Composite Replicated Data Types[⋆]

Alexey Gotsman[1] and Hongseok Yang[2]

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

Abstract. Modern large-scale distributed systems often rely on eventually consistent replicated stores, which achieve scalability in exchange for providing weak semantic guarantees. To compensate for this weakness, researchers have proposed various abstractions for programming on eventual consistency, such as replicated data types for resolving conflicting updates at different replicas and weak forms of transactions for maintaining relationships among objects. However, the subtle semantics of these abstractions makes using them correctly far from trivial.

To address this challenge, we propose composite replicated data types, which formalise a common way of organising applications on top of eventually consistent stores. Similarly to an abstract data type, a composite data type encapsulates objects of replicated data types and operations used to access them, implemented using transactions. We develop a method for reasoning about programs with composite data types that reflects their modularity: the method allows abstracting away the internals of composite data type implementations when reasoning about their clients. We express the method as a denotational semantics for a programming language with composite data types. We demonstrate the effectiveness of our semantics by applying it to verify subtle data type examples and prove that it is sound and complete with respect to a standard non-compositional semantics.

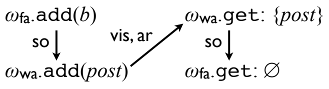
1 Introduction

Background. To achieve availability and scalability, many modern networked systems use *replicated stores*, which maintain multiple *replicas* of shared data. Clients can access the data at any of the replicas, and these replicas communicate changes to each other using message passing. For example, large-scale Internet services use data replicas in geographically distinct locations, and applications for mobile devices keep replicas locally as well as in the cloud to support offline use. Ideally, we would like replicated stores to provide *strong consistency*, i.e., to behave as if a single centralised replica handles all operations. However, achieving this ideal usually requires synchronisation among replicas, which slows down the store and even makes it unavailable if network connections between replicas fail [12, 2]. For this reason, modern replicated stores often provide weaker guarantees, described by the umbrella term of *eventual consistency* [4].

⋆ We thank Giovanni Bernardi, Sebastian Burckhardt and Andrea Cerone for many interesting discussions and thoughtful comments about our results and the paper. Our work was supported by the EU FET project ADVENT and EPSRC.

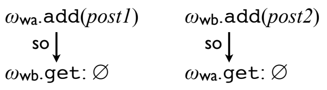
(a) Disallowed

R1: $\mathit{friends}[a]$.add(b)	R2: $v = \mathit{wall}[a]$.get // $\{post\}$
$\mathit{wall}[a]$.add($post$)	$w = \mathit{friends}[a]$.get // $\emptyset$


ω_{fa}.add(b)
so
ω_{wa}.add($post$)
vis, ar
ω_{wa}.get: $\{post\}$
so
ω_{fa}.get: $\varnothing$

(b) Allowed, even when using transactions

R1: $\mathit{wall}[a]$.add($post1$)	R2: $\mathit{wall}[b]$.add($post2$)
$v = \mathit{wall}[b]$.get // $\emptyset$	$w = \mathit{wall}[a]$.get // $\emptyset$


ω_{wa}.add($post1$)
so
ω_{wb}.get: $\varnothing$
ω_{wb}.add($post2$)
so
ω_{wa}.get: $\varnothing$

(c) Disallowed

R1: atomic {	R2: atomic {
$\mathit{friends}[a]$.add(b)	$v = \mathit{friends}[a]$.get // $\{b\}$
$\mathit{friends}[b]$.add(a) }	$w = \mathit{friends}[b]$.get // $\emptyset$ }

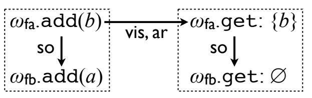

ω_{fa}.add(b)
so
ω_{fb}.add(a)
vis, ar
ω_{fa}.get: $\{b\}$
so
ω_{fb}.get: $\varnothing$

Fig. 1. Anomalies illustrating the semantics of causal consistency and causally consistent transactions. The outcomes of operations are shown in comments. The variables v and w are local to clients. The structures shown on the right are explained in §3.2.

Eventually consistent stores adopt an architecture where a replica performs an operation requested by a client locally without any synchronisation with others and immediately returns to the client; the effects of the operation are propagated to other replicas only *eventually*. As a result, different replicas may find out about an operation at different points in time. This leads to *anomalies*, one of which is illustrated by the outcome in Figure 1(a). The program shown there consists of two clients operating on set objects $\mathit{friends}[a]$ and $\mathit{wall}[a]$, which represent information about a user a in a social network application. The first client, connected to replica 1, makes b a friend of a's and then posts b's message on a's wall. After each of these operations, replica 1 might send a message with an update to replica 2. If the messages carrying the additions of b to $\mathit{friends}[a]$ and $post$ to $\mathit{wall}[a]$ arrive at replica 2 out of order, the second client can see b's $post$, but does not know that b has become a's friend. This outcome cannot be produced by any interleaving of the operations shown in Figure 1(a) and, hence, is not strongly consistent.

The *consistency model* of a replicated store restricts the anomalies that it exhibits. In this paper, we consider the popular model of *causal consistency* [17], a variant of eventual consistency that strikes a reasonable balance between programmability and efficiency. A causally consistent store disallows the anomaly in Figure 1(a), because it respects causal dependencies between operations: if the programmer sees b's $post$ to a's wall, she is also guaranteed to see all events that led to this posting, such as the addition of b to the set of a's friends. Causal consistency is weaker than strong consistency; in particular, it allows reading stale data. This is illustrated by the outcome in Figure 1(b), which cannot be produced by any interleaving of the operations shown. In a causally consistent store it may be produced because each message about an addition sent by the replica performing it may be slow to get to the other replica.

Due to such subtle semantics, writing correct applications on top of eventually consistent stores is very difficult. In fact, finding a good programming model for eventual consistency is considered one of the major research challenges in the systems community [4]. We build on two programming abstractions proposed by researchers to address this challenge, which we now describe.

One difficulty of programming for eventually consistent stores is that their clients can concurrently issue conflicting operations on the same data item at different replicas. For example, spouses sharing a shopping cart in an online store can add and concurrently remove the same item. To deal with these situations, eventually consistent stores provide *replicated data types* [22] that implement *objects*, such as registers, counters or sets, with various strategies for resolving conflicting updates to them. The strategies can be as simple as establishing a total order on all operations using timestamps and letting the last writer win, but can also be much more subtle. For example, a set data type, which can be used to implement a shopping cart, can process concurrent operations trying to add and concurrently remove the same element so that ultimately the element ends up in the set.

Another programming abstraction that eventually consistent stores are starting to provide is *transactions*, which make it easier to maintain relationships among different objects. In this paper we focus on *causally consistent transactions*, implemented (with slight variations) by a number of stores [25, 17, 18, 23, 16, 1, 3]. When a causally consistent transaction performs several updates at a replica, we are guaranteed that these will be delivered to every other replica together. For example, consider the execution in Figure 1(c), where at replica 1 two users befriend each other by adding their identifiers to set objects in the array $\mathit{friends}$. If we did not use transactions, the outcome shown would be allowed by causal consistency, as replica 2 might receive the addition of b to $\mathit{friends}[a]$, but not that of a to $\mathit{friends}[b]$. This would break the expected invariant that the friendship relation encoded by *friends* is symmetric. Causally consistent transactions disallow this anomaly, but nevertheless provide weaker guarantees than the classical serialisable ACID transactions. The latter guarantee that operations done within a transaction can be viewed as taking effect instantaneously at all replicas. With causally consistent transactions, even though each separate replica sees updates done by a transaction together, different replicas may see them at different times. For example, the outcome in Figure 1(b) could occur even if we executed the pair of commands at each replica in a transaction, again because of delays in message delivery.

A typical way of using replicated data types and transactions for writing applications on top of an eventually consistent store is to keep the application data as a set of objects of replicated data types, and update them using transactions over these objects [25, 23, 16, 1]. Then replicated data types ensure sensible conflict resolution, and transactions ensure the maintenance of relationships among objects. However, due to the subtle semantics of these abstractions, reasoning about the behaviour of applications organised in this way is far from trivial. For example, it is often difficult to trace how the choice of conflict-resolution policies on separate objects affects the policy for the whole application: as we show in §5, a wrong choice can lead to violations of integrity invariants across objects, resulting in undesirable behaviour.

Contributions. To address this challenge, we propose a new programming concept of a *composite replicated data type* that formalises the above way of organising applications using eventually consistent stores. Similarly to a class or an abstract data type, a composite replicated data type encapsulates *constituent objects* of replicated data types and *composite operations* used to access them, each implemented using a transaction. For example, a composite data type representing the friendship relation in a social network may consist of a number of set objects storing the friends of each user, with transactions used to keep the relation symmetric. Composite data types can also capture the modular structure of applications, since we can construct complex data types from simpler ones in a nested fashion.

We further propose a method for reasoning about programs with composite data types that reflects their modularity: the method allows one to abstract from the internals of composite data type implementations when reasoning about the clients of these data types. Technically, we express our reasoning method as a denotational semantics for a programming language that allows defining composite data types (§4). As any denotational semantics, ours is compositional and is thus able to give a denotation to every composite data type separately. This denotation abstracts from the internal data type structure using what we term *granularity abstraction*: it does not record *fine-grained* events describing operations on the constituent objects that are performed by composite operations, but represents every invocation of a composite operation by a single *coarse-grained* event. Thereby, the denotation allows us to pretend that the composite data type represents a single monolithic object, no different from an object of a *primitive* data type implemented natively by the store. The denotation then describes the data type behaviour using a mechanism recently proposed for specifying primitive replicated data types [9]. The granularity abstraction achieved by this *coarse-grained* denotational semantics is similar (but not identical, as we discuss in §7) to atomicity abstraction, which has been extensively investigated in the context of shared-memory concurrency [11, 24].

Our coarse-grained semantics enables modular reasoning about programs with composite replicated data types. Namely, it allows us to prove a property of a program by: *(i)* computing the denotations of the composite data types used in it; and *(ii)* proving that the program satisfies the property assuming that it uses *primitive* replicated data types with the specifications equal to the denotations of the composite ones. We thus never have to reason about composite data type implementations and their clients together.

Since we use an existing specification mechanism [9] to represent a composite data type denotation, our technical contribution lies in identifying *which* specification to pick. We show that the choice we make is correct by proving that our coarse-grained semantics is sound with respect to a *fine-grained semantics* of the programming language (§6), which records the internal execution of composite operations and follows the standard way of defining language semantics on weak consistency models [9]. We also establish that the coarse-grained semantics is complete with respect to the fine-grained one: we do not lose precision by reasoning with denotations of composite data types instead of their implementations. The soundness and completeness results also imply that our coarse-grained denotational semantics is *adequate*, i.e., can be used for proving the observational equivalence of two composite data type implementations.

