Sequential Programming for Replicated Data Stores

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

We introduce Carol, a refinement-typed programming language for replicated data stores. The salient feature of Carol is that it allows programming and verifying replicated store operations *modularly*, without consideration of other operations that might interleave, and *sequentially*, without requiring reference to or knowledge of the concurrent execution model. This is in stark contrast with existing systems, which require understanding the concurrent interactions of all pairs of operations when developing or verifying them.

The key enabling idea is the *consistency guard*, a two-state predicate relating the locally-viewed store and the hypothetical remote store that an operation's updates may eventually be applied to, which is used by the Carol programmer to declare their precise consistency requirements. Guards appear to the programmer and refinement typechecker as simple data pre-conditions, enabling sequential reasoning, while appearing to the distributed runtime as consistency control instructions.

We implement and evaluate the Carol system in two parts: (1) the algorithm used to statically translate guards into the runtime coordination actions required to enforce them, and (2) the networked-replica runtime which executes arbitrary operations, written in a Haskell DSL, according to the Carol language semantics.

CCS Concepts: • **Computer systems organization → Peer-to-peer architectures**; • **Software and its engineering** → *Functional languages*; *Formal software verification.*

Additional Key Words and Phrases: concurrency, replicated data types, dependent types, refinement types

ACM Reference Format:
Nicholas V. Lewchenko, Arjun Radhakrishna, Akash Gaonkar, and Pavol Černý. 2019. Sequential Programming for Replicated Data Stores. *Proc. ACM Program. Lang.* 3, ICFP, Article 106 (August 2019), 28 pages. https://doi.org/10.1145/3341710

1 INTRODUCTION

When software services outgrow their computing and network resources—either because they must function on unreliable networks or because they have attracted a massively global userbase—their developers must turn to a distributed design. Distributed applications use available infrastructure more efficiently than their centralized counterparts, but do so at the price of a more difficult design process. In particular, their developers face a notoriously arduous task: the careful balancing of *consistency*, the degree to which one node can trust the state it sees when making decisions, and *availability*, the ability of nodes to continue their work during network outages. As stated by the famous CAP Theorem [Brewer 2000; Gilbert and Lynch 2002], consistency and availability are in fundamental conflict, and distributed applications are forever doomed to compromise between

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

2475-1421/2019/8-ART106
https://doi.org/10.1145/3341710

them. Given this inescapable complication, what is the highest-level programming environment that can be offered to the distributed application developer?

Replicated Data Types. One popular class of distributed system design is the *replicated store*, in which a network of application replicas store and edit copies of the application state, periodically exchanging their updates to stay in sync. The application programmer in this model is presented with a familiar, sequential-style interface: a set of objects with operations for inspecting and updating their state. The mechanisms for synchronizing these operations' effects between replicas are encapsulated in a *replicated data type* (RDT) [Burckhardt et al. 2014; Shapiro et al. 2011], isolating the programmer from the details of replication.

Unfortunately, this isolation is not perfect. The programmer must still understand the *consistency model* that their replicated store implements, and whether it is strong enough to preserve the application logic they require. The models relevant to this discussion exist on a spectrum. On one end are *eventual* and *causal consistency*, which together ensure simply that all replicas of the system will eventually converge to the same state and that no update is delivered ahead of the updates that influenced its creation. *Conflict-free replicated data types* (CRDTs) [Shapiro et al. 2011] maintain this combination without any cost to availability, and are a popular basis for applications such as event-log monitoring, chat messaging, and collaborative editing. On the other end of the spectrum is *strong consistency*, which allows replicas to work with perfect knowledge of the state by taking turns or negotiating mutex locks through a central master node. This allows operations (and sequences of operations) to execute atomically, which may be necessary to avoid double-booking a flight reservation or overdrafting a bank account, missteps that cannot be automatically fixed by a CRDT. While this model will safely support any application logic, it eliminates most benefits of replication by disallowing parallel work and halting upon network or replica failure.

Mixed Consistency. In practice, application logic that requires strong consistency is usually mixed with logic that does not. For example, a bank application may require that two withdrawals do not run concurrently on different nodes in order to avoid overdrafts, but no such danger exists for deposits. The most efficient consistency model, then, is often a *mixture* of strong and eventual. Or more precisely, a division of the application into strong and eventual components. *Mixed consistency* programming systems designed to streamline this division task include RedBlue [Li et al. 2012], in which programmers specify "`withdraw`→`Red`" or "`deposit`→`Blue`" to set strong or eventual behavior for individual operations, and Quelea [Sivaramakrishnan et al. 2015], in which programmers write explicit "`withdraw` × `withdraw`" mutex annotations that the runtime system will respect.

These systems ease consistency configuration, but they do not help a programmer find the *correct* consistency settings to choose for their application-specific safety concerns. For this purpose, several mixed-consistency verification frameworks have been proposed [Balegas et al. 2015; Gotsman et al. 2016; Li et al. 2014]. In the recent proof framework of Gotsman et al. [Gotsman et al. 2016], a programmer proves that their consistency model supports an application invariant, such as the typical "bank account never goes below zero" condition, by writing axiomatic specifications of their application's operations and declaring their consistency model's inter-operation mutex guarantees (similar to Quelea's annotations).

These tools for mixed consistency programming and verification suffer two limitations to their usability. First, they are not *modular* between operations. The consistency control annotations in RedBlue and Quelea must be chosen based on a birds-eye view of all operations that exist, and how any pair may interact to break an invariant. This carries into the proof frameworks that support them, which require similar global reasoning at the highest level of interface. Second, they do not model *sequential* programming. The programming systems offer consistency control *in terms of the* *execution model*, such that a programmer must visualize the concurrent executions they wish to avoid in order to describe them in the annotations.

Refinement Types. For an example of the kind of *modular*, *sequential* verification system that distributed programming environments are missing, we look to the world of typed functional programming. *Dependent refinement types* [Rushby et al. 1998; Xi and Pfenning 1998] (hereafter simply "refinement types") are a merging of Hindley-Milner types and predicate refinements that brings the expressivity of Floyd-Hoare logic to the functional language setting. This model of verification has been extensively studied and implemented by a number of projects, including DML [Xi and Pfenning 1998], Liquid Types [Rondon et al. 2008], and Liquid Haskell [Vazou et al. 2014]. These systems reduce the refinement components of the types to an SMT constraint problem, efficiently automating the checking process to approach the convenience of standard type systems. Also like standard type systems, refinement types can be checked modularly, in isolation of other unreferenced functions and values, and they easily compose—checking the type of a large term can make use of the checked types of its component subterms. Because they are designed to specify sequentially executed programs, however, refinement types have not (until now) been used in the verification of replicated store operations.

Our Contributions. We propose Carol, a programming language for operations over mixed-consistency replicated stores that addresses the previously discussed limitations of existing tools. First, Carol offers per-operation consistency control that is *modular*—not requiring reference to other operations—and *sequential*—defined in terms of data refinements rather than concurrent executions. Second, it provides a rich verification system, in the form of a familiar refinement type system, which is also modular and sequential in nature.

A Carol programmer controls consistency by declaring the particular aspects of the store that the network must agree upon for the duration of an operation. These aspects are described using *consistency guards*, two-state predicates that compare a running operation's view of the store to the views of remote replicas. Intuitively, programmers use consistency guards in the same way as the pre-condition assertions typical of sequential code. Having protected their operations with the appropriate guards, application programmers are excused from specifying or understanding a consistency model; the Carol runtime will infer the inter-replica coordination actions necessary to deliver the guards' guarantees.

We implement Carol to show that the higher level of abstraction we provide to programmers does not come at an unacceptable performance cost. The implementation has two parts. First, using a standard SMT solver, we implement our algorithm for statically identifying *accords*—non-interference assertions packaged with a replicated data type which the runtime references to translate arbitrary guards into coordination actions. This algorithm is based on *consistency invariants*, a new invariant notion we introduce for replicated effect executions. Second, we implement an efficient replica-network runtime for Carol operations, which adjusts its consistency model dynamically to respect their guards. We evaluate this runtime to show that the guard-driven consistency engine does not introduce unreasonable computation or communication overhead.

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 verification of pre- and post-conditions for operations on the replicated store (Section 4).
- An algorithm for inferring accords using consistency invariants (Section 5).
- An implementation of the Carol runtime system, with an evaluation showing that its unique consistency control model does not incur an unreasonable performance cost (Section 7).

2 WRITING AND VERIFYING REPLICATED OPERATIONS

We now explain the design of the Carol language by first applying it to a standard replicated programming problem and then extending the problem to highlight Carol's unique capabilities.

2.1 The Replicated Bank Application

Suppose we are creating a replicated bank application, which represents an account as an integer, and for which we require safety invariant I: that the account balance is never negative. An application in the Carol sense is defined by a set of *operations*, subroutines that execute on a single replica. Operations can query the replicated store, and can modify it by issuing *effects* which are sent to be applied at all replicas. Our bank application must support deposit, withdrawal, and check-balance operations, and these should be available (i.e. they should still function when the network is down) where possible with respect to I.

