Static Serializability Analysis for Causal Consistency

Lucas Brutschy
Department of Computer Science
ETH Zurich
Switzerland
lucas.brutschy@inf.ethz.ch

Dimitar Dimitrov
Department of Computer Science
ETH Zurich
Switzerland
dimitar.dimitrov@inf.ethz.ch

Peter Müller
Department of Computer Science
ETH Zurich
Switzerland
peter.mueller@inf.ethz.ch

Martin Vechev
Department of Computer Science
ETH Zurich
Switzerland
martin.vechev@inf.ethz.ch

Abstract

Many distributed databases provide only weak consistency guarantees to reduce synchronization overhead and remain available under network partitions. However, this leads to behaviors not possible under stronger guarantees. Such behaviors can easily defy programmer intuition and lead to errors that are notoriously hard to detect.

In this paper, we propose a static analysis for detecting non-serializable behaviors of applications running on top of causally-consistent databases. Our technique is based on a novel, local serializability criterion and combines a generalization of graph-based techniques from the database literature with another, complementary analysis technique that encodes our serializability criterion into first-order logic formulas to be checked by an SMT solver. This analysis is more expensive yet more precise and produces concrete counterexamples.

We implemented our methods and evaluated them on a number of applications from two different domains: cloud-backed mobile applications and clients of a distributed database. Our experiments demonstrate that our analysis is able to detect harmful serializability violations while producing only a small number of false alarms.

CCS Concepts • **Software and its engineering → Software verification and validation**; • **Information systems** → *Parallel and distributed DBMSs*;

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5698-5/18/06…$15.00
https://doi.org/10.1145/3192366.3192415

Keywords causal consistency, static analysis, serializability

ACM Reference Format:
Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin Vechev. 2018. Static Serializability Analysis for Causal Consistency. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3192366.3192415

1 Introduction

Data stores that ensure strong consistency provide an intuitive guarantee to their client applications: if an application is correct in serial executions, it will remain correct in concurrent executions. However, following the CAP theorem [23], it is impossible for a data store to guarantee consistency and at the same time remain available under network partitions. The latter is required in many domains such as mobile applications, which may lose connection at any point, or in low-latency distributed databases that are replicated across continents. Many modern data stores therefore prioritize availability and partition-tolerance over consistency, that is, support only weak consistency models [2, 16, 19, 28].

Among weak consistency models, *causal consistency* has received increasing attention in terms of both theoretical analysis and practical implementations [2, 8, 18, 29, 30]. One reason behind this surge of interest is that causal consistency is the strongest model that can be guaranteed by the data store while remaining available under network partitions [6]. Causal consistency guarantees that if a query observes an update to the data store, then it also observes all *causal predecessors* of the update, that is, all updates that potentially may have caused the update in the first place. However, two *causally unrelated* events may be executed completely obliviously to each other, which frequently leads to surprising and non-serializable behaviors. Like many concurrency errors, these behaviors can be hard to trigger because their occurrence often depends on brittle timing effects.

This Work. We propose an end-to-end static analysis framework for client applications of causally-consistent databases. The analysis either proves the application is serializable or

detects a non-serializable behavior. Our framework is based on the following technical contributions.

Main Contributions.

- We propose a new serializability criterion inspired by our previous work [11]. Our criterion is equally precise in practice, but is more tailored to the static analysis setting (Section 4).
- Based on our criterion, we present an efficient serializability analysis that handles high-level data types and is more precise than prior characterizations for causal consistency (Section 6).
- We develop a logic-based analysis that is more precise than the above but also more expensive. It encodes our serializability criterion for a bounded number of sessions into decidable first-order formulas to be checked by an SMT solver. We provide a sufficient condition under which the analysis generalizes to an unbounded number of sessions (Section 7).
- We implemented[1] both analysis methods as well as a range of optimizations into a reusable back end framework called C^4. Our framework is designed to be independent of the data store (API) or programming language and can serve as a basis for analyzing applications in various domains.
- We provide an extensive evaluation of C^4 on applications from two different domains: a distributed database and a mobile framework. We show experimentally that C^4 effectively detects harmful serializability violations while producing only a small number of false alarms. Some of the violations are inherently difficult to detect via testing methods and are missed by a state-of-the-art dynamic analyzer (Section 9).

In our presentation, we focus on explaining the core results. The extended version of the paper [12] provides proofs, implementation details on C^4, and a classification of the bugs we found during our evaluation.

2 Overview

This section provides an informal overview of our technique and illustrates it on an example. Formal details are presented in subsequent sections.

Figure 1a shows two transactions operating on map `M` in a distributed data store: transaction `P` inserts value `v` at key `u` into `M` while transaction `G` retrieves the value at key `u`. Consider two concurrent runs of the program `P(x,y); G(z)` (for some arguments `x`, `y`, `z`). Figure 1c$_1$ shows a possible behavior of these runs (called sessions) on a weakly consistent data store. The diagram depicts sessions by outer gray boxes and transactions by inner boxes. The order of transactions inside a session is represented by so-edges (session order). In this execution, the left session writes value 1 at key `"A"` and then reads the initial value 0 from key `"B"`. Similarly, the right session inserts value 2 at key `"B"` and then reads the initial value 0 from key `"A"`. This execution is not serializable because, in any serial execution, one of the `get` operations would read the value written by a previous `put` operation instead of the initial value. This violation can arise under weak consistency when the sessions access different copies of the map (e.g., because they are connected to different replicas of the data store or operate on local caches).

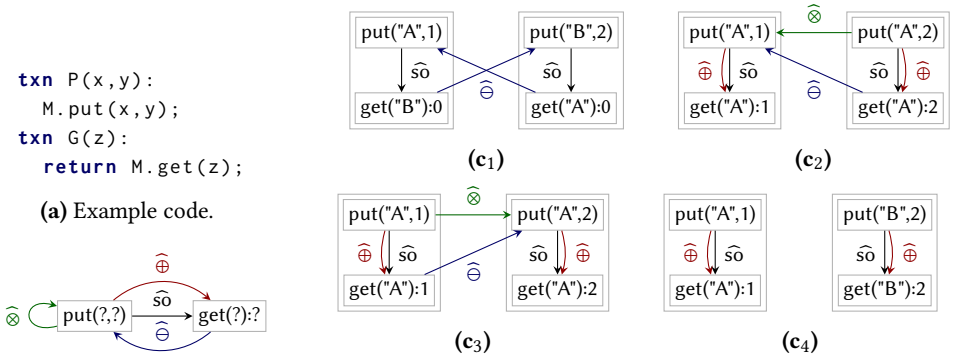
Dependency Serialization Graphs. Serializability violations in an execution can be detected by constructing a so-called *dependency serialization graph* (DSG) from the execution and checking if it contains cycles [1]. The nodes of a DSG are (executed) transactions which are connected by edges that reflect session order, dependencies $\oplus$ (a query depends on an update if the update affects the value returned by the query), anti-dependencies $\ominus$ (a query anti-depends on an update if the update is not visible to the query, but would affect its result if it were), and conflict-dependencies $\otimes$ (indicating the order in which conflicts are eventually resolved by the data store). We lift relations between operations to relations on transactions (e.g., $\oplus$ becomes $\widehat{\oplus}$), which is what we show in the figure. The graph in Figure 1c$_1$ is in fact a DSG; the cycle indicates that this execution is not serializable. The other three graphs in Figure 1c show three other possible DSGs of our example program. These DSGs do not contain cycles, that is, the represented executions are serializable.

Local Serializability Criterion. Our goal is to devise an analyzer that proves a DSG is acyclic for *any possible execution* of a given program, thus proving the program is serializable. Towards this, we compute an *abstraction* of all possible (potentially unboundedly many) concrete DSGs and then check a specific criterion on this abstraction.

To define this criterion, we build on our previous work [11], which supports high-level datatypes by leveraging algebraic properties of operations (e.g., commutativity and absorption) to define dependencies. However, our earlier criterion requires checking an entire DSG, which is problematic for static analysis as the graphs can be of unbounded size. We address this problem by proposing a novel criterion that is *local* in the following sense: removing a node which is not part of a cycle (together with its adjacent edges) cannot make the cycle infeasible. Our local criterion allows the analysis to consider only those subsets of events in which a minimal violation could be found. If their DSG does not contain cycles, no DSG of the program will contain cycles.

Static Serialization Graphs. A well-known approach to static serializability checking in the database literature [22] is to summarize all possible (concrete) DSGs for a program in a *static serialization graph* (SSG). An SSG contains one node for every syntactic transaction in the program; there is an edge between two nodes in the SSG if there *may be* an

[1] Source code available at: http://ecracer.inf.ethz.ch/



(a) Example code.

(b) The corresponding SSG.

(c) Possible DSGs of our program.

Figure 1. Figure 1a shows a simple program, which is not serializable in general. However, it is serializable if x in P and z in G are the same in the same session. Figures 1c$_1$ to 1c$_4$ show four possible dependency serialization graphs (DSGs) for the program. Figure 1b shows a static serializability graph (SSG), which summarizes all possible DSGs of the program.

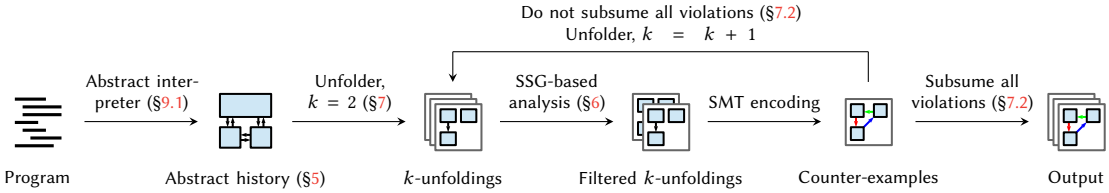


Figure 2. Flow overview of the C^4 analysis framework.

edge between the corresponding transactions in any DSG. A program is serializable if its SSG is acyclic. The cycles in the SSG of our program, shown in Figure 1b, correctly indicate that it is not serializable.

