Safe Replication through Bounded Concurrency Verification

GOWTHAM KAKI, Purdue University, USA
KAPIL EARANKY, Purdue University, USA
KC SIVARAMAKRISHNAN, University of Cambridge, UK
SURESH JAGANNATHAN, Purdue University, USA

High-level data types are often associated with semantic invariants that must be preserved by any correct implementation. While having implementations enforce strong guarantees such as linearizability or serializability can often be used to prevent invariant violations in concurrent settings, such mechanisms are impractical in geo-distributed replicated environments, the platform of choice for many scalable Web services. To achieve high-availability essential to this domain, these environments admit various forms of weak consistency that do not guarantee all replicas have a consistent view of an application’s state. Consequently, they often admit difficult-to-understand anomalous behaviors that violate a data type’s invariants, but which are extremely challenging, even for experts, to understand and debug.

In this paper, we propose a novel programming framework for replicated data types (RDTs) equipped with an automatic (bounded) verification technique that discovers and fixes weak consistency anomalies. Our approach, implemented in a tool called Q9, involves systematically exploring the state space of an application executing on top of an eventually consistent data store, under an unrestricted consistency model but with a finite concurrency bound. Q9 uncovers anomalies (i.e., invariant violations) that manifest as finite counterexamples, and automatically generates repairs for such anomalies by selectively strengthening consistency guarantees for specific operations. Using Q9, we have uncovered a range of subtle anomalies in implementations of well-known benchmarks, and have been able to apply the repairs it mandates to effectively eliminate them. Notably, these benchmarks were written adopting best practices suggested to manage distributed replicated state (e.g., they are composed of provably convergent RDTs (CRDTs), avoid mutable state, etc.). While the safety guarantees offered by our technique are constrained by the concurrency bound, we show that in practice, proving bounded safety guarantees typically generalizes to the unbounded case.

ACM Reference Format:

1 INTRODUCTION

Replicated data types (RDTs) are a common abstraction used to program sophisticated distributed applications intended to operate in geo-replicated environments [Burckhardt et al. 2014]. These environments often expose a storage layer in which different replicas may hold different states of
the same RDT instance. This occurs because RDT operations may not be atomically and uniformly applied across all replicas, as a consequence of network partitions and failures, weak ordering and delivery guarantees, etc. [Brewer 2000; Gilbert and Lynch 2002]. Consequently, replicas may hold different, potentially irreconcilable, views of an RDT object’s state as the program executes. This situation can be mitigated in some cases by carefully engineering RDT implementations to provide certain (albeit weak) properties like commutativity and convergence that ensure all instances of a replicated object found on different replicas will eventually reflect the same state [Balegas et al. 2015; Shapiro et al. 2011a,b].

In general, however, it remains a challenging exercise to determine if a distributed program built around RDTs satisfies application-specific safety properties. While recent work has shown that it is possible to check the integrity of a distributed application by employing heavyweight specification and verification techniques [Bouajjani et al. 2017; Burckhardt et al. 2014; Gotsman et al. 2016; Padon et al. 2017a, 2016; Wilcox et al. 2015], doing so with a high-degree of automation has remained an open (and important) challenge. Full verification of distributed applications is challenging in part because of the need to specify suitably strong inductive invariants [Padon et al. 2016] that justify the validity of the safety property being verified in terms of the actions performed by the application; such invariants capture deep semantic properties and are thus difficult to extract automatically. Furthermore, modern distributed storage layers [Alvaro et al. 2013; Bailis and Ghodsi 2013] provide relatively weak consistency guarantees because they are designed to achieve high-availability (low-latency) access to data, and thus expose a semantics that is inconsistent with easily-understood serializable or sequentially-consistent executions [Brutschy et al. 2017]. The tension between the need for expressive high-level invariants to facilitate verification and the requirement that any verification mechanism be robust in the face of a weakly consistent storage layer greatly complicates how we formulate and prove precise correctness arguments.

Rather than tackling the problem of full (unbounded) verification head-on, we instead consider an alternative fully-automated lightweight verification strategy that leverages symbolic execution to provide bounded guarantees on a program’s correctness. Our verification strategy explores a search space of abstract executions in which each point in the space represents a global program state parameterized over a bounded number of concurrent effects. A concurrent effect captures an effectful operation on an RDT instance that has not yet been applied. Because multiple concurrent effects may be serviced in different order on different replicas, our symbolic execution engine considers distinct permutations over sets of effects. We determine a safety property’s validity by considering all such executions, limiting the number of concurrent effects (and, by extension, the number of replicas that process these effects), to retain a tractable analysis. Other than the specification of the safety property, our approach does not require any additional programmer involvement.

Our symbolic execution engine (called Q9) operates over RDT-centric distributed applications. The engine abstracts executions in terms of path conditions and RDT operations, under an axiomatization of a data storage model that only provides weak eventual consistency guarantees on object updates. The engine checks application-specific safety properties on different state configurations induced by considering executions in which the visibility and ordering of RDT operations on different replicas may vary. Q9 tracks operations precisely (up to a bound), thus ensuring that every violation of a safety property is a true violation. Over a collection of benchmark results, including well-studied database applications [TPC 2018], Q9 was able to correctly identify anomalies that

---

1The number 9 in Q9 refers to our initial hypothesis that most replication anomalies manifest under 9 or fewer concurrent operations. The letter ‘q’ is a symbol resembling 9, hinting at our approach of using symbolic execution to uncover such anomalies.
arise because satisfiability of the application’s safety properties demand greater coordination and synchronization than manifest explicitly in the application or which is implicitly supported by the storage layer. Counterexamples generated by Q9 are used to automatically strengthen the consistency level (i.e., the degree of global synchronization required) of offending operations. Empirical results support our thesis that anomalies can be detected quickly under relatively small bounds, and repaired easily by selectively strengthening consistency requirements on RDT operations to enforce greater coordination among replicas.

The paper makes the following contributions:

(1) We define a programming framework embedded in OCaml that allows the expression of replicated data types and effectful computations over instances of these types. Our formulation has a natural runtime interpretation in terms of effectful messages sent among replicas. These messages define complex, potentially transactional, actions whose evaluation is triggered on the replicas that receive them.

(2) We describe Q9, a symbolic execution engine for programs built around replicated data types in our framework. The engine explores a search space of executions, bounded by the number of concurrent effects yet to be processed by different replicas. Each execution explores a feasible configuration in which replicas may have some number of outstanding effects (up to the specified bound) that need to be processed, with ordering constraints determined by axiomatically-defined visibility and happens-before relations incrementally constructed as execution proceeds.

(3) We present experimental evidence over a range of benchmarks, including well-studied database applications that demonstrate Q9 can precisely identify anomalies, i.e., concrete executions that violate application-specific safety properties, even with relatively small bounds on the number of concurrent effects tracked during symbolic execution. The counterexamples generated by the engine can be used to automatically strengthen the consistency requirements of offending operations to eliminate anomalies; this strengthening mechanism chooses the weakest consistency level that nonetheless satisfies the safety property under the concurrent effect bound.

To the best of our knowledge, Q9 is the first fully-automated anomaly detection and repair mechanism for distributed applications intended to execute on weakly-consistent replicated data stores.

The remainder of the paper is organized as follows. The next section motivates the problem of reasoning about replicated objects and sets the context for symbolic execution. In Section 3, we introduce a programming model for expressing RDTs and reasoning about effects. Section 4 introduces a general system model that defines a communication mechanism in which replicas broadcast effects, that are eventually executed on each replica that receives them. We present a formal description of Q9 in Section 5 that defines concrete and symbolic executions, safety properties, and a repair strategy developed within this context. Section 6 discusses how symbolic execution generalizes to transactions that manipulate multiple RDT instances. Details about Q9’s implementation and evaluation results are given in Section 7. Sections 8 and 9 present related work and conclusions, resp.

## 2 REPLICATED STATE ANOMALIES: THE MOTIVATION FOR VERIFICATION

Consider a simple distributed application that maintains a bank account with replicated state. The representation of a bank account may be in terms of a convergent replicated type (e.g., an integer PN-counter [Shapiro et al. 2011a] that admits increments and decrements) that guarantees all replicas will eventually reflect the same value of the account. However, convergence alone may not be sufficient to preserve application-specific safety properties. For example, suppose we wish to
assert that the balance of the account will always be non-negative. Given operations to deposit and withdraw amounts into the account, it is straightforward, in a sequential execution, to ensure this invariant is always preserved, by ensuring that deposit only ever adds to the balance, and that withdraw always checks if there is a sufficient balance before withdrawing. However, asynchronous replication may lead to anomalous executions that violate the invariant. Two such executions that are illustrative of the anomalies possible under asynchronous replication are shown in Fig. 1.

Fig. 1a depicts an execution that allows two withdraw operations to be applied concurrently at different replicas. Two users, Alice and Bob, assume that there is a sufficient balance in their (joint) account, and issue a withdraw operation for $1 each, to replicas $R_1$ and $R_2$, respectively. Each withdraw operation reads the local balance ($1$), checks that it is sufficient for the withdraw ($1 \geq 1$), and subsequently issues a Withdraw effect that will be asynchronously transmitted to the other replica. The effect is essentially a computational message that updates the state of the account on the replicas which receive it. When both Withdraw effects are eventually applied at both the replicas, the balance drops below $0$ resulting in an invariant violation, which gets witnessed by Alice when she queries the balance. The anomaly is reminiscent of a classical data race between two writes in a shared memory system, except that writes are not lost or overwritten.

