Conflict-Aware Replicated Data Types

NICHOLAS V. LEWCHENKO, University of Colorado Boulder
ARJUN RADHAKRISHNA, Microsoft
AKASH GAONKAR, University of Colorado Boulder
PAVOL ČERNÝ, University of Colorado Boulder

Abstract We introduce Conflict-Aware Replicated Data Types (CARDs). CARDs are significantly more expressive than Conflict-free Replicated Data Types (CRDTs) as they support operations that can conflict with each other. Introducing conflicting operations typically brings the need to block an operation in at least some executions, leading to difficulties in programming and reasoning about correctness, as well as potential inefficiencies in implementation.

The salient aspect of CARDs is that they allow ease of programming and reasoning about programs comparable to CRDTs, while enabling algorithmic inference of conflicts so that an operation is blocked only when necessary. The key idea is to have a language that allows associating with each operation a two-state predicate called a consistency guard that relates the state of the replica on which the operation is executing to a global state (which is never computed). The consistency guards bring three advantages. First, a programmer developing an operation needs only to choose a consistency guard that states what the operation will rely on. In particular, they do not need to consider the operation conflicts with other operation. This allows purely modular reasoning. Second, we show that consistency guard allow reducing the complexity of reasoning needed to prove invariants that hold as CARD operations are executing. The reason is that consistency guard allow reducing the reasoning about concurrency among operations to purely sequential reasoning. Third, conflicts among operations can be algorithmically inferred by checking whether the effect of one operation preserves the consistency guard of another operation.

We substantiate these claims by introducing a language for writing CARD operations. The language is dependently typed, and the type checking rules are based on the modular and sequential reasoning allowed by consistency guards. We also show how conflicts can be inferred at compile time, and the resulting constraints on executions can be enforced at runtime. We empirically show that the inference needed to detect conflicts between operations is well within the scope of current SMT solvers.

1 INTRODUCTION
Conflict-free replicated data types (CRDTs) have quickly gained traction in large-scale distributed systems [Attiya et al. 2016; Brown et al. 2014; Day-Richter 2010; Mehdi et al. 2014; Nédelec et al. 2013; Shapiro et al. 2011; Teixeira 2017]. They allow operations to execute efficiently and independently across different replicas without coordination while still guaranteeing strong eventual consistency. CRDTs rely on the fact that their operations are conflict-free (commutable). However, the assumption of conflict-freedom is broken in many practical scenarios either due to the presence of inherently conflicting operations, or due to the need for maintaining invariants on the data structure.

There have been several attempts to add conflicting operations to CRDTs using mixed-consistency and tunable-consistency extensions in both academia and industry [Balegas et al. 2015; Gotsman et al. 2016; Lakshman and Malik 2010; Li et al. 2014, 2012; Sivaramakrishnan et al. 2015]. However, most of these systems suffer from one of several drawbacks: (i) The programmer has to explicitly reason about and state conflicts for each pair of operations [Gotsman et al. 2016] or choose a consistency level (sequential or eventual consistency in [Li et al. 2012]) for each operation. These tasks cannot be done modularly, that is, separately for each operation. (ii) The programmer can specify consistency for each operation in isolation, but the overall consistency model does not give

Authors’ addresses: Nicholas V. Lewchenko, University of Colorado Boulder, nicholas.lewchenko@colorado.edu; Arjun Radhakrishna, Microsoft, arradha@microsoft.com; Akash Gaonkar, University of Colorado Boulder, akash.gaonkar@colorado.edu; Pavol Černý, University of Colorado Boulder, pavol.cerny@colorado.edu.

, Vol. 1, No. 1, Article . Publication date: July 2018.
clear guarantees. For example, in Cassandra [Lakshman and Malik 2010], a programmer can specify that an operation can execute with coordination across just a small number of replicas. However, it is not clear what consistency guarantees this provides the user.

Conflict-Aware Replicated Data-Types. We present a novel extension of CRDTs dubbed conflict-aware replicated data-types (CARDs), which support operations that might not be conflict-free.

From the perspective of the user of CARD operations, CARDs guarantee: (a) strong eventual consistency (SEC): all the replicas should eventually process the (emitted effects of) operations and should agree on the final value [Shapiro et al. 2011], (b) availability: replicas should operate without blocking coordination whenever possible—that is, whenever the operations do not conflict, and (c) preservation of application-specific invariants.

From the perspective of the developer of CARD operations, CARDs guarantee: (d) modular consistency specifications where the assumptions that an operation relies on are stated with only that operation in mind, and allow purely modular reasoning (e) proof system where the reasoning about concurrent behavior is reduced to sequential reasoning, and (f) automated detection of conflicts between operations.

Execution model. CARD operations are executed by a network of replicas. A client can ask a replica to execute an operation. The replica evaluates the operation, provides a return value to the client, and sends the effect of the operation to all the other replicas. The effect is a state transformer (for instance, it does not compute the return value) that the other replicas use to update their states.

Consistency guards. The key idea of our approach is to introduce a programming language that allows specifying consistency requirements for each operation separately. The consistency requirements for an operation are specified using a two-state predicate called a consistency guard. The guard relates the replica state and the global state. An operation can rely on a guard while it (the operation) is executing to ensure that no operations that could break the guard are run in parallel.

Example: key-value store. Consider a simple key-value store and an operation, \texttt{insert(k,v)}. When executed on a replica, the operation tests (using a predicate \texttt{present(k)}) whether an entry with key \(k\) is already in the store. If so, the operation has no effect. Otherwise, it inserts the pair \(\langle k, v \rangle\). Furthermore, it issues an effect \texttt{ins(k,v)} that simply tells all the other replicas to execute the insertion as well, without further tests. Without any other requirements on the store, this implementation is eventually consistent and is an example of a CRDT. There are no conflicting operations (as the \texttt{insert} operation does not conflict with another instance of itself). In Figure 1, there are three executions, all eventually consistent.

Let us consider a store that has an invariant that all the entries must have unique keys. Is this invariant maintained? The behavior of \texttt{insert(k,v)} depends on the value of \texttt{present(k)}. But \texttt{present(k)} is evaluated based only on the information the replica has. Thus it is entirely possible that another replica executes another \texttt{insert(k,v)} operation with the same value of \(k\), leading to a store with non-unique keys which violate the invariant. Thus in this case the \texttt{insert} operation can conflict with another instance of itself, but only when both want to insert an entry with the same key. In Figure 1, the execution on the left and the one in the center violate the invariant, while the execution on the right preserves it.

To ensure that the invariant is preserved, the developer writing the \texttt{insert(k,v)} operation introduces the consistency guard \texttt{presentR(k)==presentG(k)}, which requires that the replica
value \(\text{presentR}(k)\) is equal to the global value \(\text{presentG}(k)\). The guard prevents other replicas from executing \(\text{insert}(k, v)\) with the same value of \(k\) in parallel, as such executions would modify \(\text{presentG}(k)\) and thus invalidate the guard. However, the guard does not prevent parallel execution of \(\text{insert}(k', v)\) for \(k'\) different from \(k\).

**Global state.** The consistency guard refers to a global state. This global state is never computed during the distributed execution, but it is well-defined at each moment of the computation and the guard (i.e., a relation between the global state and the replica state) can be maintained. The global state is defined using the arbitration order [Burckhardt 2014] which is a total order on all events in a computation. The arbitration order can be maintained in a standard way without any synchronization. For a particular event in a computation, the global state is obtained by evaluating all the effects that are before that event in the arbitration order.

**Replica state.** During the computation, a replica of course does not have access to the global state. All it has is the effects it has seen (note that there might be effects that the replica has not seen yet that will be arbitrated before the current operation). Thus the replica state is determined using the visibility partial order \(\text{vis}\): an effect \(e\) is after an effect \(f\) in the visibility order iff the operation that produced \(e\) ran at a replica which has seen \(f\) at that time. We require that the arbitration order and the visibility order agree. This requirement is called causal consistency and can be maintained without any blocking synchronization.

**Maintaining the consistency guards.** We are now ready to explain how consistency guards are maintained. If a replica starts to execute an operation guarded by a guard \(\varphi\) and producing an effect \(\eta\), it makes sure that for every other effect \(\eta'\) either (i) \(\eta\) and \(\eta'\) were not produced in parallel, i.e. \(\text{vis}(\eta, \eta')\) or \(\text{vis}(\eta', \eta)\), or (ii) \(\eta'\) does not invalidate \(\varphi\). That is, the operations that are allowed to run in parallel do not invalidate \(\varphi\). Thus if \(\varphi\) is true when the operation starts, it is true while the operation executes. (We provide only an intuition here, see also Section 5 for a stronger version of (ii) we need.)

This condition is possible to enforce by taking a distributed lock associated with \(\varphi\), and thereby disallowing all conflicting operations (operations such that their effects can modify the global state in a way that might invalidate \(\varphi\)) to run in parallel. Another replica considers the lock released when it receives the effect of the operation that took the lock.

**CARDs for the user.** We show how our system satisfies the points (a) to (f) above. Let us first consider the key-value CARD from the point of view of the user.

---

1The language we introduce has a different syntax for specifying guards. For brevity, here we provide directly the two state predicate that the guard defines.
(a) Strong eventual consistency is achieved in a standard way by having the arbitration order. Each replica maintains a sequence of effects ordered by the arbitration order, so eventually the state at every replica will be obtained by evaluating the same effects in the same order.

(b) Availability is achieved because operations are executed without blocking synchronization when possible. For instance, if \( k \) is different from \( k' \), then \( \text{insert}(k,v) \) and \( \text{insert}(k',v) \) do not need to synchronize. Indeed, the effect \( \text{ins}(k',v) \) does not invalidate the guard, as it does not change either the replica value or the global value of \( \text{present}(k) \).

(c) Application invariants (despite the presence of conflicting operations) are maintained thanks to the consistency guards. We explained how the guard for \( \text{insert} \) protects the invariant that the store contains entries with unique keys.

CARDs for the developer. For the developer of a CARD, the following properties hold.

(d) Modular reasoning: Consistency guards allow specifying the assumptions that a method relies on without considering what other methods might be operating on the same CARD. For instance, the guard for the \( \text{insert}(k,v) \) ensures that the operation is correct regardless of what the other operations do.

(e) Sequential reasoning: The consistency guards allow sequential reasoning about correctness of each individual operation, even though these operations run in a distributed system. The reason is that the guard is the only assumption that the operation makes on its distributed environment. We will provide an overview of the reasoning needed to prove correctness of an operation in Section 2.

(f) Algorithmic conflict detection: In our setting, the conflicts are between guards and effects. For instance, the guard \( \text{presentR}(k) == \text{presentG}(k) \) is in conflict with the effect \( \text{ins}(k,v) \). Given consistency guards, we provide a weakest-precondition based algorithm that uses an SMT solver to automatically infer potential conflict between effects and guards at compile time. We use the results to introduce necessary blocking coordination (with no unnecessary coordination). In particular, this means that such a system behaves as a CRDT in cases where the data structure supports conflicting operations, but they are never executed.

Core calculus for CARDs. We introduce \( \lambda^Q \), a core calculus for specifying CARDs. It extends the \( \lambda \) calculus by introducing terms for queries (that create consistency guards) and for emitting effects. The calculus generalizes the description above by allowing a replica to issue nested queries (that impose one consistency guard each) before issuing an effect. The calculus is typed using refinement (liquid) types that allow expressing pre- and post-condition for each operation. Given an invariant \( I \), we can prove it by typechecking – we can show that each operation typechecks with its pre- and post-condition set to \( I \).

Contributions. To summarize, this paper makes the following contributions.

