AFRL-RI-RS-TR-2020-145

QUELEA: VERIFIED IMPLEMENTATION OF WEAKLY - CONSISTENT DISTRIBUTED PROGRAMS

PURDUE UNIVERSITY

AUGUST 2020

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2020-145 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
STEVEN L. DRAGER
Work Unit Manager

/ S /
GREGORY J. HADYNSKI
Assistant Technical Advisor
Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
AUGUST 2020	FINAL TECHNICAL REPORT	OCT 2016 – DEC 2019

4. TITLE AND SUBTITLE

QUELEA: VERIFIED IMPLEMENTATION OF WEAKLY-CONSISTENT DISTRIBUTED PROGRAMS

5a. CONTRACT NUMBER
N/A

5b. GRANT NUMBER
FA8750-17-1-0006

5c. PROGRAM ELEMENT NUMBER
62303E

6. AUTHOR(S)

Suresh Jagannathan

5d. PROJECT NUMBER
HACM

5e. TASK NUMBER
PU

5f. WORK UNIT NUMBER
RD

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Purdue University
305 North University Street
West Lafayette IN 47907-2114

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

DARPA
675 North Randolph St
Arlington, VA 22203

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2020-145

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The Quelea project was centered on the development of language abstractions, database technologies, compilers, program analyses, verification techniques, and runtime systems for building scalable applications intended to operate in geo-distributed environments where issues of data consistency and visibility are significant and subtle. The most significant contributions of this project are:

1. program logics for reasoning about geo-replicated distributed data stores and distributed transactions;
2. automated verification techniques for proving the safety of applications executing in such environments;
3. model-checking techniques that increase the assurance of applications that are equipped with sophisticated safety invariants;
4. language design principles that yield correct-by-construction distributed applications; and,
5. testing frameworks that can automatically discover invariant violations with greater specificity than random testing approaches collectively validate the thesis underlying the proposed effort.

15. SUBJECT TERMS
Distributed computing, consistency, isolation, databases, automated verification, theorem proving, logic, program semantics

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**STEVEN L. DRAGER**
U	U	U	UU	59	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

Table of Contents

List of Figures

List of Tables

1 SUMMARY

The Quelea project was centered on the development of language abstractions, database technologies, compilers, program analyses, verification techniques, and runtime systems for building scalable applications intended to operate in geo-distributed environments where issues of data consistency and visibility are significant and subtle. The overarching goal of the project was to ensure that produced artifacts come equipped with verifiable guarantees on program and system behavior.

The approach of the Quelea project was to create an integrated software infrastructure to enable the construction of composable, distributed verified components capable of being deployed in high-assurance environments. The project aimed to provide fundamental advances to the state-of- the-art in language design, verification, compiler design, and runtime systems, especially as they pertain to formal reasoning on weakly consistent geo-replicated distributed environments. The project's main focus was on verification and program analysis - the project examined the challenges involved in building composable whole-system cross-layer verified components, especially in the context of systems intended to operate in complex decentralized operational environments, and which may thus exhibit arbitrary failures and faults, comprise multi-tiered application stacks, with diverse clients, servers, and databases, etc.

2 INTRODUCTION

There were five major activities undertaken during the lifetime of this effort developed with these goals in mind:

1. **Logical Foundations**: The project explored new program logics tailored to issues relevant to the specific contexts under investigation - program executions in which visibility of concurrent actions do not adhere to a sequentially consistent semantics.

2. **Automated Verification**: A central focus of the project was the study of new automated reasoning and verification techniques that could be used to verify the correctness of applications executing in weakly consistent geo-replicated environments.

3. **Model-Checking and Symbolic Execution**: When full verification is not feasible, bounded model-checking and expressive symbolic execution methods can be used to increase assurance on the safety and correctness of distributed applications. The project investigated a number of techniques in this space, demonstrating the feasibility of such approaches to identify potential correctness violations.

4. **Language Design**: New language abstractions can help alleviate the complexity of verification and the challenge of static analysis by facilitating correct-by-construction program development methodologies. The project proposed new language abstractions and associated implementation and runtime frameworks significantly more expressive and compositional than existing proposals for programming weakly consistent distributed systems.

5. **Testing and Replay**: Lightweight testing frameworks are an important tool to facilitate exploration of potential correctness concerns. As part of our investigation, we developed an automated directed testing and replay framework that reveals anomalous behavior of distributed applications.

2.1 Logical Foundations

Database transactions allow users to group operations on multiple objects into a single logical unit, equipped with a set of four key properties - atomicity, consistency, isolation, and durability (ACID). Concurrency control mechanisms provide specific instantiations of these properties to yield different ACID variants that characterize how and when the effects of concurrently executing transactions become visible to one another. *Serializability* is a

particularly well-studied instantiation that imposes strong atomicity and isolation constraints on transaction execution, ensuring that any permissible concurrent schedule yields results equivalent to a serial one in which there is no interleaving of actions from different transactions.

The guarantees provided by serializability do not come for free, however - pessimistic concurrency control methods require databases to use expensive mechanisms such as two-phase locking that incur overhead to deal with deadlocks, rollbacks, and re-execution [17, 19]. Similar criticisms apply to optimistic multi-version concurrency control methods that must deal with timestamp and version management [10]. These issues are exacerbated when the database is replicated, requiring additional coordination mechanisms [4, 9, 16, 20].

Because serializable transactions favor correctness over performance, there has been long-standing interest [23] in the database community to consider weaker variants that try to recover performance, even at the expense of simplicity and ease of reasoning. These instantiations permit a transaction to witness various effects of newly committed, or even concurrently running, transactions while it executes, thus weakening serializability's strong isolation guarantees. The American National Standards Institute (ANSI) Structured Query Language (SQL) 92 standard defines three such weak isolation levels which are now implemented in many relational and NoSQL databases. Not surprisingly, weakly-isolated transactions have been found to significantly outperform serializable transactions on benchmark suites, both on single-node databases and multi- node replicated stores [4, 5, 47], leading to their overwhelming adoption. A 2013 study [6] of 18 popular ACID and "NewSQL" databases found that only three of them offer serializability by default, and half, including Oracle 11g, do not offer it at all.

A major problem with weak isolation as currently specified is that its semantics in the context of user programs is not easily understood. The original proposal [23] defines multiple "degrees" of weak isolation in terms of implementation details such as the nature and duration of locks held in each case. The ANSI SQL 92 standard defines four levels of isolation (including serializability) in terms of various undesirable *phenomena* (*e.g., dirty reads* - reading data written by an uncommitted transaction) each is required to prevent. While this is an improvement, this style of definition still requires programmers to be prescient about the possible ways various undesirable phenomena might manifest in their applications, and in each case determine if the phenomenon can be allowed without violating application invariants. This is understandably hard, especially in the absence of any formal underpinning to define weak isolation semantics. Consequently, reasoning about weak isolation remains an error prone endeavor, with major database vendors [31, 35, 39] continuing to document their isolation levels primarily in terms of the undesirable phenomena a particular isolation level may induce, placing the burden on the programmer to determine application correctness.

To address this significant shortcoming, we developed a program logic for weakly-isolated transactions along with automated verification support to allow developers to verify the soundness of their applications, without having to resort to low-level operational reasoning as they are forced to do currently. Our approach involves the development of a set of syntax-directed compositional proof rules that enable the construction of correctness proofs for transactional programs in the presence of a weakly-isolated concurrency control mechanism. Realizing that the proof burden imposed by these rules may discourage applications programmers from using them, we have also developed an inference procedure that automatically verifies the weakest isolation level for a transaction while ensuring its invariants are maintained. The key to inference is a novel formulation of database state (represented as sets of tuples) as a monad, and in which database computations are interpreted as state transformers over these sets. This interpretation leads to an encoding of database computations amenable for verification by off-the-shelf satisfiability modulo theories (SMT) solvers. Our results provide the first formalization of weakly-isolated transactions, along with an expressive and compositional proof automation framework capable of verifying the safety of high-level consistency conditions attached to these transactions. Collectively, these contributions allow weakly-isolated transactions to enjoy the same rigorous reasoning capabilities as their strongly-isolated (serializable) counterparts.

2.2 Automated Verification

Verifying Serializability in Weakly-Consistent Environments: Serializability is a well-understood correctness property of database systems. Ensuring that all executions of such programs are serializable greatly simplifies reasoning about program correctness by reducing the complexity of understanding concurrent executions to the problem of understanding sequential ones. Unfortunately, *enforcing* serializability using runtime synchronization mechanisms is problematic in geo-replicated distributed systems without sacrificing availability (low-latency) [20]. To reap the correctness benefits of serializability with the performance and scalability benefits of high-availability, we studied the conditions under which transactional programs can be statically identified to always yield a serializable execution *without* the need for global synchronization. The challenge to realizing this goal stems from the complexity in reasoning about replicated state in which not all replicas share the same view of the data they hold.

To address this challenge, we devised a fully automated verification procedure that precisely encodes salient dependencies in the program as abstract executions defined in terms of an axiomatic specification of a particular weak consistency model. The procedure leverages a theorem prover to systematically search for the presence or absence of cycles in these executions consistent with these dependencies; the presence of a cycle indicates a serializability violation. Notably, our approach can be applied to any weak consistency model

whose specification can be expressed in first-order logic, a class that subsumes all realistic data stores we are aware of. More specifically, our approach constructs a dependency graph [1] from the input program containing a cycle and then asks whether there exists a valid execution under the given consistency specification that can result in this graph. To do this, we automatically extract dependency conditions from the transactional program, and relate these dependencies to artifacts in an event-based model to find whether there exists a valid abstract execution corresponding to the dependency graph. These dependencies are encoded in a first-order logic formula that is satisfiable only if there exists an execution that violates serializability.

Verification of Conflict-Free Replicated Data Types: For distributed applications, keeping a single copy of data at one location or multiple fully-synchronized copies (i.e. state-machine replication) at different locations, makes the application susceptible to loss of availability due to network and machine failures. On the other hand, having multiple unsynchronized replicas of the data results in high availability, fault tolerance and uniform low latency, albeit at the expense of consistency. In the latter case, an update issued at one replica can be asynchronously transmitted to other replicas, allowing the system to operate continuously even in the presence of network or node failures [20]. However, mechanisms must now be provided to ensure replicas are kept consistent with each other in the face of concurrent updates and arbitrary re-ordering of such updates by the underlying network. Over the last few years, *Conflict-free Replicated Datatypes* (CRDTs) [41, 45, 46] have emerged as a popular solution to this problem. Over the years, a number of CRDTs have been developed for common datatypes such as maps, sets, lists, graphs, etc.

The primary correctness criterion for a CRDT implementation is convergence (sometimes called *strong eventual consistency* (SEC) [21, 45]): two replicas which have received the same set of effectors must converge to the same CRDT state. Because of the weak default guarantees assumed to be provided by the underlying network, however, we must consider the possibility that updates can be applied in arbitrary order on different replicas, complicating correctness arguments. This complexity is further exacerbated because CRDTs impose no limitations on how often they are invoked, and may assume additional properties on network behavior [30] that must be taken into account when formulating correctness arguments.

Given these complexities, verifying convergence of operations in a replicated setting has proven to be challenging and error-prone [21]. To overcome these challenges, we have developed a novel *automated* verification strategy that allows us to directly connect constraints on events imposed by the consistency model with constraints on states required to prove convergence. Consistency model constraints are extracted from an axiomatization of network behavior, while state constraints are generated using reasoning principles that determine the *commutativity* and *non-interference* of sequences of updates, subject to these consistency

constraints. Both sets of constraints can be solved using off-the-shelf theorem provers. Because an important advantage of our approach is that it is parametric on weak consistency schemes, we are able to analyze the problem of CRDT convergence under widely different consistency policies (e.g., eventual consistency, causal consistency, parallel snapshot isolation (PSI) [49], among others), and for the first time verify CRDT convergence under such stronger models (efficient implementations of which are supported by real-world data stores). A further pleasant by-product of our approach is a pathway to take advantage of such stronger models to simplify existing CRDT designs and allow composition of CRDTs to yield new instantiations for more complex datatypes.

2.3 Model-Checking and Symbolic Execution

While our verification efforts explored how to verify standard correctness properties such as serializability or convergence, it remains a challenging exercise to determine if a distributed program built using replicated datatypes 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 [11, 13, 22, 36, 37, 54], doing so with a high-degree of automation was an important focus of activity in this project. Full verification of distributed applications is challenging in part because of the need to specify suitably strong inductive invariants [37] 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. This leads to 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 have considered 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 a Replicated Data Type (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.

2.4 Language Design

To overcome the challenges of building provably correct distributed applications, we considered how carefully-crafted language abstractions can facilitate the *automatic* derivation of correct distributed (replicated) variants of ordinary data types. Key to our approach is the use of *invertible relational specifications* of an inductive data type definition. These specifications capture salient aspects of the data type that are independent of its execution under any system model, thus freeing the programmers from having to explicitly reason about low-level operational issues related to replication, asynchrony, visibility, etc. Their relational structure, however, provides sufficient guidance on structural properties maintained by the type (e.g., element ordering) critical to how we might correctly *merge* multiple instances in a replicated setting.

Our approach is based on a model of replication centered around *versioned states* and explicit *merges*. In particular, we model replicated state in terms of concurrently evolving *versions* of a data type that trace their origin to a common ancestor version. We assume implementations synchronize pairs of replicas by merging concurrent versions into a single convergent version that captures salient characteristics of its parents. The merge operation is further aided by context information provided by the *lowest common ancestor* (LCA) version of the merging versions.