However, unintended executions that are unfamiliar to shared memory programmers are also possible in an asynchronous replicated system. Consider the execution shown in Fig. 1b involving three users - Alice, Bob, and Cheryl, and two replicas - $R_1$ and $R_2$. The initial balance at both the replicas is $0$. Alice first submits a deposit operation for $1$ to $R_1$. Bob, who subsequently connects to $R_1$, finds there is sufficient balance to perform a withdraw for $1$. While effects from both the operations are expected to be delivered to $R_2$, it is possible that because of transient network conditions, the Withdraw effect gets delivered first while the Deposit is still in transit. This results in a transient violation of the no-negative-balance invariant, which gets witnessed when Cheryl queries the balance at $R_2$. The Deposit effect will eventually be delivered to $R_2$ resulting in the invariant being restored; however since there are no bounds on when this can happen, there are no guarantees on how this violation may affect system behavior. Indeed, it is possible that a temporary violation of safety may lead to cascading errors that compromise application integrity. For instance, a negative (albeit temporary) balance could be witnessed by a minimum balance enforcement.

\[^2\text{Note that we cannot rectify this anomaly by simply forcing each replica to check the balance before applying a received Withdraw effect as that may cause the account balance on different replicas not to converge.}\]
module BankAccount (C:CRInt) : BANKACCT =

struct
  type t = C.t
  type m = C.m

  let init = C.init_val

  let get_balance () : int =
    C.get ()

  let do_deposit ( amt : int ) : m =
    C.add amt

  let do_withdraw ( amt : int ) : m =
    if C.get () >= amt
      then C.add (0 - amt )
      else error "Insufficient Balance"

  let inv_non_neg_bal () : bool =
    CRInt.get () >= 0
end

module CRInt : sig
  type t
  type m = t -> t

  val init_val : t
  val add : int -> m
  val get : unit -> int
end

module Bank (Checking : BANKACCT) (Savings : BANKACCT) =

struct
  type t = Checking.t * Savings.t
  type m = t -> t

  let txn_transfer ( amt : int ) : m =
    let m1 = Checking.do_withdraw
      let m2 = Savings.do_deposit
      fun (x,y) -> (m1 x, m2 y)
end

Fig. 2. A Bank Account application written in Q9

module, which may erroneously impose a penalty on the account that remains even after the violation is remedied. As this example demonstrates, asynchronous replication of an application’s state can result in anomalous behaviors that could be confounding to understand and repair. It is clearly unreasonable to expect an application programmer to be prescient about the anomalies that might manifest under replication, determine if they indeed lead to invariant violations, and fix the application to avoid them.

Q9’s verification engine is based on the observation that concurrency anomalies under replication most often have small representative counterexamples involving few concurrent operations, similar to those shown in Fig. 1. Such anomalies can be exposed by exploring the state space of the application with a relatively small bound on the number of concurrent operations. Moreover, by representing the state space using an appropriate formal vocabulary that abstracts away low-level details, such as process crashes and network faults, we can compute an abstract representation of each counterexample that represents not only the counterexample, but an entire class of such counterexamples. Systematically eliminating such classes of counterexamples by consistency strengthening leads us to compute the weakest consistency configuration at which an application is free of all discovered anomalies, and hence most likely to be safe.

3 THE Q9 PROGRAMMING FRAMEWORK

The first component of Q9 is a programming framework implemented as a collection of type definitions and libraries in OCaml, intended to operate within a replicated, eventually consistent, distributed environment; for this purpose, it comes equipped with a library of convergent RDT [Shapiro et al. 2011a] definitions. The semantics of a CRDT object guarantee that even when multiple operations are applied to it in different order on different replicas, the object’s state at all
replicas, after all operations have been delivered and executed (i.e., when the system becomes quiescent), will be the same. Q9 uses CRDT specifications to specify richer RDTs through compositional abstractions, capturing notions of convergent state replication for free without having to define a specialized network/storage layer, or to prove additional semantic properties (e.g., commutativity) for the sake of convergence. In this section, we illustrate the programming model with the help of an example.

Fig. 2 shows a simple BankAccount application written in Q9 that was informally introduced in the previous section. The application manages a replicated object (a bank account) whose underlying representation is given in terms of a CRDT integer (C.t). The signature of a CRInt defines two operations: get returns the current value of the integer, and add adds its argument to the existing value of the integer. Observe that while the signature for get is unremarkable, add’s type returns a value of type m, a function type with signature C.t -> C.t. This return type captures the essence of an effectful operation in a replicated setting. The function returned by add is intended to be applied to every instance of C on every replica in the distributed environment managed by Q9 supplying the value of the integer (C.t) at that replica as the argument to this function. On the other hand, get’s signature does not appeal to m - it is expected to return the value of the integer on the replica to which it is applied, in contrast to add. This particular specification of CRDTs allows us to hide the implementation artifacts of replication behind high-level signatures that characterize an operation’s local and remote effects.

The BankAccount module defines standard banking operations that internally manipulate the CRInt CRDT. The operations include get_balance, do_deposit, and do_withdraw. get_balance function is standard - like C.get, it returns the integer value of the current state on whatever replica it is applied to. Operations do_deposit and do_withdraw, on the other hand, are effectful. These operations return an effect, which is essentially a computation that must eventually be performed on all replicas. The type of an effect is therefore t -> t for which we introduce a type synonym m. Thus, the type of an RDT operation over type t, that expects an argument of type a, and returns a t effect (i.e., m = t -> t) is as follows:

\[
\text{type } \text{a } \text{t oper } = \text{t } \rightarrow \text{a } \rightarrow \text{m}
\]

Note that, by definition, an RDT operation acts on a single instance of an RDT.

As described above, the semantics of an RDT operation follows from the CRDT computation it returns. For instance, a do_deposit operation returns a CRInt.m computation that when invoked on a replica R will add amt to C’s integer state (C.t) on R. This abstract notion of an effect can be concretized as a function that maps a replica state to a new state. As a convention, we use pascal case and uncurried arguments to denote an effect, and snake case and curried arguments for an operation. We also drop the prefix do_. Hence, the effect of (do_deposit amt) is written Deposit(amt). Its semantics is defined by ascribing it the following denotation:

\[
\text{J Deposit(amt)} \text{K} = \lambda s'. s' + \text{amt}
\]

In ascribing the above denotation to Deposit, we assumed that CRInt.t is an int and CRInt.add is basically integer addition. We can similarly ascribe a denotation to Withdraw(amt) effect. Assuming that withdraw generates an effect only when the balance check is satisfied on the origin replica, the Withdraw(amt) effect can be thought of as a function that simply decrements amt from the integer state on all the replicas:

\[
\text{J Withdraw(amt)} \text{K} = \lambda s'. s' - \text{amt}
\]

The function get_balance doesn’t generate an effect, and thus has no need for a denotation.

---

3By convention, we denote such operations by prefixing their name with do_.
As highlighted by the `oprt` type definition above, an RDT operation (do...`) is only ever allowed to operate on a single RDT. However, in general, distributed applications will need to compose multiple RDT operations to perform useful computation. To express such compositions, Q9 additionally supports transactions over replicated objects. In Fig. 2, module Bank composes two BankAccount objects - one (Checking.t) denoting a checking account and the other (Savings.t) savings. It defines a `txn_transfer` transaction (the prefix `txn_` is a naming convention for transactions) that withdraws from the former and deposits to the latter. Provided that the withdraw on the checking account is successful, it returns a composite effectful computation that when applied on a replica serves to perform the transfer on the instance of Checking and Savings on that replica. If we think of a transaction as generating an aggregate effect, that effect is a composition of effects of individual RDT-specific operations that constitute the transaction. For instance, if `txn_transfer` `amt` is thought of as generating a `Transfer(amt)` effect, its denotation is as follows:

\[
[\text{Transfer}(\text{amt})] = \lambda(s'_1, s'_2).([\text{Withdraw}(\text{amt})](s'_1), [\text{Deposit}(\text{amt})](s'_2))
\]

The result of executing `transfer` is therefore the generation of this effect.

Finally, Q9 also allows applications to specify safety properties as boolean functions, via functions whose names are prefixed by `inv_`. These safety properties become the basis for verifying RDT applications.

### 3.1 Explicit Effect Representation

Besides explicating the semantics of RDT operations, the notion of an effect serves a more concrete purpose in Q9. Named effects (e.g., `Withdraw(\text{amt})`) constitute the class of messages that are exchanged between replicas, and act as the pivot for consistency enforcement. More importantly, effects provide a tangible structure to reason about concurrent operations potentially executable on different replicas, an essential requirement for any verification exercise. For these reasons, Q9 translates high-level RDT programs to an intermediate representation (IR) that uses explicit effects. In our running example, the translated version of the BankAccount RDT in the IR includes the following definitions:

```plaintext
type eff = Deposit of int | Withdraw of int

let apply_eff (s':int) (e:eff) = match e with
  | Deposit(a) -> s' + a
  | Withdraw(a) -> s'-a
```

Note that the `eff` type definition and the `apply_eff` function reify the abstract notions of effects and their semantics in the context of the BankAccount application. Operations and transactions can now be defined in terms of these explicit effects:

```plaintext
let deposit (amt:int) = Deposit(amt)
let withdraw (amt:int) = if s>= amt then Withdraw(amt) else error "Insufficient Balance"

let transfer (amt:int) =
  let eff1 = Checking.withdraw amt in
  let eff2 = Savings.deposit amat in
  (eff1, eff2)
```

Observe that, under this formulation, CRDT definitions such as `CRInt` exist only to `transfer` CRDT semantics to their consumers. After compilation to the effect-aware IR, the application’s RDTs themselves become CRDTs; e.g., applying (via `apply_eff`) a collection of effects (i.e., values of `eff` type) in any order results in the same BankAccount state. Thus, the Q9 programming model serves as a way to engineer arbitrary distributed applications with convergent semantics, while its underlying IR directly manipulates effects in ways consistent with CRDT semantics.

