Checking Robustness Against Snapshot Isolation*

Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea

Université de Paris, IRIF, CNRS, Paris, France, {beillahi,abou,cenea}@irif.fr

Abstract. Transactional access to databases is an important abstraction allowing programmers to consider blocks of actions (transactions) as executing in isolation. The strongest consistency model is *serializability*, which ensures the atomicity abstraction of transactions executing over a sequentially consistent memory. Since ensuring serializability carries a significant penalty on availability, modern databases provide weaker consistency models, one of the most prominent being *snapshot isolation*. In general, the correctness of a program relying on serializable transactions may be broken when using weaker models. However, certain programs may also be insensitive to consistency relaxations, i.e., all their properties holding under serializability are preserved even when they are executed over a weak consistent database and without additional synchronization. In this paper, we address the issue of verifying if a given program is *robust against snapshot isolation*, i.e., all its behaviors are serializable even if it is executed over a database ensuring snapshot isolation. We show that this verification problem is polynomial time reducible to a state reachability problem in transactional programs over a sequentially consistent shared memory. This reduction opens the door to the reuse of the classic verification technology for reasoning about weakly-consistent programs. In particular, we show that it can be used to derive a proof technique based on Lipton's reduction theory that allows to prove programs robust.

1 Introduction

Transactions simplify concurrent programming by enabling computations on shared data that are isolated from other concurrent computations and resilient to failures. Modern databases provide transactions in various forms corresponding to different tradeoffs between consistency and availability. The strongest consistency level is achieved with *serializable* transactions [20] whose outcome in concurrent executions is the same as if the transactions were executed atomically in some order. Since serializability carries a significant penalty on availability, modern databases often provide weaker consistency models, one of the most prominent being *snapshot isolation* (SI) [4]. Then, an important issue is to ensure that the level of consistency needed by a given program coincides with the one that is guaranteed by its infrastructure, i.e., the database it uses. One

* This work is supported in part by the European Research Council (ERC) under the Horizon 2020 research and innovation programme (grant agreement No 678177).

way to tackle this issue is to investigate the problem of checking *robustness* of programs against consistency relaxations: Given a program P and two consistency models S and W such that S is stronger than W, we say that P is robust for S against W if for every two implementations I_S and I_W of S and W respectively, the set of computations of P when running with I_S is the same as its set of computations when running with I_W. This means that P is not sensitive to the consistency relaxation from S to W, and therefore it is possible to reason about the behaviors of P assuming that it is running over S, and no additional synchronization is required when P runs over the weak model W such that it maintains all its properties satisfied with S.

In this paper, we address the problem of verifying robustness of transactional programs for serializability, against *snapshot isolation*. Under snapshot isolation, any transaction t reads values from a snapshot of the database taken at its start and t can commit only if no other committed transaction has written to a location that t wrote to, since t started. Robustness is a form of program equivalence between two versions of the same program, obtained using two semantics, one more permissive than the other. It ensures that this permissiveness has no effect on the program under consideration. The difficulty in checking robustness is to apprehend the extra behaviors due to the relaxed model w.r.t. the strong model. This requires a priori reasoning about complex order constraints between operations in arbitrarily long computations, which may need maintaining unbounded ordered structures, and make robustness checking hard or even undecidable.

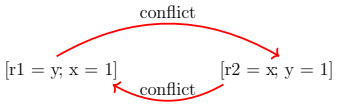
Our first contribution is to show that verifying robustness of transactional programs against snapshot isolation can be reduced in polynomial time to the reachability problem in concurrent programs under sequential consistency (SC). This allows (1) to avoid explicit handling of the snapshots from where transactions read along computations (since this may imply memorizing unbounded information), and (2) to leverage available tools for verifying invariants/reachability problems on concurrent programs. This also implies that the robustness problem is decidable for finite-state programs, PSPACE-complete when the number of sites is fixed, and EXPSPACE-complete otherwise. This is the first result on the decidability and complexity of the problem of verifying robustness in the context of transactional programs. The problem of verifying robustness has been considered in the literature for several models, including eventual and causal consistency [5, 9, 10, 11, 19]. These works provide (over- or under-)approximate analyses for checking robustness, but none of them provides precise (sound and complete) algorithmic verification methods for solving this problem.

