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$ of the triple (s, H, G). The updates make use of auxiliary predicates $\mathsf{reads}(n, x, v)$ and $\mathsf{writes}(n, x, v)$, which we further define:

- A predicate $\mathsf{reads}(n, x, v)$ holds of (s, H, G), if either $n \in N$ reads v from x non-transactionally or $n \in N$ is a transaction containing a non-local read of v from x.

- A predicate $\mathsf{writes}(n, x, v)$ holds of (s, H, G), if n is a node of the graph G such that either $n = ((\_, \_, \mathsf{write}(x, v)), (\_, \_, \mathsf{ret}(\bot))) \in \mathsf{nontxn}(H)$ holds, or $n \in \mathsf{txns}(H)$ and its last write to x is $((\_, \_, \mathsf{write}(x, v)), (\_, \_, \mathsf{ret}(\bot)))$.

We perform changes instructed by graph updates simultaneously with certain transitions of the TL2 algorithm, which we further specify for each update:

- $\mathsf{TXBEGIN}(T)$ occurs at the beginning of a transaction T;

[2] the state, in which every registers stores the initial value v_{init} and has a version 0; $\texttt{clock}$ stores 0; no lock is held and no transaction is active in any thread.

- $\mathsf{TXREAD}(T, x, v)$ occurs when a transaction T returns a value v at line 25, and a corresponding read response is appended to the current history;
- $\mathsf{TXVIS}(T)$ occurs simultaneously with the last transition of the loop at lines 54–63 in the transaction T;
- $\mathsf{NTXREAD}(\nu, x, v)$ occurs when a non-transactional read access ν reads a value v from a register x;
- $\mathsf{NTXWRITE}(\nu, x)$ occurs when a non-transactional write access ν writes a value to a register x.

A.4.4 Invariants

We prove strong opacity of TL2 by demonstrating that at each step of the construction of an opacity graph characterized by (s, H, G), where s is a concrete state, H is a history and G is its matching opacity graph, the triple satisfies a global invariant presented in Figure A.10. The invariant makes use of the following auxiliary definitions:

- Given a relation R on a set of graph nodes and a node n, we say that $\mathsf{isLastIn}(R, n)$ holds, if R is a linear order and n is the last node in R.
- We let $\mathsf{completed}(T)$ be a predicate that holds of (s, H, G), if $T \in \mathsf{txns}(H)$ is a committed or aborted transaction. We also introduce a predicate $\mathsf{aborted}(T)$ that holds of (s, H, G), when $T \in \mathsf{txns}(H)$ is an aborted transaction.

The most important invariant properties correspond to the proof obligations arising from Lemma 4.20 and Theorem 4.22. Recall that we need to demonstrate that for a history $H \in \mathcal{H}_{\mathbb{C}}|_{\mathsf{DRF}}$ and the graph $G \in \mathsf{Graph}(H)$ that we construct, the following holds:

1. H is consistent;
2. if $G = (N, \mathsf{vis}, \mathsf{HB}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$, then the relation $(\mathsf{HB} \;;\; (\mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}))$ is irreflexive;
3. G does not contain a cycle over transactions only with edges from $\mathsf{RT} \cup \mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}$.

The obligations 1–3 are discharged by proving invariants $\mathsf{INV}.2$, $\mathsf{INV}.3$ and $\mathsf{INV}.4$ accordingly. In the following, we explain each part of the invariant.

The invariant $\mathsf{INV}.1$ requires that histories be data-race free, since we only need to consider histories from $\mathcal{H}_{\mathbb{C}}|_{\mathsf{DRF}}$ in the proof. The invariant $\mathsf{INV}.2$(a) asserts consistency of TL2 histories, while $\mathsf{INV}.2$(b) is an auxiliary property relating a predicate $\mathsf{writes}(\_, \_, \_)$ with the content of write-sets of transactions. The invariant $\mathsf{INV}.3$ requires that $(\mathsf{HB} \;;\; (\mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}))$ be irreflexive. The invariant $\mathsf{INV}.4$ asserts that the opacity graph G does not contain cycles over transactions in $(\mathsf{RT} \cup \mathsf{txWR} \cup \mathsf{txWW} \cup \mathsf{txRW})$. By Theorem 4.22, together the latter two invariants imply that the graph G is acyclic.

The invariants mentioned so far are not inductive, so we strengthen them with additional auxiliary invariants. To this end, $\mathsf{INV}.5$ relates the order of

$\mathsf{TXBEGIN}(T)$:
  N := $N \cup \{T\}$;
  $\mathsf{vis}(T)$:= false;
  HB := $\mathsf{HB} \cup \{n \xrightarrow{\mathsf{HB}} T \mid \exists \psi \in n, \psi' \in T.\, \psi <_{\mathsf{hb}(H)} \psi'\}$;

$\mathsf{TXREAD}(T, x, v)$:
  if $(v = v_{\mathsf{init}})$:
    RW := $\mathsf{RW} \cup \{T \xrightarrow{\mathsf{RW}_x} n \mid \mathsf{vis}(n) \wedge \mathsf{writes}(n, x, \_)\}$
  else:
    WR := $\mathsf{WR} \cup \{n \xrightarrow{\mathsf{WR}_x} T \mid \mathsf{writes}(n, x, v)\}$;
    RW := $\mathsf{RW} \cup \{T \xrightarrow{\mathsf{RW}_x} n' \mid \mathsf{writes}(n, x, v) \wedge n \xrightarrow{\mathsf{WW}_x} n'\}$;
  HB := $\mathsf{HB} \cup \{n \xrightarrow{\mathsf{HB}} T \mid \exists T', n'.\, \mathsf{writes}(T', x, v)$
    $\wedge\, \mathtt{threadOf}(T') = \mathtt{threadOf}(n')$
    $\wedge\, n \xrightarrow{\mathsf{HB}}{}^{*} n' \xrightarrow{\mathsf{HB}} T'\}$;

$\mathsf{TXVIS}(T)$:
  $\mathsf{vis}(T)$:= true;
  foreach x in wset$[T]$:
    WW := $\mathsf{WW} \cup \{n \xrightarrow{\mathsf{WW}_x} T \mid n \neq T \wedge \mathsf{vis}(n)$
      $\wedge\, \mathsf{writes}(n, x, \_)\}$;
    RW := $\mathsf{RW} \cup \{n \xrightarrow{\mathsf{RW}_x} T \mid n \neq T \wedge \mathsf{reads}(n, x, \_)\}$;

$\mathsf{NTXREAD}(\nu, x, v)$:
  N := $N \cup \{\nu\}$;
  $\mathsf{vis}(\nu)$:= true;
  if $(v \neq v_{\mathsf{init}})$:
    WR := $\mathsf{WR} \cup \{n \xrightarrow{\mathsf{WR}_x} T \mid \mathsf{writes}(n, x, v)\}$;
  HB := $\mathsf{HB} \cup \{n \xrightarrow{\mathsf{HB}} T \mid \exists \psi \in n, \psi' \in \nu.\, \psi <_{\mathsf{hb}(H)} \psi'\}$;

$\mathsf{NTXWRITE}(\nu, x)$:
  N := $N \cup \{\nu\}$;
  $\mathsf{vis}(\nu)$:= true;
  WW := $\mathsf{WW} \cup \{n \xrightarrow{\mathsf{WW}_x} \nu \mid n \neq \nu \wedge \mathsf{vis}(n) \wedge \mathsf{writes}(n, x, \_)\}$;
  RW := $\mathsf{RW} \cup \{n \xrightarrow{\mathsf{RW}_x} \nu \mid n \neq \nu \wedge \mathsf{reads}(n, x, \_)\}$;
  HB := $\mathsf{HB} \cup \{n \xrightarrow{\mathsf{HB}} \nu \mid \exists \psi \in n, \psi' \in \nu.\, \psi <_{\mathsf{hb}(H)} \psi'\}$;

Figure A.9: Description of graph updates

INV denotes the smallest set of triples (s, H, G) of a concrete state s, a history H and an opacity graph $G = (N, \mathsf{vis}, \mathsf{HB}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}) \in \mathsf{Graph}(H)$, all satisfying the following:

1. The history H is data-race free.
2. Consistency invariants:
   (a) H is a consistent history.
   (b) The write-set of each transaction T consists of its most recent writes to registers:
   $$\forall T \in \mathsf{txns}(H), x \in \mathsf{Reg}, v \in \mathbb{Z}.\ (x, v) \in \mathtt{wset}[T] \iff \mathsf{writes}(T, x, v)$$
3. The relation $(\mathsf{HB}\ ;\ (\mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}))$ is irreflexive.
4. The relation $(\mathsf{RT} \cup \mathsf{txWR} \cup \mathsf{txWW} \cup \mathsf{txRW})$ is acyclic.
5. Read and write timestamps of transactions have the following properties:
   (a) $\forall T, T' \in \mathsf{txns}(H).\ T \xrightarrow{\mathsf{RT}} T' \implies ((\mathsf{vis}(T) \implies \mathtt{wver}[T] \leq \mathtt{rver}[T']) \wedge (\neg\mathsf{vis}(T) \implies \mathtt{rver}[T] \leq \mathtt{rver}[T'])) \vee \mathtt{rver}[T'] = \bot$
   (b) $\forall T, T' \in \mathsf{txns}(H).\ T \xrightarrow{\mathsf{WR}} T' \implies \mathtt{wver}[T] \leq \mathtt{rver}[T']$
   (c) $\forall T, T' \in \mathsf{txns}(H).\ T \xrightarrow{\mathsf{WW}} T' \implies \mathtt{wver}[T] < \mathtt{wver}[T']$
   (d) $\forall T, T' \in \mathsf{txns}(H).\ T \xrightarrow{\mathsf{RW}} T' \implies \mathtt{rver}[T] < \mathtt{wver}[T']$
   (e) $\forall T, T' \in \mathsf{txns}(H).\ T \xrightarrow{\mathsf{RW}_x} T' \wedge \mathtt{pv}[T][x] = \mathtt{true} \implies \mathtt{wver}[T] < \mathtt{wver}[T']$
6. Commit-pending transaction T' has the following properties:
   (a) $\forall T, T' \in \mathsf{txns}(H), x \in \mathsf{Reg}.\ T \neq T' \wedge \mathsf{writes}(T, x) \wedge \mathsf{vis}(T) \wedge \mathtt{lock}[x] = T' \implies \mathtt{wver}[T] < \mathtt{wver}[T']$
   (b) $\forall T, T' \in \mathsf{txns}(H), x \in \mathsf{Reg}.\ T \neq T' \wedge \mathsf{reads}(T, x, \_) \wedge \mathtt{lock}[x] = T' \implies \mathtt{rver}[T] < \mathtt{wver}[T']$
   (c) $\forall T, T' \in \mathsf{txns}(H), x \in \mathsf{Reg}.\ T \neq T' \wedge \mathtt{pv}[T][x] = \mathtt{true} \wedge \mathtt{lock}[x] = T' \implies \mathtt{wver}[T] < \mathtt{wver}[T']$
7. Well-formedness properties of read and write timestamps:
   (a) $\forall T \in \mathsf{txns}(H).\ \mathtt{rver}[T] < \mathtt{wver}[T]$
   (b) $\forall T \in \mathsf{txns}(H).\ \mathtt{rver}[T] \leq \mathtt{clock} \wedge (\mathtt{wver}[T] = \top \vee \mathtt{wver}[T] \leq \mathtt{clock})$
   (c) $\forall T \in \mathsf{txns}(H).\ \mathtt{wver}[T] \neq \top \implies \mathtt{rver}[T] \neq \bot$
   (d) $\forall T \in \mathsf{txns}(H).\ \mathsf{reads}(T, \_, \_) \implies \mathtt{rver}[T] \neq \bot)$
   (e) $\forall T \in \mathsf{txns}(H).\ (\exists x.\ \mathtt{pv}[T][x] = \mathtt{true}) \vee \mathsf{vis}(T) \implies \mathtt{wver}[T] \neq \top$

Figure A.10: The TL2 invariant (continues in Figure A.11)

8. Auxiliary invariants:

   (a) The value of each unlocked register x is ether the initial v_{init} or the value written by the last node in WW_x.

$$\begin{aligned}&\forall x \in \mathsf{Reg}.\, \mathtt{lock}[x] = \bot \implies \\ &(\mathtt{reg}[x] = v_{\mathsf{init}} \iff \neg\exists n.\, \mathsf{vis}(n) \wedge \mathsf{writes}(n, x, \_)) \wedge \\ &(\mathtt{reg}[x] \neq v_{\mathsf{init}} \iff \exists n.\, \mathsf{isLastIn}(\mathsf{WW}_x, n) \wedge \mathsf{writes}(n, x, \mathtt{reg}[x]))\end{aligned}$$

   (b) The version of each unlocked register x is either the initial version v_{init} or the write timestamp of the last transaction in txWW_x.

$$\begin{aligned}&\forall x \in \mathsf{Reg}.\, \mathtt{lock}[x] = \bot \implies \\ &(\mathtt{ver}[x] = v_{\mathsf{init}} \iff \neg\exists T \in \mathsf{txns}(H).\, \mathsf{vis}(T) \wedge \mathsf{writes}(T, x, \_)) \wedge \\ &(\mathtt{ver}[x] \neq v_{\mathsf{init}} \iff \exists T \in \mathsf{txns}(H).\, \mathsf{isLastIn}(\mathsf{txWW}_x, T) \wedge \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{ver}[x] = \mathtt{wver}[T])\end{aligned}$$

   (c) Visible transactions have their reads post-validated:

$$\forall T \in \mathsf{txns}(H), x \in \mathsf{Reg}.\, \mathsf{vis}(T) \wedge \mathsf{reads}(T, x, \_) \implies \mathtt{pv}[T][x] = \mathtt{true}$$

   (d) HB-edges do not originate from active transactions:

$$\forall T \in \mathsf{txns}(H).\, \neg\mathsf{completed}(T) \implies \neg\exists n'.\, T \xrightarrow{\mathsf{HB}} n'$$

   (e) A transaction holding a lock on a register is not completed and writes to that register and is not overwritten:

$$\begin{aligned}&\forall x \in \mathsf{Reg}, T \in \mathsf{txns}(H).\, \mathtt{lock}[x] = T \implies \\ &\qquad\qquad \neg\mathsf{completed}(T) \wedge \mathsf{writes}(T, x, \_) \wedge \neg\exists T.\, T \xrightarrow{\mathsf{WW}_x} T'\end{aligned}$$

Figure A.11: The TL2 invariant (continues Figure A.10)

timestamps to the edges of the opacity graph. In order to maintain this invariant inductively, we include an additional invariant INV.6, which helps to establish that INV.5(c–e) are preserved by the graph update TXVIS(_). To this end, it asserts the properties of every commit-pending transaction T' that has acquired a lock on a register, but has not added corresponding write and anti-dependencies yet.

The invariant INV.7 asserts various well-formedness properties of read and write timestamps of each transaction, and the invariant INV.8 asserts multiple well-formedness conditions relating the concrete state, history actions and graph edges.

Lemma A.20. *The invariant* INV *is preserved by the graph updates* $\mathsf{TXREAD}(T, x)$, $\mathsf{TXVIS}(T)$, $\mathsf{NTXREAD}(\nu)$, $\mathsf{NTXWRITE}(\nu)$ *and* $\mathsf{TXBEGIN}(T)$.

The invariant INV.1 does not require a proof of preservation. Indeed, in the Fundamental Property (Theorem 4.11) data-race freedom is an obligation that clients of a TM system need to fulfill, not a TM implementation. Hence, in the proof of strong opacity of TL2, INV.1 is an assumption. Also, it is easy to see that the well-formedness conditions of INV.7 and INV.8 are preserved trivially by construction of histories and graphs. We provide proof details for the remaining invariants in the following sections:

Section #	**Invariant**
Section A.4.6	INV.2
Section A.4.7	INV.3
Section A.4.8	INV.5
Section A.4.9	INV.4
Section A.4.10	INV.6

Note that we are considering the invariants out of the ascending order due to Section A.4.9 depending on Section A.4.8.

A.4.5 Timestamp order properties

We state Proposition A.21, which relates paths between transactions with timestamp order, and then we use it in multiple proofs of preservation of the invariant.

Proposition A.21. *If* (s, H, G) *satisfies* INV.*5,* INV.*7 and* INV.*8, then for every transactions* $T, T' \in \mathsf{txns}(H)$, *if* $T \xrightarrow{\mathsf{RT} \cup \mathsf{txWR} \cup \mathsf{txWW} \cup \mathsf{txRW}}{}^{+} T'$, *then either of the following holds:*

1. $\mathsf{vis}(T) \wedge \mathsf{vis}(T') \implies \mathtt{wver}[T] < \mathtt{wver}[T']$;
2. $\neg\mathsf{vis}(T) \wedge \mathsf{vis}(T') \implies \mathtt{rver}[T] < \mathtt{wver}[T']$;
3. $\mathsf{vis}(T) \wedge \neg\mathsf{vis}(T') \implies \mathtt{wver}[T] \leq \mathtt{rver}[T'] \vee \mathtt{rver}[T'] = \bot$;
4. $\neg\mathsf{vis}(T) \wedge \neg\mathsf{vis}(T') \implies \mathtt{rver}[T] \leq \mathtt{rver}[T'] \vee \mathtt{rver}[T'] = \bot$.

Proof of Proposition A.21. The proof is by induction on the length of the path $T \xrightarrow{\mathsf{RT} \cup \mathsf{txWR} \cup \mathsf{txWW} \cup \mathsf{txRW}}{}^{+} T'$. Let $\phi_1(T, T')$, $\phi_2(T, T')$, $\phi_3(T, T')$ and $\phi_4(T, T')$ be predicates corresponding to the implications of the proposition. For each $k \geq 1$, we prove $\Phi(k)$, which is defined as follows:

$$\Phi(k) \triangleq \forall T, T' \in \mathsf{txns}(H).\, T \xrightarrow{\mathsf{RT} \cup \mathsf{txWR} \cup \mathsf{txWW} \cup \mathsf{txRW}}{}^{k} T' \implies \phi_1(T, T') \wedge \phi_2(T, T') \wedge \phi_3(T, T') \wedge \phi_4(T, T')$$

Base of induction. We need to show that $\Phi(1)$ holds. Let us consider any two transactions T and T' in the opacity graph and assume that $T \xrightarrow{\mathsf{RT} \cup \mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}} T'$ holds. We consider each possible edge separately and demonstrate $\phi_1(T, T')$, $\phi_2(T, T')$, $\phi_3(T, T')$ or $\phi_4(T, T')$ depending on visibility of T and T'.