- We extend CRDTs to CARDs, allowing conflicting operations, and enabling programmers to modularly specify conflicts with consistency guards. [Section 3]
- We introduce \( \lambda^Q \), a core calculus for specifying CARDs. [Section 4]
- We show that invariants on CARDs can be proved sequentially and modularly. To this end, we introduce a refinement (liquid) type system for \( \lambda^Q \) and show that it is a (sequential and modular) proof system for CARD invariants and more generally for correctness of CARD operations. [Section 4]
- We provide a weakest pre-condition based algorithm for automatically inferring the minimal required synchronization between replicas in CARDs. [Section 5]
- We describe a protocol that implements CARDs and prove it correct. [Section 6]
Conflict-Free

(a)

(b)

(c)

(d)

Fig. 2. Examples of conflict-free and conflicting operations in executions in a bank account with three operations: deposit (+), withdraw (−), and interest (*). Each execution consists of two replicas, each executing one operation with no coordination (solid line), and then broadcasting their effects (dashed line). Executions (a) and (b) are conflict-free, (c) produces a negative balance (breaking an application invariant), and (d) leads to a divergent state (breaking SEC).

• We implement the automated conflict inference algorithm and evaluate it on several small, but representative replicated data-types. The results show that the inference needed to detect conflicts between operations is well within the scope of current SMT solvers. [Section 7]

2 WRITING AND VERIFYING CARD OPERATIONS

We provide an overview of CARDs and λQ on an illustrative example: a bank account where some operations conflict with each other. We explain how the application is programmed with λQ operations over a general-purpose CARD, and show how static conflict information can be inferred for the CARD and used to verify application-specific properties for the bank account. We then extend the example to show how non-commuting effects can be handled. The application consists of withdraw and deposit operations over a Counter CARD (simple integer value that supports addition and subtraction). Executing these operations at a replica emits Counter effects which will eventually be processed by other replicas.

Problem and desired result. The bank account has three requirements: strong eventual consistency, availability, and preserving application-specific invariant I: the bank account value should never be negative. The Counter effects produced by deposit and withdraw (Add n and Sub n, respectively) commute, and thus SEC can be achieved without damaging availability (as in CRDTs). However, the replicas need to coordinate in order to maintain the invariant I. The withdraw operation can be made “smart” so that it decides not emit a Sub n effect if it sees that the account is too small, but if for example two withdraw 7 operations running on separate replicas see a store value of 10 and make their decisions before they see each other, they will together reduce the account to −4, breaking the invariant anyway (See Figure 2c). Thus two withdraws cannot run in parallel; if they do, their safety logic might not work. On the other hand, multiple deposits can run in parallel, and even multiple deposits and a single withdraw can run in parallel. The desired technique should therefore statically detect a conflict between the two withdraws, and (i) avoid this conflict, while (ii) allowing all other operations run in parallel without incurring a performance penalty (and thus preserve availability to the extent possible).

A CARD D is a rich datatype consisting of a basic store type S(D), a type E(D) of effects which transform the store type, and a type C(D) of consistency guards that state conditions of partial equivalence between store values. For example, a consistency guard on a list CARD might state that two list values are identical up to some nth element. We use guards in CARD applications to state what kind of consistency is required (and thus what kind of interference is disallowed) for a particular access of replicated store data.
we described earlier: withdraw
This operation type states that the term
The term $Q$ varying application-specific safety properties they will be used to implement.
$R$, Vol. 1, No. 1, Article . Publication date: July 2018.
refer to the return value, the store value before applying the operation’s effect, and the store value
removed from the account. The
a non-negative invariant is preserved and (2) the return value ($R$) returning an integer and meeting two refinement conditions: (1) the bank account’s Counter
uses standard sequential reasoning tools to verify operation behavior. In particular, we extend the
Because guards reduce the concurrent problem of operation correctness to a sequential one, we can
Checking a Dependent Operation Type
The example CARD we are using here is the Counter, defined in Figure 3, which uses an integer
as its store type, supports simple numerical effects, and provides lower (LE) and upper (GE) bound
a replica running the withdraw operation for
We define operations over a CARD $D$ using $\lambda^O$, an extension of the $\lambda$-calculus. An
operation is a program which runs an effect and/or returns information to the caller based on
partial knowledge of the store’s current value. For example, consider the withdraw operation for
our bank account example written in $\lambda^O$:
\[
\text{withdraw} := \lambda n. (Q \text{ LE } x. (\text{if } (x \geq n) \text{ then } R.(\text{Sub } n, n) \text{ else } R.(\text{NoOp}, 0)))
\]
The term $Q \text{ LE } x. (\ldots)$ binds a snapshot of the store to $x$ for use in the if-expression. In order to
choose safely whether to subtract the argument value $n$ from the store, the snapshot bound to $x$
must not be greater than the current store value. Thus we annotate the term with the LE guard
decide that the current store must be less than or equal to the value we bind to $x$ – this safely
under-approximates the condition that $n$ should be at most the current store value. The base term
$R. (e, a)$ adds $e$ as an effect to the store and returns $a$ to the caller. In our case, we only add the
Sub $n$ to the store if we know that it is safe, and we return the value we decided to subtract (if any)
to the caller. A reader familiar with the challenges of distributed systems might be suspicious of
this “current value” for the replicated store. We will define precisely what this means in Section 4.

Notice that in writing this safe operation, we did not explicitly declare conflicts with other
operations or said anything about event orderings. A replica running withdraw uses the conflict
information previously generated for Counter to impose the network ordering constraints needed
to enforce our LE guard.

Checking a Dependent Operation Type
Because guards reduce the concurrent problem of operation correctness to a sequential one, we can
use standard sequential reasoning tools to verify operation behavior. In particular, we extend the
type inference rules of Liquid Types [Rondon et al. 2008] to cover $\lambda^O$’s unique terms. Operations
are then type-checked with respect to a specification on the behavior of the event they produce. For example, the specification we check for the withdraw operation states formally the behavior we described earlier:
\[
\text{withdraw} : (n : \text{Nat}) \rightarrow \text{Op} (\text{Counter, Int.} (s \geq 0 \Rightarrow s' \geq 0) \land (a = s - s'))
\]
This operation type states that withdraw, given a natural number amount, is an operation over
Counter returning an integer and meeting two refinement conditions: (1) the bank account’s
non-negative invariant is preserved and (2) the return value ($a$) reflects exactly the amount that is
removed from the account. The $a$, $s$, and $s'$ in the specification are special free variables used to
refer to the return value, the store value before applying the operation’s effect, and the store value

\[
\begin{align*}
S(C\text{ounter}) & := \mathbb{Z} \\
E(C\text{ounter}) & := \text{Add } \mathbb{N} \mid \text{Sub } \mathbb{N} \mid \text{Set } \mathbb{Z} \\
C(C\text{ounter}) & := \top \mid \text{LE} \mid \text{GE} \mid \text{EQ}
\end{align*}
\]

Fig. 3. Definition of the Counter CARD
after the operation completes. Our typing rules will reduce this to a Liquid Type which must be checked. The argument $n$’s type $\text{Nat}$ is itself an example of a Liquid Type which we will use in the derivation.

$$\llbracket n : \text{Nat} \rrbracket = \llbracket n : \{ v : \text{Int} \mid v \geq 0 \} \rrbracket = n \geq 0$$

We now check the operation type against our $\text{withdraw}$ definition. The correctness of $\text{withdraw}$ depends on the store value guarantee it demands via the $\text{LE}$ query guard, and the $\text{type}_Q$ typing rule adds that guarantee to the context.

$$\Gamma \vdash c : C \quad \Gamma, x : \{ v : S \mid [s/s_g][v/s_r][c] \} \vdash t : \text{Op}((S, E, C), A, \varphi)$$

Thus typing the outer term $Q \text{LE} \triangleright x$. if $\ldots$ adds $x : \{ v : \text{Int} \mid v \leq s \}$ which states that the value bound to $x$ is less than or equal to the pre-effect store value.

$$\llbracket x : \{ v : \text{Int} \mid v \leq s \} \rrbracket = x \leq s$$

Following the positive branch of the if $(x \geq n)$ then$\ldots \text{else} \ldots$ further adds $x \geq n$ to the context. We arrive at the final constraint-solving problem by applying the rule

$$\Gamma \vdash t_e : E \quad \Gamma \vdash t_a : \{ v : A \mid s' = \llbracket t_e \rrbracket(s) \Rightarrow \varphi \}$$

$$\Gamma \vdash R.(t_e, t_a) : \text{Op}((S, E, C), A, \varphi)$$

Following the $\text{type}_R$ rule, we need to show

$$\Gamma \vdash n : \{ v : \text{Int} \mid s' = ((\lambda s. s - n)(s) \Rightarrow (s \geq 0 \Rightarrow s' \geq 0) \land (a = s - s')) \}$$

to finish checking the positive then branch, which becomes the simple constraint problem

$$(n \geq 0) \land (x \leq s) \land (x \geq n) \land (s' = s - n) \Rightarrow (s \geq 0 \Rightarrow s' \geq 0) \land (a = s - s')$$
when $[\Gamma]$ is unpacked according to the Liquid Type rules. The trivial else branch check is clearly satisfied by the fact that its effect does nothing.

$$[\Gamma] \land (s' = s) \Rightarrow (s \geq 0 \Rightarrow s' \geq 0) \land (a = s - s')$$

**CARDs with Non-Commutable Effects**

Many replicated data reasoning models and implementations require all effects on the replicated store to be commutable in order to simplify the way histories are merged. In the interest of generality, CARDs do allow non-commuting store effects, and our reasoning technique and implementation technique are equipped to handle them efficiently. To demonstrate this flexibility and build some more intuition, let’s take a look at some example applications. More examples can be found in Section 7. Figure 2d illustrates how non-commutative effects (here, $+5$ and $\times 1.2$) can lead to replicas diverging, violating strong eventual consistency.

**Bank Account with Interest and Non-commuting Effects.** An obvious challenge of non-commutable effects is maintaining SEC. Our approach, following [Burckhardt et al. 2012], is to use an arbitration order, which is a total order on events which a replica chooses to evaluate the current value. The key is that the arbitration order must be chosen and maintained consistently across replicas. Such an order can be maintained using a standard combination of Lamport clocks and replica identifiers and by inserting newly received updates appropriately in history instead of appending them.

We now extend our example to show that even with non-commuting effects, strong eventual consistency can be achieved without blocking. Consider our bank account over an extended CARD Counter’ with new effect $\llbracket \text{Interest} \rrbracket := \lambda s.s + 1.2$, and suppose we write a new operation $\text{safeBalance}$ which returns a value that is definitely not less than the account’s actual value.

$$\text{safeBalance} : \text{Op}((\text{Counter'}, \mathbb{Z}), s = s' \land a \leq s)$$
The order of the Sub and Interest events matter, i.e., the effects do not commute. Most approaches [Li et al. 2012; Shapiro et al. 2011] would declare these two operations in conflict, and thus would be either disallowed (CRDTs) or declared strongly consistent (RedBlue). Furthermore, if effects are reordered at replicas, maintaining guarantees about the relationship between the return value and the global state becomes hard — so using an operation that reads this shifting state might require coordination.

However, the guard of safeBalance allows us to infer that its requirement does not conflict with either deposit or interest, so all three operations can be executed in parallel. Because the desired behavior of safeBalance was verified entirely based on its query guard, we can be sure that its behavior survives effect reorderings. Thus we achieve efficiency, even while ensuring application properties, by depending on the arbitration order rather than coordination to maintain SEC even with non-commutative effects.

**Joint Bank Account and Chained Conflicts.** We have explained how using the arbitration order allows achieving SEC. The downside is that due to non-commuting effects, detecting conflicts is in general more difficult than it was for our first bank account example. There may exist effects which cannot violate a guard, but instead can change the behavior of a non-commuting effect that does have the ability to violate a guard.