Based on this reduction, our second contribution is a proof methodology for establishing robustness which builds on Lipton's reduction theory [17]. We use the theory of movers to establish whether the relaxations allowed by SI are harmless, i.e., they don't introduce new behaviors compared to serializability.

We tested the applicability of the proposed verification techniques on a benchmark suite containing 10 challenging applications extracted from previous work [2, 5, 10, 13, 15, 18, 23]. These techniques were precise enough for proving or disproving the robustness of all of these applications.

```
    p1:                  p2:
t1: [r1 = y //0 ||  t2: [r2 = x //0
     x = 1]               y = 1]
```

(a) Write Skew (WS).



(b) A WS execution trace.

Fig. 1: Examples of non-robust programs illustrating the difference between SI and serializability. *causal dependency* means that a read in a transaction obtains its value from a write in another transaction. *conflict* means that a write in a transaction is not visible to a read in another transaction, but it would affect the read value if it were visible. Here, *happens-before* is the union of the two.

2 Overview

In this section, we give an overview of our approach for checking robustness against snapshot isolation. While serializability enforces that transactions are atomic and conflicting transactions, i.e., which read or write to a common location, *cannot* commit concurrently, SI [4] allows that conflicting transactions commit in parallel as long as they don't contain a write-write conflict, i.e., write on a common location. Moreover, under SI, each transaction reads from a snapshot of the database taken at its start. These relaxations permit the "anomaly" known as Write Skew (WS) shown in Figure 1a, where an anomaly is a program execution which is allowed by SI, but not by serializability. The execution of Write Skew under SI allows the reads of x and y to return 0 although this cannot happen under serializability. These values are possible since each transaction is executed locally (starting from the initial snapshot) without observing the writes of the other transaction.

Execution trace. Our notion of program robustness is based on an abstract representation of executions called *trace*. Informally, an execution trace is a set of events, i.e., accesses to shared variables and transaction begin/commit events, along with several standard dependency relations between events recording the data-flow. The transitive closure of the union of all these dependency relations is called *happens-before*. An execution is an anomaly if the happens-before of its trace is cyclic. Figure 1b shows the happens-before of the Write Skew anomaly. Notice that the happens-before order is cyclic in both cases.

Semantically, every transaction execution involves two main events, the issue and the commit. The issue event corresponds to a sequence of reads and/or writes where the writes are visible only to the current transaction. We interpret it as a single event since a transaction starts with a database snapshot that it updates in isolation, without observing other concurrently executing transactions. The commit event is where the writes are propagated and made visible to all processes. Under serializability, the two events coincide, i.e., they are adjacent in the execution. Under SI, this is not the case and in between the issue and the commit of the same transaction, we may have issue/commit events from concurrent transactions. When a transaction commit does not occur immediately after its issue, we say that the underlying transaction is *delayed*. For example, the

following execution of WS corresponds to the happens-before cycle in Figure 1b where the write to x was committed after t_2 finished, hence, t_1 was delayed:

$$\mathsf{begin}(p_1,t_1)\mathsf{ld}(p_1,t_1,y,0)\mathsf{isu}(p_1,t_1,x,1) \qquad \mathsf{com}(p_1,t_1)$$
$$\mathsf{begin}(p_2,t_2)\mathsf{ld}(p_2,t_2,x,0)\mathsf{isu}(p_2,t_2,y,1)\mathsf{com}(p_2,t_2)$$

Above, $\mathsf{begin}(p_1,t_1)$ stands for starting a new transaction t_1 by process p_1, ld represents read (load) actions, while isu denotes write actions that are visible only to the current transaction (not yet committed). The writes performed during t_1 become visible to all processes once the commit event $\mathsf{com}(p_1,t_1)$ takes place.