Writing Operations. We begin by defining the bank's simpler operations, demonstrating Carol's store reading and writing mechanisms:

$$\texttt{checkBalance} := \textbf{query}\ x\ \textbf{in}\ x \qquad \texttt{deposit} := \lambda n.\ \textbf{issue}\ (\texttt{Add}\ n)\ \textbf{in}\ n$$

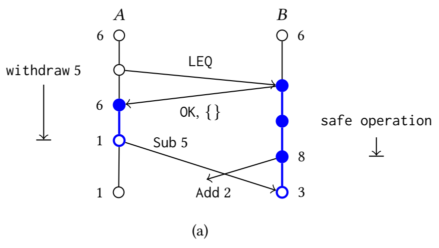
Carol is an extension of the λ-calculus; **query** x **in** t behaves similarly to $\lambda x.t$, but binds the current store value to x instead of taking an argument. The `checkBalance` operation uses this to simply return the value to the caller as its result. The **issue** e **in** t term applies e, an effect defined by the underlying store data type, to the store and then continues with t. Effects denote store-transforming functions which propagate to all replicas upon completion of an operation. For example, `Add` n denotes $\lambda s.\ s + n$. The `deposit` operation takes an amount argument n, adds it to the store, and returns it as confirmation to the caller. Neither `checkBalance` nor `deposit` have blocking behavior, so they are both available under all network conditions.

The more interesting part of the bank application is the remaining operation: `withdraw`. An operation which reduces the account value has the potential to break our invariant I, and so we must employ Carol's novel consistency control mechanism to ensure safety.

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

Here we refine the behavior of **query** using `LEQ`, a *consistency guard* which states that the store value bound to x must be less than or equal to any other store value that the operation's effects may eventually be applied against. We call these other store values *effect pre-stores* of the operation. Consistency guards in general denote two-state predicates relating the query variables they refine to effect pre-stores (the latter referred to by the special variable σ). The use of x : `LEQ` in `withdraw` denotes $x \le \sigma$, a property the programmer can rely on for the remainder of the operation. When x is bound to a concrete query result (e.g. 5), this becomes a single-state constraint on permitted effect pre-stores (e.g. σ is permitted only if $5 \le \sigma$).

Example 2.1 (Guards and effect pre-stores). Consider a network with two bank replicas $\{A, B\}$, both starting with store value 6 and executing according to Figure 1 (a). A runs `withdraw` 5, binding 6 to x in the query. Because the query used x : `LEQ`, this places the constraint $g_A = 6 \le \sigma$ on any effect pre-store σ that the `withdraw`'s effect may be applied to. The $n \le x$ check passes, and so the resulting effect is η_A = `Sub` 5. This is immediately applied to its first effect pre-store, A's store value 6, which trivially satisfies g_A ($6 \le 6$). Once the effect propagates to the other replica B, it will be applied to a second effect pre-store there. Suppose that an operation on B produces the effect η_B = `Add` 2 before η_A arrives. Then η_A's effect pre-store upon delivery at B is 8, which satisfies g_A.



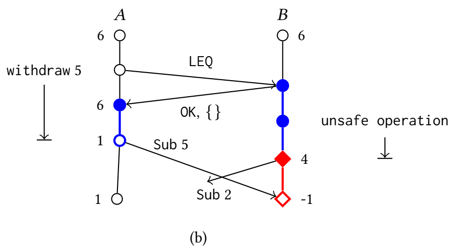


Fig. 1. A valid (a) and an invalid (b) execution under Carol. The constraint $g_A = 6 \leq \sigma$ is active in the filled sections of each replica. States are blue circles where the constraint is maintained, and red diamonds where the constraint is violated. The starting and ending nodes of Sub 5 are its effect pre-stores on A and B, respectively.

$$
\begin{aligned}
S(\texttt{Ctr}) &:= \texttt{Int} & [\![\texttt{Add } n]\!] &:= \lambda x.\ x + n \\
E(\texttt{Ctr}) &:= \texttt{Add Nat} \mid \texttt{Sub Nat} \mid \texttt{Mul Nat} & [\![\texttt{Sub } n]\!] &:= \lambda x.\ x - n \\
C(\texttt{Ctr}) &:= \top \mid \texttt{LEQ} \mid \texttt{GEQ} \mid \texttt{EQV} & [\![\texttt{Mul } n]\!] &:= \lambda x.\ x * n
\end{aligned}
$$

$$
[\![x : \top]\!] := \top \quad [\![x : \texttt{LEQ}]\!] := x \leq \sigma \quad [\![x : \texttt{GEQ}]\!] := x \geq \sigma \quad [\![x : \texttt{EQV}]\!] := x = \sigma
$$

Fig. 2. Definition of the `Ctr` CARD, showing state type S, effect type E, and guard type C. In the definitions of the consistency guards, σ refers to the effect pre-store. We use $[\![\cdot]\!]$ generically to refer to the denotations of effects and guards.

If instead, as in Figure 1 (b), it happened that η_B = Sub 2 , the effect pre-store at B for η_A would have been 4. This context does not satisfy g_A ($6 \nleq 4$), and thus the network execution leading to it would be a violation of Carol's semantics.

The Carol runtime maintains the constraints produced by queries, avoiding illegal executions, by coordinating with other replicas to block concurrent operations which could invalidate it. In the case of `withdraw`'s `LEQ` query, this means only blocking other `withdraw`s, since `deposit` and `checkBalance` cannot produce `LEQ`-breaking effects.

Compare the programming style of Carol to that of the existing mixed-consistency tools we discussed in Section 1. Regardless of tool, the network executions that need to be avoided are those in which the individually safe `withdraw` n and `withdraw` m together break the account value $s < n+m$ by running concurrently on separate replicas. But to prevent this in Carol, a programmer does not need to describe this multi-replica scenario or remind themselves what other `withdraw`-like operations have been defined. Rather, they consider the minimum information they need about the replicated store's value, and query using a consistency guard that gives them precisely that. This makes Carol's consistency control interface *sequential* (requiring no consideration of the concurrent execution model) and *modular* (requiring no knowledge of the other operations that may interleave).

Defining CARDs. To be clear, the concurrent reasoning that other tools require from programmers is not (and cannot be) simply absent from our system. Rather, we have stuffed it in a box that most programmers do not need to open. The novel, concurrency-free programming environment we have described is made possible by carefully isolating the necessary concurrent reasoning in a lower, more reusable layer of the application design that the operation programmer does not need to touch—the definition of the underlying store data type.

Operations are written over a *conflict-aware replicated data type* (CARD) that defines an interface for the store. As an example, the replicated counter CARD we have been using for the bank account's operations is given in Figure 2. A CARD's interface includes the consistency guards that operations can use in queries (such as `LEQ`) and the effects that operations can issue (such as `Add` $*$). CARDs also contain (invisible to the operation programmer) a set of *accords*, pre-computed proofs that particular effects will not invalidate particular guards. These accords are referenced at runtime to determine which effects can be safely issued by remote replicas while a guard is active. Accords depend only on guards and effects, and not on operations. Thus, they are computed (or manually identified) as part of the CARD definition, freeing the CARD's clients—the application/operation programmers—from any form of concurrency reasoning. We give an algorithm for computing sets of accords in Section 5 and evaluate our automated, SMT-based implementation in Section 7.

As an example, in the case of our `withdraw` operation's query, `Add` $*$ is in accord with `LEQ`, and `Sub` $*$ is not, and so a correct runtime will allow `deposits` to run concurrently and not `withdraws`. The declarative correctness conditions for a Carol runtime, given in Section 3, allow a variety of implementation strategies. We concretely define an efficient representative runtime in Section 6 and experimentally evaluate its real Haskell implementation in Section 7.

CARDs differ from the usual notion of a *replicated data type* by including the set of consistency guards in addition to the usual set of supported effects and leaving out replica-local logic (known variably as "prepare-updates" or "generator operations" in other systems [Li et al. 2012; Shapiro et al. 2011]). In our system, replica-local logic only appears in Carol operations (as part of an *application* rather than a datatype), making CARDs more generic. Notice that our counter CARD in Figure 2 includes guards and effects that may be useful to other applications, but which we did not need to reference or even know about to safely define our bank operations.

Operation Refinement Types. Because the Carol language is sequential in nature, we are able to adapt standard sequential reasoning tools to verifying operation behavior. Guards behave as simple pre-conditions at the operation level, and thus can be used for verification without any special concurrent logic or algorithms. This allows us to adapt a practical refinement type system, Liquid Types, to the task of verifying Carol operation properties.

