Sequential Programming for Replicated Data Stores

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

Abstract

We introduce Carol, a language for operations over replicated data stores. The salient feature of Carol is that it allows programming an operation, and even proving its correctness, *modularly* — that is, without explicitly reasoning about other operations which might be performing conflicting updates on the store. This is in stark contrast with existing systems, which require reasoning about all pairs of operations when developing them or proving them correct.

An operation is executed on one replica, and its effect is applied to all. Our key idea is that when developing an operation, a programmer can specify a two-state predicate (called a *consistency guard*) that captures what they expect of the relation between the state of the replica on which their operation executes and the state of the global system. An operation can be proven correct-in-the-distributed-setting using sequential reasoning and only one assumption: the consistency guard holds when the effect is applied at other replicas.

We develop a refinement type system for Carol that enables these proofs and describe an implementation that efficiently maintains the guards as required. We evaluate the implementation empirically and show that it introduces minimal overhead when compared to hand-tuned implementations which require substantially more low-level distributed reasoning from the developer.

1 Introduction

Why are certain data stores called replicated, rather than distributed? It is because in an ideal world, all those replicas are one. More precisely, in an ideal world, the applications that use those stores are programmed in a sequential style, as if for a single host. If an application updates a replica, the system propagates the update to the other replicas. Other parts of an application can use other replicas, but have a store that is (eventually) the same.

Can this ideal be made real? It is already real for a significant class of desirable applications — those that consist of operations which do not conflict with each other, known commonly as CRDTs [1, 5, 9, 16, 17, 20, 22].

But some applications hold inherent conflict. The withdraw operations on a bank account application can always conflict, depending on the availability of funds. If your application matches buyers and sellers, its operations will conflict – you cannot match two sellers to the same buyer. If your application allocates bandwidth in a network to your customers, your operations will conflict – you cannot allocate more bandwidth than you have. Conflicting operations cause the order of operations to matter. Worse, if you need to test a precondition of an update, and then update your local replica, you cannot simply run the same update on the other replicas. A conflicting operation might have executed in the meantime, and rendered your update invalid. Sequential programming and reasoning for operations is thus difficult. Existing systems [3, 12–15, 21] require reasoning about all pairs of operations when writing them or proving them correct.

We propose Carol, a programming lanaguage for (potentially-conflicting) operations over replicated data stores. The Carol language allows writing and verifying store operations (a) modularly, i.e., without reasoning about the other operations that have been defined, and (b) (to a large extent) sequentially, i.e., without considering interleaved activity. The key idea is that the programmer isolates their operation from its concurrent reality with a two-state predicate which we call a *consistency guard*. The consistency guard establishes the common ground on which all replicas stand — the precise set of facts shared by the programmer's context and all others.

When a replica evaluates a Carol operation, it may produce an *effect* that is applied to itself and all others. With a consistency guard in hand, the programmer is free to produce a dangerous effect such as a withdrawal while only checking that it is safe within what they understand as the local context — the guard ensures that this is the remote context as well. To be precise, the predicate in a guard relates the running operation's replica state to all other possible replica states that any effect from the operation might be applied upon. So to safely run a withdrawal, the guard must state that if the local context seems to have enough money for the action, this must be true for all others that could witness it.

Guards are not free, of course. The more stable ground you take as a guard, the more the other replicas slow to accommodate. For this reason, guards are *expressive*. Distributed stores with differing datatypes provide their own

sets of guards that allow the programmer to take only what they need, leaving maximum space for parallel activity.

The Carol language is equipped with a flexible refinement type system, based on LiquidTypes [18], which allows the programmer to formalize these safety checks — in a process nealy identical to standard sequential refinement typechecking — over whatever guards their store of choice provides. Crucially, this formal safety checking mechanism allows the programmer to aggressively shed guards in pursuit of efficiency, without worry that their misunderstanding of subtle concurrent behaviors will introduce a bug.

We implement Carol to show that the higher level of abstraction we provide to the developers does not come at an unacceptable performance cost. There are two key parts of the implementation. First, a runtime system that provides guards needs to act upon the conflicts between those guards and the effects that operations might produce. To infer these explicit conflicts between guards and effects, we introduce *consistency invariants*, a new invariant notion suitable for operations on RDTs. Using consistency invariants, we infer the conflicts using an off-the-shelf SMT solver at compile time. Second, we describe an efficient implementation that runs arbitrary operations through an off-the-shelf distributed object store, and uses the inferred conflicts to minimize synchronization between running operations.

We evaluate this implementation empirically and show that it introduces minimal overhead when compared to hand-tuned implementations which require from the developer substantially more reasoning about the distributed behavior.

Summarizing, the contributions of this paper are:

- Carol, a language for operations on replicated data stores. Its key feature is the consistency guard, enabling modular and sequential programming (Section 3).
- A rich refinement type system for Carol that enables sequential reasoning about pre/post conditions of the operations on the replicated store (Section 4).
- An algorithm for inferring guard-effect conflicts, with a reference implementation that uses the inferred conflicts to employ the least amount of synchronization possible. (Section 5)
- A practical implementation of the Carol runtime system, with an empirical evaluation showing that its novel abstractions incur a relatively small performance penalty (Section 7).

2 Writing and Verifying CARD Operations

We provide an overview of the Carol language on an illustrative example: a bank account in which some operations conflict with each other. We explain how the application is programmed as Carol operations over a general-purpose data store `Counter` (simple integer value that supports addition and subtraction). We then extend the example to show how non-commuting effects can be handled. The completed application will consist of `withdraw` and `deposit` operations. and executing these operations at a replica emits effects which will eventually be processed by other replicas.

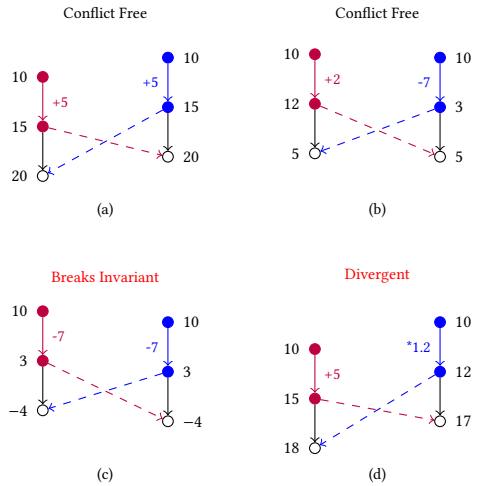

Figure 1. Examples of conflict-free and conflicting operations in executions in a bank account with three operations: deposit (+), withdraw (−), and interest (∗). Each execution consists of two replicas, each executing one operation with no coordination (solid line), and then broadcasting their effects (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).

Problem and desired result Our bank account has three requirements: *strong eventual consistency* (SEC), *availability*, and preservation for *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 — the property that all replicas will converge without coordination — can be achieved (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 1c). 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 consistency is required (and thus what kind of interference is disallowed) for a particular access of replicated store data.

$$
\begin{array}{ll}
S(\texttt{Counter}) := \mathbb{Z} & [\![\texttt{Add}\ n]\!] := \lambda x.\ x + n \\
E(\texttt{Counter}) := \texttt{Add}\ \mathbb{N} \mid \texttt{Sub}\ \mathbb{N} \mid \texttt{Set}\ \mathbb{Z} & [\![\texttt{Sub}\ n]\!] := \lambda x.\ x - n \\
C(\texttt{Counter}) := \top \mid \texttt{LE} \mid \texttt{GE} \mid \texttt{EQV} & [\![\texttt{Set}\ n]\!] := \lambda x.\ n
\end{array}
$$

$$
[\![\top]\!] := \top \quad [\![\texttt{LE}]\!] := s_r \le s_g \quad [\![\texttt{GE}]\!] := s_r \ge s_g \quad [\![\texttt{EQV}]\!] := s_r = s_g
$$

Figure 2. Definition of the `Counter` CARD

The example CARD we are using here is the `Counter`, defined in Figure 2, which uses an integer as its store type, supports simple numerical effects, and provides lower (`LE`) and upper (`GE`) bound guard measures. It thus enriches `Counter` with guards. Having defined this datatype, we can automatically infer the complete set of conflict relationships between the effects and guards up front without needing to know what varying application-specific safety properties they will be used to implement.