Reducing robustness to SC reachability. The above SI execution can be mimicked by an execution of the same program under serializability modulo an instrumentation that simulates the delayed transaction. The local writes in the issue event are simulated by writes to auxiliary registers and the commit event is replaced by copying the values from the auxiliary registers to the shared variables (actually, it is not necessary to simulate the commit event; we include it here for presentation reasons). The auxiliary registers are visible only to the delayed transaction. In order that the execution be an anomaly (i.e., not possible under serializability without the instrumentation) it is required that the issue and the commit events of the delayed transaction are linked by a chain of happens-before dependencies. For instance, the above execution for WS can be simulated by:

$$\mathsf{begin}(p_1,t_1)\mathsf{ld}(p_1,t_1,y,0)\mathsf{st}(p_1,t_1,r_x,1) \qquad \mathsf{st}(p_1,t_1,x,r_x)$$
$$\mathsf{begin}(p_2,t_2)\mathsf{ld}(p_2,t_2,x,0)\mathsf{isu}(p_2,t_2,y,1)\mathsf{com}(p_2,t_2)$$

The write to x was delayed by storing the value in the auxiliary register r_x and the happens-before chain exists because the read on y that was done by t_1 is conflicting with the write on y from t_2 and the read on x by t_2 is conflicting with the write of x in the simulation of t_1's commit event. On the other hand, the following execution of Write-Skew without the read on y in t_1:

$$\mathsf{begin}(p_1,t_1)\mathsf{st}(p_1,t_1,r_x,1) \qquad \mathsf{st}(p_1,t_1,x,r_x)$$
$$\mathsf{begin}(p_2,t_2)\mathsf{ld}(p_2,t_2,x,0)\mathsf{isu}(p_2,t_2,y,1)\mathsf{com}(p_2,t_2)$$

misses the conflict (happens-before dependency) between the issue event of t_1 and t_2. Therefore, the events of t_2 can be reordered to the left of t_1 and obtain an equivalent execution where $\mathsf{st}(p_1,t_1,x,r_x)$ occurs immediately after $\mathsf{st}(p_1,t_1,r_x,1)$. In this case, t_1 is not anymore delayed and this execution is possible under serializability (without the instrumentation).

If the number of transactions to be delayed in order to expose an anomaly is unbounded, the instrumentation described above may need an unbounded number of auxiliary registers. This would make the verification problem hard or even undecidable. However, we show that it is actually enough to delay a single transaction, i.e., a program admits an anomaly under SI iff it admits an anomaly containing a single delayed transaction. This result implies that the number of auxiliary registers needed by the instrumentation is bounded by the number of program variables, and that checking robustness against SI can be reduced in linear time to a reachability problem under serializability (the reachability problem encodes the existence of the chain of happens-before

dependencies mentioned above). The proof of this reduction relies on a non-trivial characterization of anomalies.

Proving robustness using commutativity dependency graphs. Based on the reduction above, we also devise an approximated method for checking robustness based on the concept of mover in Lipton's reduction theory [17]. An event is a left (resp., right) mover if it commutes to the left (resp., right) of another event (from a different process) while preserving the computation. We use the notion of mover to characterize happens-before dependencies between transactions. Roughly, there exists a happens-before dependency between two transactions in some execution if one doesn't commute to the left/right of the other one. We define a commutativity dependency graph which summarizes the happens-before dependencies in all executions of a given program between transactions t as they appear in the program, transactions $t \setminus \{w\}$ where the writes of t are deactivated (i.e., their effects are not visible outside the transaction), and transactions $t \setminus \{r\}$ where the reads of t obtain non-deterministic values. The transactions $t \setminus \{w\}$ are used to simulate issue events of delayed transactions (where writes are not yet visible) while the transactions $t \setminus \{r\}$ are used to simulate commit events of delayed transactions (which only write to the shared memory). Two transactions a and b are linked by an edge iff a *cannot* move to the right of b (or b cannot move to the left of a), or if they are related by the program order (i.e., issued in some order in the same process). Then a program is robust if for every transaction t, this graph *doesn't* contain a path from $t \setminus \{w\}$ to $t \setminus \{r\}$ formed of transactions that don't write to a variable that t writes to (the latter condition is enforced by `SI` since two concurrent transactions cannot commit at the same time when they write to a common variable). For example, Figure 2 shows the commutativity dependency graph of the modified WS program where the read of y is removed from t_1. The fact that it doesn't contain any path like above implies that it is robust.