The translation to this IR elaborates each operation (i.e., `do`-prefixed function on an RDT) in to a representation that returns the corresponding effect. The effect takes the place of the RDT
module Microblog(Tweet: TWEET)(Userline : USERLINE) (Timeline : TIMELINE) =

struct
    type t = Tweet.t * Userline.t * Timeline.t

let txn_new_tweet (uid: user_id) (str: string) =
    let tweet_id = UUID.new () in
    let e1 = Tweet.new tweet_id str in
    let e2 = Userline.add uid tweet_id in
    let fids = User.get_followers uid in
    let e3 = Timeline.add (Set.map (fun fid -> (fid , tweet_id )) fids ) in
    (e1, e2, e3)
end

Fig. 3. Microblog application’s txn_new_tweet transaction

computation m in the definition of the operation (for e.g., compare withdraw given above to its definition in Fig. 2). Conversely, the interpretation of the effect of an operation at state s’ is obtained by inlining the CRDT computation (m), and applying it on s’ (for e.g., compare the interpretation of Withdraw in apply_eff above to the definition of do_withdraw in Fig. 2). A transaction’s effect is a composition of effects on multiple RDT objects in the same way as the object it manipulates is a composition of multiple RDT objects. For instance, txn_transfer of Bank returns a pair of BankAccount effects for the type it manipulates (Bank.t) is a pair of BankAccount.t objects.

**Microblog.** Fig. 3 shows a more complex transaction from a Twitter-like microblogging application in explicit effect representation. The transaction manipulates objects of three different types: Tweet.t, Userline.t, and Timeline.t. Each object can be thought of as a set of records of a similar type, akin to a table in a relational database. For instance, Tweet.t represents a set of tweets. The transaction first constructs an effect (e1) for adding the new tweet to the collection of tweets, followed by an effect (e2) to add the corresponding tweet id to userline of the author (identified by uid: user_id), and finally an effect (e3) to add the tweet id to the timeline of every follower of the author. It returns a tuple of these effects to be applied on first, second and third components of Microblog.t object, respectively. We shall revisit this transaction in Sec. 7, where we describe the anomalies it exhibits, along with the fixes that Q9 discovers.

4 SYSTEM MODEL

Figure 4 presents a schematic diagram of the system model adopted by Q9. An application’s state is composed of multiple objects (x, y, . . .), each of which is replicated across multiple locations. Each location is called a replica. Each replica maintains an unordered history of an object’s effects known to that replica. For example, the history of of the object x at replica R1 (in Fig. 4) is the set \{e_1^x, e_2^x\}, whereas its history at replica n is the set \{e_1^x, e_n^x\}. The difference in histories is due to **asynchronous replication** which allows effects to be propagated lazily (through the network) and delivered asynchronously. Under a reasonable assumption that the network offers eventual delivery guarantees, all generated effects will
When an operation is invoked, it reads the current state at some with its type (Voldemort 2009], and is consistent with the models used in a number of research prototypes such as Walter [Sovran et al. 2011], Chapar [Lesani et al. 2016], Antidote [Shapiro et al. 2018], and Quelea [Sivaramakrishnan et al. 2015].

The clients of the application interact with the system by invoking operations (e.g., deposit) on objects. The sequence of operations invoked by a particular client on an object is called a session, and the operations found in this sequence are said to be in a session order with one another. Session order also relates the operations within a transaction. For example, in the txn_transfer transaction of Fig. 2, the withdraw operation precedes the deposit operation in session order. At any given instant, an application could be serving multiple concurrent clients/sessions. An operation or a transaction invoked by a client is executed at one of the replicas (e.g., operation foo executes at $R_1$ in Fig. 4). Due to transient system conditions (e.g., network partitions, load balancing etc), it is possible that the operations of the same session get executed at different replicas. For example, operation bar from the same session as foo executes at a different replica. When an operation (e.g., foo) is executed on an object ($x$) at a replica ($R_i$), it is supplied the state of the object ($x$) computed from its history at the replica ($R_i$). In this case, we say that the effects in the history ($e_1^x$ and $e_2^x$) are visible to the operation (foo).

The system described above is quite general insofar as it makes no assumptions on either the timing or order in which effects are generated and propagated. Indeed, the model abstracts many realworld distributed data stores [Lakshman and Malik 2010; Riak 2018; Sivasubramanian 2012; Voldemort 2009], and is consistent with the models used in a number of research prototypes such as Walter [Sovran et al. 2011], Chapar [Lesani et al. 2016], Antidote [Shapiro et al. 2018], and Quelea [Sivaramakrishnan et al. 2015].

5 THE Q9 VERIFICATION ENGINE

5.1 Core Calculus

Fig. 5 shows the syntax of the core calculus ($\lambda_R$) of Q9 that lets us capture the essence of Q9 programs abstractly. The calculus operates on integer and boolean values, sets, and functions. A special type $\text{eff}$ for effects is also present, and it is assumed to be a (non-recursive) sum type of $N$ effect constructors - $W_{i\in[1,N]}$, where each constructor has exactly one argument of a base type ($B$). The syntactic class of expressions includes the usual suspects - lambda abstractions, applications etc., along with set expressions which include an empty set constructor ($\emptyset$), a singleton constructor ($\{x\}$), and a set union constructor ($e_1 \cup e_2$). Applications of effect constructors ($W_{i\in[1,N]}$) are expressions of type $\text{eff}$. A match-with-else expression lets a value be matched against a pattern ($p$), and if the match is successful, evaluates the corresponding expression. Any $\lambda_R$ expression can be annotated with its type ($\tau_i$).

At the top-level, a $\lambda_R$ program ($\pi$) is a parallel composition of functions of type $B \rightarrow \text{eff}$, where $B$ is a non-effect (base) type. Intuitively, a $\lambda_R$ program models a Q9 application operating over a single RDT maintaining a state of type $B$ (e.g., $B$ could be $\text{int}$ in our running BankAccount example). Each function represents an invocation of an operation on the RDT; this construction models the generation of operations by a client in a given session; invocations can proceed in parallel ($\pi || \pi$). When an operation is invoked, it reads the current state at some replica, which is a value of type $B$, and generates a new effect (an $\text{eff}$). Since our system model does not mandate that all replicas

\[
x = \begin{cases} 
\text{Deposit}(10) \mid \text{Withdraw}(5) \mid \text{BankAccount.init}) \\
(\lambda.s.s + 10)((\lambda.s.s - 5)0) \\
5
\end{cases}
\]

be present on all replicas eventually. The state of an object at a replica is a function of the object’s history; it is computed by applying the effects, in no particular order, to the initial object of the RDT. For instance, if $x$ is a BankAccount object, and $e_1^x$ is $\text{Deposit}(10)$, and $e_2^x$ is $\text{Withdraw}(5)$, then a possible state of $x$ at $R_1$ is:

\[
x = \begin{cases} 
\text{Deposit}(10) \mid \text{Withdraw}(5) \mid \text{BankAccount.init}) \\
(\lambda.s.s + 10)((\lambda.s.s - 5)0) \\
5
\end{cases}
\]
witness the effect generated by an operation instantaneously, any operation may witness a *subset* (say, $S$) of effects generated so far by operations that executed previously. The state an operation witnesses is the result of applying the effects in $S$ to an initial state. We write $\iota : B$ to denote the initial state of the RDT being modeled.

Recall that the denotation of an effect is a function that defines what it means to apply that effect (e.g., from Sec. 3: $[\text{Deposit}(\text{amt})] = \lambda s. s + \text{amt}$). The denotation of a set of effects is simply a functional composition of the denotation of its constituents (in the following, $\cup$ denotes a disjoint union, $\epsilon$ stands for an effect, and $\circ$ is the function composition operator as in $f \circ g$):

\[
\begin{align*}
[\emptyset] &= \lambda \lambda x. x \\
[S \cup \{\epsilon\}] &= [S] \circ [\epsilon]
\end{align*}
\]

Thus, the result of applying a set $S$ of effects to $\iota$ is defined by $([S] \iota)$.

Our calculus does not support transactions - each invocation $([\lambda x. e]_{B \rightarrow \text{eff}})$ operates over a single RDT, and produces a single effect. Supporting transactions is however straightforward - invocations would need to be supplied multiple RDTs to operate over, may produce multiple effects, and each of these effects may have internal (session) ordering guarantees that would need to be preserved. We revisit these issues in Sec. 6.

Having defined what it means to apply a set of effects, we can now capture the essence of the operational semantics of a $\lambda_R$ program in a single rule defined over sets of effects, rather than replicas, or other system-level artifacts:

\[
\frac{S \subseteq A \quad (\lambda x. e) ([S] \iota) \downarrow W_j(v)}{\left( A, [\lambda x. e]_{B \rightarrow \text{eff}} || \pi \right) \rightarrow \left( A \cup \{W_j(v)\}, \pi \right)} \quad \text{[E-OPER]}
\]

The rule uses the set of effects generated thus far ($A$) as a proxy for the overall system state, and $[S]$, where $S \subseteq A$ as a proxy for a replica state. Since this covers all possible system configurations and replica states, the semantics is general enough to admit all possible behaviors of a distributed program interacting within asynchronously replicated state.

The E-OPER rule describes a small-step evaluation relation of $\lambda_R$ programs ($\pi$) with the following signature:

\[
(A, \pi) \rightarrow (A', \pi')
\]

As described above, $A$ is the set of effects generated in the system, and $\pi$ denotes the program being reduced. E-OPER is the only computation rule of the small-step relation; the rest are congruence rules that let $s$ morph into a form suitable for E-OPER. The antecedent uses a big-step evaluation relation ($\downarrow$) to interpret expressions. The definition of this relation is standard and elided here.

**Safety.** Recall that Q9 applications define their safety properties as boolean functions. Let $I = [\lambda x. e]_{B \rightarrow \text{bool}}$ denote such an invariant. We say that a certain RDT state $s : B$ satisfies the invariant iff $I(s)$ evaluates to $\text{true}$ as per the big-step semantics of $\lambda_R$. Using $I$, we can informally capture the safety of the application as follows:
The initial state \( \iota \) at every replica satisfies the invariant \( I \), and
At any given instance, if the state at every replica satisfies the invariant \( I \), and if we execute any operation \( f \) at some replica \( R \) generating an effect \( \epsilon \), then applying \( \epsilon \) at any replica \( R' \) results in a state that still satisfies \( I \).

