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.

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 [26], 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, 17, 21, 32].

Among weak consistency models, causal consistency has received increasing attention in terms of both theoretical analysis and practical implementations [2, 8, 19, 33, 34]. 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 [12]. 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 [13] 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 \(\oplus\)), 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.

---

\(^1\)Source code available at: http://ecracer.inf.ethz.ch/
We address this problem by proposing a novel criterion that is more precise than previous characterizations for causal consistency. The first to handle high-level operations and is more precise for this example. Our characterization of cycles in the SSG is problematic since any two updates do overwrite each other. We lift this criterion to SSGs and will contain infeasible cycles.

To define this criterion, we build on our previous work [12], 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 [24] 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 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 1c2 or Figure 1c4. 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 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 1c2 or Figure 1c4. 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 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 1c2 or Figure 1c4. 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 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 1c2 or Figure 1c4. 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 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 1c2 or Figure 1c4. 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 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 1c2 or Figure 1c4. 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 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 1c2 or Figure 1c4. 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 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 1c2 or Figure 1c4. 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 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 1c2 or Figure 1c4. 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 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 1c2 or Figure 1c4. However, the SSG in Figure 1b cannot capture this semantic information and will contain infeasible cycles.

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 1c2 and 1c4 are possible, but Figure 1c1 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 edge in Figure 1c2 can never appear in the same DSG as the edge in Figure 1c3 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, $C^4$
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., [11, 18]) and closest in exposition to [12]. 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 \( u \in U \) or queries \( q \in Q \). 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 \( aeβ \) is legal if the prefix \( ae \) is legal.

We model concurrent executions as histories (Figure 3a). A history \( H = \langle \text{Ev}, \text{so}, \text{Tx} \rangle \) consists of: a finite set of events \( \text{Ev} \); a session order \( \text{so} \subseteq \text{Ev} \times \text{Ev} \) whose connected components are all chains, called **sessions**; a partition \( \text{Tx} \subseteq \mathcal{P}(\text{Ev}) \) 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 \( S = \langle \text{vi}, \text{ar} \rangle \), consists of: a strict total order \( \text{ar} \subseteq \text{Ev} \times \text{Ev} \), called the **arbitration** order, which indicates the logical execution order of events; a relation \( \text{vi} \subseteq \text{ar} \), 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 \( e \in \text{Ev} \), \( \text{ar} \) restricted to \( \text{vi}^{-1}(e) \cup \{e\} \) forms a legal sequence according to the sequential semantics. Second, it must respect **causal consistency** [15]. Visibility must be transitively closed, and moreover, each event must be visible to all subsequent events in the same session:
  - **(S2)** \( \text{vi} = (\text{so} \cup \text{vi})^+ \)

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

- **(S3)** For every pair of distinct transactions \( s \neq t \in \text{Tx} \), and for all events \( \{e, f\} \subseteq s \) and \( \{e', f'\} \subseteq t \):
  \[
  e \xrightarrow{\text{vi}} e' \iff f \xrightarrow{\text{vi}} f'
  \]

A schedule is **serial** iff \( \text{vi} = \text{ar} \). 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 [41], 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 \( \alpha \) and \( \beta \) are equivalent \( (\alpha \equiv \beta) \) if substituting one for the other in any sequence leaves its legality unchanged. Then, for any pair of events \( e, f \):

\[
\begin{align*}
e \text{ and } f \text{ commute} & \iff ef \equiv fe \\
f \text{ absorbs } e & \iff ef \equiv f.
\end{align*}
\]

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 [12], 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., [10, 24], 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 [12] 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 \( \succ \) 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:

\[
\text{(R1)} \quad \succ \subseteq U \times U, \text{ and } u \succ v \iff uv = \beta v \text{ for all } \beta \subseteq U.
\]

Now, we define the far-commutativity relation \( \oslash \) 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:

\[
\text{(R2)} \quad \oslash \subseteq U \times Q, \text{ and } u \oslash q \iff uq = q \text{ and for all } v \in U,
\]

\[
uv = vu \quad \text{or} \quad v \oslash q \quad \text{or} \quad u \succ v.
\]

Here, the right-hand side use of \( \oslash \) is coinductive: we seek the largest relation satisfying (R2). We prove that this has the required effect in Appendix A. We extend the definition of \( \oslash \) 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 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:

\[
\text{(D1)} \quad \oslash \subseteq U \times Q, \text{ and if } u \oslash q \text{ and } (u, q) \notin \oslash, \text{ then } u \oslash q \text{ or there exists some } v \text{ such that } u = v \text{ and } u \oslash v \oslash q.
\]

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, \( \text{get}(a, 2) \) depends on \( \text{put}(a, 2) \), but not on \( \text{inc}(a, 1) \), which is absorbed by put.

Anti-dependencies are analogous for invisible updates:

\[
\text{(D2)} \quad \oslash \subseteq Q \times U, \text{ and if } u \oslash q \text{ and } (q, u) \notin \oslash \text{ then } u \oslash q \text{ or there exists some } v \text{ such that } u = v \text{ and } u \oslash v \oslash q.
\]

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:

\[
\text{(D3)} \quad \oslash \subseteq U \times U, \text{ and if } u \oslash v \text{ and } (u, v) \notin \oslash \text{ then } uv = vu.
\]

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

We now lift each of the relations \( R \in \{ \oslash , \oslash , \oslash , \oslash \} \) to a relation \( \bar{R} \) on transactions in the following way:

\[
(s, t) \in \bar{R} \iff s \neq t \text{ and } (e, f) \in R \text{ for some } e \in s, f \in t.
\]

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 \( \bar{R} \) for any pair \( (s, t) \in \bar{R} \in \{ \oslash , \oslash , \oslash , \oslash \} \).

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 Appendix A.2.

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 \( (v, a, t) \) with dependence triple \( (\oslash , \oslash , \oslash) \), the restriction \( (v | E, a | E) \) of the schedule to a subset \( E \subseteq Ev \) has a dependence triple \( (\oslash , \oslash , \oslash ) \) such that:

\[
\oslash ' \supseteq \oslash | E \quad \oslash ' \supseteq \oslash | E \quad \oslash ' \supseteq \oslash | E.
\]

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.
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 \( \text{Var}_L \) and \( \text{Var}_G \) 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 \( H \) consisting of

\[
\begin{align*}
\Phi & = U \cup Q: \text{a set of abstract events (updates and queries);} \\
\mathcal{E} & \subseteq \Phi: \text{the set of abstract transactions;} \\
\mathcal{E}_0 & \subseteq \mathcal{E} \times \mathcal{E}: \text{the abstract event order;} \\
\mathcal{E}_0 & \subseteq \mathcal{E} \times \mathcal{E}: \text{the session order;} \\
\text{Var}_L, \text{Var}_G: \text{sets of session-local and global variables;} \\
\text{Inv}: \mathcal{E}_0 \to \Phi(\text{arg}^\text{src} \cup \text{arg}^\text{tgt} \cup \text{Var}_G \cup \text{Var}_L): \text{a mapping from concrete transactions to abstract transactions.}
\end{align*}
\]

We assume that every transaction \( t \in \mathcal{E}_0 \) has unique entry and exit events \( \text{entry}[t], \text{exit}[t] \): the lone events in \( t \) having no predecessors and successors in \( \mathcal{E}_0 \), respectively.

**Concretization.** An abstract history \( H \) over-approximates the concrete histories of a given program, but it is consistent with a larger set of histories, namely, the concretizations \( H \in \gamma(H) \). A history belongs to \( \gamma(H) \) if it has a concretization model: a mapping from events to abstract events, and valuations of the \( \text{Var}_L \) and \( \text{Var}_G \) vars such that: (1) \( \text{so} \)-arcs map respectively to \( \text{eo} \)-arcs inside transactions and \( \text{so} \)-arcs outside transactions; (2) each invariant is satisfied by the corresponding pairs of concrete events. Note that a history in \( \gamma(H) \) 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 Appendix B for a formal definition of concretizations.

**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 \((\text{com}, \text{abs})\) of families of logical formulas over \( \text{arg}^\text{src} \) and \( \text{arg}^\text{tgt} \), each indexed by pairs of operations, such that for all events \( e, f \):

\[
\begin{align*}
\text{com}(\text{op}[e], \text{op}[f]) & \Rightarrow e \not\supset f \\
\text{abs}(\text{op}[e], \text{op}[f]) & \Rightarrow e \supset f.
\end{align*}
\]

Figure 6 shows a rewrite specification for a dictionary. We write \( \lnot \text{com}(e, f) \) if \( \lnot \text{com}(\text{op}[e], \text{op}[f]) \) is satisfiable, where \( e, f \) are abstract events, and similarly for absorption abs.
be an abstract history and, for every with the following properties:

- \( k \neq k' \) or \( v = v' \) → \( k \neq k' \)
- \( k \neq k' \)
- \( k \neq k' \)

(a) Commutativity specification.

(b) Absorption specification.

Figure 6. Rewrite specification for a dictionary.

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, 10, 24]. 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 \( T_x \) and

1. an edge \((s, t)\) labeled \(\diamond\) for every \((s, t) \in \mathcal{S}\);

2. an edge \((s, t)\) if \(\exists e \in s, f \in t\) such that \(\neg \text{com}(e, f)\)
   - labeled \(\oplus\) if \(e \in \mathcal{U}\) and \(f \in \mathcal{Q}\)
   - labeled \(\ominus\) if \(e \in \mathcal{Q}\) and \(f \in \mathcal{U}\)
   - labeled \(\odot\) if \(e \in \mathcal{U}\) and \(f \in \mathcal{U}\)

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 \(H\) be an abstract history and \(G\) be the DSG of a history in \(y(H)\). If there is a cycle in \(G\) then there is a closed walk in the SSG of \(H\) with the following properties:

(\(SC1\)) It contains at least two \(\bowtie\)-edges, or at least one \(\bowtie\)-edge and one \(\ominus\)-edge.

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

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

(\(SC2b\)) It contains \(q \in Q, u, v \in U, e \in E_v\) with \(q \rightarrow u\) 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 Appendix C. 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 \(u \in \mathcal{V}_G\), 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, \(u, u_2 \in \mathcal{V}_G\), 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).
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 SSSs 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 Appendix D. 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.