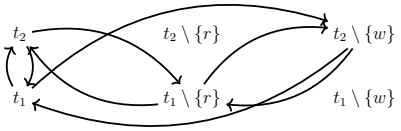


Fig. 2: Commutativity dependency graph of WS where the read of y is omitted.

3 Programs

A program is parallel composition of *processes* distinguished using a set of identifiers $\mathbb{P}$. Each process is a sequence of *transactions* and each transaction is a sequence of *labeled instructions*. Each transaction starts with a `begin` instruction and finishes with a `commit` instruction. Each other instruction is either an assignment to a process-local *register* from a set $\mathbb{R}$ or to a *shared variable* from a set $\mathbb{V}$, or an `assume` statement. The read/write assignments use values from a data domain $\mathbb{D}$. An assignment to a register $\langle reg \rangle := \langle var \rangle$ is called a *read* of the shared-variable $\langle var \rangle$ and an assignment to a shared variable $\langle var \rangle := \langle reg\text{-}expr \rangle$ is called a *write* to $\langle var \rangle$ ($\langle reg\text{-}expr \rangle$ is an expression over registers whose syntax we leave unspecified since it is irrelevant for our development). The `assume`

$\langle bexpr \rangle$ blocks the process if the Boolean expression $\langle bexpr \rangle$ over registers is false. They are used to model conditionals as usual. We use `goto` statements to model an arbitrary control-flow where the same label can be assigned to multiple instructions and multiple `goto` statements can direct the control to the same label which allows to mimic imperative constructs like loops and conditionals. To simplify the technical exposition, our syntax includes simple read/write instructions. However, our results apply as well to instructions that include SQL (select/update) queries. The experiments reported in Section 7 consider programs with SQL based transactions.

The semantics of a program under `SI` is defined as follows. The shared variables are stored in a central memory and each process keeps a replicated copy of the central memory. A process starts a transaction by discarding its local copy and fetching the values of the shared variables from the central memory. When a process commits a transaction, it merges its local copy of the shared variables with the one stored in the central memory in order to make its updates visible to all processes. During the execution of a transaction, the process stores the writes to shared variables only in its local copy and reads only from its local copy. When a process merges its local copy with the centralized one, it is required that there were no concurrent updates that occurred after the last fetch from the central memory to a shared variable that was updated by the current transaction. Otherwise, the transaction is aborted and its effects discarded.

More precisely, the semantics of a program $\mathcal{P}$ under `SI` is defined as a labeled transition system $[\mathcal{P}]_{\texttt{SI}}$ where transactions are labeled by the set of events

$$\mathbb{Ev} = \{\mathsf{begin}(p,t), \mathsf{ld}(p,t,x,v), \mathsf{isu}(p,t,x,v), \mathsf{com}(p,t) : p \in \mathbb{P}, t \in \mathbb{T}^2, x \in \mathbb{V}, v \in \mathbb{D}\}$$

where begin and com label transitions corresponding to the start and the commit of a transaction, respectively. isu and ld label transitions corresponding to writing, resp., reading, a shared variable during some transaction. The precise definition of this semantics is given in Appendix **??**.

An execution of program $\mathcal{P}$, under snapshot isolation, is a sequence of events $ev_1 \cdot ev_2 \cdot \ldots$ corresponding to a run of $[\mathcal{P}]_{\texttt{CM}}$. The set of executions of $\mathcal{P}$ under `SI` is denoted by $\mathbb{Ex}_{\texttt{SI}}(\mathcal{P})$.