Because the exact semantics of merging depends on the type and structure of replicated state, data types define merge semantics via an explicit merge function. The merge function performs a three-way merge involving a pair of concurrent versions and their LCA version that constitutes the context for the merge. The version control model of replication, therefore, allows any ordinary data type equipped with a three-way merge function to become a distributed data type. The full expressivity of merge functions can be exploited to define bespoke distributed semantics for data types that need not necessarily mirror their sequential behavior (i.e., distributed objects that are not linearizable or serializable), but which are nonetheless well-defined (i.e., convergent) and have clear utility.

We consider how to derive correct merge functions automatically over arbitrarily complex (i.e, composable) data type definitions, and in the process, ascribe to them a meaningful and useful distributed semantics. By doing so, we eliminate the need to reason about low-level operational or axiomatic details of replication when transforming sequential data types to their replicated equivalents.

2.5 Testing and Replay

Database programs are an important application class for execution on scalable geo-replicated distributed fabrics. Realistic SQL-based databases typically have sophisticated structural relationships (schemas), and clients must ensure that every possible use of a database operation preserves these relationships. Testing that client applications do so is particularly challenging, as the operations they perform depends on the control structure and initial state of the application. Generating appropriate test inputs to discover violations of these relationships necessarily requires reasoning about the behaviors of these operations in the context of the program's execution. Further complicating matters is that the initial state of the database itself needs to be chosen so that tests expose interesting integrity and assertion violations. One important simplification that helps reduce the complexity of developing a useful testing strategy for these applications is to treat database transactions as *serializable*, restricting the set of interleavings that must be considered to those that maintain transaction atomicity and isolation.

The need for a practical automated testing framework for database applications executing in weakly consistent environments is particularly acute given the complexity of reasoning about program behavior in these environments. We have addressed this challenge by developing a testing framework that systematically and efficiently explores abstract executions of database-backed Java programs for serializability violations, assuming a weakly-consistent storage abstraction. To do so, we employ a bounded model-checking serializability violation detector that is parameterized by a specification of the underlying storage model, described above. Abstract executions capture various visibility and ordering relations among read and write operations on the database generated by queries; the structure of these relations is informed by the underlying data consistency model. Potential serializability violations in an abstract execution manifest as cycles in a dependency graph induced by these relations. When such violations are discovered, `CLOTHO` synthesizes concrete tests that can be used to drive executions that manifest the problem for assessment by application developers.

Through our experimental evaluation, we have been able to demonstrate that our approach can efficiently, automatically, and reliably detect and concretely manifest serializability anomalies of database programs executing on top of weakly-consistent data stores. An important by-product of our design is that it does not bake-in any specific consistency or isolation assumptions about the underlying storage infrastructure, allowing users to strengthen visibility and ordering constraints as desired.

3 METHODS, ASSUMPTIONS, AND PROCEDURES

The Quelea effort centers around the specification, analysis, verification, and development of distributed applications operating in environments that provide weaker guarantees than afforded by strong (or sequentially) consistent data stores. To illustrate the issue, 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 Positive-Negative-counter [46] 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 Figure 1.

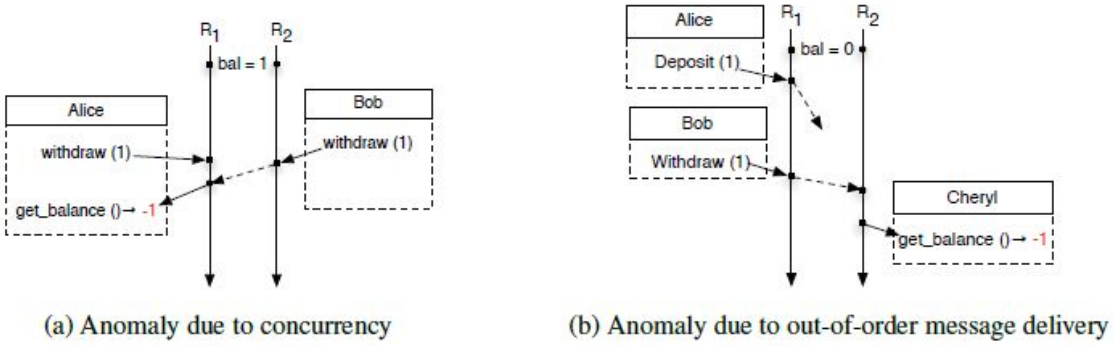


Figure 1: Anomalous executions of a simple bank account application. All operations operate over a single replicated account object.

Figure 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 R1 and R2, respectively. Each withdraw operation reads the local balance ($1), checks that it is sufficient for the withdraw ($1 ≥ $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.[1]

However, unintended executions that are unfamiliar to shared memory programmers are also possible in an asynchronous replicated system. Consider the execution shown in Figure 1b involving three users - Alice, Bob, and Cheryl, and two replicas - *R1* and *R2*. The initial balance at both the replicas is $0. Alice first submits a `Deposit` operation for $1 to *R1*. Bob, who subsequently connects to *R1*, finds there is sufficient balance to perform a `Withdraw` for $1. While effects from both the operations are expected to be delivered to *R2*, 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 *R2*. The `Deposit` effect will eventually be delivered to *R2* 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 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.

Developing tools capable of discovering, preventing, and/or repairing such concurrency anomalies is the central focus of this effort. Figure 2 shows the Quelea verification pipeline. We have adopted techniques that involve generating axiomatic constraints over program executions, encoding these constraints into a form amenable for interpretation by a theorem prover. This methodology can be used, for example, to identify representative counterexamples involving few concurrent operations, similar to those shown in Figure 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.

[1] 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.

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.

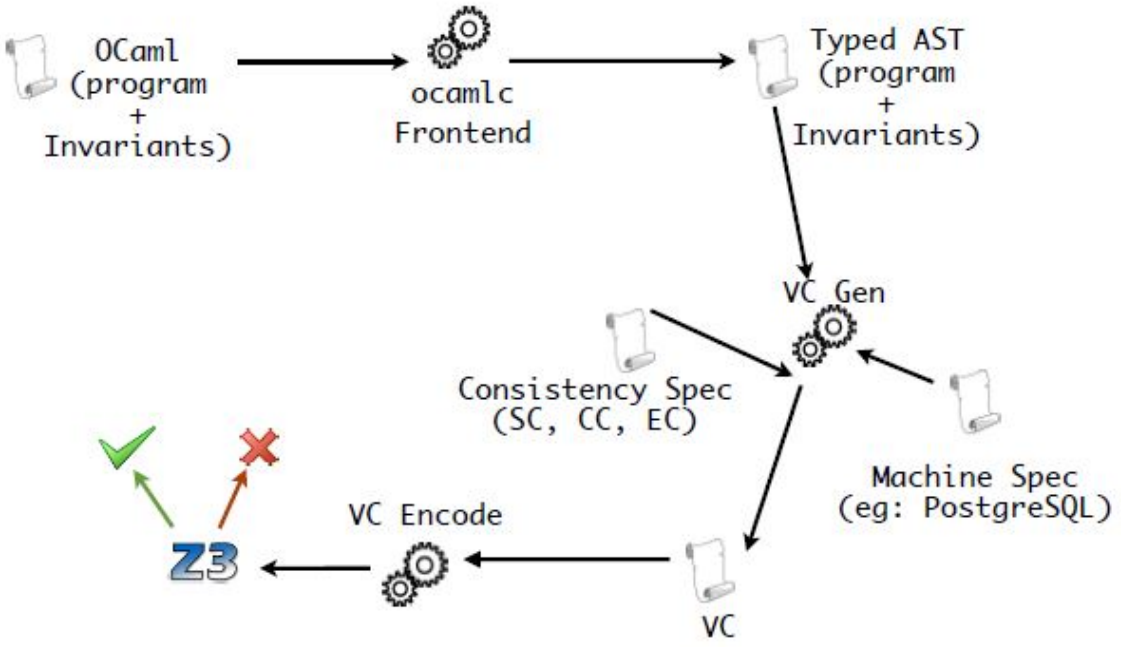


Figure 2: Overall verification pipeline used in Quelea.

4 RESULTS AND DISCUSSION

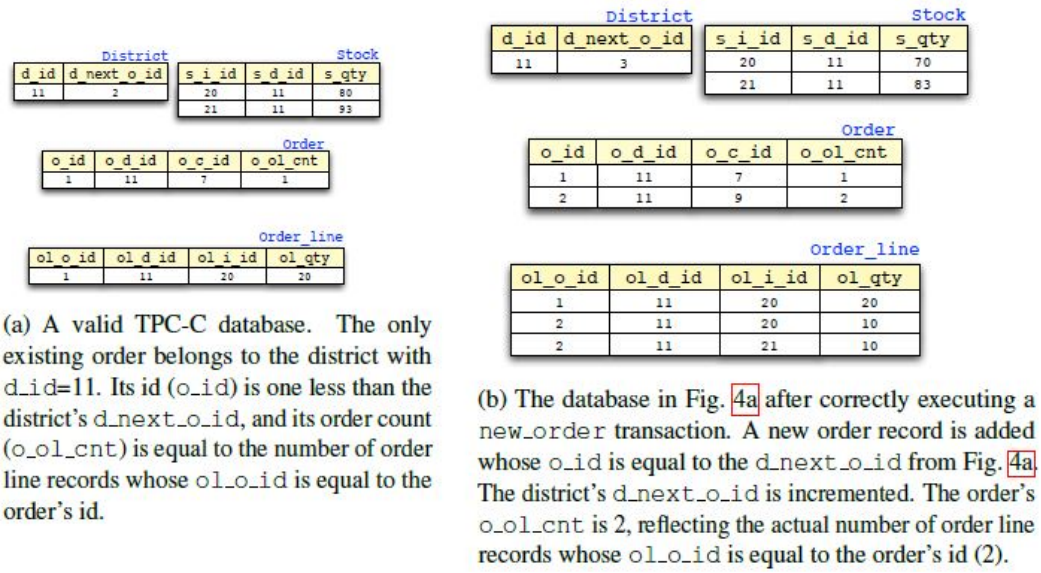
4.1 Reasoning about Weak Isolation

We have developed a program logic and verification methodology for reasoning about weakly isolated database transactions. To investigate our ideas, we have designed a domain-specific language (DSL) embedded in Objective Categorical Abstract Machine Language (OCaml) that manages an abstract database state that can be manipulated via a well-defined `SQL` interface. Arbitrary database computations can be built around this interface, which can then be run as transactions using the `atomically_do` combinator provided by the DSL.

Figure 3 shows a simplified version of the Transaction Processing Consortium benchmark C (TPC-C) `new_order` transaction written in this language. TPC-C [52] is a widely-used and well-studied Online Transaction Processing (OLTP) benchmark that models an order-processing system for a wholesale parts supply business. The business logic is captured in 5 database transactions that operate on 9 tables; `new_order` is one such transaction that uses `District`, `Order`, `New_order`, `Stock`, and `Order_line` tables. The transaction acts on the behalf of a customer, whose id is `c_id`, to place a new order for a given set of items (`item_reqs`), to be served by a warehouse under the district identified by `d_id`. Figure 4 illustrates the relationship among these different tables.

```
let new_order (d_id, c_id, item_reqs) = atomically_do @@ fun () ->
  let dist = SQL.select1 District (fun d -> d.d_id = d_id) in
  let o_id = dist.d_next_o_id in
  begin
    SQL.update (* UPDATE *) District
               (* SET *)(fun d -> {d with d_next_o_id = d.d_next_o_id + 1})
               (* WHERE *)(fun d -> d.d_id = d_id );
    SQL.insert (* INSERT INTO *) Order (* VALUES *){o_id=o_id;
              o_d_id=d_id; o_c_id=c_id; o_ol_cnt=S.size item_reqs; };
    foreach item_reqs @@ fun item_req ->
      let stk = SQL.select1 (* SELECT * FROM *) Stock
                (* WHERE *)(fun s -> s.s_i_id = item_req.ol_i_id &&
                                     s.s_d_id = d_id) (* LIMIT 1 *) in
      let s_qty' = if stk.s_qty >= item_req.ol_qty + 10
                   then stk.s_qty - item_req.ol_qty
                   else stk.s_qty - item_req.ol_qty + 91 in
      SQL.update Stock (fun s -> {s with s_qty = s_qty'})
                       (fun s -> s.s_i_id = item_req.ol_i_id);
      SQL.insert Order_line {ol_o_id=o_id; ol_d_id=d_id;
                             ol_i_id=item_req.ol_i_id; ol_qty=item_req.ol_qty}
  end
```

Figure 3: TPC-C `new_order` transaction



District

d_id	d_next_o_id
11	2

Stock

s_i_id	s_d_id	s_qty
20	11	80
21	11	93

Order

o_id	o_d_id	o_c_id	o_ol_cnt
1	11	7	1

Order_line

ol_o_id	ol_d_id	ol_i_id	ol_qty
1	11	20	20

(a) A valid TPC-C database. The only existing order belongs to the district with d_id=11. Its id (o_id) is one less than the district's d_next_o_id, and its order count (o_ol_cnt) is equal to the number of order line records whose ol_o_id is equal to the order's id.

District

d_id	d_next_o_id
11	3

Stock

s_i_id	s_d_id	s_qty
20	11	70
21	11	83

Order

o_id	o_d_id	o_c_id	o_ol_cnt
1	11	7	1
2	11	9	2

Order_line

ol_o_id	ol_d_id	ol_i_id	ol_qty
1	11	20	20
2	11	20	10
2	11	21	10

(b) The database in Fig. 4a after correctly executing a new_order transaction. A new order record is added whose o_id is equal to the d_next_o_id from Fig. 4a. The district's d_next_o_id is incremented. The order's o_ol_cnt is 2, reflecting the actual number of order line records whose ol_o_id is equal to the order's id (2).

Figure 4: Database schema of TPC-C's order management system. The naming convention indicates primary keys and foreign keys. For e.g., `ol_id` is the primary key column of the order line table, whereas `ol_o_id` is a foreign key that refers to the `o_id` column of the order table.