We can transplant this informal characterization into the framework of our calculus thus:

**Definition 5.1.** An execution of a program \( \pi \) is safe with respect to invariant \( I \) if and only if \( \forall A, A', \iota, \text{ if:} \)

- \( I \iota \Downarrow \text{true} \)
- \( \forall S \subseteq A, I(\llbracket S \rrbracket \iota) \Downarrow \text{true} \)
- \( (A, \pi) \longrightarrow (A', \pi') \)

then \( \forall S' \subseteq A', I(\llbracket S' \rrbracket \iota) \Downarrow \text{true} \).

Note that the definition folds the generation of effect \( \epsilon \) into the small-step transition \( (A, \pi) \longrightarrow (A', \pi') \); it is expected that \( A' = A \cup \{ \epsilon \} \). The definition also takes into account the effects that are generated concurrently with \( \epsilon \): these are the effects that are present in \( A \), but not in the set \( S \subseteq A \) witnessed by \( \epsilon \). The effects in \( A - S \) may have been generated before or been concurrent with \( \epsilon \) in real-time, but our asynchronous model doesn’t make such distinctions: if an effect is not visible to \( \epsilon \) (i.e., not present on the replica where \( \epsilon \) was generated), it is a concurrent effect insofar as the operational semantics is concerned.

A subtle yet consequential aspect of the above formal development is that it only loosely constrains the notion of an initial state \( \iota \). Def. 5.1 defines the safety of a concrete execution starting from any state \( \iota \) that satisfies the invariant \( I \). In particular, \( \iota \) is not obligated to denote either an empty state (i.e., a state with no effects), or a state reachable from an empty state; instead, it need only be a state that satisfies the invariant \( I \). This rather broad definition of \( \iota \) nonetheless captures the reality of a database application, which is allowed to start its execution from an arbitrary database state created independently of the application’s interface, as long as the state satisfies the application’s integrity constraints \( I \). For instance, TPC-C’s reference implementation [TPC 2018] ships with a sample workload generator that executes the application against a database state not reachable from an empty state by a TPC-C execution, but one that satisfies all of the TPC-C’s stated integrity constraints. A downside to this relaxed specification of \( \iota \) however is that it may require \( I \) to be strengthened to include all valid database states, failing which the application could be judged unsafe by Def. 5.1. This problem is revisited in Sec. 5.3, and again in Sec. 7 in the context of real applications.

The calculus and operational semantics described here succinctly capture the semantics of concurrent programs under asynchronous state replication without having to concretize low-level aspects of the system model such as message communication, process creation, replica organization, etc. Notwithstanding its succinctness, checking safety in the sense described above by naively exploring a concrete state space of executions is clearly infeasible given the large set of behaviors that are possible. We therefore refine our semantics to leverage symbolic reasoning to enable us to characterize and represent many concrete states at once.

**5.2 Abstract Relations**

In this section, we introduce the formal vocabulary that lets Q9 represent anomalies and consistency specifications abstractly. We say effect \( e_1 \) is visible to another effect \( e_2 \) (vis(\( e_1, e_2 \))) if the operation that generated \( e_2 \) was executed against a state to which \( e_1 \) has already been applied. For instance, in Fig. 1b, the effect (call it \( e_1 \)) of Alice’s Deposit is visible to the effect (call it \( e_2 \)) of Bob’s Withdraw. The visibility relation is irreflexive - effects cannot see themselves; asymmetric - if \( e_1 \) is visible to \( e_2 \), then \( e_1 \) necessarily happened before \( e_2 \), therefore cannot see \( e_2 \); and, non-transitive - if \( e_1 \) is
visible to \( e_2 \), and \( e_2 \) is visible to \( e_3 \), then \( e_1 \) need not necessarily be visible to \( e_3 \); Fig. 1b captures such a scenario. Another important aspect of visibility is that it only ever relates effects on the same object. This follows from the fact that an operation is only allowed to access the state of a single object, hence can only witness the effects of previous operations on that object. Given a relation sameobj that relates effects on the same object, we have:

\[ \forall e_1, e_2. \, \text{vis}(e_1, e_2) \Rightarrow \text{sameobj}(e_1, e_2) \]

The session-order relation relates effects of the same session in the order they are generated. For instance, in Fig. 4, effects \( e_4^k \) and \( e_2^k \) are in a session order (written \( \text{so}(e_4^k, e_2^k) \)). Session order is irreflexive and asymmetric for the same reason as visibility, but it is transitive because the order of operations in a session is a total order.

Having formalized visibility (vis) and session order (so), we can define what it means for an effect \( e_1 \) to happen before an effect \( e_2 \). Clearly, \( e_1 \) has happened before \( e_2 \) if \( \text{vis}(e_1, e_2) \) or \( \text{so}(e_1, e_2) \). Moreover, if we already know that an effect \( e_0 \) has happened before \( e_1 \), and if \( \text{vis}(e_1, e_2) \) or \( \text{so}(e_1, e_2) \), then it follows that \( e_0 \) has happened before \( e_2 \). Observe that these are the only two ways that the asynchronous system model we defined in Sec. 4 lets us define a happens-before ordering of effects. Thus, the happens-before relation (denoted hb) is simply a transitive closure of \( \text{vis} \cup \text{so} \):

\[ \text{hb} = (\text{vis} \cup \text{so})^* \]

Visibility, session order, and happens-before are the major (binary) relations that let us capture dependencies among effects generated by an application. To aid our exposition, we also define various helper (unary) functions - oper and arg, that let us project various attributes of an effect. Recall that the type of effects in Q9’s IR (Sec. 3) is a tagged union of effect arguments. Functions oper and arg project the tag and the argument, respectively, of an effect. For instance, \( \text{oper}(\text{Withdraw}(5)) = \text{Withdraw} \) and \( \text{arg}(\text{Withdraw}(5)) = 5 \).

### 5.3 Symbolic Execution

In this section, we consider a replacement of the concrete evaluation relations (\( \rightarrow, \downarrow \)) with symbolic counterparts (\( \rightarrow_s, \downarrow_s \)) to facilitate symbolic reasoning over states and effects. In the process, we also take into account the specific characteristics of asynchronous state replication so as to bound the state space and expedite the process of anomaly detection.

#### 5.3.1 Intuition

Recall that the semantics of \( \lambda_R \) tracks the state of a program’s execution as \( A \), the set of effects generated thus far, and the state of a replica as a subset \( S \) of \( A \). This construction accounts for any number of replicas, and a liberal network semantics with arbitrary latency and message reordering. (In reality though, there are only a finite number of replicas), and inter-replica latency is usually comparable to (i.e., a small multiple of) replica-local execution time. As a result, in practice, a non-trivial subset \( S_b \) of effects in \( A \) can be expected to be already present on all replicas [Bailis et al. 2012].

The corresponding state \( b = [S_b] \) is therefore a “common prefix” of all the replica states, which is extended at each replica by applying a subset of effects from \( S_c = A - S_b \). The effects in \( S_c \) are called concurrent effects. A concurrent effect is an effect that is not present on (i.e., not applied to the state of) at least one replica. Each replica contains a subset of concurrent effects, which are applied to the common prefix \( b \) to compute the state at that replica. Let \( k \) denote the number of concurrent effects. Lower \( k \) values represent concrete executions where replicas are more-or-less in sync with each other. Conversely, high \( k \) values indicate executions characterized by, e.g., network partitions, process crashes, high network latency etc, that result in divergence between replicas.

Note that representing executions as a pair of a common prefix state \( b \), and a set \( S_c \) of concurrent effects is not any less general than the scheme used by operational semantics, which tracks the set \( A \) of all effects. We can let the former simulate the latter by setting \( b = \iota \) (\( \iota \) is the initial state) and
which is matched against an eff value of type eff.

Here, a set expression evaluates a non-destructible symbolic value \( v \), which is matched against a union pattern \( x \cup y \). Since the execution cannot determine whether \( v \) matches \( x \cup y \) or not (\( v \)}
We now redefine our notion of safety to consider a replica state \( R \). The conclusion of the rule indicates that the symbolic evaluation prevents this by halting the evaluation and returning a fresh symbol with the same type as the match expression. It uses a (meta) function \( H \) for this purpose. The function \( H \) essentially performs memoization; it takes enough arguments to ensure that if the symbolic execution evaluates the same match expression again in the same context, \( \Gamma \), it returns the same symbol \( z \). To avoid cluttering the rule with technicalities, we assume that the type binding for \( z \) is already present in \( \Gamma \), and \( z \) does not occur free in the match expression. The rule \( S-App \) deals

**Example.** Consider the following version of \( \text{apply\_eff} \), which is slightly modified from Sec. 3 to make it conform to the syntax of \( \lambda R \) (for brevity, we use \( D \) for \( \text{Deposit} \), and \( W \) for \( \text{Withdraw} \)):

\[
\lambda(s: \text{int}).\lambda(e: \text{eff}). \begin{cases} 
    D(a) \Rightarrow s + a \\
    W(a) \Rightarrow s - a \text{ else } s
\end{cases}
\]

The result of symbolically evaluating the body of the function is a guarded symbolic value shown below. We name it \( \nu_{\text{app}} \), and parameterize it on the free symbols \( e \) and \( s \):

\[
\nu_{\text{app}}(e, s) = (\text{oper}(e) = D)? (s + \text{arg}(e)) : (\text{oper}(e) = W)? (s - \text{arg}(e)) : s
\]

As mentioned previously, symbolic evaluation explores the state space of the application starting from a symbolic state \( b \) that satisfies the invariant (I), and assuming that the number of concurrent effects \( |S_c| \) never exceeds a fixed value \( k \). We write \( S_c^k \) to explicitly denote a set \( S_c \) that has cardinality \( k \). Let us name the \( k \) concurrent effects as \( E_{i\in[1,k]} \). Thus:

\[
S_c^k = \bigcup_{i=1}^{k} \{E_i\}
\]

A replica state \( s \) is computed by applying a subset of \( S_c^k \) to \( b \). Let us say an operation \( f \) executes against the state \( s \) at replica \( R \) and generates an effect \( e \). From the definition of \( \text{vis} \), it follows that a subset of \( S_c^k \) effects at \( R \) is visible to \( e \); this subset can be constructed thus:

\[
\bigcup_{i=1}^{k} \text{vis}(E_i, e)? \{E_i\} : \emptyset
\]

where \( E_i \in S_c^k \). That is, the effect \( E_i \) is included in the set only if it is visible to \( e \). We call this set a projection of \( S_c^k \) on \( e \), and denote it as \( S_c^k \upharpoonright e \). We define what it means to apply such a set of visible effects by defining its denotation as follows:

\[
\begin{align*}
[\emptyset \upharpoonright e] &= \lambda.s.s \\
[S \uplus \{E_i\} \upharpoonright e] &= [S \upharpoonright e] \circ \lambda.s.\text{vis}(E_i, e)?([E_i]s) : s
\end{align*}
\]

Intuitively, the state at replica \( R \) witnessed is \( [S_c^k \upharpoonright e]b \), where \( b \) is the common prefix state.