1. Consider the edge $T \xrightarrow{\mathsf{RT}} T'$. According to the invariant $\mathsf{INV}.5$(a), we consider make the following conclusions depending on visibility of T and T'.

   - Let us assume that both T and T' are visible. We demonstrate that $\phi_1(T, T')$ holds. From the invariant $\mathsf{INV}.5$(a), we know that either $\mathtt{wver}[T] \leq \mathtt{rver}[T']$ or $\mathtt{rver}[T'] = \bot$ is the case. Note that the contra-positive of the invariant $\mathsf{INV}.7$(c,e) states that $\mathtt{rver}[T'] = \bot$ contradicts visibility of T'. Therefore, only $\mathtt{wver}[T] \leq \mathtt{rver}[T']$ can hold in this case. By $\mathsf{INV}.7$(a), $\mathtt{rver}[T'] < \mathtt{wver}[T']$ holds, which allows us to conclude $\mathtt{wver}[T] \leq \mathtt{wver}[T']$ (coinciding with $\phi_1(T, T')$).
   - Let us assume that T is not visible and T' is. We demonstrate that $\phi_2(T, T')$ holds. From the invariant $\mathsf{INV}.5$(a), we know that either $\mathtt{rver}[T] \leq \mathtt{rver}[T']$ or $\mathtt{rver}[T'] = \bot$ is the case. Note that the contra-positive of the invariant $\mathsf{INV}.7$(c,e) states that $\mathtt{rver}[T'] = \bot$ contradicts visibility of T'. Therefore, only $\mathtt{rver}[T] \leq \mathtt{rver}[T']$ can hold in this case. By $\mathsf{INV}.7$(a), $\mathtt{rver}[T'] < \mathtt{wver}[T']$ holds, which allows us to conclude $\mathtt{rver}[T] < \mathtt{wver}[T']$ (coinciding with $\phi_2(T, T')$).
   - Let us assume that T is visible and T' is not. It is easy to see that $\phi_3(T, T')$ holds, since, by $\mathsf{INV}.5$(a), either $\mathtt{wver}[T] \leq \mathtt{rver}[T']$ or $\mathtt{rver}[T'] = \bot$ is true.
   - Let us assume that neither T nor T' is visible. It is easy to see that $\phi_4(T, T')$ holds, since, by $\mathsf{INV}.5$(a), either $\mathtt{rver}[T] \leq \mathtt{rver}[T']$ or $\mathtt{rver}[T'] = \bot$ is true.

2. $T \xrightarrow{\mathsf{WR}} T'$. By Definition 4.19, $\mathsf{vis}(T)$ holds. From invariant $\mathsf{INV}.5$(b), we obtain that $\mathtt{wver}[T] \leq \mathtt{rver}[T']$ holds. Hence, if $\mathsf{vis}(T')$ does not hold, we can conclude $\phi_3(T, T')$. Let us consider the case when $\mathsf{vis}(T')$ holds. By $\mathsf{INV}.7$(a), $\mathtt{rver}[T'] < \mathtt{wver}[T']$ holds. Therefore, $\mathtt{wver}[T] < \mathtt{wver}[T']$ holds, which allows us to conclude $\phi_1(T, T')$.

3. $T \xrightarrow{\mathsf{WW}} T'$. By Definition 4.19, $\mathsf{vis}(T)$ and $\mathsf{vis}(T')$ both hold. By invariant $\mathsf{INV}.5$(c), $\mathtt{wver}[T] \leq \mathtt{wver}[T']$ holds, which allows us to conclude $\phi_1(T, T')$.

4. $T \xrightarrow{\mathsf{RW}} T'$. By Definition 4.19, $\mathsf{vis}(T')$ holds. Let us first assume that $\mathsf{vis}(T)$ does not hold. By $\mathsf{INV}.5$(d), $\mathtt{rver}[T] < \mathtt{wver}[T']$ holds, which allows us to conclude $\phi_2(T, T')$. Let us now consider the case when $\mathsf{vis}(T)$ holds. By $\mathsf{INV}.8$(c), T post-validated all of its reads. Therefore, by $\mathsf{INV}.5$(e), $\mathtt{wver}[T] < \mathtt{wver}[T']$ holds, which allows us to conclude $\phi_1(T, T')$

Induction step. Let us consider any $k \geq 1$ and assume that $\Phi(k')$ holds for all $k' \leq k$, i.e.:

$$\forall T, T'.\, T \xrightarrow{\mathsf{RT} \cup \mathsf{txWR} \cup \mathsf{txWW} \cup \mathsf{txRW}}{}^{k'} T' \implies \phi_1(T, T') \wedge \phi_2(T, T') \tag{A.22}$$

We need to show that $\Phi(k+1)$ holds too. To this end, we consider any path $T \xrightarrow{\mathsf{RT} \cup \mathsf{txWR} \cup \mathsf{txWW} \cup \mathsf{txRW}}{}^{k+1} T'$.

Let T'' be the next transaction after T on the path, i.e., such that:

$$T \xrightarrow{\mathsf{RT} \cup \mathsf{txWR} \cup \mathsf{txWW} \cup \mathsf{txRW}} T'' \xrightarrow{\mathsf{RT} \cup \mathsf{txWR} \cup \mathsf{txWW} \cup \mathsf{txRW}}{}^{k} T'.$$

It is easy to see that $\Phi(k+1)$ can be obtained from the induction hypotheses $\Phi(1)$ (instantiated for T and T'') and $\Phi(k)$ (instantiated for T'' and T'). □

Anti-dependencies of read updates

We now formulate and establish a property of anti-dependency edges added at the graph update $\mathsf{TXREAD}(T, x, v)$, which we use in proofs of preservation of several invariants. To this end, we introduce an auxiliary predicate $\mathsf{PAD}(T, x, v)$ denoting a set of triples (s'', H'', G'') such that:

- in the history H'', T is a transaction whose last action is a read request $(\_, \_, \mathsf{read}(x))$;
- if $v = v_{\mathsf{init}}$ holds, then

$$\forall n'.\, \mathsf{writes}(n', x, \_) \wedge \neg \mathsf{aborted}(n') \implies (n' \in \mathsf{txns}(H)) \wedge \mathtt{rver}[T] < \mathtt{wver}[n']$$

- if $v \neq v_{\mathsf{init}}$ holds, then there exists a node n such that $\mathsf{vis}(n)$, $\mathsf{writes}(n, x, v)$ and the following all hold:

$$\forall n'.\, \mathsf{writes}(n', x, \_) \wedge \neg \mathsf{aborted}(n') \wedge \neg n' \xrightarrow{\mathsf{WW}_x} n \implies (n' \in \mathsf{txns}(H)) \wedge \mathtt{rver}[T] < \mathtt{wver}[n']$$

The following proposition asserts that $\mathsf{PAD}(T, x, v)$ is *stable under interference* from other threads, i.e., transitions and graph updates of other threads do not invalidate it.

Proposition A.23. *If $(s, H, G) \in \mathsf{PAD}(T, x, v)$ and (s', H', G') is a result of a graph update or a transition in a thread different from $\mathtt{threadOf}(T)$, then $(s', H', G') \in \mathsf{PAD}(T, x, v)$.*

Proof. Note that opacity graphs are constructed monotonously, i.e., once we add an edge, it stays in the graph. Hence, in a proof of stability of $\mathsf{PAD}(T, x, v)$ it is important to consider only graph updates $\mathsf{NTXWRITE}(n', x)$ and $\mathsf{TXVIS}(n')$, since they introduce new nodes writing to x, which $\mathsf{PAD}(T, x, v)$ requires to have a timestamp greater than $\mathtt{rver}[T]$. Out of all primitive commands by other threads, we only consider the ones at line 30 and at line 53 (the rest of primitive commands preserve $\mathsf{PAD}(T, x, v)$ trivially).

Firstly, we consider the graph update $\mathsf{NTXWRITE}(n', x)$, which adds a new non-transactional node n' writing to x. Since we only consider data-race free histories in our proof of TL2, either $n' \xrightarrow{\mathsf{HB}} T$ or $T \xrightarrow{\mathsf{HB}} n'$ must occur in G'. The former cannot be the case, since n' is a new node, and the graph update only adds edges ending in n'. Also, T is a live transaction, so by $\mathsf{INV}.8(\mathrm{d})$, $T \xrightarrow{\mathsf{HB}} n'$ cannot be in G'.

Secondly, we consider a primitive command at line 30, which happens when a transaction T' writes to the register x and results in adding a corresponding write response into the history. Hence, after this command, $\mathsf{writes}(T', x, \_)$ and $\neg\mathsf{aborted}(T')$ both hold. Note that when T reads a non-initial value written by a node n, $\neg(T' \xrightarrow{\mathsf{WW}_x} n)$ also holds, since T' is not visible. Additionally, T' is also live, meaning that it has not generated its write timestamp yet, i.e., $\mathtt{wver}[T'] = \top$ holds. Therefore, $\mathsf{PAD}(T, x, v)$ holds too.

Thirdly, we now consider the primitive command at line 53 occurring in some transaction T' writing to x. The primitive command replaces previously maximal write timestamp $\mathtt{wver}[T']$ with the incremented value of `clock`. By INV.7(b), all other read and write timestamps, such as $\mathtt{rver}[T]$, are less or equal to `clock` in (s, H, G). Hence, $\mathtt{rver}[T] < \mathtt{wver}[n']$ holds after the primitive command executes.