The transaction manages order placement by invoking appropriate SQL functionality, captured by various calls to functions defined by the `SQL` module. All `SQL` operators supported by the module take a table name (a nullary constructor) as their first argument. The higher-order `SQL.select1` function accepts a boolean function that describes the selection criteria, and returns any record that meets the criteria (it models the SQL query `SELECT ... LIMIT 1`). `SQL.update` also accepts a boolean function (its 3*rd* argument) to select the records to be updated. Its 2*nd* argument is a function that maps each selected record to a new (updated) record. `SQL.insert` inserts a given record into the specified table in the database.

The `new_order` transaction inserts a new `Order` record, whose id is the sequence number of the next order under the given district (`d_id`). The sequence number is stored in the corresponding `District` record, and updated each time a new order is added to the system. Since each order may request multiple items (`item_reqs`), an `Order_line` record is created for each requested item to relate the order with the item. Each item has a corresponding record in the `Stock` table, which keeps track of the quantity of the item left in stock (`s_qty`). The quantity is updated by the transaction to reflect the processing of new orders (if the stock quantity falls below 10, it is automatically replenished by 91).

TPC-C defines multiple invariants, called *consistency conditions*, over the state of the application in the database. One such consistency condition is the requirement that for a given order `o`, the *order-line-count* field (`o.o_ol_cnt`) should reflect the number of order lines under the order; this is the number of `Order_line` records whose `ol_o_id` field is the same as `o.o_id`. In a sequential execution, it is easy to see how this condition is preserved. A new

Order record is added with its `o_id` distinct from existing order ids, and its `o_ol_cnt` is set to be equal to the size of the `item_reqs` set. The `foreach` loop runs once for each `item_req`, adding a new `Order_line` record for each requested item, with its `ol_o_id` field set to `o_id`. Thus, at the end of the loop, the number of `Order_line` records in the database (i.e., the number of records whose `ol_o_id` field is equal to `o_id`) is guaranteed to be equal to the size of the `item_reqs` set, which in turn is equal to the `Order` record's `o_ol_cnt` field; these constraints ensure that the transaction's consistency condition is preserved.

Because the aforementioned reasoning is reasonably simple to perform manually, verifying the soundness of TPC-C's consistency conditions would appear to be feasible. Serializability aids the tractability of verification by preventing any interference among concurrently executing transactions, while the `new_order` transaction executes, essentially yielding serial behaviors. Under weak isolation[2], however, interferences of various kinds are permitted, leading to executions superficially similar to executions permitted by concurrent (racy) programs [18, 24]. To illustrate, consider the behavior of the `new_order` transaction when executed under a *Read Committed* (RC) isolation level, the default isolation level in 8 of the 18 databases studied in [6]. An executing RC transaction is isolated from *dirty writes*, i.e., writes of uncommitted transactions, but is allowed to witness the writes of concurrent transactions as soon as they are committed. Thus, with two concurrent instances of the `new_ order` transaction (call them T_1 and T_2), both concurrently placing new orders for different customers under the same district (`d_id`), RC isolation allows the execution shown in Figure 5.

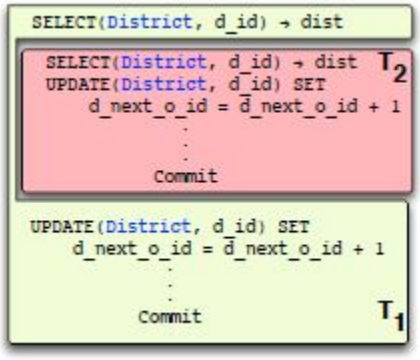


Figure 5: An RC execution involving two instances (T_1 and T_2) of the `new_order` transaction depicted in Figure 3. Both instances read the `d_id` `District` record concurrently, because neither transaction is committed when the reads are executed. The subsequent operations are effectively sequentialized, since T_2 commits before T_1. Nonetheless, both transactions read the same value for `d_next_o_id` resulting in them adding `Order` records with the same ids, which in turn triggers a violation of TPC-C's consistency condition.

[2]Weak isolation does not violate atomicity as long as the witnessed effects are those of committed transactions

Figure 5 depicts an execution as a series of SQL operations. In the execution, the `new_order` instance T_1 (green) reads the `d_next_o_id` field of the district record for `d_id`, but before it increments the field, another `new_order` instance T_2 (red) begins its execution and commits. Note that T_2 reads the same `d_next_o_id` value as T_1, and inserts new `Order` and `Order_line` records with their `o_id` and `ol_o_id` fields (resptively) equal to `d_next_o_id`. T_2 also increments the `d_next_o_id` field, which T_1 has already acccessed. This is allowed because reads typically do not obtain a mutually exclusive lock on most databases. After T_2's commit, T_1 resumes execution and adds new `Order` and `Order_line` fields with the same order id as T_1. Thus, at the end of the execution, `Order_line` records inserted by T_1 and T_2 all bear the same order id. There are also two `Order` records with the same district id (`d_id`) and order id, none of whose `o_ol_cnt` reflects the actual number of `Order_line` records inserted with that order id. This clearly violates TPC-C's consistency condition.

Our approach to identify and prevent such anomalous behavior is to *lift* isolation semantics (not their implementations) to the application layer, providing a principled framework to simultaneously reason about application invariants and isolation properties. To illustrate this idea, consider how we might verify that `new_order` is sound when executed under *Snapshot Isolation* (SI), a stronger isolation level than RC. Snapshot isolation allows transactions to be executed against a private snapshot of the database, thus admitting concurrency, but it also requires that there not be any write-write conflicts (i.e., such a conflict occurs if concurrently executing transactions modify the same record) among concurrent transactions when they commit. Write-write conflicts can be eliminated in various ways, e.g., through conflict detection followed by a rollback, or through exclusive locks, or a combination of both. For instance, one possible implementation of SI, close to the one used by PostgreSQL [39], executes a transaction against its private snapshot of the database, but obtains exclusive locks on the actual records in the database before performing writes. A write is performed only if the record that is to be written has not already been updated by a concurrent transaction. Conflicts are resolved by abort and roll back.

As this discussion hints, implementations of SI on real databases such as PostgreSQL are highly complicated, often running into thousands of lines of code. Nonetheless, the semantics of SI, in terms of how it effects transitions on the database state, can be captured in a fairly simple model. First, effects induced by one transaction (call it T) are not visible to another concurrently executing one during T's execution. Thus, from T's perspective, the global state does not change during its execution. More formally, for every operation performed by T, the global state T witnesses before (Δ) and after (Δ') executing the operation is the same ($\Delta' = \Delta$). After T finishes execution, it commits its changes to the actual database, which may have already incorporated the effects of concurrent transactions. In executions where T successfully commits, concurrent transactions are guaranteed to not be in write-write conflict with T. Thus,

if Δ is the global state that T witnessed when it finished execution (the snapshot state), and Δ' is the state to which T commits, then the difference (diff) between Δ and Δ' should not result in a write-write conflict with T. To concretize this notion, let the database state be a map from database locations to values, and let δ denote a transaction-local log that maps the locations being written to their updated values. The absence of write-write conflicts between T and the diff between Δ and Δ' can be expressed as: $\forall x \in dom\ (\delta)$, $\Delta'(x) = \Delta(x)$. In other words, the semantics of SI can be captured as an axiomatization over transitions of the database state ($\Delta \rightarrow \Delta'$) during a transaction's (T) lifetime:

- While T executes, $\Delta' = \Delta$.
- After T finishes execution, but before it commits its local state δ, $\forall(x \in dom\ (\delta))$. $\Delta'(x) = \Delta(x)$.

This simple characterization of SI isolation allows us to verify the consistency conditions associated with the `new_order` transaction. First, since the database does not change ($\Delta' = \Delta$) during execution of the transaction's body, we can reason about `new_order` as though it executed in complete isolation until its commit point, leading to a verification process similar to what would have been applied when reasoning sequentially. When `new_order` finishes execution, however, but before it commits, the SI axiomatization shown above requires us to consider global state transitions $\Delta \rightarrow \Delta'$ that do not include changes to the records (δ) written by `new_order`, i.e., $\forall(x \in dom\ (\delta))$. $\Delta'(x) = \Delta(x)$. The axiomatization precludes any execution in which there are concurrent updates to shared table fields (e.g., `d_next_o_id` on the same `District` table), but does not prohibit interferences that write to different tables, or write to different records in the same table. We need to reason about the safety of such interferences with respect to `new_order`'s consistency invariants to verify `new_order`.

We approach the verification problem by first observing that a relational database is a significantly simpler abstraction than shared memory. Its primary data structure is a table, with no primitive support for pointers, linked data structures, or aliasing. Although a database essentially abstracts a mutable state, this state is managed through a well-defined fixed number of interfaces (SQL statements), each tagged with a logical formula describing what records are accessed and updated.

This observation leads us away from thinking of a collection of database transactions as a simple variant of a concurrent imperative program. Instead, we see value in viewing them as essentially functional computations that manage database state abstractly, mirroring the structure of our DSL. By doing so, we can formulate the semantics of database operations as state transformers that explicitly relate an operation's pre- and post-states, defining the

semantics of the corresponding transformer algorithmically, just like classical predicate transformer semantics (e.g., weakest pre-condition or strongest post-condition). In our case, a transformer interprets a SQL statement in the set domain, modeling the database as a set of records, and a SQL statement as a function over this set.

Among other things, one benefit of this approach is that low-level loops can now be substituted with higher-order combinators that automatically lift the state transformer of its higher-order argument, i.e., the loop body, to the state transformer of the combined expression, i.e., the loop. Figure 6 illustrates this intuition on a simple example.

```
foreach item_reqs @@ fun item_req ->
  SQL.update Stock (fun s -> {s with s_qty = k1})
                   (fun s -> s.s_i_id = item_req.ol_i_id);
  SQL.insert Order_line {ol_o_id=k2; ol_d_id=k3;
                         ol_i_id=item_req.ol_i_id; ol_qty=item_req.ol_qty}
```

Figure 6: `foreach` *loop from Figure 3.*

Figure 6 shows a (simplified) snippet of code taken from Figure 3. Some irrelevant expressions have been replaced with constants (`k1`, `k2`, and `k3`). The body of the loop executes a SQL update followed by an insert. Recall that a transaction reads from the global database (Δ), and writes to a transaction-local database (δ) before committing these updates. An update statement filters the records that match the search criteria from Δ and computes the updated records that are to be added to the local database. Thus, the state transformer for the update statement (call it T_U) is the following function on sets[3]:

$$\lambda(\delta, \Delta).\ \delta \cup \Delta \gg= (\lambda s.\text{if table}(\texttt{s}) = \textsf{Stock} \wedge \texttt{s.s\_i\_id} = \texttt{item\_req.ol\_i\_id} \text{ then } \{\langle \texttt{s\_i\_id} = \texttt{s.s\_i\_id}; \texttt{s\_d\_id} = \texttt{s.s\_d\_id}; \texttt{s\_qty} = \texttt{k1}\rangle\} \text{ else } \emptyset)$$

Here, the set bind operator extracts record elements (`s`) from the database, checks the precondition of the update action, and if satisfied, constructs a new set containing a single record that is identical to `s` except that it binds field `s_qty` to value `k1`. This new set is added (via set union) to the existing local database state δ[4].

The transformer ($T_I(\delta, \Delta)$) for the subsequent `insert` statement can be similarly constructed:

$$\lambda(\delta, \Delta).\ \delta \cup \{\langle \texttt{ol\_o\_id} = \texttt{k2}; \texttt{ol\_d\_id} = \texttt{k3}; \texttt{ol\_i\_id} = \texttt{item\_req.ol\_i\_id}; \texttt{ol\_qty} = \texttt{item\_req.ol\_qty}\rangle\}$$

Observe that both transformers are of the form $\mathsf{T}(\delta, \Delta) = \delta \cup \mathsf{F}(\Delta)$, where F is a function that

[3]Bind ($\gg=$) has higher precedence than union ($\cup$). Angle braces ($\langle\ldots\rangle$) are used to denote records.
[4]For now, assume that the record being added is not already present in δ.

returns the set of records added to the transaction-local database (δ). Let F_U and F_I be the corresponding functions for T_U and T_I shown above. The state transformation induced by the loop body in Figure 3 can be expressed as the following composition of F_U and F_I:

$$\lambda(\delta, \Delta).\ \delta \cup F_U(\Delta) \cup F_I(\Delta)$$

The transformer for the loop itself can now be computed to be:

$$\lambda(\delta, \Delta).\ \delta \cup \texttt{item\_reqs} \gg\!= (\lambda \texttt{item\_req}.\ F_U(\Delta) \cup F_I(\Delta)$$

Observe that the structure of the transformer mirrors the structure of the program itself. In particular, SQL statements become set operations, and the `foreach` combinator becomes set monad's bind (≫=) combinator. As we demonstrate, the advantage of inferring such transformers is that we can now make use of a semantics-preserving translation from the domain of sets equipped with ≫= to a decidable fragment of first-order logic, allowing us to leverage SMT solvers for automated proofs without having to infer potentially complex thread-local invariants or intermediate assertions.

In the exposition thus far, we assumed Δ remains invariant, which is clearly not the case when we admit concurrency. Necessary concurrency extensions of the state transformer semantics to deal with interference was therefore additionally developed.