We demonstrate the usefulness of the coarse-grained semantics by applying the composite data type denotation it defines to specify and verify small but subtle data types, such as a social graph (§5). In particular, we show how our semantics lets one

$$\begin{array}{llcl@{\qquad}llcl}
\text{Primitive data types} & B & \in & \mathsf{PrimType} & \text{Data-type variables} & \alpha,\beta & \in & \mathsf{DVar} \\
\text{Ordinary variables} & v, w & \in & \mathsf{Var} = \{v_{\mathsf{in}}, v_{\mathsf{out}}, \ldots\} & \text{Object variables} & x, y & \in & \mathsf{OVar} \\
\text{Data-type contexts} & \Gamma & ::= & \alpha_1 : O_1, \ldots, \alpha_k : O_k & \text{Ordinary contexts} & \Sigma & ::= & v_1, \ldots, v_k \\
\text{Object contexts} & \Delta & ::= & x_1 : O_1, \ldots, x_k : O_k & & & &
\end{array}$$

$$\begin{array}{l}
D ::= \mathsf{let}\ \{x_j = \mathsf{new}\, T_j\}_{j=1..m}\ \mathsf{in}\ \{o = \mathsf{atomic}\ \{C_o\}\}_{o \in O} \qquad T ::= B \mid D \mid \alpha \\
G ::= v \mid G + G \mid G \wedge G \mid G \vee G \mid \neg G \mid \ldots \\
C ::= \mathsf{var}\ v.\, C \mid v = x.o(G) \mid v = G \mid C; C \mid \mathsf{if}\ G\ \mathsf{then}\ C\ \mathsf{else}\ C \mid \mathsf{while}\ G\ \mathsf{do}\ C \mid \mathsf{atomic}\ \{C\} \\
P ::= C_1 \parallel \ldots \parallel C_n \mid \mathsf{let}\ \alpha = T\ \mathsf{in}\ P \mid \mathsf{let}\ x = \mathsf{new}\ T\ \mathsf{in}\ P
\end{array}$$

$$\boxed{\Delta \mid \Sigma \vdash C} \quad \frac{\mathsf{FV}(G) \cup \{v\} \subseteq (\Sigma - \{v_{\mathsf{in}}, v_{\mathsf{out}}\})}{\Delta, x : \{o\} \cup O \mid \Sigma \vdash v = x.o(G)} \quad \frac{\Delta \mid \Sigma, v \vdash C}{\Delta \mid \Sigma \vdash \mathsf{var}\ v.\, C} \quad \frac{\Delta \mid \Sigma \vdash C}{\Delta \mid \Sigma \vdash \mathsf{atomic}\ \{C\}}$$

$$\boxed{\Gamma \vdash T : O} \quad \frac{}{\Gamma \vdash B : \mathsf{sig}(B)} \quad \frac{}{\Gamma, \alpha : O \vdash \alpha : O} \quad \frac{\begin{array}{c}\Gamma \vdash T_j : O_j \ \text{ for all } j = 1..m \\ x_1 : O_1, \ldots, x_m : O_m \mid v_{\mathsf{in}}, v_{\mathsf{out}} \vdash C_o \ \text{ for all } o \in O\end{array}}{\Gamma \vdash \mathsf{let}\ \{x_j = \mathsf{new}\ T_j\}_{j=1..m}\ \mathsf{in}\ \{o = \mathsf{atomic}\ \{C_o\}\}_{o \in O} : O}$$

$$\boxed{\Gamma \mid \Delta \vdash P} \quad \frac{\Gamma \vdash T : O \qquad \Gamma \mid \Delta, x : O \vdash P}{\Gamma \mid \Delta \vdash \mathsf{let}\ x = \mathsf{new}\ T\ \mathsf{in}\ P} \quad \frac{\Delta \mid \emptyset \vdash C_j \ \text{ for all } j = 1..n}{\Gamma \mid \Delta \vdash C_1 \parallel \ldots \parallel C_n}$$

Fig. 2. Programming language and sample typing rules.

understand the consequences of different design decisions in the implementation of a composite data type on its behaviour.

2 Programming Language and Composite Replicated Data Types

Store Data Model. We consider a replicated store organised as a collection of ***primitive objects***. Clients interact with the store by invoking ***operations*** on objects from a set Op, ranged over by o. Every object in the store belongs to one of the ***primitive replicated data types*** $B \in \mathsf{PrimType}$, implemented by the store natively. The ***signature*** $\mathsf{sig}(B) \subseteq \mathsf{Op}$ determines the set of operations allowed on objects of the type B. As we explain in §3, the data type also determines the semantics of the operations and, in particular, the conflict-resolution policies implemented by them. For uniformity of definitions, we assume that each operation takes a single parameter and returns a single value from a set of ***values*** Val, whose elements are ranged over by a, b, c, d. We assume that Val includes at least Booleans and integers, their sets and tuples thereof. We use a special value $\bot \in \mathsf{Val}$ to model operations that take no parameter or return no value. For example, primitive data types can include sets with operations `add`, `remove`, `contains` and `get` (the latter returning the set contents).

Composite Replicated Data Types. We develop our results for a language of client programs interacting with the replicated store, whose syntax we show in Figure 2. We consider only programs well-typed according to the rules also shown in the figure. The interface to the store provided by the language is typical of existing implementations [25, 1]. It allows programs to declare objects of primitive replicated data types, residing in the store, invoke operations on them, and combine these into transactions. Crucially, the language also allows declaring ***composite replicated data types*** from the given primitive ones and ***composite objects*** of these types. These composite objects do not actually reside in the store, but serve as client-side anchors for compositions of primitive objects. A declaration D of a composite data type includes several ***constituent objects*** of specified types T_j, which can be primitive types, composite data type declarations or ***data-type variables*** $\alpha \in \mathsf{DVar}$, bound to either. The constituent objects are bound to distinct ***object variables*** x_j, $j = 1..m$ from a set OVar. The declaration D also defines a set of ***composite operations*** O (the type's ***signature***), with each $o \in O$ implemented by a ***command*** C_o executed as a ***transaction*** accessing the objects x_j. We emphasise the use of transactions by wrapping C_o into an atomic block. Since a store implementation executes a transaction at a replica without synchronising with other replicas, transactions never abort.

The syntax of commands includes the form $\mathsf{var}\ v.\, C$ for declaring ***ordinary variables*** $v, w \in \mathsf{Var}$, to be used by C, which store values from Val and are initialised to $\bot$. Commands C_o in composite data type declarations D can additionally access two distinguished ordinary variables v_{in} and v_{out} (never declared explicitly), used to pass parameters and return values of operations: the parameter gets assigned to v_{in} at the beginning of the execution of C_o and the return value is read from v_{out} at the end. The command $v = x.o(G)$ executes the operation o on the object bound to the variable x with parameter G and assigns the result to v.[3]

Our type system enforces that commands only invoke operations on objects consistent with the signatures of their types and that all variables be used within the correct scope; in particular, constituent objects of composite types can only be accessed by their composite operations. For simplicity, we do not adopt a similar type discipline for values and treat all expressions as untyped. Finally, for convenience of future definitions, the typing rule for $v = x.o(G)$ requires that v_{in} and v_{out} do not appear in v or G.

Example: Social Graph. Figure 3 gives our running example of a composite data type `soc`, which maintains friendship relationships and requests between accounts in a toy social network application. To concentrate on core issues of composite data type correctness, we consider a language that does not allow creating unboundedly many objects; hence, we assume a fixed number of accounts N. Using syntactic sugar, the constituent objects are grouped into arrays *friends* and *requesters* and have the type `RWset` of sets with a particular conflict-resolution policy (defined in §3.1). We use these sets to store account identifiers: $\mathit{friends}[a]$ gives the set of a's friends, and $\mathit{requesters}[a]$ the set of accounts with pending friendship requests to a. The implementation maintains the expected integrity invariants that the friendship relation is symmetric and the friend and requester sets of any account are disjoint:

$$\forall a, b.\, \mathit{friends}[a].\mathtt{contains}(b) \Leftrightarrow \mathit{friends}[b].\mathtt{contains}(a); \tag{1}$$

$$\forall a.\, \mathit{friends}[a].\mathtt{get} \cap \mathit{requesters}[a].\mathtt{get} = \emptyset. \tag{2}$$

[3] Since the object bound to x may itself be composite, this may result in atomic blocks being nested. Their semantics is the same as the one obtained by discarding all blocks except the top-level one. In particular, the atomic blocks that we include into the syntax of commands have no effect inside operations of composite data types.

```
$D_{\mathsf{soc}}$ = let { $friends$ = new RWset[$N$]; $requesters$ = new RWset[$N$] } in {
    request($from$, $to$) = atomic {
      if ($friends[to]$.contains($from$) $\vee$ $requesters[to]$.contains($from$)) then $v_{\mathsf{out}}$ = false
      else { $requesters[to]$.add($from$); $v_{\mathsf{out}}$ = true } };
    accept($from$, $to$) = atomic {
      if ($\neg requesters[to]$.contains($from$)) then $v_{\mathsf{out}}$ = false
      else { $requesters[to]$.remove($from$); $requesters[from]$.remove($to$);
              $friends[to]$.add($from$); $friends[from]$.add($to$); $v_{\mathsf{out}}$ = true } };
    reject($from$, $to$) = atomic {
      if ($\neg requesters[to]$.contains($from$)) then $v_{\mathsf{out}}$ = false
      else { $requesters[to]$.remove($from$); $requesters[from]$.remove($to$); $v_{\mathsf{out}}$ = true } };
    breakup($from$, $to$) = atomic {
      if ($\neg friends[to]$.contains($from$)) then $v_{\mathsf{out}}$ = false
      else { $friends[to]$.remove($from$); $friends[from]$.remove($to$); $v_{\mathsf{out}}$ = true } };
    get($id$) = atomic {$v_{\mathsf{out}}$ = ($friends[id]$.get, $requesters[id]$.get) } }
```

Fig. 3. A social graph data type soc.

The composite operations allow issuing a friendship request, accepting or rejecting it, breaking up and getting the information about a given account. For readability, we use some further syntactic sugar in the operations. Thus, we replace v_{in} with more descriptive names, recorded after the operation name and, in the case when the parameter is meant to be a tuple, introduce separate names for its components. Thus, $from$ and to desugar to $\mathsf{fst}(v_{\mathsf{in}})$ and $\mathsf{snd}(v_{\mathsf{in}})$. We also allow invoking operations on objects inside expressions and omit unimportant parameters to operations.