To demonstrate, we extend the example to a bank account which is jointly owned by two users, in which a user must first request a withdraw (via request) and wait for someone else to approve (via approve) before actually performing it.

We use a (Counter, Bool, Bool) tuple as the store, which supports the effects and guards of the Counter as well as effects and guard

\[
\begin{align*}
[\text{Request}] & := \lambda(s, b_1, b_2).(s, T, b_2) \\
[\text{Approve}] & := \lambda(s, b_1, b_2).(s, b_1, b_1) \\
[\text{Reset}] & := \lambda(s, b_1, b_2).(s, \bot, \bot)
\end{align*}
\]

in which Approve guarantees that the second boolean seen has the same value as the second boolean on the global store.

In this case, a user must first request a withdraw (via Request) and wait for someone else to approve (via Approve) in order for the withdrawal to have an effect.

\[
\text{withdrawJ} := Q \land \exists \text{Approve} \land (s, b_1, b_2).((s \geq n \land b_2) \\
\text{then } R.(\text{Sub } n \circ \text{Reset}, n) \\
\text{else } R.(\text{NoOp}, 0))
\]

The operation withdrawJ is guarded by Approve to be sure that the actual withdrawal of funds happens only if it was approved. The operation withdrawJ must not be concurrent with itself (as before), but it is now also in conflict with anything that emits Approve, as Approve can invalidate Approve.

Now note that Approve and Request are non-commuting: the behavior of Approve is changed by a Request existing before it. Consider a situation (illustrated in Fig. 4) where replica \( r_1 \) emits Approve and then runs withdrawJ, while concurrently, replica \( r_2 \) emits Request. Let us assume that the arbitration order will eventually put the Request before the effect of Approve. Then an execution can look as follows: replica \( r_1 \) sees an Approve (which does not set app to true as there is no request pending) and then \( r_1 \) executes a withdraw while guaranteeing that there are no concurrent Sub \( n \) or Approve effects. However, when the Request from replica \( r_2 \) is received by \( r_1 \), and the arbitration causes this effect to be ordered before the Approve, then suddenly the behavior of the Approve changes: it sets the second boolean to true.
We define CARDs, an abstract model of replicated data stores, and executions based upon them.

**CARDs**

A *conflict-aware replicated datatype* is a tuple $D = (S, E, C)$ where $S$ is the store type, $E$ is the type of effects, and $C$ is the type of consistency guards. Effects and consistency guards are detailed below. Informally, effects are store transformers and consistency guards specify the exact semantic restrictions on consistency under which each operation may execute under. The key point behind CARDs is to automate the reasoning about the interaction between effects and consistency guards. This allows a developer to program CARD operations modularly, letting the system handle conflicts in an automated manner.

**CARD Effects.** The type $E$ is the type of effects on the store. A value $e : E$ has a denotation $[e]$ which is an $S \rightarrow S$ function modifying a store value.

**Example 3.1.** In the bank account example, we have the effect type $E := \text{Add Nat} \mid \text{Sub Nat}$. Each effect is of the form $\text{Add } n$ or $\text{Sub } n$ for some positive integer $n$. The denotations of $\text{Add } n$ and $\text{Sub } n$ are given by $\lambda s. s + n$ and $\lambda s. s - n$, respectively.

**Consistency Guards.** The type $C$ is the type of consistency guards on the store which describe measures of “accuracy” for partial knowledge of the store value. Consistency guards are *semantic in nature*, i.e., they do not restrict the ordering of operations like traditional consistency models (e.g., sequential consistency, etc), but instead semantically restrict the updates to the store. Formally, a value $c : C$ has a denotation $[c]$ which is a two-state predicate (of type $S \times S \rightarrow \mathbb{B}$) relating the “global store value” ($s_g$) and a “local store view” ($s_r$) that some replica has. We will write $c(s_1, s_2)$ to mean $[s_1/s_g][s_2/s_r][c]$. We restrict all guards to be reflexive, as in $\forall s. c(s, s) = \top$ – a replica store view equal to the global store value represents complete knowledge of the store. Replicas and local store views are described fully in Section 6.

**Example 3.2.** In the running bank account example, the denotation of consistency guards have type $\text{Int} \times \text{Int} \rightarrow \mathbb{B}$. The guard $\text{LE} := s_g \geq s_r$ restricts the global store value to be at least as great as the local store value. Intuitively, we will use the guard LE to "guard" withdraw operations – any replica executing a withdraw operation will have a local store value that is at most the global value, ensuring that the withdraw does not decrease the balance below 0. Informally, this implies that we need to restrict the global value from being decreased by other withdraw operations once the local

Note that at the time of execution of $\text{withdraw}$, the guard $\text{Approve}$ would hold; however, the arrival of the Request and consequent re-evaluation of Approve would retroactively invalidate the guard. Thus $\text{Approve}$ must be in conflict with not just Approve, but also with Request, as it changes the behavior of Approve, potentially causing violation. We provide an algorithm that finds such chained conflicts in Section 5.2.

3 CONFLICT-AWARE REPLICATED DATATYPES

We define CARDs, an abstract model of replicated data stores, and executions based upon them.

3.1 CARDs

}. Vol. 1, No. 1, Article . Publication date: July 2018.
replica has decided on a value of balance for the current withdraw operation. Another guards we will use in the bank account examples is $\text{EQ} := s_g = s_r$.

**Effect Classes.** A CARD’s effect type $E$ will often generate an infinite set of effect values. For example, the Counter CARD includes an $\text{Add } n := \lambda s. s + n$ effect for all $n : \mathbb{N}$. In order to facilitate automated reasoning about effects and guards that is necessary for runtime locking decisions, we assume that this set of infinite effects are divided into a finite set $\overline{E}$ of **parametric effect classes**. The choice of classes must be made by the developer of the CARD, and is most effective when each class is characterized by the relationship to the set of relevant guards. In our examples, we assume that the type $E$ is a non-recursive algebraic data type, with values of each type variant being one class. We will elide this classification detail for the rest of the paper; when an algorithm quantifies $\forall e : E$ we assume that we are using a finite $E$ or a quantification over the finitely many parametric classes of $\overline{E}$.

**Example 3.3.** For the bank account example, the obvious choice is to classify effects by constructor: $\overline{E} := \{\text{Add}^y, \text{Sub}^y\}$ where $\text{Add}^y$ and $\text{Sub}^y$ include events of the form $\text{Add } n$ and $\text{Sub } n$, respectively. Each effect in the effect class behaves similarly with respect to the guards $\text{LE}$ and $\text{EQ}$. For example, all $\text{Sub } n$ effects may cause the condition $\text{LE} := s_g \geq s_r$ to be violated if the global store is updated with it, while $\text{Add } n$ cannot cause the same.

### 3.2 CARD Executions

Following standard practice (see [Burckhardt 2014; Burckhardt et al. 2012, 2014; Gotsman et al. 2016]), we describe the execution history of an eventually consistent replicated store using a set of **events** that each represent the execution of a single operation on the data store. Events contain an effect that changes the store and a return value that gives some information about the store back to the caller. In addition, CARD events contain a set of **active guards** that represent the semantic consistency restrictions on the event. Events are ordered by an **arbitration total order** in order to support CARDs with non-commutable effects. Such an order must be decided consistently by all members of the replicated store without coordination – time stamps and Lamport clocks can be used for this purpose, or it can be omitted in implementation for systems which only make commutable store updates.

**Active Guards.** Each event has a set of (zero or more) **active guards**, (or AGs for short). An event’s AGs represent consistency guards that a replica had when producing the effect. Since we will allow a replica to impose a series of consistency guards to produce one effect, each event might have more than one AG. Associated with each AG is the subset of previous events that the replica witnessed when it imposed the consistency guard. This encodes the standard visibility relation between an AG and an event. The AG is also associated with the consistency guard it represents.

**D-executions.** Formally, a **D-execution** for a CARD $D$ is a tuple $L = (s_0, W, G, \text{grd}, \text{ar}, \text{vis})$ where:

- $s_0 : S(D)$ is the initial store value
- $W$ is a finite set of events.
- $G$ is a finite set of active guards.
- $\text{grd} : W \rightarrow \mathbb{P}(G)$ gives the set of AGs for an event. Every AG is associated with a single event, which we denote by $\text{grd}^{-1} : G \rightarrow W$.
- $\text{ar} \subseteq (W \times W)$ is the arbitrary total ordering on events.
- $\text{vis} \subseteq (W \times G)$ is our guard-based visibility relation, which indicates whether an AG witnesses an event. We denote by $\text{vis}^{-1} : G \rightarrow \mathbb{P}(W)$ the set of all events witnessed by an AG.

A D-execution also defines the following functions for examining events and active guards:
• \( \text{eff} : W \rightarrow E(D) \) gives the \( D \)-effect an event holds
• \( \text{rval} : W \rightarrow A \) gives the return value (of some type \( A \)) an event holds
• \( \text{gc} : G \rightarrow C(D) \) gives the consistency guard an active guard was formed from.

**Example 3.4.** In our running bank account example, two instances of events can be:

- A withdraw event \( \eta_w \) with effect \( \text{eff}(\eta_w) = \lambda s. \: s - 10 \) reducing the store by 10 while returning the value \( \text{rval}(\eta_w) = 10 \) and guarded by a singleton active guard set \( \text{grd}(\eta_w) = \{ g \} \) which maintains consistency guard \( \text{gc}(g) = \text{LE} \) for the store with respect to 100, the store value it witnessed when \( \eta_w \) was being created.
- A deposit event \( \eta_d \) with effect \( \text{eff}(\eta_d) = \lambda s. \: s + 100 \) indicating that the effect of the event increases the store value by 100, while returning \( \text{rval}(\eta_d) = 100 \), and being (not) guarded by an empty set \( \text{grd}(\eta_d) = \emptyset \) indicating that the replica made no store queries when creating \( \eta_d \).

**Evaluations.** The store evaluation of a \( D \)-execution \( L \), written as \( \text{eval}(L) \), is the store value arrived at by starting with \( s_0 \) and applying \( \text{eff}(\eta) \) for each \( \eta \in W \) in \( \text{ar} \) order. Formally, if \( W = \{ \eta_0, \eta_1, \ldots, \eta_n \} \) with each \( i < j \implies \text{ar}(\eta_i, \eta_j) \), then \( \text{eval}(L) = (\text{eval}(\eta_n)) \circ (\text{eval}(\eta_{n-1})) \circ \cdots \circ (\text{eval}(\eta_0))(s_0) \).

**Example 3.5.** Continuing Example 3.4, given a \( D \)-execution \( L = (0, \{ \eta_w, \eta_d \}, G, \text{grd}, \text{ar}, \text{vis}) \) where \( \text{ar}(\eta_d, \eta_w) \), the store evaluation \( \text{eval}(L) \) is given by \( ((\lambda s. \: s - 10 \circ \lambda s. \: s + 100))(0) \), i.e., 90.

**Definition 3.6 (sub-executions).** We define a sub-execution of a \( D \)-execution \( L = (s_0, W, G, \text{grd}, \text{ar}, \text{vis}) \) as any other \( D \)-execution \( L' = (s_0, W', G', \text{grd}', \text{ar}', \text{vis}') \) for which \( W' \subseteq W, G' \subseteq G, \text{grd}' \subseteq \text{grd}, \text{ar}' \subseteq \text{ar}, \text{vis}' \subseteq \text{vis}, \) and \( \forall \eta \in W' \). \( \text{grd}(\eta) = \text{grd}'(\eta) \) (so that any remaining event retains all it’s active guards).

The above definition says that for \( L' \) to be a sub-execution, \( W' \) must retain any event that is visible to any guards remaining in \( G' \) (and thus which has “caused” any observable effect).