Having defined what it means to apply a projection, we can now define a symbolic equivalent of the concrete small-step evaluation rule. The rule represents the global state as a tuple of the common prefix and concurrent effect set \( (b, S_c^k) \), instead of the set of all effects (A):

\[
\frac{(\lambda x. e) ([S_c^k \upharpoonright e] b) \downarrow e}{(b, S_c^k), \left[\lambda x. e\right]_{B\rightarrow eff} \parallel \pi} \xhookrightarrow{\pi} \pi \quad [S-Oper]
\]

The conclusion of the rule indicates that the \( \lambda R \) program \( \left[\lambda x. e\right]_{B\rightarrow eff} \parallel \pi \) symbolically reduces to \( \pi \) under the state \( (b, S_c^k) \) while generating an effect \( e \). The antecedent requires that the effect \( e \) be the result of symbolically executing \( \lambda x. e \) against a state that applies a subset of effects visible to \( e \) to \( b \). We now redefine our notion of safety to consider \( k \)-bounded symbolic execution:
Definition 5.2 (k-safety). A symbolic execution of a program \( \pi \) bounded with respect to invariant \( I \) if \( \forall b, k, S^k, \epsilon, e_f \) s.t:

- \( I[[S^k] \triangleright e_f] b \downarrow \text{true} \)
- \( \forall \pi, \pi', ((b, S^k), \pi) \xrightarrow{\epsilon} \pi' \)
  then \( I[[S^k \cup \{ \epsilon \}] \triangleright e_f] b \downarrow \text{true} \)

In the above definition, \( \epsilon \) is the effect generated by the small-step reduction of the program \( \pi \), whereas \( e_f \) is an effect generated by some operation \( f \) witnessing the state. The first premise asserts that \( f \) initially sees an invariant-satisfying state regardless of what subset of concurrent effects it witnesses. In the context of a replicated state system, it means that all replicas initially satisfy the invariant. Invariant satisfiability is defined by asserting that the symbolic value resulting from symbolically evaluating the invariant function is equal to \( \text{true} \). The second premise (interpreted using the \( \text{S-Oper} \) rule) states that the effect \( \epsilon \) is the result of symbolically evaluating an operation in \( \pi \) against a state containing a subset of concurrent effects. Concretely, it means that \( \epsilon \) is an effect generated by executing an operation in \( \pi \) at some replica. Under these premises, proving k-safety of \( \pi \) requires proving that \( f \) continues to see an invariant satisfying state even when the set \( S^k \) of concurrent effects is extended with \( \epsilon \). That is, even if \( f \) is executed on a replica that includes the newly generated effect \( \epsilon \), it still sees an invariant satisfying state.

Note that, since k-bounded verification only explores a limited state space, k-safety does not guarantee the unconditional safety of \( \lambda_R \) programs, but it does guarantee that any counterexample to safety it discovers is a real counterexample, assuming the invariant \( I \) is a complete specification of valid program states, i.e., any assignment to the symbolic pre-state \( b \) that satisfies \( I \) is a valid assignment\(^4\). We call the counterexample discovered by symbolic execution of program \( \pi \) as a witness to the k-unsafety of \( \pi \), and a counterexample discovered by the concrete execution as a witness to the unsafety of \( \pi \). The soundness of bounded verification can now be stated thus:

**Theorem 5.3.** If a \( \lambda_R \) program is k-unsafe with witness \( \omega \), then it unsafe with witness \( \omega \), provided that its invariant \( I \) is a complete specification of valid program states.

The proof of the theorem follows from the fact that the symbolic execution computes an underapproximation of the set of behaviors a \( \lambda_R \) program can exhibit. Thus, any execution captured by the symbolic encoding of the program is a valid program execution, including an unsafe execution.

The soundness guarantee of Theorem 5.3 is conditional to the invariant \( I \) being a complete specification of valid program states. If on the other hand \( I \) is only a partial specification, then symbolic execution may capture executions that do not manifest concretely, thereby leading to false k-safety violations. However, as explained in Sec. 5.1, completeness of \( I \) is a reasonable assumption to make in the context of database programs, which are often executed against databases populated independently of such programs. In this setting, the only valid assumptions about the database state are the stated integrity constraints (I).

5.3.3 Example. Let us say we would like to verify the 3-safety of a \text{withdraw}(amt) operation, i.e., safety of \text{withdraw}(amt) assuming three concurrent effects \( S^3 = \{E_1, E_2, E_3\} \).

As per Def. 5.2 the invariant can be assumed to be valid in any pre-state. That is, for some effect \( e_f \), assume \( \text{inv\_non\_neg\_bal}([S^k] \triangleright e_f] b) \downarrow \text{true} \). The term \([S^k] \triangleright e_f\] denotes the application of effects visible to \( e_f \) via BankAccount’s apply\_eff. Recall (from the previous example) the symbolic value \( v_{\text{app}}(e, s) \), which is the result of symbolically executing apply\_eff on a symbolic effect \( e \)

\(^4\)This assumption is already captured by Def. 5.1, which defines a valid concrete execution as one starting from any invariant satisfying state. To avoid potential sources of confusion, we explicitly qualify our soundness guarantees with this assumption wherever required.
and a symbolic state \( s \). Since \( [S^2_c \triangleright \epsilon_f] \) applies an effect \( E_i \in S^2_c \) only if \( \text{vis}(E_i, \epsilon_f) \), it reduces to the following symbolic value (let bindings are used for the sake of clarity):

\[
\begin{align*}
\text{let } s_1 &= \text{vis}(E_1, \epsilon_f) \land v_{\text{app}}(E_1, b) : b \text{ in} \\
\text{let } s_2 &= \text{vis}(E_2, \epsilon_f) \land v_{\text{app}}(E_2, s_1) : s_1 \text{ in} \\
\text{vis}(E_3, \epsilon_f) \land v_{\text{app}}(E_3, s_2) : s_2
\end{align*}
\]

Let us name the above symbolic value \( v_{s}^{\text{pre}}(\epsilon_f) \). The parameterization on \( \epsilon_f \) underscores that \( \epsilon_f \) could be any effect witnessing the pre-state. Since the invariant is valid initially, \( v_{s}^{\text{pre}}(\epsilon_f) \geq 0 \).

Next, Def. 5.2 lets us assert the conditions under which the program \( \pi \) generates the effect \( \epsilon \). In other words, it lets us capture the local safety of \( \epsilon \). Here, \( \pi \) is simply the \texttt{withdraw(amt)} operation, and \( \epsilon \) is a \texttt{withdraw} effect. The operation generates the effect only if the balance it reads is not less than \( \text{amt} \). The symbolic execution captures this condition as a path constraint (i.e., a logical formula whose satisfiability determines the feasibility of the current program path), which is then allowed to be asserted as a premise of \( k \)-safety. The balance that \texttt{withdraw(amt)} reads is \( [S^2_c \triangleright \epsilon] b \), which expands to \( v_{s}^{\text{pre}}(\epsilon) \). Thus, the premise that \texttt{withdraw(amt)} is locally-safe translates to the assertion \( v_{s}^{\text{pre}}(\epsilon) \geq \text{amt} \).