4.2 Automated Verification

4.2.1 Verifying Serializability

Consider a simple banking application which maintains the balance of multiple accounts in a table `Account` which is indexed using the primary key `AccID` and contains the field `Balance`. Consider a `withdraw` operation, shown in Figure 7, written in a SQL-style language, which takes `ID` and `Amount` as input, and deducts the amount from the account with account number `ID`, if the balance is sufficient. Suppose the application is deployed in a distributed, replicated environment which allows concurrent invocations of the `withdraw` operation at potentially different replicas, with the only guarantee provided being eventual consistency - eventually all replicas will witness all updates to the `Balance` field. Under eventual consistency, the application is clearly not serializable, since concurrent `withdraw` operations to the same account–whose total withdrawn amount exceeds the balance of the account–could both succeed, which is not possible in a serializable execution.

```
withdraw (ID, Amount)
    SELECT Balance AS bal WHERE AccID=ID
    IF bal > Amount
        UPDATE SET Balance=bal-Amount WHERE AccID=ID
```

Figure 7: Example Application

A convenient way to express executions in such an environment is to use an axiomatic event-based representation. In this framework, an abstract execution [15] is expressed as the tuple (T, vis, ar), where T is the set of transaction invocations, vis $\subseteq T \times T$ is a visibility relation such that if $t \xrightarrow{vis} t'$ then updates of t are visible to t', and ar $\subseteq T \times T$ is an arbitration relation which totally orders all writes to the same location and ensures eventual consistency [14]. For example, if t_1 = `withdraw`(1,50), t_2 = `withdraw`(1,60), then $E = (\{t_1, t_2\}, \{\}, \{(t_1, t_2)\})$ is an abstract execution which is not serializable, because the final value of `Balance` in the account number 1 will only reflect the `withdraw` operation t_2 (assuming an initial `Balance` of 100 in `AccID 1`), since there is no visibility constraint enforced between the two operations. This is an example of a *lost update* [8] anomaly. Our goal is to automatically construct such anomalous executions.

A useful technique to detect serializability violations is to build dependency graphs from abstract executions, and then search for cycles in the dependency graph. The nodes of the dependency graph are invocations, and edges indicate dependencies between them. There are three type of dependencies relevant to serializability detection: $t_1 \xrightarrow{WR} t_2$ is a read dependency, which means that t_2 reads a value written by t_1, $t_1 \xrightarrow{WW} t_2$ is a write dependency, which means that both t_1 and t_2 write to the same location, with the write of t_2 arbitrated after t_1, and $t_1 \xrightarrow{RW} t_2$ is an anti-dependency, which means that t_1 does *not* read a value written by t_2 but instead reads an older version. For example, the dependency graph of the anomalous execution E described above is shown in Figure 8.

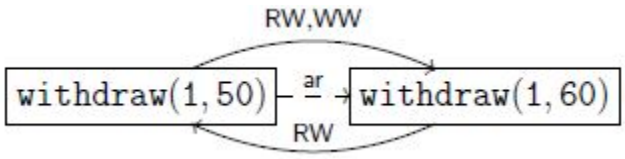


Figure 8: Abstract Execution E and its Dependency Graph

We thus start with a dependency graph containing a cycle, and then ask whether an execution corresponding to the dependency graph is possible. From the transaction code, we automatically extract the conditions under which a dependency edge can manifest between invocations of the transactions. In our running example, a dependency edge (of any type) between two `withdraw` invocations can only manifest if they are called with the same account `ID`. Further, we link the dependency edges with the relations vis and ar of the corresponding

abstract execution. For example, $t_1 \xrightarrow{RW} t_2 \Rightarrow \neg(t_2 \xrightarrow{vis} t_1)$, because otherwise, t_1 would read the value written by t_2. This is useful because different consistency schemes can be axiomatically expressed by placing constraints on the vis and ar relations. In order to prevent the anomalous execution in our running example, we can use PSI [49] which ensures that if two invocations write to the same location, then they cannot be concurrent. While PSI is implemented using a complex, distributed protocol, in our abstract framework, it can be simply expressed using the following constraint: $\forall t, t' \,.\, t \xrightarrow{WW} t' \Rightarrow t \xrightarrow{vis} t'$. Now, the anomalous execution E is not possible, because, $t_1 \xrightarrow{WW} t_2 \Rightarrow t_1 \xrightarrow{vis} t_2$ which contradicts $t_2 \xrightarrow{RW} t_1$.

To summarize, the following is the relevant portion of formulae that we generate for the above application under PSI:

$$\forall t, t' \,.\, t \xrightarrow{RW} t' \Rightarrow (\exists r.\text{AccID(r)} = \text{ID(t)} \wedge \text{AccID(r)} = \text{ID(t')} \wedge \text{bal(t')} > \text{Amount(t')}) \quad (1)$$

$$\forall t, t', \texttt{r}.\,(\texttt{AccID(r)} = \texttt{ID(t)} \wedge \text{bal(t)} > \text{Amount(t)} \wedge \text{AccID(r)} = \text{ID(t')} \wedge \text{bal(t')} > \text{Amount(t')} \wedge t \xrightarrow{ar} t' \Rightarrow t \xrightarrow{WW} t' \quad (2)$$

$$\forall t, t' \forall t, t'.t \xrightarrow{RW} t' \Rightarrow \neg(t \xrightarrow{RW} t') \quad (3)$$

$$\forall t, t' \,.\, t \xrightarrow{RW} t' \Rightarrow t \xrightarrow{RW} t' \quad (4)$$

We use t, t' to denote invocations of the transaction, and r to denote a record in the database. We define the function `AccID` to access the primary key of a record. Similarly, `ID`, `Amount`, etc. are functions which map an invocation to its parameters and local variables. The existence of a dependence between two invocations forces the existence of a record that both invocations must access, as well as conditions on the local variables required to perform the access (Eqn. 1). On the other hand, if two invocations are guaranteed to write to the same location, there must exist a WW dependency between them (Eqn. 2). Now, it is not possible to have invocations t_1 and t_2, obeying Eqns. (1)-(4) such that $t_1 \xrightarrow{RW} t_2$ and $t_2 \xrightarrow{RW} t_1$, the condition necessary to induce a cycle and thus manifest a serializability violation.

In fact, it is not possible to have a cycle of any arbitrary length in a dependency graph of this application under PSI. To show this, we use the following observation: any long path in a dependency graph generated by the above application will have chords in it, resulting in a shorter path. In fact, it can be shown that the shortest path between any two invocations in any dependency graph of the application (if there is a path) will always be less than or equal to 3. This can be shown by using the above constraints (1)-(4) (and adding similar constraints for WR edges), instantiating a path of length 4 such that there is no chord between any of the nodes involved in the path, and then showing the unsatisfiability of such an encoding. Since a cycle

is also a path, it is now sufficient to only check for cycles of length 3, since any longer cycle will necessarily induce a cycle of length less than or equal to 3.

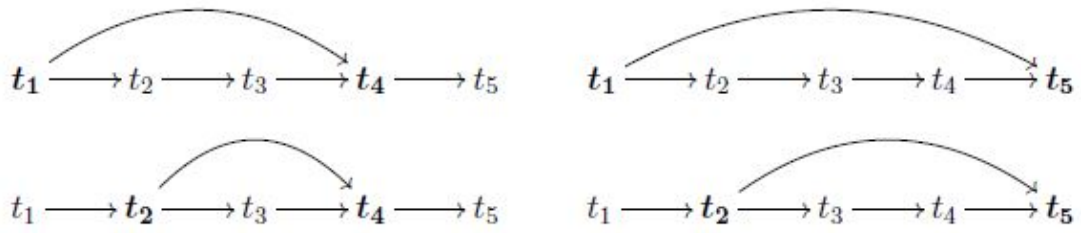


Figure 9: Different possibilities for paths of length 4 in the dependency graphs of the banking application. Note that transactions in bold perform writes.

Intuitively, this is happening in the banking application because the presence of any dependency edge between two nodes implies that both invocations must access the same account, and at least one of them must perform a write. Further, any two writes are always related by a WW edge. Now, as shown in Figure 9, in any path of length 4 in the dependency graph, one of t_1 or t_2 and one of t_4 or t_5 must be a write, which implies a chord between the two writes. Hence, there will always be a shorter path of length less than or equal to 3 between t_1 and t_5.

We have developed a tool called ANODE which takes a transactional program written in the style given above and a consistency specification, and uses the encoding rules described above to automatically generate an encoding in first-order logic (FOL). We use the Z3 SMT solver to determine the satisfiabiliy of the generated formulae. In order to evaluate the effectiveness of our approach, we have applied the proposed technique on TPC-C [52], a well-known OLTP benchmark widely used in the database community. TPC-C has a complex database schema with 9 tables, and complex application logic in its 5 transactions. The transactions contain loops and conditionals, have multiple parameters and behave differently depending upon the values of the parameters; they also use complex queries such as `SELECT MIN` and `SELECT MAX`. To the best of our knowledge, this is the first automated static analysis for validating serializability of TPC-C under weak consistency.

Under eventual consistency, TPC-C has a number of 'lost update' anomalies, similar to the anomaly in the banking application described in §2. These anomalies are small in length and were automatically detected using encoding presented in §5.1 (with k = 2). To get rid of these anomalies, we automatically upgraded the consistency specification to PSI [49]. Under PSI, we did not find any anomalies for $k = 2$ or $k = 3$, but for $k = 4$, the 'long fork' anomaly involving the `New-Order`, `Payment` and `Order-Status` transactions was discovered, as shown in Figure 10.

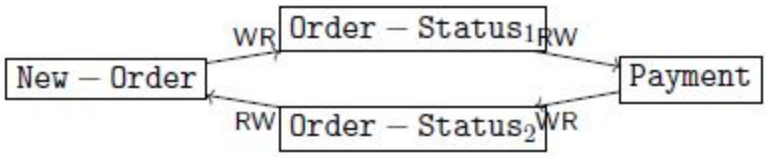


Figure 10: Long fork anomaly in TPC-C under PSI.

This anomaly happens because the `New-Order` and `Payment` transactions update two different tables (`Order` and `Customer` table respectively), while the `Order-Status` transaction reads both those tables. Since there is no synchronization between `New-Order` and `Payment` transactions, it is possible for $\texttt{Order-Status}_1$ to see the update of `New-Order` but not `Payment`, and vice versa for $\texttt{Order-Status}_2$. We also discovered a similar anomaly involving two instances of `New-Order` and two instances of `Stock-level` transactions.

4.2.2 Verifying Conflict-Free Replicated Datatypes

A Conflict-Free Replicated Datatype (Σ, O, σ_{init}) is characterized by a set of states Σ, a set of operations O and an initial state $\sigma_{init} \in \Sigma$, where each operation $o \in O$ is a function with signature $\Sigma \rightarrow (\Sigma \rightarrow \Sigma)$. The state of a CRDT is replicated, and when operation o is issued at a replica with state σ, the effector $o(\sigma)$ is generated, which is immediately applied at the local replica (which we also call the source replica) and transmitted to all other replicas, where it is subsequently applied upon receipt.

Additional constraints on the order in which effectors can be received and applied at different replicas are specified by a consistency policy, discussed below. In the absence of any such additional constraints, *however*, we assume the underlying network only offers *eventually consistent* guarantees - all replicas eventually *receive* all effectors generated by all other replicas, with no constraints on the order in which these effectors are received.

$$
\begin{array}{ll}
S \in E & \\
\texttt{Add(a):S} & \lambda S'.\, S' \cup \{a\} \\
\texttt{Remove(a):S} & \lambda S'.\, S' \setminus \{a\} \\
\texttt{Lookup(a):S} & a \in S
\end{array}
$$

Figure 11: A simple Set CRDT definition.

Consider the Set CRDT specification shown in Figure 11. Let E be an arbitrary set of elements. The state space Σ is `E`. `Add(a):S` denotes the operation `Add(a)` applied on a replica with state `S`, which generates an effector which simply adds a to the state of all other replicas it is applied to. Similarly, `Remove(a):S` generates an effector that removes `a` on all replicas to which it is applied. `Lookup(a):S` is a query operation which checks whether the queried element is present in the source replica `S`.

A CRDT is *convergent* if during any execution, any two replicas which have received the same set of effectors have the same state. Our strategy to prove convergence is to show that any two effectors of the CRDT pairwise commute with each other modulo a consistency policy, i.e. for two effectors e_1 and e_2, $e_1\ e_2 = e_2\ e_1$. Our simple Set CRDT clearly does not converge when executed on an eventually consistent data store since the effectors e_1 = `Add(a)`:S_1 and e_2 = `Remove(a)`:S_2 do not commute, and the semantics of eventual consistency imposes no additional constraints on the visibility or ordering of these operations that could be used to guarantee convergence. For example, if e_1 is applied to the state at some replica followed by the application of e_2, the resulting state does not include the element `a`; conversely, applying e_2 to a state at some replica followed by e_1 leads to a state that does contain the element `a`. However, while commutativity is a sufficient property to show convergence, it is not always a necessary one. In particular, different consistency models impose different constraints on the visibility and ordering of effectors that can obviate the need to reason about their commutativity. For example, if the consistency model enforces `Add(a)` and `Remove(a)` effectors to be applied in the same order at all replicas, then the Set CRDT will converge. As we will demonstrate later, the PSI consistency model exactly matches this requirement. To further illustrate this, consider the definition of the ORSet CRDT shown in Figure 12 and its variant in Figure 13. Here, every element is tagged with a unique identifier (coming from the set *I*). `Add(a,i):S` simply adds the element a tagged with i[5], while `Remove(a):S` returns an effector that when applied to a replica state will remove all tagged versions of `a` that were present in `S`, the source replica.

```
  S∈ E × I
Add(a,i):S
  λS'.S'∪{(a,i)}
Remove(a):S
  λS'.S'\{(a,i):(a,i)∈S}
Lookup(a):S
  ∃(a,i)∈A
```

Figure 12: A definition of an ORSet CRDT.