**Pre-Executions.** We define the pre-execution of an event \( \eta \) in a \( D \)-execution \( L \) as the sub-execution of \( L \)’s components to the events ordered by \( \text{ar} \) before \( \eta \), and we write this as \( L_{\eta} \) for short. The pre-store of \( \eta \) is then the evaluation of \( L_{\eta} \), and the post-store is \( \text{eval}(\text{eff}(\eta))(L_{\eta}) \). In further discussion, the global store value when an operation is being executed at a replica, refers to the pre-store value in the abstract execution (as per the arbitration order). Note that this global store value is not stored explicitly, and the replica executing an operation cannot learn the global store value without additional coordination with other replicas.

**Example 3.7.** Continuing Example 3.5, the pre-execution of \( \eta_w \) is given by \( L_{\eta_w} = (0, \{ \eta_d \}, \emptyset, \emptyset, \emptyset, \emptyset) \). The pre-store and post-store values are 100 and 90, respectively.

Similarly, we define the vis-execution of a guard \( g \) in a \( D \)-execution \( L \) as the pre-execution of \( L \)’s components to the events in \( \text{vis}^{-1}(g) \), and we write this as \( L_g \) for short. The vis-store of \( g \) is then the evaluation of \( L_g \).

**Well-Formed Executions.** We consider a \( D \)-execution well-formed if all of the following hold:

1. An event’s AGs can only be influenced by other events which are preceding (ar respects vis, causal consistency), i.e., \( \forall \eta_1, \eta_2 \in W. \: \forall g \in \text{grd}(\eta_1). \: \forall g_2 \in \text{vis}^{-1}(g). \: \text{ar}(\eta_1, \eta_2) \)
2. All AGs are satisfied, meaning that their pre-store and vis-store satisfy their consistency guard (guard-compliance), i.e., \( \forall \eta_1 \in W. \: \forall g \in \text{grd}(\eta_1). \: \text{gc}(g)(\text{eval}(L_{\eta_1}), \text{eval}(L_g)) \)
3. An AG that sees an event also sees the preceding events seen by that event’s AGs (transitivity of vis), i.e., \( \forall \eta_1, \eta_2 \in W. \: \forall g_2, g_3 \in G. \: \text{vis}(\eta_1, g_2) \land g_2 \in \text{grd}(\eta_2) \land \text{vis}(\eta_2, g_3) \Rightarrow \text{vis}(\eta_1, g_3) \)
Fig. 5. Terms, values, and dependent types of \( \lambda^Q \). The rules for deriving \( \tau \) types for terms are found in Figure 6. The \( k \) metavariable represents \texttt{Bool} and \texttt{Int} constants, and \( c \) represents consistency guards.

### Event Specifications

We specify correctness of events using constraints on the relation between the pre-store value \( s \) before the execution of the event, the post-store value \( s' \) after the execution of the event, and the return value \( a \) associated with the event. Formally, an event specification is a predicate \( \varphi \) of type \( S \times S \times A \rightarrow B \).

\[
\text{Definition 3.8 (Satisfaction of an Event Specification). An event } \eta \text{ in an execution } L \text{ satisfies an specification } \varphi, \text{ written } \eta \models_L \varphi, \text{ iff } \varphi \text{ holds for } \eta \text{'s pre-store as } s, \text{ } \eta \text{'s post-store as } s' \text{ and } \eta \text{'s return value as } a.
\]

\[
\eta \models_L \varphi \iff s = \text{eval}(L\eta) \land s' = \text{eff}(\eta)(s) \land a = \text{rval}(\eta) \Rightarrow \varphi(s, s', a)
\]

**Example 3.9.** For the running bank account example, we may want the properties that (a) the post-store value is non-negative, and (b) the change in the store value is equal to the return value of each event. The event specification \( \varphi(s, s', a) := s' \geq 0 \land s - s' = a \) exactly states this specification. Both the events \( \eta_w \) and \( \eta_d \) satisfy this specification: for example, in the case of \( \eta_w \), we have \( \psi \land s' = e(s) := s \geq 100 \land s - s' = 10 \implies s' \geq 0 \land s - s' = 10 := \varphi \).

In Section 4, we describe \( \lambda^Q \), a programming language for writing CARD operations, programs that dynamically produce an event based on a replicas (limited) knowledge of the current store value. The operational semantics of \( \lambda^Q \) operations only produce well-formed executions (Theorem 4.5). The type system of \( \lambda^Q \) can be used to check that an operation only produces events which satisfy a particular specification (Theorem 4.7). This property makes proving invariants straightforward (Theorem 4.8).

## 4 LANGUAGE AND TYPE SYSTEM FOR CARD OPERATIONS

In this section we describe the syntax, operational semantics, and refinement typing rules for \( \lambda^Q \), a core calculus language extending the CBV \( \lambda \)-calculus for defining CARD operations.

### 4.1 CARD Operations

The \( \lambda^Q \) syntax includes two special value terms that interact with a replicated store.

- **Query** The \( QC \triangleright x.t \) term defines an operation that queries the global store value up to the consistency predicate \( c \), binding the value to \( x \) before executing the sub-operation \( t \). As stated before, the global store value is not explicitly stored. Intuitively, to execute the query, a replica coordinates with other replicas ensuring that any effects that violate \( c \) are either arbitrated before the current operation, or after the current operation has finished executing.

- **Return+Emit** The \( R.(t_e, t_a) \) term defines a trivial operation which performs no query and applies \((t_e, t_a)\) as the operational result, in which \( t_e \) that is an effect emitted onto the store and \( t_a \) is a return value that is evaluated and returned to the caller. If the \( R \) term is nested inside a \( Q \) term, the effect and return values may include information read from the store.
Example 4.1. The basic withdraw bank account operation is expressed in λQ as follows:

\[
\text{withdraw} := λn. \ Q(s_g \geq s_r) \triangleright x. \ \text{if} \ (x > n) \ \text{then} \ R(\text{Sub} \ n, n) \ \text{else} \ R(\text{Add} \ 0, 0)
\]

Here, the global store value is queried up to the predicate \(s_g \geq s_r\), i.e., the value bound to \(x\) is at most the global value, and the operation is executed assuming that the store value is \(x\).

The more involved "strong" withdraw operation would be expressed as:

\[
\text{swithdraw} := λn. \ Q(s_g \geq s_r) \triangleright x. \ \text{if} \ (x > n) \ \text{then} \ R(\text{Sub} \ n, n) \ \text{else} \ Q(s_g = s_r) \triangleright x. \ \text{if} \ (x > n) \ \text{then} \ R(\text{Sub} \ n, n) \ \text{else} \ R(\text{Add} \ 0, 0)
\]

The first query and the then branch act as the standard withdraw operation, while the second query (with the stronger consistency predicate \(s_g = s_r\)) learns the exact value of the global store (forcing pending deposit operations to commit), and then executes the withdraw. This operation avoids the stronger coordination needed for the second, "full" query if it can work safely from just the first partial one, while still always making the withdrawal if it's absolutely possible.

For completeness, the deposit operation (which does not need a query) would be expressed as deposit := \(λn. \ R. \ (\text{Add} \ n, n)\).

\[
\begin{array}{c}
\text{TYPE}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{Γ} \vdash t : \tau
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{Γ}, x : \tau \vdash x : \tau
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{Γ}, x : \tau_1 \vdash t : \tau_2
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{Γ} \vdash \lambda x. t : \tau_1 \rightarrow \tau_2
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{Γ} \vdash \text{if} \ t_1 \ \text{then} \ t_2 \ \text{else} \ t_3 : \tau
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\bullet \vdash c : C
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{Γ}, x : \{v : S \mid [s/s_g][v/s_r][\{c\}]\} \vdash t : \text{Op} ((S, E, C), \ A, \varphi)
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{Γ} \vdash \text{Q}(c \triangleright x.t : \text{Op} ((S, E, C), \ A, \varphi)
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{Γ} \vdash t_e : E
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{Γ} \vdash t_a : \{v : A \mid s' = [t_e](s) \Rightarrow \varphi\}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{Γ} \vdash R.(t_e, t_a) : \text{Op} ((S, E, C), \ A, \varphi)
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\Gamma \vdash t : S_1 \quad \Gamma \vdash S_1 <: S_2 \quad \Gamma \vdash S_2 \quad \text{Valid}(\Gamma) \land \{t_1\} \Rightarrow \{t_2\}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{Γ} \vdash \{v : B \mid t_1\} <: \{v : B \mid t_2\}
\end{array}
\end{array}
\]

\[
\text{Fig. 6. Typing and sub-typing rules for } \lambda Q.
\]

4.2 Operation Types

The type system for λQ (detailed in Figure 6) extends Liquid Types [Rondon et al. 2008] on the CBV λ-calculus. For those unfamiliar, liquid types refine standard types with predicates on the values. For example, the typing judgement \(t : \{v : \text{Int} \mid x > 5\}\) asserts that the term \(t\) is an integer, as well as that the value is greater than 5.

In Figure 6, standard terms in the language are typed as per standard liquid types, while CARD operations are typed under a special \(\text{Op}\) type. The typing judgement \(t : \text{Op}(D, A, \varphi)\) indicates that \(t\) is an operation for the CARD \(D\) that returns a value of type \(A\) and that any \(D\)-execution event that results from the operation satisfies the event specification \(\varphi\).

Intuitively, the \(\text{TYPE}_Q\) rule is similar to a conditional guard rule: if a term \(t\) is of type \(\text{Op}(D, A, \varphi)\) given the additional premise \([c]\), the term \(Qc \triangleright x.t\) is of type \(\text{Op}(D, A, \varphi)\). The \(\text{TYPE}_R\) rule derives our \(\text{Op}\) type for a base \(R\) term from a standard Liquid Type judgment, stating that the return value and the denotation of the effect in the \(R\) term must together (in the logical constraint context of
We first follow the derivation in Figure 7, storing in the context the constraint on which may look strange since \(x \geq n\) that the query on \(\text{LE}\) can verify by hand or with an SMT solver, and in which we are aided by the opposite. Like the query constraints, these assumptions are added to the context.

\[
\begin{align*}
\text{n : Nat, x : } & \{v : S(\text{Counter}) \mid x \leq s\} \vdash (x \geq n) : \text{Bool} \\
\text{n : Nat, x : } & \{v : S(\text{Counter}) \mid x \leq s, x \geq n + \{\ldots(\text{then})\} \\
\text{n : Nat, x : } & \{v : S(\text{Counter}) \mid x \leq s, \neg(x \geq n) + \{\ldots(\text{else})\}
\end{align*}
\]

\[
\begin{align*}
\text{n : Nat, x : } & \{v : S(\text{Counter}) \mid x \leq s\} \vdash \{\text{if} \ldots\} : \text{Op(Counter, Int, } \varphi) \\
\text{n : Nat} & \vdash \text{LE} : C(\text{Counter}) \\
\text{n} & \vdash Q : \text{LE} \triangleright x. \{\text{if} \ldots\} : \text{Op(Counter, Int, } \varphi) \\
\text{\bullet \vdash } & \lambda n. \text{Q LE} \triangleright x. \{\text{if} \ldots\} : (n : \text{Nat}) \rightarrow \text{Op(Counter, Int, } \varphi)
\end{align*}
\]

Fig. 7. Derivation of withdraw type down to branches with base \(R\) terms.

\[
\begin{align*}
\Gamma^+ & \vdash n : \text{Nat} \\
\Gamma^+ & \vdash \text{Sub } n : E(\text{Counter}) \\
\Gamma^+ & \vdash R. (\text{Sub } n, n) : \text{Op(Counter, Int, } \varphi)
\end{align*}
\]

Fig. 8. Derivation of \(R\) term for withdraw’s success branch down to standard Liquid Type.

\[
\begin{align*}
\text{Valid}([\Gamma] \land [\text{Nat}] & \Rightarrow [(v : \text{Int} \mid s' = [\text{Sub } n](s) \Rightarrow \varphi)]) \\
\Gamma^+ & \vdash \text{Nat} < : \{v : \text{Int} \mid s' = [\text{Sub } n](s) \Rightarrow \varphi\} \\
\Gamma^+ & \vdash n : \{v : \text{Int} \mid s' = [\text{Sub } n](s) \Rightarrow \varphi\}
\end{align*}
\]

Fig. 9. Derivation for one of withdraw's Liquid Type obligations into logical constraint problem.