Having captured the two premises of Def. 5.2 as constraints on symbolic values, we are now required to prove that if we include \( e \) in the set of concurrent effects, the invariant still evaluates to \texttt{true}, i.e., \( \text{inv\_non\_neg\_bal}([S^2_c \cup \{e\}] \triangleright e] b) \downarrow \texttt{true} \). The expression \( [S^2_c \cup \{e\}] \triangleright e] b \) essentially applies \( e \) to the result of \( [S^2_c \triangleright \epsilon_f] \) if and only if \( \text{vis}(\epsilon, \epsilon_f) \). Recalling that the result of applying a symbolic BankAccount effect \( e \) on a symbolic state \( s \) is \( v_{\text{app}}(e, s) \), and that \( [S^2_c \triangleright \epsilon_f] = v_{s}^{\text{pre}}(\epsilon_f) \), we deduce that the result of \( [S^2_c \cup \{e\}] \triangleright e] b \) is the following symbolic value:

\[
\text{vis}(\epsilon, \epsilon_f) \And v_{\text{app}}(e, v_{s}^{\text{pre}}(\epsilon_f)) : v_{s}^{\text{pre}}(\epsilon_f)
\]

Let us call the above value \( v_{s}^{\text{post}}(\epsilon_f) \). The proof obligation generated by Def. 5.2 is:

\[
v_{s}^{\text{pre}}(\epsilon_f) \geq 0 \And v_{s}^{\text{pre}}(\epsilon) \geq \text{amt} \Rightarrow v_{s}^{\text{post}}(\epsilon_f) \geq 0
\]

The validity of the above implication is equivalent to the unsatisfiability of the following conjunction:

\[
v_{s}^{\text{pre}}(\epsilon_f) \geq 0 \And v_{s}^{\text{pre}}(\epsilon) \geq \text{amt} \And \neg(v_{s}^{\text{post}}(\epsilon_f) \geq 0)
\]

An SMT solver, such as Z3 determines the conjunction to be satisfiable\(^5\), thus proving that \texttt{withdraw} is 3-unsafe. The counterexample that Z3 returns involves assigning \( \text{amt} = 1 \), and making \( E_1 \) a \texttt{Deposit(1)} effect that is visible to the current effect \( \epsilon = \text{Withdraw(1)} \), but not making it visible to the (reference) effect \( \epsilon_f \). The counterexample is visualized in Fig. 7a, and is an abstract representation of the concrete anomaly described in Fig. 1b. Fixing the anomaly (as described in the next section), and rechecking the satisfiability lets us discover another counterexample, visualized in Fig. 7b. This counterexample is an abstraction of the concurrent withdraws anomaly of Fig. 1a. Fixing the second anomaly is enough to show that the constraints are unsatisfiable, thus \texttt{withdraw} is 3-safe.

\(^5\)In general, the efficacy of bounded verification in Q9 depends on the ability of the solver to reason about the theories required to encode the path constraints of the program and the invariant. Fortunately, there exist decidable theories, such as linear arithmetic, that are useful in practice.
5.4 Automated Repair

Once a counterexample demonstrating an anomaly has been discovered, Q9 helps to automatically repair the application by appropriately strengthening the consistency of the offending operation. We equip Q9 with a set of consistency levels, each designed to exempt a few classes of anomalous executions. Realization of these levels incurs a performance penalty in proportion to their strength (stronger consistency levels prohibit more anomalies and incur heavier penalty). The challenge is to determine the weakest consistency model that prohibits the anomaly exhibited by the counterexample. Fortunately, this step can be automated by observing that consistency levels can be captured in the same abstract language as the counterexamples that the symbolic execution discovers.

Consider the counterexample execution depicted in Fig 7a. We can formally express behaviors that admit this execution as the following counterexample (call it $\varphi_{cex}$):

$$
\text{oper}(E_1) = \text{Deposit} \land \text{oper}(\epsilon) = \text{Withdraw} \land \text{sameobj}(E_1, \epsilon) \land \text{sameobj}(\epsilon, \epsilon_f) \\
\land \text{vis}(E_1, \epsilon) \land \text{vis}(\epsilon, \epsilon_f) \land \neg\text{vis}(E_1, \epsilon_f) \land \neg\text{vis}(\epsilon_f, E_1)
$$

Given this characterization, we are interested in finding the weakest consistency assignment to withdraw (the operation that generated $\epsilon$) that would prevent the counterexample. Here, we are aided by the fact that consistency levels can be specified in terms of the anomalies they prohibit. For instance, consider causal write, a consistency level that ensures a write is applied at a replica only after all the causally preceding writes have been applied. Thus, if $\epsilon$ is a causal write, then any anomalous execution involving three effects $a$, $\epsilon$, and $b$, where (i). $a$ causally precedes $\epsilon$, and (ii). $\epsilon$ is visible to $b$, but (iii). $a$ is not visible to $b$, is prohibited. Causal precedence is effectively captured by the happens-before relation (hb), which is a composition of $\text{vis}$ and so (recall: $\text{hb} = (\text{vis} \cup \text{so})^\ast$). Thus, if $\epsilon$ is a causal write, then the following must be true:

$$
\forall a, b. \neg(\text{hb}(a, \epsilon) \land \text{vis}(\epsilon, b) \land \neg\text{vis}(a, b))
$$

Or, equivalently:

$$
\forall a, b. \text{hb}(a, \epsilon) \land \text{vis}(\epsilon, b) \Rightarrow \text{vis}(a, b)
$$

If we name the above proposition $\varphi_{cw}$, and can prove that $\varphi_{cex} \land \varphi_{cw}$ is UNSAT, then it would be sufficient to make withdraw a causal write to prevent this anomaly. An off-the-shelf SMT solver like Z3 confirms that this is indeed the case.\footnote{We adopt an approach similar to [Itzhaky et al. 2014] to encode an overapproximation of $\text{hb}$, a transitive closure relation, in first-order logic.} Similar reasoning can be applied to second counterexample execution in 7b to determine that withdraw also needs to be totally-ordered w.r.t other withdraws, i.e., it must be a Total Write.

The state space of consistency models found in the literature and implemented on various systems can be characterized in terms of a finite partially-ordered lattice [Viotti and Vukolic 2015], where the partial order denotes the relative strength of the models under consideration. It is therefore possible to determine the consistency level of an operation by systematically traversing the lattice and checking whether each consistency model is sufficient to prevent the counterexample. Among the consistency models that are at the same level of the lattice, the order of traversal can be heuristic, perhaps based on their relative run-time costs on a specific system. Systematic traversal of the consistency lattice is indeed the search strategy adopted by Q9.

The consistency models that Q9 considers in its search are shown in Table 1. The relation $\text{txn}$ relates a transaction (its name $\tau$) to the set of its constituent effects. Note that Atomicity is a property of a transaction. It is in fact a transaction’s baseline consistency level in Q9. A transaction’s consistency and isolation properties can also be strengthened in various ways, just like an operation’s. One such way is to obtain a (conceptual) write lock on an object each time a write is performed, releasing all locks only when the transaction commits. The resultant consistency
level, called Parallel Snapshot Isolation (PSI) [Sovran et al. 2011], is specified in Table 1. As shown, it results in the writes outside a PSI transaction τ being totally ordered with τ’s writes on the same objects. Other consistency levels found in the literature, e.g., Session Guarantees [Terry et al. 1994], can be specified in a similar way.

### 6 TRANSACTIONS

In our exposition thus far, we have focused on operations that generate a single effect. Q9 programming framework also supports transactions that operate on multiple RDT objects, and generate multiple effects. To deal with such transactions, symbolic execution and verification require a few extensions, which we describe below.

First, the concept of state has to be generalized from a single object to a collection of objects. Thus, the common prefix state is δ instead of b, and bi denotes the common prefix state for the i’th object. The denotation of an effect e, i.e., [e] is now a function that operates on a sequence of objects, but only updates (i.e., computes an updated value for) the object for which e was generated. Building on these refined notions of state and effect denotation, we now generalize the S-Oper symbolic execution rule to deal with transactions. In λR, we formalize transactions simply as functions that accept multiple arguments, and generate a set σ of effects. The generalized S-Oper is shown below:

$$
S \subseteq S^k, \quad \lambda x. e \begin{cases} [S] \delta \downarrow \sigma \\ \forall (e' \in S), \lambda x. e' \in \sigma, \text{sameobj}(e', e) \Rightarrow \text{vis}(e', e) \\ \end{cases} (\delta, S^k), [\lambda x. e]_{B\rightarrow eff} \parallel \pi \xrightarrow{\sigma} \pi \quad \text{[S-Oper]}
$$

The rule reflects the system model of Q9, where a transaction, is executed at a single replica atomically, leading to all of its effects (e) witnessing the same state that includes a set S of concurrent effects. The quantified assertion in the premise captures the constraint that all effects e ∈ σ witness the same set of concurrent effects that are on the same object as e.

Safety is defined w.r.t an invariant function I, which relates multiple objects, i.e., it is a boolean function on multiple objects. The generalized k-safety definition that extends k-safety to transactions is defined thus:

---

**Table 1. Consistency Models**

<table>
<thead>
<tr>
<th>Model Name</th>
<th>Specification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Causal Write (CW)</td>
<td>∀a, b. hb(a, e) ∧ vis(e, b) ⇒ vis(a, b)</td>
<td>A write e is applied only after all the causally preceding writes (a) are applied. [Bouajjani et al. 2017]</td>
</tr>
<tr>
<td>Monotonic Write (MW)</td>
<td>∀a, b. so(a, e) ∧ vis(e, b) ⇒ vis(a, b)</td>
<td>A write e is applied only after all the previous writes (a) from the session are applied [Terry et al. 1994].</td>
</tr>
<tr>
<td>Total-Order Write (TW)</td>
<td>∀a. oper(a) = oper(e) ∧ a ≠ e ⇒ vis(a, e) ∨ vis(e, a)</td>
<td>All writes of the same operation as e are applied in the same order everywhere.</td>
</tr>
<tr>
<td>SC Write (SC)</td>
<td>∀a, b, c. txn(τ, {a, b}) ∧ ¬txn(τ, {c}) ∧ vis(a, c) ∧ sameobj(b, c) ⇒ vis(b, c)</td>
<td>Writes from a transaction τ are applied atomically</td>
</tr>
<tr>
<td>Atomicity (ATOM)</td>
<td>ATOM ∧ ∀a, b, c. txn(τ, {a, b}) ∧ ¬txn(τ, {c}) ∧ vis(a, c) ∧ sameobj(b, c) ⇒ vis(b, c)</td>
<td>Writes from a transaction τ are made SC, and applied atomically.</td>
</tr>
<tr>
<td>Parallel Snapshot Isolation (PSI)</td>
<td>ATOM ∧ ∀a, b. txn(τ, {a}) ∧ ¬txn(τ, {b}) ∧ vis(a, b) ∨ vis(b, a)</td>
<td></td>
</tr>
</tbody>
</table>
Definition 6.1 (\(k\)-safety). A symbolic execution of a program \(\pi\) bounded by \(k\) concurrent effects is \(k\)-safe with respect to invariant \(I\) if \(\forall b, S, S^k, \epsilon, \sigma\) s.t:

- \(S \subseteq S^k\)
- \(I([S] b) \Downarrow \text{true}\)
- \(\forall \pi, \pi', ((b, S^k), \pi) \xrightarrow{} \pi'\)
  then \(I(([S] \circ [\sigma]) b) \Downarrow \text{true}\)

The generalized definition is similar to the previous definition in the sense that it allows us to assume invariant on any subset \(S\) of the pre-state, and asks us to prove the invariant in the post state when \(S\) is extended with all the effects (\(\sigma\)) generated by the transaction, thus guaranteeing atomicity. Note that \([S] \circ [\sigma]\) denotes the functional composition of denotations of the sets \(S\) and \(\sigma\). With the updated \(k\)-safety definition, anomaly detection and repair can work just as described in the previous section.