However, cycle detection in SSGs can be imprecise and lead to many false alarms as SSGs do not capture relevant semantic properties of the programs they abstract. Assume for instance the keys in all runs of our program are always the same. Then the program is serializable as all possible executions have a DSG as in Figure 1c$_2$ or Figure 1c$_3$. However, the SSG in Figure 1b cannot capture this semantic information and will contain infeasible cycles.

To recover precision, we propose a novel characterization of cycles in SSGs that exploits the semantics of arbitrary data types. For example, we can use the insight that any cycle in a DSG (under certain restrictions) must contain two updates that do *not* overwrite each other. We lift this criterion to SSGs in order to determine whether cycles in SSGs are feasible. Under the assumption that all keys used as an argument to put are the same, the SSG from Figure 1b does not contain problematic cycles since any two updates do overwrite each other. Consequently, our analysis will not report a false alarm for this example. Our characterization of cycles in the SSG is the first to handle high-level operations and is more precise than previous characterizations for causal consistency.

Logical Serializability Checking. Cycle detection in SSGs is practically useful because it tends to be very efficient. However, even with our new characterization of cycles, it can produce false alarms in common scenarios. For instance, assume the keys in our example are always the same *within one session* but may vary between sessions. Then the program only produces serializable behaviors: Figures 1c$_2$ to 1c$_4$ are possible, but Figure 1c$_1$ is not. In this scenario, our characterization of cycles in SSGs does not prevent infeasible cycles; it is now possible to have cycles with two updates that do not overwrite each other because they use different keys.

To capture more semantic information than SSGs, we encode a precise abstraction of a program's DSGs into logical formulas to be checked by SMT solvers. In contrast to the SSG approach, this encoding lets us determine whether edges can *coexist* in the same execution; for instance, the $\widehat{\ominus}$-edge in Figure 1c$_2$ can never appear in the same DSG as the $\widehat{\ominus}$-edge in Figure 1c$_3$ and, thus, cycles including both edges are infeasible. Such an encoding also lets us precisely reflect control-flow between operations to eliminate cycles that arise only with infeasible control-flow paths, and model data store operations that create records with guaranteed unique identities. Both properties are important in practice.

Small-Model Property and Generalization. The logical encoding is feasible only because we split the problem into a series of sub-problems that satisfy a *small-model property*: if

the logical encoding of each of the sub-problems has a model then this model is of bounded size, making each sub-problem efficiently checkable. We show that a bounded number of such sub-problems, called the k-unfoldings, is sufficient to model the serializability problem for a fixed number of sessions k. We generalize our technique to an arbitrary number of sessions by providing a sufficient condition that guarantees that any serializability violation in a program can be detected by considering at most k sessions.

Static Analysis Framework. We integrated all components described above into an end-to-end static analysis framework called C^4, illustrated in Figure 2. C^4 infers the abstract history of the program, which represents all possible ways it may interact with the data store. C^4 then checks the abstract history iteratively for serializability violations that involve at most k sessions. For each k, it computes all k-unfoldings of the abstract history and applies the fast SSG-based analysis to each of them. To reduce the number of false alarms, C^4 applies the precise SMT-based analysis to those k-unfoldings whose SSG indicates a potential serializability violation. The SMT-based analysis produces a counter-example for each detected serializability violation. This process is repeated for increasing values of k until either we can generalize from k to an arbitrary number of sessions as explained above (in this case we have found *all* serializability violations) or until a time-out occurs (in this case we have found all violations that span up to k sessions).

C^4 is a back end that can be used by various front ends. We implemented two such analyses, one for TouchDevelop [16, 34] and one for Cassandra/Java [28] (discussed later).

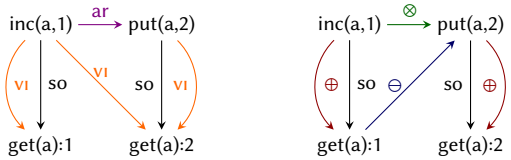
3 Formal Model

We begin with the data store model that we use to frame our analysis. The model is fairly standard (see, e.g., [10, 17]) and closest in exposition to [11]. We consider a store accessed via a fixed set of update and query operations:

1. **Updates** modify the store but have neither preconditions nor return values; examples include storing a value in a record, adding an element to a set, or incrementing a counter.
2. **Queries** do not modify the store but return a value to the client, e.g., the value of a record, the size of a set, or the value of a counter.

An execution of a single operation is called an *event*: formally, a tuple $m(a_1, \ldots a_{n-1}) : a_n$ tagged with a unique identifier. Here, m is an operation, $a_1, \ldots, a_{n-1}$ are concrete arguments, and a_n is an optional return value. Analogously to operations, events come as either updates $u \in \mathsf{U}$ or queries $q \in \mathsf{Q}$. As standard, we build upon the operations' sequential semantics, which we assume is specified as a prefix-closed set of event sequences. We call these sequences *legal*, and we say that an event e in a sequence $\alpha e \beta$ is legal if the prefix αe is legal.

We model concurrent executions as histories (Figure 3a). A *history* $H = (\mathsf{Ev}, \mathsf{so}, \mathsf{Tx})$ consists of: a finite set of events



(a) A history with a schedule. We omit superfluous ar-edges. **(b)** Dependency serialization graph for the given schedule.

Figure 3. An example history and schedule and its DSG.

Ev; a session order $\mathsf{so} \subseteq \mathsf{Ev} \times \mathsf{Ev}$ whose connected components are all chains, called *sessions*; a partition $\mathsf{Tx} \subseteq \mathcal{P}(\mathsf{Ev})$ of the sessions into contiguous blocks, called *transactions*. In order to provide sensible guarantees, the store does not permit arbitrary histories but only those that possess a suitable schedule. A *schedule* $S = (\mathsf{vi}, \mathsf{ar})$, consists of: a strict total order $\mathsf{ar} \subseteq \mathsf{Ev} \times \mathsf{Ev}$, called the *arbitration* order, which indicates the logical execution order of events; a relation $\mathsf{vi} \subseteq \mathsf{ar}$, called the *visibility* order, which indicates the events visible to any given other event (thus, determining the outcome of that event).

In this work, we require that a legal schedule satisfies three properties. First, it must ensure that each query's outcome is consistent with the updates it observes:

(S1) For every event $e \in \mathsf{Ev}$, ar restricted to $\mathsf{vi}^{-1}(e) \cup \{e\}$ forms a legal sequence according to the sequential semantics.

Second, it must respect *causal consistency* [14]. Visibility must be transitively closed, and moreover, each event must be visible to all subsequent events in the same session:

(S2) $\mathsf{vi} = (\mathsf{so} \cup \mathsf{vi})^+$

Third, it must ensure *atomic visibility* [7], stating that events on the same transaction never interleave with events from other transactions in the vi and ar orders:

(S3) For every pair of distinct transactions $s \neq t \in \mathsf{Tx}$, and for all events $\{e, f\} \subseteq s$ and $\{e', f'\} \subseteq t$:

$$e \xrightarrow{\mathsf{vi}} e' \iff f \xrightarrow{\mathsf{vi}} f' \qquad e \xrightarrow{\mathsf{ar}} e' \iff f \xrightarrow{\mathsf{ar}} f'.$$

A schedule is *serial* iff $\mathsf{vi} = \mathsf{ar}$. A history is *serializable* iff it possesses at least one serial schedule.

Algebraic Reasoning. When analyzing serializability we need to reason about legality **(S1)**. As common in the presence of high-level operations [37], our reasoning about legality is based on *algebraic* properties of events that can be used to show equivalences between sequences of events. We will employ two such properties: commutativity and absorption. Formally, sequences α and β are *equivalent* ($\alpha \equiv \beta$) if substituting one for the other in any sequence leaves its legality unchanged. Then, for any pair of events e, f

$$e \text{ and } f \textit{ commute} \iff ef \equiv fe$$
$$f \textit{ absorbs } e \iff ef \equiv f.$$

E.g., the update put(a, 2) and the query get(b):1 commute; put(a, 2) absorbs the update inc(a, 1) but not vice versa.

4 A Local Serializability Criterion

We now describe a new serializability criterion for weakly consistent data stores. Our criterion is inspired by our earlier work [11], but is local: removing a node from a DSG that is not part of a cycle (together with its adjacent edges) cannot make the cycle infeasible. Locality allows the analysis to consider only those subsets of events in which a minimal violation could be found. If their DSG does not contain cycles, no DSG of the program will contain cycles. Locality is implicit in earlier static analysis approaches, e.g. [9, 22], which focus on low-level reads and writes. Our work is the first local criterion that supports high-level operations.

4.1 Far Commutativity and Absorption

The serializability criterion of [11] is non-local due to its use of commutativity and absorption as defined in the previous section. These properties apply only to adjacent events and thus, inserting or removing unrelated events between two events potentially affects whether they commute or absorb each other and, as a result, whether the serializability criterion holds. We remove this non-locality by defining far versions of commutativity and absorption that apply to events far apart and, thus, are not affected by adding or removing unrelated intermediate events.

We first define the *far-absorption* relation $\triangleright$ on updates. In the plain (as opposed to far) version, a given update absorbs all the effects of the update immediately before it. The far version allows the absorbed update to be arbitrarily far away:

(R1) $\triangleright \subseteq \mathsf{U} \times \mathsf{U}$, and $u \triangleright v$ iff $u\beta v \equiv \beta v$ for all $\beta \subseteq \mathsf{U}$;

Now, we define the *far-commutativity* relation $\between$ from updates to queries. Our goal is to generalize the following use of plain commutativity: if a query q is legal and commutes with an update u immediately before it then the query remains legal if we remove u. To be able to remove u even if it is far away from q, we strengthen commutativity as follows:

(R2) $\between \subseteq \mathsf{U} \times \mathsf{Q}$, and $u \between q$ iff $uq \equiv qu$ and for all $v \in \mathsf{U}$,

$$uv \equiv vu \quad \text{or} \quad v \between q \quad \text{or} \quad u \triangleright v.$$

Here, the right-hand side use of $\between$ is coinductive: we seek the largest relation satisfying **(R2)**. We prove that this has the required effect in [12]. We extend the definition of $\between$ to all events by treating query-update pairs symmetrically, letting queries always far-commute, and using plain commutativity for updates.