4 Robustness Against `SI`

A *trace* abstracts the order in which shared-variables are accessed inside a transaction and the order between transactions accessing different variables. Formally, the trace of an execution ρ is obtained by (1) replacing each sub-sequence of transitions in ρ corresponding to the same transaction, but excluding the com transition, with a single "macro-event" $\mathsf{isu}(p,t)$, and (2) adding several standard relations between these macro-events $\mathsf{isu}(p,t)$ and commit events $\mathsf{com}(p,t)$ to record the data-flow in ρ, e.g. which transaction wrote the value read by another transaction. The sequence of $\mathsf{isu}(p,t)$ and $\mathsf{com}(p,t)$ events obtained in the first step is called a *summary of* ρ. We say that a transaction t in ρ performs an *external read* of a variable x if ρ contains an event $\mathsf{ld}(p,t,x,v)$ which is not

preceded by a write on x of t, i.e., an event $\mathsf{isu}(p, t, x, v)$. Also, we say that a transaction t *writes* a variable x if ρ contains an event $\mathsf{isu}(p, t, x, v)$, for some v.

The *trace* $\mathsf{tr}(\rho) = (\tau, \mathsf{PO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}, \mathsf{STO})$ of an execution ρ consists of the summary τ of ρ along with the *program order* PO, which relates any two issue events $\mathsf{isu}(p, t)$ and $\mathsf{isu}(p, t')$ that occur in this order in τ, *write-read* relation WR (also called *read-from*), which relates any two events $\mathsf{com}(p, t)$ and $\mathsf{isu}(p', t')$ that occur in this order in τ such that t' performs an external read of x, and $\mathsf{com}(p, t)$ is the last event in τ before $\mathsf{isu}(p', t')$ that writes to x (to mark the variable x, we may use $\mathsf{WR}(x)$), the *write-write* order WW (also called store-order), which relates any two store events $\mathsf{com}(p, t)$ and $\mathsf{com}(p', t')$ that occur in this order in τ and write to the same variable x (to mark the variable x, we may use $\mathsf{WW}(x)$), the *read-write* relation RW (also called *conflict*), which relates any two events $\mathsf{isu}(p, t)$ and $\mathsf{com}(p', t')$ that occur in this order in τ such that t reads a value that is overwritten by t', and the *same-transaction* relation STO, which relates the issue event with the commit event of the same transaction. The read-write relation RW is formally defined as $\mathsf{RW}(x) = \mathsf{WR}^{-1}(x); \mathsf{WW}(x)$ (we use ; to denote the standard composition of relations) and $\mathsf{RW} = \bigcup_{x \in \mathbb{V}} \mathsf{RW}(x)$. If a transaction t reads the initial value of x then $\mathsf{RW}(x)$ relates $\mathsf{isu}(p, t)$ to $\mathsf{com}(p', t')$ of any other transaction t' which writes to x (i.e., $(\mathsf{isu}(p, t), \mathsf{com}(p', t')) \in \mathsf{RW}(x)$) (note that in the above relations, p and p' might designate the same process).

Since we reason about only one trace at a time, to simplify the writing, we may say that a trace is simply a sequence τ as above, keeping the relations PO, WR, WW, RW, and STO implicit. The set of traces of executions of a program $\mathcal{P}$ under $\mathtt{SI}$ is denoted by $\mathbb{T}\mathsf{r}(\mathcal{P})_{\mathtt{SI}}$.

Serializability semantics. The semantics of a program under serializability can be defined using a transition system where the configurations keep a single shared-variable valuation (accessed by all processes) with the standard interpretation of read and write statements. Each transaction executes in isolation. Alternatively, the serializability semantics can be defined as a restriction of $[\mathcal{P}]_{\mathtt{SI}}$ to the set of executions where each transaction is *immediately* delivered when it starts, i.e., the start and commit time of transaction coincide $t.st = t.ct$. Such executions are called *serializable* and the set of serializable executions of a program $\mathcal{P}$ is denoted by $\mathbb{E}\mathsf{x}_{\mathtt{SER}}(\mathcal{P})$. The latter definition is easier to reason about when relating executions under snapshot isolation and serializability, respectively.

Serializable trace. A trace tr is called *serializable* if it is the trace of a serializable execution. Let $\mathbb{T}\mathsf{r}_{\mathtt{SER}}(\mathcal{P})$ denote the set of serializable traces. Given a serializable trace $tr = (\tau, \mathsf{PO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}, \mathsf{STO})$ we have that every event $\mathsf{isu}(p, t)$ in τ is immediately followed by the corresponding $\mathsf{com}(p, t)$ event.