7 IMPLEMENTATION AND EVALUATION

The Q9 programming framework is implemented in OCaml as a collection of data and module type definitions, and modules that implement various CRDT semantics, such as counters, sets, maps, and boolean flags [Shapiro et al. 2011c]. The Q9 symbolic execution engine is implemented as a compiler pass that follows typechecking in the OCaml 4.03 compiler\(^7\). Its first component is a translator that translates high-level RDT programs with implicit effects to their intermediate representation with explicit effects in preparation for analysis and verification. The second component performs bounded verification, given the \(k\)-bound as an input, and works in a tight loop with an SMT solver. The third component handles consistency repair. Verification progresses one operation at a time, followed by one transaction at a time. Each operation/transaction is verified for safety against its current consistency setting, starting with eventual consistency; this baseline reflects the system model described in Sec. 4. If verification fails, the verifier obtains a counterexample from the solver, computes its abstract representation, and passes it on to the repair engine, which then traverses a lattice of consistency models as described in Sec. 5.4, using the solver to check if a particular model is sufficient to preempt the counterexample. It returns the weakest such model to the verifier, which repeats verification with the new setting. This process continues until the verification of the operation/transaction succeeds, or the top of the consistency lattice has been reached, and no consistency setting was found to be adequate to guarantee safety\(^8\).

The main component of the verifier is a symbolic execution engine that executes the body of an operation/transaction against symbolic inputs. The crux of the symbolic execution algorithm is as described in Sec. 5.3, but the engine also includes a number of optimizations aimed at rewriting symbolic values so as to keep their size roughly proportional to the length of the program traversed. An example of such a rewrite rule is shown below:

\[(v_1 \land v_2 : v_3)? v_4 : v_5 \rightarrow ((v_1 \land v_2) \lor (\neg v_1 \land v_3))? v_4 : v_5\]

Symbolic execution generates verification conditions (VCs) based on the \(k\)-safety definition (Def. 5.2). A VC-Encode component encodes these VCs as satisfiability queries in Z3, after asserting the required axioms on special relations such as \(\text{vis}\) and \(\sigma\) (e.g., \(\text{so}\) is transitive, \(\text{hb}\) is irreflexive etc). If the query is satisfiable, then a model is obtained and passed on to the verifier, which then uses it for consistency repair as described above.

\(^7\)https://github.com/tycon/q9

\(^8\)This might happen if the consistency lattice given to the analysis is not strong enough. If the lattice describes the consistency levels of a data store, then the failure means that the safety of the program cannot be guaranteed on that store.
Table 2. A sample of the anomalies found and fixes discovered by Q9

<table>
<thead>
<tr>
<th>Oper/Txn</th>
<th>Violated Inv.</th>
<th>Anomalies</th>
<th>Fix</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>eBanking</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>withdraw</td>
<td>bal ≥ 0</td>
<td>i. Deposits &amp; Withdraws not being applied in causal order.</td>
<td>CW +</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ii. Concurrent Withdraws independently succeed resulting in a negative balance.</td>
<td>TW</td>
</tr>
<tr>
<td><strong>txn_transfer</strong></td>
<td></td>
<td></td>
<td>PSI</td>
</tr>
<tr>
<td></td>
<td>bal ≥ 0</td>
<td>Concurrent txn_transfers independently succeed resulting in a negative balance.</td>
<td></td>
</tr>
<tr>
<td><strong>Twissandra</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tx_new_tweet</td>
<td>Timeline → Tweet</td>
<td>Write to Timeline is applied before the previous write to Tweet</td>
<td>MW</td>
</tr>
<tr>
<td>add_username</td>
<td>Uniqueness of usernames</td>
<td>Concurrent checks for the uniqueness of a username succeed independently, resulting in duplicates.</td>
<td>TW</td>
</tr>
<tr>
<td><strong>RUBiS</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tx_new_item</td>
<td>WalletBids → Bids</td>
<td>Write to WalletBids is applied before the previous write to Bids</td>
<td>MW</td>
</tr>
<tr>
<td><strong>eCart</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>checkout</td>
<td>∀(a ∈ stock). qty(a) ≥ 0</td>
<td>Concurrent checkouts of same items succeed independently resulting in negative stock.</td>
<td>TW</td>
</tr>
<tr>
<td><strong>TPC-C</strong></td>
<td></td>
<td></td>
<td>PSI</td>
</tr>
<tr>
<td>tx_new_order</td>
<td>Per-district order ids are unique and sequential</td>
<td>Concurrent tx new_order transactions read the same next_oid from a District record, and insert new orders with this id, resulting in orders with duplicate ids.</td>
<td></td>
</tr>
<tr>
<td><strong>TPC-E</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>complete_trade</td>
<td>Broker → COUNT(Trade)</td>
<td>Update to Trade is applied before the previous insert to Trade.</td>
<td>CW</td>
</tr>
<tr>
<td>tx_new_result</td>
<td>Broker ref → COUNT(Trade)</td>
<td>Concurrent trade_result txns complete the same trade, and independently increment Broker′ num_trades.</td>
<td>PSI</td>
</tr>
</tbody>
</table>

7.1 Verification Experiments

To test the effectiveness of Q9 in detecting and fixing replication anomalies, we ported a range of applications, including several standard database benchmarks, to the Q9 programming model, and verified them under various values of bound (k). The applications are briefly described below:

- **eBanking**: A banking application that extends the running example with additional functionality.
- **Twissandra**: A Twitter-like microblogging application based on a popular Cassandra application with the same name [Twissandra 2014].
- **RUBiS**: Rice University Bidding System [RUBiS 2014] - an eBay-like auction site.
- **eCart**: An eCommerce application that lets users jointly control a shopping cart.
- **TPC-C**: A database benchmark that emulates a warehouse application.
- **TPC-E**: A database benchmark that emulates a brokerage application.
Both TPC-C and TPC-E, which were originally written for testing relational databases [TPC 2018], were reimplemented to leverage CRDTs to make them amenable for execution in a distributed environment. Specifically, each TPC-C/TPC-E table translates into an RDT. For instance, TPC-C’s Order table is implemented by an Order.t RDT, which internally uses a set CRDT to manage its contents. Every INSERT, UPDATE and DELETE operation on the table is implemented by a dedicated operation on the RDT. For example, a SQL INSERT operation that inserts an order record is implemented by an operation do_add_order that adds the order information to the set. The operation is eventually translated into an AddOrder effect, and symbolic reasoning is performed on such effect representations.

Each application described above defines one or more invariants that capture its salient safety properties. During verification, we found anomalies that violate a subset of the invariants for each application. Table 2 presents an interesting sample of the violations we found. These anomalies can be broadly classified into the following categories:

- **(In)equality invariants**: Invariants on integers involving equalities and inequalities. An example is \( \text{bal} \geq 0 \) found in the eBanking application.
- **Uniqueness invariants**: Invariants that require a value of a particular type to be unique. An example is TPC-C’s requirement that every order under a district to have a unique identifier. Another example is the requirement that user names be unique in Twissandra.
- **One-to-one referential integrity**: Invariants that require references between objects to be valid. That is, if an object of type \( A \) refers to another object of type \( B \), then the corresponding \( B \) object must be present whenever an \( A \) object is present. We denote such one-to-one referential integrity relations as \( A \underset{ref}{\rightarrow} B \), whenever \( A \) and \( B \) are both objects. For example, Twissandra requires references from users’ timelines to tweets to be valid.
- **One-to-many referential integrity**: Whenever an object of type \( A \) refers to a certain property \( (f) \) of (some) objects of type \( B \), then the property must hold of the corresponding \( B \) objects whenever an \( A \) object is present. We denote such a relation as \( A \underset{ref}{\rightarrow} f(B) \), and call it one-to-many referential integrity provided \( A \) and \( B \) are both object types, and \( f \) is a function from \( B \) to some base type. An example of one-to-many referential integrity is the Order \( \underset{ref}{\rightarrow} \text{COUNT(OrderLine)} \) invariant in TPC-C, which requires an order’s \( \text{o.ol_count} \) field to accurately reflect the number of OrderLine records referring back to the order. Another example is TPC-C’s Warehouse \( \underset{ref}{\rightarrow} \text{SUM(History)} \) that requires a warehouse’s year-to-date balance to agree with its ledger stored in the History table. Similar constraints are also present in TPC-E.

Table 2 lists various operations and transactions that violate the invariants of the kind described above. For each violation, the table briefly describes the anomaly that was discovered, and also lists the consistency level suggested to preempt the anomaly (c.f., Table 1 for a description of consistency levels). As an example of the kind of repair Q9 was able to perform, consider TPC-C’s txn_new_order transaction, which adds a new Order record with an id \( \text{Order.o_id} \) equal to the sequence number of the next order for the corresponding district \( \text{District.next_o_id} \). The transaction also increments the district’s order sequence number. During the verification of txn_new_order, Q9 was able to discover an anomaly that violates TPC-C’s safety requirement that every order must have a unique id. The anomaly consists of two concurrent txn_new_order transactions reading the same id of the district’s next order \( \text{District.next_o_id} \), and inserting duplicate Order records with that id. Subsequent to the discovery of anomaly, Q9 was also able to use the
counterexample to reason that if txn_new_order is executed under Parallel Snapshot Isolation (PSI) consistency model, then the anomaly can be preempted. While Q9 found a violation of uniqueness invariant in TPC-C, through a similar reasoning it found a violation of one-to-many referential integrity invariant (Broker \( \xrightarrow{\text{ref}} \text{COUNT} \) Trade) in TPC-E. The fix, again, is to strengthen the transaction’s consistency level to PSI.