Lastly, we consider $\mathsf{TXVIS}(n')$ occurring in a transaction n' writing to x. Knowing that $(s, H, G) \in \mathsf{PAD}(T, x, v)$, we get that $\mathtt{rver}[T] < \mathtt{wver}[n']$ holds. Since the graph update simply makes n' visible, $\mathsf{PAD}(T, x, v)$ holds. □

We now prove the property of anti-dependencies added at the graph update $\mathsf{TXREAD}(T, x, v)$.

Proposition A.24. *If $(s, H, G) \in \mathsf{INV}$, $T \in \mathsf{txns}(H)$ and (s', H', G') is a result of a graph update $\mathsf{TXREAD}(T, x, v)$, then for every anti-dependency $T \xrightarrow{\mathsf{WR}_x} n$ added by the update, both $n \in \mathsf{txns}(H')$ and $\mathtt{rver}[T] < \mathtt{wver}[n]$ hold of (s', H', G').*

Proof. Before the graph update, the read operation stores the value of $\mathtt{ver}[x]$ into a local variable `ts1`, then it stores $\mathtt{reg}[x]$ in a local variable `value` as well as the lock status $\mathtt{lock}[x].\mathtt{test}()$ in `locked`. Afterwards, the read operation stores the value of $\mathtt{ver}[x]$ once again into a local variable `ts2`. The graph update $\mathsf{TXREAD}(T, x)$ happens provided that the following holds of (s, H, G) prior to the update:

$$\neg\mathtt{locked} \wedge \mathtt{ts1} = \mathtt{ts2} \wedge \mathtt{ts2} \leq \mathtt{rver}[T] \tag{A.25}$$

Note that by checking the above, the read operation ensures that there was a moment in between the two accesses to $\mathtt{ver}[x]$, when $\mathtt{ver}[x] = \mathtt{ts1} = \mathtt{ts2}$, $\mathtt{reg}[x] = \mathtt{value}$ and $\mathtt{lock}[x] = \bot$ all simultaneously held. Let (s'', H'', G'') correspond to that moment. Since that is the past of the execution, $(s'', H'', G'') \in \mathsf{INV}$ holds.

In the following, we demonstrate that $\mathsf{PAD}(T, x, v)$ holds of (s'', H'', G''). Once that is established, Proposition A.23 will ensure that $\mathsf{PAD}(T, x, v)$ is not invalidated till the read operation returns and the graph update executes.

Let us assume that T reads the initial value, i.e. $\mathtt{value} = v_{\mathsf{init}}$. Note that $\mathtt{reg}[x] = v_{\mathsf{init}}$ holds of (s'', H'', G''). By INV.8(a,b), there is no visible node in G'' that writes to x. Hence, $\mathsf{PAD}(T, x, v)$ holds trivially of (s'', H'', G''). Now let us consider the case when T does not read an initial value, i.e., $\mathtt{value} \neq v_{\mathsf{init}}$. Note that $\mathtt{reg}[x] = \mathtt{value} \neq v_{\mathsf{init}}$ holds of (s'', H'', G''). By INV.8(b), n is last in WW_x. Once again, $\mathsf{PAD}(T, x, v)$ holds trivially of (s'', H'', G''), since no node n' follows n in WW_x. □

Corollary A.26. *If $(s, H, G) \in \mathsf{INV}$, $T \in \mathsf{txns}(H)$ and (s', H', G') is a result of a graph update $\mathsf{TXREAD}(T, x, v)$, then for every anti-dependency $T \xrightarrow{\mathsf{WR}_x} n$*

added by the update, both $n \in \mathsf{txns}(H')$ *and* $\texttt{rver}[T] < \texttt{wver}[n]$ *hold of* (s', H', G').

Proof sketch. Note that the graph update $\mathsf{TXREAD}(T, x, v)$ adds the following anti-dependency edges into G:

- if $v = v_{\mathsf{init}}$, then $\{T \xrightarrow{\mathsf{RW}_x} n \mid \mathsf{vis}(n) \wedge \mathsf{writes}(n, x, \_)\}$;
- otherwise, $\{T \xrightarrow{\mathsf{RW}_x} n' \mid \mathsf{writes}(n', x, v) \wedge n' \xrightarrow{\mathsf{WW}_x} n\}$.

Thus, it is sufficient to prove that $\mathsf{PAD}(T, x, v)$ holds of (s', H', G'). This follows from Proposition A.24. □

A.4.6 Preservation of INV.2

It is easy to see that the invariants INV.2(b,c) hold by construction of the history. In the following, we argue that INV.2(a) is preserved throughout each execution.

In order to ensure consistency of a history H, according to Definition 4.18, we need to demonstrate the following for every read request $\psi = (\_, \_, \mathsf{read}(x))$ and its matching response $\psi' = (\_, \_, \mathsf{ret}(v))$:

- when $(\psi, \psi') \in \mathsf{Local}(H)$ and performed by a transaction T, v is the value written by the most recent write $(\_, \_, \mathsf{write}(x, v))$ preceding the read in T;
- when $(\psi, \psi') \notin \mathsf{Local}(H)$, there exists a non-local β not located in an aborted or live transaction and such that $\beta <_{\mathsf{wr}_x(H)} \psi'$; if there is no such write, $v = v_{\mathsf{init}}$.

It is easy to see that it is sufficient to ensure preservation of consistency only when we add read-response actions into history, which happens at line 17 or line 25 of the `read` function.

Let us consider any transaction T that initiated a read operation from a register x (the case of a non-transactional read is analogous to a non-local read by a transaction). The last action by T in history is a read-request $\psi = (\_, \_, \mathsf{read}(x))$. Let a corresponding read response be $\psi' = (\_, \_, \mathsf{ret}(v))$ and consider the moment it is added into history H, which results in a history H'.

We first consider the case when $(\psi, \psi') \in \mathsf{Local}(H)$ holds. By Definition 4.17, T writes to x before ψ. Let v' be the value of the last such write; then $\mathsf{writes}(T, x, v')$ holds of H and H'. By INV.2(b), $(x, v') \in \texttt{wset}[T]$. Note that $(x, v') \in \texttt{wset}[T]$ holds during the entire read operation, because $\texttt{wset}[T]$ is a local variable of transaction T. It is easy to see that in this case the read operation returns a value from the write-set at line 17, meaning that $v = v'$.

We now consider the case when $(\psi, \psi') \notin \mathsf{Local}(H)$ holds. By Definition 4.17, T does not write to x before ψ, meaning that there is no value v' such that $\mathsf{writes}(T, x, v')$ holds of history during the execution of the read operation. Hence, by INV.2(b), $\texttt{wset}[T]$ does not contain a value for x. Moreover, the latter is the case during the entire read operation, because $\texttt{wset}[T]$ is a local variable of transaction T. It is easy to see that in this case the read operation at line 25 returns the value v read directly from the register; however, the result of the operation is determined at line 21. Based on INV.8(a) (note that the read

operation ensures that $\texttt{lock}[x]$ is unlocked), the following two observations can be made about the value v at the moment of executing line 21:

1. $v \neq v_{\mathsf{init}}$ if and only if there exists a node n such that $\mathsf{isLastIn}(\mathsf{WW}_x, n)$ and $\mathsf{writes}(n, x, v)$ both hold;
2. $v = v_{\mathsf{init}}$ if and only if there is no visible node writing to x.

First, let us consider the case when there exists a node n such that $\mathsf{isLastIn}(\mathsf{WW}_x, n)$ and $\mathsf{writes}(n, x, v)$ both hold. Since it is ordered by WW_x, by Definition 4.19, it is visible, and, therefore, does not denote a live or an aborted transaction. According to $\mathsf{writes}(n, x, v)$, n contains a non-local write action β writing v to x. Thus, $\beta <_{\mathsf{wr}_x(H')} \psi'$ holds. Overall, in this case, if $(\psi, \psi') \notin \mathsf{Local}(H')$ and $v \neq v_{\mathsf{init}}$, then there exists a non-local $\beta \in n$, which is not an aborted or live transaction, such that $\beta <_{\mathsf{wr}_x(H)} \psi'$.

We now consider the case when there does not exist a visible node writing to x, which is when $v = v_{\mathsf{init}}$. Thus, any node n writing to x must be either a live, aborted or commit-pending non-visible transaction. Additionally, in case of the latter, if β is the non-local write to x by n, $\beta <_{\mathsf{wr}_x(H')} \psi'$ does not hold, since ψ' returns $v = v_{\mathsf{init}}$ and β writes a non-initial value. Overall, in this case, if $(\psi, \psi') \notin \mathsf{Local}(H')$ and $v = v_{\mathsf{init}}$, then there does not exists a non-local β not located in an aborted or live transaction and such that $\beta <_{\mathsf{wr}_x(H)} \psi'$.

A.4.7 Preservation of INV.3

Proposition A.27. *If $(s, H, G) \in \mathsf{INV}$ and (s', H', G') is a result adding a new node n by graph updates $\mathsf{TXBEGIN}(n)$, $\mathsf{NTXREAD}(n, \_)$ or $\mathsf{NTXWRITE}(n, \_)$, the following observation holds of (s', H', G'): $\neg\exists n'. n \xrightarrow{\mathsf{HB} \cup \mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}} n'$.*

Proof sketch. The proof of the proposition is almost trivial, as $\mathsf{TXBEGIN}(n)$ and $\mathsf{NTXWRITE}(n, \_)$ always order new graph nodes after existing ones. The same is true of $\mathsf{NTXREAD}(n, x)$ as well (for every register x), however, it is necessary to demonstrate that this graph update does not introduce anti-dependencies originating in n.