Happens before order. Since multiple executions may have the same trace, it is possible that an execution ρ produced by snapshot isolation has a serializable trace $\mathsf{tr}(\rho)$ even though $\mathsf{isu}(p, t)$ events may not be immediately followed by $\mathsf{com}(p, t)$ actions. However, ρ would be equivalent, up to reordering of "independent" (or commutative) transitions, to a serializable execution. To check whether the trace of an execution is serializable, we introduce the *happens-before*

relation on the events of a given trace as the transitive closure of the union of all the relations in the trace, i.e., $\mathsf{HB} = (\mathsf{PO} \cup \mathsf{WW} \cup \mathsf{WR} \cup \mathsf{RW} \cup \mathsf{STO})^+$.

Finally, the happens-before relation between events is extended to transactions as follows: a transaction t_1 *happens-before* another transaction $t_2 \neq t_1$ if the trace tr contains an event of transaction t_1 which happens-before an event of t_2. The happens-before relation between transactions is denoted by HB_t and called *transactional happens-before*. The following characterizes serializable traces.

Theorem 1 ([1, 22]). *A trace tr is serializable iff HB_t is acyclic.*

A program is called robust if it produces the same set of traces as the serializability semantics.

Definition 1. *A program $\mathcal{P}$ is called* robust *against* `SI` *iff* $\mathbb{T}\mathsf{r}_{\mathtt{SI}}(\mathcal{P}) = \mathbb{T}\mathsf{r}_{\mathtt{SER}}(\mathcal{P})$.

Since $\mathbb{T}\mathsf{r}_{\mathtt{SER}}(\mathcal{P}) \subseteq \mathbb{T}\mathsf{r}_{\mathsf{X}}(\mathcal{P})$, the problem of checking robustness of a program $\mathcal{P}$ is reduced to checking whether there exists a trace $tr \in \mathbb{T}\mathsf{r}_{\mathtt{SI}}(\mathcal{P}) \setminus \mathbb{T}\mathsf{r}_{\mathtt{SER}}(\mathcal{P})$.

5 Reducing Robustness against `SI` to SC Reachability

A trace which is not serializable must contain at least an issue and a commit event of the same transaction that don't occur one after the other even after reordering of "independent" events. Thus, there must exist an event that occur between the two which is related to both events via the happens-before relation, forbidding the issue and commit to be adjacent. Otherwise, we can build another trace with the same happens-before where events are reordered such that the issue is immediately followed by the corresponding commit. The latter is a serializable trace which contradicts the initial assumption. We define a program instrumentation which mimics the delay of transactions by doing the writes on auxiliary variables which are not visible to other transactions. After the delay of a transaction, we track happens-before dependencies until we execute a transaction that does a "read" on one of the variables that the delayed transaction writes to (this would expose a read-write dependency to the commit event of the delayed transaction). While tracking happens-before dependencies we cannot execute a transaction that writes to a variable that the delayed transaction writes to since `SI` forbids write-write conflicts between concurrent transactions.

Concretely, given a program $\mathcal{P}$, we define an instrumentation of $\mathcal{P}$ such that $\mathcal{P}$ is not robust against `SI` iff the instrumentation reaches an error state under serializability. The instrumentation uses auxiliary variables in order to simulate a *single* delayed transaction which we prove that it is enough for deciding robustness. Let $\mathsf{isu}(p, t)$ be the issue event of the only delayed transaction. The process p that delayed t is called the *Attacker*. When the attacker finishes executing the delayed transaction it stops. Other processes that execute transactions afterwards are called *Happens-Before Helpers*.

The instrumentation uses two copies of the set of shared variables in the original program to simulate the delayed transaction. We use primed variables

x' to denote the second copy. Thus, when a process becomes the attacker, it will only write to the second copy that is not visible to other processes including the happens-before helpers. The writes made by the other processes including the happens-before helpers are made visible to all processes.