Q9 was also able to perform the reasoning in the opposite direction, i.e., it was able to discover that certain transactions need not be atomic when we made atomicity optional for transactions. Instead, Q9 suggested weaker alternatives to atomicity that are nonetheless safe in that context. For instance, consider txn_new_tweet transaction in Twissandra, which adds a new tweet to the Tweet table, and then adds the corresponding tweet id to a subset of objects in the Timeline table. Without atomicity (ATOM), the transaction (temporarily) violates the referential integrity between timelines and tweets if the latter write to Timeline is applied before the former write to Tweet, and the intermediate state becomes visible to an operation. While atomicity is sufficient to restore the safety, it is however not necessary; Q9 discovers that the anomaly can be preempted by executing the transaction under Monotonic Writes consistency model, which is weaker than atomicity, and is cheaper on some systems [Terry et al. 1994, 1995]. Similar deductions were made for txn_bid_for_item transaction in RUBiS.

Table 3 shows various statistics quantifying the cost and efficacy of bounded verification. The table demonstrates Q9 was able to successfully find a number of anomalies for each application. The fact that anomalies were found in TPC-C and TPC-E might be surprising, considering that these benchmarks were well-studied. Clearly, as our experiments demonstrate, migration of concurrent applications to replicated environments is error-prone without tool support of the kind that Q9 provides.

The main takeaway from Table 3 is that all the anomalies were found within a small \( k \) bound, the maximum being 10 for TPC-C and TPC-E. The time that Q9 took to discover an anomaly is also reasonable, with the worst case being around 2 minutes for an anomaly in TPC-E. To test the limits of bounded verification through symbolic execution, we ran Q9 overnight (6-8 hours) on select (typically the most complex) transactions from each application and noted the maximum \( k \) for which it was able to verify the transaction (for TPC-C and TPC-E, we were able to verify all transactions). The maximum \( k \) thus found is listed against each application in Table 3. As shown, we were able to verify \( k \)-safety of some applications to \( k \) values that are significantly higher than the \( k \) values at which anomalies were discovered. Taken together, these statistics vindicate Q9’s approach of using symbolic execution-driven bounded verification to discover anomalies in real distributed applications.
7.2 Validation Experiments

Since Q9 does bounded verification, we validate the consistency assignments discovered by Q9 by testing the applications on a distributed database. Our goal is two-fold. First, we would like to check if the consistency assignments discovered by Q9 through bounded verification are indeed sufficient to avert anomalies in the general case. And second, we would like to ascertain that any weaker consistency assignment invariably leads to the anomalies discovered by Q9 during verification; i.e., there are no false positives.

Our experimental setup consists of a distributed database equipped with RDT operations, with support for various consistency levels for operations and transactions. The distributed database itself is implemented as a shim layer on top of Cassandra [Lakshman and Malik 2010] in the same vein as [Bailis et al. 2013; Sivaramakrishnan et al. 2015]. We instantiated 2 replicas within the same data center with a inter-replica latency of 5ms, and 16 clients in total performing transactions. In order to tease out the anomalies that arise due to the asynchronous nature of the distributed database, we induced the shim layer to drop 50% of the effects transmitted over the network between the replicas. The replicas perform retransmission of the dropped effects until all the effects are received everywhere. Consequently, every replica receives every effect eventually, but the effects may be applied out of order.

We evaluated the TPC-C benchmark, where each client simulates the workflow for purchasing by performing a series of NEW-ORDER, PAYMENT and DELIVERY transactions. We call one such sequence of three transactions as a purchase. First, we ran the TPC-C workload with Q9 recommended consistency levels. We observed no anomalies for 1000 purchases per client. On the other hand, running the NEW-ORDER transaction at a level weaker than PSI consistency level (i.e., with only atomicity (ATOM)) led to anomalies; there were multiple orders with the same id, thus violating the safety requirement of TPC-C. The results demonstrate that Q9 is effective at finding appropriate consistency configuration: no anomalies were observed at the recommended consistency configuration, while any weaker configuration leads to manifestation of anomalies.

8 RELATED WORK

CRDTs [Shapiro et al. 2011c] define abstract data types such as counters, sets, etc., with commutative operations such that the state of the data type always converges. This property makes them especially attractive as a basis for dealing with replication in highly-available distributed systems. However, reasoning over CRDTs can be difficult, and the nuances of their implementations relate poorly to understanding if and how they might preserve high-level application invariants.

[Burckhardt et al. 2015] presents an operational model of a replicated data store that is based on the abstract system model presented in [Burckhardt et al. 2014]. As with our system model, coordination among replicas involves transmitting operations on replicated objects that are performed locally on each replica. The verification strategy given in [Burckhardt et al. 2014] is based on a replication-aware simulation argument that does not have an obvious automation pathway. [Sivaramakrishnan et al. 2015] describes an automated verification strategy for replicated data types that requires programmers to write low-level (axiomatic) contracts that capture desired consistency properties in terms of visibility and happens-before relations, but which does not consider how these contracts relate to higher-level application-specific invariants. [Gotsman et al. 2016] addresses this issue by developing a rely-guarantee methodology and proof rule that can establish whether a particular choice of consistency guarantees for various operations on a replicated database is enough to ensure preservation of a given data integrity invariant. [Kaki et al. 2017] develops a similar framework for reasoning about weak isolation [Bailis et al. 2014, 2015], a variant of weak consistency.
et al. 2016] is a tool for verifying the correctness of distributed protocols as sophisticated as Paxos [Padon et al. 2017b].

These efforts require support from developers who must either write detailed contracts [Sivaramakrishnan et al. 2015], define deep specifications within a mechanized theorem prover [Lesani et al. 2016; Wilcox et al. 2015], state and prove various kinds of local and global assertions within the context of a program logic [Gotsman et al. 2016], and/or fix counterexamples that prevent inductive generalization [Padon et al. 2016]. Given such input, these approaches are capable of addressing important verification challenges in realistic distributed systems. The work presented here contrasts significantly from these other efforts because it demands no additional effort from the programmer other than a specification of a safety property. While Q9 cannot provide the same level of guarantees that full verification can, empirical evidence suggests that it nonetheless provides a high degree of utility, effectively serving as a principled anomaly detection tool for geo-replicated distributed programs, with minimal overhead demanded of the developer.

Context-bounded model-checking [Musuvathi and Qadeer 2007; Musuvathi et al. 2008], a bounded verification technique comparable to ours but for shared memory, critically assumes SC semantics, making it ineffective in discovering any of the anomalies discussed in Table 2.

Some of the challenges faced in reasoning about replicated data types explored here are reminiscent of issues that arise in reasoning about weak memory systems. However, the differences between weak memory and weak consistency are sufficiently significant that reasoning techniques possible in the former are difficult to transparently migrate to the latter. In particular, weak memory models usually guarantee coherence (total ordering) of writes to a single location, a property not feasible under weak consistency since it requires global coordination [Bailis et al. 2014]; reads in a single thread witness a monotonically progressing state under weak memory, but the same is not guaranteed under weak consistency; and, formalizations of the former reason over memory operations (reads and writes), whereas our formalization reasons at the level of abstract atomic effects (e.g., Deposit). This generalization allows us to scale the reasoning beyond litmus tests [Alglave et al. 2010; Bornholt and Torlak 2017] to real programs. Finally, repair mechanisms for weak memory are defined in terms of fences - low-level architecture-dependent artifacts that "flush" the local state. In contrast, the repair mechanisms for weak consistency are defined in terms of fine-grained high-level consistency models (Table 1) expressed in terms of causality, ordering, and visibility relations over groups of related objects. Collectively, these differences make reasoning techniques proposed for weak (shared) memory ineffective in reasoning about weak (distributed) consistency, and mandate new formalizations of the kind proposed in this paper.

Representative examples of testing and checking frameworks for distributed systems include MaceMC[Killian et al. 2007] a model-checker that discovers liveness bugs in distributed programs, and [Jepsen 2018], a random testing tool that checks partition tolerance of NoSQL distributed database systems with varying consistency levels to enable high-availability. Q9 differs from these systems in significant ways: among other things, MaceMC does not consider safety issues related to replication, while Jepsen is purely a dynamic analysis that does not leverage semantic properties of the application in searching for faulty executions.

Finally, there has been a vast body of work produced over the years that explore the use of symbolic execution as a means for more effective testing and bug-finding [Cadar and Sen 2013]. Surprisingly, we are unaware of any effort in this space that examines the applicability of symbolic execution to the problem of anomaly detection for highly-available geo-replicated distributed applications, a class of programs that are becoming increasingly important and pervasive.
9 CONCLUSION

This paper presents a programming model and symbolic analysis framework for detecting anomalies (i.e., executions that violate an application invariant) in distributed programs built using replicated data types. Our approach leverages a bounded symbolic execution technique that explores a state space of a distributed execution. The Q9 symbolic execution engine undertakes this exploration constrained by a bound on the number of concurrent effects, outstanding (symbolic) updates to a state that have not been applied on all replicas. Our experimental results demonstrate that this strategy is very effective, capable of identifying and repairing subtle safety violations over a range of well-studied applications.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their careful scrutiny and insightful comments. This material is based upon work supported by the National Science Foundation under Grant No. CCF-SHF 1717741 and the Air Force Research Lab under Grant No. FA8750-17-1-0006.

REFERENCES