We consider the register x. Let us first assume that it is locked at the moment of the graph update $\mathsf{NTXREAD}(n, x)$, i.e., there exists a transaction T such that $\texttt{lock}[x] = T$. By INV.8(e), T writes to x, and, therefore, conflicts with n. Since we only consider data-race free histories, it must be the case that either $T \xrightarrow{\mathsf{HB}} n$ or $n \xrightarrow{\mathsf{HB}} T$. However, neither is possible: the former, because according to INV.8(d,e) T is not completed and therefore cannot have outgoing happens-before edges, and the latter, because n is a fresh node, and $\mathsf{NTXREAD}(n, x)$ does not introduce edges of the form $n \xrightarrow{\mathsf{HB}} \_$. Thus, we obtained a contradiction. We conclude that x is not locked at the moment of the graph update.

When $\texttt{lock}[x] = \bot$, by INV.8(a), the value of $\texttt{reg}[x]$ is either the value written by the last node in WW_x or v_{init}, if there is no such node. Thus, it is easy to see that in both cases no anti-dependency of the form $n \xrightarrow{\mathsf{RW}_x} \_$ is added into G by $\mathsf{NTXREAD}(n, x)$. □

Proposition A.28. *If $(s, H, G) \in \mathsf{INV}$ and (s', H', G') is a result of a graph update, then (s', H', G') satisfies INV.3.*

Proof. We consider all possible graph updates separately in the four following cases.

Case 1: non-transactional graph update $\mathsf{NTXWRITE}(\nu, \_)$ or $\mathsf{NTXREAD}(\nu, \_)$. Both updates add a new node ν to the graph, which by Proposition A.27 does not have outgoing edges. Hence, it is easy to see that $(\mathsf{HB} \,;\, (\mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}))$ remains irreflexive after either of the graph updates.

Case 2: transactional graph update $\mathsf{TXBEGIN}(T)$. This update adds a new node T to the graph and happens-before edges of the following form: $\_ \xrightarrow{\mathsf{HB}} T$. Note that by Proposition A.27, no edge starts from T. Hence, it is easy to see that $(\mathsf{HB} \,;\, (\mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}))$ remains irreflexive after the graph update.

Case 3: transactional graph update $\mathsf{TXVIS}(T)$. It adds edges only of the following form: $\_ \xrightarrow{\mathsf{WW}} T$ and $\_ \xrightarrow{\mathsf{RW}} T$. Adding such edges cannot invalidate irreflexivity of $(\mathsf{HB} \,;\, (\mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}))$, unless there is a node n such that $T \xrightarrow{\mathsf{HB}} n$ holds of (s, H, G). However, by invariant $\mathsf{INV}.8(\mathrm{d})$, HB-edges do not originate in T, so we can conclude that $(\mathsf{HB} \,;\, (\mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}))$ remains irreflexive after $\mathsf{TXVIS}(T)$.

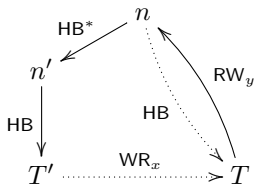
Case 4: transactional graph update $\mathsf{TXREAD}(T, x, \_)$. According to Figure A.9, it adds edges of the form $\_ \xrightarrow{\mathsf{WR}_x} T$, $T \xrightarrow{\mathsf{RW}_x} \_$ and $\_ \xrightarrow{\mathsf{HB}} T$. Note that the transaction T is not completed at the moment (s, H, G). Hence, by invariant $\mathsf{INV}.8(\mathrm{d})$, HB-edges do not originate from T prior to the update. The transaction T is also not visible at the moment of the graph update, so WW and WR-edges do not originate from T either according to Definition 4.19. Therefore, the only kind of edges going from T in G and G' is anti-dependencies of the form $T \xrightarrow{\mathsf{RW}} \_$. We need to demonstrate that there is no node n such that a cycle $T \xrightarrow{\mathsf{RW}} n \xrightarrow{\mathsf{HB}} T$ appears in G' after the graph update. To this end, we consider three possibilities:

1. only the edge $T \xrightarrow{\mathsf{RW}_x} n$ is added by the graph update, and $n \xrightarrow{\mathsf{HB}} T$ is present in G;
2. only the edge $n \xrightarrow{\mathsf{HB}} T$ is added by the graph update, and $T \xrightarrow{\mathsf{RW}_y} n$ is present in G (for some register y);
3. edges $T \xrightarrow{\mathsf{RW}_x} n$ and $n \xrightarrow{\mathsf{HB}} T$ are both added by the graph update.

We start with the first potential cycle, and demonstrate by contradiction that it never takes place. By Proposition A.24, n is a transaction and $\mathtt{rver}[T] < \mathtt{wver}[n]$. Since n is a transaction, Lemma 4.29 gives us that there is a path $n \xrightarrow{\mathsf{RT} \cup \mathsf{TXWR}}{}^{+} T$ in the graph G. Recall that T is not visible and, according to the definition of $\mathsf{TXREAD}(T, x, \_)$ in Figure A.9, n is visible. By applying Proposition A.29 to the path $n \xrightarrow{\mathsf{RT} \cup \mathsf{TXWR}}{}^{+} T$, we learn that either $\mathtt{wver}[n] \leq \mathtt{rver}[T]$ or $\mathtt{rver}[T] = \bot$ holds of (s, H, G). However, at the moment of the graph update, transactions already have generated their read timestamps, meaning that only $\mathtt{wver}[n] \leq \mathtt{rver}[T]$ can be the case. Thus, we arrived to a contradiction.

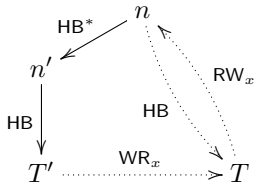
We now consider the second potential cycle, and demonstrate by contradiction that it is never added. Let us assume that $T \xrightarrow{\mathsf{RW}_y} n$ is in the graph G. Hence, by definition of RW_y, T must have already read from y, from which

we conclude that $y \neq x$. We also assume that the edge $n \xrightarrow{\mathsf{hb}} T$ is added by the graph update, which, according to Figure A.9, only happens when T' is a transaction from which T reads from and there is a node n' in the thread of T' such that the following configuration takes place:



where the solid lines depict edges present in the original graph G, and the dotted lines are new edges added by the graph update. We consider two possibilities depending on whether n represents a transaction or not. Let us first assume it is a transaction. By Lemma 4.29, $n \xrightarrow{\mathsf{RT} \cup \mathsf{TXWR}}{}^{+} T'$ holds of the graph G. By applying Proposition A.29 to the path $T \xrightarrow{\mathsf{RW}_y} n \xrightarrow{\mathsf{RT} \cup \mathsf{TXWR}}{}^{+} T'$, we learn that $\mathtt{rver}[T] < \mathtt{wver}[T']$. However, the graph update happens only if $\mathtt{wver}[T'] \leq \mathtt{rver}[T]$, meaning that this possibility never arises. Let us now assume that n is non-transactional. Since H is data-race free, either $T \xrightarrow{\mathsf{HB}} n$ or $n \xrightarrow{\mathsf{HB}} T$ is the case for the graph G. Knowing that $(s, H, G) \in \mathsf{INV}.3$ and that $T \xrightarrow{\mathsf{RW}_y} n$ is in G, we conclude that only $T \xrightarrow{\mathsf{HB}} n$ is possible. Hence, $T \xrightarrow{\mathsf{HB}} T'$ holds, and, by Lemma 4.29, so does $T \xrightarrow{\mathsf{RT} \cup \mathsf{TXWR}}{}^{+} T'$. By applying Proposition A.29 to the latter path, we learn that $\mathtt{rver}[T] < \mathtt{wver}[T']$. However, the graph update happens only if $\mathtt{wver}[T'] \leq \mathtt{rver}[T]$, meaning that this possibility never arises.

Finally, we consider the last possible cycle, and demonstrate by contradiction that it is never added. Let us assume that the edges $T \xrightarrow{\mathsf{RW}_x} n$ and $n \xrightarrow{\mathsf{HB}} T$ are both added by the graph update and form a cycle. This is only possible if the following configuration (analogous to the case of the second cycle) takes place:



where the solid lines depict edges present in the original graph G, and the dotted lines are new edges added by the graph update. By Proposition A.24, n is a transaction and $\mathtt{rver}[T] < \mathtt{wver}[n]$. By Lemma 4.29, $n \xrightarrow{\mathsf{RT} \cup \mathsf{TXWR}}{}^{+} T'$ holds of the graph G. Note that n is visible transaction, and T is not. By applying Proposition A.29 to the path $n \xrightarrow{\mathsf{RT} \cup \mathsf{TXWR}}{}^{+} T' \xrightarrow{\mathsf{WR}_x} T$, we learn that either $\mathtt{wver}[n] \leq \mathtt{rver}[T]$ or $\mathtt{rver}[T] = \bot$. However, the latter cannot be the case at the moment of the graph update. Hence, we obtained a contradiction: $\mathtt{rver}[T] < \mathtt{wver}[n]$ and $\mathtt{wver}[n] \leq \mathtt{rver}[T]$ both hold. □

A.4.8 Preservation of INV.5

Proposition A.29. *If $(s, H, G) \in \mathsf{INV}$, $T \in \mathsf{txns}(H)$ and (s', H', G') is a result of a graph update, then (s', H', G') satisfies $\mathsf{INV}.5$.*

Proof. Graph updates $\mathsf{NTXREAD}(\_, \_)$ and $\mathsf{NTXWRITE}(\_, \_)$ do not affect the invariant. Therefore, we consider the following cases in the proof: graph updates $\mathsf{TXBEGIN}(T')$, $\mathsf{TXREAD}(T', x)$, $\mathsf{TXVIS}(T')$ and also primitive commands at at line 11, line 58 and line 53 in Figure A.7. We consider primitive commands in the proof, because the ones at line 11 and line 53 may change the implication of INV.5 for existing nodes and edges satisfying its premise, meaning that we need to prove that the implication remains true. The rest of the graph updates may add new nodes and edges satisfying the premise of the invariant, so we demonstrate that the implication holds of them too.