Comparison to the Plain Versions. The plain and far versions of absorption and commutativity coincide for the prominent replicated data stores in use today. Differences can occur in the presence of some not widely supported operations such as cp(a, b), which copies the value of record a to record b. Now put(a, 2) no longer far-absorbs inc(a, 1) since:

$$\mathsf{inc(a,\ 1)\ cp(a,\ b)\ put(a,\ 2)} \not\equiv \mathsf{cp(a,\ b)\ put(a,2)}.$$

Similarly, put(a,2) no longer far-commutes with get(b):2, since cp(a,b) commutes with or absorbs neither of them.

4.2 Serializability Criterion

Our serializability criterion takes as input a history and a schedule and determines whether the history is serializable. The criterion belongs to a broad class of criteria based on dependence graphs [1]. The general idea is to compute a digraph of dependencies between the events (Figure 3b), and then interpret its arcs as ordering constraints. Any permutation of the events that satisfies these constraints is a legal schedule. To prove serializability, one lifts the constraints from events to transactions and checks whether these lifted constraints are satisfiable. That is, one collapses the events of each transaction to a single node, and then checks whether the resulting digraph is acyclic.

Given a history and a schedule, we build that digraph from a triple of relations, which in turn are built with the help of far-commutativity, far-absorption, and plain commutativity.

We say that a query *does not depend* on a visible update if the update far-commutes with the query, or if the update is far-absorbed by some intermediate visible update:

(D1) $\oplus \subseteq \mathsf{U} \times \mathsf{Q}$, and if $u \xrightarrow{\text{vi}} q$ and $(u, q) \notin \oplus$, then $u \between q$ or there exists some v such that $u \triangleright v$ and $u \xrightarrow{\text{ar}} v \xrightarrow{\text{vi}} q$.

The *dependencies* of a query are those visible updates for which the not-depends property does *not* hold. Intuitively, hiding a dependency from a query might affect the query outcome. For example, get(a):2 in Figure 3 depends on put(a, 2), but not on inc(a,1), which is absorbed by put.

Anti-dependencies are analogous for *invisible* updates:

(D2) $\ominus \subseteq \mathsf{Q} \times \mathsf{U}$, and if $u \not\xrightarrow{\text{vi}} q$ and $(q, u) \notin \ominus$ then $u \between q$ or there exists some v such that $u \triangleright v$ and $u \xrightarrow{\text{ar}} v \xrightarrow{\text{vi}} q$.

Intuitively, making a given anti-dependency visible might affect the query outcome. In Figure 3, the query get(a):1 anti-depends on the invisible update put(a, 2).

Finally, an update *does not conflict-depend* on an update arbitrated before it if the two commute plainly:

(D3) $\otimes \subseteq \mathsf{U} \times \mathsf{U}$, and if $u \xrightarrow{\text{ar}} v$ and $(u, v) \notin \otimes$ then $uv \equiv vu$.

Intuitively, arbitrating a conflict-dependency after the update might change the store state observed by a later query.

We now lift each of the relations $R \in \{\text{so}, \text{vi}, \text{ar}, \oplus, \ominus, \otimes\}$ to a relation $\widehat{R}$ on transactions in the following way:

$$(s, t) \in \widehat{R} \iff s \neq t \text{ and } (e, f) \in R \text{ for some } e \in s, f \in t.$$

The *dependency serialization graph* (DSG) is the multi-digraph that has the given history's transactions as nodes, and has an arc (s, t) labeled $\widehat{R}$ for any pair $(s, t) \in \widehat{R} \in \{\widehat{\text{so}}, \widehat{\oplus}, \widehat{\ominus}, \widehat{\otimes}\}$. As mentioned, each arc represents an ordering constraint on the transactions:

Theorem 1. *If a schedule of a history induces an acyclic DSG then the history is serializable.*

We prove the theorem in [12].

```
txn P(k,v):     txn I(k,v):
  M.put(k,v)      if M.get(k) < 10: M.inc(k,v)
```

Figure 4. A program with a put and a conditional increment.

Locality. We can now state precisely the locality property of our criterion: if we restrict a history and its schedule to any subset of events E and build the DSG anew, then none of the old dependencies in E disappear:

Theorem 2. *For any schedule* (vi, ar) *with dependence triple* $(\oplus, \ominus, \otimes)$*, the restriction* $(vi \restriction E, \mathsf{ar} \restriction E)$ *of the schedule to a subset* $E \subseteq \mathsf{Ev}$ *has a dependence triple* $(\oplus', \ominus', \otimes')$ *such that:*

$$\oplus' \supseteq \oplus \restriction E \qquad \ominus' \supseteq \ominus \restriction E \qquad \otimes' \supseteq \otimes \restriction E.$$

The theorem follows by simple case analysis. Importantly, if the DSG of the original schedule contains a cycle (that is, a serializability violation) then this cycle is also present in the DSG of the schedule restricted to that cycle's events.

5 Abstraction for Serializability

Our objective is to detect the presence of DSG cycles statically. The first step is to define an abstraction of all the histories that a given data store client may have. Later, we will analyze this abstraction to soundly detect cycles.

Abstract Events and Transactions. Given a client program, we abstract its concrete histories into one *abstract history* $\underline{H}$. Figure 4 shows an example program, and Figure 5 illustrates the abstract history of the program together with one of its concrete histories. Each syntactic invocation in the program corresponds to an *abstract event* $e \in \underline{\mathsf{Ev}}$ of the abstract history. The abstract event abstracts all events produced by that invocation. Thus, each program history comes with a mapping of its events into the abstract events of the abstract history. The abstract history also contains a partition of the abstract events into abstract transactions $t \in \underline{\mathsf{Tx}}$ according to the syntactic transactions that the events originate from.

Ordering between Abstract Elements. We track the control-flow within each transaction to over-approximate the session order so inside transactions. This materializes in the form of *abstract event order* $\underline{\mathsf{eo}}$, a binary relation between the abstract events. Figure 5 shows $\underline{\mathsf{eo}}$ as arcs inside the abstract transactions. Moreover, we over-approximate the session order outside transactions with the *abstract session order* $\underline{\mathsf{so}}$, a transitive relation between abstract transactions.

Invariants. To make the abstraction more precise, we infer simple invariants between pairs of abstract events related by the abstract event order $\underline{\mathsf{eo}}$. For example, in the `I` transaction in Figure 4, the `get` query and the `inc` update always use the same key. We express this as the logical formula $\mathsf{arg}_0^{\mathsf{src}} = \mathsf{arg}_0^{\mathsf{tgt}}$ attached to the $\underline{\mathsf{eo}}$ arc between the two abstract events.

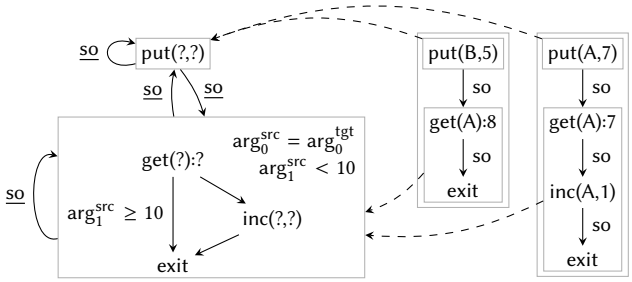


Figure 5. An abstract history (left) of the program in Figure 4, and a concretization (right). Dashed edges show the mapping from concrete transactions to abstract transactions.

The formula states that the 0th argument of the arc source equals the 0th argument of the arc target. More generally, the abstract history includes a map Inv from $\underline{\mathsf{eo}}$ to logical formulas over the variables in $\mathsf{arg}^{\mathsf{src}} \cup \mathsf{arg}^{\mathsf{tgt}}$.

Local and Global Constants. We also allow invariants over certain immutable data shared across concrete transactions. Web applications, for example, store a session identifier in the state of the web browser and transmit it with every request. We consider two types of data: session-local constants and global constants. We model these with corresponding sets Var_L and Var_G of variables that invariants can refer to.

Altogether. We gather the above in the following definition, where $\Phi(X)$ is a fragment of formulas over the variables X:

Definition 1. An *abstract history* is a tuple $\underline{H}$ consisting of

1. $\underline{\mathsf{Ev}} = \underline{U} \cup \underline{Q}$: a set of abstract events (updates and queries);
2. $\underline{\mathsf{Tx}}$: the set of abstract transactions;
3. $\underline{\mathsf{eo}} \subseteq \underline{\mathsf{Ev}} \times \underline{\mathsf{Ev}}$: the abstract event order;
4. $\underline{\mathsf{so}} \subseteq \underline{\mathsf{Tx}} \times \underline{\mathsf{Tx}}$: the abstract session order;
5. Var_L, Var_G: sets of session-local and global variables;
6. Inv: a mapping $\underline{\mathsf{eo}} \to \Phi(\mathsf{arg}^{\mathsf{src}} \cup \mathsf{arg}^{\mathsf{tgt}} \cup \mathsf{Var}_G \cup \mathsf{Var}_L)$.

We assume that every transaction $t \in \underline{\mathsf{Tx}}$ has unique entry and exit events $\mathsf{entry}[t]$, $\mathsf{exit}[t]$: the lone events in t having no predecessors and successors in $\underline{\mathsf{eo}}$, respectively.