[5] Assume that every call to `Add` uses a unique identifier, which can be easily arranged, for example by keeping a local counter at every replica which is incremented at every operation invocation, and using the id of the replica and the value of the counter as a unique identifier

```
  S∈ E × I × E × I
Add(a,i):(A,R)
  λ(A',R').(A'U{(a,i)},R')

Remove(a):(A,R)
  λ(A',R').(A',R'U{(a,i):(a,i)∈A}

Lookup(a):(A,R)
  ∃(a,i)∈A∧(a,i)∉R
```

Figure 13: A variant of the ORSet using tombstones.

Suppose e_1 =Add(a,i):S_1 and e_2 =Remove(a):S_2. If it is the case that S_2 does not contain (a,i), then these two effectors are guaranteed to commute because e_2 is unaware of (a,i) and thus behaves as a no-op with respect to effector e_1 when it is applied to any replica state. Suppose, however, that e_1's effect was visible to e_2; in other words, e_1 is applied to S_2 before e_2 is generated. There are two possible scenarios that must be considered. (1) Another replica (call it S′) has e_2 applied before e_1. Its final state reflects the effect of the Add operation, while S_2's final state reflects the effect of applying the Remove; clearly, convergence is violated in this case. (2) All replicas apply $e1$ and e_2 in the same order; the interesting case here is when the effect of e_1 is always applied before e_2 on every replica. The constraint that induces an effector order between e_1 and e_2 on every replica as a consequence of e_1's visibility to e_2 on S_2 is supported by a causally consistent distributed storage model. Under causal consistency, whenever e_2 is applied to a replica state, we are guaranteed that e_1's effect, which adds (a,i) to the state, would have occurred. Thus, even though e_1 and e_2 do not commute when applied to an arbitrary state, their execution under causal consistency nonetheless allows us to show that all replica states converge. The essence of our proof methodology is therefore to reason about *commutativity modulo consistency* - it is only for those CRDT operations unaffected by the constraints imposed by the consistency model that proving commutativity is required. Consistency properties that affect the visibility of effectors are instead used to guide and simplify our analysis. Applying this notion to pairs of effectors in arbitrarily long executions requires incorporating commutativity properties under a more general induction principle to allow us to generalize the commutativity of effectors in bounded executions to the unbounded case. This generalization forms the heart of our automated verification strategy.

Results. To demonstrate the feasibility of our approach, we collected CRDT implementations from a number of sources [3,41,45] and since all of the existing implementations assume a very weak consistency model (primarily CC), we additionally implemented a few CRDTs on our own intended to only work under stronger consistency schemes but which are better in terms of time/space complexity and ease of development. Our implementations are not written

in any specific language but instead are specified abstractly akin to the definitions given in Figures 11 and 12. To specify CRDT states and operations, we fix an abstract language that contains uninterpreted datatypes (used for specifying elements of sets, lists, etc.), a set datatype with support for various set operations (add, delete, union, intersection, projection, lookup), a tuple datatype (along with operations to create tuples and project components) and a special uninterpreted datatype equipped with a total order for identifiers. Note that the set datatype used in our abstract language is different from the Set CRDT, as it is only intended to perform set operations locally at a replica. Our data types are uninterpreted since our verification technique generates verification conditions in first-order logic with universal quantifiers. All existing CRDT definitions can be naturally expressed in this framework.

CRDT	EC	CC	PSI+RB	PSI	Verif. Time (s)
Set					
Simple-Set	✗	✗	✓	✓	0.23
ORSet [45]	✗	✓	✓	✓	0.6
ORSet with Tombstones	✓	✓	✓	✓	0.04
USet [45]	✗	✗	✗	✓	0.1
List					
RGA [3]	✗	✓	✓	✓	5.3
RGA-No-Tomb	✗	✗	✓	✓	3
Graph					
2P2P-Graph [45]	✗	✓	✓	✓	3.5
Graph-with-ORSet	✗	✗	✓	✓	46.3

Figure 14: Convergence of CRDTs under different consistency policies.

Figure 14 shows the results of applying the proposed methodology on different CRDTs. We used Z3 to discharge our satisfiability queries. For every combination of a CRDT and a consistency policy, we write × to indicate that verification failed, while ✓ indicates that it was satisfied. We also report the verification time taken by Z3 for every CRDT across all consistency policies executing on a standard desktop machine. We have picked the three collection datatypes for which CRDTs have been proposed i.e. Set, List and Graph, and for each such datatype, we consider multiple variants that provide a tradeoff between consistency requirements and implementation complexity. Apart from eventual consistency (EC), causal consistency (CC) and parallel snapshot isolation, we also use a combination of PSI and red-black (RB), which only enforce PSI between selected pairs of operations (in contrast to simple RB which would enforce strong consistency (SC) between all selected pairs). Note that when verifying a CRDT under PSI, we assume that the set operations are implemented as Boolean assignments, and the write set Wr consists of elements added/removed. We are unaware of any

prior effort that has been successful in automatically verifying *any* CRDT, let alone those that exhibit the complexity of the ones considered here.

4.3 Model Checking

We have implemented a model checker and symbolic execution engine called ACIDIFIER to discover invariant violations of distributed transactional programs executing in weakly-consistent geo-replicated environments. The ACIDIFIER 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 [46]. The ACIDIFIER symbolic execution engine is implemented as a compiler pass that follows typechecking in the OCaml 4.03 compiler[6]. Its first component is a translator that translates high-level replicated datatype 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. 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 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[7].

The main component of the verifier is a symbolic execution engine that executes the body of an operation/transaction against symbolic inputs. Symbolic execution generates verification conditions (VCs) based on a formal definition of bounded safety. A `VC-Encode` component encodes these VCs as satisfiability queries in Z3, after asserting the required axioms on special relations defining visibility, session order, happens-before, 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.

Verification Experiments. To test the effectiveness of ACIDIFIER in detecting and fixing replication anomalies, we ported a range of applications, including several standard database

[6] https://github.com/tycon/q9

[7] 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.

benchmarks, to the ACIDIFIER 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 [53].
- **RUBiS**: Rice University Bidding System (RUBiS) [44] - 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 [52], 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.

Table 1: A sample of the anomalies found and fixes discovered by ACIDIFIER.

Oper/Txn	Violated Inv.	Anomalies	Fix
eBanking			
`withdraw`	`bal` ≥ 0	I. `Deposits` & `Withdraws` not being applied in causal order. II. Concurrent `Withdraws` independently succeed resulting in a negative balance.	CW + TW
`txn_transfer`	`bal` ≥ 0	Concurrent `txn_transfers` independently succeed resulting in a negative balance.	PSI
Twissandra			
`txn_new_tweet`	`Timeline` $\xrightarrow{\text{ref}}$ `Tweet`	Write to `Timeline` is applied before the previous write to `Tweet`	MW
`add_username`	Uniqueness of usernames	Concurrent checks for the uniqueness of a username succeed independently, resulting in duplicates.	TW
RUBiS			
`txn_bid_for_item`	`WalletBids` $\xrightarrow{\text{ref}}$ `Bids`	Write to `WalletBids` is applied before the previous write to `Bids`	MW
eCart			
`checkout`	$\forall(a \in \text{stock}).\ \text{qty}(a) \geq 0$	Concurrent checkouts of same items succeed independently resulting in negative stock.	TW
TPC-C			
`txn_new_order`	Per-district order ids are unique and sequential	Concurrent `txn_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.	PSI
TPC-E			
`complete_trade`	`Broker` $\xrightarrow{\text{ref}}$ `COUNT(Trade)`	Update to `Trade` is applied before the previous insert to `Trade`.	CW
`txn_trade_result`	`Broker` $\xrightarrow{\text{ref}}$ `COUNT(Trade)`	Concurrent `trade_result` txns complete the same trade, and independently increment `Broker`' `num_trades`.	PSI

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 1 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 `bal` ≥ 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 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 $A \xrightarrow{ref} \texttt{B}$, 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 $A \xrightarrow{ref} f(\texttt{B})$, 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. Usually, whenever $A \xrightarrow{ref} f(\texttt{B})$ there is also an inverse one-to-one relation, i.e., $B \xrightarrow{ref} A$. An example of one-to-many referential integrity is the $\texttt{Order} \xrightarrow{ref} \texttt{COUNT(OrderLine)}$ invariant in TPC-C, which requires an order's `o_ol_count` field to accurately reflect the number of `OrderLine` records referring back to the order. Another example is TPC-C's $\texttt{Warehouse} \xrightarrow{ref} \texttt{SUM(History)}$ 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 1 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. As an example of the kind of repair ACIDIFIER was able to perform, consider TPC-C's `txn_new_order` transaction, which adds a new `Order` record with an id (`Order.o_id`) equal to the sequence number of the next order for the corresponding district (`District.next_o_id`). The transaction also increments the district's order sequence number. During the verification of `txn_new_order`, ACIDIFIER 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_orders` reading the same id of the district's next order (`District.next_o_id`), and inserting duplicate `Order` records with that id. Subsequent to the discovery of the anomaly, ACIDIFIER was also able to use the counterexample to reason that if `txn_new_order` is executed under the Parallel

Snapshot Isolation (PSI) consistency model, then the anomaly can be preempted. While ACIDIFIER found a violation of uniqueness invariant in TPC-C, through a similar reasoning it found a violation of one-to-many referential integrity invariant ($\texttt{Broker} \xrightarrow{ref} \texttt{COUNT(Trade)}$) in TPC-E. The fix, again, is to strengthen the transaction's consistency level to PSI.

ACIDIFIER 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, ACIDIFIER 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; ACIDIFIER discovers that the anomaly can be preempted by executing the transaction under the Monotonic Writes consistency model, which is weaker than atomicity, and is cheaper on some systems [50, 51]. Similar deductions were made for the `txn_bid_for_item` transaction in RUBiS.

Table 2: Verification Statistics

Application	Opers	Txns	Anomalies found	Max k for an anomaly	Max time (s) for an anomaly	Max k verified
eBanking	3	2	3	5	0.28	60
Twissandra	20	10	5	5	6.59	50
RUBiS	17	6	5	5	3.03	50
eCart	10	5	5	6	1.09	60
TPC-C	18	5	6	10	51.79	18
TPC-E	44	10	3	10	113.53	17

Table 2 shows various statistics quantifying the cost and efficacy of bounded verification. The table demonstrates ACIDIFIER 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 ACIDIFIER provides.

The main takeaway from Table 2 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 ACIDIFIER 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 ACIDIFIER 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 2. 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 ACIDIFIER's approach of using symbolic execution-driven bounded verification to discover anomalies in real distributed applications.

Validation Experiments. Since ACIDIFIER does bounded verification, we validated the consistency assignments discovered by ACIDIFIER by testing the applications on a distributed database. Our goal was two-fold. First, we would like to check if the consistency assignments discovered by ACIDIFIER 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 ACIDIFIER during verification; i.e., there are no false positives.

Our experimental setup consisted 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 [29] in the same vein as [7, 48]. We instantiated 2 replicas within the same data center with an 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 performed retransmission of the dropped effects until all the effects were received everywhere. Consequently, every replica received 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 ACIDIFIER 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 ACIDIFIER 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.

4.4 Language Design

We have also explored new language design principles that can facilitate correct-by-construction distributed applications in which issues of consistency and visibility can confound program reasoning.

```
module Queue: sig
  type 'a t
  val push: 'a -> 'a t -> 'a t
  val pop: 'a t -> 'a option * 'a
      t
end = ...
```

Figure 15: The signature of a queue in OCaml

Consider a queue data structure whose OCaml interface is shown in Figure 15. `Queue` supports two operations: `push a`, that adds an element `a` to the tail end of the queue, and `pop`, that removes and returns the element at the head of the queue (or returns `None` if the queue is empty). We say the client that performed `pop` has *consumed* the popped element. For simplicity, we realize queue as a list of elements, i.e., we concretize the type `'a Queue.t` as `' a list` for this discussion. Like `Counter` with `mult`, `Queue`'s implementation does not qualify it as a CRDT, since `push` and `pop` do not commute. Hence, its semantics under (operation-centric) asynchronous replication is ill-founded as illustrated in Figure 16.

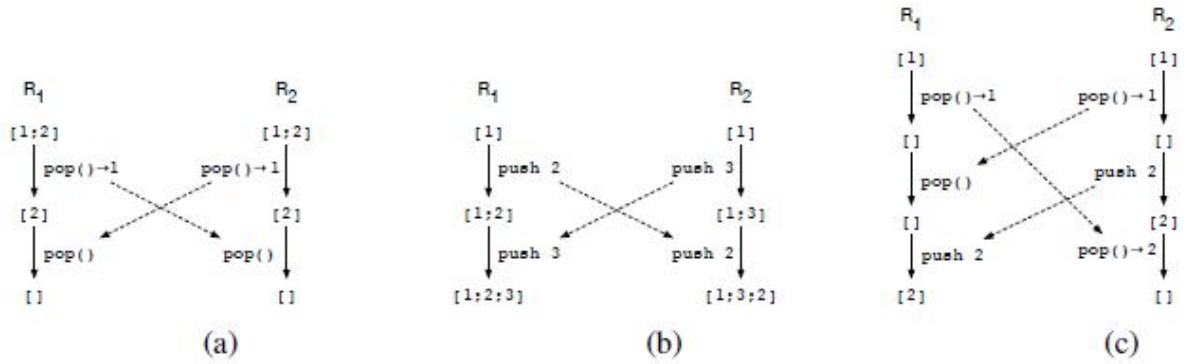


Figure 16: Ill-formed queue executions

The execution shown in Figure 16a starts with two replicas, R_1 and R_2, of a queue containing the elements 1 followed by 2. Two distinct clients connect to each of the replicas and concurrently perform `pop` operations, simultaneously consuming 1. The `pops` are then propagated over the network and applied at the respective remote replicas to keep them consistent with the origin. However, due to a concurrent `pop` already being applied at the remote replica, the subsequently arriving `pop` operation pops a different and yet-to-be-consumed element 2 in each case. The result is a convergent yet incorrect final state, where

the element 2 vanishes without ever being consumed. Figure 16b shows a very similar execution that involves `pushes` instead of `pops`. Starting from a singleton queue containing 1, two concurrent `push` operations push elements 2 and 3 respectively on different replicas. When these operations are eventually applied at the remotes, they are applied in different orders, resulting in the divergence of replica states. Figure 16c shows another example of divergence, this time involving both `pushes` and `pops`. The execution starts with two replicas, R_1 and R_2, of a singleton queue containing the 1. Two `pop` operations are concurrently issued by clients, both (independently) consuming 1. The `pops` are then applied at the respective remotes after a delay. During this delay, R_1 sees no activity, leaving the queue empty for R_2's pop, which effectively becomes a `Nop`. On R_2 however, a `push 2` operation is performed meanwhile, so when R_1's `pop` is subsequently applied, it pops the (yet unconsumed) element 2. As a result, the final state of the queue on R_2 is empty. Like the `pops`, the `push 2` operation is also propagated and eventually applied on R_1, resulting in the final state on R_1 being a singleton queue. Thus the replicas R_1 and R_2 of the final state of the queue diverge, which preempts any consistent semantics of the queue operations from being applied to explain the execution.