Algorithm 1 Serializability checking by unfolding.

```
1: function CheckBounded(H, k, V)
2:   for \( H' \in \text{Unfoldings}(H, k) \) do
3:     if \text{CyclePossible}(H') then
4:       if \( \exists m. m \models \phi_{\text{cyclic}}(H') \) then
5:         \( V \leftarrow V \cup \{ m \} \)
6:     return \( V \)
7:   function Check(H)
8:     \( V \leftarrow \emptyset, k \leftarrow 2 \)
9:   repeat
10:    \( V \leftarrow V \cup \text{CheckBounded}(H, k, V) \)
11:    \( k \leftarrow k + 1 \)
12: until \text{SubsumptionGeneralizes}(H, k, V)
13: return \( V \)
```

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 \text{CheckBounded}(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 \( \phi_{\text{cyclic}} \).

Function \text{Check} iteratively calls \text{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 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 \( \phi_{\text{cyclic}} \) in Appendix E.

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. A similar property has been observed in the analysis of sequential consistency [35].

**General structure.** The $k$-session unfoldings $H' \in \mathbb{U}_k[H]$ of a given abstract history $H$ 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 $\sqsubseteq$ from the original abstract history. The unfolding places unfolded copies of these abstract transactions in their corresponding sessions and links them with $\sqsubseteq'$ in the indicated order. Transactions with an acyclic abstract event order $eo$ unfold to themselves, just like in Figure 7b.

**Unfolding of transactions.** Unfolding of transactions is necessary to ensure that the DSG cycles of $H$ 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 $eo$ 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 insert 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 corresponding inclusion maps $i_1: V \rightarrow V_1$, $i_2: V \rightarrow V_2$ 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:****

- $I \subseteq E$ — incoming edges ($Ev \setminus V \rightarrow V$)
- $O \subseteq E$ — outgoing edges ($V \rightarrow Ev \setminus V$)
- $B \subseteq E$ — back edges in any DFS of $V$
- $R \subseteq E$ — the remaining edges in $E$.

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

\[
\begin{align*}
Ev' &= (Ev \setminus V) \cup V_1 \cup V_2 \\
eo' &= (eo \setminus E) \cup I' \cup O' \cup B' \cup R' \\
I' &= (1 \times i_1)[I \cup I_x \times B_1] \\
B' &= (i_1 \times i_2)[B_x \times B_1] \\
O' &= (i_1 \times 1)[O] \cup (i_2 \times 1)[O \cup (B_x \times O_t)] \\
R' &= (1 \times i_1)[R] \cup (i_2 \times i_2)[R].
\end{align*}
\]

**Invariants:**

\[
\begin{align*}
\text{Inv}^+ | \{I' \cup O' \cup B'\} &= T \\
\text{Inv}^+ | R' &= (\text{Inv}^+ | R) \circ [(i_1 \times i_1) \cup (i_2 \times i_2)]^{-1}.
\end{align*}
\]

### 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 $s < l$ 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 Appendix F.

### 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 approach with asymmetric commutativity, uniqueness information and data monotonicity to further increase precision. 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 Appendix H). 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. The addFollower transaction adds a follower \(n2\) to a given user. Data stores typically create a record upon modification if the record does not exist. 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 addFollower 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 [13]). 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...
We have not proved the soundness of this approach but which are guaranteed to have a fresh identity. This is akin whether a record exists before inserting a record. Therefore, contains("A") :true as inserted earlier. This leads to anti-dependency edges where contains("A") :true 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 limitation of commutativity, we use an asymmetric version of the model.

First, we require all fresh unique values to be unequal: ∀ e \in Ev, m \in \mathbb{N}, n \neq m \land Uniq(op[e], n) \land Uniq(op[f], m) \implies arg_e[f] \neq arg_m[f]

Second, all operations that use a fresh unique value must observe the event creating it (T[e] denotes the transaction of an event e):

∀ e, f \in \mathcal{E}\cup\mathcal{V}, n, m \in \mathbb{N}, n \neq m \land Uniq(op[e], n) \land Uniq(op[f], m) \implies arg_e[f] \neq arg_m[f] \lor T[e] \wedge T[f]

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.

**Exploiting monotone behaviors** Finally, we employ another empirical observation to increase the precision of our analyzer: Applications using weakly consistent data stores often exhibit monotone behavior to avoid consistency-related anomalies: records are created but can never be deleted, flags are enabled but never disabled, and counters are increased but never decreased. For example, out of the 41 tables queried in the open source projects using CASSANDRA in Section 9, on only five tables a DELETE operation is performed. We can make use of monotone behavior to increase the precision of the analysis. Consider the false alarm in Figure 13, also arising from the code in Figure 11a. Two sessions concurrently add a follower to the user record of A. Then both check in a second transaction if user A exists (we omitted irrelevant

**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 (ignore the red text for now). 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, reads the created row without observing its creation, which shows that this is a false alarm.

To capture the uniqueness of records, we extend our rewrite specification such that it provides for each operation o and each argument index n a predicate Uniq(o, n) that is true exactly if the nth argument is a fresh unique value. For operations that implicitly create a fresh unique value, that is, the value is neither an argument nor a return value of the operation, a corresponding ghost argument can be added to the model.

Using this predicate, we can encode constraints over the arguments of values into the SMT formula Conjunct (dsg). First, we require all fresh unique values to be unequal:

∀ e, f \in Ev, n, m \in \mathbb{N}, n \neq m \land Uniq(op[e], n) \land Uniq(op[f], m) \implies arg_e[f] \neq arg_m[f]

Second, all operations that use a fresh unique value must observe the event creating it (T[e] denotes the transaction of an event e):

∀ e, f \in Ev, n, m \in \mathbb{N}, n \neq m \land Uniq(op[e], n) \land Uniq(op[f], m) \implies arg_e[f] \neq arg_m[f] \lor T[e] \wedge T[f]

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.
operations here for clarity). If users are never deleted, this is not a true violation, because both sessions must observe that user A exists.

We address this problem by further extending our rewrite specification with a family of logical formulas \( \text{Legal}(o, p) \) for all operations \( o, p \), over the variables from \( \text{arg}^{\text{src}} \) and \( \text{arg}^{\text{tgt}} \). We require that:

\[
\text{arg}^{\text{src}}_0 \equiv \text{arg}^{\text{tgt}} \implies \text{arg}^{\text{tgt}}_1 = \text{true}
\]

In \( \phi_{\text{incl}} \) from Section 7, the following universally quantified constraint is added:

\[
\forall e, f \in \text{Ev}, e \sqsubseteq f \implies \text{Legal}(op[e], op[f])(\text{arg}_e, \text{arg}_f)
\]

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 [36], COPS [33], and Eiger [34].

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 Appendix H.

**TouchDevelop.** Our first front end targets the mobile environment TouchDevelop and is based on an existing static analyzer for that language [14]. TouchDevelop includes a weakly consistent framework for replicating data between devices based on the global sequence protocol [19]. 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 [12].

**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 [39]. 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, 34]. 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 [40], for which serializability is checked independently. More details are described in Appendix G. Atomic sets are currently only implemented for TouchDevelop. Second, we employ the display code heuristic [12]: queries whose results are never used in the business logic but only displayed to the user are excluded from the serializability analysis.

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-based check cannot exceed that of the SSG-based check. We give more details on the effectiveness of the various features of the SMT-based approach, because neither is the SSG-based check cannot exceed that of the SSG-based check.

The filters reduce the number of violations to be inspected by the developer significantly in almost all cases (compare \(\Sigma\) 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 (\(k_{11}lr:\text{chat}\)).

<table>
<thead>
<tr>
<th>Program</th>
<th>Size</th>
<th>Time [s]</th>
<th>#Violations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T</td>
<td>E</td>
<td>FE</td>
</tr>
<tr>
<td>Cloud List</td>
<td>4</td>
<td>7</td>
<td>4.8</td>
</tr>
<tr>
<td>Super Chat</td>
<td>8</td>
<td>28</td>
<td>10.8</td>
</tr>
<tr>
<td>Save Passwords</td>
<td>7</td>
<td>13</td>
<td>1.5</td>
</tr>
<tr>
<td>EC2 Demo Chat</td>
<td>2</td>
<td>4</td>
<td>0.5</td>
</tr>
<tr>
<td>Contest Voting</td>
<td>2</td>
<td>3</td>
<td>1.7</td>
</tr>
<tr>
<td>Chatter Box</td>
<td>5</td>
<td>19</td>
<td>8.2</td>
</tr>
<tr>
<td>Tetris</td>
<td>3</td>
<td>12</td>
<td>77.1</td>
</tr>
<tr>
<td>NuvolaList 2</td>
<td>5</td>
<td>9</td>
<td>0.5</td>
</tr>
<tr>
<td>FieldGPS</td>
<td>4</td>
<td>5</td>
<td>4.1</td>
</tr>
<tr>
<td>Instant Poll</td>
<td>4</td>
<td>6</td>
<td>2.2</td>
</tr>
<tr>
<td>Expense Rec.</td>
<td>5</td>
<td>9</td>
<td>2.4</td>
</tr>
<tr>
<td>Sky Locale</td>
<td>12</td>
<td>32</td>
<td>17.0</td>
</tr>
<tr>
<td>Events</td>
<td>4</td>
<td>29</td>
<td>3.1</td>
</tr>
<tr>
<td>Cloud Card</td>
<td>9</td>
<td>25</td>
<td>11.1</td>
</tr>
<tr>
<td>Relatd</td>
<td>14</td>
<td>69</td>
<td>15.7</td>
</tr>
<tr>
<td>Color Line</td>
<td>3</td>
<td>10</td>
<td>21.4</td>
</tr>
<tr>
<td>Unique Poll</td>
<td>4</td>
<td>4</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Table 1. An overview of the analysis results. T, E denote the number of abstract transactions and abstract events before unfolding, resp., FE, BE, \(\Sigma\) denote the time spent in seconds in the front end, the back end, and in total, resp., and E, H, F, \(\Sigma\) 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).

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 14a. We analyze four features:

1. **Commutativity** In the SMT-encoding, replace \(\neg \text{com}(e, f)\) by true if it is satisfiable, false if unsatisfiable.

2. **Absorption** In the SMT-encoding, replace \(\text{abs}(e, f)\) by false.

3. **Constraints** Let Inv be a constant function true.

4. **Control-Flow** Let 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 in the analysis.
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.

(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 14. 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 14b. 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 14b2 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 Appendix I. As expected due to the soundness of our approach, C4 found all violations for TouchDevelop that were detected by our dynamic analysis [12]. 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 [15] and our serializability criterion is inspired by our previous work [12]. We extend these concepts to static analysis.