The code of the composite operations is mostly as expected. For example, `request` adds the user sending the request to the requester set of the user being asked, after checking, e.g., that the former is not already a friend of the latter. However, this simplicity is deceptive: when reasoning about the behaviour of the data type, we need to consider the possibility of operations being issued concurrently at different replicas. For example, what happens if two users concurrently issue friendship requests to each other? What if two users managing the same institutional account take conflicting decisions, such as concurrently accepting and rejecting a request? As we argue in §5, it is nontrivial to implement the data type so that the behaviour in the above situations be acceptable. Using the results in this paper, we can specify the desired social graph behaviour and prove that the composite data type in Figure 3 satisfies such a specification. Our specification abstracts from the internal structure of the data type, thereby allowing us to view it as no different from the primitive set data types it is constructed from. This facilitates reasoning about programs using the data type, which we describe next.

Programs. A ***program*** P consists of a series of data type and object variable declarations followed by a ***client***. The latter consists of several commands $C_1, \dots, C_n$, each representing a user ***session*** accessing the store concurrently with others; a session is thus an analogue of a thread in shared-memory languages. An implementation would connect each session to one of the store replicas (as in examples in Figure 1), but this is transparent on the language level. Data type variables declared in P are used to specify the types of objects declared afterwards, and object variables are used inside sessions C_j, as per the typing rules. Sessions can thus invoke operations on a number of objects of primitive or composite types. By default, every such operation is executed within a dedicated transaction. However, like in composite data type implementations, we allow sessions to group multiple operations into transactions using atomic blocks included into the syntax of commands. We consider data types T and programs P up to the standard alpha-equivalence, adjusted so that v_{in} and v_{out} are not renamed.

Technical Restriction. To simplify definitions, we assume that commands inside atomic blocks always terminate and, thus, so do all operations of composite data types. We formalise this restriction when presenting the semantics of the language in §4. It can be lifted at the expense of complicating the presentation. Note that the sessions C_j do not have to terminate, thereby allowing us to model the reactive nature of store clients.

3 Replicated Store Semantics

A replicated store holds objects of primitive replicated data types and implements operations on these objects. The language of §2 allows us to write programs that interact with the store by invoking the operations while grouping primitive objects into composite ones to achieve modularity. The main contribution of this paper is a denotational semantics of the language that allows the reasoning about a program to reflect this modularity. But before presenting it (in §4), we need to define the semantics of the store itself: which values can operations on primitive objects return in an execution of the store? This is determined by the consistency model of causally consistent transactions [25, 17, 18, 23, 16, 10, 3], which we informally described in §1. To formalise it, we use a variant of the framework proposed by Burckhardt et al. [9, 10, 8], which defines the store semantics declaratively, without referring to implementation-level concepts such as replicas or messages. The framework models store executions using structures on events and relations in the style of weak memory models and allows us to define the semantics of the store in two stages. We first specify the semantics of single operations on primitive objects using *replicated data type specifications* (§3.1), which are certain functions on events and relations. We then specify allowed executions of the store, including multiple operations on different objects, by constraining the events and relations using *consistency axioms* (§3.2).

A correspondence between the declarative store specification and operational models closer to implementations was established elsewhere [9, 10, 8]. Although we do not present an operational model in this paper, we often explain various features of the store specification framework by referring to the implementation-level concepts they are meant to model.

The granularity abstraction of the denotational semantics we define in §4 allows us to pretend that a composite data type is a primitive one. Hence, when defining the semantics, we reuse the replicated data type specifications introduced here to specify the behaviour of a composite data type, such as the one in Figure 3, while abstracting from the internals of its implementation.

3.1 Semantics of Primitive Replicated Data Types

In a strongly consistent system, there is a total order on all operations on an object, and each operation takes into account the effects of all operations preceding it in this order. In an eventually consistent system, the result of an operation o is determined in a more complex way:

1. The result of o depends on the set of operations information about which has been delivered to the replica performing o—those *visible* to o. For example, in Figure 1(a) the operation $\mathit{friends}[a].\mathtt{get}$ returns $\emptyset$ because the message about $\mathit{friends}[a].\mathtt{add}(b)$ has not yet been delivered to the replica performing the `get`.
2. The result of o may also depend on additional information used to order some events. For example, we may decide to order concurrent updates to an object using timestamps, as is the case when we use the last-writer-wins conflicts resolution policy mentioned in §1.

Hence, we specify the semantics of a replicated data type by a function F that computes the return value of an operation o given its *operation context*, which includes all we need to know about the store execution to determine the value: the set of events visible to o, together with a pair of relations on them that specify the above relationships.

Assume a countably-infinite set Event of ***events***, representing operations issued to the store. A relation is a ***strict partial order*** if it is transitive and irreflexive. A ***total order*** is a strict partial order such that for every two distinct elements e and f, the order relates e to f or f to e. We call a pair $p \in \mathsf{Op} \times \mathsf{Val} = \mathsf{AOp}$ of an operation o together with its parameter a an ***applied operation***, written as $o(a)$.

DEFINITION 1 *An **operation context** is a tuple $N = (p, E, \mathsf{aop}, \mathsf{vis}, \mathsf{ar})$, where $p \in \mathsf{AOp}$, E is a finite subset of Event, $\mathsf{aop} : E \to \mathsf{AOp}$, and vis (**visibility**) and ar (**arbitration**) are strict partial orders on E such that $\mathsf{vis} \subseteq \mathsf{ar}$.*

*We call the tuple $M = (E, \mathsf{aop}, \mathsf{vis}, \mathsf{ar})$ a **partial operation context**.*

We write Ctxt for the set of all operation contexts and denote components of N and similar structures as in $N.E$. For a relation R we write $(e, f) \in R$ and $e \xrightarrow{R} f$ interchangeably. Informally, the orders vis and ar record the relationships between events in E motivated by the above points 1 and 2, respectively. In implementation terms, the requirement $\mathsf{vis} \subseteq \mathsf{ar}$ guarantees that timestamps are consistent with message delivery: if e is visible to f, then e has a lower timestamp than f. We define where vis and ar come from formally in §3.2; for now we just assume that they are given and define *replicated data type specifications* as certain functions of operation contexts including them.

DEFINITION 2 *A **replicated data type specification** is a partial function $F : \mathsf{Ctxt} \rightharpoonup \mathsf{Val}$ that returns the same value on isomorphic operation contexts and preserves it on arbitration extensions. Formally, let us order operation contexts by the pre-order $\sqsubseteq$:*

$$(p, E, \mathsf{aop}, \mathsf{vis}, \mathsf{ar}) \sqsubseteq (p', E', \mathsf{aop}', \mathsf{vis}', \mathsf{ar}') \iff$$
$$p = p' \wedge \exists \pi \in E \to_{\text{bijective}} E'.\, \pi(\mathsf{aop}) = \mathsf{aop}' \wedge \pi(\mathsf{vis}) = \mathsf{vis}' \wedge \pi(\mathsf{ar}) \subseteq \mathsf{ar}',$$

where we use the expected lifting of π to relations. Then we require

$$\forall N, N' \in \mathsf{Ctxt}.\, N \sqsubseteq N' \wedge N \in \mathsf{dom}(F) \implies N' \in \mathsf{dom}(F) \wedge F(N) = F(N'). \quad (3)$$

Let Spec be the set of data type specifications F and assume a fixed F_B for every primitive type $B \in \mathsf{PrimType}$ provided by the store. The requirement (3) states that, once arbitration gives all the information that is needed in addition to visibility to determine the outcome of an operation, arbitrating more events does not change this outcome.

Replicated Sets. We illustrate the above definitions by specifying replicated set data types with different conflict-resolution policies. The semantics of a replicated set is straightforward when it is ***add-only***, i.e., its signature is $\{\mathtt{add}, \mathtt{contains}, \mathtt{get}\}$. An element a is in the set if there is an $\mathtt{add}(a)$ event in the context, or informally, if the replica performing $\mathtt{contains}(a)$ has received a message about the addition of a:

$$F_{\mathtt{AOset}}(\mathtt{contains}(a), E, \mathsf{aop}, \mathsf{vis}, \mathsf{ar}) \;=\; (\exists e \in E.\, \mathsf{aop}(e) = \mathtt{add}(a)).$$

We define the result to be $\bot$ for $\mathtt{add}$ operations and define the result of $\mathtt{get}$ as expected.[4]

Things become more subtle if we allow removing elements, since we need to define the outcome of concurrent operations adding and removing the same element, as in the context $N = (\mathtt{contains}(42), \{e, f\}, \mathsf{aop}, \mathsf{vis}, \mathsf{ar})$, where $\mathsf{aop}(e) = \mathtt{add}(42)$ and $\mathsf{aop}(f) = \mathtt{remove}(42)$. There are several possible ways of resolving this conflict [6]: in ***add-wins sets*** ($\mathtt{AWset}$) adds always win against concurrent removes (so that the element ends up in the set), ***remove-wins sets*** ($\mathtt{RWset}$) act vice versa, and ***last-writer-wins sets*** ($\mathtt{LWWset}$) apply operations in the order of their timestamps. We specify the result of $\mathtt{contains}$ in these cases using the vis and ar orders in the operation context:

$$\begin{aligned}
&F_{\mathtt{AWset}}(\mathtt{contains}(a), E, \mathsf{aop}, \mathsf{vis}, \mathsf{ar}) = \\
&\quad \exists e \in E.\, \mathsf{aop}(e) = \mathtt{add}(a) \wedge (\forall f \in E.\, \mathsf{aop}(f) = \mathtt{remove}(a) \implies \neg(e \xrightarrow{\mathsf{vis}} f)); \\
&F_{\mathtt{RWset}}(\mathtt{contains}(a), E, \mathsf{aop}, \mathsf{vis}, \mathsf{ar}) = \\
&\quad \exists e \in E.\, \mathsf{aop}(e) = \mathtt{add}(a) \wedge (\forall f \in E.\, \mathsf{aop}(f) = \mathtt{remove}(a) \implies f \xrightarrow{\mathsf{vis}} e); \\
&F_{\mathtt{LWWset}}(\mathtt{contains}(a), E, \mathsf{aop}, \mathsf{vis}, \mathsf{ar}) = \\
&\quad \exists e \in E.\, \mathsf{aop}(e) = \mathtt{add}(a) \wedge (\forall f \in E.\, \mathsf{aop}(f) = \mathtt{remove}(a) \implies f \xrightarrow{\mathsf{ar}} e), \\
&\quad \text{if } \mathsf{ar} \text{ is total on } \{e \in E \mid \mathsf{aop}(e) \in \{\mathtt{add}(\_), \mathtt{remove}(\_)\}\}; \\
&F_{\mathtt{LWWset}}(\mathtt{contains}(a), E, \mathsf{aop}, \mathsf{vis}, \mathsf{ar}) = \text{undefined, otherwise.}
\end{aligned}$$

Thus, the add-wins semantics is formalised by mandating that $\mathtt{remove}$ operations cancel only the $\mathtt{add}$ operations that are visible to them; the remove-wins semantics additionally mandates that they cancel concurrent $\mathtt{add}$ operations, but not those that follow them in visibility. On the above context N, the operation $\mathtt{contains}(42)$ returns $\mathtt{true}$ iff: $\neg(e \xrightarrow{\mathsf{vis}} f)$ for $\mathtt{AWset}$; $f \xrightarrow{\mathsf{vis}} e$ for $\mathtt{RWset}$; and $f \xrightarrow{\mathsf{ar}} e$ for $\mathtt{LWWset}$. As we show in §5, using a remove-wins set for *requesters* in Figure 3 is crucial for preserving the integrity invariant (2); *friends* could well be add-wins, which would lead to different, but also sensible, data type behaviour.