\(\Gamma^+\) ensure the \(\text{Op}\) type’s \(\varphi\) specification holds. The refinement part of this Liquid Type judgment becomes a simple logical constraint problem according to the rules in Figure 6. In these rules, \(<:\) is the "subtype" relation, which states that the left hand side has the same basic type as the right hand side, and that the left’s refinement implies the right’s refinement. The denotational brackets on \([\Gamma]\) reduce the context to the set of logical statements contained in its refinements.

**Example 4.2.** As an end-to-end demonstration, we now type-check the withdraw operation according to the specification we have been using, for which

\[
\varphi := (s \geq 0 \Rightarrow s' \geq 0) \land (a = s - s')
\]

We first follow the derivation in Figure 7, storing in the context the constraint on \(s\) (the pre-store value) that the query on \(\text{LE}\) gives us. This produces two unsolved branches, one for the then branch of the if term on which we can assume \(x \geq n\), and one on the else branch where we assume the opposite. Like the query constraints, these assumptions are added to the context.

We now elide the trivial else branch and follow the then branch, referring to the context so far (including \(x \geq n\)) as \(\Gamma^+\), in Figure 8. This takes us to the standard Liquid Type obligation

\[
\Gamma^+ \vdash n : \{v : \text{Int} \mid s' = [\text{Sub } n](s) \Rightarrow \varphi\}
\]

which may look strange since \(n\) already has the type \(\text{Nat}\) in \(\Gamma^+\). This is where, in Figure 9, we use the Liquid Type subtyping rules to reduce the obligation to a logical constraint problem which we can verify by hand or with an SMT solver, and in which we are aided by the \(s\) constraint from our
guarded query:
\[ ([\Gamma^+] \land [\text{Nat}]) \implies [(v : \text{Int} | s' = [\text{Sub } n](s) \implies \varphi)] = \]
\[ (n \geq 0) \land (x \leq s) \land (x \geq n) \land (s' = s - n) \implies (s \geq 0 \implies s' \geq 0) \land (a = s - s') \]

Deciding this as valid, we have thus verified that withdraw has our desired behavior in a concurrent setting.

### 4.3 Operation Executions

\(\lambda^Q\) follows the standard semantics of the CBV \(\lambda\)-calculus for evaluating standard terms (terms with standard refinement types, excluding the Op type). We use the judgement \(t \Downarrow_\lambda t'\) to represent the standard big-step semantics for CBV \(\lambda\)-calculus.

Operations, i.e., terms of type \(\text{Op}(D, A, \varphi)\), cannot be evaluated in a pure setting. Rather, they are executed by replicas, which may query values from the global replicated store. The state of the operational evaluation is represented by \((s, \psi, t)\) where \(s\) is the the global store value, \(\psi\) is the accumulated active consistency guard, and \(t\) is the term to be evaluated. Each execution step is described abstractly by the operation execution rules (Figure 10):

- **Query-evaluation step:** A query evaluation step represents a replica executing \(Qc \ni x. \ t\), i.e., querying the evaluation global store under the query predicate \(c\), and evaluating the term \(t\) with \(x\) bound to the value of the query. The replica obtains (non-deterministically, at this level) a value \(s_x\) such that \(c(s_x, s_r)\) holds, and the value of \(\psi\) is updated with \([s_x/s_r]c\) and the resulting term is obtained by substituting the value \(s_r\) in \(t\).

- **Drift step:** A drift step represents the value of the global store value changing due to the execution of a different replica. However, the \(\psi\) value in the execution context restricts the change so that snapshots which have been substituted into \(t\) (by steps of the query rule) remain consistent according to the guards they were queried with. Note that this rule makes the execution non-deterministic.

Fully executing an operation \(t\) with type \(\text{Op}(C, A, \varphi)\) from \((s, \psi, t)\) produces \((s, \psi, R(e, a))\) where \(a : A\) is the return value and \([e](s)\) is the final value of the global store. By the soundness of liquid types, we get that \((s, [e](s), a) \models \varphi\).

\[
\begin{align*}
(s, s_x) &\models c & [s_x/x]t &\Downarrow_\lambda t' \\
(s, \psi, Qc \ni x.t) &\mapsto (s, \psi \land [s_x/s_r]c, t') & \text{QUERY} & \exists e : E. s' = [e](s) & s' &\models \psi & \text{DRIFT}
\end{align*}
\]

**Example 4.3.** We describe one execution each of the deposit, withdraw, and strong withdraw operations in the bank account example. The steps resulting from query and drift steps are superscripted with \(Q\) and \(D\), respectively:

- The evaluation of deposit 100 can produce the following sequence: \((0, \top, \text{deposit } 100) \mapsto^Q (0, \top, \text{R(Add } 100, -100))\)
- The evaluation of withdraw 10 can produce the following sequence: \((0, \top, \text{withdraw } 10) \mapsto^D (100, \top, \text{withdraw } 10) \mapsto^Q (100, s_g \geq 100, \text{R(Sub } 10, 10))\)
- The nested queries in withdraw lead to multiple query steps in the evaluation. The following is a valid evaluation sequence: \((0, \top, \text{withdraw } 10) \mapsto^Q (0, s_g \geq 0, t_{iq}) \mapsto^D (100, s_g \geq 0, t_{iq}) \mapsto^D (90, s_g \geq 0 \land s_g = 90, \text{R(Sub } 10, 10))\) where \(t_{iq} := Q(s_g = s_r) \ni x.\ \text{if } (x > 10) \text{ then } \text{R(Add } 0, 0)\).
Combining Multiple Operational Executions. The operation execution rules produce a sequence of evaluation steps corresponding to the invocation of a single operation. We now describe how a number of different (possibly concurrent) operation invocations correspond to a CARD execution. Intuitively, the CARD execution must be produced by combining the update steps of an operation execution for each invocation. The drift steps in the operation execution of \( t \) correspond exactly to the updates of all the operations arbitrated before the effect produced by \( t \), and the query steps must take as their \( s_\lambda \) value a post-store of some subset of the effects arbitrated before.

Given a set of \( D \)-operation invocations \( T \) with op : \( T \rightarrow Op(D, A, \varphi) \) giving the operation term for each invocation, we say a CARD execution \( L = (s_0, W, G, \text{grd}, \text{ar}, \text{vis}) \) is produced by \( T \) iff there exists a one-to-one correspondence between events \( \eta_i \in W \) and operation invocations \( t_i \in T \) such that:

- there exists an operation execution for \( t_i \) of the form \( (s_0, T, \text{op}(t_i)) \rightarrow^* (s_i, \psi_i, R(e_i, a_i)) \) in which \( \psi_i = [s_{x_0}/s_r]c_0 \wedge [s_{x_1}/s_r]c_1 \wedge \ldots \wedge [s_{x_n}/s_r]c_n \),
- \( \text{grd}(\eta_i) \) contains \( n \) active guards corresponding to the \( n \) clauses in \( \psi_i; \varphi_j \) corresponds to clause \( [s_{x_j}/s_r]c_j \) in \( \psi_i \) such that gc(\( \varphi_j \)) = \( \varphi_j \), eval(\( L_{\eta_i} \)) = \( s_j \) and \( \text{vis}^{-1}(\varphi_j) \) includes the \( \text{vis}^{-1} \) set of each guard event in \( \text{vis}^{-1}(\varphi_j) \).
- the drift steps in \( t_i \)'s operation execution correspond, in order, to the preceding events in \( L_{\eta_i} \) such that for \( \eta_j \in L_{\eta_i} \), eff(\( \eta_j \)) is the effect quantified in the corresponding drift step’s premise,
- eval(\( L_{\eta_i} \)) = \( s_i \),
- eff(\( \eta_i \)) = \( e_i \), and
- rval(\( \eta_i \)) = \( a_i \).

Example 4.4. The operational executions of the deposit, withdraw and strong withdraw operations from Example 4.3 can produce the abstract execution \( L = (0, \{ \eta_d, \eta_w, \eta_{sw} \}, G, \text{grd}, \text{ar}, \text{vis}) \) where: (a) \( \eta_d := (T, \lambda s. s + 100, -100) \), (b) \( \eta_w := (s_g \geq 100, \lambda s. s - 10, 10) \), and (c) \( \eta_{sw} := (s_g \geq 0 \wedge s_g \geq 100, \lambda s. s - 10, 10) \). The exact correspondence between the abstract execution and the operational executions is depicted in Figure 11.

Theorem 4.5 (Well-Formedness of Operation Executions). Any \( D \)-execution \( L \) that is produced by a set of \( D \)-operations \( T \) is well-formed (by the definition in Section 3.2).

Proof. The non-trivial part is guard-compliance. We prove guard-compliance by induction on the operation execution step sequence corresponding to each event \( \eta_i \) with I.H. \( s \models \psi \).

- Base: \( s = s_0, \psi \) is empty, trivially satisfied.
- Step with query: I.H. gives \( s \models \psi \), query premise gives \( s \models [s_x/s_r]c \), thus \( s \models \psi \wedge [s_x/s_r]c \).
- Step with drift: Premise gives \( s' \models \psi \).
The pre-store of $η$ must be equal to the $s$ value of it’s operation’s final context because each event in $L_0$ applies the same effect as its corresponding DRIFT step. The vis-store of any $g \in \text{grd}(η)$ is equal to the $s_x$ in it’s $ψ$ clause by definition of producing a $D$-execution. Thus all guards in $L$ are satisfied by their pre-store and vis-store values. □

**Theorem 4.6 (Preservation for Operation Executions).** For any derived term $Γ ⊢ t : Op(C, A, φ)$ and starting state $s_0$, if an operation execution $(s_0, τ, t) \mapsto σ(s′, ψ′, R(e, a))$ exists, then $ψ′ \Rightarrow φ(s′, [c](s′), a)$.

**Proof.** We must show that $ψ$ is made strong enough to guarantee $φ$ for a term $Γ ⊢ t : Op(D, A, φ)$. We begin by inductively evaluating and analyzing the type derivation of $t$ side by side, showing that at each step, $[Γ] \Rightarrow ψ$.

**Base:** $Γ = ψ = T$.

**Case $Q c \triangleright x. t′$:** We evaluate this term by a query step, adding $[s_x/s_r]c$ to $ψ$ and replacing $x$ with $s_x$ in $t′$. We type this term by the TYPE Q rule, adding $[x/s_r]c$ to $[Γ]$. So our knowledge of $s_x$ in the evaluated $t′$ is matched by our knowledge of $x$ in the typed $t′$, and $([Γ] \Rightarrow ψ) \Rightarrow (Γ \wedge [x/s_r]c) ⇒ ψ \wedge [s_x/s_r]c$.

**Case (any other):** This term is evaluated by the standard λ-calculus rules and does not add any obligations to $ψ$.

We have thus evaluated $t$ to a configuration $(s, ψ, R. (t_e, t_a))$ and followed its type derivation to a term $Γ ⊢ R. (t_e, t_a) : Op(D, A, φ)$ such that $[Γ] ⇒ ψ$ (when $x$’s in $Γ$ are replaced with their corresponding $s_x$’s). The remaining obligation of the type derivation shows that the contents of $[Γ]$ ensure that the final term satisfies $φ$ under any compatible store value, and so $ψ$ must be strong enough to ensure the same (Def. 3.8). □

**Theorem 4.7 (Produced D-Events Satisfy Operation Specifications).** Given an operation invocation $t_i$ in a set of invocations $T$ for which $Op(t_i) : Op(D, A, φ)$, the event $η_i$ corresponding to $t_i$ in any $D$-execution $L$ produced by $T$ via the operation execution rules satisfies $φ$ (in the sense of Def. 3.8).