Bad executions, such as those in Figure 16, can be avoided if every queue operation is globally synchronized. However, enforcing global synchronization requires sacrificing availability (i.e., latency), an undesirable tradeoff for most applications [12]. It may therefore seem impossible to replicate queues with meaningful and useful semantics without losing availability. Fortunately, this turns out not to be the case. In the context of real applications, there exist implementations of highly available replicated queues whose semantics, albeit non-standard, i.e., not linearizable or serializable, have nonetheless proven to be useful. Amazon's Simple Queue Service (SQS) [2] is one such queue implementation with a non-standard *at-least-once delivery* semantics, which guarantees, among other things, that a queued message is delivered to a client for consumption at least once. Devoid of a formal context, such semantics may seem *ad hoc*; however, casting the `Queue` data type as a mergeable type would let us *derive* such semantics from first principles, thus giving us a formal basis to reason about its correctness.

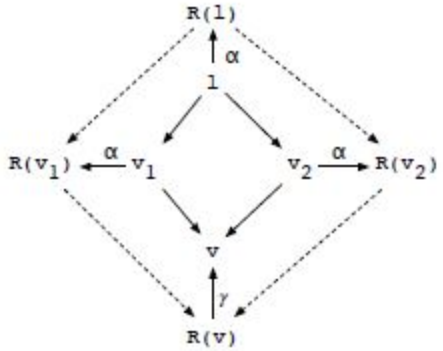


Figure 17: Merging values in relational domain with help of abstractions (α) and concretization (γ) functions. Solid (respectively dashed) unlabeled arrows represent a merge in the concrete (respectively abstract) domain.

Our underlying execution model is based on a state-centric model of replication with versioned state and explicit three-way merges (which we synthesize automatically). Under this model, two concurrent versions v_1 and v_2 of a queue can independently evolve from a common ancestor, LCA, version l. The semantics of the queue under replication depends on how these versions are merged into a single version v (Figure 17). The concurrent versions v_1 and v_2 would have evolved from l through several `push` and `pop` applications, however let us ignore the operations for a while and focus on the relationship between the queue states l, v_1, and v_2. Intuitively, the following relationships must hold among the three queues:

1. For every element $x \in l$, if $x \in v_1$ and $x \in v_2$, i.e., if x is not popped in either of the concurrent versions, then $x \in v$, i.e., x must be in the merged version. In other words, a queue element that was never consumed should not be deleted.

2. For every $x \in l$ if $x \notin v_1$ or $x \notin v_2$, i.e., if x is popped in either v_1 or v_2, then $x \notin v$. That is, a consumed element (regardless of how many times it was consumed) should never reappear in the queue.

3. For every $x \in v_1$ (respectively v_2), if $x \notin l$, that is x is newly pushed into v_1 (respectively v_2), then $x \in v$. That is, an element that is newly added in either concurrent versions must be present in the merged version.

4. For every $x, y \in l$ (respectively v_1 and v_2), if x occurs before y in l (respectively v_1 and v_2), and if $x, y \in v$, i.e., x and y are not deleted, then x also occurs before y in v. In other words, the order of elements in each queue must be preserved in the merged queue.

To formalize these properties more succinctly, we define two relations on lists: (1) A *membership* relation on a list l (written $R_{mem}(l)$) is a unary relation, i.e., a set, containing all the elements in l, and (2). An *occurs-before* relation on l (written $R_{ob}(l)$) is a binary relation relating every pair of elements x and y in l, such that x occurs before y in l. For a concrete list l = [1; 2;

3], $R_{mem}(l)$ is the set {1, 2, 3}, and $R_{ob}(l)$ is the set {(1, 2), (1, 3), (2, 3)}. Note that for any list l $R_{ob}(l) \subseteq R_{mem}(l) \times R_{mem}(l)$, i.e., $R_{ob}(l)$ is only defined for the elements in $R_{mem}(l)$. Using R_{mem}, we can succinctly specify the relationship among the members of l, v_1, v_2, and v, where v = merge l v_1 v_2, as follows[8]:

$$R_{mem}(v) = R_{mem}(l) \cap R_{mem}(v_1) \cap R_{mem}(v_2) \cup R_{mem}(v_1) - R_{mem}(l) \cup R_{mem}(v_2) - R_{mem}(l) \quad (5)$$

The left-hand side denotes the set of elements in the merged version v. The right-hand side is a union of three components: (1) The elements common among three versions l, v_1, and v_2, (2) The elements in v_1 not in l, i.e., newly added in v_1, and (3) The elements in v_2 not in l, i.e., newly added in v_2. Observe that we applied the same intuitions as the counter merge to arrive at the above specification, namely merging concurrent versions by computing, composing and applying their respective *differences* to the common ancestor. However, we have interpreted the *difference* through the means of a relation over sets that abstracts the structure of a queue and captures only its membership property. Another important point to note is that the specification does not appeal to any operational characteristics of queues, either sequentially or in the context of replication.

Similar intuitions can be applied to manage the structural aspects of merging queues by capturing their respective *orders* via the *occurs-before* relation (R_{ob}) over lists, but after accounting for a couple of caveats. First, since $R_{ob} \subseteq R_{mem} \times R_{mem}$, $R_{ob}(v)$ has to be confined to the domain of $R_{mem}(v) \times R_{mem}(v)$. Second, the order between a pair of elements where each comes from a distinct concurrent version is indeterminate, thus $R_{ob}(v)$ can only be underspecified. Taking these caveats into account, $R_{ob}(v)$ of the merged version v can be specified thus:

$$\begin{aligned} \text{Rob (v)} \supseteq\ & (\text{Rob(l)} \cap \text{Rob(v1)} \cap \text{Rob(v2)} \cup \text{Rob(v1)} - \text{Rob(l)} \cup \text{Rob(v2)} - \text{Rob(l)}) \\ & \cap (\text{rmem(v)} \times \text{Rmem(v)}) \end{aligned} \quad (6)$$

[8] We elide parentheses for perspicuity. Any ambiguity in parsing should be resolved by assuming that ∩ and – bind tighter than ∪

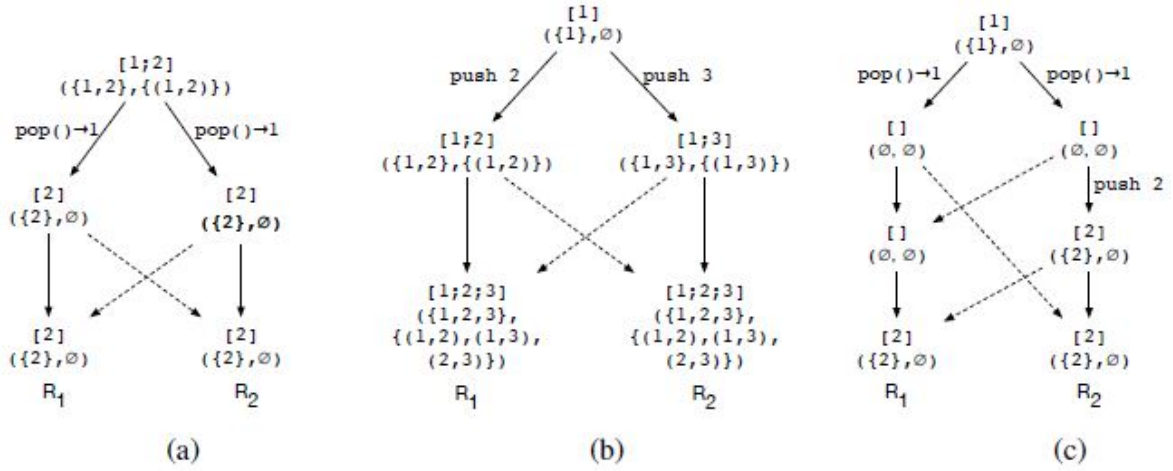


Figure 18: State-centric view of queue replication aided by context-aware merges (shown in dashed lines)

Note the $\supseteq$ capturing the underspecification. The right hand side is essentially the same as the right hand side of the R_{mem} equation (above), except that R_{ob} replaces R_{mem}, and we compute an intersection with $R_{mem}(v) \times R_{mem}(v)$ at the top level to confine $R_{ob}(v)$ to the elements in v. As mentioned earlier, the specification does not induce a fixed order among elements coming from different queues. To recover convergence, a merge function on queues can choose to order such elements through a consistent ordering relation, such as a lexicographic order.

The *membership* and *occurs-before* specifications together characterize the merge semantics of the queue data type that we derived from basic principles we enumerated above. We shall now reconsider the executions from Figure 16, this time under a state-centric model of replication, and demonstrate how our merge specification leads us to a consistent distributed semantics for queue, which subsumes a *at-least-once delivery* semantics. The corresponding executions under this model are shown in Figure 18.

Figure 18a is the same execution as in Figure 16a with the dashed line representing a version propagation followed by a merge, rather than an operation propagation followed by an application. For each version, the R_{mem} and R_{ob} relations are shown below its actual value. If the version is a result of a merge, then we compute its R_{mem} and R_{ob} sets using equations 5 and 6 of the merge specification above. For both the merges shown in the figure, the concurrent versions (v_1 and v_2) are the same: the singleton queue [2], and their LCA version (l) is the initial queue [1;2]. Thus each concurrent version is a result of popping 1 from the LCA (which is consumed/delivered twice as acceptable under *at-least-once delivery* semantics). Intuitively, the result of the merge should be a version that incorporates the effect of popping 1, while leaving the rest of the queue unchanged from the LCA. This leaves the queue [2] as the only possible result of the merge (and the execution). Indeed, this is the result we would obtain if reconstructing the queue from the merged R_{mem} and R_{ob} relations shown in the figure.

The execution depicted in Figure 18b corresponds to the one in Figure 16b. Here we have two merges: one into R_1 and other into R_2. The concurrent versions for both the merges are the same: [1;2] and [1;3], and their LCA is the queue [1]. Each concurrent version pushes a new element (2 and 3, respectively) to the queue *after* the existing element 1. Intuitively, the merged queue should contain both the new elements ordered after 1. Indeed, this is also what the merged R_{mem} and R_{ob} relations suggest. The order between new elements, however, is left unspecified by R_{ob}. As mentioned earlier, a consistent ordering relation has to be used to order such elements. Choosing the less-than relation, we obtain the result of the merge as [1;2;3]. In Figure 18c, there are three merges: two into R_1 and one into R_2. For the first merge into R_1, the concurrent versions are both empty queues, and their LCA is the singleton queue [1]. Thus both versions represent a `pop` of 1, and their merged version, which reconciles both the pops, should be an empty queue, which is also what the merged relations suggest. The second merge into R_1 and the only merge into R_2, both merge an empty queue ([]) and a singleton queue [2], with the LCA version being the initial queue [1]. While the version [] can be understood as resulting from popping an element from LCA, the concurrent version [2] goes one step ahead and pushes a new element 2. Consequently, the merged version should be a queue not containing 1, but containing the new element 2, i.e., [2], which is again consistent with the result obtained by merging the R_{mem} and R_{ob} relations. Thus in all three executions discussed above, the relational merge specification (Eqs. 5 and 6) consistently guides us towards a meaningful result, imparting a well-defined distributed semantics to the queue data type in the process.

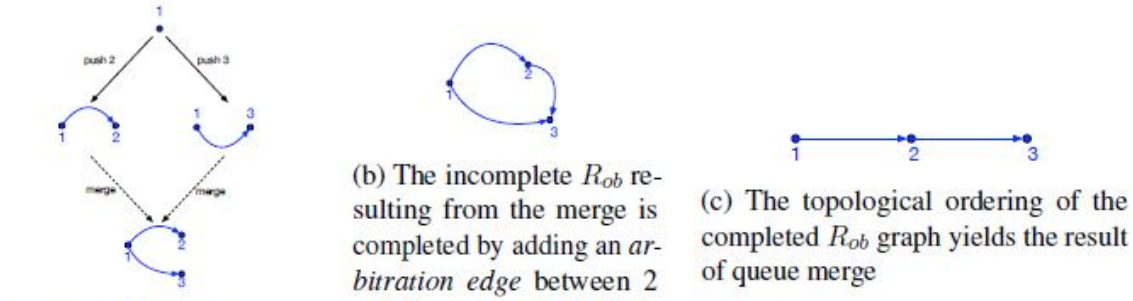
```
let rec R_mem = function
    | [] -> ∅
    | x::xs -> {x} ∪ R_mem(xs)
```

```
let rec R_ob = function
  | [] -> ∅
  | x::xs -> ({x} × R_mem(xs)) ∪ R_ob(xs)
```

Figure 19: Functions that compute R_{mem} and R_{ob} relations for a list. Syntax is stylized to aid comprehension.