Concretization. An abstract history $\underline{H}$ over-approximates the concrete histories of a given program, but it is consistent with a larger set of histories, namely, the concretizations $H \in \gamma(\underline{H})$. A history belongs to $\gamma(\underline{H})$ if it has a *concretization model*: a mapping from events to abstract events, and valuations of the Var_L and Var_G vars such that: (1) so-arcs map respectively to $\underline{\mathsf{eo}}$-arcs inside transactions and $\underline{\mathsf{so}}$ arcs outside transactions; (2) each invariant is satisfied by the corresponding pairs of concrete events. Note that a history in $\gamma(\underline{H})$ need not possess a schedule but it always possesses a *pre-schedule*: a schedule that may violate **(S1)**. We will also refer to these as the pre-schedules of the given abstract history itself. See [12] for a formal definition of concretizations.

	put(k', v')	get(k'):v'	size():n'
put(k, v)	$k \neq k'$ or $v = v'$	$k \neq k'$	false
get(k):v	$k \neq k'$	true	true
size():n	false	true	false

(a) Commutativity specification.

	put(k', v')
put(k, v)	$k = k'$

(b) Absorption specification.

Figure 6. Rewrite specification for a dictionary.

Rewrite Specification. To check serializability, one needs to have some knowledge about the operations available in the data store. We assume a *rewrite specification*, logical formulas that give sufficient conditions for commutativity and absorption between events:

Definition 2. An *rewrite specification* is a pair (com, abs) of families of logical formulas over arg^{src} and arg^{tgt}, each indexed by pairs of operations, such that for all events e, f:

$$\text{com}(\text{op}[e], \text{op}[f])(\text{arg}[e], \text{arg}[f]) \implies e \circlearrowright f$$
$$\text{abs}(\text{op}[e], \text{op}[f])(\text{arg}[e], \text{arg}[f]) \implies e \triangleright f.$$

Figure 6 shows a rewrite specification for a dictionary. We write $\neg\text{com}(e, f)$ if $\neg\text{com}(\text{op}[e], \text{op}[f])$ is satisfiable, where e, f are abstract events, and similarly for absorption abs.

6 A Fast Serializability Analysis

In this section, we present an efficient serializability analyzer based on abstract histories. As shown in Section 4, a history is serializable if it has at least *one* schedule with an acyclic DSG. In the following, we will instead check whether *all* schedules have an acyclic DSG (or equivalently, if there exists *any* schedule with a cyclic DSG). This simplifies the problem, as it is easier to find a cyclic DSG than to prove the absence of acyclic DSGs. Further, we are only aware of artificial examples where the answers to the two questions differ. The same approach was also implicitly followed by prior work [3, 9, 22]. Even then, the problem remains challenging because there are infinitely many histories in the concretization of most abstract histories.

The basic idea of the analysis introduced in this section is to lift the definition of DSG to abstract histories and detect cycles in this lifting. More precisely, we build a graph whose nodes are abstract transactions and where nodes are connected by an edge if there are two concrete transactions in any history in the concretization such that the two transactions are connected by an edge in any DSG of the history. We call this graph a *static serialization graph* (SSG).

Definition 3. Given an rewrite specification (com, abs), the *static dependence serialization graph* of an abstract history is an edge-labeled directed multigraph with vertices $\underline{\text{Tx}}$ and

1. an edge (s, t) labeled $\underline{\text{so}}$ for every $(s, t) \in \underline{\text{so}}$;
2. an edge (s, t) if $\exists e \in s, f \in t$ such that $\neg\text{com}(e, f)$
   - labeled $\oplus$ if $e \in \underline{\text{U}}$ and $f \in \underline{\text{Q}}$
   - labeled $\ominus$ if $e \in \underline{\text{Q}}$ and $f \in \underline{\text{U}}$
   - labeled $\otimes$ if $e \in \underline{\text{U}}$ and $f \in \underline{\text{U}}$

Intuitively, the satisfiability of $\neg\text{com}(\text{op}[e], \text{op}[f])$ is used here as a *necessary* condition for the existence of a dependency, anti-dependency, or conflict-dependency between any two concrete events, which are summarized by e and f, respectively. In previous work based on reads and writes, this condition was given by "e and f access the same location". Here, we use the locality of our criterion to avoid having to reason about intermediate events between e and f, which may introduce extra (anti-)dependencies.

The absence of cycles in an SSG is a *sufficient* condition for the absence of cycles in all DSGs, and thereby a sufficient condition for serializability. However, this condition alone is very imprecise, as most SSGs contain trivial cycles, such as the one we have seen in Figure 1b. Therefore, we show the following, stronger condition:

Theorem 3. *Let $\underline{H}$ be an abstract history and G be the DSG of a history in $\gamma(\underline{H})$. If there is a cycle in G then there is a closed walk in the SSG of $\underline{H}$ with the following properties:*

(SC1) *It contains at least two $\widehat{\ominus}$-edges, or at least one $\widehat{\ominus}$-edge and one $\widehat{\otimes}$-edge.*

(SC2) *At least one of the following conditions hold:*

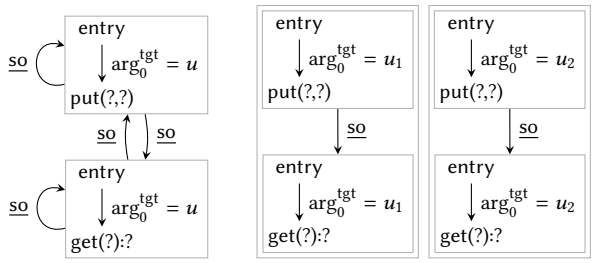
(SC2a) *It contains $u, v \in \underline{\text{U}}$ such that $\neg\text{abs}(u, v)$.*

(SC2b) *It contains $q \in \underline{Q}, u, v \in \underline{U}, e \in \underline{\text{Ev}}$ with $q \overset{\text{eo}^+}{\rightarrow} u$ and both $\neg\text{com}(u, e)$ and $\neg\text{com}(q, v)$.*

We explain the soundness of our algorithm and prove the theorem in [12]. The theorem lets us check serializability of abstract histories by (1) pre-computing which pairs of abstract events satisfy plain commutativity and plain absorption, (2) detecting strongly-connected components in the SSG, and (3) checking whether **(SC1)** and **(SC2)** hold for each component.

For example, consider the abstract history in Figure 7a, and assume for now that $u \in \text{Var}_G$, i.e., both sessions are guaranteed to use the same key. Then we can use **(SC2)** to decide that the program is completely serializable: put abstract events will always absorb each other (since they write the same key u), and there is no transaction that executes a query before an update, so both **(SC2a)** and **(SC2b)** are not satisfied for the only connected component. However, if instead, the sessions may use different keys (that is, $u \in \text{Var}_L$ as in the original abstract history) then we fail to show serializability of the program, as the two put abstract events may *not* absorb each other.

Since SSG-based cycle detection is efficient, but not always sufficiently precise, we employ it in a staged fashion. First, we find potential violations using the SSG-based analysis and then apply a more expensive algorithm on these potential violations (developed in the next section).



(a) Abstract history, where $u \in \mathrm{Var}_L$.

(b) One of the two-session unfoldings, where $u_1, u_2 \in \mathrm{Var}_G$.

Figure 7. An abstract history for the program in Figure 1a and one of its unfoldings. The history includes the invariant that within a session, all transactions access the same key u.

Algorithm 1 Serializability checking by unfolding.

```
1:  function CheckBounded(H, k, V)
2:      for H' ∈ Unfoldings(H, k) do
3:          if CyclePossible(H') ∧ ¬Subsumed(H', V) then
4:              if ∃m.m ⊨ φ_cyclic(H') then
5:                  V ← V ∪ {m}
6:      return V
7:  function Check(H)
8:      V ← ∅, k ← 2
9:      repeat
10:         V ← V ∪ CheckBounded(H, k, V)
11:         k ← k + 1
12:     until SubsumptionGeneralizes(H, k, V)
13:     return V
```

7 Serializability Analysis by Unfolding

SSGs offer a very fast serializability analysis but, as discussed in the overview, their precision is limited because they completely ignore the invariants in the given abstract history. Consequently, one also cannot use SSGs to find concrete violations of our criterion, i.e., concrete DSG cycles, and report these counter-examples back to the user. In this section, we will address these two important shortcomings.

The basic idea is to let an SMT solver reason about the pre-schedules of a given abstract history directly. We do that in a sequence of SMT queries that encode our serializability criterion together with some of the invariants present in the abstract history. Each query is designed so that its models describe concrete DSG cycles in pre-schedules that satisfy the encoded invariants. In this way, we report both, concrete violations and improve precision by ruling out false positives that do not satisfy the given invariants.

Managing Complexity. In order to manage the complexity of our SMT queries, we design them to have a *small-model property*: a reasonable bound on the size of the models that the solver needs to explore. However, even the smallest concrete DSG cycles may be larger than the abstract history because a single abstract event might abstract many events on the cycle. Therefore, there is in general no bound on the model size that holds across the whole serializability analysis. We solve this problem by subdividing the serializability check into smaller problems of more manageable complexity such that a small-model property holds for each of them.

For each $k = 2 \ldots \infty$, we consider the problem of finding concrete DSG cycles that span at most k sessions. We embed the set of these cycles in a finite sequence U_k of particularly nice abstract histories, which we call *unfoldings*:

(U1) Any minimal DSG cycle that spans at most k sessions maps one-to-one into a cycle C of the unfolding for at most k sessions.

With this property, we can simply detect cycles in each of the unfoldings. A key virtue of our unfoldings is that we can restrict our attention to concretizations for which a single abstract event abstracts a single concrete event:

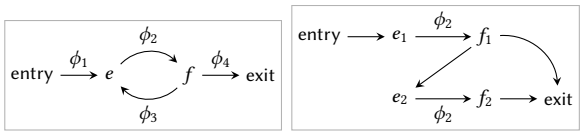
(U2) Each cycle C in **(U1)** is realized by a schedule of some concretization that maps one-to-one into the unfolding.