Operation types in our system take the form $\{\ \mathbf{Op}\ D\ A \mid \varphi\ \}$, in which D is the CARD the operation runs over, A is the operation's return type, and φ is a logical specification relating the possible values of the operation's effect pre-stores (σ), its effect (η), and its return value (ρ). As an example, the type we check for the `withdraw` operation formally states the safety condition we described earlier as φ_1:

$$\varphi_1 := \sigma \geq 0 \Rightarrow [\![\eta]\!](\sigma) \geq 0 \qquad \varphi_2 := \rho = \sigma - [\![\eta]\!](\sigma)$$

$$\vdash \texttt{withdraw} : (n : \texttt{Nat}) \rightarrow \{\ \mathbf{Op}\ \texttt{Ctr Int} \mid \varphi_1 \wedge \varphi_2\ \}$$

Additionally, φ_2 states that the return value of `withdraw` is guaranteed to represent its effect's change to the store—it will not return n and then later revert its decision to subtract it.

We give the complete rules for operation type-checking in Section 4. At a high level, checking works by adding the guard of each **query** term to its context as a constraint on σ. Thus when the checking is complete, σ is sufficiently constrained to prove the effect and return value conditions that depend on it.

2.2 Extending the Example

We have shown how Carol can be used to solve a standard replicated programming problem at a higher level than that of existing tools. We now show how the flexibility of our programming system permits useful, non-standard operation behavior beyond the scope of these tools.

Dynamic Consistency. Carol allows **query** terms (and whole operations) to be arbitrarily nested. This allows the result of one query to determine the strength of a sub-query's guard, changing the operation's consistency model mid-execution. This property, which we call *dynamic consistency*, is not provided by other existing replicated store programming tools.

As a motivating example, we extend our bank account with an "aggressive withdraw" operation that tries harder to complete its task:

```
aggWithdraw := $\lambda n.$ if withdraw $n = n$
   then $n$
   else (query $x$ : EQV in
     if $x \geq n$ then (issue Sub $n$ in $n$) else 0)
```

This operation tries a standard `withdraw`, which only asks for a lower bound on the store when making its go/no-go decision. If that `withdraw` does not go through, `aggWithdraw` makes sure it hasn't missed any enabling deposits by querying (without releasing its hold on `LEQ`) for the *exact* value of the store using `EQV`. This will in some cases make a harder impact on availability by blocking concurrent `deposits`, but adds a useful refinement property to `aggWithdraw`'s type: $\varphi_3 := (\sigma \geq n) \Rightarrow (\rho = n)$. This is a *guarantee* that the withdrawal will go through under a particular condition, which the standard `withdraw` can not provide. Such a guarantee may be necessary for an application in which a failed withdraw triggers an expensive investigation or reporting action.

You may note that, because nested queries allow requested consistency to be value-dependent, the efficiency of their use compared to static consistency (i.e. a single `EQV` query) is also value-dependent. A programmer writing an operation like `aggWithdraw` would need to know, for example, how frequently overdrafts or near-overdrafts occur in their bank's day-to-day operations in order to understand how often their operation will trigger the higher level of consistency. But if their system needs `aggWithdraw`'s extra refinement property (perhaps for incident reporting on overdraft attempts), then the nested query solution delivers it, at worst, using the same consistency burden on the network as the static `EQV` alternative.

Non-Commutable Effects. Replicated stores can provide for concurrent, non-commutable updates by sorting and re-evaluating received updates according to agreed-upon rules (thus creating an arbitration total-order on updates), but this produces data anomalies that breaks most notions of consistency stronger than eventual. As a result, non-commutable concurrency has little support in replicated programming and verification tools. Carol's guard-based consistency model *allows concurrency* in general between non-commutable effects, automatically restricting this only when it specifically endangers a guard.

For example, let us extend our bank application with an `accrueInterest` operation that is run every once in a while, producing a `Mul 2` effect that (quite generously) doubles the account value. Because `Mul` $*$ does not commute with either `Add` $*$ or `Sub` $*$, the order of executions that include it will affect the final value, and so a runtime delivering it will need to arbitrate its place in history. However, no matter where it is inserted, it can only cause the account's value to increase, and thus it is in accord with the `LEQ` guard and `accrueInterest` can run concurrently with `withdraw`. Compare this with the recent proof framework [Gotsman et al. 2016] we discussed in Section 1, which supports an identical "accrue interest for bank account" operation *only* when the operation is made non-concurrent with all others. Our Carol runtime implementation (Sections 6, 7) takes advantage of this feature, supporting non-commutable effects using coordination-free arbitration, and we demonstrate in our experimental results that this does not incur a significant overhead.

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

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

Reusable, Composable Data Types. We have explained how concurrent reasoning, though it must always exist at some level, is isolated from the Carol language inside CARDs. This is especially useful because CARDs are generic and composable enough to be adapted to many different applications, passing through the hands of many programmers who can avoid revisiting the concurrent logic sealed inside.

As an example of this reuse, we employ our familiar `Ctr` in a new application: aircraft coordination. Suppose a replicated system for coordinating the flight-paths of aircraft—and, critically, preventing their collisions—represents aircraft positions as a map of $\{x : \mathtt{Ctr}, y : \mathtt{Ctr}, z : \mathtt{Ctr}\}$ structures. Aircraft connected to this system emit `Add` $*$ and `Sub` $*$ effects for the relevant axes to track their movement.

Now suppose a plane enters a new region of airspace and must quickly evaluate collision risks.

$$\mathtt{evalRisk} := \lambda i.\ \textbf{query}\ s : ([i].z.\mathtt{EQV})\ \textbf{in}\ (\textbf{if}\ s[i].z \neq \mathtt{myZ}\ \textbf{then}\ 0\ \textbf{else}\ \ldots)$$

The `evalRisk` operation takes an aircraft ID and first checks the z-axis position of the aircraft by querying the store using the guard $[i].z.\mathtt{EQV}$, which ensures precisely that the bound value and all remote stores agree on the exact value of i's z-position. An aircraft's altitude is generally constant for long periods of time and can be reliably held, so this query is not very demanding. If the z is found to be sufficiently distant from the newly arrived plane's z position, the operation can quickly terminate without any further coordination or measurement.

This represents an entirely new application domain, with different safety requirements and different data structures than the bank application. And yet, because it bases upon the same "primitive-type" `Ctr` as the bank, which already has the necessary guard-effect accords compiled into it, the operation development process can proceed in an entirely concurrency-free environment.

3 THE CAROL LANGUAGE

We now define Carol in detail. The language syntax is given in Section 3.1. The presentation of the semantics is divided into two parts: replica-local operation evaluation rules in Section 3.2 and distributed composition rules for operation results in Section 3.3. We combine these to state the precise requirements for a semantics-respecting Carol runtime system in Section 3.4.

3.1 Syntax and Intuition

The syntax of Carol terms, or *operations*, is shown in Figure 3. While most constructs are familiar, 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. An effect e has a denotation $[\![e]\!] \in S \rightarrow S$, 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 $\bar{e}$ 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 $S = \texttt{Int}$), 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 on a query variable x has a denotation $[\![x : c]\!] \in S \rightarrow \texttt{Bool}$ giving a predicate that relates the "local" value x to the "remote" store values (σ) that t's effects may be run against. The trivial consistency guard $[\![x : \top]\!] := \top$ and total guard $[\![x : \texttt{EQV}]\!] := x = \sigma$ are available in any Carol operation, while the "interesting" ones (represented by $\bar{c}$ 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, $[\![x : \texttt{EQV}]\!] \Rightarrow [\![x : c]\!]$. As an example, the distributed counter store we have used for our running bank account example supports $[\![x : \texttt{LEQ}]\!] := x \leq \sigma$ and $[\![x : \texttt{GEQ}]\!] := x \geq \sigma$ for querying reliable lower and upper bounds on the store, respectively. Both are reflexive, because $x = \sigma \Rightarrow x \leq \sigma$ and $x = \sigma \Rightarrow x \geq \sigma$.

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 clear data refinement on the value of x.