Operations We define *operations* over a CARD D using Carol. An operation is a program which atomically runs effects 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 Carol:

$$
\begin{array}{l}
\texttt{withdraw} := \lambda n.\ \textbf{query}\ x : \texttt{LE}\ \textbf{in} \\
\qquad\qquad \textbf{if}\ x \ge n\ \textbf{then}\ (\textbf{issue}\ \texttt{Sub}\ n\ \textbf{in}\ n)\ \textbf{else}\ 0
\end{array}
$$

The term "**query** x : `LE` **in** ..." 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 store value on any replica where the subtraction will be applied (which we abstractly refer to as the "current" store, and define precisely in Section 3). Thus we annotate the term with the `LE` guard to declare that the current store must be less than or equal to the value we bind to x. This safely underapproximates the condition that n should be at most the current store value. The remaining term "**issue** `Sub` n **in** n" adds `Sub` n as an effect to the store and returns n, as an inert value, 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.

Notice that in writing this safe operation, we did not explicitly declare conflicts with other operations or say 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.

Dependent Operation Types Because guards reduce the problem of distributed 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 [18] to cover Carol'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:

$$
\begin{array}{c}
\varphi := (\sigma \ge 0 \Rightarrow [\![\eta]\!](\sigma) \ge 0) \wedge (\rho = \sigma - [\![\eta]\!](\sigma)) \\
\texttt{withdraw} : (n : \texttt{Nat}) \rightarrow \{\texttt{Counter} \mid \rho : \texttt{Int} \mid \varphi\}
\end{array}
$$

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 ρ, σ, and η 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 effect being applied. Our typing rules will reduce this to a Liquid Type which must be checked. The key part is that the type system can reason about the store value guarantee the operation demands via the `LE` query.

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 allow non-commuting store effects, and our reasoning technique and implementation technique are equipped to handle them efficiently. Figure 1d illustrates how non-commutative effects (here, +5 and ×1.2) can lead to replicas diverging, violating strong eventual consistency.

An obvious challenge for non-commutable effects is maintaining SEC. Our approach, following [7], 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 $[\![\texttt{Interest}]\!] := \lambda s.s * 1.2$, and suppose we write a new operation `safeBalance` which returns a value that is definitely

$$
\begin{array}{lll}
e ::= \bullet \mid \overline{e} \mid e_2 \circ e_1 & & \textit{effects} \\
c ::= \top \mid \texttt{EQV} \mid \overline{c} \mid c_1 \wedge c_2 & & \textit{con. guards} \\
t ::= x \mid k \mid \lambda x.t \mid t_1\ t_2 & & \textit{terms} \\
\quad \mid \textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3 & & \\
\quad \mid \textbf{query}\ x : c\ \textbf{in}\ t \mid \textbf{issue}\ e\ \textbf{in}\ t & & \\
v ::= k \mid \lambda x.t & & \textit{values}
\end{array}
$$

Figure 3. Syntax of Carol, in which k stands in for constants. $\overline{e}$ and $\overline{c}$ stand in for store-defined base effects and consistency guards, respectively.

not less than the account's actual value.

$$\texttt{safeBalance} : \{\texttt{Counter}' \mid \rho : \mathbb{Z} \mid [\![\eta]\!](\sigma) = \sigma \wedge \rho \leq \sigma)\}.$$

The order of the Sub, Add and Interest events matter, i.e., the effects do not commute. Most approaches [15, 20] would therefore declare these 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 the state might require coordination.

However, the guard of safeBalance allows us to infer that its requirement only conflicts with withdraw, so the deposit, interest, and safeBalance operations can all 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 any consequences of the effect reordering, without taking the time to understand what they might be. Thus we achieve efficiency, even while ensuring application properties, by depending on the arbitration order rather than coordination to maintain SEC even with non-commutable effects.

3 Language

We present Carol, a programming language for operating on replicated data stores. Section 3.1 gives the precise syntax of Carol programs and Sections 3.2 and 3.3 describe its operational semantics, dividing the definition between replica-local evaluations and the composition of their results in a distributed network.

3.1 Syntax and Intuition

The syntax of Carol terms, or *operations*, is shown in Figure 3. While most constructs are standard, the syntax includes two special terms that interact with a replicated store.

Store Updates with Effects The store can be updated in an operation by using the **issue** e **in** t term, which stages a change to the store and continues with t. The e used in this term is an *effect* — a deterministic update on the store that will be applied at every replica. Formally, an effect e has a denotation $[\![e]\!] \in S \to S$, for a set of store values S, providing a function that will be applied to the store.

When issuing e using the **issue** e **in** t term, e will always be applied before any effects issued by the t subterm. The "no-op" effect $\bullet$ is available in any Carol operation, while non-trivial effects (represented by $\overline{e}$ productions in Figure 3) are specific to the store the operation is written for. As an example, the distributed counter store we have used for our running bank account example supports $[\![\texttt{Add}\ n]\!] := \lambda s.\ s + n$ and $[\![\texttt{Sub}\ n]\!] := \lambda s.\ s - n$ for adding and subtracting from the store value (for store value set $S = \mathbb{Z}$), respectively.

Store Queries with Consistency Guards A Carol operation can make decisions based on the state of the store by using the **query** $x : c$ **in** t term to bind a store value to x in the subterm t. The c used in this term is a *consistency guard* — a measure of accuracy (or completeness) for the information bound to x. A consistency guard c has a denotation $[\![c]\!] \in S \times S \to \texttt{Bool}$ giving a two-state predicate that relates the "local" value bound to x (s_r)to the true "global" value of the store (s_g) for the duration of t's evaluation. Our precise notion of global store value is given in Section 3.3. The trivial consistency guard $[\![\top]\!] := \top$ and total guard $[\![\texttt{EQV}]\!] := s_r = s_g$ are available in any Carol operation, while the "interesting" ones (represented by $\overline{c}$ in the Figure 3 grammar) are specific to the store the operation is written for. All consistency guards must be *reflexive*, meaning that for any guard c, $[\![\texttt{EQV}]\!](s_1, s_2) \Rightarrow [\![c]\!](s_1, s_2)$.

As an example, the distributed counter store we have used for our running bank account example supports $[\![\texttt{LE}]\!] := s_r \leq s_g$ and $[\![\texttt{GE}]\!] := s_r \geq s_g$ for querying ensured lower and upper bounds on the store, respectively.

Consistency guards are the *semantic* equivalent of traditional consistency levels in mixed-consistency systems. Instead of reading the store value "atomically", "with sequential consistency" or "with acquire order" as certain systems allow, a Carol operation reads the store up to the consistency guard c. Operationally, consistency guards can be understood to restrict certain interfering activity in the replica network. From the programming point of view, guards place a semantic property on the value of x.

Example 3.1. The basic withdraw bank account operation is expressed in Carol as follows:

$$
\begin{array}{l}
\texttt{withdraw} := \lambda n.\ \textbf{query}\ x : \texttt{LE}\ \textbf{in} \\
\quad \textbf{if}\ x \geq n\ \textbf{then}\ (\textbf{issue}\ \texttt{Sub}\ n\ \textbf{in}\ n)\ \textbf{else}\ 0
\end{array}
$$

Here, the global store value is queried up to the predicate $\texttt{LE} := s_r \leq s_g$, 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, reusing the basic withdraw, as:

$$\begin{aligned}&\texttt{swithdraw} := \lambda n.\ \textbf{if}\ \texttt{withdraw}\ n = n\\ &\quad\textbf{then}\ n\\ &\quad\textbf{else}\ (\textbf{query}\ x : \texttt{EQ}\ \textbf{in}\\ &\qquad\textbf{if}\ x \geq n\ \textbf{then}\ (\textbf{issue}\ \texttt{Sub}\ n\ \textbf{in}\ n)\ \textbf{else}\ 0)\end{aligned}$$

The first query and the then branch act as the standard withdraw operation, while the second query (with the stronger consistency predicate $\texttt{EQ} \equiv s_g = s_r$) 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

$$\texttt{deposit} := \lambda n.\ \textbf{issue}\ \texttt{Add}\ n\ \textbf{in}\ (-n).$$

CARDs We formalize the interface of guards and effects provided by a particular replicate store as a *conflict-aware replicated datatype*, or CARD, defined as a three-tuple $D = (S, E, C)$ in which S is the underlying set of store values, E is the set of base effects supported by the store, and C, is the set of base consistency guards supported by the store. We call elements of S *D-states*, effects using base effects only from E *D-effects*, guards based on C *D-guards*, and Carol terms containing only D-effects and D-guards *D-operations*.