To operationalize the merge specification discussed above, i.e., to derive a merge function that *implements* the specification, we require functions (α and γ respectively) to map a queue to the relational domain and back. The abstraction function α is simply a pair-wise composition of functions that compute R_{mem} and R_{ob} relations for a given list. The eponymous functions are shown in Figure 19. The R_{mem} function computes the set of elements in a given list l, which is its unary membership relation. The function R_{ob} computes the set of all pairs (x, y) such that x occurs before y in l. The concretization function γ reconstructs a list/queue given a subset of its R_{mem} and R_{ob} relations (The subsets are a consequence of underspecification, e.g., R_{ob} specification in Eq. 6). One way γ can materialize a list from the given subsets of its R_{mem} and R_{ob} relations is by constructing a directed graph G whose vertices are $R_{mem}(v)$, and edges are

$R_{ob}(v)$. A topological ordering of vertices in G, where ties are broken as per a consistent *arbitration* order (e.g., lexicographic order) yields the merged list/queue. Figure 20 demonstrates this approach for the queue merge example in Figure 18b.



(a) R_{ob} relations of merging queues from Fig. 18b visualized as graphs

(b) The incomplete R_{ob} resulting from the merge is completed by adding an *arbitration edge* between 2 and 3 consistent with their domain ordering

(c) The topological ordering of the completed R_{ob} graph yields the result of queue merge

Figure 20: Concretizing a queue from a subset of its R_{ob} relation resulting from a merge

We have generalized the aforementioned graph-based approach for concretizing ordering relations, and abstracted it away as a library function γ_{ord}. Given *ord* and an *arbitration order*, the function γ_{ord} concretizes an ordering relation of a data structure (not necessarily a total order) as a graph isomorphic to that structure, using the arbitration order to break ties (as shown in Figure 20b). Instantiating *ord* with less-than relation (<) on integers, the concretization function of a queue can be written as shown in Figure 21a. The result of $\gamma_<$(`rmem`,`robs`) is a list-like graph as shown in Figure 20c. The function `mk_list` traverses the graph beginning from its root to construct a list isomorphic with the graph. Standard library function `Set.elements` returns a list of elements in a set. The `DiGraph` library is assumed to support a function `root` that returns a root (vertex with indegree 0) of a directed graph, and a function `succ` that returns the list of successors of the given vertex in the graph.

```
let γ (rmem, robs) =
  if robs = ∅
  then Set.elements rmem
  else
    let g = γ< (rmem,robs) in
    let rec mk_list x =
      match DiGraph.succ x with
      | [] -> [x]
      | [y] -> x::(mk_list y)
      | _ -> error()
    mk_list (DiGraph.root g)
```

(a) Queue concretization function in OCaml

```
let merge l v1 v2 =
  let (rmem_l, robs_l) = α(l) in
  let (rmem_v1, robs_v1) = α(v1) in
  let (rmem_v2, robs_v2) = α(v2) in
  let rmem_v = rmem_l ◇ rmem_v1 ◇
      rmem_v2 in
  let robs_v = (robs_l ◇ robs_v1 ◇
      robs_v2)
                ∩ (rmem_v × rmem_v) in
  γ(rmem_v, robs_v)
```

(b) Queue merge composed of abstraction (α) and concretization (γ) functions

Figure 21: Relational approach to queue merge materialized in OCaml (Along with Figure 19)

The γ_{ord} function thus (mostly) automates the task of concretizing orders, which is usually the non-trivial part of writing γ. Given both α and γ, the merge function for queues (lists, in general) follows straightforwardly from the merge specification as shown in Figure 21b. For brevity, we write `A ◊ B ◊ C` to denote the three-way merge of sets `A`, `B`, and `C`, which is defined thus:

$$A \lozenge B \lozenge = (A \cap B \cap C) \cup (B - A) \cup (C - A)$$

Implementation and Experimental Results. The infrastructure necessary to implement Mergeable Replicated Data Types (MRDTs), and execute them in an asynchronously replicated setting has been developed in terms of three major components collectively referred to as QUARK. The first component of QUARK is a library of MRDT modules corresponding to basic data structures, such as lists and binary trees, along with a collection of signatures (e.g., `MERGEABLE`) and functions (e.g., *concretize_order*), that aid in the development of new MRDTs. The second component is an OCaml compiler extension, developed modularly using PPX [40], that performs a dual function. Firstly, aided by module signatures and compiler directives (e.g., `@@deriving merge`), the PPX extension identifies the OCaml type definitions of MRDTs, along with their abstraction and concretization functions, and composes them together, to generate the corresponding merge functions. While adding a merge function makes an OCaml data type *mergeable*, it is however not *replicated* for replication requires addressing low-level concerns, such as serialization, network fault tolerance etc. The third component of QUARK is a content-addressable distributed storage abstraction, called the QUARK store, that addresses these concerns, and the secondary function of the PPX extension is to generate the code that translates between the high-level, OCaml, representation of a data type, and its corresponding low-level representation in the QUARK store. The schematic diagram of this workflow is shown in Figure 22.

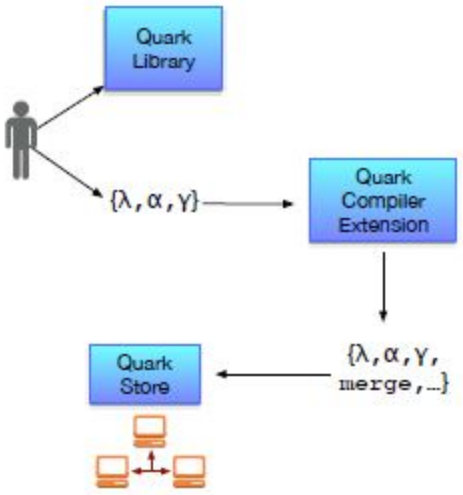
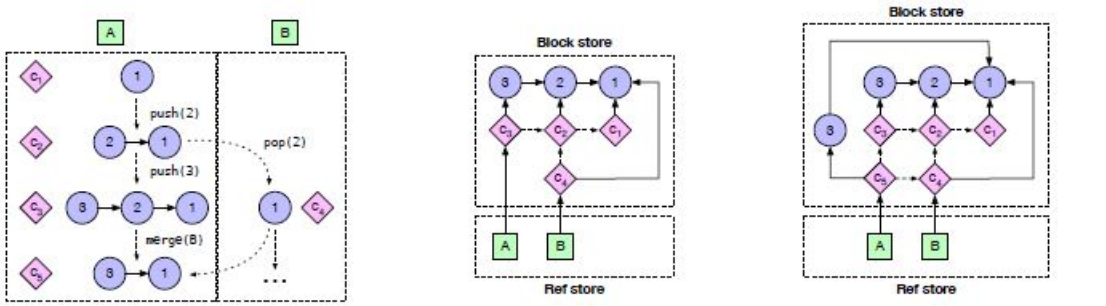


Figure 22: QUARK architecture: Programmer extends OCaml data types (λ) with abstraction (α) and concretization (γ) functions. The QUARK compiler extension generates merge and low-level code to interface with the QUARK store, which handles replication.

The key innovation of the QUARK store is the use of a storage layer that exposes a Git-like API, supporting common Git operations such as cloning a remote repository, forking off branches and merging branches using a three-way merge function. QUARK builds on top of these features to achieve a fault-tolerant, highly-available geo-replicated data storage system. For example, creating a new replica is realized by cloning a repository, and remote pushes and pulls are used to achieve inter-replica communication. QUARK store also supports a variety of storage back-ends including in-memory, file systems and fast key-value storage databases, and distributed data stores. We have additionally built a programming model around QUARK store's Git-like API to build distributed applications using MRDTs.

The main challenge in realizing MRDTs as a practical programming model is the need to efficiently store, compute and retrieve the LCA given two concurrent versions. QUARK uses a content-addressable block store for storing the data objects corresponding to concurrent versions of the MRDT as well as the history of each of the versions. Given that any data structure is likely to share most of the contents with concurrent and historical versions, content-addressability maximizes sharing between the different versions.



(a) An execution trace for stack MRDT. (b) Quark store before merging the commit c4 into c3. (c) Quark store after merging the commit c4 into c3.

Figure 23: The behavior of QUARK *content-addressable storage layer for a stack MRDT.* `A` *and* `B` *are two versions of the stack MRDT. Diamonds represent the commits and circles represent data objects.*

Consider the example presented in Figure 23a which shows an execution trace on a stack MRDT. There are two versions `A` and `B`. Version `B` is forked off from `A` and is merged onto `A`. Since `B` pops the element 2, it is no longer present in the merged version. `B` is of course free to further evolve concurrently with respect to `A`. The diamonds represent the *commits* that correspond to each historical version of the stack and circles represent data objects.

Figure 23b and Figure 23c represent the layout of the QUARK store before and after the merge. QUARK uses a content-addressable append-only *block* store for data and commit information. Objects in the block store are addressed by the content of their hashes. Correspondingly, links between the objects are hashes of the contents of the objects. The reference to the two versions `A` and `B` are stored in a mutable *ref* store. The versions point to a particular commit. The commits in turn may point to parent commits (represented by dashed lines between the diamonds), and additionally may point to a single data object. Data objects stored in the block store may only point to other data objects.

Observe that in Figure 23b, there is only one copy of the stack which is shared among both the concurrent and historical versions. Notice also that the branching structure of the history is apparent in the commit graph. In this example, we are merging the commits c_3 and c_4. QUARK traverses the commit graph to identify the lowest common ancestor c_2 and fetches the version of the stack that corresponds to the commit. After the merge, a new commit object c_5 is added along with a new data object for 3 which points to the existing data object 1 in the block store. The version ref for `A` in the ref store is updated to point to the new commit c_5. As our experimental results indicate, the use of a content-addressable store makes it efficient to implement MRDTs in practice.

4.5 Testing

As discussed earlier, TPC-C is a canonical Online Transaction Processing benchmark application that emulates a warehouse management and order entry/delivery system. The

benchmark was originally designed to validate a mixture of requirements, including transactional safety, expected from relational database systems [52]. To illustrate our approach to build an automated testing and replay facility for validating the correctness of distributed applications executing in weakly-consistent environments, consider Figure 24 (left) that presents a pruned code snippet implementing a transaction from TPC-C named txn*payment*. The snippet shows a procedure that updates the total number of successful payments from a customer, where the customer's associated row from `CUST` table is initially retrieved (lines 3-5) and is then rewritten with an incremented value (lines 10-13). Now, consider the deployment of TPC-C on a replicated cloud database presented in Figure 24 (right) where two txn*payment* transactions are concurrently executed on weakly-consistent replicas `A` and `B` that do not enforce serializability. The initial database state in this execution consists of the table `CUST(id, c_pay_cnt)` with a single row r_0 :=(10, 50). Additionally, the transaction instances txn1 and txn2 are given arguments such that both transactions internally access and update r_0. Both transactions concurrently read the initial value of r_0 at different replicas (depicted as incoming labeled arrows to the transaction) and then both locally replace it with updated values r^A :=(10, 51) and r^B :=(10, 51) (depicted by outgoing labeled arrows). Each replica then propagates the locally updated row to the other replica (depicted by dashed arrows), where the written value at `B` supersedes value written at `A`, leaving the database with the final state of r_0 :=(10, 51).

```
1  public void payment(int c_id)
2          throws CustNotFoundException{
3   stmt = prepareStatement("SELECT C_PAY_CNT
         FROM CUST WHERE C_ID=?");
4   stmt.setInt(1, c_id);
5   rs = stmt.executeQuery();
6   if(!rs.next())
7    throw new CustNotFoundException(c_id);
8   int c_pay_cnt = rs.getInt("C_PAY_CNT");
9   c_pay_cnt++;
10  stmt = prepareStatement("UPDATE CUST SET
        C_PAY_CNT=? WHERE C_ID=?");
11  stmt.setInt(1, c_pay_cnt);
12  stmt.setInt(2, c_id);
13  stmt.executeUpdate();
14 }
```

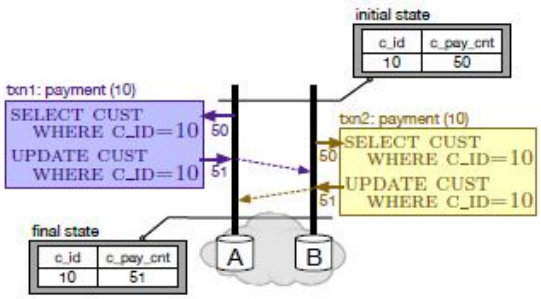


Figure 24: A transaction from TPC-C benchmark in Java (eft) and an anomalous execution (right)

Because the above scenario is not equivalent to any sequential execution of the two transaction instances, this behavior is properly classified as a *serializability anomaly*. Unfortunately, buggy executions like this are often permitted by modern cloud databases which drop support for atomic and isolated transactions in favor of fault tolerance, scalability, and availability. This foists a significant challenge onto OLTP application developers, requiring them to detect the potential anomalies that can occur in a particular application and validate them against different database systems in order to diagnose and remedy undesired behaviors.

Developing a testing framework that discovers anomalies of this kind is challenging, however. Part of the problem is due to the large set of possible interleaved executions that are possible. For example, the serializability anomaly described above is crucially dependent on the exact order in which operations are transmitted to different replicas and occurs only when the same customer (out of 30,000 customers) is accessed by both instances. Combined with the possibility that operations within a single transaction instance can also be unreachable in some program paths, and that the execution paths taken are highly dependent on the initial database state, the chance of randomly encountering conflicts of this kind becomes even smaller.