Databases. Fekete et al. [24] 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 [28] 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 [28] 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 [10] 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 \([3, 22, 29, 30, 35, 37]\). 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 \([4, 16, 42]\), 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 \([5, 23]\). 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. \([27]\) 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.
References


A  Soundness of the serializability criterion
In this appendix, we show that the dependence from Section 4.2 satisfies the axioms for dependence in [12], which in turn implies Theorem 1.

A.1 Properties of far absorption/commutativity

Lemma 4. Far absorption $*$ between updates is transitive.
Proof. If $u \triangleright v \triangleright w$ and $\beta$ a sequence of updates, then
\[
\begin{align*}
u \beta w & \equiv u \beta v w \qquad \text{(by $u \triangleright w$)} \\
& \equiv \beta v w \qquad \text{(by $u \triangleright v$)} \\
& \equiv \beta w \qquad \text{(by $v \triangleright w$)}
\end{align*}
\]
\qed

Lemma 5. Let $u, v, \alpha \subseteq U$ be a sequence of updates where no far absorption happens and let $u \triangleright v \gamma q$. Then, $\alpha$ can be partitioned into two subsequences $\beta, \gamma$ so that $f g \equiv g f$ for all $f \in \beta$, $g \in \gamma q$.

Proof. Consider the digraph $G = (V, E)$ with a vertex set $V = \{u\} \cup \alpha \cup \{q\}$ and arcs $(f, g) \in E$ iff $g$ occurs before $f$ in $u a q$ and $f g \neq g f$. It is sufficient to prove that $q$ is unreachable from $u$ in $G$, and let $\beta = \{\text{vertices reachable from $u$}\}$, $\gamma = \{\text{vertices reaching $q$}\}$. We go on by induction on $|V|$. In the base case $|V| = 2$ the two cannot be connected by an edge because $u q \equiv q u$. Assume the inductive hypothesis for all graphs with less than $|V|$ vertices. If $v$ is any successor of $u$, then $uv \neq vu$, thus $u \triangleright v \gamma q$ by the definition of far absorption. By the inductive hypothesis, $q$ is not reachable from $v$ in the subgraph $G \setminus \{u\}$ and thus in $G$ as well. \qed

A.2 Dependence

Lemma 6. For any pre-schedule $(v, ar)$, any relation $\ominus \subseteq R \subseteq vi$, and any query $q$:
\[
q - \text{legal in $(v, ar)$} \iff q - \text{legal in $(R, ar)$}.
\]
Proof. Consider the updates observed by $q$ and ordered into a sequence $\gamma$ according to $ar$. Recall that by definition, $q$ is legal in $(v, ar)$ iff $\gamma q$ is a legal. It is, therefore, sufficient to prove that $ag$ is legal iff $\beta q$ is legal, where $a \subseteq \gamma$ are the dependencies of $q$, and $a \subseteq \beta \subseteq \gamma$. To do so, we will remove all elements $u_1, \ldots, u_n \in \beta \setminus a$ from $\beta$ such that for every $\beta_i = \beta \setminus \{u_1, \ldots, u_i\}$:
\[
\beta_i_{-1}q - \text{legal} \iff \beta_i q - \text{legal}.
\]
We order the elements in $\beta \setminus a$ into two sets:
1. Elements $u_i \in \{u_1, \ldots, u_n\}$ absorbed by elements $v \in \gamma$ appearing later than $u_i$. If it happens that $v \in \beta_{i-1}$, then by absorption $\beta_{i-1}q \equiv \beta_i q$. Otherwise $v \notin \beta_{i-1}$ and by the transitivity of $\triangleright$ we can assume that $v$ is the last such element, i.e., it is not absorbed by any later element in $\gamma$. But then because $(v, q) \notin \ominus$, it must be the case that $v \triangleright q$. Since updates are free of preconditions, we can use $v$ to absorb $u_i$:
\[
\begin{align*}
\beta_{i-1}q - \text{legal} & \iff \beta_{i-1}qv - \text{legal} \\
\beta_{i-1}qv & \equiv \beta_{i-1}vq & \equiv \beta_iqv & \equiv \beta_iqv \\
\beta_iqv - \text{legal} & \iff \beta_iq - \text{legal}.
\end{align*}
\]
2. Elements $u_i \in \{u_{k+1}, \ldots, u_n\}$ not absorbed by any later elements in $\gamma$. Then, by the definition of $\ominus$, it must be the case that $u_i \triangleright q$. Letting $\beta_{i-1}q = \beta_{i-1}u_i \zeta q$, observe that no far absorption happens in $u_i \zeta q$ since all possible absorptions were done at steps $1 \ldots k$. Thus, by Lemma 5, $u_i \zeta q$ partitions into the pairwisely commuting $u_i \zeta q_1$ and $\zeta q_2$ q, and so:
\[
\begin{align*}
\beta_{i-1}q & \equiv \delta u_i \zeta q_1 \zeta q_2 q & \equiv \delta \gamma q u_i \zeta q_1 \\
\delta \gamma q u_i \zeta q_1 - \text{legal} & \iff \delta \gamma q \zeta 2q_1 - \text{legal} \\
\delta \gamma q \zeta 2q_1 & \equiv \delta \gamma q \zeta 2 q \equiv \beta_i q.
\end{align*}
\]
Proof. Assume that $\ominus^{-1} \cap v_2 = \emptyset$, and let $u \in U$ and $q \in Q$. By definition, $(u, q) \in \ominus$ iff $u \triangleright v_1 q$, $u \not\triangleright v_2 q$ and $u \not\triangleright v$ for all $u \not\triangleright v_1 q, i = 1 \ldots 2$. Thus, every $(u, q) \in \ominus \cap v_1$ belongs to $\ominus_1$ as well, that is, $\ominus_1 \supseteq \ominus \cap v_1$. On the other hand $\ominus_2 \setminus v_1 = \emptyset$ since any $(u, q) \in \ominus_2 \setminus v_1$ belongs to $\ominus_1^{-1} \cap v_2$. \qed

B  Concretization of abstract histories

We now define a concretization from abstract histories to concrete histories in the style of Cousot and Cousot [20]. $\text{op}[e]$ and $\text{arg}[e]$ denote the operation and argument vector of an event $e$. The domain of all arguments is represented by the set $\text{Val}$. $\text{entry}[t]$ and $\text{exit}[t]$ denote the first and last event of a (concrete or abstract) transaction. $\text{sess}[t]$ denotes the session of transaction $t$. We write $a \overset{R}{\rightarrow} b$ if $a$ and $b$ are related by the transitive reduction of relation $R$.

Definition 5. A concretization $H \in \gamma(H)$ of an abstract history $H$ is a history for which there are
\begin{enumerate}
\item $\phi_E : \text{Ev} \rightarrow \text{Ev}$, a mapping from concrete to abstract events;
\item $\phi_T : \text{Tx} \rightarrow \text{Tx}$, a mapping from concrete to abstract transactions;
\item $\iota_S : \text{Var}_H \rightarrow \text{Val}$, a local variable interpretation for every concrete session $S$;
\item $\iota_G : \text{Var}_G \rightarrow \text{Val}$, a global variable interpretation.
\end{enumerate}
The mappings need to reflect the abstract history’s structure and invariants:
\[
\begin{align*}
e \in Ev & \implies op[e] = op[\phi_E(e)] \\
e = entry[t] & \implies \phi_E(e) = entry[\phi_T(t)] \\
e \in t \in Tx & \implies \phi_E(e) \in \phi_T(t) \\
e = exit[t] & \implies \phi_E(e) = exit[\phi_T(t)] \\
s \ni e : f \in t, s \neq t & \implies \phi_T(s) \ni \phi_T(t) \\
s \ni e : f \in t, s = t & \implies \phi_E(e) \ni \phi_E(f)
\end{align*}
\]

We call the tuple \((\phi_E, \phi_T, t_L, t_G)\) a concretization model for the pair \(H, H_e\).

It is important to see that not all histories in the concretization of an abstract history have a schedule. When using abstract histories we often consider a weakened definition of a schedule called a pre-schedule. A pre-schedule is a pair \((v_i, ar)\) which is a schedule that may violate \((S1)\). Every history has a non-empty set of possible pre-schedules. A pre-schedule is sufficient to construct a DSG.

The following lemma follows from Definition 5 and will be useful for the analysis in Section 7.

**Lemma 8.** Any set \(\gamma(H)\) of concretizations is closed under restriction to subsets of transactions.

### C Soundness of the fast serializability check

To derive a more precise criterion than simply checking the acyclicity of the SSG, we need to study the properties of cycles in DSGs first; once we have established a characterization of all feasible cycles in a DSG, we can lift it to SSGs. We start by introducing a notion of minimality of DSG cycles.

**Definition 6.** A simple cycle in a DSG is called

- **(M1)** chordless if for any two transactions \(s\) and \(t\) on the cycle \(1\) the shortest path from \(s\) to \(t\) on the cycle has length 1 or \(2\) there is no edge \((s, t)\) in the DSG
- **(M2)** \(\preceq\)-minimal if there is no \(\preceq\)-edge \((s, t)\) in the cycle, for which there is a differently labeled edge \((s, t)\) in the graph
- **(M3)** minimal if it chordless and \(\preceq\)-minimal

Clearly, if there is a cycle in a DSG, then there is also a minimal cycle. Intuitively, given any cycle, we can acquire a minimal cycle by shortcutting the cycle using all chords in the cycle and then replacing all \(\preceq\)-edges in the cycle that violate \((M2)\) by a parallel, differently labeled edge. We say that a cycle in a DSG contains an event \(e\) if the cycle contains a transaction that contains \(e\).

**Lemma 9.** For every minimal cycle in a DSG the following three properties hold:

- **(C1)** The cycle contains at least two \(\preceq\)-edges or at least one \(\preceq\)-edge and one \(\preceq\)-edge.
- **(C2)** It contains at most two transactions from each session. If it contains two transactions from a session, they are adjacent in the cycle.
- **(C3)** At least one of the following two properties hold:
  - **(C3a)** It contains two transactions \(s \neq t\) and two updates \(u \in s, v \in t\) such that \(u \preceq v\).
  - **(C3b)** It contains a transaction \(s\) with a query \(q\) and an update \(v\) such that \(q \preceq u, v\) as well as some update \(u \in t\) and some event \(e \in t'\) where \(t, t'\) are other transactions on the cycle with \(s \neq t, s \neq t', q \preceq u, v\), and either \(e \preceq u, v\) or \(e \preceq u\).