CARDs can be thought of as traditional RDTs extended with declarative measures of consistency, given by the guards. This makes CARDs more general and reusable by lifting two application-specific aspects of standard RDTs from the datatype core into the Carol language: blocking safety semantics and replica-local computation.

3.2 Replica-Local Evaluation Semantics

In order to modularly formalize the operational semantics of Carol, we first present declarative rules by which Carol terms can be evaluated to *guarded events*, abstract values which encode the result and distributed store interactions produced by the term. We then explain the network executions these guarded events correspond to in Section 3.3.

Carol programs are small in scope; they define a single atomic transaction on the distributed store. A guarded event $d = (g, e, v)$ describes this transaction and the concurrent network context in which it is allowed to succeed. The e value is the composition of events issued by the operation, which will be applied atomically to the store value. The v value is the value returned to the caller of the operation, usually to report some details of the issued effects which the caller cannot directly observe. The g value is the event's guard and takes the form $\{(c_0, v_0), (c_1, v_1), \ldots, (c_n, v_n)\}$, in which a c_i corresponds to a store query made by the operation which reported store value v_i. The collection of queries and results in g form a constraint on the concurrent network activity that the event d tolerates, and restrict the store contexts in which the event can ever be witnessed by a replica.

$$\boxed{t \Downarrow g, e, v}$$

$$\frac{}{v \Downarrow \{\}, \bullet, v}\ \textsc{eval\_val}$$

$$\frac{\begin{array}{c}t_1 \Downarrow g_1, e_1, v_1\\ t_2 \Downarrow g_2, e_2, \lambda x.t_3\\ [v_1/x]t_3 \Downarrow g_3, e_3, v_3\end{array}}{t_2\ t_1 \Downarrow (g_3 \cup g_2 \cup g_1), (e_3 \circ e_2 \circ e_1), v_3}\ \textsc{eval\_lam}$$

$$\frac{\begin{array}{c}t_1 \Downarrow g_1, e_1, \texttt{true}\\ t_2 \Downarrow g_2, e_2, v_2\end{array}}{\textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3 \Downarrow (g_2 \cup g_1), (e_2 \circ e_1), v_2}\ \textsc{eval\_ite\_true}$$

$$\frac{\begin{array}{c}t_1 \Downarrow g_1, e_1, \texttt{false}\\ t_3 \Downarrow g_3, e_3, v_3\end{array}}{\textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3 \Downarrow (g_3 \cup g_1), (e_3 \circ e_1), v_3}\ \textsc{eval\_ite\_false}$$

$$\frac{[v_1/x]t \Downarrow g, e, v_2}{\textbf{query}\ x : c\ \textbf{in}\ t \Downarrow (\{c \triangleright v_1\} \cup g), e, v_2}\ \textsc{eval\_query}$$

$$\frac{t \Downarrow g, e_2, v}{\textbf{issue}\ e_1\ \textbf{in}\ t \Downarrow g, (e_2 \circ e_1), v}\ \textsc{eval\_issue}$$

Figure 4. Replica-local evaluation rules for Carol terms.

Definition 3.2 (Guard-permitted contexts)**.** A store value s is a *g-permitted context*, written as $s \models g$, iff $\forall (c_i, v_i) \in g.\ [\![c_i]\!](s, v_i) = \top$.

Notice that most of the rules in Figure 4 are standard to the CBV λ-calculus. Indeed, a Carol term containing no **query** or **issue** will evaluate to a "trivial" guarded event $(\emptyset, \bullet, v)$, where v is the exact result of CBV λ-calculus evaluation.

3.3 Distributed Execution Semantics

We now fill in the distributed semantics of Carol programs by relating guarded events to *abstract executions*, a standard model [6–8, 12] for describing and reasoning about executions of distributed systems.

Abstract Executions Formally, an abstract execution is a tuple $L = (W, s_0, \text{eff}, \text{rval}, \text{vis}, \text{ar})$ where:

- s_0 is the initial store value.
- W is a finite set of abstract events representing atomic store interactions.
- $\text{eff} : W \to (S \to S)$ gives the update an event makes to the store, for some set S of store values
- $\text{rval} : W \to R$ gives the return value associated with an event, for some set R of return values
- $\text{vis} \subseteq (W \times W)$ is the visibility relation, where $\text{vis}(w_1, w_2)$ indicates that an event w_1 was part of the

context of w_2 (also known as the *happens-before* relation) We denote by $\text{vis}^{-1} : W \to \mathbb{P}(W)$ the set of all events witnessed by an event.
- ar $\subseteq (W \times W)$ is an arbitrary total order on events, respecting vis such that vis $\subseteq$ ar.

Definition 3.3 (Sub-executions)**.** A *sub-execution* $L' = (W', s_0, \text{eff}, \text{rval}, \text{vis}', \text{ar}')$ of $L = (W, s_0, \text{eff}, \text{rval}, \text{vis}, \text{ar})$, written as $L' \subseteq L$, is the execution L restricted to an vis^{-1}-closed subset $W' \subseteq W$ of events. Each of vis′ and ar′ are equal to vis and ar, restricted to the domain W'.

We define two special sub-executions for abstract events. Given an abstract execution $L = (W, s_0, \text{eff}, \text{rval}, \text{vis}, \text{ar})$ and event $w \in W$, we call the sub-execution $L' = (W', \ldots)$ for which $w' \in W'$ iff $\text{ar}(w', w)$ the *pre-execution* of w (written as L_w^{pre}). The *vis-execution* (L_w^{vis}) is similar for $\text{vis}(w', w)$.

Definition 3.4 (Evaluations of abstract executions)**.** The *store evaluation* of an abstract execution L, written as $\text{eval}(L)$ is the store value arrived at by starting with s_0 and applying $\text{eff}(w_i)$ for each $w_i \in W$ in ar order. Formally, if $W = \{w_0, w_1, \ldots w_n\}$ with each $i < j \implies \text{ar}(w_i, w_j)$, then $\text{eval}(L) = (\llbracket \text{eff}(w_n) \rrbracket \circ \llbracket \text{eff}(w_{n-1}) \rrbracket \cdots \llbracket \text{eff}(w_0) \rrbracket)(s_0)$.

Execution Semantics for Guarded Events In terms of the abstract execution model, a guarded event corresponds to a single abstract event and a constraint on the surrounding executions it can be contained in.

Definition 3.5 (Event models)**.** Given an abstract execution $L = (s_0, W, \text{eff}, \text{rval}, \text{vis}, \text{ar})$ and a guarded event $d = (g, e, v)$, an abstract event $w \in W$ is an *event model* of d, written $(L, w) \models d$, iff $\text{eff}(w) = e$, $\text{rval}(w) = v$, and

$$\forall L_r.\ L_w^{\text{vis}} \subseteq L_r \subseteq L_w^{\text{pre}} \Rightarrow \text{eval}(L_r) \models g.$$

The definition of event model makes precise the notion of "global store value" with which we described the purpose of the **query** $x : c$ **in** t term in Section 3.1. Given an ordering on events which arbitrarily but canonically extends the happens-before relationship to a total order, the global store value in relation to a use of **query** is the evalution result of all events which will precede the event produced by the **query**'s operation in the final abstract execution. Obviously, a replica evaluating the **query** cannot know this value precisely; rather, implementations of the Carol semantics must establish a safe bound on this value via coordination before binding a choice of "local view" to x.

3.4 CARD Carriers

We now couple together the local and distributed components of Carol's semantics to define the complete requirements of a distributed store system for Carol operations.

Definition 3.6. Given a CARD D, a *D-carrier* is an abstract system which processes partially ordered sets of D-operations into abstract executions. Given a partial order $(O, <)$ of D-operations and resulting abstract execution $L = (W, \ldots)$, there must exist an intermediate set H of guarded events with one-to-one-correspondences $j : H \to O$ and $h : H \to W$ such that:

1. $\forall d \in H.\ j(d) \Downarrow d$
2. $\forall d \in H.\ (L, h(d)) \models d$
3. $\forall d_1, d_2 \in H.\ j(d_1) < j(d_2) \Rightarrow \text{vis}(h(d_1), h(d_2))$

Intuitionally, a carrier for a CARD is a set of replicas, each holding a queue of operations to evaluate. The replicas must then evaluate the operations according to the replica-local evaluation rules, concretizing the non-deterministic choices such that the produced events fit together into an execution. In this model, two operations t_1 and t_2 are ordered by $\leq$ iff they are in the same replica's queue.