**Proof.** By Theorem 4.6, we know that for any operation execution step sequence for $t_i$ ending with $(s′, ψ′, R. (t_e, t_a))$, we have $ψ′ \Rightarrow φ(s′, [t_i](s′), t_a)$. And so have this statement for the operation execution sequence that produces $η_i$, for which $\text{eff}(η_i) = t_e$ and $\text{rval}(η_i) = t_a$. The guards in $\text{grd}(η_i)$ are together satisfied by the same store values that $ψ′$ is satisfied by, and so guard compliance (a component of well-formedness of $L$, which we have by Theorem 4.5) ensures that $ψ′(\text{eval}(L_{η_i}))$. Thus for $s = \text{eval}(L_{η_i})$ we have $φ(s, [eff(η_i)](s), rval(a))$, meaning that $η \models L φ$. □

Because operation-produced events respect to their specifications, it is easy to show that invariants can be maintained.

**Theorem 4.8 (Execution Invariants).** Given a $D$-store predicate $I$ and a set of $D$-operation invocations $T$, each of which has a type which includes $I(s) ⇒ I(s′)$ in its specification, any $D$-execution, which is produced by $T$ and for which $I(s_0)$ holds, preserves $I$.

**Proof.** This follows immediately from Theorem 4.7. Every event in the produced execution will respect $I(s) ⇒ I(s′)$, and so $I$ is preserved over each effect application.

**Example 4.9.** Suppose we want to ensure that the invariant $I := s ≥ 0$ holds for the bank account example, i.e., that the account value is always non-negative. The key insight from Theorem 4.8 is that the task of ensuring this invariant can be split into guaranteeing two separate properties:

- the system only produces events that are sound for the specification $I(s) ⇒ I(s′)$, and
- the executions are well-formed.
withdraw 1
withdraw 1
withdraw 10
withdraw 5
deposit 10

Fig. 12. Blind luck executions. There can be executions which have events of not-in-accord operations that are invisible to one another, and still produce a well-defined result.

For example, if every event produced by the system is in one of the forms of $\eta_w$ or $\eta_d$ from Section 3 (with the constants 10 and 100 replaced by any non-negative integer), all these events are guaranteed to be sound for the specification. Further, the system would need to ensure that these events are executed only in the contexts where the guards hold.

5 INFERRING CONFLICT AVOIDANCE REQUIREMENTS

The specifications verified for operations in Section 4 depend on query guards being maintained while concurrent events enter the execution history. It is simple to state this requirement in the operation execution rules, in which each new event is appended in order to the evolving store value, but we need a more complete picture of effect-guard interactions in order to design a realistic system in which events will appear to replicas out of order.

5.1 Measures of Non-Conflict

First, we define the following notion of an immediate accord between an effect and a guard. An immediate accord existing between an effect $e : E$ and a guard $c : C$ implies that the effect updating the global store cannot violate the consistency guard in an execution of an action bound by a $c$ query, i.e., actions of the form $Qc \triangleright x.t.$

**Definition 5.1 (Immediate Accord).** Given a CARD $D = (S, E, C)$, guard $c : C$ and effect $e : E$, an immediate accord exists between them, written as $IA(c, e)$, iff

$$\forall s_g, s_r : S. c(s_g, s_r) \implies c(\begin{bmatrix} e \end{bmatrix}(s_g), s_r).$$

We denote by $IAS_D(c)$ the set of all $D$-effects in immediate accord with $c$.

**Example 5.2.** In the running example, there is an immediate accord between the effect $Add n$ and the guard $LE$. However, there is no immediate accord between $Add n$ and $EQ$, or between $Sub n$ and either of $EQ$ and $LE$.

**Definition 5.3 (Careful Executions).** We call a $D$-execution $D = (L, G, grd, ar, vis)$ careful iff for each $g \in G$ guarding an event $\eta$, $L_g$ contains all events $\eta_i$ in $L_\eta$ for which $eff(\eta_i)$ is not in immediate accord with $gc(g)$.

A careful execution is always produced when a replica resolving a query must see every event in the network which is not in immediate accord with its guard. This safety measure over-approximates the guard satisfaction condition followed by the operation rules by excluding invisible subsets that satisfy the guard “by blind luck”, such as an invisible account-emptying withdrawal followed by an invisible deposit that undoes it (see Figure 12). Intuitively, allowing an undetected "lucky pair" also allows an undetected “unlucky single” which would make the query resolution unsound. We thus use the careful, well-formed $D$-execution as our basis for the following definitions.

**Transitive Accords.** As illustrated in Section 2 (the joint account CARD), it is not sufficient for a replica maintaining $c$ to coordinate with replicas concurrently emitting effects $e \notin IAS_D(c)$. A second effect $e' \in IAS_D(c)$ that is concurrent to $e$ might change the behavior of $e$ if it is arbitrated earlier. Hence, we now describe a stronger notion of accords.
We begin by deciding the immediate accord set of \( c \):

Therefore, the immediate accord set of \( TA = M \) and an effect \( D \) meaning that \( gc(g) = c \) and for any event \( \eta \neq W \) for which \( eff(\eta) = e \), the guard \( g \) remains satisfied in \( L' = (W \cup \{\eta\}, G, grd, ar \cup \{(\eta, \eta)\}, vis) \).

A **transitive accord set** for \( c \) is a set of effects for which transitive accords exist. Intuitively, any replica maintaining a guard \( c \) needs to coordinate with replicas emitting effects which are not in its transitive accord set because a new event arriving at the replica may be inserted somewhere in the middle of history by the arbitrary ordering. The following theorem states that finding the largest transitive accord set is undecidable.

**Theorem 5.5.** Given a CARD \( D \) and \( D \)-guard \( c \), finding the largest cardinality transitive accord set for \( c \) is undecidable.

**Proof.** Sketch: the proof relies on constructing an effect \( e \) which can induce a violation of the guard \( g \) only from a single store state. Now, \( e \in TAS(c) \) if and only if that single store state is reachable through the effects of the system. Such store value reachability problems are undecidable.

**Example 5.6.** In the joint bank account example, let’s intuit the transitive accord set for the guard of withdraw\( J \), \( c = LE \land App? \). Recall that the state is expressed as a tuple \((s : \text{Int}, b_1 : \text{Boo1}, b_2 : \text{Boo1})\), and that \( \text{withdraw} J := Q \text{LE} \land \text{App}>(\cdots) \), where \([\text{LE}] := s(s_r) \leq s(s_g)\) and \([\text{App}?] := b_2(s_r) = b_2(s_g)\). We begin by deciding the immediate accord set of \( c, \text{IAS}_D(c)\):

- \([\text{Request}] \) only changes \( b_1 \), which is not used in either of \( \text{withdraw} J \)’s guards. Therefore the effect is in \( \text{IAS}_D(c) \).
- \([\text{Approve}] \) and \([\text{Reset}] \) can both change \( b_2 \), violating \( \text{App}? \), so neither is in \( \text{IAS}_D(c) \).
- \([\text{Add n}] \) only increases \( s \), satisfying \( LE \) and \( \text{App}? \) (trivially), so it is in \( \text{IAS}_D(c) \).
- \([\text{Sub n}] \) and \([\text{Set n}]\) can both decrease \( s \), violating \( LE \), so neither is in \( \text{IAS}_D(c) \).

Therefore, the immediate accord set of \( LE \land App? \) contains \( \text{Request} \) and \( \text{Add n} \). Now let’s see which of these two is also in the transitive accord set. Notice the presence of an additional \( \text{Add n} \) can never decrease \( s \), even when combined with other rules. Nor can it change \( b_2 \). This shows that \( TA(\text{Add n}, c) \). Request is more complicated, since it toggles \( b_1 \), which sets \( b_2 \) when combined with Approve. Consider an abstract execution consisting of an Approve followed by a Withdraw 10. Because there is no request, \( b_1 = \perp \), the Approve will keep \( b_2 = \perp \). This will result in the Withdraw 10 acting as a NoOp. Now suppose we produce a new execution using the same events preceded by a Request. This time \( b_1 = \top \), and could lead to the withdrawing being executed. Therefore, the only effect with a transitive accord with \( LE \land App? \) is \( \text{Add} \).

### 5.2 Inferring Minimal Locking Conditions

**Consistency Invariants.** A **consistency invariant** in a CARD \( D \) is a \( D \)-guard \( c \) for which, given any pair of \( D \)-states \((s_g, s_r)\) and \( D \)-effect \( e \), \( c(s_g, s_r) \Rightarrow c(\llbracket e \rrbracket(s_g), \llbracket e \rrbracket(s_r)) \).

**Theorem 5.7 (CINV + IA = TA).** For a CARD \( D = (S, E, C) \), if a \( c : C \) is a consistency invariant in \( D \) and an effect \( e : E \) is in immediate accord with \( c \), then \( e \) is also in transitive accord with \( c \).

**Proof.** Suppose we have a careful, well-formed \( D \)-execution \( L \) containing event \( \eta \) and active guard \( g \in \text{grd}(\eta) \) for which \( gc(g) \) is a consistency invariant. As \( L \) is well-formed, \( g \) is satisfied, meaning that \( gc(g)(\text{eval}(L), \text{eval}(L_g)) \) holds.

We now take a new event \( \eta' \) for which \( IA(gc(g), eff(\eta_m)) \) holds and create a new execution \( M = (W \cup \{\eta_m\}, G, \text{grd}, ar \cup \{\eta_m \times \eta\}, \text{vis}) \). Showing that \( g \) is also satisfied in \( M \) is proof that \( TA(gc(g), eff(\eta_m)) \) holds. We show this by inductively evaluating \( M_g \) and \( L_g \) alongside each other.

, Vol. 1, No. 1, Article . Publication date: July 2018.
and noting that at each step, the post-states of the two sub-executions satisfy $g$’s consistency guard. This will give us that $gc(g)(\text{val}(M_\eta), \text{val}(L_g))$, showing that $g$ is satisfied in $M$.

At the base case, $gc(g)(s_0, s_0)$ holds by definition of consistency guards (they are always implied by equality). For our inductive step, we examine an event $\eta'$ which is in some combination of the executions $L_\eta, M_\eta$, and $L_g$, with $gc(g)(s_M, s_L)$ as our inductive hypothesis:

**Case** $\eta' \in L_\eta \cap M_\eta \cap L_g$: The fact that $gc(g)$ is a consistency invariant gives us $gc(g)(\langle\text{eff}(\eta')\rangle(s_M), \langle\text{eff}(\eta')\rangle(s_L))$.

**Case** $\eta' \in L_\eta \cap M_\eta \land \eta' \notin L_g$: Because $L$ is careful and $\eta'$ is not in $L_g$, we must have $\text{IA}(gc(g), \text{eff}(\eta'))$.

This gives us $gc(g)(\langle\text{eff}(\eta')\rangle(s_M), s_L)$.

**Case** $\eta' \in M_\eta \land \eta' \notin L_\eta \cup L_{x_1}$: This can only be our new event $\eta_m$ for which we have $\text{IA}(gc(g), \text{eff}(\eta_m))$ by assumption. This gives us that $c(\langle\text{eff}(\eta')\rangle(s_M), s_L)$.

Thus we have $\text{TAS}(gc(g), \text{eff}(\eta_m))$ because $g$ remains satisfied when $\eta_m$ is added to $L$. □

Consistency invariants for CARDS play the role equivalent to standard inductive loop invariants in sequential program verification — they are a strengthening of the required property that is preserved by operations. We show that every consistency invariant that implies a given $c$ defines a transitive accord set for $c$.