**Proof.** We will show each part of the claim separately.

**Proof of (C1):** We will use three properties of the lifting to transactions defined in Section 4.2, which follow directly from its definition:

- It preserves subset relationships, i.e. \(R \subseteq T \implies \hat{R} \subseteq \hat{T}\) for all \(R, T \subseteq (Ev \times Ev)\).
- Under atomic visibility \((S3)\), \(\hat{ar}\) has no cycles of size \(n > 1\) (this would require a cyclic ar).
- Under causal consistency \((S2)\), \(\hat{ar}\) has no loops (this would require a cycle in so \(\cup\) ar).
- If \(R\) is transitive, then \(\hat{R}\) is also transitive.

Since we assume both atomic visibility and causal consistency, the lifting of \(\hat{ar}\) is acyclic. By \((S2)\), so \(\subseteq v_i \subseteq ar\) and by our observation above, we get \(s_0 \subseteq v_i \subseteq \hat{ar}\). Therefore, \(s_0 \cup v_i \cup \hat{ar}\) is acyclic.

Now, we know that \(\odot \subseteq v_i\) and \(\odot \subseteq ar\) and therefore \(\odot \subseteq v_i\) and \(\odot \subseteq \hat{ar}\), such that we can conclude that even \(s_0 \cup \odot \cup \hat{ar}\) is acyclic. Therefore a cycle must contain at least one \(\odot\)-edge.

Let this \(\odot\)-edge be from \(s\) to \(t\). Then there must be a path from \(t\) to \(s\), let us assume for now it is only using edges labeled \(\odot\) and \(\odot\). Then since \(\odot \cup \odot \subseteq v_i\) and \(\odot\) transitive, we have \(t \odot s\). This, however, contradicts that there is an anti-dependency in the opposite direction by \((D2)\). Therefore, the path from \(t\) to \(s\) must involve at least one \(\odot\)-edge or one \(\odot\)-edge.

**Proof of (C2):** Assume there are (at least) three transactions from one session in the minimal cycle. Since \(s_0\) is total on the session, we can give them names \(s, t, u\) such that \(s \odot t \odot u\) and also \(s \odot u\). Clearly, an edge from \(s\) to \(t\), as well as an edge from \(t\) to \(u\), must be part of the cycle, otherwise, minimality is violated because the corresponding \(\odot\)-edges would form chords. But then the \(\odot\)-edge from \(s\) to \(u\) forms a chord, again violating minimality of the cycle. Adjacency of the two transactions can be shown in the same way.

**Proof of (C3):** By \((C1)\), every minimal cycle in a DSG must contain a sub-path

\[
\begin{align*}
s' \overset{\odot \cup\hat{u}}{\longrightarrow} t', (\odot \cup\hat{u})' \longrightarrow s \overset{\odot}{\longrightarrow} t
\end{align*}
\]

Here, we know that \(s' \neq t', s \neq t, s \neq s', t \neq t'\) by the cycle being minimal, however, we may have \(s = t'\) or \(t = s'\). By the definition of the lifting to transactions, we can find...
corresponding events \( e \in s', v \in t', q \in s, u \in t \) with the following relations (remember that \( s \) is the equivalence relation induced by transactions on events).

\[
\begin{array}{c}
e \xrightarrow{(\ominus \cup \odot) \setminus \text{st}} v \xrightarrow{[\text{so} \cup \text{last}]^{+}} q \xrightarrow{\ominus \setminus \text{st}} u
\end{array}
\]

Here, \( u = e \) is possible, otherwise, all events are distinct from each other.

Let's now assume the contrary of (C3), that is, \( u \not\succ v \), and \( s \not\succ t' \) or \( q \not\succ v \).

By (D2) and \( u \not\succ v \), having \( u \not\succ v \) would make \( q \not\succ u \) impossible, therefore we must have \( v \not\succ u \).

If \( s = t' \), then by assumption \( v \not\succ q \) (since \( s \) is total inside transactions), therefore \( v \not\succ u \). Similarly, if \( s \not\succ t' \) then by assumption \( t' \not\succ s \), and therefore \( v \not\succ q \) as well.

We can conclude that under this assumption we have the following form:

\[
\begin{array}{c}
e \xrightarrow{(\ominus \cup \odot) \setminus \text{st}} v \xrightarrow{\ominus \setminus \text{st}} u
\end{array}
\]

If \( u \not\succ v \) then \( t' \not\succ u \), whereby chordlessness (in case \( t' = s \)) or \( \ominus \)-minimality (in case \( t \neq s \)) would be violated, so we can assume that \( u \not\succ v \). But then, since \( u \) and \( v \) absorb each other and commute at the same time, they are equivalent:

\[
\begin{array}{c}
u \equiv u \equiv u \equiv v
\end{array}
\]

By the existence of the \( \ominus \cup \odot \) relation between \( v \) and \( e \) and definition (D2) and (D3), we have \( u \not\succ e \). Since \( u \equiv u \), this gives us \( u \not\succ e \).

We now check two cases:

**Case 1.** \( e \not\succ \ominus v \). Then \( e \not\succ u \) due to the acyclicity of \( ar \) and with \( e \not\succ u \) we get \( e \not\succ u \). By (S3), we get that \( s' \neq t \) (we have \( e \not\succ u \) with \( v \) in a different transaction that \( e \) and \( u \)). Now since \( s' \neq t \), \( e \not\succ u \) lifts to \( s' \not\succ u \), violating the chordlessness of the cycle since it shortcuits the path through \( s', t, s \) of length \( \geq 2 \).

**Case 2.** \( e \not\succ \ominus v \). We must have \( e \not\succ u \), since \( u \not\succ e \) and if there was a \( w \) with \( u \not\succ w \not\succ e \) and \( u \not\succ w \), then also \( v \not\succ w \) and \( v \not\succ e \) and therefore \( e \not\succ v \) would be infeasible. Now, if \( t = s' \), we must have \( e \not\succ u \) (since otherwise \( u \not\succ e \)), but this violates our assumption that (C3) does not hold. On the other hand, if \( t \neq s' \), then \( e \not\succ u \) lifts to \( s' \not\succ u \), violating the minimality of the cycle since it shortcuits the path through \( s', s, t \) of length \( \geq 2 \).

We can conclude that if we assume the contrary of (C3), there are no feasible minimal cycles, which means the opposite must be true for all minimal cycles.

The lemma lets us rule out cycles that do not satisfy all three properties as potential serializability violations. The first two properties are adopted from different but related settings. (C1) has previously been observed for cycles in causal consistency (e.g., Bernardi and Gotsman [10]) and is useful to rule out cycles without \( \Theta \)-edges, as described later. (C2) applies to all models in which a total session or program order is to be preserved [35, Theorem 3.9]; we will use it in the logical encoding of Section 7.

Property (C3) is novel and lets us rule out even more cycles, namely those where all updates in the cycle absorb each other (C3a) and at the same time, there is no transaction \( s \) that orders a query \( q \) before an update \( u \), and the ordering constraints in the DSG force other transactions (the other nodes in the cycle) to be ordered between the two events (C3b). More precisely, (C3b) requires \( q \) to form an anti-dependency to some other transaction \( t \) in the cycle, and \( s \) to be the target of a conflict- or anti-dependency edge from some other transaction \( t' \) in the cycle. Intuitively, the non-overlapping constraint of serial schedule (S3) is one cause of the cycle.

We can lift some of the properties from DSGs to SSGs, and state the soundness theorem for cycle detection in SSGs.

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

**SC1** It contains at least two \( \Theta \)-edges or at least one \( \Theta \)-edge and one \( \ominus \)-edge.

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

**SC2a** It contains two abstract events \( u, v \in \cup \) such that \( \neg \text{abs}(\text{op}[u], \text{op}[v]) \) is satisfiable.

**SC2b** It contains two abstract events \( q \in \cup, u \in \cup \) with \( q \not\succ u \), as well as two abstract events \( v \not\succ e \) such that both \( \neg \text{com}(\text{op}[u],\text{op}[e]) \) and \( \neg \text{com}(\text{op}[q],\text{op}[v]) \) are satisfiable.

**Proof.** Let \( C \) be the cycle in \( G \), we can assume that it is minimal. Let \( \phi_T, \phi_E \) be the transaction and event mapping of the concretization model of the history of \( G \). For the existence of a closed walk in \( G \), it is sufficient that all edges in \( C \) can be lifted to \( G \) via \( \phi_T \). For each edge in \( (s, t) \in C \), we have two cases: (Case A) \((s, t) \in \Theta\). Then \( \phi_T(s), \phi_T(t) \in \Theta \) is an edge in \( G \); (Case B) \((s, t) \in \ominus \cup \Theta \). Then there must be two events \( e \in s \) and \( f \in t \), such that \( (e, f) \in \ominus \cup \Theta \), and therefore \( e \not\succ f \). By contraposition of Definition 2, we get that \( \neg \text{com}(\text{op}[e], \text{op}[f]) \) must be satisfiable. There are two abstract events \( \phi_E(e) \in \phi_T(s) \) and \( \phi_E(f) \in \phi_T(t) \) with \( \text{op}[\phi_E(e)] = \text{op}[e] \) and \( \text{op}[\phi_E(f)] = \text{op}[f] \). Then also \( \neg \text{com}(\text{op}[\phi_E(e)], \text{op}[\phi_E(f)]) \) by which we can conclude that there is an edge from \( \phi_T(s) \) to \( \phi_T(s) \) in \( G \).

**SC1:** This follows directly from the lifting above.
in which every abstract cycle of $H$ has a corresponding abstract cycle which is induced by a cycle with an injective concretization model in one of the abstract histories in $U_k[H]$. More precisely, we require the existence of a function $A_k$ between $\text{AbsCycles}_k[H]$ and the set $\text{InjCycles}_k[U_k[H]]$ of abstract cycles in any abstract history of $U_k[H]$ induced by DSG cycles that span at most $k$ sessions and have injective concretization models. $A_k$ need not be surjective, that is, we allow our unfolding to contain cycles that were not possible in the original abstract history. Our SMT-check always reports the cycles in $\text{InjCycles}_k[U_k[H]]$; therefore, it may contain false positives.

We now prove (U1),(U2) from Section 7 as the following

**Theorem 10.** For every sequence of unfoldings $U_k[H]$ of an abstract history $h$ there exists a function $A_k : \text{AbsCycles}_k[H] \rightarrow \text{InjCycles}_k[U_k[H]]$.