4 Verification

In this section we describe specifications for distributed store events, building up to a refinement type system for Carol, in particular extending LiquidTypes [18] to the replicated data setting. We prove soundness for this type system and describe the correctness properties of Carol systems that the type semantics provide.

4.1 Event Specifications and Invariants

An *event specification* is a predicate with three free variables: σ represents the state of the store immediately before the event is applied, η represents the effect the event will run on the store, and ρ represents the return value that has been given to the caller that produced the event.

Definition 4.1 (Satisfying specs)**.** A guarded event (g, e, v) satisfies a spec φ, written $(g, e, v) \models \varphi$ iff for any g-permitted context s, the substitution $[s/\sigma][e/\eta][v/\rho]\varphi$ is satisfied.

Example 4.2. 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 $\varphi(\sigma, \eta, \rho) := (\sigma \geq 0 \implies \llbracket \eta \rrbracket(\sigma) \geq 0) \wedge (\rho = \sigma - \llbracket \eta \rrbracket(\sigma))$ exactly states these requirements. A guarded event from a `withdraw`, for example $(\{(\text{LE}, 10)\}, \texttt{Sub}\ \ 5, 5)$, satisfies this by placing the $\llbracket \text{LE} \rrbracket$ restriction on $(\sigma, 10)$.

4.2 Refinement Types for Carol

We now present a refinement type system for Carol which incorporates CARDs and event specifications.

The type system for Carol (detailed in Figure 5) extends LiquidTypes [18] on the CBV λ-calculus. For those unfamiliar, liquid types refine standard types with predicates on the values. Types in our system take the form $\{D \mid \rho : A \mid \varphi\}$, in which D is the CARD the operation runs over (consisting of store, effect, and guard base types), A is the base type of the return value component of the resulting guarded event, and

$$\boxed{\Gamma \vdash t : T}$$

$$\frac{\mathbf{ty}\,(k) <: \{v : A \mid \varphi\}}{\Gamma \vdash k : \{D \mid \rho : A \mid \varphi \wedge \eta = \bullet\}}\ \textsc{type\_const} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}\ \textsc{type\_var} \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x.t : (x : T_1) \to T_2}\ \textsc{type\_lambda}$$

$$\frac{\Gamma \vdash t_1 : \mathsf{Bool} \qquad \Gamma, t_1 \vdash t_2 : T \qquad \Gamma, \neg\, t_1 \vdash t_3 : T}{\Gamma \vdash \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 : T}\ \textsc{type\_ite}$$

$$\frac{\Gamma \vdash c : \mathbf{Guard}\,(D) \qquad \Gamma, x : \{D \mid \rho : \mathbf{Store}\,(D) \mid [\![\{c \triangleright \rho\}]\!]\,(\sigma)\} \vdash t : \{D \mid \rho : A \mid \varphi\}}{\Gamma \vdash \mathbf{query}\ x : c\ \mathbf{in}\ t : \{D \mid \rho : A \mid \varphi\}}\ \textsc{type\_query}$$

$$\frac{\Gamma \vdash e : \mathbf{Effect}\,(D) \qquad \Gamma \vdash t : \{D \mid \rho : A \mid [(\eta \circ e)/\eta]\varphi\}}{\Gamma \vdash \mathbf{issue}\ e\ \mathbf{in}\ t : \{D \mid \rho : A \mid \varphi\}}\ \textsc{type\_issue}$$

$$\frac{D = (S, E, C) \qquad e \in E}{\Gamma \vdash e : \mathsf{Effect}(D)} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Effect}(D) \qquad \Gamma \vdash e_2 : \mathsf{Effect}(D)}{\Gamma \vdash e_2 \circ e_1 : \mathsf{Effect}(D)} \qquad \frac{}{\Gamma \vdash \bullet : \mathsf{Effect}(D)} \qquad \frac{D = (S, E, C) \qquad c \in C}{\Gamma \vdash c : \mathsf{Guard}(D)}$$

$$\frac{D = (S, E, C) \qquad \Gamma \vdash c_1 : \mathsf{Guard}(D) \qquad \Gamma \vdash c_2 : \mathsf{Guard}(D)}{\Gamma \vdash c_1 \wedge c_2 : \mathsf{Guard}(D)} \qquad \frac{}{\Gamma \vdash \top : \mathsf{Guard}(D)} \qquad \frac{}{\Gamma \vdash \mathsf{EQV} : \mathsf{Guard}(D)}$$

$$\frac{D = (S, E, C) \qquad s \in S}{\Gamma \vdash s : \mathsf{State}(D)} \qquad \frac{\Gamma \vdash t : T_1 \qquad \Gamma \vdash T_1 <: T_2 \qquad \Gamma \vdash T_2}{\Gamma \vdash t : T_2}\ \textsc{LT-Sub} \qquad \frac{\mathrm{Valid}([\![\Gamma]\!] \wedge [\![t_1]\!] \Rightarrow [\![t_2]\!])}{\Gamma \vdash \{v : B \mid t_1\} <: \{v : B \mid t_2\}}\ \textsc{Dec-<:-Base}$$

Figure 5. Typing and sub-typing rules for Carol.

φ is an event specification for the resulting guarded event. For example, the typing judgement

$$\bullet \vdash \texttt{withdraw}\ 5 : \{\texttt{Counter} \mid \rho : \texttt{Int} \mid (\sigma \geq 0 \Rightarrow [\![\eta]\!](\sigma) \geq 0) \wedge (\rho = [\![\eta]\!](\sigma) - \sigma)\}$$

states that evaluating `withdraw 5` will not bring the store value below 0 and the result value will represent precisely the amount withdrawn from the store.

Intuitively, the TYPE_Q rule is similar to a conditional guard rule: if a term t is of type $\{D \mid \rho : A \mid \varphi\}$ given the additional premise $[\![c]\!]$, the term **query** $x : c$ **in** t is also of type $\{D \mid \rho : A \mid \varphi\}$. The TYPE_R rule derives our `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 Γ) ensure the `Op` type's φ specification holds. The refinement part of this Liquid Type judgment becomes a simple logical constraint problem according to the rules in Figure 5. 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 4.3. As an end-to-end demonstration, we now type-check the `withdraw` operation according to the specification we have been using, for which

$$\varphi := (\sigma \geq 0 \Rightarrow [\![\eta]\!](\sigma) \geq 0) \wedge (\rho = [\![\eta]\!](\sigma) - \sigma)$$

We first follow the derivation in Figure 6, storing in the context the constraint on σ (the pre-store value) that the query on `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 Γ^+, in Figure 7. This takes us to the standard Liquid Type obligation

$$\Gamma^+ \vdash n : \{v : \texttt{Int} \mid [\texttt{Sub}\ n/\eta]\varphi\}$$

which may look strange since n already has the type `Nat` in Γ^+. This is where, in Figure 8, we use the Liquid Type sub-typing 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 σ constraint from our guarded query:

$$[\![\Gamma^+]\!] \wedge [\![\texttt{Nat}]\!] \Rightarrow [\![\{v : \texttt{Int} \mid [\texttt{Sub}\ n/\eta]\varphi\}]\!] =$$

$$(n \geq 0) \wedge (x \leq \sigma) \wedge (x \geq n) \wedge \eta = \texttt{Sub}\ n \wedge \rho = n \Rightarrow (\sigma \geq$$

$$0 \Rightarrow [\![\eta]\!](\sigma) \geq 0) \wedge (\rho = \sigma - [\![\eta]\!](\sigma))$$