This is our small-model property: the size of a minimal DSG cycle for at most k sessions is at most that of the unfolding. We prove properties **(U1)** and **(U2)** in [12]. The locality of our criterion plays a crucial role in the proof.

Example. Consider again the program in Figure 1a under the assumption that all accesses within a session operate on the same key. Figure 7 shows an abstract history for that program, together with one of its unfoldings. The unfolding arranges copies of abstract transactions from the original abstract history into chains that represent *abstract sessions*. Figure 1 shows a sample of concrete histories that map one-to-one into the unfolding. The first one is unserializable, but it is not a concretization of the unfolding, and therefore, it would not be reported as a violation. The other three histories are serializable, and so, they also would not be reported.

Algorithm. We sketch the complete serializability check in Algorithm 1. The function CheckBounded$(\underline{H}, k, V)$ detects concrete DSG cycles that span k sessions. It iterates through all the k-session unfoldings of the given abstract history and accumulates the detected cycles in the set V. To reduce the calls to the SMT solver, the procedure makes two other checks as a pre-filter. First, it calls the fast SSG check to see if any cycles are possible at all. If so, it tests whether the cycles of the current unfolding are *subsumed* by cycles discovered previously. We consider one cycle to subsume another if its syntactic transactions are a subset of the other's ones. If no subsumption happens, the SMT solver is asked to find a new cycle as a model of the query ϕ_{cyclic}.

Function Check iteratively calls CheckBounded to detect cycles up to k sessions. To obtain a soundness guarantee for an unbounded number of sessions, we introduce a check that attempts to *prove* that we actually detected all cycles up to subsumption. That is, we attempt to prove via an SMT



(a) Transaction. **(b)** Unfolding.

Figure 8. Unfolding an abstract transaction.

query that each cycle on more than k sessions is subsumed by some cycle on at most k sessions. If this check succeeds, the iteration terminates. The algorithm can be combined with a time-out to ensure that it will terminate eventually, but that was never necessary in our experiments.

In the rest of the section, we describe in more details the unfolding procedure and the subsumption check. We describe the actual cycle query ϕ_{cyclic} in [12].

7.1 Unfolding Abstract Histories

Unfolding is founded on two properties. First, due to the locality of our criterion, DSG cycles are preserved under removal of events not lying on the cycle. That is, if C is a DSG cycle in a concrete history H then C remains a cycle in any restriction of H that includes the events in C. This property lets us remove events from the concrete histories that the unfoldings must abstract and remain sound. Second, each minimal DSG cycle is induced by at most two events per session. This property allows us to abstract events from at most two transactions per session. A similar property has been observed in the analysis of sequential consistency [31].

General Structure. The k-session unfoldings $\underline{H}' \in U_k[\underline{H}]$ of a given abstract history $\underline{H}$ are *acyclic* abstract histories organized into k abstract sessions. Each abstract session is constructed by selecting one abstract transaction or a pair of abstract transactions linked by $\underline{\text{so}}$ from the original abstract history. The unfolding places unfolded copies of these abstract transactions in their corresponding sessions and links them with $\underline{\text{so}}'$ in the indicated order. Transactions with an acyclic abstract event order $\underline{\text{eo}}$ unfold to themselves, just like in Figure 7b.

Unfolding of Transactions. Unfolding of transactions is necessary to ensure that the DSG cycles of $\underline{H}$ can be detected in the small one-to-one concretizations of its unfoldings as postulated in condition **(U2)**. We unfold each non-trivial strongly connected component (SCC) of the abstract event order $\underline{\text{eo}}$ independently of the rest. An example of unfolding a single component is shown in Figure 8. The goal is to make the SCC acyclic while keeping it an abstraction of each pair of events that might be part of a minimal cycle. To do that, we copy the events in the SCC twice and then reinsert back as much of the control-flow and the invariants as possible.

Definition 4. The unfolding of an SCC $V \subseteq t$ of an abstract transaction t involves two disjoint copies V_1, V_2 of V with

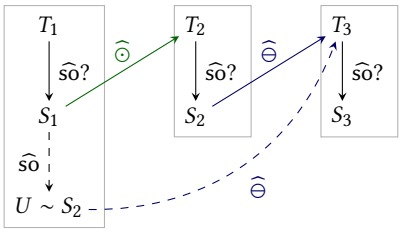


Figure 9. Short-cutting for $k = 2$. Here, $\widehat{\odot} = \widehat{\oplus} \cup \widehat{\ominus} \cup \widehat{\otimes}$.

corresponding inclusion maps $i_1 : V \hookrightarrow V_1$, $i_2 : V \hookrightarrow V_2$ as well as the set E of edges incident to vertices in V. It is defined as follows, where $h_1 \times h_2$ is the map $(x, y) \mapsto (h_1(x), h_2(y))$:

Edge types:

$$I \subseteq E \text{ — incoming edges } (\underline{\text{Ev}} \setminus V \to V)$$
$$O \subseteq E \text{ — outgoing edges } (V \to \underline{\text{Ev}} \setminus V)$$
$$B \subseteq E \text{ — back edges in any DFS of } V$$
$$R \subseteq E \text{ — the remaining edges in } E.$$

Unfolding: Here 1 denotes the identity $V \to V$ and A_s, A_t denote source and target vertex sets of any edge set A:

$$\underline{\text{Ev}}' = (\underline{\text{Ev}} \setminus V) \cup V_1 \cup V_2$$
$$\underline{\text{eo}}' = (\underline{\text{eo}} \setminus E) \cup I' \cup O' \cup B' \cup R'$$
$$I' = (1 \times i_1)[I \cup I_s \times B_t]$$
$$B' = (i_1 \times i_2)[B_s \times B_t]$$
$$O' = (i_1 \times 1)[O] \cup (i_2 \times 1)[O \cup (B_s \times O_t)]$$
$$R' = (i_1 \times i_1)[R] \cup (i_2 \times i_2)[R].$$

Invariants:

$$\mathsf{Inv}' \restriction (I' \cup O' \cup B') = \top$$
$$\mathsf{Inv}' \restriction R' = (\mathsf{Inv} \restriction R) \circ \left[(i_1 \times i_1) \cup (i_2 \times i_2)\right]^{-1}.$$

7.2 From K to Any: Generalizing Results

After each iteration of CheckBounded, we have inferred a set V of DSG cycles that subsume all DSG cycles that span at most k sessions. To generalize this result to an arbitrary number of sessions, we check whether this set V subsumes all DSG cycles spanning *any* number of sessions. This is implied if each cycle C that spans $l > k$ sessions is subsumed by (a) a cycle in V or (b) a cycle that spans $< l$ sessions.

We sketch the check briefly. Instead of checking (a) and (b) for all cycles C, which may be of unbounded size, we check a sufficient condition for all possible DSG paths P containing an anti-dependency and spanning exactly $k + 1$ sessions: every cycle C must contain such a segment. P is schematically shown for the case $k = 2$ in Figure 9. If some cycle in V subsumes the segment P, it also subsumes C, fulfilling (a). Otherwise, we try to show that (b) by checking whether every history that admits P transforms into a history that admits a segment that short-cuts P, skipping some sessions.

To show the existence of such a short-cut, we use the fact that most abstract transactions can be instantiated on any session. For example, for the segment in Figure 9, we try to instantiate the abstract transaction of S_2 at the end of session 1, in such a way that it forms an anti-dependency with T_3 to create a new segment Q that short-cuts session 2. This way, $C - P + Q$ is a cycle that subsumes the cycle C and spans $l - 1$ sessions as required.

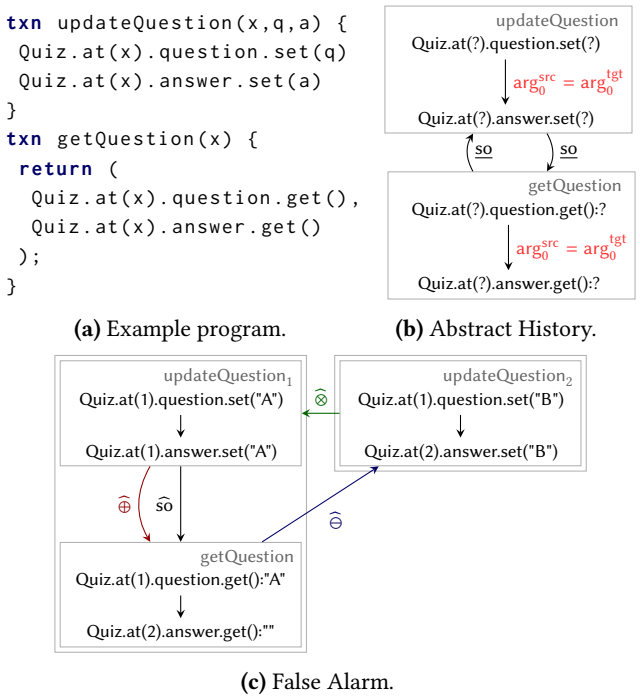
We can automate this check for all possible segments P again based on unfoldings: every segment on $k + 1$ sessions is a model of the logical encoding of a $(k + 1)$-unfolding. After filtering all $(k + 1)$-unfoldings for which all models are subsumed by a cycle in V, the above check can be formulated as a with a suitable SMT-query. If we manage to show that all path segments spanning $k + 1$ are either subsumed or can be cut short, we can conclude that V is a complete set of violations, subsuming all possible DSG cycles. We give more details about the query and the procedure in [12].

8 Reducing False Positives for Real-World Scenarios

In the previous sections, we provided a general formal method to statically check for serializability of a given application. In the first part of this section, we show how to instantiate our method in order to achieve precision on real-world examples (equality of arguments and control-flow). In the second part, we extend our method with asymmetric commutativity and uniqueness information to further increase precision. More details and further optimizations are described in [12]. All examples in this section are fragments of real applications and false alarms that we encountered; we omit many details and simplify the violations to help the exposition.