CARDs. We formalize the interface of guards and effects provided by a particular replicated 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 give the declarative rules in Figure 4 by which Carol terms are 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 transaction (with atomic update) 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 effects 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

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

$$\text{E-VAL}\ \frac{}{v \Downarrow \emptyset, \bullet, v} \qquad \text{E-APP}\ \frac{t_1 \Downarrow g_1, e_1, v_1 \qquad t_2 \Downarrow g_2, e_2, \lambda x.t_3 \qquad [v_1/x]t_3 \Downarrow g_3, e_3, v_3}{t_2\ t_1 \Downarrow (g_3 \cup g_2 \cup g_1), (e_3 \circ e_2 \circ e_1), v_3}$$

$$\text{E-ITE-TRUE}\ \frac{t_1 \Downarrow g_1, e_1, \texttt{true} \qquad t_2 \Downarrow g_2, e_2, v_2}{\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 \Downarrow (g_2 \cup g_1), (e_2 \circ e_1), v_2} \qquad \text{E-ITE-FALSE}\ \frac{t_1 \Downarrow g_1, e_1, \texttt{false} \qquad t_3 \Downarrow g_3, e_3, v_3}{\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 \Downarrow (g_3 \cup g_1), (e_3 \circ e_1), v_3}$$

$$\text{E-QUERY}\ \frac{[v_1/x]t \Downarrow g, e, v_2}{\textbf{query } x : c \textbf{ in } t \Downarrow (\{(c, v_1)\} \cup g), e, v_2} \qquad \text{E-ISSUE}\ \frac{t \Downarrow g, e_2, v}{\textbf{issue } e_1 \textbf{ in } t \Downarrow g, (e_2 \circ e_1), v}$$

Fig. 4. Big-step, replica-local evaluation rules for Carol terms. The E-QUERY rule does not constrain the value bound by a query (v_1), and is thus non-deterministic. Section 3.3 gives event composition rules which restrict which evaluations may coexist in a multi-replica execution, and Section 6 describes a replica implementation that binds query results deterministically based on the state of the network while following these restrictions.

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 the i-th element corresponds to a query on c_i which bound store value v_i. This set of query results forms a constraint on the concurrent network activity that the event d tolerates, restricting the contexts under which the event can be delivered and applied to a replica (the effect pre-stores discussed in Section 2).

Definition 3.1 (Guard-permitted pre-stores). A store value s is a *g-permitted pre-store*, written as $s \models g$, iff $\forall (c_i, v_i) \in g.\ [s/\sigma][\![v_i : c_i]\!]$.

Notice that some rules in Figure 4 are just more complicated versions of those standard to the CBV λ-calculus. Despite the complication of producing events, they do have the standard λ-calculus behavior—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. Yet all values are technically operations, and so the typical rules must be extended with the plumbing to collect effects and guard constraints from every sub-term. This pattern will continue in the typing rules of Section 4.

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 [Burckhardt 2014; Burckhardt et al. 2012, 2014; Gotsman et al. 2016] 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. We denote by $\text{vis}^{-1} : W \to \mathbb{P}(W)$ the set of all events witnessed by an event.

- $\mathsf{ar} \subseteq (W \times W)$ is an arbitrary total order on events, respecting vis such that $\mathsf{vis} \subseteq \mathsf{ar}$.

Definition 3.2 (Sub-executions). A *sub-execution* $L' = (W', s_0, \mathsf{eff}, \mathsf{rval}, \mathsf{vis}', \mathsf{ar}')$ of $L = (W, s_0, \mathsf{eff}, \mathsf{rval}, \mathsf{vis}, \mathsf{ar})$, 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 $L = (W, s_0, \mathsf{eff}, \mathsf{rval}, \mathsf{vis}, \mathsf{ar})$ and event $w \in W$, we call the sub-execution $L' = (W', \ldots)$ for which $w' \in W'$ iff $\mathsf{ar}(w', w)$ the *pre-execution* of w (written as L_w^{pre}). The *vis-execution* (L_w^{vis}) is similar for $\mathsf{vis}(w', w)$.

Definition 3.3 (Evaluations of abstract executions). The *store evaluation* of an abstract execution L, written as $\mathsf{eval}(L)$ is the store value arrived at by starting with s_0 and applying $\mathsf{eff}(w_i)$ for each $w_i \in W$ in ar order. Formally, if $W = \{w_0, w_1, \ldots w_n\}$ with each $i < j \implies \mathsf{ar}(w_i, w_j)$, then $\mathsf{eval}(L) = (\llbracket \mathsf{eff}(w_n) \rrbracket \circ \llbracket \mathsf{eff}(w_{n-1}) \rrbracket \cdots \llbracket \mathsf{eff}(w_0) \rrbracket)(s_0)$.

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. We relate abstract events (the elements of abstract executions) and guarded events (the products of $\Downarrow$-evaluated operations) precisely in the following definition.

Definition 3.4 (Event models). Given an abstract execution $L = (s_0, W, \mathsf{eff}, \mathsf{rval}, \mathsf{vis}, \mathsf{ar})$ 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 $\mathsf{eff}(w) = e$, $\mathsf{rval}(w) = v$, and $\forall L_r.\ L_w^{\mathrm{vis}} \subseteq L_r \subseteq L_w^{\mathrm{pre}} \Rightarrow \mathsf{eval}(L_r) \models g$.

This definition of event model gives our notion of an operation's effect pre-stores a precise meaning with respect to an abstract execution L in which w represents the operation's result—they are the values ranged over by $\{\ \mathsf{eval}(L_r) \mid L_w^{\mathrm{vis}} \subseteq L_r \subseteq L_w^{\mathrm{pre}}\ \}$. The definition states that each $\llbracket x : c \rrbracket$ generated by the queries in the operation that produced d is satisfied by all of these values.

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 runtime for Carol operations.

Definition 3.5. 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, a D-carrier must produce an abstract execution $L = (W, \ldots)$ for which there exists an intermediate set H of guarded events with one-to-one-correspondences $j : H \longrightarrow O$ and $h : H \longrightarrow W$, such that:

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

Intuitively, 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, filling in the non-deterministic choices such that the produced events fit together into an execution. In this model, operations t_1 and t_2 are ordered by $<$ iff they are in the same replica's queue.

4 REFINEMENT TYPES FOR CAROL

In this section we describe specifications for distributed store events, building up to a refinement type system for Carol. We prove soundness for this type system and describe the correctness properties of Carol carriers that the type semantics provide.

4.1 Event Specifications and Invariants

An *event specification* is a predicate with three free variables: σ represents the event's effect pre-stores, η represents the effect the event will run on, 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 pre-store 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 $\varphi := (\sigma \geq 0 \implies [\![\eta]\!](\sigma) \geq 0) \land (\rho = \sigma - [\![\eta]\!](\sigma))$ exactly states these requirements. A guarded event from a `withdraw`, for example $(\{(\mathrm{LE}, 10)\}, \texttt{Sub}\ 5, 5)$, satisfies this by placing the $[\![10 : \texttt{LEQ}]\!]$ restriction on σ.

4.2 Operation Types

We detail our type system for Carol operations, using event specifications as operation refinements, in Figure 5. This system follows the model of existing, sequential refinement type systems—in particular that of Liquid Types [Rondon et al. 2008]—but non-trivially extends this model to consider the replicated store side-effects and constraints that Carol's **query** and **issue** terms produce. Operation refinement types in our system come in two forms. First is the *base type*, of the form $\{\ \mathbf{Op}\ D\ A \mid \varphi\ \}$, in which D is the CARD the operation runs over, A is the operation's simple return type, and φ is an event specification describing the possible resulting guarded events. For example, the typing judgment

$$\vdash \texttt{withdraw}\ 5 : \{\ \mathbf{Op}\ \texttt{Ctr}\ \texttt{Int} \mid (\sigma \geq 0 \Rightarrow [\![\eta]\!](\sigma) \geq 0) \land (\rho = [\![\eta]\!](\sigma) - \sigma)\ \}$$

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.

The second is the *dependent function type*, of the form $(x : T_1) \longrightarrow T_2$, where T_1 is a base type, T_2 is a base or further dependent function type, and the event specification refinements inside T_2 can reference x. For example, the typing judgment

$$\vdash \texttt{withdraw} : (n : \texttt{Nat}) \longrightarrow \{\ \mathbf{Op}\ \texttt{Ctr}\ \texttt{Int} \mid \rho = n \lor \rho = 0\ \}$$

states that when applying any natural number to `withdraw` will either return 0 or n, the argument that was given to it. Note that, because all Carol terms are operations, the argument n is also an operation, and `Nat` is also an operation type. We omit the details of a base type for such *simple operations*, those with empty effects and no guard constraints. For example, $n : \texttt{Nat}$ as we have just used it (and as we used it in the Section 2 example) could be expanded to $n : \{\ \mathbf{Op}\ D\ \texttt{Int} \mid \rho \geq 0 \land \eta = \bullet\ \}$.

Definition 4.3 (Semantics for operation types). Given a CARD $D = (S, E, C)$, an operation t has base type $\{\ \mathbf{Op}\ D\ A \mid \varphi\ \}$ (written as $t \in \{\ \mathbf{Op}\ D\ A \mid \varphi\ \}$) iff all guarded events that t can evaluate to are well typed and satisfy φ, i.e.,

$$t \Downarrow (g, e, v) \implies \forall (c_i, v_i) \in g.\ (c_i : C \land v_i : S \land e : E) \land v : A \land (g, e, v) \models \varphi.$$

An operation t_2 has dependent function type $(x : T_1) \longrightarrow \{\ \mathbf{Op}\ D\ A \mid \varphi\ \}$ iff for any $t_1 \in T_1$ for which $t_1 \Downarrow g_1, e_1, v_1$, it is the case that $[v_1/x]t_2 \in \{\ \mathbf{Op}\ D\ A \mid [\![g]\!] \Rightarrow [\eta \circ e_1/\eta]\varphi\ \}$.