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

Definition 4.4 (Semantics for operation types)**.** Given a CARD $D = (S, E, C)$, an operation t has type $\{D \mid \rho : A \mid \varphi\}$ (written as $t \in \{D \mid \rho : A \mid \varphi\}$) iff all guarded events that t can evaluate to are well typed and satisfy φ, i.e., $t \Downarrow (g, e, v) \implies \forall (c_i, v_i) \in g.\ c_i \in C \wedge v_i \in S \wedge e : E \wedge v : A \wedge (g, e, v) \models \varphi$

$$\dfrac{\dfrac{\dfrac{}{n : \mathsf{Nat}, x : \{D \mid \rho : \mathbf{Store}(\mathsf{Counter}) \mid x \leq \sigma\} \vdash (x \geq n) : \mathsf{Bool}}}{\begin{array}{c} n : \mathsf{Nat}, x : \{D \mid \rho : \mathbf{Store}(\mathsf{Counter}) \mid x \leq \sigma\}, x \geq n \vdash \{\ldots (\mathit{then})\} : \{\mathsf{Counter} \mid \rho : \mathsf{Int} \mid \varphi\} \\ n : \mathsf{Nat}, x : \{D \mid \rho : \mathbf{Store}(\mathsf{Counter}) \mid x \leq \sigma\}, \neg(x \geq n) \vdash \{\ldots (\mathit{else})\} : \{\mathsf{Counter} \mid \rho : \mathsf{Int} \mid \varphi\} \end{array}}{n : \mathsf{Nat}, x : \{\rho : \mathbf{Store}(\mathsf{Counter}) \mid x \leq \sigma\} \vdash \{\mathtt{if}\ldots\} : \{\mathsf{Counter} \mid \rho : \mathsf{Int} \mid \varphi\}}\ \textsc{type\_ite}}{\dfrac{\dfrac{}{n : \mathsf{Nat} \vdash \mathsf{LE} : \mathbf{Guard}(\mathsf{Counter})}}{n : \mathsf{Nat} \vdash \mathbf{query}\ x : \mathsf{LE}\ \mathbf{in}\ \{\mathtt{if}\ldots\} : \{\mathsf{Counter} \mid \rho : \mathsf{Int} \mid \varphi\}}\ \textsc{type\_query}}{\bullet \vdash \lambda n.\ \mathbf{query}\ x : \mathsf{LE}\ \mathbf{in}\ \{\mathtt{if}\ldots\} : (n : \mathsf{Nat}) \rightarrow \{\mathsf{Counter} \mid \rho : \mathsf{Int} \mid \varphi\}}\ \textsc{type\_lambda}$$

Figure 6. Derivation of `withdraw` type down to branches with base (non-**query**) terms, for Example 4.3.

$$\dfrac{\dfrac{\dfrac{}{\Gamma^+ \vdash n : \mathsf{Nat}}\ \textsc{type\_var}}{\Gamma^+ \vdash \mathsf{Sub}\ n : \mathbf{Effect}(\mathsf{Counter})} \quad \Gamma^+ \vdash n : \{\mathsf{Counter} \mid \rho : \mathsf{Int} \mid [\mathsf{Sub}\ n/\eta]\varphi\}}{\Gamma^+ \vdash \mathbf{issue}\ \mathsf{Sub}\ n\ \mathbf{in}\ n : \{\mathsf{Counter} \mid \rho : \mathsf{Int} \mid \varphi\}}\ \textsc{type\_issue}$$

Figure 7. Derivation of **issue** term in `withdraw`'s success branch down to standard Liquid Type, for Example 4.3.

$$\dfrac{\dfrac{\dfrac{\mathrm{Valid}([\![\Gamma]\!] \wedge [\![\mathsf{Nat}]\!] \Rightarrow [\![\{\nu : \mathsf{Int} \mid [\mathsf{Sub}\ n/\eta]\varphi\}]\!])}{\Gamma^+ \vdash \mathsf{Nat} <: \{\nu : \mathsf{Int} \mid [\mathsf{Sub}\ n/\eta]\varphi\}}}{\Gamma^+ \vdash n : \{\nu : \mathsf{Int} \mid [\mathsf{Sub}\ n/\eta]\varphi\}}}{\Gamma^+ \vdash n : \{\mathsf{Counter} \mid \rho : \mathsf{Int} \mid [\mathsf{Sub}\ n/\eta]\varphi\}}$$

Figure 8. Derivation for one of `withdraw`'s Liquid Type obligations into logical constraint problem, for Example 4.3. In bottom-up order, first TYPE_CONST, then DEC-<:-BASE, and then LT-SUB is used.

Theorem 4.5 (Soundness of typing rules)**.** *The type system presented in Figure 5 is sound, i.e.,* $\vdash t : \{D \mid \rho : A \mid \varphi\} \implies t \in \{D \mid \rho : A \mid \varphi\}$.

Proof. By case analysis on the rules for deriving types. Here are the interesting cases:

Case: TYPE_ISSUE Our premise is that $\forall(g_1, e_1, v_1).\ t \Downarrow g_1, e_1, v_1 \Rightarrow (g_1, e_1, v_1 \models [(\eta \circ e)/\eta]\varphi)$ for some t and e. We must show that

$$\forall(g_2, e_2, v_2).\ \text{issue } e \text{ in } t \Downarrow g_2, e_2, v_2.\ (g_2, e_2, v_2 \models \varphi).$$

Looking at our evaluation rules, the only one that matches our term is EVAL_ISSUE, which means that for any g_2, e_2, v_2 we choose, $g_2 = g_1$, $e_2 = e_1 \circ e$, and $v_2 = v_1$. Clearly, by substitution,

$$(g_2, e_1, v_2 \models [(\eta \circ e)/\eta]\varphi) \Rightarrow (g_2, e_1 \circ e, v_2 \models \varphi),$$

which is our goal.

Case: TYPE_QUERY Our premise is that $\forall(g_1, e_1, v_1).\ [v_x/x]t \Downarrow g_1, e_1, v_1 \wedge (\sigma, v_x \models c) \Rightarrow (g_1, e_1, v_1 \models \varphi)$ for some t and c. We must show that $\forall(g_2, e_2, v_2).\ \text{query } c \text{ as } x \text{ in } t \Downarrow g_2, e_2, v_2.\ (g_2, e_2, v_2 \models \varphi)$. Only EVAL_QUERY matches our term, so we can be sure that for any g_2, e_2, v_2 we choose, $g_2 = \{c \triangleright v_x\} \cup g_1$, $e_2 = e_1$, and $v_2 = v_1$. Any qualifying history (L', L) for (g_2, e_2, v_2) must respect that $(\mathrm{eval}(L'), v_x) \models c$, and thus we get our goal by substitution $(\sigma, v_x \models c) \Rightarrow ((g_1, e_1, v_1) \models \varphi$
$(\{c \triangleright v_x\} \cup g_1, e_2, v_2) \models \varphi$ □

Soundness for CAROL types means that we can prove an operation will always produce events that satsify an event specification. Thus, a D-carrier run only over operations typed to respect some D-invariant, such as $\sigma \geq 0$ for our bank account, will always preserve that invariant.

Deferred Conflict Reasoning CAROL allows us to write and verify programs over a CARD D without referencing the conflict relationships between effects and guards, isolating the programmer from such concurrency details. A D-carrier must identify these relationships in order to operationally ensure the semantics of guards, but the qualified interface provided by D makes this task finite and reusable between applications. We define and provide an algorithm for this identification process in the next section.

5 Inferring Conflict Avoidance Requirements

The semantics defined for CARD carriers in Section 3 require that such a system limit concurrent activity in accordance with guards as operations are evaluated. This requirement is simple to define and use for modular verification reasoning, but its implementation depends on a more complete picture of effect-guard relationships — in particular, the impact of particular effects on guard guarantees. We define several useful relationships between constistency refinements and effects in Section 5.1 and then give an algorithm for computing these relationships over a given CARD in Section 5.2.

5.1 Accord between guards and effects

Definition 5.1 ((D, c)-compliance)**.** For a CARD D and D-guard c, two D-states s_1 and s_2 are (D, c)-compliant iff $\forall e : \mathrm{Effect}(D)\ (e(s_1), e(s_2)) \models c$.

Note that $s_1 = s_2$ trivially ensures (s_1, s_2) is (D, c)-compliant.

Definition 5.2 (Accord)**.** A D-guard c and D-effect e are in *accord* iff for all s_1, s_2 in D, if s_1, s_2 are (D, c)-compliant, then so are $e(s_1)$ and s_2.