We first consider a primitive command at line 11 assigning a read version to a transaction T', which happens only when $\texttt{rver}[T'] = \bot$. As follows from invariants INV.7(c,d,e), in order to have the minimal read timestamp T' must be neither visible nor have non-locally read from any register. This is not the case for transactions in the invariants INV.5(b-e), so we focus on the invariant INV.5(a). Let us consider a transaction T such that $T \xrightarrow{\mathsf{RT}} T'$ holds. By invariant INV.7(b), $\texttt{rver}[T] \leq \texttt{clock}$ and either $\texttt{wver}[T] = \top$ or $\texttt{wver}[T] \leq \texttt{clock}$. Let us assume that $\mathsf{vis}(T)$ holds. By INV.7(e), $\texttt{wver}[T] \neq \top$, so only $\texttt{wver}[T] \leq \texttt{clock}$ is possible. Since the new value of $\texttt{rver}[T']$ is clock, it is the case that $\texttt{wver}[T] \leq \texttt{rver}[T']$, meaning that the invariant INV.5(a) is preserved. Let us now consider the case when $\mathsf{vis}(T)$ does not hold. Knowing that $\texttt{rver}[T] \leq \texttt{clock}$ and that the new value of $\texttt{rver}[T']$ is clock, we conclude $\texttt{rver}[T] \leq \texttt{rver}[T']$, which means that the invariant INV.5(a) is preserved.

Second, we consider a primitive command at line 53 assigning a write timestamp to a transaction, which happens only when the transaction has the write timestamp $\top$. The contra-positive of INV.7(e) states that such transaction is not visible. Note that every occurrence of $\texttt{wver}[\cdot]$ in INV.5 corresponds to a visible transaction. Therefore, line 53 does not invalidate INV.5.

Third, we consider a graph update $\mathsf{TXBEGIN}(T')$. This graph update adds a new node corresponding to the transaction T'. We also need to consider real-time order between completed transactions and T'. Let T be any completed transaction such that $T \xrightarrow{\mathsf{RT}} T'$. Since $\texttt{rver}[T']$ is initialized with $\texttt{rver}[T'] = \bot$, INV.6(a) holds trivially.

Forth, we consider a graph update $\mathsf{TXREAD}(T', x, \_)$ adding read and anti-dependencies between transaction. Let us first consider the new read dependency $T \xrightarrow{\mathsf{WR}_x} T'$. Since the graph update only happens if $\texttt{wver}[T] \leq \texttt{rver}[T']$ holds, the invariant INV.5 is not invalidated by adding the new read dependency. We now consider any new anti-dependency $T' \xrightarrow{\mathsf{RW}_x} T''$. By Proposition A.24, $\texttt{rver}[T'] < \texttt{wver}[T'']$, meaning that INV.5(d) is preserved too.

Fifth, we consider a graph update $\mathsf{TXVIS}(T')$. Let us consider any new edge $T \xrightarrow{\mathsf{WW}_x \cup \mathsf{RW}_x} T'$. Since the transaction T' first locks registers of its write-set and then performs $\mathsf{TXVIS}(T')$, $\texttt{lock}[x] = T'$ holds at the moment of the graph update. Preservation of INV.5(c–e) by the graph update then follows immediately from INV.6(a–c) accordingly.

Finally, we consider a possibility of a primitive command by T at line 58 setting $\texttt{pv}[T][x]$ to true for some register x. Let us assume that there exists a transaction T' such that $T \xrightarrow{\mathsf{RW}_x} T'$ holds prior to the execution of line 58 (if there are several such transactions, we let T' be the one occurring the last in WW_x). By INV.5(d), $\texttt{rver}[T] < \texttt{wver}[T']$ holds then. Note that line 58 sets

$\texttt{pv}[T][x]$ to true only if $\texttt{lock}[x] = \bot$ and $\texttt{ts} \leq \texttt{rver}[T]$, where $\texttt{ts} = \texttt{ver}[x]$. By INV.8(b), $\texttt{ver}[x] = \texttt{wver}[T']$. As a consequence, $\texttt{wver}[T'] \leq \texttt{rver}[T]$ holds, which contradicts our previous observation about $\texttt{wver}[T']$ and $\texttt{rver}[T]$. Hence, there does not exist a transaction T' such that $T \xrightarrow{\mathsf{RW}_x} T'$, and INV.5(e) is preserved by the primitive command at line 58. □

A.4.9 Preservation of INV.4

In this section, we demonstrate that all graph updates preserve the invariant INV.4. To this end, firstly, we observe that graph updates NTXREAD(_, _) and NTXWRITE(_, _) do not introduce edges between pairs of transactions and therefore cannot possibly invalidate INV.4. Secondly, we argue that the graph update TXINIT(T), which adds a new transaction T into the graph and implies new real-time order edges between transactions ending in T, also straightforwardly preserves INV.4. As we previously demonstrated in Proposition A.27, no edges originate from the new transaction T, so it is easy to see that the graph update does not introduce any cycle and, thus, preserves INV.4. In the rest of the section, we consider the remaining graph updates TXREAD(_, _) and TXVIS(_), we prove Proposition A.30 and Proposition A.32.

Proposition A.30. *If $(s, H, G) \in \mathsf{INV}$, $T' \in \mathsf{txns}(H)$ and (s', H', G') is a result of a graph update $\mathsf{TXREAD}(T', x)$, then (s', H', G') satisfies INV.4.*

Proof. Before the graph update, the read operation stores the value of $\texttt{ver}[x]$ into a local variable ts1, then it stores $\texttt{reg}[x]$ in a local variable value as well as the lock status $\texttt{lock}[x].\texttt{test}()$ in locked. Afterwards, when the read operation stores the value of $\texttt{ver}[x]$ once again into a local variable ts2, the graph update $\mathsf{TXREAD}(T', x)$ may happen, provided that the following holds of (s, H, G) prior to the update:

$$\neg\texttt{locked} \wedge \texttt{ts1} = \texttt{ts2} \wedge \texttt{ts2} \leq \texttt{rver}[T'] \tag{A.31}$$

Note that by checking the above, the read operation ensures that there was a moment in between the two accesses to $\texttt{ver}[x]$, when $\texttt{ver}[x] = \texttt{ts1} = \texttt{ts2}$, $\texttt{reg}[x] = \texttt{value}$ and $\texttt{lock}[x] = \bot$ all simultaneously held. Let (s'', H'', G'') correspond to that moment.

Let us assume that the condition (A.31) holds, and the graph update takes place. Since $\mathsf{TXREAD}(T', x)$ adds different edges depending on whether $\texttt{value} = v_{\mathsf{init}}$ holds, we consider these two cases in the proof separately.

Firstly, we consider the case when $\texttt{value} = v_{\mathsf{init}}$ holds. According to Figure A.9, only RW edges are added into the graph in this case. We prove that such edges do not create cycles by contradiction. Let us imagine a cycle in the graph G' going through a new anti-dependency edge $T' \xrightarrow{\mathsf{RW}_x} T''$. The path $T'' \xrightarrow{\mathsf{RT} \cup \mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}}{}^{+} T'$ is present in the graph G. Hence, after applying Proposition A.21 to visible T'' and non-visible T', we obtain that $\texttt{wver}[T''] \leq \texttt{rver}[T']$. Additionally, by Proposition A.24, the anti-dependency $T' \xrightarrow{\mathsf{RW}_x} T''$ implies that $\texttt{rver}[T'] < \texttt{wver}[T'']$. Overall, we obtained a contradiction.

Secondly, we consider the case when $\texttt{value} \neq v_{\mathsf{init}}$. According to Figure A.9, HB, RW and WR edges are added into the graph. However, in this proof we do not consider the new HB edges, since they cannot cause cycles invalidating INV.4. For the same reason, we only consider WR-edges originating from transactions.

Let $T \xrightarrow{\mathsf{WR}_x} T'$ be the read dependency added and let $T' \xrightarrow{\mathsf{RW}_x} T''$ be any of the new anti-dependencies added by the graph update. We prove by contradiction that neither of them causes a cycle in $(\mathsf{RT} \cup \mathsf{txWR} \cup \mathsf{txWW} \cup \mathsf{txRW})$.

The invariant INV.8(a) asserts that the value of the register $\texttt{reg}[x]$ previously stored in a local variable `value` is the value written by the last transaction in WW_x. From this observation, it is easy to conclude that T is this transaction at the moment (s'', H'', G''). By INV.8(b), $\texttt{ver}[x]$ coincides with the write version $\texttt{wver}[T]$. Knowing that the condition (A.31) holds, we conclude that so does $\texttt{wver}[T] \leq \texttt{rver}[T']$. Also, by Proposition A.24, the anti-dependency $T' \xrightarrow{\mathsf{RW}_x} T''$ implies that $\texttt{rver}[T'] < \texttt{wver}[T'']$.