Using Equality of Arguments. In Figure 10a, transactions updateQuestion and getQuestion access two fields of a given row x, one writing to them, and the other reading from them. The important invariant here is that, inside a transaction, both accesses happen on the same row, even though rows might differ from one transaction to another. If we do not infer that both set-operations and both get-operations access the same row, we will detect the false alarm seen in Figure 10c. Here, updateQuestion_1 conflicts with updateQuestion_2 but does not completely absorb its effect, which leads to an anti-dependency from getQuestion to updateQuestion_2, and in turn, to a cycle. To avoid such false alarms we track equalities between local variables, and add them to the invariants for the SMT-based check from Section 7 (see also [12]). Here, the inferred equalities shown in red in Figure 10b are sufficient to prevent the false alarm.

Control-Flow. Figure 11a shows a fragment of a Twitter-like application that uses a contains query to check the existence of a key, a get query to retrieve a record at a key, and an add update to add a value to a set-valued field of a record.



(a) Example program. **(b)** Abstract History.

(c) False Alarm.

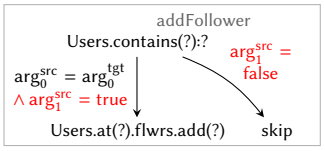
Figure 10. An example of a false alarm caused by missing equalities.

The addFollower transaction adds a follower n2 to a given user. Data stores typically create a record upon modification if the record does not exists. That is why the transaction guards against implicit creation by checking for existence before adding the follower. Since add updates commute, the transaction is serializable under the assumption of atomic visibility. However, if we ignore the control-flow between events, then we cannot rule out the false alarm in Figure 11c. There, two instances of addFollowers implicitly create the same user A while first observing that such a record does not yet exist. Since the contains query does not (far-)commute with creation, two anti-dependencies lead to a cycle. We therefore instantiate our static analysis to infer constraints on the control flow between abstract events (more details in [12]). For example, we infer the constraints shown in red in Figure 11b. With these extra constraints, the history in Figure 11c is not a concretization of the abstract history to the left, and our analysis will not report the false alarm.

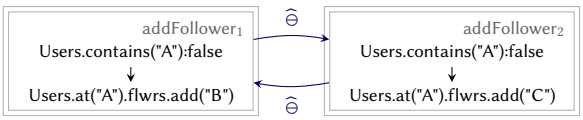
Asymmetric Commutativity. Control-flow constraints did eliminate the false alarm in Figure 11a, but the static analysis can still report another one, which is a feasible serializable execution. Consider a variation where both contains queries return true. Because of the implicit record creation semantics, the contains and the add operations do not commute, e.g., Users.at("A").flwrs.add("A") Users.contains("A"):true is legal but becomes illegal when swapped in case the record "A" was inserted earlier. This leads to anti-dependency edges similar to the ones in Figure 11c. To address this fundamental

```
txn addFollower(n1,n2) {
 if (Users.contains(n1))
  Users.at(n1).
   flwrs.add(n2);
}
```

(a) Example program.



(b) Abstract History.



(c) False Alarm.

Figure 11. An example of a false alarm caused by missing control-flow constraints.

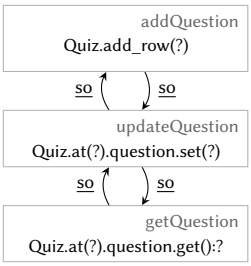
limitation of commutativity, we use an asymmetric version where contains("A"):true can always be moved to the right of an implicit creation operation. That is, in the paradoxical situation where the record "A" existed before its creation, it will also exist after the creation. We do not need an anti-dependency edge here since swapping the two operations does not affect the query result. In our experiments, we computed anti-dependencies using this asymmetric notion. We have not proved the soundness of this approach but confirmed manually that all eliminated alarms were indeed false alarms. We consider it important future work to extend our formal model to asymmetric commutativity.

Fresh Unique Values. Weakly-consistent data stores typically do not provide an efficient way to atomically check whether a record exists before inserting a record. Therefore, creating a record with a combined create/write operation as discussed in the previous subsection might accidentally overwrite an already existing record. To avoid this problem, most data stores provide a way to generate new records, which are guaranteed to have a fresh identity. This is akin to dynamic memory allocation in shared memory environments. For example, TouchDevelop provides the operation add_row, which adds a fresh row with a unique identity to a table; Cassandra can be instructed to generate a fresh key using the uuid() value.

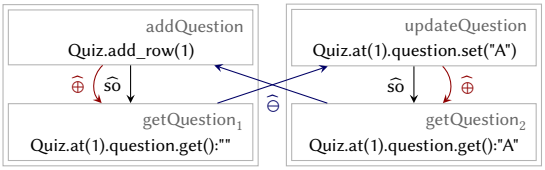
Figure 12a contains three transactions, one creating a new row in a table, one setting a field of a row, and one reading the value of the row. Figure 12b shows the corresponding abstract history . Our baseline analysis will report the violations shown in Figure 12c. A row is created and accessed in the left session, while the same row is accessed twice in the right session. The assumption that the identity of the row is fresh and unique implies that the only way (assuming no side-channels) for the right session to learn about the existence of the row is to *observe* its creation. However, $getQuestion_2$ reads the created row without observing its creation, which shows that this is a false alarm.

```
txn addQuestion() {
 return Quiz.add_row();
}
txn updateQuestion(x,q) {
 Quiz.at(x).question.set(q);
}
txn getQuestion(x,q) {
 return
  Quiz.at(x).question.get();
}
```

(a) Example program.



(b) Abstract History.



(c) False Alarm.

Figure 12. An example of a false alarm caused by ignoring fresh unique values.

We add an encoding of unique values into our SMT-based check (more details in [12]). Using this extension, we learn that the updateQuestion transaction in Figure 12c either accesses a row that is not equal to the unique row created in addQuestion or that it must have observed the insertion of the row in addQuestion. In both cases, no cycle is created, and the example is correctly shown to be serializable.

9 Implementation and Experiments

We implemented the concepts introduced in this paper in a static analysis back end called C^4. In this section, we present experimental results when using this tool as the basis of two static analyzers. Our evaluation demonstrates the effectiveness of our technique: the analyses have a low false alarm rate of 10% and, after filtering, 43% of all reported serializability violations point to actual bugs (the remaining alarms indicate harmless serializability violations). Our experiments also show that all four core features of the analyzer (commutativity, absorption, control-flow, and constraints between events) are essential to achieving these results.

9.1 Implementation

C^4 is independent of the data store, its API, and the programming language, and thus serves as a basis for the analysis of any kind of system that satisfies our assumptions from Section 4: atomic visibility and causal consistency. Our tool can therefore be used as a back end for a broad range of static analyses, e.g., for weakly consistent mobile synchronization frameworks like TouchDevelop and distributed databases like Antidote [2], Walter [32], COPS [29], and Eiger [30].

C^4 is interfaced by static analysis front ends, which are responsible for inferring a sound abstract history from application source code and providing a precise rewrite specification. We have implemented two front ends based on standard static analysis techniques, which we briefly describe in this subsection; more details can be found in [12].

TouchDevelop. Our first front end targets the mobile environment TouchDevelop and is based on an existing static analyzer for that language [13]. TouchDevelop includes a weakly consistent framework for replicating data between devices based on the global sequence protocol [18]. The data store provides atomic visibility and a consistency model slightly stronger than causal consistency and can therefore be directly used with our back end. We statically analyzed 17 TouchDevelop benchmarks; these were also analyzed dynamically in our previous work [11].

Cassandra. Our second front end supports Java programs accessing the distributed database Cassandra through its standard API. It is based on the static analyzer Soot [35]. Plain Cassandra supports only eventual consistency and no transactional guarantees; however, versions that provide causal consistency as well as the necessary means to support weak transactions with atomic visibility and arbitration have been proposed [8, 30]. For the purpose of our experiments, we assume the analyzed open-source applications are run on an implementation providing such stronger guarantees, and that each web-request corresponds to one transaction. The violations we detected also occur when the applications are run on plain Cassandra. Our analysis still provides a strong guarantee in that setting: it will find all bugs in a well-defined class (all serializability violations in which causal consistency and atomic visibility are not violated).

Using our Cassandra front end, we analyzed 11 open-source projects from GitHub of varying complexity. The projects include three libraries for distributed locks and queues (`cassieq`, `cassandra-lock`, `dstax-queueing`), three sample implementations of a Twitter-like service (`cassatwitter`, `cassandra-twitter`, `twissandra`), a trade service (`curr-exchange`), a chat room logging service (`roomstore`), an example implementation of a chatting platform (`killrchat`), and a service for managing music playlists (`playlist`). We analyze a core fragment of the `cassieq` framework, which we refer to as `cassieq-core`.

Filtering Harmless Violations. Requiring serializability on all events in a history is too strong for some applications; many histories involve some permitted non-serializable behaviors. As is standard in the concurrent programming literature, serializability analysis of larger applications is best applied in a *targeted* way. In our experiments, we adopt two previously employed approaches to this problem to our setting. First, we focus the analysis on logically-related subsets of the data in the application called *atomic sets* [36], for which serializability is checked independently. More details are described in [12]. Atomic sets are currently only implemented for TouchDevelop. Second, we employ the *display code* heuristic [11]: queries whose results are never used in the business logic but only displayed to the user are excluded from the serializability analysis.

Table 1. An overview of the analysis results. **T, E** denote the number of abstract transactions and abstract events before unfolding, resp., **FE, BE,** Σ denote the time spent in seconds in the front end, the back end, and in total, resp., and **E, H, F,** Σ denote the number of violations detected, split up into harmful violations (errors), harmless violations, false alarms, and the total number. We provide the number of violations both unfiltered and filtered (with heuristics enabled).