Example 5.3. In our bank account, a pair of states (s_g, s_r) are (`Counter`, `Le`)-compliant when $s_r \leq s_g$, since they satisfy `Le` and no `Counter` effect applied to both can invalidate that. Then, `Le` and `deposit` n are in accord, since a deposit would only increase s_g. However, `Le` and `withdraw` n are not in accord, as a large withdraw may result in $s_g < s_r$.

Lemma 5.4 (Safe effect insertion)**.** *Given a pair of D-states (s_1, s_2), D-effects e and e', and D-guard c, for which e and c are in accord and (s_1, s_2) are (D, c)-compliant, then $([\![e' \circ e]\!](s_1), [\![e']\!](s_2))$ is also (D, c)-compliant.*

Proof. Apply definition of accord to show that $([\![e]\!](s_1), s_2)$ is (D, c)-compliant, and then use definition of compliance to show that $([\![e' \circ e]\!](s_1), [\![e']\!](s_2))$ is also (D, c)-compliant.

Theorem 5.5 (Arbitrary safe insertion into effect sequences)**.** *Given a starting D-state s_0 and sequence of D-effects e^*, we can expand this sequence to f^* by inserting an arbitrary number of D-effects which are all in accord and with a D-guard c and be sure that $([\![f^*]\!](s_0), [\![e^*]\!](s_0))$ is (D, c)-compliant.*

Proof. Insert new in-accord effects into $e*$ end-first, stepping them to their $f*$ location by repeated applications of Lemma 5.4, depending on the fact that the starting state (s_0, s_0) is (D, c)-compliant by $s_0 = s_0$.

Theorem 5.5 gives us the main result of this section: a D-carrier system that requires accords for effects in events concurrent to a current operation evaluation with respect to its queries' guards will fulfill its consistency obligations. We provide a detailed carrier model that uses accords in this way in Section 6. The following theorem states that finding the largest accord set is undecidable.

Theorem 5.6. *Given a CARD D and D-guard c, finding the largest cardinality accord set for c is undecidable.*

Proof. Sketch: checking compliance for pairs of states involves enumerating all effects (including compositions of the base effects) in a CARD. Such unbounded reachability problems are undecidable.

5.2 Inferring Minimal Locking Conditions

Inferring minimal locking conditions is similar in spirit to verifying specifications in sequential settings. We define two concepts, *1-accords* and *consistency-invariants*. 1-accords are equivalent to showing a required property holds over a single line of a program, while consistency invariants are like standard inductive loop invariants — they are a strengthening of the property that is preserved by operations.

Definition 5.7 (1-accord)**.** A guard c and an effect e are in *1-accord* iff for all s_1, s_2 in D, we have that

$$(s_1, s_2) \models c \implies (e(s_1), s_2) \models c.$$

Definition 5.8 (Consistency Invariant)**.** Given a CARD $D = (S, E, C)$, a D-guard c is a *consistency invariant* in D iff $\forall e \in E.\ \forall s_1, s_2 \in D.\ (s_1, s_2) \models c \implies (e(s_1), e(s_2)) \models c$.

Note that if c is a consistency invariant, then all D-effects e which are in 1-accord with c are also in accord with c. The *1-accord set* $\mathsf{AS1}(c)$ for a guard c is a set of effects which are in 1-accord with c. The *accord set* $\mathsf{AS}(c)$ for a guard c is a set of effects which are in accord with c. We show that every consistency invariant that implies a given c approximates the accord set for c.

Lemma 5.9. *Let D be a CARD and c, c' be D-guards. If c' is a consistency invariant and $[\![c']\!] \Rightarrow [\![c]\!]$, then $AS1(c') \subseteq AS(c)$.*

Proof. Immediate from defintion of consistency invariant and 1-accord.

The total guard `EQV` (the identity relation) itself is always a consistency invariant, similar to how $\bot$ is always a loop invariant in the sequential setting. However, this invariant leads to an 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 accord set.

In spite of Theorem 5.6, we present a simple semi-procedure that computes an accord set in practice through consistency invariants. First, let the *weakest consistency precondition* of a guard c and effect e, $\mathtt{WCP}(e, c)$, be the weakest guard such that $(s_g, s_r) \models \mathtt{WCP}(e, c)$ implies that $([\![e]\!](s_g), [\![e]\!](s_r)) \models [\![c]\!]$. Now, we compute accords for a D-guard c, where $D = (S, E, C)$, with:

$$\mathrm{PAS}_D(c) := \textbf{let } c' = \bigwedge_{e \in E} \mathtt{WCP}(e, c) \textbf{ in}$$
$$\textbf{if } c \Rightarrow c' \textbf{ then } \mathrm{AS1}_D(c) \textbf{ else } \mathrm{PAS}_D(c \wedge c')$$

Note that we are checking `WCP` against D's base effects (E), not the by-definition-infinite space of D-effects. We discuss how the size of base effect set to check can be kept small at the end of this section.

The following theorem states the soundness of the above procedure.

Theorem 5.10. *Given a CARD D, and D-guard c, the procedure $PAS_D(c) \subseteq AS(c)$.*

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 $\forall e : E.\ c_i \Rightarrow \mathtt{WCP}(e, c_i)$.
- Therefore, because $c_i \Rightarrow c$ and c_i is a consistency invariant, then we conclude by Lemma 5.9.

The procedure PAS is computing the greatest fixed-point c'_L of the equation $\mu c' : c' \implies c \wedge ((s_g, s_r) \models c') \implies \bigwedge_{e:E}([\![e]\!](s_g), [\![e]\!](s_r)) \models c'$ as a consistency invariant and using it to decide 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).

Having found such an accord set (or another fixpoint) for a guard $c \in C$, any query guarded by c can safely proceed so long as it makes sure that any effect $e \in E$ not in $\text{PAS}_D(c)$ will not be emitted while the guard is active. In Section 6, we design such a system, where only effects not in accord with an active guard are blocked.

Partitioning Effect Types The PAS algorithm requires a finite set of effects, and becomes more efficient as effect sets get smaller. In order aid accord reasoning, we would like to partition an effect set E, which may be infinite or just very large, into a finite set of equivalence classes $\overline{E}$, which we call *accord classes*, such that two effects in the same class will be related by accords to the same set of guards. In general, we will create partitions that as closely overapproximate true accord equivalence as possible, such that each class holds the greatest common accord set of its effects.

Example 5.11. The obvious partition of an effect type into good accord classes is by constructor. For our `Counter` example, $\overline{E} := \{\texttt{Add}^\forall, \texttt{Sub}^\forall\}$ where $\texttt{Add}^\forall$ and $\texttt{Sub}^\forall$ include events of the form `Add` n and `Sub` n, respectively. If 0 is considered a valid parameter for `Counter` effects, then a more complete partitioning would put `Add` 0 and `Sub` 0 together in a third "no-op" partition for which all possible accords exist.

6 Replica Networks

In this section, we use inferred locking conditions for a CARD D to implement a network of replicas that safely process concurrent Carol operations, meeting the requirements of a D-carrier. In Figure 9, we give the small-step semantics by which such a network executes operations using the accord sets computed using the procedure detailed in Section 5.

6.1 Operational Network Semantics

Definition 6.1 (Network Configurations and Executions)**.** A *network configuration* is a tuple $(L \mid C \mid R)$, in which L is an abstract execution describing the history of events in the network, C is a coordination configuration describing lock aquisitions agreed upon by all replicas, and R is a set of replicas. R is a set of (L_i, w_i, e, O) replicas, in which w_i is a unique pending event, $L_i \subseteq L$ is the seen sub-execution of history, e_i is a staged store effect, and O is the replica's queue of operations to process. The coordination configuration C is a set of mappings $w : c$ from a pending event w to a consistency guard c.

We define an *initial network configuration* as one with empty locks and history, a *final network configuration* as one with empty operation sequences and completely delivered histories on all replicas, and a *terminating network execution* as an initial configuration paired with a sequence of replica rule steps that take it to a (unique) final configuration.

Abstract Execution Updates The explicit replica execution rules are shown in Figure 9. We update abstract executions with the following shorthand:
$L ::_{L_i} (w, e, v)$ denotes an extension of L, where $L_i \subseteq L$, that adds a new event w with effect e and rval v to L such that w's vis-execution is L_i.
$L :: (w, e, v)$ is short for $L ::_L (w, e, v)$, "appending" w to L.
$L_i \cup_L w$ adds w and its dependency closure from L to L_i, where $L_i \subseteq L$, such that $L^{\text{vis}}_w \subseteq L'_i$.