**Proof.** We will only give a proof sketch. A schematic picture of the argument is shown in Figure 15. We show that the function $A_k$ exists by lifting a transformation $P : S \rightarrow S'$, where $S$ and $S'$ are representative sets of DSG cycles, which together induce $\text{AbsCycles}_k(H)$ and $\text{InjCycles}_k[U_k[H]]$, respectively. Given a cycle $C \in S$, the transformation $P$ removes events from the cycle’s history $H$ to produce a history $H'$ such that $C$ projects to a DSG cycle $C' \in S'$ of $H'$. We choose the two sets $S$ and $S'$ so that:

1. the histories of all the cycles in $S$ and $S'$ have no more than $k$ sessions;
2. the cycles in $S$ have histories in $\gamma(H)$, the cycles in $S'$ have histories in $\gamma(H'), H' \in U_k[H]$;
3. every abstract cycle $C \in \text{AbsCycles}_k[H]$ is induced by a minimal DSG cycle $C \in S$;
4. the history of every DSG cycle in $S'$ has an injective concretization model for an abstract history in $U_k[H]$.

![Figure 15. Schematic overview of the soundness argument of the unfolding](image-url)
We lift the transformation \( P \) to the function \( A_k \) by fixing a representative DSG cycle \( C \in \mathcal{S} \) for each abstract cycle \( \mathcal{C} \in \text{AbsCycles}_{\mathcal{A}_k}[H] \). Then, to map a given abstract cycle \( \mathcal{C} \) we project its chosen representative to a DSG cycle \( C' \) and take the abstract cycle that \( C' \) induces in an abstract history in \( U_k[H] \).

A set of concrete witnesses: To define the projection’s domain \( \mathcal{S} \), we crucially rely on the locality of our serializability criterion from Section 4 (cf. lower left square in Figure 15). Every abstract cycle \( \mathcal{C} \in \text{AbsCycles}_{\mathcal{A}_k}[H] \) is induced by some minimal DSG cycle \( C \) in a history \( H_0 \in \gamma(H) \). Together with Lemma 8, locality implies that the same \( C \) also exists in a history \( H \), in which it traverses every transaction \( H \) can be constructed by simply removing all transactions from \( H_0 \) which are not on \( C \). All cycles \( C \) define our domain \( \mathcal{S} \). Minimality and the absence of extra transactions imply that the histories of cycles in \( \mathcal{S} \) are very structured: they have at most \( k \) sessions, and at most two transactions per session (cf. Lemma 9).

To show that the required injective concretization models exist, we start with a DSG cycle \( C \in \mathcal{S} \) and a concretization model \( (\phi_{E}, \phi_{T}, t_{\text{fst}}, t_{\text{snd}}) \) of the cycle’s history \( H \in \gamma(H) \). We remove some of the events of \( H \) to obtain \( H' \) with an injective concretization model \( (\phi'_{E}, \phi'_{T}, t'_{\text{fst}}, t'_{\text{snd}}) \), as we explain next.

Injectivity of \( \phi'_{T} \): Recall that we consider a DSG cycle \( C \in \mathcal{S} \) of a history \( H \in \gamma(H) \). By our definition of \( \mathcal{S} \), \( H \) contains at most two transactions \( s_{\text{fst}}, s_{\text{snd}} \) for every session \( i \). By the construction of the unfolding of abstract histories, there must be an \( H' \in U_k[H] \) with exactly the same number of abstract transactions per session (modulo renaming). If the \( \phi_{T} \) maps both \( s_{j} \) to an abstract transaction \( t \) of \( H \) then we let \( \phi'_{T}(s_{j}) = t'_{j} \), where \( t'_{\text{fst}}, t'_{\text{snd}} \) denote the unfolded copies of \( t \). Since then, all transactions in \( H \) map to different abstract transactions in \( H' \). \( \phi'_{T} \) is indeed injective.

Injectivity of \( \phi'_{E} \): It remains to show that the construction of the unfolding of individual transactions of \( H \) ensure that an injective \( \phi'_{E} \) exists. We will ensure that a projection \( H' \) of \( H \) onto a subset of its events maps injectively to the corresponding unfolding \( H' \) (cf. Figure 15). The key observation is that the DSG cycle \( C \) is induced by a set of events \( E \) and edges in \( \mathcal{S}_{o} \cup \mathcal{S}_{a} \cup \mathcal{S}_{c} \cup \mathcal{S}_{r} \). The minimality of \( C \) implies that \( E \) intersects each transaction \( s \) at most two events: one, \( e_{\text{in}} \), for the edge that enters \( s \) and one, \( e_{\text{out}} \), for the edge that leaves \( s \). Unfolding (and then duplicating) all transactions guarantees that the whole set \( E \) is part of some concretization \( H' \). By the locality of our criterion, the same set of events induces a DSG cycle \( C' \) of \( H' \).

If the abstract event order \( \mathcal{E}_{o} \) is acyclic, we are done: Duplication of transactions (Section 7.1) guarantees that any abstract event in \( t \) concretizes to at most one event in \( \mathcal{E}_{v} \) if it forms a trivial strongly connected component (SCC), and therefore the concretization model \( \phi_{E} \) will be injective. We only need to consider each non-trivial strongly connected component \( (V, E) \) of \( (\mathcal{E}_{v}, \mathcal{E}_{o}) \), and if the result of applying the transformation to every SCC is an acyclic \( (\mathcal{E}_{v}'', \mathcal{E}_{o}) \), we have ensured an injective \( \phi''_{E} \).

Let \( P_{e} \) be the acyclic path over vertices in \( t \) ordered by \( \phi_{E} \). Using \( \phi_{E} \), this path lifts to a (potentially cyclic) path \( P_{\phi} \) in \( (\mathcal{E}_{v}, \mathcal{E}_{o}) \). Let \( P \) be the maximal segment of \( P_{\phi} \) traversing only nodes from \( V \), and \( P \) the corresponding segment of \( P_{e} \). Now let \( B \) be the back edges in \( E \) of any DFS in \( \mathcal{E}_{o} \) from entry[\( \phi_{T}(t) \)]. By splitting the path \( P \) at edges from \( B \), we get \( n \) path segments \( P_{1}, \ldots, P_{n} \). This splitting induces a corresponding splitting of the underlying path \( P \) into \( P_{1}, \ldots, P_{n} \). The key insight is that for the events traversed by each \( P_{i} \), \( \phi_{E} \) already is injective since the lifting of \( P_{i} \) via \( \phi_{E} \) cannot traverse nodes repeatedly (as it lies entirely in \( (V, E \setminus B) \), a DAG).

Our transformation is based on the idea of removing all path segments \( P_{i} \) with \( 1 \leq i \leq n \) from \( H \) which do not contain the events we want to preserve, namely \( e_{\text{in}} \) and \( e_{\text{out}} \). We will assume now that there are some \( 1 \leq i \leq j \leq n \) such that \( e_{\text{in}} \in P_{i} \) and \( e_{\text{out}} \in P_{j} \), which is the most interesting case of many analogous ones (e.g. \( e_{\text{in}}, e_{\text{out}} \) are in the same segment, or only one of them is contained in \( V \)). The projected history \( H' \) is the result of removing the events on \( P \setminus (P_{1} \cup P_{j}) \) from \( H \).

To see that the above construction is correct, we define \( \phi_{E}' \) to be \( i_{1} \cdot \phi_{E} \) on \( P_{1} \), \( i_{2} \cdot \phi_{E} \) on \( P_{j} \) and \( \phi_{E} \) on all other events and show that it is a valid concretization model for \( H' \) and \( H' \). First, observe the following properties on \( \phi_{E} \), which are due to the construction of the splitting \( P_{1}, \ldots, P_{n} \). We here assume that the first event of \( P_{1} \) is not the entry, and the last event of \( P_{n} \) is no exit of the transaction, those cases follow equivalently.

1. The first event of \( P_{i} \) lifts to an element of \( I_{i} \) or \( B_{i} \).
2. The last event of \( P_{i} \) lifts to an element of \( B_{i} \).
3. The first event of \( P_{j} \) lifts to an element of \( B_{j} \).
4. The last event of \( P_{j} \) lifts to an element of \( B_{i} \) or \( O_{i} \).

In other words, the lifted segment \( P_{j} \) can only be entered via a back-edge or from outside the SCC (in the case \( i = 1 \)) and exited via a back-edge. Similarly, the lifted segment \( P_{j} \) can only be entered via a back-edge and exited via a back-edge or going outside the SCC (in the case \( j = n \)). Based on these observations, we can easily check that so edge in \( H' \) maps to an \( \mathcal{E}_{o} \) edge in \( H' \) via \( \phi_{E}' \). What remains to see is that the invariants on the edges in the lifting are in fact satisfied by the underlying events. However, this is trivial, since invariants on \( R \) remain unchanged, and all other invariants are set to \( \top \).

\[ \square \]

We extend soundness result above informally to show also that we find an overapproximation of all abstract cycles in the unfoldings. Observe, that we can transform an abstract cycle \( C \) in an abstract history in \( U_k[H] \) into an abstract cycle \( O[C] \).
in $H$ by reverting the renaming from Section 7.1, i.e. renaming $t^t_i$ and $t^t_{in}$ back to $t$ for all $i = 1...k$ and $t \in T_x$. By pointwise application of $O[-]$, we can map $lnCycles_k[U_k[H]]$ to a set $AbsCycles_k^H[H]$. Following the construction of $A_k$ above, we can determine that

$$AbsCycles_k^H[H] \supseteq AbsCycles_k[H]$$

Our algorithm will exploit that fact, by finding a set of abstract cycles $S$ that subsumes $AbsCycles_k^H[H]$ and thereby also $AbsCycles_k[H]$.  

**Discussion.** In the unfolding presented above, we only preserved two path segments which are minimally required for soundness. We can, however, decide to preserve any constant number of additional path segments. In practice, it can make sense to also preserve the first and the last path segment as well, because, otherwise, we can lose some precision in the unfolding (e.g. invariants that hold when entering or exiting a loop).

## E  DSG cycle query

In Figure 16, we present an SMT query whose logical models capture DSG cycles witnessed by injective concretization models of an abstract history in the sequence $U_k[H]$. We ask the solver to pick a pre-schedule, an ordering of the events in each transaction, arguments for all events, and finally a “bad” transaction that is part of a cycle in the DSG.