We now explain the rules of Figure 5. Intuitively, the t-query rule is similar to a conditional guard rule: if a term t is of type $\{\ \mathbf{Op}\ D\ A \mid \varphi\ \}$ given the additional premise $[\![x : c]\!]$, then term **query** $x : c$ **in** t is also of type $\{\ \mathbf{Op}\ D\ A \mid \varphi\ \}$. The t-const rule derives the simple operation

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

T-CONST
$$\frac{\mathbf{ty}(k) <: \{\nu : A \mid \varphi\}}{\Gamma \vdash k : \{\ \mathbf{Op}\ D\ A \mid \varphi \wedge \eta = \bullet\ \}}$$

T-LAM
$$\frac{\Gamma, x : \{\ \mathbf{Op}\ D\ A \mid \varphi_1\ \} \vdash t : \{\ \mathbf{Op}\ D\ A \mid [\eta \circ \eta_x/\eta]\varphi_2\ \}}{\Gamma \vdash \lambda x.t : (x : \{\ \mathbf{Op}\ D\ A \mid \varphi_1\ \}) \to \{\ \mathbf{Op}\ D\ B \mid \varphi_2\ \}}$$

T-VAR
$$\frac{\Gamma(x) = \{\ \mathbf{Op}\ D\ A \mid \varphi\ \}}{\Gamma \vdash x : \{\ \mathbf{Op}\ D\ A \mid \rho = x \wedge \eta = \bullet\ \}}$$

T-ISSUE
$$\frac{\Gamma \vdash e : \mathsf{Effect}(D) \qquad \Gamma \vdash t : \{\ \mathbf{Op}\ D\ A \mid [(\eta \circ e)/\eta]\varphi\ \}}{\Gamma \vdash \mathbf{issue}\ e\ \mathbf{in}\ t : \{\ \mathbf{Op}\ D\ A \mid \varphi\ \}}$$

T-QUERY
$$\frac{\Gamma \vdash c : \mathsf{Guard}(D) \qquad \Gamma, x : \{\ \mathbf{Op}\ D\ (\mathsf{Store}\ D) \mid [\![\rho : c]\!]\ \} \vdash t : \{\ \mathbf{Op}\ D\ A \mid \varphi\ \}}{\Gamma \vdash \mathbf{query}\ x : c\ \mathbf{in}\ t : \{\ \mathbf{Op}\ D\ A \mid \varphi\ \}}$$

T-APP
$$\frac{\Gamma \vdash t_1 : \{\ \mathbf{Op}\ D\ A \mid \varphi_1\ \} \qquad \Gamma \vdash t_2 : (x : \{\ \mathbf{Op}\ D\ A \mid \varphi_1\ \}) \to \{\ \mathbf{Op}\ D\ B \mid \varphi_2\ \}}{\Gamma \vdash t_2\ t_1 : \{\ \mathbf{Op}\ D\ B \mid [x/\rho, \eta_x/\eta]\varphi_1 \wedge \varphi_2\ \}}$$

T-ITE
$$\frac{\begin{array}{c} x \notin \Gamma \qquad \Gamma \vdash t_1 : \{\ \mathbf{Op}\ D\ \mathtt{Bool} \mid \varphi_1\ \} \qquad \Gamma, [\eta_x/\eta, \top/\rho]\varphi_1 \vdash t_2 : \{\ \mathbf{Op}\ D\ A \mid [\eta \circ \eta_x/\eta]\varphi\ \} \\ \Gamma, [\eta_x/\eta, \bot/\rho]\varphi_1 \vdash t_3 : \{\ \mathbf{Op}\ D\ A \mid [\eta \circ \eta_x/\eta]\varphi\ \} \end{array}}{\Gamma \vdash \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 : \{\ \mathbf{Op}\ D\ A \mid \varphi\ \}}$$

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

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

$$\frac{}{\Gamma \vdash \top : \mathsf{Guard}(D)} \qquad \frac{}{\Gamma \vdash \mathsf{EQV} : \mathsf{Guard}(D)} \qquad \frac{D = (S, E, C) \qquad s \in S}{\Gamma \vdash s : \mathsf{Store}(D)}$$

T-SUB
$$\frac{\Gamma \vdash t : T_1 \qquad \Gamma \vdash T_1 <: T_2 \qquad \Gamma \vdash T_2}{\Gamma \vdash t : T_2}$$

T-SUB-DECIDE
$$\frac{[\![\Gamma]\!] \wedge \varphi_1 \Rightarrow \varphi_2}{\Gamma \vdash \{\ \mathbf{Op}\ D\ A \mid \varphi_1\ \} <: \{\ \mathbf{Op}\ D\ A \mid \varphi_2\ \}}$$

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

type for a constant term from a standard refinement type judgment, stating that the value has any refinement implied by its literal definition (such as $\rho \geq 0$ when the value is 5) and no store effects or store constraints. In these rules, <: is the *subtype* relation, which states that the left hand side has the same core 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:

$$[\![\bullet]\!] := \top \qquad [\![\Gamma, x : \{\ \mathbf{Op}\ D\ A \mid \varphi\ \}]\!] := [\![\Gamma]\!] \wedge [x/\rho, \eta_x/\eta]\varphi \qquad [\![\Gamma, \varphi]\!] := [\![\Gamma]\!] \wedge \varphi$$

$$\dfrac{\dfrac{\dfrac{\begin{array}{c} n : \mathsf{Nat}, x : \{\ \mathbf{Op}\ \mathtt{Ctr}\ (\mathsf{Store}\ \mathtt{Ctr}) \mid x \leq \sigma\ \}\} \vdash (\rho \geq n \land \eta = \bullet) : \mathsf{Bool} \\ n : \mathsf{Nat}, x : \{\ \mathbf{Op}\ \mathtt{Ctr}\ (\mathsf{Store}\ \mathtt{Ctr}) \mid x \leq \sigma\ \}, \top = x \geq n \vdash \{\ldots(\mathit{then})\} : \{\ \mathbf{Op}\ \mathtt{Ctr}\ \mathtt{Int} \mid \varphi\ \} \\ n : \mathsf{Nat}, x : \{\ \mathbf{Op}\ \mathtt{Ctr}\ (\mathsf{Store}\ \mathtt{Ctr}) \mid x \leq \sigma\ \}, \bot = x \geq n \vdash \{\ldots(\mathit{else})\} : \{\ \mathbf{Op}\ \mathtt{Ctr}\ \mathtt{Int} \mid \varphi\ \} \end{array}}{n : \mathsf{Nat}, x : \{\ \mathbf{Op}\ \mathtt{Ctr}\ (\mathsf{Store}\ \mathtt{Ctr}) \mid x \leq \sigma\ \} \vdash \{\mathtt{if}\ldots\} : \{\ \mathbf{Op}\ \mathtt{Ctr}\ \mathtt{Int} \mid \varphi\ \}}\ \textsc{t-ite} \\ \dfrac{}{n : \mathsf{Nat} \vdash \mathsf{LEQ} : \mathsf{Guard}(\mathtt{Ctr})}}{n : \mathsf{Nat} \vdash \mathbf{query}\ x : \mathsf{LEQ}\ \mathbf{in}\ \{\mathtt{if}\ldots\} : \{\ \mathbf{Op}\ \mathtt{Ctr}\ \mathtt{Int} \mid \varphi\ \}}\ \textsc{t-query}}{\vdash \lambda n.\ \mathbf{query}\ x : \mathsf{LEQ}\ \mathbf{in}\ \{\mathtt{if}\ldots\} : (n : \mathsf{Nat}) \to \{\ \mathbf{Op}\ \mathtt{Ctr}\ \mathtt{Int} \mid \varphi\ \}}\ \textsc{t-lam}$$

Fig. 6. Derivation of withdraw type down to branches with base (non-**query**) terms, for Example 4.4. Certain refinement elements, such as unreferenced η_x's, have been omitted.

$$\dfrac{\dfrac{\dfrac{}{\Gamma^+ \vdash n : \mathsf{Nat}}\ \textsc{t-var}}{\Gamma^+ \vdash \mathsf{Sub}\ n : \mathsf{Effect}(\mathtt{Ctr})} \qquad \Gamma^+ \vdash n : \{\ \mathbf{Op}\ \mathtt{Ctr}\ \mathtt{Int} \mid [\eta \circ \mathsf{Sub}\ n/\eta]\varphi\ \}}{\Gamma^+ \vdash \mathbf{issue}\ \mathsf{Sub}\ n\ \mathbf{in}\ n : \{\ \mathbf{Op}\ \mathtt{Ctr}\ \mathtt{Int} \mid \varphi\ \}}\ \textsc{t-issue}$$