Replica-network Rules. The replica rules are an extension of standard term evaluation, providing:

Querying. The Query rule describes the conditions for acquiring a guarantee. A replica may acquire a given consistency guard if all emitted effects not yet visible to the replica are in accord with the guard being processed.

Effect staging. A replica can stage changes to the store via the Issue rule. When it finishes evaluating the operation, it can emit the combined changes and return.

Operation completion. The Val rule simulates the completed evaluation of an operation by consuming the return value and staged changes and adding an event representing them to the global abstract execution. Importantly, it can only do this if the planned effect is in accord with all guarantees the emitting-replica has given. If this is not the case, the operation can be trivally restarted because no effects have been emitted so far (ergo no side-effects).

Effect delivery. The effect delivery rule copies an event from the network history into the local history of a replica.

Definition 6.2 (Execution products)**.** The *execution product* of an an operation t in an initial replica configuration for a terminating network execution that defines a final history L is the event w which was was added to L in the unique R_VAL step completing t's evaluation.

Execution products allow us to link systems implementing the replica rules to D-carriers.

Lemma 6.3 (Execution products are event models)**.** *For an initial network configuration with term t and terminating network execution producing t's execution product w in L, there exists a guarded event d such that $t \Downarrow d$, and also $(L, w) \models d$.*

Proof. The proof is given in three parts. First, we derive a unique d result for each t in a given network execution. Second, we show that for these derived guarded events, $t \Downarrow d$. Third, we show that for a t with execution product w in an execution ending with L, $(L, w) \models d$.

$$\frac{L = (\text{eff}, \ldots) \qquad \forall w \in L.\ \text{eff}(w) \in \text{AS}_D(c) \vee w \in L_i \qquad v = \text{eval}(L_i)}{L \mid C, w_i : c_1 \mid R, (L_i, w_i, e, \textbf{query}\ x : c_2\ \textbf{in}\ t\ ::\ O) \longmapsto L \mid C, w_i : c_1 \wedge c_2 \mid R, (L_i, w_i, e, [v/x]t\ ::\ O)}\ \textsc{query}$$

$$\frac{}{L \mid C \mid R, (L_i, w_i, e_1, \textbf{issue}\ e_2\ \textbf{in}\ t\ ::\ O) \longmapsto L \mid C \mid R, (L_i, w_i, e_2 \circ e_1, t\ ::\ O)}\ \textsc{issue}$$

$$\frac{\forall (w_r : c_r) \in C.\ e \in \text{AS}_D(c_r) \vee w_r \in (L_i \cup \{w_i\}) \qquad w_i' \notin (L \cup \{w_i\})}{L \mid C \mid R, (L_i, w_i, e, v\ ::\ O) \longmapsto L ::_{L_i} (w_i, e, v) \mid C, (w_i' : \bullet) \mid R, (L_i :: (w_i, e, v), w_i', \bullet, O)}\ \textsc{val}$$

$$\frac{w \in L}{L \mid C \mid R, (L_i, w_i, e, O) \longmapsto L \mid C \mid R, (L_i \cup_L w, w_i, e, O)}\ \textsc{deliver}$$

Figure 9. Core replica execution rules. Rules which only update a subterm and pass through context modifications to the parent term are omitted. When we use abstract executions as sets, as in $w' \in (L \cup \{w\})$, we mean the contained event set W.

Identifying Guarded Events To identify guarded events for operations, we define an algorithm that runs over a terminating network execution with rules of the form

$$j, t, g, e \overset{\text{REP\_RULE}}{\longmapsto} j', t', g', e'$$

where j is one-to-one map between guarded events and operations and t is the full form of the currently evaluating operation. Free variables on the right side of each rule refer to their values in the replica rule step that has been matched.

$$j, t_i, g_i, e_- \overset{\text{QUERY}}{\longmapsto} j, t_i, g_i \cup \{c_2 \triangleright v\}, e_i$$
$$j, t_i, g_i, e_- \overset{\text{ISSUE}}{\longmapsto} j, t_i, g_i, e_2 \circ e_i$$
$$j, t_i, g_i, e_i \overset{\text{VAL}}{\longmapsto} j \cup \{(g_i, e_i, v) \to t_i\}, t_{i+1}, \emptyset, \bullet$$
$$j, t_i, g_i, e_i \overset{\text{DELIVER}}{\longmapsto} j, t_i, g_i, e_i$$

These rules consider the operations of a single replica; the full j is derived by running this algorithm for each replica and combining the results.

Showing Guarded Event Correctness To show that the above algorithm follows the Carol evaluation rules in constructing guarded events, we abstract each non-val rule into a simpler rule of the form

$$g, e, t \longmapsto g', e', t'$$

(in which t and t' are the pre- and post-term in the matched replica rule) such that the rules serve as a small-step semantics for operations. We then show by case analysis that the small-step rules are equivalent to the rules for $t \Downarrow d$, proving that

$$\frac{g_1, e_1, t_1 \longmapsto g_2 \cup g_1, e_2 \circ e_1, t_2 \qquad t_2 \Downarrow g_3, e_3, v}{t_1 \Downarrow g_3 \cup g_2, e_3 \circ e_2, v}$$

The cases for the core rules follow.

Query Given the term **query** $x : c$ **in** t, the small-step rule adds $\{c \triangleright v\}$ (where v is the evaluation of the replica-local history) to g. The big-step rule makes the same addition but with *any* v; thus the small-step rule's choice makes one of multiple valid steps.

Issue Given the term **issue** e **in** t, both big and small-step rules add e to the effect such that it is run before any effect issued by t.

Showing Event Models We first note that, trivially, the e and v values of the guarded event (g, e, v) constructed for a term t by our algorithm are the same as those assigned to t's event product w in the network execution. The remaining requirement to show for $(L, w) \models (g, e, v)$ is that

$$\forall L_r.\ L_w^{\text{vis}} \subseteq L_r \subseteq L_w^{\text{pre}} \Rightarrow \text{eval}(L_r) \models g.$$

Using Theorem 5.5, we can prove this by showing that all $w' \in L_w^{\text{pre}}$ are either visible to w or have effects in accord with each guard in g.

The premises of the query rule require this to be the case for L when a guard c is added to g. This rule also adds c to C for the event w, putting an important restriction on all val steps that follow. Each following val step that adds another w' with effect e' to L requires that either $e' \in \text{AS}_D(c)$, or that $w \in L_{w'}^{\text{vis}}$ (meaning that $w' \notin L_w^{\text{pre}}$). Thus g is guaranteed to be protected in the final L.

This completes our proof that for any t with execution product w in a terminating network execution ending with L, there exists a d for which $t \Downarrow d$ and $(L, w) \models d$. □

Theorem 6.4 (Replica Rules Implement a CARD Carrier)**.** *Given a set $(O, <)$ of D-operations and an initial network configuration in which replicas hold disjoint subsets of O and any o_1, o_2 on a single replica are related by $<$, the L of any reachable final configuration is a valid D-carrier output for input $(O, <)$.*

Proof. The definition of a valid carrier output requires the existance of two one-to-one correspondences, j and h, with some conditions. If we choose the guarded event construction algorithm defined for Lemma 6.3 for j and compose j^{-1} with the definition of execution product for h, we can show that these conditions are met:

1. $\forall d \in H.\ j(d) \Downarrow d$, given by Lemma 6.3.
2. $\forall d \in H.\ (L, h(d)) \models d$, given by Lemma 6.3.
3. $\forall d_1, d_2 \in H.\ j(d_1) < j(d_2) \implies \text{vis}(h(d_1), h(d_2))$, given by the fact that replicas always add an emitted effect w_i to their local L_i history before adding the next w_{i+1} to any history, guaranteeing that $\text{vis}(w_i, w_{i+1})$.

The replica rules assume a correct accord set for a CARD D, and thus they correctly implement a D-carrier. □

6.2 Arbitration Order

Throughout the paper, we use a global arbitration order ar. In our setting, the arbitration order is achieved *without distributed coordination*; it is locally computed at each replica as events are received. Our system already maintains a causal order on events (which itself does not need any blocking coordination). Events incomparable in the causal order can be simply compared by their replica ID to achieve deterministic ordering, the same at each replica.