3.2 Whole-Store Semantics

We define the semantics of a causally consistent store by the set of its *histories*, which are certain structures on events recording all client-store interactions that can be pro-

[4] $F_{\mathtt{AOset}}$ is undefined on contexts with operations other than those from the signature. The type system of our language ensures that such contexts do not arise in its semantics.

duced during a run of the store; these include operations invoked on all objects and their return values. The store has no control over the operations occurring in histories, since these are chosen by the client; hence, the semantics only constrains return values. Replicated data type specifications define return values of operations in terms of visibility and arbitration, but where do these orders come from? As we explained in §3.1, intuitively, they are determined by the way messages are delivered and timestamps assigned in a run of a store implementation. Since this highly non-deterministic, in general, visibility and arbitration orders are arbitrary, but not entirely. A causally consistent store provides to its clients a guarantee that these orders in the contexts of different operations in the same run are related in certain ways, and this guarantee disallows anomalies such as the one in Figure 1(a).

We formalise the guarantee using the notion of an *execution*, which extends a history with visibility and arbitration orders on its events. A history is allowed by the store semantics if there is a way to extend it to an execution such that: *(i)* the return values of operations in the execution are obtained by applying replicated data type specifications to contexts extracted from it; and *(ii)* the execution satisfies certain *consistency axioms*, which constrain visibility and arbitration and, therefore, operation contexts.

Histories, Executions and the Satisfaction of Data Type Specifications. We identify objects (primitive or composite) by elements of the set Obj, ranged over by ω. A strict partial order R is ***prefix-finite*** if $\{f \mid (f, e) \in R\}$ is finite for every e.

DEFINITION 3 *A **history** is a tuple* $H = (E, \mathsf{label}, \mathsf{so}, \sim)$*, where:*

- $E \subseteq \mathsf{Event}$.
- $\mathsf{label} : E \to \mathsf{Obj} \times \mathsf{AOp} \times \mathsf{Val}$ *describes the events in E: if* $\mathsf{label}(e) = (\omega, p, a)$*, then the event* e *describes the applied operation* p *on the object* ω *returning the value* a*.*
- $\mathsf{so} \subseteq E \times E$ *is a **session order**, ordering events in the same session according to the order in which they were submitted to the store. We require that* so *be prefix-finite and be the union of finitely many total orders defined on disjoint subsets of E, which correspond to events in different sessions.*
- $\sim \subseteq E \times E$ *is an equivalence relation grouping events in the same transaction. Since all transactions terminate (§2), we require that every equivalence class of* $\sim$ *be a finite set. Since every transaction is performed by a single session, we require that any two distinct events by the same transaction be related by* so *one way or another:*

$$\forall e, f.\, e \sim f \wedge e \neq f \implies e \xrightarrow{\mathsf{so}} f \vee f \xrightarrow{\mathsf{so}} e.$$

We also require that a transaction be contiguous in so*:*

$$\forall e, f, g.\, e \xrightarrow{\mathsf{so}} f \xrightarrow{\mathsf{so}} g \wedge e \sim g \implies e \sim f \sim g.$$

*An **execution** is a triple* $X = (H, \mathsf{vis}, \mathsf{ar})$ *of a history* H *and prefix-finite strict partial orders* vis *and* ar *on* $H.E$*, such that* $\mathsf{vis} \cup \mathsf{ar} \subseteq \{(e, f) \mid H.\mathsf{obj}(e) = H.\mathsf{obj}(f)\}$ *and* $\mathsf{vis} \subseteq \mathsf{ar}$.

We denote the sets of all histories and executions by Hist and Exec. We write $H.\mathsf{obj}(e)$, $H.\mathsf{aop}(e)$ and $H.\mathsf{rval}(e)$ for the components of $H.\mathsf{label}(e)$ and shorten, e.g., $X.H.\mathsf{so}$

to $X.\mathsf{so}$. Note that the set $H.E$ can be infinite, which models infinite runs. Figure 1(a) graphically represents an execution corresponding to the causality violation anomaly explained in §1. The relation $\sim$ is an identity in this case, and the objects in this and other executions in Figure 1 are add-only sets (`AOset`, §3.1).

Given an execution X, we extract the operation context of an event $e \in X.E$ by selecting all events visible to it according to $X.\mathsf{vis}$:

$$\mathsf{ctxt}(X, e) = (X.\mathsf{aop}(e), E, (X.\mathsf{aop})|_E, (X.\mathsf{vis})|_E, (X.\mathsf{ar})|_E), \tag{4}$$

where $E = (X.\mathsf{vis})^{-1}(e)$ and $\cdot|_E$ is the restriction to events in E. Then, given a function $\mathbb{F} : \mathsf{Obj} \rightharpoonup \mathsf{Spec}$ that associates data type specifications with some objects, we say that an execution X satisfies $\mathbb{F}$ if the return value of every event in X is computed on its context according to the specification that $\mathbb{F}$ gives for the accessed object.

DEFINITION 4 *An execution X **satisfies** $\mathbb{F}$, written $X \models \mathbb{F}$, if*

$$\forall e \in X.E.\, (X.\mathsf{obj}(e) \in \mathsf{dom}(\mathbb{F}) \implies X.\mathsf{rval}(e) = \mathbb{F}(X.\mathsf{obj}(e))(\mathsf{ctxt}(X, e))).$$

Since a context does not include return values, the above equation determines them uniquely for the events e satisfying the premise. For example, in the execution in Figure 1(a) the context of the `get` from ω_{fa} is empty. Hence, to satisfy $\mathbb{F} = (\lambda\omega.\, F_{\mathtt{AOset}})$, the `get` returns $\emptyset$. If we had a vis edge from the `add`(b) to the `get`, then the latter would have to return $\{b\}$.

Consistency Axioms. We now formulate additional constraints that executions have to satisfy. They restrict the anomalies allowed by the consistency model we consider and, in particular, rule out the execution in Figure 1(a).

To define the semantics of transactions, we use the following operation. For a relation R on a set of events E and an equivalence relation $\sim$ on E (meant to group events in the same transaction), we define the ***factoring*** $R/{\sim}$ of R over $\sim$ as follows:

$$R/{\sim} \;=\; R \cup ((\sim; R; \sim) - (\sim)), \tag{5}$$

where ; composes relations. Thus, $R/{\sim}$ includes all edges from R and those obtained from such edges by relating any actions coming from the same transactions as their endpoints, excluding the case when the endpoints themselves are from the same transaction. We also let $\mathsf{sameobj}(X)(e, f) \iff X.\mathsf{obj}(e) = X.\mathsf{obj}(f)$.

DEFINITION 5 *An execution $X = ((E, \mathsf{label}, \mathsf{so}, \sim), \mathsf{vis}, \mathsf{ar})$ is **causally consistent** if it satisfies the following **consistency axioms**:*

CAUSALVIS. $((\mathsf{so} \cup \mathsf{vis})/{\sim})^+ \cap \mathsf{sameobj}(X) \subseteq \mathsf{vis}$*;*
CAUSALAR. $(\mathsf{so} \cup \mathsf{ar})/{\sim}$ *is acyclic;*
EVENTUAL. $\forall e \in E.\, \left|\{f \in E \mid \mathsf{sameobj}(X)(e, f) \wedge \neg(e \xrightarrow{\mathsf{vis}} f)\}\right| < \infty$.

We write $X \models_{\mathsf{CC}} \mathbb{F}$ if $X \models \mathbb{F}$ and X is causally consistent.

The axioms follow the informal description of the consistency model we gave in §1. We explain them below; however, their details are not crucial for understanding the rest of the paper. Before explaining the axioms, we note that Definitions 4 and 5 allow us to define the semantics of a store with object specifications given by $\mathbb{F} : \mathsf{Obj} \rightharpoonup \mathsf{Spec}$ as the set of histories that can be extended to a causally consistent execution satisfying $\mathbb{F}$:

$$\mathsf{HistCC}(\mathbb{F}) = \{H \mid \exists \mathsf{vis}, \mathsf{ar}.\, (H, \mathsf{vis}, \mathsf{ar}) \models_{\mathsf{CC}} \mathbb{F}\}. \qquad (6)$$

To prove that a particular store implementation satisfies this specification, for every history H the implementation produces we have to come up with vis and ar that satisfy the constraint in (6); this is usually done by constructing them from message delivery and timestamps in the run of the implementation producing H. Here we rely on previous correctness proofs of store implementations [9, 10, 8] and use the above declarative specification of the store semantics without fixing the store implementation.

Causal Consistency. The axioms CAUSALVIS and CAUSALAR in Definition 5 ensure that visibility and arbitration respect causality between operations. CAUSALVIS guarantees that an event sees all events on the same object that causally affect it, i.e., those preceding it in a chain of session order and visibility edges (ignore the use of factoring over $\sim$ for now). Thus, CAUSALVIS disallows the execution in Figure 1(a). CAUSALAR similarly requires that arbitration be consistent with session order on all objects (recall that $X.\mathsf{vis} \subseteq X.\mathsf{ar}$). EVENTUAL formalises the liveness property that every replica eventually sees every update: it ensures that an event cannot be invisible to infinitely many other events on the same object.

Transactions. The use of factoring over the $\sim$ relation in CAUSALVIS formalises the guarantee provided by causally consistent transactions that we noted in §1: updates done by a transaction get delivered to replicas together. According to CAUSALVIS, a causal dependency established between two actions of different transactions results in a dependency also being established between any other actions in the two transactions. Thus, CAUSALVIS disallows the execution in Figure 1(c), where the dashed rectangles group events into transactions. The axioms allow the execution in Figure 1(b) even when the operations by the same session are done within a transaction—an outcome that would not be allowed with serialisable transactions.

4 Coarse-grained Language Semantics

We now describe our main contribution—a coarse-grained denotational semantics of programs in the language of §2 that enables modular reasoning. We establish a correspondence between this semantics and the reference fine-grained semantics in §6.