Fig. 7. Derivation of **issue** term in withdraw's success branch down to its final subterm n, for Example 4.4. n's type, Nat, has nothing useful to say about σ or η, so verifying φ will depend mostly on Γ^+.

$$\dfrac{\dfrac{\begin{array}{c} [\![\Gamma^+]\!] \land (\rho = n \land \eta = \bullet) \Rightarrow \\ ((\sigma \geq 0 \Rightarrow [\![\eta \circ (\mathsf{Sub}\ n)]\!]\sigma \geq 0) \land (\rho = \sigma - [\![\eta \circ (\mathsf{Sub}\ n)]\!]\sigma)) \end{array}}{\Gamma^+ \vdash T_1 <: T_2}\ \textsc{t-sub-decide} \quad \dfrac{}{\Gamma^+ \vdash n : T_1} \\ T_1 = \{\ \mathbf{Op}\ \mathtt{Ctr}\ \mathtt{Int} \mid \rho = n \land \eta = \bullet\ \} \qquad T_2 = \{\ \mathbf{Op}\ \mathtt{Ctr}\ \mathtt{Int} \mid [\eta \circ \mathsf{Sub}\ n/\eta]\varphi\ \}}{\Gamma^+ \vdash n : \{\ \mathbf{Op}\ \mathtt{Ctr}\ \mathtt{Int} \mid [\eta \circ \mathsf{Sub}\ n/\eta]\varphi\ \}}\ \textsc{t-sub}$$

Fig. 8. End of derivation for **then** branch of withdraw, in which a verification condition (the logical formula at the top) is produced to check that the context and n term's type (T_1), a simple reference to the context, is strong enough to imply the goal type (T_2). The rule that gives $\Gamma^+ \vdash n : T_1$ is T-VAR.

Example 4.4. As an end-to-end demonstration, we now typecheck the withdraw operation according to the specification we have been using, for which

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

We first follow the derivation in Figure 6, storing in the context the constraint on σ (the effect pre-store) that the query using LEQ 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, but as simple statements rather than variable bindings.

We elide the trivial **else** branch and follow the **then** branch, referring to the context so far (including $\top = x \geq n$) as Γ^+, in Figure 7. This takes us to the requirement for the final subterm of withdraw.

$$\Gamma^+ \vdash n : \{\ \mathbf{Op}\ \mathtt{Ctr}\ \mathtt{Int} \mid [\eta \circ \mathsf{Sub}\ n/\eta]\varphi\ \}$$

Looking up n in Γ^+ with the T-VAR rule, we get T_1, which is strong enough to imply our goal but isn't quite it. To go from T_1 (and Γ^+) to our final goal T_2, we generate a verification condition for their subtyping relationship with T-SUB, leaving the following logical formula to be solved.

$$\llbracket \Gamma^+ \rrbracket \land \llbracket T_1 \rrbracket \Rightarrow \llbracket T_2 \rrbracket = (n \geq 0 \land x \leq \sigma \land n \leq x) \land (\rho = n \land \eta = \bullet)$$
$$\Rightarrow ((\sigma \geq 0 \Rightarrow \llbracket \eta \circ (\mathsf{Sub}\ n) \rrbracket \sigma \geq 0) \land (\rho = \sigma - \llbracket \eta \circ (\mathsf{Sub}\ n) \rrbracket \sigma))$$

Note how η in the original φ has been replaced by $\eta \circ \mathsf{Sub}\ n$, such that T_1's statement of $\eta = \bullet$ makes the transformed φ consider only the final, concrete effect $\mathsf{Sub}\ n$. Deciding this as valid, we have thus verified that `withdraw` has our desired behavior in the replicated setting.

THEOREM 4.5 (SOUNDNESS OF TYPING RULES). *The type system presented in Figure 5 is sound, i.e.,* $\vdash t : T \implies t \in T$.

PROOF. By case analysis on the rules for deriving types. Here are the interesting cases:
Case: T-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

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

Looking at our evaluation rules, the only one that matches our term is E-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,

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

Case: T-QUERY Our premise is that

$$\forall(g_1, e_1, v_1).\ [v_x/x]t \Downarrow g_1, e_1, v_1 \land (\sigma, v_x \models c) \Rightarrow (g_1, e_1, v_1 \models \varphi)$$

for some t and c. We must show that

$$\forall(g_2, e_2, v_2).\ \textbf{query}\ x : c\ \textbf{in}\ t\ \Downarrow g_2, e_2, v_2.\ (g_2, e_2, v_2 \models \varphi).$$

Only E-QUERY matches our term, so we can be sure that for any g_2, e_2, v_2 we choose, $g_2 = \{c, v_x\} \cup g_1$, $e_2 = e_1$, 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 $(\sigma, v_x \models c) \Rightarrow ((g_1, e_1, v_1) \models \varphi$
$(\{c, v_x\} \cup g_1, e_2, v_2) \models \varphi \quad \square$
Case: T-LAM Our premise is that $[v_x/x]t_2 \Downarrow g_2, e_2, v_2 \land [v_x/\rho, \eta_x/\eta]\varphi_1 \Rightarrow (g_2, e_2, v_2 \models [\eta \circ \eta_x]\varphi_2)$. Our semantics for dependent function types requires that we show, for any t_1 for which $t_1 \Downarrow g, e, v$ and $(g, e, v) \models \varphi_1$, that $\vdash [v/x]t_2 \in \{\ \textbf{Op}\ D\ B \mid \llbracket g \rrbracket \Rightarrow [\eta \circ e/\eta]\varphi\ \}$. This translates to $[v/x]t_2 \Downarrow g_2, e_2, v_2 \Rightarrow g_2, e_2, v_2 \models (\llbracket g \rrbracket \Rightarrow [\eta \circ e/\eta]\varphi)$. Because we substituted v into t_2, our premise states that its products will satisfy $[\eta \circ e/\eta]\varphi$ as long as $[v/\rho, e/\eta]\varphi_1$ holds. And the fact that $t_1 \in \{\ \textbf{Op}\ D\ A \mid \varphi_1\ \}$ ensures that the extra precondition $\llbracket g \rrbracket$ makes $[v/\rho, e/\eta]\varphi_1$ always hold. $\square$

Soundness for CAROL types means that we can prove an operation will always produce events that satisfy 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.

Sequential 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 and allowing us to use a sequential-style verification system.

A D-carrier must identify these relationships before runtime in order to operationally ensure the semantics of guards, but the qualified interface of guards and effects 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 consistency refinements and effects in Section 5.1 and then give an algorithm for computing the relationships within a given CARD in Section 5.2.

5.1 Accords Between Guards and Effects

We now work to a definition of the guard-effect accords (hereafter referred to simply as accords) that a CARD must hold in order to efficiently enforce guards at runtime. Intuitively, an accord states that a particular effect (or class of effects) cannot invalidate a particular consistency guard's permitted effect pre-stores, no matter when or how it is applied to them.

Definition 5.1 (Compliance). For a CARD D and D-guard c, two D-states (s_1,s_2) are (D,c)-compliant iff $\forall e : \mathsf{Effect}(D).\ [[\![e]\!](s_1)/\sigma][\![[\![e]\!](s_2) : c]\!]$.

Note that $s_1 = s_2$ trivially ensures (s_1, s_2) is (D,c)-compliant, and that the definition covers the case where $e = \bullet$ such that $[s_1/\sigma][\![s_2 : c]\!]$ is satisfied (i.e. s_1 is a permitted effect pre-store for $s_2 : c$).

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 $(\mathtt{Ctr}, \mathtt{Le})$-compliant when $s_r \leq s_g$, since they satisfy `Le` and no `Ctr` effect applied to both can invalidate that. Then, `Le` and `Add` n are in accord, since a deposit would only increase s_g. However, `Le` and `Sub` n are not in accord, as a large enough n will 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. We apply the 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. We insert new in-accord effects into e^* end-first, stepping them to their f^* location by repeated applications of Lemma 5.4. Since the starting state (s_0, s_0) is (D,c)-compliant by $s_0 = s_0$, we fulfill the first application's requirement that the current state-pair is (D,c)-compliant, and by using the lemma, we continually ensure that the next state-pair is (D,c)-compliant, meeting the requirements of the next application.