We will now briefly explain each conjunct of the formula. Conjunct (pre-schedule) requires that $vi$ and $ar$ define a pre-schedule over the transactions in $T_x$. Conjunct (process) requires that $eo$ defines a path through the abstract event order of each transaction, where $Path(R,b,c)$ constrains $R$ to be the transitive reduction of a chain from $b$ to $c$. Furthermore, we require that the invariants from $Inv$ on that path are satisfied by the arguments of the events. Conjunct (active) defines an auxiliary set active, which is used to track events that are part of the solver-chosen event order $eo$. Only such events can form dependencies. Conjunct (dsg) defines a relation $dsg$ representing the dependency graph, using the helper formulas $\phi_{\exists}$, $\phi_{\forall}$, and $\phi_{\sigma}$. Figure 16c shows how $\phi_{\sigma}$ is defined, mostly replicating (D1) ($\phi_{\exists}$ and $\phi_{\forall}$ are analogous). We just need to ensure that only events that lay on the execution path (a subset of $eo$) can form dependencies, for which we use active. Finally, Figure 16a uses $\phi_{dsg}$ to define a formula that has a model exactly if there is a cycle in the DSG.

**Decidability.** All sets and relations that we quantify over in $\phi_{cyclic}$ are finite, so they unfold into boolean constants representing set-membership. Similarly, universal quantifiers over sets unfold into conjunctions, subset-constraints unfold into implications, etc. Furthermore, we can also unfold arg into a bounded number of Val-sorted constants. Similarly, we can eliminate transitive closure operators $dsg^+$ and $eo^+$ by defining helper relations that are axiomatized to be the transitive closure of $dsg$ and $eo$, respectively [25]. So, if $Inv(\cdot)$, $com(\cdot)$, and $abs(\cdot)$ are expressed in a decidable fragment then $\phi_{cyclic}$ is decidable as a boolean combination of decidable formulas.

**Discussion.** We could instead define a single abstract history that combines all $k$-unfoldings and define an SMT-check directly for this one unfolding. Then, the SMT-solver would choose which subset of transactions are chosen to construct a cycle in the DSG. This, however, would make the query significantly larger since we would need to encode the transitive closure of relations which are of size proportional to $T_x$, increasing the query size by $\Theta(|T_x|^3)$. As the number of transactions increases, this quickly leads to sizes that modern SMT-solvers cannot handle. Therefore, we decided to split the query into multiple queries, motivated by the following empirical observation, which we validated on our benchmarks in Section 9: when applications get larger, the number of transactions grows, but the average number of events per transaction stays roughly constant. Counting all possible transaction choices, we obtain $O(|T_x|^k)$ queries of size $O(poly(|Ev|))$ where $k = 2$ in practice. We found the multiple queries to perform better than one large query.

## F  From k to any: checking for termination

**Assumptions.** We make an extra assumptions in this section, which are satisfied in our systems and benchmarks, but which is only required for the termination check to work. We require the property on events, that for all $u,v \in U$, $q \in Q$ we have

$$u \circ v \land v \supseteq q \implies u \supseteq q \tag{1}$$

That is, if an update $v$ absorbs $u$ and is non-commutative with $q$, then $u$ must also be non-commutative with $q$.

This condition holds for the dictionary in Figure 6. For a put-event $u$ to absorb a put-event $v$ and $v$ to be non-commuting with a get-query $q$, they all must access the same key $k$. But of course, then all put-updates to $k$ are non-commuting with $q$, in particular, $u$.

**SMT-encoding.** We first provide an SMT-encoding of the question whether a particular sub-path of a cycle can be cut short. Let $H$ be an abstract history. We again use its sequence of unfoldings $U_k[H]$ to get abstract histories in which all possible $k$-session path fragments have an injective concretization model. For each such unfolding $H' \in U_k[H]$,
Algorithm 2 checks a given abstract history for serializability for an arbitrary number of sessions. The main function CheckArbitrary starts with $k = 2$ sessions, and then iteratively increases $k$ while computing unfolded abstract histories for $k$ sessions (see Section 7.1) and then detecting all violations for $k$ sessions using CheckBounded from the previous section. We add a parameter $V$ to CheckBounded that initializes the variable $V$ in the function to a provided value so that we do not detect violations that are subsumed by $V$ from earlier iterations. After every iteration, termination of the loop is checked using the Generalizes function.

Generalizes works similarly to CheckBounded, in that it iterates over the possible $k$-unfoldings and for each that does not contain a previously detected violation, it encodes and checks the satisfiability of $\phi_{\text{term}}$. Here, ChainPossible takes a similar role as CyclePossible, only that it checks using the SSG, whether there may be a history in the concretization of $H$, which has a DSG which contains a Hamilton path.

Theorem 11. If Generalizes returns true, any set of abstract cycles $V$ that subsumes $\text{AbsCycles}_k[H]$ also subsumes $\text{AbsCycles}[H]$.

Proof. We only give a proof sketch. Assume there is an abstract cycle $C \subseteq \text{AbsCycles}[H] \setminus \text{AbsCycles}_k[H]$, which is not subsumed by any $C' \in V$. By the definition of such an abstract cycle $C$, we know that (1) there is a minimal cycle $C_{>k}$ in a DSG of a history $H \in \gamma(H)$ and pre-schedule $(\nu, \alpha)$ spanning more than $k$ sessions and whose transactions can be mapped to abstract transactions from $C$ and (2) there is no cycle $C_{\leq k}$ in any other history in the concretization of $H$, spanning $k$ or less sessions and whose transactions can be mapped to abstract transactions from $C$. Again, we can assume that $H$ contains only the transactions also traversed by $C_{>k}$.
\[ \phi_{\text{term}}(H) = \phi_{\text{dsg}}(H) \land \text{Path}(P, \text{dsg}) \]
\[ \land \forall i, j \in [1, k], \exists m \in [1, k], t_{\text{snd}}^i \rightarrow L_{\text{snd}} \quad \exists \rho \quad t_{\text{fst}}^j \]
\[ \implies \forall i, m, \left( \begin{array}{l}
\forall L, t \xrightarrow{\text{ar}} t_{\text{snd}}^i \\
\forall L, t_{\text{snd}} \xrightarrow{\text{wr}} t \\
\phi_{\text{dsg}}(H_m) \\
\neg \text{Adep}(t_{\text{snd}}^i, t_{\text{snd}}^m, t_{\text{fst}}^j) \end{array} \right) \]

**Figure 17.** The formula to check termination

The pre-schedule \((v_i, \text{ar})\) can be lifted to \(v_i, \text{ar} \cdot H, v_i, \text{ar}\), projected down to some subset of \(k\) sessions must provide a model to the term \(\phi_{\text{dsg}}(H) \land \text{Chain}(P)\) in one of the abstract histories enumerated in **Generalizes**. Let us call the enumerated abstract history \(H'\) and the model \(M\), representing a history and a pre-schedule. Assuming the correct implementation of the sub-procedures, **ChainPossible** must be true and \(\neg \text{Subsumed}(H', V)\) must return true as well, as we cannot have found a subsuming violation by our assumption. Therefore, it is sufficient to show that the second part of \(\phi_{\text{term}}\) must be true given \(M\).

Assume the opposite: There are two sessions \(i, j\), such that \(j\) is preceded in \(P\) by some session \(m \neq i\) through an anti-dependency. Furthermore, there is an extension of model \(M' \supseteq M\) by assigning the variables in \(V_i, m\) to a new concrete history and schedule which models \(\phi_{\text{dsg}}(H_{i, m})\), such that \(t_{\text{snd}}^i\) is arbitrated after all other transactions, and not visible to any. For the sub-formula to be false, in this history and schedule, there must be an edge \(t_{\text{snd}}^i \rightarrow t_{\text{snd}}^m\), which short-cuts the sequence \(t_{\text{snd}}^i \rightarrow \text{dsg}^i \rightarrow \text{snd}^i \rightarrow \text{fst}^i\), which short-cuts the sequence of transactions \(t_{\text{fst}}^i, t_{\text{snd}}^i, \ldots\) is injectively mapped to the set of abstract transactions \(t_{\text{fst}}^i, t_{\text{snd}}^i, \ldots\). What remains to show is that from the short-cut of the path we can construct a cycle \(C_{\geq k}\). Consider again the original history and pre-schedule \(H, v_i, \text{ar}\) containing \(C_{\geq k}\). We can assume \(H\) contains only the transactions involved in the cycle (by Lemma 8). We extend it by inserting \(t_{\text{snd}}^i\) from \(M'\) at the end of the arbitration order and the session order of session \(i\) while adopting \(v_i\) in the minimal necessary way to satisfy the properties of a pre-schedule; in particular, all transactions observed by \(t_{\text{snd}}^i\) are made visible to \(t_{\text{snd}}^i\) as well to satisfy causal consistency.

Now, observe that the resulting history and pre-schedule yield a DSG with a cycle spanning at least one session fewer than \(C_{\geq k}\). Inserting a transaction at the end of the arbitration order without making it visible cannot eliminate any of the existing edges from \(C_{\geq k}\) (e.g. by absorption). Furthermore, since there is an anti-dependency from \(t_{\text{snd}}^i\) to \(t_{\text{fst}}^j\) in \(H_M, v_M, \text{ar}_M\), it must also exist in \(H\), unless we have absorption by some other transaction \(s\) that was visible to \(t_{\text{snd}}^i\) and is now also visible to \(t_{\text{snd}}^j\).

Since \(s\) is not in \(H_M\), but must be on the cycle, its distance on the cycle from \(t_{\text{snd}}^m\) is at least two (the cycle passes through session \(j\) first, which is completely inside \(H_M\)). \(s\) cannot be visible to \(t_{\text{snd}}^m\), since otherwise, \(t_{\text{snd}}^m\) would not anti-depend on \(t_{\text{fst}}^i\) (since an event in \(s\) breaks the anti-dependency by absorption).

By ADep, there must be a \(q \in t_{\text{snd}}^i, a' \in t_{\text{snd}}^m\) and \(a \in t_{\text{fst}}^i\), such that \(q \xrightarrow{\text{wr}} u\) and \(a' \xrightarrow{\text{wr}} u\). Now let \(u \in s\) be the update that absorbs \(u\). Then \(t_{\text{snd}}^m\) must anti-depend on \(s\), because by \(u \rightarrow v, u \rightarrow q'\) and the contrapositive of Equation (1), we get \(v \not\rightarrow q'\) (we can assume w.l.o.g. that \(v\) is the last such update by arbitration, and is not absorbed afterwards). This, however, breaks minimality of the cycle, because it now contains a \(\ominus\)-chord. Therefore, \(s\) cannot exist.

We have shown that we can short-cut any minimal \(C_{\geq k}\) to a smaller cycle by changing the history. By induction, it follows that we can construct a \(C_{\leq k}\) by iterative short-cutting.