This mechanism works for our system because we allow events newly received by a replica to be inserted into the replica's history before the end, if arbitrated so. This requires the tail of its history to be recomputed. Local recomputation is preferable over network coordination in the vast majority of systems, and the local recomputation can be (heuristically) made more efficient by summarization of older history.

6.3 Coordination/Locking Protocols

The replica rules we present here are declarative; they specify *when* a replica is allowed to proceed with querying and issuing but do not give instructions for actively getting to that state. In particular, the QUERY rule requires that all not-in-accord events have been seen by the querying replica at the time of evaluation, and both QUERY and VAL modify/read a global coordination configuration. We now briefly describe how this global coordination can be performed, see Section 7 for more details and an evaluation.

QUERY First consider a replica with id i seeking to update the coordination configuration from $i : c_1$ to $i : c_1 \wedge c$ (i.e. acquire the guard c) for the QUERY rule. In our implementation, the replica requests all other replicas to guarantee that they won't invalidate c, and each replica $j \neq i$ responds with an acknowledgement and an update of its abstract execution L_j. After receiving all of these responses, the requesting replica knows the other replicas will only emit events that are in accord with c, and by merging its abstract execution with the ones it has recieved (a series of DELIVERS), it has met the requirements of QUERY and can proceed with the evaluation.

VAL Now consider a replica seeking return and emit the combined staged effects e via the VAL rule. If e is in accord with all guards, it would be safe to simply emit it, but it could be the case that e conflicts with some guard c that the replica has guaranteed to respect. In order to keep track of such guarantees, the replica keeps a local record of guards it has promised not to violate, and simply cycles through the record to make sure it can emit. Depending on whether e is in accord with every such guard, the replica either emits the corresponding event or restarts the operation. Either way, the replica releases all the guards it has acquired so that the other replicas can emit in the meantime.

7 Evaluation

We performed our evaluation to answer two key research questions concerning the performance of CAROL implementations: 1. Is the inference of accord sets for via the PAS algorithm efficient for a variety of CARDs? 2. Is the runtime performance of a CAROL carrier implementation scalable to real geo-distributed applications?

7.1 Static Conflict Identification

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

Figure 10. Conflict avoidance set inference

We empirically evaluated whether the core computational task necessary for implementing CARDs — inferring accord sets — is efficient and complete. We implemented the PAS algorithm using the Z3 SMT solver [10] for logical reasoning. We modeled CARD applications of varying complexity, and computed AS sets for their consistency guards. Our applications were 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. For all tested applications, our solver found AS sets in less than 175ms. Manual examination proved that these conflict avoidance sets are the smallest possible ones. We detail two cases.

Bank account with reset We extended the 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. Our technique automatically inferred this: the AS set for LE contains `Reset`, but the AS set for the trivial guard of a safe `reset` operation is empty.

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

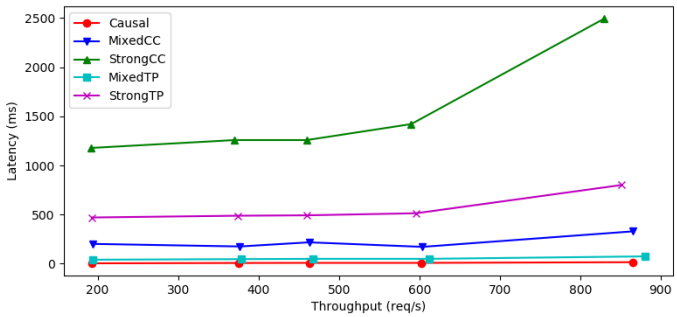

Figure 11. Runtime evaluation results, comparing contention-management techniques for differing workloads.

reads the state under the guard ($s_g = s_c \Leftrightarrow s_r = s_c$), i.e., if the global state is some critical state s_c, the operation is guaranteed to see it. The AS set for this new guard includes not only "offenders" — those operations leading into and out of s_c — but also any that enable the offenders.

7.2 Runtime Performance

We implemented the semantics of Carol as a Haskell library which realizes the language as a monadic eDSL and provides a replica system that correctly evaluates the embedded Carol operations according to the model given in Section 6.

Coordination, based on the accords inferred in the previous subsection, is managed using a state-based CvRDT [20] (a state with a monotonic merge function) similar to the abstract coordination configuration used in the Figure 9 replica rules. This shared CvRDT maps replicas to their guard holdings, listing the other replicas that have granted each request. A replica can proceed to evaluate the body of its query when all others have granted its current holdings.

Managing Lock/Emit Contention The last step of evaluating an operation is to emit any generated effect into the store. If the evaluating replica has granted guard holdings that block this effect, it must defer the emission. In fact, the operation must be backed-out of any non-trivial queries it has made so that the evaluating can release the guards to avoid deadlock. At high load, some order must be imposed on the requesting of locks so that there are not effects which retry continually, never making it into the store. We have implemented two forms of this contention management, and compared their performance in our experiment.

Congestion Control (CC) Similar to TCP congestion control, replicas track their rate of failure for effects blocked by a particular guard, and accordingly adjust the rate at which they retry those operations and grant requests.

Token Passing (TP) Instead of requesting and granting a guard ad-hoc, replicas pass it in a circle such that only one replica holds it at a time. This reduces wasted time from retries and timeouts, but disallows some safe concurrency; read-only operations which use the guard but emit nothing cannot be run simultaneously on multiple replicas.

Store History as Distributed DAG The CARD state is stored as a second CvRDT — a Merkle-DAG of events stored and distributed by an off-the-shelf distributed object store (IFPS [22]), headed by a Lamport clock allowing efficient merges. This model of history is required to maintain the arbitration total we depend on for convergence in applications with non-commutable effects. In our evaluation, we looked to confirm that this method scales to a non-trivial workload.

Experiment We ran replicas on two machines, geo-separated such that their rtt was on average 176ms. We chose operation workloads requiring mixed (some using a non-trivial query and some not), causal-only, and strong-only consistency (each with 15% issuing updates and 85% query-only). The replicas were continuously given operation requests from these workloads at rates from $200/s$ to $1000/s$, and we measured the average latency with which the operations were completed.

Our results in Figure 11 show that Carol can be implemented without unreasonable latency cost. Both contention-management techniques were functional, though the token-passing technique was the clear winner in both mixed and strong consistency cases. Carefully designing a contention-management system that takes advantage of the unique accord information derivable from Carol operations is an interesting line of future work.

Our distributed history merging system ran underneath in all experiment cases, including the very fast and very concurrent causal-consistency case, indicating that maintaining the arbitrary total order is not a significant runtime bottleneck.

8 Related Work

We described how our work builds on CRDTs (Shapiro et al. [20] provide a comprehensive overview). Several frameworks allow both conflict-free, and conflicting operations [3, 12, 14, 15, 21, 24], offering different trade-offs between consistency and availability. Such mixed-consistency systems are typically built upon key-value databases that offer tunable transaction isolation [2, 13, 23].

Our work is closest to the work of [12], which also focuses on on reasoning about data types with such conflicting operations. The approach of [12] 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 [12], the most important difference is how these are used. [12] 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 [3], 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 [15, 21] allows manual selection of consistency levels for operations. Quelea [21] 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 [19] 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 [24] 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 [1, 8]. 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 [4, 11] — we aim to preserve eventual consistency, while maximizing availability.

References

[1] 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

[2] Peter Bailis, 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

[3] 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

[4] E. Brewer. 2000. Towards robust distributed systems (abstract). *PODC* (2000), 7.

[5] 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

[6] 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/

[7] Sebastian Burckhardt, Manuel Fahndrich, Daan Leijen, and Mooly 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/

[8] 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

[9] 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).

[10] 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

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

[12] 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

[13] 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

[14] 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

[15] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-replicated 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

[16] Ahmed-Nacer Mehdi, Pascal Urso, Valter Balegas, and Nuno Perguiça. 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

[17] 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

[18] Patrick M. Rondon, Ming Kawaguci, 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

[19] Sudip Roy, Lucja 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.

[20] 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

[21] 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

[22] 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).

[23] 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

[24] 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.