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.

$$S(\texttt{PostSolver}) := (A : \Sigma^*, B : \Sigma^*, sol : \{\top, \bot\}) \qquad [\![x : \top]\!] := \top$$
$$E(\texttt{PostSolver}) := \texttt{Append } \{1, 2, \cdots, n\} \qquad [\![x : \texttt{SOLEQV}]\!] := sol_x = sol_\sigma$$
$$C(\texttt{PostSolver}) := \top \mid \texttt{SOLEQV} \mid \texttt{EQV} \qquad [\![x : \texttt{EQV}]\!] := (A_x, B_x, sol_x) = (A_\sigma, B_\sigma, sol_\sigma)$$
$$[\![\texttt{Append } i]\!] := \lambda(A, B, sol).\ (Aa_i, Bb_i, sol \vee Aa_i = Bb_i)$$

Fig. 9. The CARD for our Post Correspondence Problem solver. The state is a tuple (A, B, sol) representing the concatenation $a_{i_1} \cdots a_{i_k}$, the concatenation $b_{i_1} \cdots b_{i_k}$, and whether a solution has been found. The subscripts r and g stand for replica and global state respectively.

Intuitively, the more accords a D-carrier system has, the less contention is needed to run correctly. An *accord set* $\text{AS}(c)$ for a guard c is a set of effects which are in accord with c. The following theorem states that finding the largest such 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. We show that if finding the largest cardinality accord set is decidable, the Post Correspondence Problem is decidable. For an alphabet, Σ, given two sets of strings $\{a_1, a_2, \cdots, a_n\}$ and $\{b_1, b_2, \cdots, b_n\}$ in Σ^*, the goal of the Post Correspondence Problem is to decide whether or not there is some list of indices $i_1, i_2, \cdots, i_k$ such that $a_{i_1} a_{i_2} \cdots a_{i_k} = b_{i_1} b_{i_2} \cdots b_{i_k}$, i.e. the concatenations of the corresponding strings in each set are equal. We now show this can be done by finding the largest cardinality accord set for the `PostSolver` CARD in Figure 9.

Consider an arbitrary instance of the Post Correspondence Problem, and its associated `PostSolver` CARD. If there is some list of indices $i_1 \cdots i_k$ such that $a_{i_1} \cdots a_{i_k} = b_{i_1} \cdots b_{i_k}$, then we know that there is a sequence of effects $e^* = \texttt{Append } i_k \circ \texttt{Append } i_{k-1} \circ \cdots \circ \texttt{Append } i_1$, applied to the starting state $s_0 = (\varepsilon, \varepsilon, \bot)$ such that $[\![e^*]\!](s_0) = [\![\texttt{Append } i_k \circ \texttt{Append } i_{k-1} \circ \cdots \circ \texttt{Append } i_1]\!]((\varepsilon, \varepsilon, \bot)) = (a_{i_1} \cdots a_{i_k}, b_{i_1} \cdots b_{i_k}, \top)$. Since $\neg[s_0/\sigma][\![[\![e^*]\!](s_0) : \texttt{SOLEQV}]\!]$, we can take the contrapositive of Theorem 5.5 to know that at least one of $\texttt{Append } i_1, \cdots, \texttt{Append } i_k$ must not be in accord with SOLEQV.

On the other hand, if no list of indices is a solution, then we use the fact that two states s_1, s_2 are (`PostSolver`,SOLEQV)-compliant iff their solution status is the same. If s_1 has the form $(A, B, \bot)$, then because no effect Append i reaches a solution, we know all effects produce a state $(A', B', \bot)$, maintaining (`PostSolver`,SOLEQV)-compliance. Likewise, if s_1 has the form $(A, B, \top)$, every effect will produce $(A', B', \top)$ by definition of Append, also maintaining compliance.

Thus, if we can find the largest cardinality accord set for SOLEQV, we can check if all of $\texttt{Append } 1, \cdots, \texttt{Append } n$ are within the set: if some are absent, then there is a solution; while if all are present, then no solutions exist. This is a contradiction since the Post Correspondence Problem is undecidable. □

5.2 Finding Accord Sets

Finding accord sets 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 $[s_1/\sigma][\![s_2 : c]\!] \implies [[\![e]\!](s_1)/\sigma][\![s_2 : c]\!]$.

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

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. 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 $[\![x : c']\!] \Rightarrow [\![x : c]\!]$, then $AS1(c') \subseteq AS(c)$.*

Proof. This is immediate from the definitions 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, $\mathsf{WCP}(e, c)$, be the weakest guard such that $(s_g, s_r) \models \mathsf{WCP}(e, c)$ 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:

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

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 $\forall e : E.\ c_i \Rightarrow \mathsf{WCP}(e, c_i)$.
- 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 $\mathrm{PAS}_D(c)$ 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 to 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 `Ctr` 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 `Ctr` 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 the identified accord sets 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 10, 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 acquisitions 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 10. 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 $L_w^{\text{vis}} \subseteq L_i'$.

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 r-issue rule. When it finishes evaluating the operation, it can emit the combined changes and return.
Operation completion. The r-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 trivially 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.

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

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

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

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

Fig. 10. Core replica execution rules. Rules which only update a subterm and pass through context modifications to the parent term are omitted. $\text{AS}_D(c_r)$, as used in R-VAL, refers to an accord set statically inferred for c_r. When we use abstract executions as sets, as in $w' \in (L \cup \{w\})$, we mean the contained event set W.

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

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

Fig. 11. Replica-driven operation evaluation rules. These rules give an algorithm that consumes a terminating network execution to evaluate the operations of one replica into guarded events.

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, $(L, w) \models d$.

Identifying Guarded Events. To identify guarded events for operations, we define an algorithm in Figure 11 that runs over a terminating network execution with rules of the form $j, t, g, e \longmapsto j', t', g', e'$, where j is a 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. 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-R-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 R-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 R-VAL steps that follow. Each following R-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 existence 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) $\forall d_1, d_2 \in H.\ j(d_1) < j(d_2) \Rightarrow \text{vis}(h(d_1), h(d_2))$, 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 Total Order

In order to support non-commutable effects, our replica network design uses a total arbitration order. 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, except where forced by the use of guards). Events incomparable in the causal order can be simply compared by their replica ID to achieve deterministic ordering, the same at each replica. This is a standard technique for deciding total orders on distributed system events [Lamport 1978]. 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 majority of settings. Unlike other consistency models, consistency guards enable a programmer to work safely even in the presence of recomputation anomalies.

6.3 Consistency Coordination Protocol

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 R-QUERY rule requires that all not-in-accord events have been seen by the querying replica at the time of evaluation, and both R-QUERY and R-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.

R-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 R-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 acknowledgment 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 effects that are in accord with c, and by merging its abstract execution with the ones it has received (a series of R-DELIVERS), it has met the requirements of R-QUERY and can proceed with the evaluation.

R-VAL. Now consider a replica seeking return and emit the combined staged effects e via the R-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. First, (Section 7.1) is the inference of guard-effect accord sets via the PAS algorithm efficient for a variety of CARDs? Second, (Section 7.2) is the runtime performance of a CAROL carrier implementation scalable to real geo-distributed applications?

7.1 Static Accord Identification (DSV)

We empirically evaluated whether our algorithm for the core computational task of defining CARDs—inferring accord sets—is efficient and complete. We implemented the PAS algorithm, using

Table 1. Accord set inference evaluation results.

Application	Guards	Effect Classes	Time (ms)	Complete?
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

the Z3 SMT solver [De Moura and Bjørner 2008] for logical reasoning, as a Haskell library called *DSV* [1]. We modeled CARDs of varying complexity, using SMT-representable integers, booleans, and arrays, and computed accord sets for their consistency guards and effects. Each CARD's consistency guards included the empty guard, the total guard (the identity relation), and interesting non-trivial guards that may provide useful information to an operation. The results of our evaluation are given in Figure 1. For all tested CARDs, our solver found accord sets in less than 175ms. The times are favorably comparable with the evaluation results of previous systems such as Indigo (ranging 19ms to 320ms) [Balegas et al. 2015] and the verifier of Gotsman et al. (ranging 385ms to 5297ms) [Gotsman et al. 2016], with an important caveat: our PAS algorithm analyzes only the concurrent interaction of a CARD's guards and effects, leaving sequential operation code to a refinement type checker. These previous systems analyze full applications, including the sequential code, all at once.

The data types we used for the CARDs in our experiment are comparable in complexity to those on which previous verification systems have been evaluated. While they may appear toy-like in comparison to data structures used by practical production systems, we note that complex real-world structures tend to be compositions of smaller "primitive" datatypes; we believe that encountering an indivisible structure with 9+ effect classes is unlikely (indeed, our stretched-out KV bank accounts example could be broken down in this way). The complete analysis of such compound structures could mostly reuse the analysis results of their parts; a flexible framework for such "ecological" accord set analyses is an interesting line of future work.

Manual examination proved that the inferred accord sets were complete, i.e. that they contained all guard-effect pairs that were semantically in accord. We detail three cases.

Bank Account with Reset. We extended the `Ctr` 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 resetting operation is empty.