4.1 Session-Local Semantics of Commands

The semantics of the replicated store defined by (6) in §3 describes the store behaviour under any client and thus produces histories with all possible sets of client operations. However, a particular command C in the language of §2 generates only histories with certain sequences of operations. Thus, our first step is to define a ***session-local*** semantics that, for each (sequential) command C, gives the set of histories that C can possibly generate. This semantics takes into account only the structure of the command C and

$$\langle \Delta \mid \Sigma \vdash C\rangle \ :\ (\mathsf{dom}(\Delta) \to_{\mathsf{inj}} \mathsf{Obj}) \times \mathsf{LState}(\Sigma) \to \mathcal{P}((\mathsf{FHist} \times \mathsf{LState}(\Sigma)) \cup \mathsf{IHist})$$

$$\langle v = G\rangle(obj, \sigma) = \{(H_{\mathrm{emp}}, \sigma[v \mapsto [\![G]\!]\sigma]) \mid H_{\mathrm{emp}} = (\emptyset, [\,], \emptyset, \emptyset)\}$$

$$\langle v = x.o(G)\rangle(obj, \sigma) = \{(H_e, \sigma[v \mapsto a]) \mid e \in \mathsf{Event} \wedge a \in \mathsf{Val} \wedge H_e = (\{e\}, [e \mapsto (obj(x), o([\![G]\!]\sigma), a)], \emptyset, \{(e, e)\})\}$$

$$\langle \mathsf{atomic}\ \{C\}\rangle(obj, \sigma) = \{((E, \mathsf{label}, \mathsf{so}, E \times E), \sigma') \mid ((E, \mathsf{label}, \mathsf{so}, \sim), \sigma') \in \langle C\rangle(obj, \sigma)\}$$

Fig. 4. Key clauses of the session-local semantics of commands. Here FHist and IHist are respectively sets of histories with finite and infinite event sets; $\sigma[v \mapsto a]$ denotes the function that has the same value as σ everywhere except v, where it has the value a; and $[\,]$ is a nowhere-defined function. We assume a standard semantics of expressions $[\![G]\!] : \mathsf{LState}(\Sigma) \to \mathsf{Val}$.

operations on local variables; the return values of operations executed on objects in the store are chosen arbitrarily. Later (§4.3), we intersect the set of histories produced by the session-local semantics with (6) to take the store semantics into account.

To track the values of local variables Σ in the session-local semantics of a command $\Delta \mid \Sigma \vdash C$ (Figure 2), we use ***local states*** $\sigma \in \mathsf{LState}(\Sigma) = \Sigma \to \mathsf{Val}$. The semantics interprets commands by the function $\langle \Delta \mid \Sigma \vdash C\rangle$ in Figure 4. Its first parameter obj determines the identities of objects bound to object variables in Δ. Given an initial local state σ as the other parameter, $\langle \Delta \mid \Sigma \vdash C\rangle$ returns the set of histories produced by C when run from σ, together with final local states when applicable. The semantics is mostly standard and therefore we give only key clauses; see [13, §A] for the remaining ones. Recall that, to simplify our formalism, we require every transaction to terminate (§2). To formalise this assumption, the clause for atomic filters out infinite histories.

4.2 Composite Data Type Semantics

The distinguishing feature of our coarse-grained semantics is its support for granularity abstraction: the denotation of a composite data type abstracts from its internal structure. Technically, this means that composite data types are interpreted in terms of replicated data type specifications, which we originally used for describing the meaning of primitive data types (§3.1). Thus, type variable environments Γ and data types $\Gamma \vdash T : O$ (Figure 2) are interpreted over the following domains:

$$[\![\Gamma]\!] = \mathsf{dom}(\Gamma) \to \mathsf{Spec}; \qquad [\![\Gamma \vdash T : O]\!] \in [\![\Gamma]\!] \to \mathsf{Spec}.$$

We use $type$ to range over elements of $[\![\Gamma]\!]$. Two cases in the definition of $[\![\Gamma \vdash T : O]\!]$ are simple. We interpret a primitive data type $B \in \mathsf{PrimType}$ as the corresponding data type specification F_B, which is provided as part of the store specification (§3.1): $[\![B]\!]type = F_B$. We define the denotation of a type variable α by looking it up in the environment $type$: $[\![\alpha]\!]type = type(\alpha)$.

The remaining and most interesting case is the interpretation $[\![\Gamma \vdash D : O]\!]$ of a composite data type

$$D \ = \ \mathsf{let}\ \{x_j = \mathsf{new}\ T_j\}_{j=1..m}\ \mathsf{in}\ \{o = \mathsf{atomic}\ \{C_o\}\}_{o \in O}. \qquad (7)$$

For $type \in [\![\Gamma]\!]$, the data type specification $F = [\![\Gamma \vdash D : O]\!]type$ returns a value given a context consisting of ***coarse-grained*** events that represent composite operations on

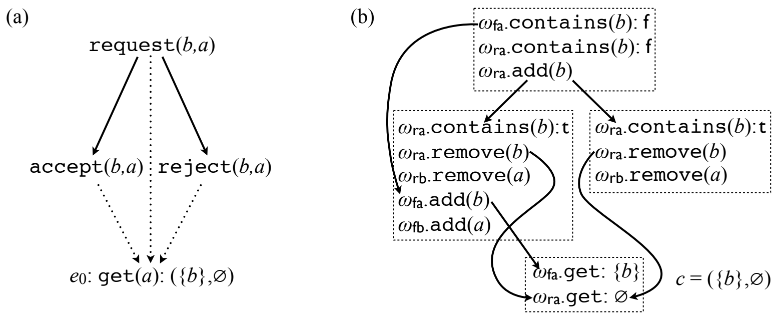


Fig. 5. (a) A context N of coarse-grained events for the social graph data type soc in Figure 3, with an event e_0 added to represent the operation $N.p$. Solid edges denote both visibility and arbitration (equal, since the data type does not use arbitration). The dashed edges show the additional edges in vis' and ar' introduced in Definition 7. **(b)** An execution X belonging to the concretisation of N. The objects ω_{fa}, ω_{fb}, ω_{ra}, ω_{rb} correspond to the variables $\mathit{friends}[a]$, $\mathit{friends}[b]$, $\mathit{requesters}[a]$, $\mathit{requesters}[b]$ of type RWset. Solid edges denote both visibility and arbitration. We have omitted the session order inside transactions, the visibility and arbitration edges it induces and the transitive consequences of the edges shown. Dashed rectangles group events into transactions. The function β maps events in X to the horizontally aligned events in N.

an object of type D (e.g., the one in Figure 5(a)). This achieves granularity abstraction, because, once a denotation of this form is computed, it can be used to determine the return value of a composite operation without knowing the operations on the constituent objects x_j that were done by the implementations C_o of the composite operations in its context (e.g., the ones in Figure 3). We call events describing the operations on x_j ***fine-grained***.

Informally, our approach to defining the denotation F of D is to determine the value that F has to return on a context N of coarse-grained events by "running" the implementations C_o of the composite operations invoked in N. This produces an execution X over fine-grained events that describes how C_o acts on the constituent objects x_j—a *concretisation* of N. The execution X has to be causally consistent and satisfy the data type specifications for the objects x_j. We then define $F(N)$ to be the return value that the implementation of the composite operation $N.p$ gives in X. However, concretising N into X is easier said than done: while the history part of X is determined by the session-local semantics of the implementations C_o (§4.1), determining the visibility and arbitration orders so that the resulting denotation be sound (in the sense described in §6) is nontrivial and represents our main insight.

To define the denotation of (7) formally, we first gather all histories that an implementation C_o of a composite operation can produce in the session-local semantics $\langle\cdot\rangle$ into a *summary*: given an applied composite operation and a return value, a summary defines the set of histories that its implementation produces when returning the value.

DEFINITION 6 *A **summary** ρ is a partial map $\rho : \mathsf{AOp} \times \mathsf{Val} \rightharpoonup \mathcal{P}(\mathsf{FHist})$ such that for every $(p, a) \in \mathsf{dom}(\rho)$, $\rho(p, a)$ is closed under the renaming of events, and for every $H \in \rho(p, a)$, $H.\mathsf{so}$ is a total order on $H.E$ and $H.\sim = H.E \times H.E$.*

For a family of commands $\{\Delta \mid v_{\mathsf{in}}, v_{\mathsf{out}} \vdash C_o\}_{o \in O}$ and $obj : \mathsf{dom}(\Delta) \to_{\mathsf{inj}} \mathsf{Obj}$, we define the corresponding summary $[\![\{C_o\}_{o\in O}]\!](obj) : \mathsf{AOp} \times \mathsf{Val} \rightharpoonup \mathcal{P}(\mathsf{FHist})$ as follows: for $o' \in O$ and $a, b \in \mathsf{Val}$, we let

$$[\![\{C_o\}_{o\in O}]\!](obj)(o'(a), b) =$$
$$\{H \mid (H, [v_{\mathsf{in}} \mapsto \_, v_{\mathsf{out}} \mapsto b]) \in \langle \mathsf{atomic}\ \{C_{o'}\}\rangle(obj, [v_{\mathsf{in}} \mapsto a, v_{\mathsf{out}} \mapsto \bot])\}.$$

For example, the method bodies C_o in Figure 3 and an appropriate obj define the summary $\rho_{\mathsf{soc}} = [\![\{C_o\}_{o\in\{\mathtt{request},\mathtt{accept},\ldots\}}]\!](obj)$. This maps the get operation in Figure 5(a) to a set of histories including the one shown to the right of it in Figure 5(b).

We now define the executions X that may result from "running" the implementations of composite operations in a coarse-grained context N given by a summary ρ. The definition below pairs these executions X with the value c returned in them by the implementation of $N.p$, since this is what we are ultimately interested in. We first state the formal definition, and then explain it in detail. We write id for the identity relation.

DEFINITION 7 *A pair $(X, c) \in \mathsf{Exec} \times \mathsf{Val}$ is a* ***concretisation*** *of a context N with respect to a summary $\rho : \mathsf{AOp} \times \mathsf{Val} \rightharpoonup \mathcal{P}(\mathsf{FHist})$ if for some event $e_0 \notin N.E$ and function $\beta : X.E \to N.E \uplus \{e_0\}$ we have*

$$(\forall f \in (N.E).\ (X.H)|_{\beta^{-1}(f)} \in \rho(N.\mathsf{aop}(f), \_)) \wedge ((X.H)|_{\beta^{-1}(e_0)} \in \rho(N.p, c)); \quad (8)$$
$$\beta(X.\mathsf{so}) \subseteq \mathsf{id}; \quad (9)$$
$$\beta(X.\mathsf{vis}) - \mathsf{id} \subseteq \mathsf{vis}'; \quad (10)$$
$$\beta^{-1}(\mathsf{vis}') \cap \mathsf{sameobj}(X) \subseteq X.\mathsf{vis}; \quad (11)$$
$$\beta(X.\mathsf{ar}) - \mathsf{id} \subseteq \mathsf{ar}', \quad (12)$$

where $\mathsf{vis}' = N.\mathsf{vis} \cup \{(f, e_0) \mid f \in N.E\}$ *and* $\mathsf{ar}' = N.\mathsf{ar} \cup \{(f, e_0) \mid f \in N.E\}$.