Adding edges $T \xrightarrow{\mathsf{WR}_x} T'$ or $T' \xrightarrow{\mathsf{RW}_x} T''$ may cause three kinds of cycles: the ones containing exactly one of the two edges, and the one containing both. We consider these cases separately:

Case #1: $T \xrightarrow{\mathsf{WR}_x} T'$. Previously, we have shown that $\texttt{wver}[T] \leq \texttt{rver}[T']$ holds. Also, by INV.7(d), $\texttt{rver}[T'] \neq \bot$ holds. Overall, both $\neg(\texttt{rver}[T'] < \texttt{wver}[T])$ and $\texttt{rver}[T'] \neq \bot$ hold. Note that T' is not visible and T is. When all of the aforementioned takes place, the contra-positive of Proposition A.21 states that there is no path $T' \xrightarrow{\mathsf{RT} \cup \mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}}{}^{+} T$. Hence, the edge $T \xrightarrow{\mathsf{WR}_x} T'$ alone does not cause a cycle in $\mathsf{RT} \cup \mathsf{txWR} \cup \mathsf{txWW} \cup \mathsf{txRW}$.

Case #2: $T' \xrightarrow{\mathsf{RW}_x} T''$. Previously, we have shown that $\texttt{rver}[T'] < \texttt{wver}[T'']$ holds. Also, by INV.7(d), $\texttt{rver}[T'] \neq \bot$ holds. Overall, both $\neg(\texttt{wver}[T''] \leq \texttt{rver}[T'])$ and $\texttt{rver}[T'] \neq \bot$ hold. Note that T'' is visible and T' is not. When all of the aforementioned takes place, the contra-positive of Proposition A.21 states that there is no path $T'' \xrightarrow{\mathsf{RT} \cup \mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}}{}^{+} T'$. Hence, the edge $T' \xrightarrow{\mathsf{RW}_x} T''$ alone does not cause a cycle in $\mathsf{RT} \cup \mathsf{txWR} \cup \mathsf{txWW} \cup \mathsf{txRW}$.

Case #3: $T \xrightarrow{\mathsf{WR}_x} T' \xrightarrow{\mathsf{RW}_x} T''$. Previously, we have shown that $\texttt{wver}[T] \leq \texttt{rver}[T'] < \texttt{wver}[T'']$ holds. Note that T'' and T are both visible. As the contra-positive of Proposition A.21 states, if $\texttt{wver}[T''] < \texttt{wver}[T]$ does not hold, then there is no path $T'' \xrightarrow{\mathsf{RT} \cup \mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}}{}^{+} T$. Hence, the edges $T \xrightarrow{\mathsf{WR}_x} T' \xrightarrow{\mathsf{RW}_x} T''$ do not cause a cycle in $\mathsf{RT} \cup \mathsf{txWR} \cup \mathsf{txWW} \cup \mathsf{txRW}$.

□

Proposition A.32. *If $(s, H, G) \in \mathsf{INV}$, $T \in \mathsf{txns}(H)$ and (s', H', G') is a result of a graph update $\mathsf{TXVIS}(T')$, then (s', H', G') satisfies INV.4.*

Proof. Note the graph update $\mathsf{TXVIS}(T')$ occurs after line 63 in the transaction T', meaning that the following holds of T' in (s, H, G):

- $\forall (x, \_) \in \texttt{wset}[T'].\,\texttt{lock}[x] = T'$;
- $\forall (x, \_) \in \texttt{rset}[T'].\,\texttt{pv}[T'][x] = \texttt{true}$.

The graph update $\mathsf{TXVIS}(T')$ adds the following new edges into the opacity graph $G = (N, \mathsf{vis}, \mathsf{HB}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$ for every register x such that $(x, \_) \in \texttt{wset}[T']$ holds:

- $\{n \xrightarrow{\mathsf{WW}_x} T' \mid n \in N \wedge n \neq T' \wedge \mathsf{vis}(n) \wedge \mathsf{writes}(n, x, \_)\}$

- $\{n \xrightarrow{\mathsf{RW}_x} T' \mid n \in N \wedge n \neq T' \wedge \mathsf{reads}(n, x, \_)\}$

The update also makes T' visible in (s', H', G'). It is easy to see that only new edges between transactions can invalidate INV.4, since it asserts absence of cycles over transactions only. As the new dependencies end in the same node T', they will not appear in the same simple cycle. These two observations together allow us to consider each edge individually and prove that it does not create a cycle in $(\mathsf{RT} \cup \mathsf{txWR} \cup \mathsf{txWW} \cup \mathsf{txRW})$.

WW-edges. We show that adding an edge $T \xrightarrow{\mathsf{WW}_x} T'$ into the opacity graph G preserves INV.4.

Note that $\mathtt{lock}[x] = T'$ and that T is such that $\mathsf{vis}(T)$, $T \neq T'$ and $\mathsf{writes}(T, x, \_)$ all hold of (s, H, G); then from INV.6(a) we infer that $\mathtt{wver}[T] < \mathtt{wver}[T']$. The latter also holds of (s', H', G'), as the graph update does not change write timestamps of transactions.

We demonstrated in Proposition A.29 that (s', H', G') satisfies INV.5 and argued that it also satisfies INV.8. This enables applying Proposition A.21 to (s', H', G'). Note that $\mathsf{vis}(T)$ and $\mathsf{vis}(T')$ both hold of (s', H', G'). The contrapositive of Proposition A.21 asserts that $\mathsf{vis}(T)$, $\mathsf{vis}(T')$ and $\mathtt{wver}[T] < \mathtt{wver}[T']$ together imply that $T' \xrightarrow{\mathsf{RT} \cup \mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}}{}^{+} T$ does not hold, meaning that adding the edge $T \xrightarrow{\mathsf{WW}_x} T'$ into G does not create cycles in $\mathsf{RT} \cup \mathsf{txWR} \cup \mathsf{txWW} \cup \mathsf{txRW}$.

RW-edges. We show that adding the edge $T \xrightarrow{\mathsf{RW}_x} T'$ into the opacity graph G preserves INV.4. We consider the triple (s, H, G) of the state prior to the graph update and split the proof in two cases depending on whether T is visible or not.

First, we consider the case when T is visible. By INV.8(c), $\mathtt{pv}[T][x] = \mathtt{true}$ holds of (s, H, G). Also, recall that T' holds a lock on x. When that is the case, according to INV.6(c), $\mathtt{wver}[T] < \mathtt{wver}[T']$ holds. The latter also holds of (s', H', G'), as the graph update does not change write timestamps of transactions.

We demonstrated in Proposition A.29 that (s', H', G') satisfies INV.5 and argued that it also satisfies INV.8. This enables applying Proposition A.21 to (s', H', G'). Note that $\mathsf{vis}(T)$ and $\mathsf{vis}(T')$ both hold of (s', H', G'). The contrapositive of Proposition A.21 asserts that $\mathsf{vis}(T)$, $\mathsf{vis}(T')$ and $\mathtt{wver}[T] < \mathtt{wver}[T']$ together imply that $T' \xrightarrow{\mathsf{RT} \cup \mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}}{}^{+} T$ does not hold, meaning that adding the edge $T \xrightarrow{\mathsf{RW}_x} T'$ into G does not create cycles in $\mathsf{RT} \cup \mathsf{txWR} \cup \mathsf{txWW} \cup \mathsf{txRW}$.

We now return to the case when T is not visible. Recall that T' holds a lock on x. When that is the case, according to INV.6(b), $\mathtt{rver}[T] < \mathtt{wver}[T']$ holds of (s, H, G). The latter also holds of (s', H', G'), as the graph update does not change write timestamps of transactions.

We demonstrated in Proposition A.29 that (s', H', G') satisfies INV.5 and argued that it also satisfies INV.8. This enables applying Proposition A.21 to (s', H', G'). Note that $\neg\mathsf{vis}(T)$ and $\mathsf{vis}(T')$ both hold of (s', H', G'). Also, $\mathtt{rver}[T] \neq \bot$, since by INV.7(d) transaction satisfying $\mathsf{reads}(T, x, \_)$ already has its read timestamp initialized. The contra-positive of Proposition A.21 asserts that $\mathsf{vis}(T')$, $\neg\mathsf{vis}(T)$ and $\neg(\mathtt{wver}[T'] \leq \mathtt{wver}[T] \vee \mathtt{rver}[T] = \bot)$ together imply that $T' \xrightarrow{\mathsf{RT} \cup \mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW}}{}^{+} T$ does not hold, meaning that adding the edge $T \xrightarrow{\mathsf{RW}_x} T'$ into G does not create cycles in $\mathsf{RT} \cup \mathsf{txWR} \cup \mathsf{txWW} \cup \mathsf{txRW}$. □

A.4.10 Preservation of INV.6

Proposition A.33. *When* $(s, H, G) \in \mathsf{INV}$*,* $T \in \mathsf{txns}(H)$ *and* (s', H', G') *is a result of execution of line 58 by* T*, if* $\texttt{lock}[x]$ *is held by some transaction* T'*, then* $\texttt{rver}[T] < \texttt{wver}[T']$*.*

Proof sketch. The transaction T post-validates its read from x at line 58 simultaneously with loading the value of $\texttt{ver}[x]$. At the previous line, the commit operation of T stores the value of $\texttt{lock}[x]$ in a local variables `locked`. The post-validation is successful only if $\neg\texttt{locked}$ and $\texttt{ver}[x] \leq \texttt{rver}[T]$. Note that this check has already been performed when T performed a read operation from x. This way TL2 ensures that there has been a moment between the execution of line 21 in the read operation and line 58, at which both $\texttt{lock}[x] = \bot$ and $\texttt{ver}[x] \leq \texttt{rver}[T]$ hold simultaneously. Let (s'', H'', G'') correspond to that moment.