Finite State Machine. We modeled a finite state machine CARD for which the store values were the set of $\{s_a, s_b, s_c\}$ states and the effects were a set of (non-commutable) transition labels between them. This served to further test our algorithm's sensitivity to the impact of non-commutability anomalies on accords. We defined a guard $[\![x : \texttt{InState}(s_c)]\!] := x = s_c \Leftrightarrow \sigma = s_c$ for each state s_c which provides precise knowledge of whether any replica is in the critical state s_c. The accord set for this guard included not only "offenders"—those effects leading directly into and out of s_c—but also any that enabled the offenders by taking the system closer to s_c in some way.

Key-Value Bank Accounts. This example modeled an array of ten indexed bank accounts, supporting the same effect classes as the regular bank account but with an additional index parameter. The

[1] https://github.com/cuplv/dsv

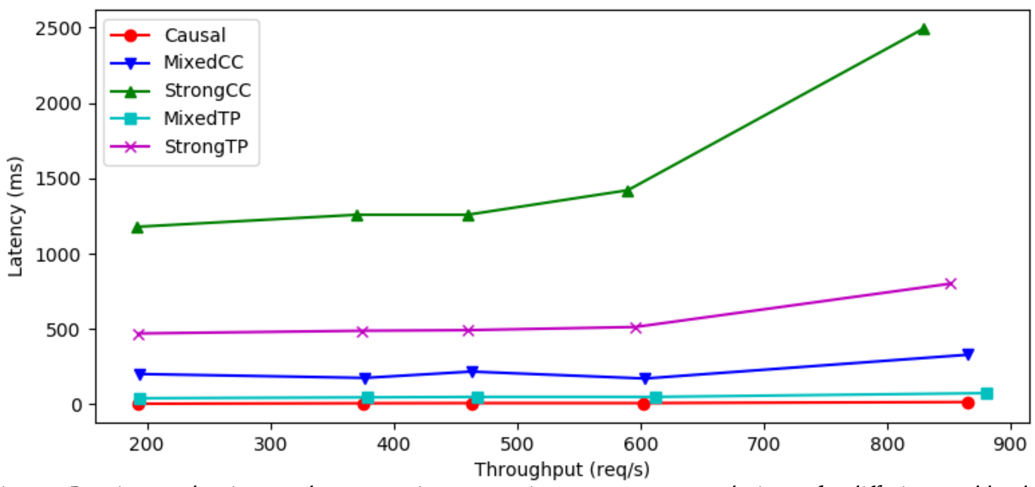


Fig. 12. Runtime evaluation results, comparing contention-management techniques for differing workloads. *CC* refers to the congestion control technique and *TP* to token passing. *Causal* refers to workloads including no guards, *mixed* to mixed-consistency workloads, and `strong` to workloads in which all operations conflict.

logical reasoning for this example involved using the array SMT theory. We inferred accord sets both for guards on individual accounts, and new guards concerning the summation of the accounts. The accord set for the summation `LEQ` guard included the `Sub` $*$ effects for all bank accounts, while the accord sets for the individual account `LEQ` guards included only the `Sub` $*$ effects for that bank account. This illustrates that our algorithm automatically discovers boundaries to an effect's interference on complex states.

7.2 Runtime Design and Performance (Discard)

In this section we explain the design of our concrete Carol runtime and then discuss its measured performance. Our implementation is a Haskell library called *Discard* [2], which realizes the language as a monadic DSL (structured to provide the proper CBV semantics) and provides a replica system that correctly evaluates the embedded Carol operations according to the model given in Section 6. Guard-preserving coordination, based on the CARDs' accord sets, is managed using a state-based CvRDT [Shapiro et al. 2011] (a state with a monotonic merge function) similar to the abstract coordination configuration used in the Figure 10 replica rules. This shared CvRDT maps replicas to their active 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.

[2]https://github.com/cuplv/discard

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 for that guard.

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 (IPFS [Teixeira 2017] v0.4.17), headed by a vector clock for efficient merges. This model of history is required to maintain the arbitration total-order that allows applications to use non-commutable effects. In our evaluation, we looked to confirm that this method of store distribution scales to a non-trivial workload.

Experiment. We ran replicas on two machines, geo-separated such that their round-trip-time 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 at which the operations completed.

Discussion. Our results in Figure 12 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. 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 coordination-free arbitrary total order is not a significant runtime bottleneck.

This implementation and its basic evaluation is only a preliminary step in the study of runtime execution strategies that follow Carol's novel consistency model. The purpose of this small experiment has been to validate the model—to show that the unique requirements of our language's semantics are not basically intractable. Though the token-passing strategy won in our experimental setup, we believe the congestion-control approach should not be ignored in future exploration; it may prove more suitable as network size increases or latency decreases.

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 that of [Gotsman et al. 2016], which also focuses on reasoning about data types with conflicting operations. However, our contributions diverge from theirs in three major ways. First, we contribute a novel mechanism for data-centric consistency control (queries with consistency guards), in our fully specified programming language Carol. It is data-centric, as it allows a programmer to control their application's synchronization in terms of assertions on operation-local data values. We further implement a runtime engine that uses guards to maintain consistency. The work of [Gotsman et al. 2016] does not offer a programming language or novel consistency control engine; their contribution is a proof framework. Second, comparing the proof frameworks: we define a *modular* proof system for operations in the user-friendly form of a refinement type system. Our type system allows us to verify safety properties for a Carol operation in isolation from other operations of the application. In contrast, in the system of [Gotsman et al. 2016], when adding new a operation to an application, a user needs to specify the conflicts with all existing operations. Third, our type system can verify the behavior of two kinds of useful operations that are not supported by [Gotsman et al. 2016]: nested-query operations and non-commutable, yet coordination-free operations. These are explained in full in Section 2.2.

The second closest work is that of [Balegas et al. 2015]. They introduce *explicit consistency*, in which concurrent executions are restricted by 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. Thus only consistency guards can ensure a property like "if the query returns a value v, then the account balance is at least v" (see the bank account with interest in Section 2.2). Second, our consistency guards allow verification by checking conditions on sequential programs. In contrast, application invariants of Balegas et al. require checking conditions on concurrent programs, a significantly harder task. A more recent work in this line, IPA [Balegas et al. 2018], follows the identification of invariant violations with suggestions of coordination-free mechanisms to fix them. These are useful, but limited in the problems that can be fixed; our work is motivated instead by applications that cannot escape the use of coordination for certain tasks.

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. Quelea contracts are also statically assigned to operations; they do not support dynamic consistency. A recent approach to verification, Q9 [Kaki et al. 2018], uses bounded symbolic execution to detect invariant violations under static consistency levels (similar to Quelea contracts). For future work, a similar bounded symbolic execution algorithm could be useful in the automatic inference of accords for complex CARD store types.

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

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 AND FUTURE WORK

We have presented Carol, a programming language for replicated data store operations. Carol features a novel consistency control mechanism, the consistency guard, which allows store operations to be written and verified in a modular and sequential style not previously available in the concurrent execution setting. In support of this new programming model, we have described and implemented representatives of the non-standard engines for static conflict analysis and runtime conflict avoidance it requires.

We believe these implemented designs are the seeds of two distinct directions for future work. The first branch is the design of analysis algorithms and proof frameworks for efficient, complete accord-set generation for more complex store datatypes. The second branch is the design of efficient, adaptable runtime evaluation strategies and optimizations that implement Carol's uniquely flexible dynamic consistency model. In particular, we would like to find environment-specific runtime designs which optimize for specific network sizes or topologies while implementing the same language semantics. We intend for our formalization of the Carol language's semantics and execution requirements to clearly mark out the boundaries of this interesting new design space.

ACKNOWLEDGMENTS

We thank the ICFP'19 reviewers and those readers of earlier drafts for their time and the feedback they have provided on this work. This research was supported in part by DARPA under agreement FA8750-15-2-0096.

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 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

Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, and Nuno Preguiça. 2018. IPA: Invariant-preserving Applications for Weakly Consistent Replicated Databases. *Proc. VLDB Endow.* 12, 4 (Dec. 2018), 404–418. https://doi.org/10.14778/3297753.3297760

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.

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 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/

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

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

Seth Gilbert and Nancy Lynch. 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News* 33, 2 (June 2002), 51–59. https://doi.org/10.1145/564585.564601

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

Gowtham Kaki, Kapil Earanky, KC Sivaramakrishnan, and Suresh Jagannathan. 2018. Safe Replication Through Bounded Concurrency Verification. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 164 (Oct. 2018), 27 pages. https://doi.org/10.1145/3276534

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

Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

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 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

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

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.

John Rushby, Sam Owre, and Natarajan Shankar. 1998. Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE Trans. Softw. Eng.* 24, 9 (Sept. 1998), 709–720. https://doi.org/10.1109/32.713327

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.

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.

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 269–282. https://doi.org/10.1145/2628136.2628161

Hongwei Xi and Frank Pfenning. 1998. Eliminating Array Bound Checking Through Dependent Types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98)*. ACM, New York, NY, USA, 249–257. https://doi.org/10.1145/277650.277732