In addition to the above challenges, the high computational cost of detecting serializability anomalies at runtime [38] means that practical testing methods often leverage a set of user-provided application-level invariants and assertions to check for serializability failure-induced violations. But, these assertions are often underspecified. For example, the anomaly depicted in Figure 24, which most developers would classify as a bug, does not directly violate any of the twelve officially specified invariants of the TPC-C benchmark.

CLOTHO. To overcome these challenges, we have developed a principled test construction and administration framework called CLOTHO that targets database applications intended to be deployed on weakly-consistent storage environments. Our approach reasons over *abstract executions* of a database application. Serializability anomalies manifest as cycles in such executions. CLOTHO translates the discovered abstract anomalies back to concrete executions which can be then automatically replayed and validated. These abstract executions are generated using the verification methodology described in Section 2.2.

The test configuration details produced by CLOTHO include:

1. Concrete transaction instances and parameters,
2. Concurrent execution schedules,
3. Network partitioning throughout the execution,
4. Session management, and
5. Initial database state.

Our test administration framework allows arbitrary distributed or centralized databases to be plugged into its managed containers. Given a set of test configurations, CLOTHO automatically executes multiple application instances according to each test configuration, in order to effectively manifest and validate the intended serializability anomaly against a variety of actual database systems.

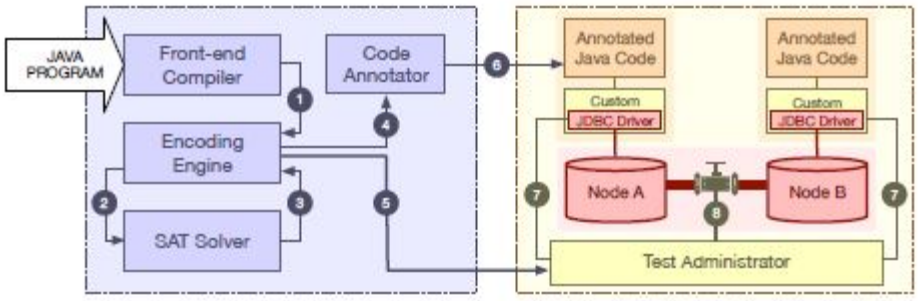


Figure 25: CLOTHO pipeline

Figure 25 presents an overview of CLOTHO's design. CLOTHO's static analysis backend is based on an intermediate abstract representation of database programs, automatically generated from input Java source code via a front-end compiler. The abstract program representation is passed to an encoding engine (❶) that constructs FOL formulae that capture the necessary conditions under which a dependency cycle forms over instances of database operations. This fine-grained Boolean Satisfiability (SAT) representation of the problem is then passed to an off-the-shelf theorem prover (❷); responses with a satisfying solution (❸) are recorded and re-encoded into the context to be passed to the solver again with the intention of finding another anomaly. This iterative approach allows for a complete search of the space of anomalies within a configurable cycle-detection length. Once the bounded space is completely covered (i.e., there are no more satisfying solutions), test configuration files (❺) are generated from the collected abstract anomalies that provide details about concrete executions that can potentially manifest the intended anomaly. Additionally, the source code of the program is passed through a code annotator (❹,❻) which returns multiple instances of the source code including concrete function parameters and annotations on all database access operations.

```
@Parameters(10)
public void payment ... {
  ...
  @Sched(node="A", order=1)
  rs = stmt.executeQuery();
  ...
  @Sched(node="A", order=2)
  stmt.executeUpdate();
}
```

A1_Ins1.java

```
@Parameters(10)
public void payment ... {
  ...
  @Sched(node="B", order=1)
  rs = stmt.executeQuery();
  ...
  @Sched(node="B", order=2)
  stmt.executeUpdate();
}
```

A1_Ins2.java

```
# initialize:
INSERT INTO
  CUST(c_id,c_pay_cnt)
  VALUES (10,50);
# schedule:
@T1@partitions(A,B): Ins1-O1
@T2@partitions(A,B): Ins2-O1
@T3@partitions(A,B): Ins1-O2
@T4@partitions(A,B): Ins2-O2
```

A1.conf

Figure 26: Annotated code of two transaction instances and the test configuration file

For example, `A1.conf` in Figure 26 is the configuration file that specifies the initial state of the database needed to manifest the previously discussed anomaly on the `CUST` table. It also specifies the execution order of operations from the two txn*payment* transaction instances and the network status at each logical step (`T1` to `T4`). The configuration is completed with annotated source code that specifies the input parameters to the txn*payment* transaction and marks the database operations with their relative local orders (A1_Ins1.java and A1_Ins2.java).

`CLOTHO` additionally offers a fully automated test administration framework, which consists of centrally managed wrappers for arbitrary database drivers which can temporarily block database access requests in order to enforce a specific execution order (❼). Similarly, database replicas are deployed in managed containers, communication among which can be throttled (❽) in order to induce temporary network partitionings. This enables `CLOTHO` to efficiently administer a wide range of statically constructed potential serializability anomalies, allowing developers to witness and study undesired application behaviors that frequently occur in the wild but which are extremely difficult to catch using existing testing frameworks.

5 CONCLUSIONS

The Quelea project was centered on the development of language abstractions, database technologies, compilers, program analyses, verification techniques, and runtime systems for building scalable applications intended to operate in geo-distributed environments where issues of data consistency and visibility are significant and subtle. The most significant contributions of this project are:

1. program logics for reasoning about geo-replicated distributed data stores and distributed transactions;

2. automated verification techniques for proving the safety of applications executing in such environments;

3. model-checking techniques that increase the assurance of applications that are equipped with sophisticated safety invariants;

4. language design principles that yield correct-by-construction distributed applications; and,

5. testing frameworks that can automatically discover invariant violations with greater specificity than random testing approaches collectively validate the thesis underlying the proposed effort.

All tools built to explore and validate these ideas have been published as open-source artifacts. The project has produced a number of publications in top-tier conferences and journals [25–28, 32–34, 42, 43].

6 Bibliography

[1] Atul Adya, Barbara Liskov, and Patrick E. O'Neil. Generalized isolation level definitions. In *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*, pages 67–78, 2000.

[2] Amazon Simple Queue Service.

[3] Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. Specification and complexity of collaborative text editing. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 259–268, 2016.

[4] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly Available Transactions: Virtues and Limitations. *PVLDB*, 7(3):181–192,2013.

[5] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination Avoidance in Database Systems. *Proc. VLDB Endow.*, 8(3):185–196, November 2014.

[6] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. HAT, Not CAP: Towards Highly Available Transactions. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 24–24, Berkeley, CA, USA, 2013. USENIX Association.

[7] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM.

[8] Giovanni Bernardi and Alexey Gotsman. Robustness against consistency models with atomic visibility. In *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, pages 7:1–7:15, 2016.

[9] Philip A. Bernstein and Sudipto Das. Rethinking Eventual Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 923–928, New York, NY, USA, 2013. ACM.

[10] Philip A. Bernstein and Nathan Goodman. Multiversion Concurrency Control - Theory and Algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, December 1983.

[11] Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. On verifying causal consistency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 626–638, New York, NY, USA, 2017. ACM.

[12] Eric Brewer. Towards Robust Distributed Systems (Invited Talk),2000.

[13] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated Data Types: Specification, Verification, Optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 271–284, New York, NY, USA, 2014. ACM.

[14] Sebastian Burckhardt, Daan Leijen, Manuel Fähndrich, and Mooly Sagiv. Eventually consistent transactions. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*,

pages 67–86, 2012.

[15] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. A framework for transactional consistency models with atomic visibility. In *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*, pages 58–71, 2015.

[16] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in a Partitioned Network: A Survey. *ACM Comput. Surv.*, 17(3):341–370, September 1985.

[17] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM*, 19(11):624–633, November 1976.

[18] Peter Gammie, Antony L. Hosking, and Kai Engelhardt. Relaxing safely: Verified on-the-fly garbage collection for x86-tso. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 99–109, New York, NY, USA, 2015. ACM.

[19] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[20] Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, 33(2):51–59, June 2002.

[21] Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. Verifying strong eventual consistency in distributed systems. *PACMPL*, 1(OOPSLA):109:1–109:28, 2017.

[22] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. '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 2016, pages 371–384, New York, NY, USA, 2016. ACM.

[23] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Data Base. pages 365–394, 1976.

[24] Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. Automated and Modular Refinement Reasoning for Concurrent Programs. In *Computer Aided Verification: 27th International Conference*, pages 449–465. Springer International Publishing, 2015.

[25] Gowtham Kaki, Kapil Earanky, K. C. Sivaramakrishnan, and Suresh Jagannathan. Safe Replication through Bounded Concurrency Verification. *PACMPL*, 2(OOPSLA):164:1–164:27, 2018.

[26] Gowtham Kaki, Kartik Nagar, Mahsa Najafzadeh, and Suresh Jagannathan. Alone Together: Composi- tional Reasoning and Inference for Weak Isolation. *PACMPL*, 2(POPL):27:1–27:34, 2018.

[27] Gowtham Kaki, Swarn Priya, K. C. Sivaramakrishnan, and Suresh Jagannathan. Mergeable Replicated Data Types. *PACMPL*, 3(OOPSLA):154:1–154:29, 2019.

[28] Gowtham Kaki, K. C. Sivaramakrishnan, and Suresh Jagannathan. Version Control Is for Your Data Too. In *3rd Summit on Advances in Programming Languages, SNAPL 2019, May 16-17, 2019, Providence, RI, USA*, pages 8:1–8:18, 2019.

[29] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Operating Systems Review*, 44(2):35–40, April 2010.

[30] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the*

23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011, pages 401–416, 2011.

[31] Transaction Isolation Levels, 2016. Accessed: 2016-07-1 10:00:00.

[32] Kartik Nagar and Suresh Jagannathan. Automated Detection of Serializability Violations Under Weak Consistency. In *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China*, pages 41:1–41:18, 2018.

[33] Kartik Nagar and Suresh Jagannathan. Automated Parameterized Verification of CRDTs. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*, pages 459–477, 2019.

[34] Mahsa Najafzadeh and Suresh Jagannathan. Geo-Replication Models. In *Encyclopedia of Big Data Technologies*. 2019.

[35] Data Concurrency and Consistency, 2016. Accessed: 2016-07-1 10:00:00.

[36] Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. Reducing Liveness to Safety in First-order Logic. *Proc. ACM Program. Lang.*, 2(POPL):26:1–26:33, December 2017.

[37] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety Verification by Interactive Generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 614–630, New York, NY, USA, 2016. ACM.

[38] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, October 1979.

[39] Transaction Isolation, 2016. Accessed: 2016-07-1 10:00:00.

[40] PPX extension points, 2017. Accessed: 2017-01-04 10:12:00.

[41] Nuno M. Preguiça, Carlos Baquero, and Marc Shapiro. Conflict-free replicated data types (crdts). *CoRR*, abs/1805.06358, 2018.

[42] Kia Rahmani, Gowtham Kaki, and Suresh Jagannathan. Fine-grained distributed consistency guarantees with effect orchestration. In *Proceedings of the 5th Workshop on the Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 6:1–6:5, 2018.

[43] Kia Rahmani, Kartik Nagar, Benjamin Delaware, and Suresh Jagannathan. CLOTHO: directed test generation for weakly consistent database systems. *PACMPL*, 3(OOPSLA):117:1–117:28, 2019.

[44] Rice University Bidding System, 2014. Accessed: 2014-11-4 13:21:00.

[45] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report RR-7506, INRIA, Inria – Centre Paris-Rocquencourt, 2011.

[46] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-Free Replicated Data Types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer Berlin Heidelberg, 2011.

[47] Dennis Shasha and Philippe Bonnet. *Database Tuning: Principles, Experiments, and Troubleshooting Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[48] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative Programming over Even- tually Consistent Data Stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 413–424, New York, NY, USA, 2015. ACM.

[49] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 385–400, 2011.

[50] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM.

[51] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, PDIS '94, pages 140–149, Washington, DC, USA, 1994. IEEE Computer Society.

[52] TPC-C Benchmark. http://www.tpc.org/tpc documents current versions/pdf/tpc-c v5.11.0.pdf. Online; Accessed 20 April 2018.

[53] Twitter clone on Cassandra, 2014. Accessed: 2014-11-4 13:21:00.

[54] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 357–368, New York, NY, USA, 2015. ACM.

7 List of Symbols, Abbreviations, and Acronyms

ACID	Atomicity, Consistency, Isolation, Durability
ANSI	American National Standards Institute
CC	Causal Consistency
CLOTHO	not an acronym, directed test generation for weakly consistent database systems
CRDT	Conflict-Free Replicated Data Type
diff	difference
DSL	Domain-Specific Language
EC	Eventual Consistency
FOL	First-Order Logic
LCA	Least Common Ancestor
MRDT	Mergeable Replicated Data Type
OCaml	Objective Categorical Abstract Machine Language
OLTP	Online Transaction Processing
PPX	not an acronym, OCaml preprocessors
PSI	Parallel Snapshot Isolation
QUARK	not an acronym, an end-to-end OCaml-based system for the creation of applications with mergeable, distributable state
Quelea	not an acronym, a programming system for eventually consistent distributed stores
RB	red-black
RC	Read Committed
RDT	Replicated Data Type
RUBiS	Rice University Bidding System
SAT	Boolean Satisfiability
SC	Strong Consistency
SEC	Strong Eventual Consistency
SI	Snapshot Isolation
SMT	Satisfiability Modulo Theories

SQL	Structured Query Language
SQS	Simple Queue Service
TPC-C	Transaction Processing Consortium benchmark C
VC	Verification Condition
Z3	not an acronym, a theorem prover from Microsoft Research