	Size	Time [s]	#Violations	
			Unfiltered	Filtered
Program	**T / E**	**FE / BE / Σ**	**E / H / F / Σ**	**E / H / F / Σ**
Cloud List	4 / 7	4.8 / 1.0 / 5.8	0 / 3 / 0 / 3	0 / 0 / 0 / 0
Super Chat	8 / 28	10.8 / 2.3 / 13.1	0 / 7 / 0 / 7	0 / 3 / 0 / 3
Save Passwords	7 / 13	1.5 / 5.6 / 7.2	0 / 11 / 2 / 13	0 / 1 / 0 / 1
EC2 Demo Chat	2 / 4	0.5 / 0.4 / 0.9	0 / 1 / 0 / 1	0 / 0 / 0 / 0
Contest Voting	2 / 3	1.7 / 0.6 / 2.3	0 / 1 / 0 / 1	0 / 0 / 0 / 0
Chatter Box	5 / 19	8.2 / 8.8 / 17.0	0 / 5 / 4 / 9	0 / 0 / 0 / 0
Tetris	3 / 12	77.1 / 1.0 / 78.1	3 / 0 / 0 / 3	3 / 0 / 0 / 3
NuvolaList 2	5 / 9	0.5 / 6.5 / 7.0	0 / 8 / 0 / 8	0 / 0 / 0 / 0
FieldGPS	4 / 5	4.1 / 4.5 / 8.7	0 / 0 / 0 / 0	0 / 0 / 0 / 0
Instant Poll	4 / 6	2.2 / 3.9 / 6.0	0 / 2 / 0 / 2	0 / 0 / 0 / 0
Expense Rec.	5 / 9	2.4 / 3.0 / 5.4	0 / 1 / 1 / 2	0 / 0 / 0 / 0
Sky Locale	12 / 32	17.0 / 10.6 / 27.6	1 / 34 / 0 / 35	1 / 4 / 0 / 5
Events	4 / 29	3.1 / 1.7 / 4.8	1 / 1 / 0 / 2	1 / 0 / 0 / 1
Cloud Card	9 / 25	11.1 / 7.5 / 18.6	1 / 5 / 0 / 6	1 / 0 / 0 / 1
Relatd	14 / 69	15.7 / 28.0 / 43.7	1 / 18 / 0 / 19	1 / 3 / 0 / 4
Color Line	3 / 10	21.4 / 1.0 / 22.4	3 / 0 / 0 / 3	3 / 0 / 0 / 3
Unique Poll	4 / 4	0.6 / 1.5 / 2.1	0 / 4 / 0 / 4	0 / 0 / 0 / 0
cassandra-lock	3 / 3	6.6 / 0.3 / 6.9	0 / 0 / 0 / 0	0 / 0 / 0 / 0
cassandra-twitter	5 / 26	7.3 / 3.3 / 10.5	1 / 5 / 0 / 6	1 / 1 / 0 / 2
cassatwitter	6 / 19	7.2 / 3.7 / 10.8	1 / 6 / 0 / 7	1 / 1 / 0 / 2
cassieq-core	7 / 10	57.3 / 3.8 / 61.1	2 / 2 / 0 / 4	2 / 1 / 0 / 3
curr-exchange	2 / 2	7.6 / 0.8 / 8.3	0 / 1 / 0 / 1	0 / 0 / 0 / 0
dstax-queueing	2 / 8	6.2 / 0.8 / 7.0	2 / 0 / 0 / 2	2 / 0 / 0 / 2
killrchat	11 / 20	10.1 / 15.3 / 25.4	0 / 31 / 13 / 44	0 / 0 / 4 / 4
playlist	11 / 34	8.5 / 24.1 / 32.6	0 / 13 / 0 / 13	0 / 2 / 0 / 2
roomstore	5 / 13	7.2 / 1.5 / 8.8	0 / 4 / 0 / 4	0 / 0 / 0 / 0
shopping-cart	4 / 5	2.9 / 0.1 / 3.0	0 / 0 / 0 / 0	0 / 0 / 0 / 0
twissandra	7 / 20	7.5 / 4.9 / 12.3	0 / 7 / 0 / 7	0 / 1 / 0 / 1

9.2 Quantitative Results and Manual Inspection

Table 1 shows the results of executing the analysis on all 27 benchmarks. The analysis was run on a Fedora 25 system with an Intel Core i7-4600U and 12GB RAM. All benchmarks except for `cassieq-core` and `Tetris` can be analyzed in less than a minute, with the front end (**FE**) usually taking the majority of the analysis time. The four benchmarks with the highest back end analysis time (**BE**) are also the benchmarks with

the highest number of violations. In both, large numbers of potential violations (i.e., cycles in the SSG) have to be checked using the SMT-based approach, because neither is the SSG-check precise enough to rule them out nor is there a smaller serializability violation that subsumes the potential violation. For all benchmarks, the algorithm in Section 7.2 terminated for $k = 2$, that is, we found a set of violations using 2 sessions that subsumed all possible violations with any number of sessions. The SSG-based check reported 31 violations for TouchDevelop and 139 violations for Cassandra that were then ruled out as infeasible by the SMT-based check. We give more details on the effectiveness of the various features of the SMT-based check below.

The filters reduce the number of violations to be inspected by the developer significantly in almost all cases (compare Σ in the "Unfiltered" and "Filtered" columns). On average, 7.3 violations have to be inspected per project before filtering and 1.3 violations after filtering.

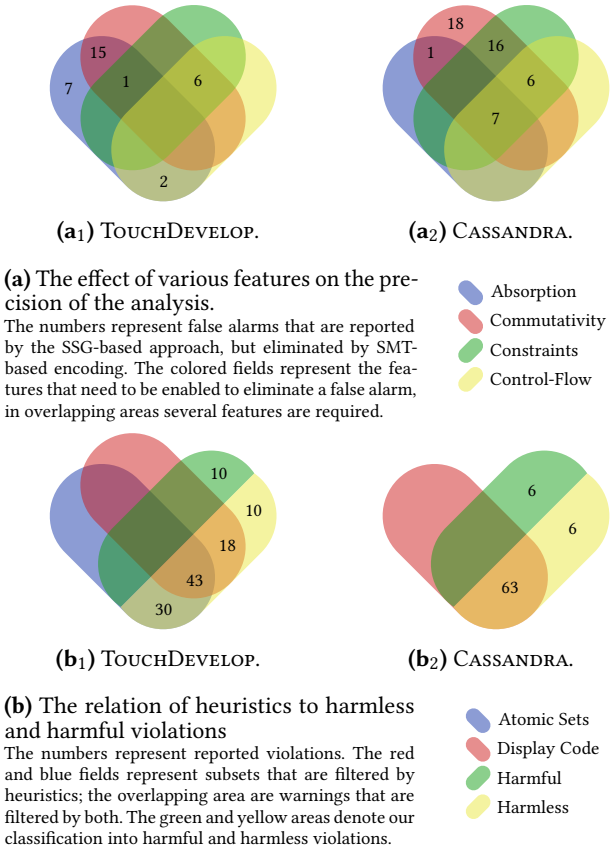
Manual Inspection. When filtering is enabled, 43% of all reported violations point towards clearly harmful behavior in the program. Note, however, that not every harmful violation points to a unique bug since a single bug can cause several violations. 45% of the violations were harmless, and 10% were false alarms. With no filtering, the false alarm rate is even lower, with 7%, since the number of true but harmless violations increases. For Cassandra, virtually all false alarms appear in one challenging example (`killrchat`).

9.3 Interplay of Analysis Features

To determine which features of our analyzer increase precision most effectively, we selectively disabled precision features and observed which additional false alarms were reported by the analysis. The results are shown in the Venn-diagram in Figure 13a. We analyze four features:

Commutativity In the SMT-encoding, replace $\neg\mathrm{com}(e, f)$ by `true` if it is satisfiable, `false` if unsatisfiable.
Absorption In the SMT-encoding, replace $\mathrm{abs}(e, f)$ by `false`.
Constraints Let Inv be a constant function `true`.
Control-Flow Let $\underline{\mathrm{eo}}$ relate all events of a transaction.

If we disable all four features, the precision of the SMT-based check cannot exceed that of the SSG-based check. We can clearly see that all four features are essential for the precision of the analysis. Commutativity plays a much greater role for Cassandra, due to its more complex SQL-like operations, while absorption is more important for TouchDevelop, likely because apps often use the data store to replicate user-private data between devices, making patterns such as the example in Section 2 common, while the open-source projects using Cassandra are of more collaborative nature. Interestingly, there are 7 false alarms for Cassandra that require all four features to be eliminated.



(a₁) TouchDevelop. **(a₂)** Cassandra.

(a) The effect of various features on the precision of the analysis.
The numbers represent false alarms that are reported by the SSG-based approach, but eliminated by SMT-based encoding. The colored fields represent the features that need to be enabled to eliminate a false alarm, in overlapping areas several features are required.

(b₁) TouchDevelop. **(b₂)** Cassandra.

(b) The relation of heuristics to harmless and harmful violations
The numbers represent reported violations. The red and blue fields represent subsets that are filtered by heuristics; the overlapping area are warnings that are filtered by both. The green and yellow areas denote our classification into harmful and harmless violations.

Figure 13. Interplay of analysis features.

9.4 Harmful and Harmless Violations

In a similar experiment, we compare the sets of violations that (1) were classified as harmful resp. harmless by manual inspection and (2) were filtered by the atomic-sets and display-code heuristics. The results are shown in Figure 13b. No harmful violations are filtered out, and only a low number of harmless violations are not filtered. For TouchDevelop, we can observe that while atomic sets and display code overlap significantly, omitting one of them would significantly increase the number of harmless violations shown to developers. Figure $13b_2$ shows why we did not implement atomic sets for Cassandra: in our experiments, the display code heuristic was very effective in filtering out harmless violations (91%) while preserving all harmful violations.

9.5 Discovered Bugs

We describe the discovered bugs in detail in [12]. As expected due to the soundness of our approach, C^4 found all violations for TouchDevelop that were detected by our dynamic analysis [11]. Moreover, our approach found three new bugs that were missed by the dynamic analysis. All three additional bugs are unlikely to be triggered by dynamic analysis. For Cassandra, we found clearly harmful violations in 4 out of 10 applications.