We write $\gamma(N, \rho)$ for the set of all concretisations of N with respect to ρ.

For example, the pair of the execution and the value in Figure 5(b) belongs to $\gamma(N, \rho_{\mathsf{soc}})$ for N in Figure 5(a). When X concretises N with respect to ρ, the history $X.H$ is a result of expanding every composite operation in N into a history of its implementation according to ρ. The function β maps every event in $X.E$ to the event from N it came from, with an event e_0 added to $N.E$ to represent the operation $N.p$; this is formalised by (8). The condition (9) further requires that the implementation of every composite operation be executed in a dedicated session. As it happens, it is enough to consider concretisations of this form to define the denotation.

The conditions (10)–(12) represent the main insight of our definition of the denotation: they tell us how to select the visibility and arbitration orders in X given those in N. They are best understood by appealing to the intuition about how an implementation of the store operates. Recall that, from this perspective, visibility captures message delivery: an event is visible to another event if and only if the information about the former has been delivered to the replica of the latter (§3.1). Also, in implementations of causally consistent transactions, updates done by a transaction are delivered to every replica together (§1). Since composite operations execute inside transactions, the visibility order in N can thus be intuitively thought of as specifying the delivery of groups of updates made by them: we have an edge $e' \xrightarrow{\mathsf{vis}'} f'$ between coarse-grained events e' and f' in N (e.g., `request` and `accept` in Figure 5(a)) if and only if the updates performed by the transaction denoted by e' have been delivered to the replica of f'. Now consider fine-grained events $e, f \in X.E$ on the same constituent object describing updates made inside the transactions of e' and f', so that $\beta(e) = e'$ and $\beta(f) = f'$ (e.g., $\omega_{\mathsf{ra}}.\mathtt{add}(b)$ and $\omega_{\mathsf{ra}}.\mathtt{contains}(b)$ in Figure 5(b)). Then we can have $e \xrightarrow{X.\mathsf{vis}} f$ if and only if $e' \xrightarrow{\mathsf{vis}'} f'$. This is formalised by (10) and (11).

To explain (12), recall that arbitration captures the order of timestamps assigned to events by the store implementation. Also, in implementations the timestamps of all updates done by a transaction are contiguous in this order. Thus, arbitration in N can be thought of as specifying the timestamp order on the level of whole transactions corresponding to the composite operations in N. Then (12) states that the order of timestamps of fine-grained events in X is consistent with that over transactions these events come from.

To define the denotation, we need to consider only those executions concretising N that are causally consistent and satisfy data type specifications. Hence, for $\mathbb{F} : \mathsf{Obj} \rightharpoonup \mathsf{Spec}$ we let

$$\gamma(N, \rho, \mathbb{F}) = \{(X, c) \in \gamma(N, \rho) \mid X \models_{\mathsf{CC}} \mathbb{F}\}.$$

For example, the execution in Figure 5(b) belongs to $\gamma(N, \rho_{\mathsf{soc}}, \mathbb{F})$ for N in Figure 5(a) and $\mathbb{F} = (\lambda\omega.\, F_{\mathtt{RWset}})$. As the following theorem shows, the constraints (8)–(12) are so tight that the set of concretisations defined in this way never contains two different return values; this holds even if we allow choosing object identities differently.

THEOREM 8 *Given a family* $\{\Delta \mid v_{\mathsf{in}}, v_{\mathsf{out}} \vdash C_o\}_{o \in O}$, *we have:*

$$\begin{array}{l}
\forall N.\, \forall obj_1, obj_2 \in [\mathsf{dom}(\Delta) \to_{\mathsf{inj}} \mathsf{Obj}].\\
\forall \mathbb{F}_1 \in [\mathsf{range}(obj_1) \to \mathsf{Spec}].\, \forall \mathbb{F}_2 \in [\mathsf{range}(obj_2) \to \mathsf{Spec}].\\
\quad (\forall x \in \mathsf{dom}(\Delta).\, \mathbb{F}_1(obj_1(x)) = \mathbb{F}_2(obj_2(x))) \implies\\
\quad\quad \forall (X_1, c_1) \in \gamma(N, [\![\{C_o\}_{o \in O}]\!](obj_1), \mathbb{F}_1).\\
\quad\quad \forall (X_2, c_2) \in \gamma(N, [\![\{C_o\}_{o \in O}]\!](obj_2), \mathbb{F}_2).\, c_1 = c_2.
\end{array}$$

This allows us to define the denotation of (7) according to the outline we gave before.

DEFINITION 9 *For (7) we let* $[\![\Gamma \vdash D]\!]type = F$, *where* $F : \mathsf{Ctxt} \rightharpoonup \mathsf{Val}$ *is defined as follows: for* $N \in \mathsf{Ctxt}$ *and* $c \in \mathsf{Val}$, *if*

$$\begin{array}{l}
\exists obj \in [\{x_j \mid j = 1..m\} \to_{\mathsf{inj}} \mathsf{Obj}].\, \exists \mathbb{F} \in [\mathsf{range}(obj) \to \mathsf{Spec}].\\
\quad (\forall j = 1..m.\, \mathbb{F}(obj(x_j)) = [\![T_j]\!]type) \wedge (\_, c) \in \gamma(N, [\![\{C_o\}_{o \in O}]\!](obj), \mathbb{F}),
\end{array}$$

then $F(N) = c$; *otherwise* $F(N)$ *is undefined.*

The existence and uniqueness of F in the definition follow from Theorem 8. It is easy to check that F defined above satisfies all the properties required in Definition 2 and, hence, $F \in \mathsf{Spec}$. According to the above definition, the denotation of the data type in Figure 3 has to give $(\{b\}, \emptyset)$ on the context in Figure 5(a).

$$
\begin{aligned}
[\![\mathit{\Gamma} \mid \mathit{\Delta} \vdash P]\!] \;:\; [\![\mathit{\Gamma}]\!] &\to \textstyle\prod_{obj \in [\mathsf{dom}(\mathit{\Delta}) \to_{\mathsf{inj}} \mathsf{Obj}]} ((\mathsf{range}(obj) \rightharpoonup \mathsf{Spec}) \to \mathcal{P}(\mathsf{Hist})) \\
[\![\mathsf{let}\ \alpha = T\ \mathsf{in}\ P]\!](type, obj, \mathbb{F}) &= [\![P]\!](type[\alpha \mapsto [\![T]\!]type], obj, \mathbb{F}) \\
[\![\mathsf{let}\ x = \mathsf{new}\ T\ \mathsf{in}\ P]\!](type, obj, \mathbb{F}) &= \bigcup\{[\![P]\!](type, obj[x \mapsto \omega], \mathbb{F}[\omega \mapsto [\![T]\!]type]) \mid \\
&\qquad\qquad\qquad\qquad \omega \notin \mathsf{range}(obj)\} \\
[\![C_1 \parallel \ldots \parallel C_n]\!](type, obj, \mathbb{F}) &= \mathsf{HistCC}(\mathbb{F}) \cap \{\, \textstyle\biguplus_{j=1}^{n} H_j \mid \forall j = 1..n. \\
&\qquad (H_j, \_) \in \langle C_j \rangle(obj, [\,]) \vee H_j \in \langle C_j \rangle(obj, [\,])\}
\end{aligned}
$$

Fig. 6. Semantics of $\mathit{\Gamma} \mid \mathit{\Delta} \vdash P$. Here $H \uplus H' = (H.E \uplus H'.E,\ H.\mathsf{label} \uplus H'.\mathsf{label},\ H.\mathsf{so} \cup H'.\mathsf{so},\ H.{\sim} \cup H'.{\sim})$; undefined if so is $H.E \uplus H'.E$.

4.3 Program Semantics

Having defined the denotations of composite data types, we give the semantics to a program in the language of §2 by instantiating (6) with an $\mathbb{F}$ computed from these denotations and by intersecting the result with the set of histories that can be produced by the program according to the session-local semantics of its sessions (§4.1). A program $\mathit{\Gamma} \mid \mathit{\Delta} \vdash P$ is interpreted with respect to environments $type$, obj and $\mathbb{F}$, which give the semantics of data type variables in $\mathit{\Gamma}$, the identities of objects in $\mathit{\Delta}$ and the specifications associated with these objects (Figure 6). A data type variable declaration extends the $type$ environment with the specification of the data type computed from its declaration as described in §4.2. An object variable declaration extends obj with a fresh object and $\mathbb{F}$ with the specification corresponding to its type. A client is interpreted by combining all histories its sessions produce in the session-local semantics with respect to obj and intersecting the result with (6). Note that we originally defined the store semantics (6) under the assumption that all replicated data types are primitive. Here we are able to reuse the definition because our denotations of composite data types have the same form as those of primitive ones.

Using the Semantics. Our denotational semantics enables modular reasoning about programs with composite replicated data types. Namely, it allows us to check if a program P can produce a given history H by: *(i)* computing the denotations $\mathbb{F}$ of the composite data types used in P; and *(ii)* checking if the client of P can produce H assuming it uses *primitive* data types with the specifications $\mathbb{F}$. Due to the granularity abstraction in our denotation, it represents every invocation of a composite operation by a single event and thereby abstracts from its internal structure. In particular, different composite data type implementations can have the same denotation describing the data type behaviour. As a consequence, in *(ii)* we can pretend that composite data types are primitive and thus do not have to reason about the behaviour of their implementations and the client together. For example, we can determine how a program using the social graph data type behaves in the situation shown in Figure 5(a) using the result the data type denotation gives on this context, without considering how its implementation behaves (cf. Figure 5(b)). We get the same benefits when reasoning about a complex composite data type D constructed from simpler composite data types T_j as in (7): we can first compute the denotations of T_j and then use the results in reasoning about D.

In practice, we do not compute the denotation of a composite data type D using Definition 9 directly. Instead, we typically invent a specification F that describes the desired behaviour of D, and then prove that F is equal to the denotation of D, i.e., that D is *correct* with respect to F. Definition 9 and, in particular, constraints (8)–(12), give a *proof method* for establishing this. The next section illustrates this on an example.

5 Example: Social Graph

We have applied the composite data type denotation in §4 to specify and prove the correctness of three composite data types: *(i)* the social graph data type in Figure 3; *(ii)* a shopping cart data type implemented using an add-wins set, which resolves conflicts between concurrent changes to the quantity of the same product; *(iii)* a data type that uses transactions to simultaneously update several objects that resolve conflicts using the last-writer-wins policy (cf. `LWWset` from §3.1). The latter example uses arbitration in a nontrivial way. Due to space constraints, we focus here on the social graph data type and defer the others to [13, §D].