Finally, observe that, modulo renaming via \(O[\cdot]\), the abstract cycle \(C\) induced by \(C_{\leq k}\) contains a superset of the abstract transactions of the abstract cycle \(C'\) induced by \(C_{\leq k}\). Therefore \(C\) is subsumed by \(C'\). But \(C'\) must be in \(V\) or subsumed by some abstract cycle \(C'' \in V\). Since subsumption is transitive, in both cases, we can conclude that \(C\) is subsumed by some \(C'' \in V\), which shows our claim.

**Discussion.** **Generalizes** only checks a sufficient condition for termination. That is, even after having found a subsuming set of violations, it is possible that Algorithm 2 keeps looping forever. The reason is that short-cutting is not possible in all cases. In particular, abstract transactions may allow different concrete instantiations depending on which session they are executed on using session-local variables \(\text{Var}_{\text{L}}\). Consider, e.g., a queuing implementation, in which applications take fixed roles as writers and readers, based on a parameter given to each session. Then there could be a path spanning 3 sessions as in Figure 9, where three sessions 1, 2 and 3 have the roles...
writer, reader, and writer, then such a path could likely not be short-cut because $t_2^{nd}$ is a transaction executed on reader-sessions, while session 1 is a writer. In the experiments in Section 9, we will see that for the simple applications we analyzed, GENERALIZES always returns true for $k = 2$.

G Atomic sets with high-level data types

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, by focusing the analysis on so-called atomic sets, logically-related subsets of the data in the application [40]. In our context, an atomic set is a subset of all objects in a data store; a serializability criterion based on atomic sets requires the existence of a serial order on the events on objects only within each atomic set, but not across different sets. This allows different clients to have inconsistent views of unrelated data. In this section, we generalize atomic sets to high-level data types by providing support for container data types and by integrating them with commutativity specifications for data type operations.

Atomic sets. In the example from Figure 18a, two orthogonal sessions create inconsistent views of data: the left session updates the balance of an account and then retrieves profile information (such as the width and height of a profile picture). Concurrently, a second session updates the profile picture and then checks the balance. This results in a true serializability violation: in no serial execution, the left session can observe the old profile after updating the balance, while the right session observes the old balance after updating the profile. However, in many such cases, the correctness of the application does not require a consistent view of the profile picture and the unrelated balance. Atomic sets provide a way to express developer intent: an atomic set is a uniquely named set of program locations for which serializability is checked separately. In our example, placing the picture dimensions and the balance into different atomic sets avoids the undesired serializability error. We will discuss in Appendix G.2 how to associate objects to atomic sets.

Atomic sets on containers. Whereas previous work uses atomic sets on individual objects, we extend them to support containers such as lists or maps. The example in Figure 18b motivates this extension. Here, both sessions update the time a user ($A$ and $B$, resp.) was last seen. Later, a prune transaction retrieves all users and removes inactive ones. At first, this seems like a clear serializability violation, because both prune-transactions have inconsistent views of the Users map. However, since the prune transaction processes each member of the Users map separately (Figure 18c), the inconsistent views of the entire map do not matter. To avoid undesired alarms in such cases, we allow each entry of a container to be in its own atomic set.

For the purpose of this explanation, assume a container is a table where keys $K = \{k_0, ..., k_{m-1}\}$ select columns, and each row defines a value $v_i$ for each key $k_i$. For a given set of relevant keys $K' = \{i_0, ..., i_{m-1}\} \subseteq [0, n - 1]$, we define an unbounded number of atomic sets $\{A_{v_{i_0}, ..., v_{i_m}} | v_{i_j} \text{ in the domain of } k_{i_j}\}$. Defining atomic sets based on the values found in the table is very expressive. Assume for instance a table appointments whose first column contains dates and the second column contains times. Using the above definition, we can express that the times must be consistent within one day by choosing $K' = \{0\}$ and putting all events of the same day into the same atomic set, but events of different days in different atomic sets. In Figure 18b, we can introduce two distinct atomic sets Users-$A'$ and Users-$B'$ for the Users table. Thereby, the violation in Figure 18b will not be flagged, because it involves an inconsistent view of two different atomic sets, which is permitted.

G.1 Integration into our framework

We cannot directly define atomic set serializability with our specifications in Definition 2, since commutativity and absorption have no notion of distinct objects, and instead define logical conditions over tuples of arguments to object-less operations. However, the required information can be completely expressed through a modified notion of commutativity, thus requiring no changes to our formal framework. We will now give an informal overview.

Checking serializability for an atomic set corresponds to weakening the sequential specification of queries, so that state not part of a given atomic set behaves non-deterministically. For example, if our atomic set contains just the row $A$ then our sequential specification for it would allow the following event sequence, where $C$ is any sequence of rows including $A$, but not necessarily $B$:

$$T.add_row(A);T.add_row(B);T.get_all:C$$

To achieve this behavior, we redefine our sequential specifications as parametric w.r.t. a predicate $\text{INAtomicSet}(\cdot)$, which takes as argument a tuple representing an object in the system, and return whether the object is contained in the current atomic set. For example, the operation $T.get\_all:C$ is legal after an event sequence $a$ in the modified sequential specification if for any row $r$:

1. $\neg \text{INAtomicSet}(T, r)$ or
2. $r \in C \iff r \in C'$ for some legal $a; T.get\_all:C$

The sequential specification gives rise to a corresponding commutativity specification, which is also parametric w.r.t. the $\text{INAtomicSet}$:

$$\text{com}(T.add\_row(?), T.get\_all(?))(e, f) = \neg \text{INAtomicSet}(T, \text{arg}[e]_0))$$


AtomicSet accesses The above approach relies on a mapping from objects to consistent values. For independent iterations, we assume heuristically that atomicity between iterations can be safely violated, as in the prune transaction (Figure 18c).

SMT encoding. As explained in Section 7, the model returned by the SMT solver represents a serializability violation. To incorporate atomic sets, we need to refine the SMT query such that models represent violations within one atomic set. For this purpose, we introduce a boolean variable AtomicSet_A for each atomic set name A in a given program and assume that exactly one of these variables is true. Moreover, for each container, we introduce a variable Key_i for each relevant key k_i. So when computing a model, the SMT solver selects exactly one atomic set A (by setting AtomicSet_A to true) and, if applicable, a value for each relevant key. The predicate InAtomicSet in the commutativity specification above can then be defined as follows:

\[ \forall x. \text{InAtomicSet}((T, x)) \iff \text{AtomicSet}_A \land x = \text{Key}_0 \]

It connects table T to the atomic set A_x.

\section{G.2 Heuristic}

The above approach relies on a mapping from objects to atomic sets. This mapping could be provided by the developer through the annotations introduced in the previous subsection. To alleviate this (small) overhead, we propose the following heuristic, which we also applied in our experiments (see next section).

\textbf{Heuristic 1 (Independent use heuristic).} If there is an abstract transaction \( T \in \text{Tx} \) that may access two objects \( o, p \) then there is an atomic set \( A \in \text{AtomicSets} \) such that \( o, p \in A \) unless \( T \) accesses \( o \) and \( p \) only in independent iterations of the same loop.

This heuristic reflects that if \( o \) and \( p \) are accessed by the same transaction, this transaction is likely to rely on observing consistent values. For independent iterations, we assume heuristically that atomicity between iterations can be safely violated, as in the prune transaction (Figure 18c). We take some liberty in defining independent iterations: for example, we ignore dependencies that affect only the order of appearance in user-interface elements.

\section{H Implementation}

We implemented the concepts introduced in this chapter in a static analysis back end called C^4. C^4 is designed to be independent of the data store, data store API, and programming language, and thus to serve as a basis for the analysis of any kind of system that satisfies our assumptions stated in the introduction to this chapter: atomic visibility and causal consistency. Our tool can, therefore, be used as a back end for a broad range of static analyses: in the context of distributed systems and databases it can be applied to weakly consistent mobile synchronization frameworks like TouchDevelop, to distributed databases like Antidote [2], Walter [36], CORS [33], and Eiger [34], and even to traditional relational databases with guarantees weaker than serializability but at least as strong as our model, e.g., Snapshot Isolation as implemented in MySQL or Oracle. One could even apply our tool to weak memory models like TSO by treating each operation as a separate transaction; however, applications in this space are unlikely to fulfill the assumptions that make our analysis precise for mobile and web applications.

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 section.

The general flow of C^4 is shown in Figure 19. The main component of the implementation is the back end, which takes as input a description of an abstract history (Definition 1), enriched with extra information on uniqueness and monotonicity of values (Section 8), as well as display code and atomic sets (Section 9), together with commutativity and absorption specifications for all operations used in the history (Definition 2). The input format is JSON. The tool implements Algorithm 1. When the solver terminates, a set

![Figure 18. Examples of harmless violations](image-url)
of violations is returned that subsumes the set of all abstract cycles².

The main work of a front end to C⁴ is to infer a sound abstract history from an application using a replicated data store. How this is done in detail depends on the database, the database interfacing library, and the programming language, and, furthermore, can typically be solved using a recombination of existing program analysis technology, which is why we focused our technical exposition on the back end up to here. In the rest of this section, we will briefly explain how to implement a front end for two example systems, TouchDevelop and Cassandra/Java.

H.1  Front end for TouchDevelop

TouchDevelop [38] is programming language and environment for developing mobile applications. It has a built-in system for replication called cloud types [17], which can be used by application developers, e.g. to replicate user data between devices or build collaboration features. First, we describe the consistency model, the programmer-facing API and the synchronization features of TouchDevelop briefly. Then, we show how we can adapt an existing static analyzer to produce abstract histories.

Consistency model, cloud types and synchronization. TouchDevelop employs the global sequence protocol [19] to provide prefix consistency, which is stronger than causal consistency, and can therefore be used with our criterion. Prefix consistency defines that a session observes a prefix of all updates from other sessions plus all of its own updates.

Definition 9. A prefix consistent schedule \((vi, ar)\) of a history \((E, so, T)\) and a synchronization relation \(sso \subseteq so\) is a causally consistent schedule for which \(ar; ((vi \setminus so) \cup sso) \subseteq vi\).

In TouchDevelop, transactions with atomic arbitration and visibility are implicitly defined to be all code segments that are continuously executed without blocking operations. That is, each event handler is a transaction which may be split into several transactions by blocking operations. To allow synchronization (similar to its memory-model counterpart TSO), the programmer can insert fences, which force all updates arbitrated before the sessions last update to become visible. We capture fences in the above definition by \(sso\) - if a fence is executed between \((t_1, t_2) \in so\) then we have \((t_1, t_2) \in sso\). To capture the model of TouchDevelop precisely in the SMT-check, it is sufficient to include the axiom from Definition 9 in the SMT formula. We can also make the cycle-based check slightly more precise for the case \(k > 2\), we omit the details here.