In general, most harmful violations belong to one of the following four categories: (1) they try to establish uniqueness of user-provided values such as user-names without using proper synchronization; (2) they read, modify, and write high-level data types such as sets without using appropriate high-level operations; (3) they modify data that is concurrently deleted, often resulting in partial revival of the deleted data; (4) they add data to an entity that is concurrently deleted, thereby creating garbage data and sometimes breaking implicit foreign-key constraints.

10 Related Work

Our model of weakly consistent executions is based on Burckhardt [14] and our serializability criterion is inspired by our previous work [11]. We extend these concepts to static analysis.

Databases. Fekete et al. [22] were the first to propose static serializability checking when the database provides only weak guarantees based on an SSG similar to the one described in this paper. The technique they propose is entirely manual, but it is shown in [25] that some steps of the analysis can be automated. Both works handle only the consistency model of snapshot isolation, which is stronger than the causal consistency considered in our paper. In particular, snapshot isolation ensures that for each pair of concurrent transactions that write to the same entity of the data store, one will abort. These additional guarantees remove the need to reason about commutativity and absorption between updates, which are two of the major technical difficulties addressed by our work (see the example in Section 2). Conflict-detection also enables them to work around the imprecisions of the SSG-based approach [25] by eliminating false cycles with conflicting updates. We cannot make this assumption in our work. Furthermore, their analysis approach is neither sound (in their experiments, they extract the operations from database logs), nor fully automatic (they perform manual splitting of transactions to handle control flow).

Bernardi and Gotsman [9] describe a static serializability criterion for a fixed set of transactions with concrete inputs, but without a fixed schedule. They also briefly sketch how to extend their approach to arbitrary sequences of transactions and lift a definition of critical cycles to a graph representing all possible transaction sequences. Our cycle-based criterion in Section 6 is inspired by that work; however, we generalize the criterion to arbitrary data types using commutativity and absorption and make it more precise by taking absorption into account. Further, we show that an approach based on a summarizing graph is useful as an efficient pre-filter, but not precise enough for many examples.

Weak Memory Models. Many publications have addressed the problem of static analysis to determine whether all executions of a program executed under a weak memory model are equivalent to a sequentially consistent execution [3, 20, 26, 27, 31, 33]. The closest one to our approach is the work by Alglave et al. [3]. Their construction of the *abstract event graph* is similar to our logical encoding, since both use the fact that cycles in a dependency graph only ever contain two nodes per session to obtain a sound bounded unrolling of loops. However, the memory guarantees (TSO/Power), available operations (reads, writes, and fences), and reference model (sequential consistency instead of serializability) considered in their work differ significantly from our model.

Some of the works on weak memory models use encodings of axiomatic execution models into a logical formula [4, 15, 38], similar to our encoding. These approaches bound the number of loop iterations. We only need to bound the number of sessions, and we give a sufficient condition for the generalization to an arbitrary number of sessions.

Concurrent Programming. Our bounded encoding of serializability checking in Section 7 has similarities with work on atomicity checking based on conflict-serializability [5, 21]. However, we operate under a substantially different abstraction: we do not require a finite data abstraction, a finite number of objects, or a boolean abstraction of the program.

Verification. Gotsman et al. [24] propose a proof rule for showing that applications accessing a causally consistent data store preserves a given integrity invariant. They require the user to supply such an invariant, while our correctness condition requires no annotations.

11 Conclusion

We presented a static serializability analysis for applications running on top of causally consistent data stores. Based on a novel, local consistency criterion, our analysis first performs cycle detection on static serialization graphs as a pre-filter and then uses SMT-based logical analysis to obtain precise results. Both techniques are fully automatic and use commutativity and absorption to handle high-level replicated data types. We implemented both analyzers in a reusable back end and evaluated it for reasoning about two distributed systems, TouchDevelop and Cassandra/Java, demonstrating the effectiveness of our method.

Acknowledgments

We thank Arthur Kurath for the implementation of the Cassandra front end, Alex Summers for helpful discussions on SMT encodings, and Alexey Gotsman for his comments on earlier drafts of this paper.

References

[1] Atul Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* PhD Thesis. Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science.

[2] Deepthi Devaki Akkoorath and Annette Bieniusa. 2016. *Antidote: The Highly-Available Geo-Replicated Database with Strongest Guarantees.* Technical Report. Tech. U. Kaiserslautern. https://syncfree.lip6.fr/attachments/article/59/antidote-white-paper.pdf

[3] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. 2014. Don't Sit on the Fence. In *Lecture Notes in Computer Science (Lecture Notes in Computer Science).* Springer, 508–524.

[4] Jade Alglave, Daniel Kroening, and Michael Tautschnig. 2013. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In *CAV'13.* Springer, 141–157.

[5] Rajeev Alur, Ken McMillan, and Doron Peled. 2000. Model-Checking of Correctness Conditions for Concurrent Objects. *Inf. Comput.* 160, 1 (2000), 167–188.

[6] Hagit Attiya, Faith Ellen, and Adam Morrison. 2017. Limitations of Highly-Available Eventually-Consistent Data Stores. *IEEE Transactions on Parallel and Distributed Systems* 28, 1 (2017), 141–155.

[7] Peter Bailis, Alan Fekete, Joseph M. Hellerstein, Ali Ghodsi, and Ion Stoica. 2014. Scalable Atomic Visibility with RAMP Transactions. In *SIGMOD '14.* ACM, 27–38.

[8] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Bolt-on Causal Consistency. In *SIGMOD '13.* ACM, 761–772.

[9] Giovanni Bernardi and Alexey Gotsman. 2016. Robustness against Consistency Models with Atomic Visibility. In *CONCUR'16.*

[10] Ahmed Bouajjani, Constantin Enea, and Jad Hamza. 2014. Verifying Eventual Consistency of Optimistic Replication Systems. In *POPL '14.* ACM, 285–296.

[11] Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin Vechev. 2017. Serializability for Eventual Consistency: Criterion, Analysis, and Applications. In *POPL '17.* ACM, 458–472.

[12] Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin Vechev. 2018. *Static Serializability Analysis for Causal Consistency (extended version).* Technical Report. ETH Zurich. http://dx.doi.org/10.3929/ethz-b-000258176

[13] Lucas Brutschy, Pietro Ferrara, and Peter Müller. 2014. Static Analysis for Independent App Developers. In *OOPSLA '14.* ACM, 847–860.

[14] Sebastian Burckhardt. 2014. Principles of Eventual Consistency. *Found. Trends Program. Lang.* 1, 1-2 (2014), 1–150.

[15] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. 2007. CheckFence: Checking Consistency of Concurrent Data Types on Relaxed Memory Models. In *PLDI '07.* ACM, 12–21.

[16] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. 2012. Cloud Types for Eventual Consistency. In *ECOOP'12.* Springer, 283–307.

[17] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated Data Types: Specification, Verification, Optimality. In *POPL '14.* ACM, 271–284.

[18] Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. 2015. Global Sequence Protocol: A Robust Abstraction for Replicated Shared State. In *Leibniz International Proceedings in Informatics (LIPIcs)*, John Tang Boyland (Ed.), Vol. 37. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 568–590.

[19] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. In *SOSP '07.* ACM, 205–220.

[20] Xing Fang, Jaejin Lee, and Samuel P. Midkiff. 2003. Automatic Fence Insertion for Shared Memory Multiprocessing. In *ICS '03.* ACM, 285–294.

[21] Azadeh Farzan and P. Madhusudan. 2008. Monitoring Atomicity in Concurrent Programs. In *CAV '08.* Springer, 52–65.

[22] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. 2005. Making Snapshot Isolation Serializable. *ACM Trans. Database Syst.* 30, 2 (2005), 492–528.

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

[24] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. 'Cause I'm Strong Enough: Reasoning About Consistency Choices in Distributed Systems. In *POPL '16.* ACM, 371–384.

[25] Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S. Sudarshan. 2007. Automating the Detection of Snapshot Isolation Anomalies. In *VLDB '07.* VLDB Endowment, 1263–1274.

[26] Arvind Krishnamurthy and Katherine Yelick. 1996. Analyses and Optimizations for Shared Address Space Programs. *J. Parallel Distrib. Comput.* 38, 2 (1996), 130–144.

[27] Michael Kuperstein, Martin Vechev, and Eran Yahav. 2010. Automatic Inference of Memory Fences. In *FMCAD '10.* FMCAD Inc, 111–120.

[28] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (2010), 35–40.

[29] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *SOSP '11.* ACM, 401–416.

[30] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *NSDI '13.* USENIX Association, 313–328.

[31] Dennis Shasha and Marc Snir. 1988. Efficient and Correct Execution of Parallel Programs That Share Memory. *ACM Trans. Program. Lang. Syst.* 10, 2 (1988), 282–312.

[32] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional Storage for Geo-Replicated Systems. In *SOSP '11.* ACM, 385–400.

[33] Zehra Sura, Xing Fang, Chi-Leung Wong, Samuel P. Midkiff, Jaejin Lee, and David Padua. 2005. Compiler Techniques for High Performance Sequentially Consistent Java Programs. In *PPoPP '05.* ACM, 2–13.

[34] Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, and Manuel Fahndrich. 2011. TouchDevelop: Programming Cloud-Connected Mobile Devices via Touchscreen. In *Onward! 2011.* ACM, 49–60.

[35] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *CASCON '99.* IBM Press.

[36] Mandana Vaziri, Frank Tip, and Julian Dolby. 2006. Associating Synchronization Constraints with Data in an Object-Oriented Language. In *POPL '06.* ACM, 334–345.

[37] William E. Weihl. 1988. Commutativity-Based Concurrency Control for Abstract Data Types. *IEEE Trans. Comput.* 37, 12 (1988), 1488–1505.

[38] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically Comparing Memory Consistency Models. In *POPL 2017.* ACM, 190–204.