**Theorem 5.8.** Let $D$ be a CARD and $c$ and $c'$ be $D$-guards. If $c'$ is a consistency invariant and $c' \Rightarrow c$, then $\text{IAS}_D(c')$ is a transitive accord set for $c$.

Note that the identity relation itself ($=$) is always a consistency invariant, similar to how $\bot$ is always a loop invariant in the sequential setting. However, this consistency invariant leads to a transitive accord set that rejects all state mutating effects in the CARD. The challenge is to identify the consistency invariant that leads to the most complete transitive accord set.

In spite of Theorem 5.5, we present a simple semi-procedure that computes a reasonable transitive accord set in practice through consistency invariants. First, let the *weakest consistency precondition* of a guard $c$ and effect $e$, $\text{WCP}(e, c)$, be the weakest guard such that $(s_g, s_r) \models \text{WCP}(e, c)$ implies that $(\langle e \rangle(s_g), \langle e \rangle(s_r)) \models c$. Now, we decide transitive accords with:

$$\text{TAS}_D(c) := \text{let } c' = \bigwedge_{e : E} \text{WCP}(e, c) \text{ in if } c \Rightarrow c' \text{ then } \text{IAS}_D(c') \text{ else } \text{TAS}_D(c \land c')$$

The following theorem states the soundness of the above procedure.

**Theorem 5.9.** Given a CARD $D$, a $D$-guard $c$, and a $D$-effect $e$, the procedure $\text{TAS}_D(e, c)$ returns a transitive accord set for $c$.

**Proof.** The proof follows from the following:

- The guard argument at recursive call $i$ (which we will call $c_i$) is a strengthening of $c$.
- If, at recursive call $i$, the condition $c_i \Rightarrow c'$ holds, then $c_i$ is a consistency invariant in $D$ because $\forall e : E. c_i \Rightarrow \text{WCP}(e, c)$.
- Therefore, because $c_i \Rightarrow c$ and $c_i$ is a consistency invariant, then the returned $\text{IAS}_D(c_i)$ is a transitive accord set for $c$ by Theorem 5.8.

The procedure $\text{TAS}$ is computing the greatest fixed-point $c'_L$ of the equation $\mu c' : c' \Rightarrow c \land ((s_g, s_r) \models c') \Rightarrow \bigwedge_{e : E} \langle e \rangle(s_g), \langle e \rangle(s_r) \models c'$ as a consistency invariant and using it to decide transitive accords. However, any fixed-point of the equation is sufficient, and any technique used in standard sequential program reasoning can be applied to compute this fixed-point (e.g., widening from abstract interpretation, logical interpolant computation, etc).
6 IMPLEMENTING A REPLICA NETWORK

In this section, we show how inferred locking conditions can be used to implement a network of replicas that correctly execute concurrent CARD operations. In Figure 13, we detail the small-step semantics by which a network of replicas executes operations, which refines the behavior of the previously defined operation execution rules. The semantics leverage the transitive accord sets computed using the procedure detailed in Section 5.

Replica-network State. We represent the state of a replica in the network as \((r, h_r, ts)\) where:

(a) \(r\) is the unique replica id; (b) \(h_r\) is the replica’s view of the network execution history, initially set to the empty history; and (c) \(ts\) is the sequence of operations yet to be executed, initialized non-deterministically to the set of operations a replica will execute. The state of a network is given by \((h, ls, rs)\) where:

- The history \(h\) is an set of events of the form \(v = \text{event}(r, e, a, h_r)\), in which \(r\) is a unique replica ID, \(e\) is an effect, \(a\) is a return value for the operation, and \(h_r \subseteq (h \setminus \{v\})\) is the part of the history that influenced the creation of the event \(v\). The values \(h_r\) together represent a DAG of events ordered by happens-before. Delivering an event \(v\) to a replica requires that the replica already has all events it depended upon, such that causal consistency is maintained. Note that the history \(h\) is not explicitly stored in any replica, and cannot be directly read.
- The locking configuration \(ls\) is a map of replica ID \(r\) to guard \(c\), which describes the network constraints which need to be maintained in order to preserve the assumptions of operations currently under execution. \(\text{permits}(ls, e)\) determines whether an effect \(e\) can possibly invalidate any \(c\) in \(ls\), and precisely states that \(e\) is in transitive accord (see Section 5) with all guards in \(ls\) except the emitting replica’s. Possible implementations of a decision procedure for \(\text{permits}(ls, e)\) are described in Section 5.
- The replica set \(rs\) is the set of replicas in the network. Each replica has a unique ID \(r\), its own partial view of history \(h_r \subseteq h\), and a sequence of operations to execute \(ts\).

The explicit replica execution rules are shown in Figure 13.

- **Lock acquisition.** The \(R\_Lock\) rule describes the precise condition for a lock acquisition. The rule adds the guard \(c_2\) to the replica’s guards in the lock state, in the scenario that there are no events that are present in the network, but not in the replica history whose effects are not in transitive accord with \(c_2\). In practice, implementing this rule involves communicating with each replica in the network, gathering any effects not in \(TAS_D(c_2)\), and acquiring a license from each of them.
- **Operation evaluation.** The \(R\_Query\) rule describes the local execution of the operation in the replica.
- **Effect emission.** In case the lock state permits the emission of the effect, the effect emission rule adds an event to the local history. A lock state permits an emission of effect \(e\) if no replica has a lock on a guard \(c\) such that \(e \notin TAS_D(c)\). Once the effect is emitted, the lock state is updated by removing all locked guards for the replica \(r\).
- **Effect delivery.** The effect delivery rule transmits an effect that is in the network history into the local history of a replica.

Locking protocol. The replica rules we present here are declarative; they specify when a replica is allowed to proceed with locking or querying but do not give instructions for actively getting to that state. For this purpose we can use any distributed locking protocol. A simple locking scheme would require a replica making a query to contact all other replicas, requesting from them an agreement to not emit effects that could violate the querying replica’s guard, and further to immediately send all already-emitted effects that could violate it. Upon finishing its operation, the querying replica
produces a CARD execution. Given a set of replicas $R$ with $D$-operation invocation sequences on a replica network, the invocations make a partially ordered set ($T$, $\leq$) where $t_1 \leq t_2$ iff $t_1$ occurs before $t_2$ in the invocation sequence on a single replica and with $\text{op} : T \rightarrow \text{Op}(D, A, \phi)$ giving the operation term for each invocation. Then we say a CARD execution $L = (W, G, \text{grd}, \text{ar}, \text{vis})$ is produced by $(T, \leq)$ from the replica network execution rules iff there exists a one-to-one correspondence between events $\eta_i \in W$ and operation invocations $t_i \in T$ such that:

- $t_1 \leq t_2 \Rightarrow \text{ar}(\eta_1, \eta_2)$,
- there exists a replica execution for $R$ which fully evaluates all operation invocations,
- the $\text{R\_EMIT}$ replica execution step for an invocation $t_i$ of the form:

  $$(h|ls, r : c_1|rs, (r, h_r, Q c_2 \triangleright x.t :: ts)) \twoheadrightarrow (h|ls, r : c_1 \land c_2|rs, (r, h_r, Q c_2 \triangleright x.t :: ts))$$

  corresponds CARD event $\eta_i$ in that $\text{eff}(\eta_i) = e$ and $\text{eval}(\eta_i) = a$,
- the $\text{R\_EMIT}$ step for $t_i$ is preceded by $n$ $\text{R\_QUERY}$ steps of the form:

  $$(h|ls, r : c_1|rs, (r, h_r, Q c_2 \triangleright x.t :: ts)) \twoheadrightarrow (h|ls, r : c_1|rs, (r, h_r, [\text{eval } h_r/x]t :: ts))$$

  on $t_i$’s replica which correspond to the $n$ guards in $\text{grd}(\eta_i)$ such that $\text{vis}^{-1}(g_n)$ contains all events in $h_r$ and $\text{gc}(g) = c_2$.

**Theorem 6.1 (Replicas Implement Operation Rules).** If a CARD execution $L$ is produced by a partially ordered set of operations $(T, \leq)$ via the replica rules, then $L$ is also produced by $T$ via the operation rules.

**Proof.** We generate an operation execution rule sequence for each invocation $T$ from our replica rule sequence and show that it is a proof that $L$ is produced by $T$.

The operation rule sequence for $t_i \in T$ is created from the replica rule steps leading up to $t_i$’s $\text{EMIT\_R}$ step as follows:
• For every step in the operation rule sequence, the starting and ending $s$ values are the evaluations of the starting and ending $h$ values of the replica rule step that generated it.
• An emit\_r step for a different invocation creates a drift step. We satisfy the "$\exists e$" premise of the drift step using the $e$ that is emitted in the corresponding emit\_r step. The $p(r, e)$ premise ensures that $e(s) = \psi$, because every $[s_e/s_r]c$ clause in $\psi$ is overapproximated by the inclusion of $c$ in $\ell(s)$ of the replica rule context.
• An r\_query step that contributes to the evaluation of $t_i$ creates a query step. The $(s, s) \models c$ premise of query is guaranteed because the r\_query step was preceeded by a r\_lock step with an equivalent premise, and intervening r\_emit steps are prevented from invalidating it.
• An r\_query step that contributes to another evaluation is ignored.
• An r\_lock step is ignored.
• An r\_deliver step is ignored.

We now show that this generated operation execution sequence for $t_i$ satisfies the execution production requirements.

• The final $\psi$ value must have a clause for every guard in $\text{grd}(\eta)$. Every r\_query step that adds a guard to $\text{grd}(\eta)$ also adds a query step adding the necessary clause to $\psi$.
• The drift steps must correspond to the events preceeding $\eta$. Each event preceeding $\eta$ came from a r\_emit step in the replica rule sequence, which generated the necessary emit step in the operation rule sequence.
• We need $\text{eval}(L_{\eta} = s_i, \text{eff}(\eta) = e_i, \text{and} \text{rval}(\eta) = a_i$ for the final operation rule context. The correspondance of added drift rules to events in $L_{\eta}$ give the first. The $e$ and $a$ in the final operation rule context matches the $e$ and $a$ in the final replica rule context, which are what $\eta$ is created from, giving the second and third.

Thus we generate all the necessary evidence that $L$ is produced by $T$ via the operation rules. □

**Lemma 6.2 (Well-Formed Executions from Replicas).** Any CARD execution $L$ produced by a set of operation invocations $T$ through the replica network execution rules is well-formed.

**Proof.** For $L$ to be produced by $T$ through the replica network rules, it must also be produced by $T$ through the operation execution rules and thus must be well-formed by Theorem 4.5. □

### 7 CONFLICT DETECTION EVALUATION

| Application            | Guards | Effect Classes | Time (ms) | Minimal? |
|------------------------|--------|----------------|-----------|----------|
| Bank account           | 4      | 3              | 35        | Yes      |
| Bank account with reset| 4      | 4              | 33        | Yes      |
| Conspiring booleans (2)| 4      | 3              | 31        | Yes      |
| Joint bank account     | 6      | 8              | 59        | Yes      |
| KV bank accounts (10)  | 11     | 9              | 175       | Yes      |
| State machine (3 states)| 3     | 3              | 46        | Yes      |

Fig. 14. Conflict avoidance set inference

We empirically evaluated whether the core computational task necessary for implementing CARDS — inferring transitive accord sets — is efficient and complete. We implemented the TAS algorithm 2, using the Z3 SMT solver [De Moura and Bjørner 2008] for logical reasoning. We modeled CARD applications of varying complexity, and computed TA sets for their consistency guards. Our applications were simple SMT-representable data structures using integers, booleans, and arrays. Each application’s guards included the empty guard, the total guard (the identity relation), and interesting non-trivial guards required by operations or providing useful information.