TouchDevelop includes two instructions to control the consistency of the cloud-based data store: A \(yield\) acts a transaction-boundary, pulling new changes from the remote data store and pushing own updates to the data store, if it is reachable. A \(f\)lush also represents a transaction-boundary, however it blocks until the most recent changes has been pulled from the server and update have been pushed to the server. In that, \(f\)lush acts similar to a memory fence.

Adaptations in the back end. Since TouchDevelop uses prefix consistency, we slightly adapt our back end to avoid producing false alarms that are possible under causal consistency but impossible under prefix consistency. To capture prefix consistency precisely in the SMT-check, it is sufficient to include the axiom from Definition 9 in the SMT formula. For this, we extend the definition of an abstract history to include a static under-approximation \(sso\) of the synchronization relation \(sso\), which will be constructed below.

Adaptations to the front end. Our front end static analysis is based on the static analyzer described by Brutschy et al. [14]. As in that work, we employ a whole-program flow- and context-sensitive interprocedural analysis, which can use various abstract domains to infer properties of TouchDevelop programs and uses the event- and environment-model described in [14], and combines an efficient heap abstraction with domains for string constants and numerical values.

Cloud types are represented like other heap-allocated objects; however, we lose all information about such objects every time a flush or yield is executed. This handling is very similar to how elements of the environment which are in \(EId\_ocasional\) in [14] are reset every time an event handler starts.

Figure 19. Overview of the analysis infrastructure of C⁴

²In its current implementation, the SMT-based check and the termination check is only implemented for the special case of \(k = 2\), which was sufficient for our benchmarks, as we always terminated after the first iteration.
To gain precision when manipulating heap-allocated objects compared to [14], we adopt a recency-abstraction of the heap [9]. We combine the recency-abstraction with an abstract domain to track equalities between variables which are not numbers (which are handled by the numerical analysis), in particular between strings and certain immutable heap types which can be stored in cloud types.

**Inferring transactions, control-flow, and invariants.** To make use of the existing static analysis infrastructure as much as possible, we build the entire analysis using auxiliary variables, relations among which can be tracked by existing domains. To this end, we rewrite a program which interacts with the data store into a program that does not interact with it, by replacing all such interactions with updates to auxiliary variables, as shown in Figure 20. Abstract events are identified by their label \( l \). aux_lastEvent stores the last event executed, so that determining the possible values of aux_lastEvent before an operation will yield the possible predecessors in the abstract event order \( eo \). aux_arg_*_* store the values of argument and return value of the database operation \( op \) (note that operations on cloud types only have immutable types as parameters).

Since we do not take data store state into account, return value of the operation itself is replaced by the nondeterministic statement \( \text{nondet} \). Abstract transactions are identified by the label of the \text{yield} or \text{flush} operation right before it, and control-flow between the transactions is tracked using aux_lastTxn, while aux_curTxn can be used to determine which transactions an abstract event may be part of.

We implement two auxiliary operations, paths and keys, which we give abstract transformers but which do not have direct corresponding operations in the concrete TouchDevelop semantics. They assume a heap-graph based abstraction and a collection abstraction like the one described in Brutschy et al. [14]. \text{paths}(obj) determines the path in the heap graph that leads from the singleton from which all replicated data is reachable (the singleton \text{cloud}) to obj. Since elements in the replicated data structures cannot alias each other, only one such path exists in concrete executions, however, we may have to consider multiple such paths in the abstract. \text{keys}(p, obj) determines all abstract identifiers of keys \( \{ l, key \} \) along the path \( p \) which are required to reach an object. They are recorded as arguments of the abstract event.

**Building abstract histories.** Let \( \hat{\sigma} \) be the result of flow-sensitive static analysis of a rewritten program, where \( \hat{\sigma}(l, v) \) gives the set of possible values of variable \( v \) at label \( l \) and \( \hat{\sigma}(l, V) \) gives the strongest inferred invariant that must hold over the set of variables \( V \) at position \( l \). Furthermore, let

\[
\begin{align*}
\text{aux_lastEvent} & := \text{nondet} \\
\text{aux_arg}_l & := e_k \\
\text{aux_arg}_{l+1} & := e_k + 1 \\
\text{aux_arg}_l & := e_n \\
\text{aux_arg}_{l-1} & := \text{nondet} \\
\text{aux_arg}_l & := \text{nondet}; \\
\text{aux_arg}_{l+1} & := \text{nondet}; \\
\text{aux_arg}_{l+2} & := \text{nondet}; \\
\text{aux_lastTxn} & := \text{nondet}; \\
\text{aux_curTxn} & := \text{nondet}.
\end{align*}
\]

\( L_{op} \) be the labels of database operations, \( L_{yield} \) be the labels of yield operations and \( L_{flush} \) the labels of flush operations. We define \( \text{Var}_L \) and \( \text{Var}_G \) to be all global variables in the program that are always constant for one session and the same for all sessions, resp. The set of abstract events \( \mathcal{E}_v \) is defined to be \( L_{op} \), the set of abstract transactions is given indirectly by the codomain of the function \( T(t) = \{ l \in \mathcal{E}_v \mid t \in \hat{\sigma}(l, \text{aux_curTxn}) \} \) which is defined for each label \( t \in (L_{yield} \cup L_{flush}) \). Then we define the abstract history \( \{ \mathcal{E}_v, eo, Tx, so, sso, \text{Inv}, \text{Var}_L, \text{Var}_G \} \) in the following way:

\[
\begin{align*}
eo & = \{(l', l) \mid l \in L_{op}, l' \in \hat{\sigma}(l, \text{aux_lastEvent})\} \\
Tx & = \{(t) \mid t \in L_{yield} \cup L_{flush}\} \\
so & = \{(T(t'), T(t)) \mid t \in Tx, t' \in \hat{\sigma}(t, \text{aux_lastTxn})\} \\
sso & = \{(T, T(t)) \mid (T, T(t)) \in so, (true) = \hat{\sigma}(t, \text{aux_flush})\} \\
\text{Inv} & = \lambda(l_1, l_2).\hat{\sigma}(l_2, \{ \text{aux_arg}_{l-n} \mid n \in \mathbb{N}, l \in \{l_1, l_2\} \} \cup \text{Var}_L \cup \text{Var}_G)
\end{align*}
\]

For simplicity, we omitted the handling of entries and exits of transactions here, assuming that every transaction has a single entry and exit operation.

**H.2 Front end for 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 [39]. 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, 33, 34]. 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 that causal consistency and atomic visibility are not violated). The details of the approach are described by Kurath [31].

I Detected bugs

In this appendix, we discuss the nature of the violations that turned out to be clearly harmful violations in our manual inspection.

1.1 TouchDevelop: Comparison to dynamic analysis

As expected due to the soundness of the approach, C^4 found a superset of all violations detected by our dynamic analysis. In particular, our approach correctly identifies all 4 bugs in Section 6.5 of that work (which we do not describe here). Moreover, our approach found 4 new bugs that were previously missed by the dynamic analysis. Relatd contains code that tries to enforce uniqueness of usernames but fails to do so due to the weak transactional semantics of TouchDevelop. This behavior is unlikely to be triggered in a dynamic analysis since it requires two sessions to register with the same name at the same time. A similar violation was missed by the dynamic analysis in Cloud Card, which allows two text notes created on different devices to overwrite each other if the titles are the same, and both devices may temporarily see their text note to be in the data store. The new bugs found in Color Line (which is the same as the discovered one in tetriss) and Save Passwords would require similarly unlikely schedules to be discovered by a dynamic analysis.

1.2 Cassandra benchmarks

Since most serious Cassandra-based development is closed-source, many of the benchmarks we analyze here are simple, and one would not expect many critical bugs. However, we found clearly harmful violations in 4 out of 10 applications. Details about the detected bugs are described by Kurath [31]. In general, the alarms discovered by our analysis typically belong to one of four categories:

Category 1: Uniqueness. The most common bug across all benchmarks is the attempt to establish uniqueness of user-chosen values (such as a username, an email address, a chat room name, etc.). Both cassatwitter and cassandra-twitter, as well as the Relatd app, contain this bug. An example violation is shown in Figure 21a. Both systems provide synchronization primitives to solve this problem: CASSANDRA provides a Paxos-based IF NOT EXISTS synchronization primitive, while TouchDevelop apps can solve the problem using an unsynchronized set_if_empty operation, combined with a flush to check if the write succeeded.

Category 2: Non-atomic updates. The second category is the non-atomic update of existing values in the database. For example, cassieq reads, modifies, and writes the set of keys that are stored for each account, instead of using Cassandra’s atomic set-addition operator.

Category 3: Disappearing and reappearing data. The third category of bugs is that data is unexpectedly lost when two updates overwrite each other. For example, in dstax-queueing, when two concurrent writers insert jobs into the queue, one job may overwrite the other, violating the typical contract of a queue (see Figure 21b). Similarly, deleted data may reappear when an update is performed on a record which is concurrently deleted. For example, cassieq may recreate half-empty accounts when an account is deleted concurrently with an update, a bug that can be fixed by using the Paxos-based IF EXISTS synchronization primitive (see Figure 21c). Similar violations we categorized as harmless: in the IRC logger roomstore, messages may be lost if they are sent at the exact same time. In a more exotic case in playlist, music tracks may be lost if they are inserted in the exact same millisecond.

Category 4: Garbage data. The fourth category is the creation of garbage data, which we marked as harmless in the evaluation above, but we still think can be useful to report to the developer. The situation typically occurs when an entity and associated data is deleted from the database, while on a different replica associated data is added to the entity. For example, in the playlist benchmark, a playlist can be deleted, while concurrently a track is added to the same playlist. The track entry remains as a curious artifact in the database, referencing a non-existing playlist identifier, and potentially reappearing when a playlist with the same identifier is created. Similarly, killrchat allows messages to be sent to just-deleted chat-rooms, creating garbage messages in the database. Garbage data may often be permitted in replicated systems since proper synchronization is expensive and the data by itself can be harmless. However, garbage data also typically defies the developer’s intuition of what should even be stored in the database; therefore, it may cause bugs. We see reporting such violations as useful, if not of direct bug-finding utility, at least as a help to document possible anomalies of the system.
Figure 21. Serializability violations uncovering bugs