Below we give a specification F_{soc} to the social graph data type, which we have proved to be the denotation of its implementation D_{soc} in Figure 3. The proof is done by considering an arbitrary context N and its concretisation (X, c) according to Definition 7 and showing that $F_{\mathsf{soc}}(N) = c$. The constraints (8)–(12) make the required reasoning mostly mechanical and therefore we defer the easy proof to [13, §D] and only illustrate the correspondence between D_{soc} and F_{soc} on examples.

The function F_{soc} is defined recursively using the following operation that selects a subcontext of a given event in a context, analogously to the ctxt operation on executions (4) from §3.2. For a partial context M and an event $e \in M.E$, we let

$$\mathsf{ctxt}(M, e) = (M.\mathsf{aop}(e), E, (M.\mathsf{aop})|_E, (M.\mathsf{vis})|_E, (M.\mathsf{ar})|_E),$$

where $E = (M.\mathsf{vis})^{-1}(e)$. Then

$$\begin{aligned}
&F_{\mathsf{soc}}(\mathtt{get}(a), M) = \\
&(\{b \mid \exists e \in (M.E).\, (M.\mathsf{aop}(e) = \mathtt{accept}((b,a) \mid (a,b))) \wedge F_{\mathsf{soc}}(\mathsf{ctxt}(M,e)) \wedge \\
&\quad (\forall f \in (M.E).\, (M.\mathsf{aop}(f) \in \mathtt{breakup}((b,a) \mid (a,b))) \wedge F_{\mathsf{soc}}(\mathsf{ctxt}(M,f)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \implies f \xrightarrow{\mathsf{vis}} e)\}, \\
&\ \{b \mid \exists e \in (M.E).\, (M.\mathsf{aop}(e) = \mathtt{request}(b,a)) \wedge F_{\mathsf{soc}}(\mathsf{ctxt}(M,e)) \wedge \\
&\quad (\forall f \in (M.E).\, (M.\mathsf{aop}(f) \in (\mathtt{accept} \mid \mathtt{reject})((b,a) \mid (a,b))) \wedge F_{\mathsf{soc}}(\mathsf{ctxt}(M,f)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \implies f \xrightarrow{\mathsf{vis}} e)\}); \\
&F_{\mathsf{soc}}(\mathtt{accept}(b,a), M) = (b \in \mathsf{snd}(F_{\mathsf{soc}}(\mathtt{get}(a), M))).
\end{aligned}$$

The results of `request`, `reject` and `breakup` are defined similarly to `accept`. For brevity, we use the notation $(G_1 \mid G_2)$ above to denote the set arising from picking either G_1 or G_2 as the subexpression of the expression where it occurs. Even though the definition looks complicated, its conceptual idea is simple and has a temporal flavour. Our definition takes into account that: after breaking up, users can become friends again; and sometimes data type operations are unsuccessful, in which case they return false. According to the two components of $F_{\mathsf{soc}}(\mathtt{get}(a), M)$:

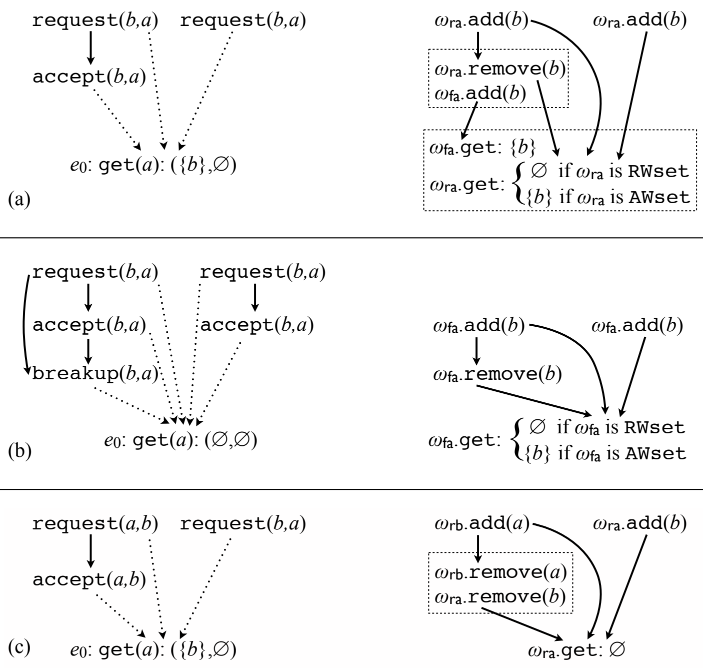


Fig. 7. (Left) Coarse-grained contexts of the social graph data type together with the result that F_{soc} gives on them. (Right) Relevant events of the fine-grained executions of the implementation in Figure 3 resulting from concretising the contexts according to Definition 7. We use the same conventions as in Figure 5.

1. a's friends are the accounts b with a successful `accept` operation between a and b such that any successful `breakup` between them was in its past, as formalised by visibility. We determine whether an operation was successful by calling F_{soc} recursively on its subcontext.
2. a's requesters are the accounts b with a successful $\mathtt{request}(b, a)$ operation such that any successful `accept` or `reject` between a and b was in its past.

This specifies the behaviour of the data type while abstracting from its implementation, thereby enabling modular reasoning about programs using it (§4.3).

Our specification F_{soc} can be used to analyse the behaviour of the implementation in Figure 3. By a simple unrolling of the definition of F_{soc}, it is easy to check that the two sets returned by $F_{\mathsf{soc}}(\mathtt{get}(a), M)$ are disjoint and, hence, the invariant (2) in §2 holds; (1) can be checked similarly. Also, since F_{soc} returns $(\{b\}, \emptyset)$ on the context in Figure 5(a), when the same friendship request is concurrently accepted and rejected, the accept wins. Different behaviour could also be reasonable; the decision ultimately depends on application requirements.

We now illustrate the correspondence between D_{soc} and F_{soc} on examples and, on the way, show that our coarse-grained semantics lets one understand how the choice of conflict-resolution policies on constituent objects affects the policy of the composite data type. First, we argue that making $\mathit{requesters}$ remove-wins in Figure 3 is crucial for preserving the integrity invariant (2) and satisfying F_{soc}. Indeed, consider the scenario shown in Figure 7(a). Here two users managing the same account b concurrently issue friendship requests to a, which initially sees only one of them. If $\mathit{requesters}$ were add-wins, the `accept` by a would affect only the request that it sees. The remaining request would eventually propagate to all replicas in the system, and the calls to `get` in the implementation would thus return b as being both a friend and a requester of a's, violating (2). The remove-wins policy of $\mathit{requesters}$ ensures that, when a user accepts or rejects a request, this also removes all identical requests issued concurrently.

If we made $\mathit{friends}$ add-wins, this would make the data type behave differently, but sensibly, as illustrated in Figure 7(b). Here we again have two concurrently issued requests from b to a. The account a may also be managed by multiple users, which concurrently accept the requests they happen to see. One of the users then immediately breaks up with a. Since $\mathit{friends}$ are remove-wins, this cancels the addition of b to $\mathit{friends}[a]$ (i.e., ω_{fa}) resulting from the concurrent `accept` by the other user; thus, b ends up not being a's friend, as prescribed by F_{soc}. Making $\mathit{friends}$ add-wins would result in the reverse outcome, and F_{soc} would have to change accordingly. Thus, the conflict-resolution policy on $\mathit{friends}$ determines the way conflicts between `accept` and `breakup` are resolved.

Finally, if users a and b issue friendship requests to each other concurrently, a decision such as an `accept` taken on one of them will also affect the other, as illustrated in Figure 7(c). To handle this situation without violating (2), `accept` removes not only the request it is resolving, but also the symmetric one.

6 Fine-grained Language Semantics, Soundness and Completeness

To justify that the coarse-grained semantics from §4 is sensible, we relate it to a *fine-grained semantics* that follows the standard way of defining language semantics on weak consistency models [9]. Unlike the coarse-grained semantics, the fine-grained one is defined non-compositionally: it considers only certain *complete* programs and defines the denotation of a program as a whole, without separately defining denotations of composite data types in it. This denotation is computed using histories that record all operations on all primitive objects comprising the composite data types in the program; hence, the name fine-grained. The semantics includes those histories that can be produced by the program in the session-local semantics (§4.1) and are allowed by the semantics of the store managing the primitive objects the program uses (§3).

We state the correspondence between the coarse-grained and fine-grained semantics as an equivalence of the *externally-observable behaviour* of a program in the two semantics. Let us fix a variable $x_{\mathsf{io}} \in \mathsf{OVar}$ and an object $\mathsf{io} \in \mathsf{Obj}$ used to interpret x_{io}. A program P is ***complete*** if $\emptyset \mid x_{\mathsf{io}} : \{o_{\mathsf{io}}\} \vdash P$. The operation o_{io} on x_{io} models a combined user input-output action, rather than an operation on the store, and the externally-observable behaviour of a complete program P is given by operations on x_{io} it performs. Formally, for a history H let $\mathsf{observ}(H)$ be its projection to events on io: $\{e \in H \mid H.\mathsf{obj}(e) = \mathsf{io}\}$. We lift observ to sets of histories pointwise. Then we define the set of externally-observable behaviours of a complete program P in the coarse-grained semantics of §4 as $[\![P]\!]_{\mathsf{CG}} = \mathsf{observ}([\![P]\!]([\,], [x_{\mathsf{io}} : \mathsf{io}], [\,]))$. Note that our semantics does not restrict the values returned by o_{io}, thus accepting any input.

To define the fine-grained semantics of a complete program P, we flatten P by inlining composite data type definitions using a series of reductions $\longrightarrow$ on programs (defined shortly). Applying the reductions exhaustively yields programs with only objects of primitive data types, which have the following normal form:

$$\bar{P} \ ::= \ C_1 \parallel \ldots \parallel C_n \ \mid \ \mathsf{let}\ x = \mathsf{new}\ B\ \mathsf{in}\ \bar{P}$$

Given a complete program P, consider the unique $\bar{P}$ such that $P \longrightarrow^* \bar{P}$ and $\bar{P} \not\longrightarrow \_$. Then we define the denotation of P in the fine-grained semantics by the set of externally-observable behaviours that $\bar{P}$ produces when interacting with a causally consistent store managing the primitive objects it uses. To formalise this, we reuse the definition of the coarse-grained semantics and define the denotation of P in the fine-grained semantics as $[\![P]\!]_{\mathsf{FG}} = [\![\bar{P}]\!]_{\mathsf{CG}}$. Since $\bar{P}$ contains only primitive data types, this does not use the composite data type denotation of §4.2.