For convenience, we introduce a predicate $\mathsf{PVP}(T, x)$, which holds of (s, H, G) whenever the following does: $\mathsf{reads}(T, x, \_)$ holds, and if $\texttt{lock}[x]$ is held by some transaction T', then $\texttt{rver}[T] < \texttt{wver}[T']$. It is easy to see that $\mathsf{PVP}(T, x)$ holds trivially of (s'', H'', G''), since $\texttt{lock}[x]$ is not held at that moment. We further demonstrate that $\mathsf{PVP}(T, x)$ is never invalidated by transitions and graph updates of T or by other threads. This allows us to conclude that $\mathsf{PVP}(T, x)$ holds of (s', H', G').

Note that once $\mathsf{PVP}(T, x)$ is established, it could only be possibly invalidated by:

- a transition by T' at line 44 acquiring the lock $\texttt{lock}[x]$;
- a transition by T' at line 53 setting the write timestamps $\texttt{wver}[T']$ from $\top$ to the incremented clock value.

However, both preserve $\mathsf{PVP}(T, x)$. Indeed, when the former transition occurs, $\texttt{wver}[T'] = \top$ holds. Since $\top$ is the maximal possible timestamp, $\texttt{rver}[T] < \texttt{wver}[T']$ holds then. When the second transition occurs in a transaction T' holding a lock on x, by INV.7(b), we observe that $\texttt{rver}[T] < \texttt{clock} + 1$ and that $\texttt{wver}[T'] = \texttt{clock} + 1$, which also allows to conclude $\texttt{rver}[T] < \texttt{wver}[T']$. □

Proposition A.34. *If* $(s, H, G) \in \mathsf{INV}$*,* $T \in \mathsf{txns}(H)$ *and* (s', H', G') *is a result of a graph update, then* (s', H', G') *satisfies* INV.*6.*

Proof. Graph updates $\mathsf{NTXREAD}(\_, \_)$ and $\mathsf{NTXWRITE}(\_, \_)$ do not affect the invariant, since it only asserts properties of transactions. Thus, in this proof we consider the following graph updates and transitions:

1. the graph update $\mathsf{TXVIS}(T)$, which may enable the premise of INV.6(a);
2. the graph update $\mathsf{TXREAD}(T, x, \_)$, which may enable the premise of INV.6(b);
3. the transition by T at line 58 setting $\texttt{pv}[T][x]$ to `true`, which may enable the premise of INV.6(c);
4. the transition by T' at line 44 acquiring the lock $\texttt{lock}[x]$, which may enable the premise of INV.6(a,b,c);

5. the transition by T at line 11 setting the read timestamp $\texttt{rver}[T]$ from $\bot$ to the current clock value, which may affect the implication of INV.6(b);

6. the transition by T (or T') at line 53 setting the write timestamps $\texttt{wver}[T]$ (or $\texttt{wver}[T']$) from $\top$ to the incremented clock value, which may affect the implication of INV.6(a,b,c).

Firstly, we discuss a possibility when the graph update $\mathsf{TXVIS}(T)$ in a transaction T writing to x makes T visible, when $\texttt{lock}[x] = T'$ holds already. Note that $\mathsf{TXVIS}(T)$ only happens when T holds the lock on $\texttt{lock}[x]$, meaning that the possibility in discussion never arises.

Second, we consider a possibility when the graph update $\mathsf{TXREAD}(T, x, v)$ (or rather, the read response added into the history) in a transaction T makes $\mathsf{reads}(T, x, v)$ hold, when $\texttt{lock}[x] = T'$ holds already. In the proof of Proposition A.24, we shown $\mathsf{PAD}(T, x, v)$, meaning that the following holds of T in this case:

- if T reads the initial value, then:

$$\forall n'.\, \mathsf{writes}(n', x, \_) \wedge \neg\mathsf{aborted}(n') \implies (n' \in \mathsf{txns}(H)) \wedge \texttt{rver}[T] < \texttt{wver}[n']$$

- if T reads the value written by a node n, then:

$$\forall n'.\, \mathsf{writes}(n', x, \_) \wedge \neg\mathsf{aborted}(n') \wedge \neg n' \xrightarrow{\mathsf{WW}_x} n \implies (n' \in \mathsf{txns}(H)) \wedge \texttt{rver}[T] < \texttt{wver}[n']$$

We further argue that premises of the two properties above hold of T'. Note that T' holds a lock on x. By INV.8(e), T' is not completed (and, therefore, not aborted), writes to x and is not followed in WW_x by any other transaction. Let us show by contradiction that it is also not followed in WW_x by any non-transactional node either. Let us assume that there is a non-transactional n such that $T' \xrightarrow{\mathsf{WW}_x} n$ holds. By INV.1, the history is DRF, meaning that either $T' \xrightarrow{\mathsf{HB}} n$ or $n \xrightarrow{\mathsf{HB}} T'$ must hold. However, the former contradicts INV.8(d) and the latter contradicts INV.3. Hence, such node n does not exist. Overall, we demonstrated that $\mathsf{writes}(T', x, \_)$, $\neg\mathsf{aborted}(T')$ and $\forall n.\, \neg T' \xrightarrow{\mathsf{WW}_x} n$ all hold, hence, from $\mathsf{PAD}(T, x, v)$ we obtain $\texttt{rver}[T] < \texttt{wver}[T']$.

Third, we consider the transition by T at line 58 setting $\texttt{pv}[T][x]$ to $\texttt{true}$, when $\texttt{lock}[x] = T'$ holds already. By Proposition A.33, $\texttt{rver}[T] < \texttt{wver}[T']$ holds in this case.

Forth, we consider a possibility of the transition by T' at line 44 acquiring $\texttt{lock}[x]$ and enabling the premise of either of the invariants, which we consider separately. To this end, we start with considering the case when $\mathsf{writes}(T, x)$ and $\mathsf{vis}(T)$ hold, and T' acquires $\texttt{lock}[x]$. We need to demonstrate that $\texttt{wver}[T] < \texttt{wver}[T']$. By INV.7(e), $\texttt{wver}[T] \neq \top$. However, at line 44, $\texttt{wver}[T'] = \top$. Hence, $\texttt{wver}[T] < \texttt{wver}[T']$, which concludes INV.6(a). Let us now consider the case when $\mathsf{reads}(T, x, \_)$ holds, and T' acquires $\texttt{lock}[x]$. We need to demonstrate that $\texttt{rver}[T] < \texttt{wver}[T']$. By INV.7(b), $\texttt{rver}[T]$ is smaller than the value of the global clock, so it cannot be $\top$. Knowing that $\texttt{wver}[T'] = \top$ holds, we get that so does $\texttt{rver}[T] < \texttt{wver}[T']$, which concludes INV.6(b). We now consider the

case when $\texttt{pv}[T][x]$ holds, and T' acquires $\texttt{lock}[x]$. We need to demonstrate that $\texttt{wver}[T] < \texttt{wver}[T']$. By INV.7(e), if T post-validated at least one of its reads, $\texttt{wver}[T] \neq \top$. Knowing that at line 44, $\texttt{wver}[T'] = \top$ holds, we get that so does $\texttt{wver}[T] < \texttt{wver}[T']$, which concludes INV.6(c).

Fifth, we consider a possibility of the transition by T at line 11 affecting the implication of INV.6(b). We assume that $\mathsf{reads}(T, x, \_)$ and $\texttt{lock}[x] = T'$ both hold prior to the transition. By INV.7(d), $\texttt{rver}[T] \neq \bot$ then. Since the transition by T at line 11 only happens when $\texttt{rver}[T] = \bot$, it cannot possibly invalidate INV.6(b).

Finally, we consider a possibility of the transition by T or T' at line 53 affecting the implication of INV.6(a, b, c). Let us first show that such transition in fact cannot happen in T. For each of the invariants, we assume that its premise holds, and that $\texttt{wver}[T] < \texttt{wver}[T']$ prior to the transition. This means that $\texttt{wver}[T] \neq \top$, since $\top$ is the maximal timestamp value. Knowing that line 53 executes only when $\texttt{wver}[T] = \top$, we conclude that it cannot invalidate INV.6. We now consider line 53 changing the write timestamp $\texttt{wver}[T']$ from $\top$ to ($\texttt{clock}$+1). For each of the invariants, we assume that its premise holds. Let us show that $\texttt{wver}[T] < \texttt{wver}[T']$ is preserved by the transition. As we have shown, when the aforementioned inequality holds before the transition, $\texttt{wver}[T] \neq \top$. By INV.7(b), the value ($\texttt{clock}$+1) is greater than every write timestamp distinct from $\top$, such as $\texttt{wver}[T]$. Hence, $\texttt{wver}[T] < \texttt{wver}[T']$ holds after the transition. We now show that $\texttt{rver}[T] < \texttt{wver}[T']$ is preserved by the transition. By INV.7(b), the value ($\texttt{clock}$+1) is greater than every read timestamp, such as $\texttt{rver}[T]$. Hence, $\texttt{rver}[T] < \texttt{wver}[T']$ holds after the transition. □