2https://github.com/cuplv/dsv
For all tested applications, our solver found TA sets in less than 175ms. Manual examination proved that these conflict avoidance sets are the smallest possible ones. We now detail the applications tested.

**Bank account.** This is the simplest form of our running example, including deposit and withdraw operations which each take a positive amount parameter and produce Add or Sub Counter effects. The guard necessary for withdraw in order to preserve the positive account invariant, \( LE := s_r \leq s_g \), was found to conflict only with Sub, thus matching the intuitive reasoning: “Withdrawals must not be concurrent”.

**Joint bank account.** This example models the joint bank account from Section 2 involving the request/approval sequence. The inference procedure correctly inferred that the TA set for \( LE := s_r \leq s_g \) should include the Sub, Request, and Approve effect classes.

**Bank account with reset.** We extended Counter CARD backing the bank account with a Reset effect which sets the store value to 0. Reset never drops the value below 0 by itself, and thus an operation can safely (with respect to the bank account invariant) emit a Reset without looking at the store. We note two interesting aspects about this example: (a) Intuitively, Resets can execute freely on their own, but Sub requires coordination to halt Resets and Subs. Our technique automatically infers this: the TA set for \( LE \) contains Reset, but the TA set for the trivial guard of a safe reset operation is empty. This is unlike other mixed-consistency systems such Quelea and RedBlue\[Li et al. 2012; Sivaramakrishnan et al. 2015\] where conflicts are symmetric. (b) Due to the arbitration total ordering, the non-commutability of Reset has no impact on SEC.

**Finite state machine.** We modeled a distributed finite state machine with a CARD where \( S \) is the set of states and \( E \) is the set of transition labels. Though the effects are non-commutable, the arbitration maintains SEC without any coordination. Now, suppose that we write an operation that reads the state under the guard \( (s_g = s_c \Leftrightarrow s_r = s_s) \), i.e., if the global state is some critical state \( s_c \), the operation is guaranteed to see it. The TA set for this new guard includes not only “offenders” — those operations leading into and out of \( s_c \) — but also any that determine whether offenders will take that action. In our case, we used an FSM with 3 states and 3 transition effects, and found that the TA set included the effect \( A \) that led into the critical state, and one other effect \( B \) that led to the state from which \( A \) led to the critical state. Note that executing a new operation which is interested in the critical state completely changes the coordination behavior of the CARD application, without any other operations or invariants needing to be rewritten.

**Key-value bank accounts.** This example models an array of ten indexed bank accounts, supporting the same effect classes as the regular bank account but an additional index parameter – the logical reasoning for this example involved using the array SMT theory. We inferred TA sets both for guards that constrained the global-local values of the individual accounts, and for guards that constrained the global-local values of the summation of the accounts. The TA set for the summation \( LE \) guard included the Sub effects for all bank accounts (indices), while the TA sets for the individual account \( LE \) guards included only the Sub effects for that bank account. This illustrates that CARDs allow operations pertaining to different parts of the state to run in parallel even when they are not purely conflict-free.

8 RELATED WORK
We described how our work builds on CRDTs (Shapiro et al.\[Shapiro et al. 2011\] provide a comprehensive overview). Several frameworks allow both conflict-free, and conflicting operations [Balegas et al. 2015; Gotsman et al. 2016; Li et al. 2014, 2012; Sivaramakrishnan et al. 2015; Terry et al. 1995],
offering different trade-offs between consistency and availability. Such mixed-consistency systems are typically built upon key-value databases that offer tunable transaction isolation [Bailis et al. 2013; Lakshman and Malik 2010; Terry et al. 2013].

Our work is closest to the work of [Gotsman et al. 2016], which also focuses on reasoning about data types with such conflicting operations. The approach of [Gotsman et al. 2016] allow the programmer to specify for every pair of operations whether there is a conflict, using a token based system. In contrast, our consistency guards are specified for each operation separately, which allows the developer to reason only about the operation they are currently writing. Note that while our consistency guards (replica state - global state relations) are related to the guarantee relations (replica state - replica state relations) of [Gotsman et al. 2016], the most important difference is how these are used. [Gotsman et al. 2016] use the guarantee relations only in the proof of correctness of a program (as a manual step). The programmer cannot write these guarantees, they can only declare conflicts explicitly between each pair of operations. In contrast, our language lets the programmer specify the guards directly, leading to modular specifications, from which conflicts can be algorithmically inferred.

The second closest work is that of [Balegas et al. 2015], introduces explicit consistency, in which concurrent executions are restricted using an application invariant. Two technically most important differences are: first, our consistency guards are significantly more expressive than invariants. The consistency guards relate the global state to the local state, whereas invariants talk only about one state. That means that in the framework of Balegas et al., one cannot specify a property such as “if getBalance returns a value \(v\), then the account balance is at least \(v\)” (see the bank account with interest in Section 2). Second, our consistency predicates allow finding conflicts by checking conditions on sequential programs. In contrast, application invariants of Balegas et al. require to check conditions on concurrent programs, a significantly harder task.

A related approach [Li et al. 2012; Sivaramakrishnan et al. 2015] allows manual selection of consistency levels for operations. Quelea [Sivaramakrishnan et al. 2015] allows specifying contracts (ordering constraints) on effects. In contrast, our system hides the concept of effect ordering in history, and allows modular conflict specification. CARDs can use such systems as a backend, automatically generating the contracts via the conflict inference technique.

The homeostasis protocol [Roy et al. 2015] addresses conflicts between operations by allowing bounded inconsistencies as long as other forms of correctness are preserved. It may be possible to fruitfully combine consistency guards with relaxed consistency notions. We leave this for future work.

Bayou [Terry et al. 1995] is an early system for detecting and managing conflicts. The conflicts are detected (translated to our terminology) by re-running a check on every replica where an effect is propagated to see if the data has been updated in parallel. This approach to conflict detection is very different from our consistency guard (which are predicates that link a global and local state).

The axiomatic specification which we used to define CARDs is based on the model presented in [Attiya et al. 2016; Burckhardt et al. 2014]. We built on the model to define consistency guard compliance, as well as type checking soundness. The tension between consistency and availability in distributed systems is captured by the CAP theorem [Brewer 2000; Gilbert and Lynch 2012] — we aim to preserve eventual consistency, while maximizing availability.

9 CONCLUSION

We present CARDs, a new extension of CRDTs which allow conflicting operations. The key idea was to develop a language that gives programmers the ability to specify consistency guards that establish what a CARD operation expects from its distributed environments. This enables modular and sequential reasoning about CARD operations.
This paper opens several possible directions for future work. Among these, we plan to pursue extending our language to allow composition of CARDs, as well as transactions with multiple emits. We also plan to work on quantitative relaxations of our invariant requirements. Furthermore, we will investigate systems aspects of our approach: we will empirically investigate different approaches to implementation of our conflict avoidance algorithm.
REFERENCES

Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. 2016. Specification and Complexity of Collaborative Text Editing. In Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing (PODC ’16). ACM, New York, NY, USA, 259–268. https://doi.org/10.1145/2933057.2933090

Peter Baillis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Bolt-on Causal Consistency. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD ’13). ACM, New York, NY, USA, 761–772. https://doi.org/10.1145/2463676.2465279

Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. 2015. Putting Consistency Back into Eventual Consistency. In Proceedings of the Tenth European Conference on Computer Systems (EuroSys ’15). ACM, New York, NY, USA, Article 6, 16 pages. https://doi.org/10.1145/2741948.2741972

E. Brewer. 2000. Towards robust distributed systems (abstract). PODC (2000), 7.

Russell Brown, Sean Cribbs, Christopher Meiklejohn, and Sam Elliott. 2014. Riak DT Map: A Composable, Convergent Replicated Dictionary. In Proceedings of the First Workshop on Principles and Practice of Eventual Consistency (PaPEC ’14). ACM, New York, NY, USA, Article 1, 1 pages. https://doi.org/10.1145/2596631.2596633

Sebastian Burckhardt. 2014. Principles of Eventual Consistency. Vol. 1. now publishers. 1–150 pages. https://www.microsoft.com/en-us/research/publication/principles-of-eventual-consistency/

Sebastian Burckhardt, Manuel Fahndrich, Daan Leijen, and Mooyi Sagiv. 2012. Eventually Consistent Transactions, In Proceedings of the 22n European Symposium on Programming (ESOP). https://www.microsoft.com/en-us/research/publication/eventually-consistent-transactions/

Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated Data Types: Specification, Verification, Optimality. In Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’14). ACM, New York, NY, USA, 271–284. https://doi.org/10.1145/2535838.2535848

John Day-Richter. 2010. What’s different about the new Google Docs: Making collaboration fast. https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html. (2010).

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’08/ETAPS’08). Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

S. Gilbert and N. Lynch. 2012. Perspectives on the CAP Theorem. IEEE Computer 45, 2 (2012), 30–36.

Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. ‘Cause I’m Strong Enough: Reasoning About Consistency Choices in Distributed Systems. In Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’16). ACM, New York, NY, USA, 371–384. https://doi.org/10.1145/2837614.2837625

Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. SIGOPS Oper. Syst. Rev. 44, 2 (April 2010), 35–40. https://doi.org/10.1145/1773912.1773922

Cheng Li, João Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. 2014. Automating the Choice of Consistency Levels in Replicated Systems. In Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC’14). USENIX Association, Berkeley, CA, USA, 281–292. http://dl.acm.org/citation.cfm?id=2643634.2643664

Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making Georeplicated Systems Fast As Possible, Consistent when Necessary. In Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI’12). USENIX Association, Berkeley, CA, USA, 265–278. http://dl.acm.org/citation.cfm?id=2387880.2387906

Ahmed-Nacer Mehdiz, Pascal Urso, Valter Balegas, and Nuno Preguiça. 2014. Merging OT and CRDT Algorithms. In Proceedings of the First Workshop on Principles and Practice of Eventual Consistency (PaPEC ’14). ACM, New York, NY, USA, Article 9, 4 pages. https://doi.org/10.1145/2596631.2596636

Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. 2013. LSEQ: An Adaptive Structure for Sequences in Distributed Collaborative Editing. In Proceedings of the 2013 ACM Symposium on Document Engineering (DocEng ’13). ACM, New York, NY, USA, 37–46. https://doi.org/10.1145/2494266.2494278

Patrick M. Rondon, Ming Kawagucci, and Ranjit Jhala. 2008. Liquid Types. In Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’08). ACM, New York, NY, USA, 159–169. https://doi.org/10.1145/1375581.1375602

Sudipt Roy, Lucia Kot, Gabriel Bender, Bailu Ding, Hossein Hojjat, Christoph Koch, Nate Foster, and Johannes Gehrke. 2015. The Homeostasis Protocol: Avoiding Transaction Coordination Through Program Analysis. In SIGMOD. 1311–1326.

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free Replicated Data Types. In Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems (SSS’11). Springer-Verlag, Berlin, Heidelberg, 386–400. http://dl.acm.org/citation.cfm?id=2050613.2050642


KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative Programming over Eventually Consistent Data Stores. In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’15). ACM, New York, NY, USA, 413–424. https://doi.org/10.1145/2737924.2737981

Pedro Teixeira. 2017. Decentralized Real-Time Collaborative Documents - Conflict-free editing in the browser using js-ipfs and CRDTs. https://ipfs.io/blog/30-js-ipfs-crdts.md. (2017).

Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. 2013. Consistency-based Service Level Agreements for Cloud Storage. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP ’13). ACM, New York, NY, USA, 309–324. https://doi.org/10.1145/2517349.2522731

Douglas B. Terry, Marvin Theimer, Karin Petersen, Alan J. Demers, Mike Spreitzer, and Carl Hauser. 1995. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In SOSP. 172–183.