We now define the reduction $\longrightarrow$. Let Comm be the set of commands C in Figure 2. We use an operator $subst$ that takes a mapping $S : \mathsf{OVar} \times \mathsf{Op} \rightharpoonup \mathsf{Comm}$ and a command C or a program P, and replaces invocations of object operations in C or P according to S. The key clauses defining $subst$ are as follows:

$$\begin{aligned}
subst(S, v = x.o(G)) \ &= \ \mathsf{if}\ ((x, o) \notin \mathsf{dom}(S))\ \mathsf{then}\ (v = x.o(G)) \\
&\quad\ \mathsf{else}\ (\mathsf{atomic}\ \{\mathsf{var}\ v_1.\, \mathsf{var}\ v_2.\, v_1 = G; (S(x, o)[v_1/v_{\mathsf{in}}, v_2/v_{\mathsf{out}}]); v = v_2\}) \\
subst(S, \mathsf{let}\ x = \mathsf{new}\ T\ \mathsf{in}\ P) \ &= \ \mathsf{let}\ x = \mathsf{new}\ T\ \mathsf{in}\ subst(S|_{\neg x}, P) \\
subst(S, \mathsf{let}\ \alpha = T\ \mathsf{in}\ P) \ &= \ \mathsf{let}\ \alpha = T\ \mathsf{in}\ subst(S, P) \\
subst(S, C_1 \parallel \ldots \parallel C_n) \ &= \ subst(S, C_1) \parallel \ldots \parallel subst(S, C_n)
\end{aligned}$$

Here v_1, v_2 are fresh ordinary variables, and $S|_{\neg x}$ denotes S with its domain restricted to $(\mathsf{OVar} \setminus \{x\}) \times \mathsf{Op}$. Applying $subst$ to an assignment command does not change the command, and applying it to all others results in recursive applications of $subst$ to their subexpressions. Then the relation $\longrightarrow$ is defined as follows:

$$\begin{aligned}
&\mathcal{P} \ ::= \ [-] \mid \mathsf{let}\ x = \mathsf{new}\ T\ \mathsf{in}\ \mathcal{P} \mid \mathsf{let}\ \alpha = T\ \mathsf{in}\ \mathcal{P} \\
&\mathcal{P}[\mathsf{let}\ \alpha = T\ \mathsf{in}\ P] \ \longrightarrow \ \mathcal{P}[P[T/\alpha]] \\
&\mathcal{P}[\mathsf{let}\ x = \mathsf{new}\ (\mathsf{let}\ \{x_j = \mathsf{new}\ T_j\}_{j=1..m}\ \mathsf{in}\ \{o = \mathsf{atomic}\ \{C_o\}\}_{o \in O})\ \mathsf{in}\ P] \\
&\qquad\qquad \longrightarrow \ \mathcal{P}[\mathsf{let}\ \{x_j = \mathsf{new}\ T_j\}_{j=1..m}\ \mathsf{in}\ subst(\{(x, o) \mapsto C_o \mid o \in O\}, P)],
\end{aligned}$$

where x_j do not occur in P. The first reduction rule replaces data-type variables by their definitions, and the second defines the semantics of composite operations via inlining.

Our central technical result is that the coarse-grained semantics of §4 is sound and complete with respect to the fine-grained semantics presented here: the sets of externally-observable behaviours of programs in the two semantics coincide.

THEOREM 10 *For every complete program P we have $[\![P]\!]_{\mathsf{FG}} = [\![P]\!]_{\mathsf{CG}}$.*

We give a (highly nontrivial) proof in [13, §C]. The theorem allows us to reason about programs using the coarse-grained semantics, which enables granularity abstraction and modular reasoning (§4.3). It also implies that our denotational semantics is *adequate*, i.e., can be used to prove the observational equivalence of two data type implementations D_1 and D_2: if $[\![D_1]\!] = [\![D_2]\!]$, then $[\![\mathcal{C}[D_1]]\!]_{\mathsf{FG}} = [\![\mathcal{C}[D_2]]\!]_{\mathsf{FG}}$ for all contexts $\mathcal{C}$ of the form $\mathcal{P}[\mathsf{let}\ \alpha = [-]\ \mathsf{in}\ P]$. Note that both soundness and completeness are needed to imply this property.

7 Related Work

One of the classical questions of data abstraction is: how can we define the semantics of a data type implementation that abstracts away the implementation details, including a particular choice of data representation? Our results can be viewed as revisiting this question, which has so far been investigated in the context of sequential [14] and shared-memory concurrent [11, 24] programs, in the emerging domain of eventually consistent distributed systems. Most of the work on data abstraction for concurrency has considered a strongly consistent setting [11, 24]. Thus, it usually aimed to achieve *atomicity abstraction*, which allows one to pretend that a composite command takes effect atomically throughout the system. Here we consider data abstraction in the more challenging setting of weak consistency and achieve a weaker and more subtle guarantee of *granularity abstraction*: although our coarse-grained semantics represents composite operations by single events, these events are still subject to anomalies of causal consistency, with different replicas being able to see the events at different times.

We are aware of only a few previous data abstraction results for weak consistency [15, 7, 5]. The most closely related is the one for the C/C++ memory model by Batty et al. [5]. Like the consistency model we consider, the C/C++ model is defined axiomatically, which leads to some similarities in the general approach followed in [5] and in this paper. However, other features of the settings considered are different. First, we consider arbitrary replicated data types, whereas, as any model of a shared-memory language, the C/C++ one considers only registers with the last-writer-wins conflict-resolution policy. Second, the artefacts related during abstraction in [5] and in this paper are different. Instead of composite replicated data types, [5] considers *libraries*, which encapsulate last-writer-wins registers and operations accessing them implemented by arbitrary code without using transactions. A specification of a library is then just another library, but with operations implemented using atomic blocks reminiscent of our transactions. Hence, a single invocation of an operation of a specification library is still represented by multiple events and therefore [5] does not support granularity abstraction to the extent achieved here. Our work can roughly be viewed as starting where [5] left off, with composite constructions whose operations are implemented using transactions, and specifying their behaviour more declaratively with replicated data type specifications over contexts of coarse-grained events. It is thus possible that our approach can be adapted to give more declarative specifications to C/C++ libraries.

Researchers and developers have often implemented complex objects with domain-specific conflict resolution policies inside replicated stores [21], which requires dealing with low-level details, such as message exchange between replicas. Burckhardt et al. [9] also proposed a method for proving the correctness of such replicated data type implementations with respect to specifications of Definition 2. Our results show that, using causally consistent transactions, complex domain-specific objects can often be implemented as composite replicated data types, using a high-level programming model to compose replicated objects and their conflict-resolution policies. Furthermore, due to the granularity abstraction we established, the resulting objects can be viewed as no different from those implemented inside the store. The higher-level programming model we consider makes our proof method significantly different from that of Burckhardt et al.

Partial orders, such as event structures [19] and Mazurkiewicz traces [20], have been used to define semantics of concurrent or distributed programs by explicitly expressing the dependency relationships among events such programs generate. Our results extend this line of semantics research by considering new kinds of relations among events, describing computations of eventually consistent replicated stores, and studying how consistency axioms on these relations interact with the granularity abstraction for composite replicated data types.

8 Conclusion

In this paper we have proposed the concept of composite replicated data types, which formalises a common way of organising applications on top of eventually consistent stores. We have presented a coarse-grained denotational semantics for these data types that supports granularity abstraction: the semantics allows us to abstract from the internals of a composite data type implementation and pretend that it represents a single monolithic object, which simplifies reasoning about client programs. We have also shown that our semantics is sound and complete with respect to a standard non-compositional semantics.

One important derivative of our semantics is a mechanism for specifying composite data types where we regard all operations of these data types as atomic, and describe their return values for executions that consist of such atomic operations. As our soundness and completeness results show, this mechanism is powerful enough to capture all essential aspects of composite replicated data types. Using a nontrivial example, we have illustrated how the denotation of a data type in our semantics specifies its behaviour in tricky situations and thereby lets one understand the consequences of different design decisions in its implementation.

As we explained in §1, developing correct programs on top of eventually consistent stores is a challenging yet unavoidable task. Our results mark the first step towards providing developers with methods and tools for specifying and verifying programs in this new programming environment and expanding the rich theories of programming languages, such as data abstraction, to this environment. Even though our results were developed for a particular popular variant of eventual consistency—causally consistent transactions—we hope that in the future the results can be generalised to other consistency models with similar formalisations [8, 3]. Another natural future direction is to use our coarse-grained semantics to propose a logic for reasoning about composite data types symbolically.

References

1. Microsoft TouchDevelop. https://www.touchdevelop.com/.
2. D. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer*, 2012.
3. P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly Available Transactions: virtues and limitations. In *VLDB*, 2014.
4. P. Bailis and A. Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *CACM*, 56(5), 2013.
5. M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In *POPL*, 2013.
6. A. Bieniusa, M. Zawirski, N. M. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. Semantics of eventually consistent replicated sets. In *DISC*, 2012.
7. S. Burckhardt, A. Gotsman, M. Musuvathi, and H. Yang. Concurrent library correctness on the TSO memory model. In *ESOP*, 2012.
8. S. Burckhardt, A. Gotsman, and H. Yang. Understanding eventual consistency. Technical Report MSR-TR-2013-39, Microsoft, 2013.
9. S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: specification, verification, optimality. In *POPL*, 2014.
10. S. Burckhardt, D. Leijen, M. Fähndrich, and M. Sagiv. Eventually consistent transactions. In *ESOP*, 2012.
11. I. Filipovic, P. W. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52), 2010.
12. S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2), 2002.
13. A. Gotsman and H. Yang. Composite replicated data types (extended version). Available from http://software.imdea.org/∼gotsman, 2015.
14. C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1, 1972.
15. R. Jagadeesan, G. Petri, C. Pitcher, and J. Riely. Quarantining weakness - compositional reasoning under relaxed memory models (extended abstract). In *ESOP*, 2013.
16. C. Li, D. Porto, A. Clement, R. Rodrigues, N. Preguiça, and J. Gehrke. Making geo-replicated systems fast if possible, consistent when necessary. In *OSDI*, 2012.
17. W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *SOSP*, 2011.
18. W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *NSDI*, 2013.
19. M. Nielsen, G. D. Plotkin, and G. Winskel. Petri nets, event structures and domains. In *Semantics of Concurrent Computation*, 1979.
20. M. Nielsen, V. Sassone, and G. Winskel. Relationships between models of concurrency. In *REX School/Symposium*, 1993.
21. M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report 7506, INRIA, 2011.
22. M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *SSS*, 2011.
23. Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, 2011.
24. A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, 2013.
25. M. Zawirski, A. Bieniusa, V. Balegas, S. Duarte, C. Baquero, M. Shapiro, and N. Preguiça. SwiftCloud: Fault-tolerant geo-replication integrated all the way to the client machine. Technical Report 8347, INRIA, 2013.