When the attacker delays the transaction t, it keeps track of the variables it accessed, in particular, it stores the name of one of the variables it writes to, x, it tracks every variable y that it reads from and every variable z that it writes to. When the attacker finishes executing t, and some other process wants to execute some other transaction, the underlying transaction must contain a write to a variable y that the attacker reads from. Also, the underlying transaction must not write to a variable that t writes to. We say that this process has joined happens-before helpers through the underlying transaction. While executing this transaction, we keep track of each variable that was accessed and the type of operation, wheather it is a read or write. Afterward, in order for some other transaction to "join" the happens-before path, it must not write to a variable that t writes to so it does not violate the fact that `SI` forbids write-write conflicts, and it has to satisfy one of the following conditions in order to ensure the continuity of the happens-before dependencies: (1) the transaction is issued by a process that has already another transaction in the happens-before dependency (program order dependency), (2) the transaction is reading from a shared variable that was updated by a previous transaction in the happens-before dependency (write-read dependency), (3) the transaction writes to a shared variable that was read by a previous transaction in the happens-before dependency (read-write dependency), or (4) the transaction writes to a shared variable that was updated by a previous transaction in the happens-before dependency (write-write dependency). We introduce a flag for each shared variable to mark the fact that the variable was read or written by a previous transaction.

Processes continue executing transactions as part of the chain of happens-before dependencies, until a transaction does a read on the variable x that t wrote to. In this case, we reached an error state which signals that we found a cycle in the transactional happens-before relation.

The instrumentation uses four varieties of flags: a) global flags (i.e., HB, $a_{\mathsf{tr_A}}$, $a_{\mathsf{st_A}}$), b) flags local to a process (i.e., $p.a$ and $p.hbh$), and c) flags per shared variable (i.e., $x.event$, $x.event'$, and $x.eventI$). We will explain the meaning of these flags along with the instrumentation. At the start of the execution, all flags are initialized to null ($\perp$).

Whether a process is an attacker or happens-before helper is not enforced syntactically by the instrumentation. It is set non-deterministically during the execution using some additional process-local flags. Each process chooses to set to true at most one of the flags $p.a$ and $p.hbh$, implying that the process becomes an attacker or happens-before helper, respectively. At most one process can be an attacker, i.e., set $p.a$ to true. In the following, we detail the instrumentation for read and write instructions of the attacker and happens-before helpers.

$[\![\mathsf{l}_1 \colon r := x;\ \mathtt{goto}\ \mathsf{l}_2;]\!]_{\mathsf{A}} =$
// Read before the delayed transaction
$\mathsf{l}_1 \colon \mathtt{assume}\ a_{\mathsf{tr}_{\mathsf{A}}} = \bot\ ;\ \mathtt{goto}\ \mathsf{l}_{x1};$
$\mathsf{l}_{x1} \colon r := x;\ \mathtt{goto}\ \mathsf{l}_2;$
// Read in the delayed transaction
$\mathsf{l}_1 \colon \mathtt{assume}\ a_{\mathsf{tr}_{\mathsf{A}}} \neq \bot \wedge p.a \neq \bot\ ;\ \mathtt{goto}\ \mathsf{l}_{x2};$
$\mathsf{l}_{x2} \colon r := x';\ \mathtt{goto}\ \mathsf{l}_{x3};$
$\mathsf{l}_{x3} \colon x.event := \mathsf{ld};\ \mathtt{goto}\ \mathsf{l}_{x4};$ (1)
$\mathsf{l}_{x4} \colon \mathtt{assume}\ \mathsf{HB} = \bot\ ;\ \mathtt{goto}\ \mathsf{l}_{x5};$
$\mathsf{l}_{x5} \colon \mathsf{HB} := \mathsf{true};\ \mathtt{goto}\ \mathsf{l}_2;$ (2)
$\mathsf{l}_{x4} \colon \mathtt{assume}\ \mathsf{HB} \neq \bot\ ;\ \mathtt{goto}\ \mathsf{l}_2;$

$[\![\mathsf{l}_1 \colon x := e;\ \mathtt{goto}\ \mathsf{l}_2;]\!]_{\mathsf{A}} =$
// Write before the delayed transaction
$\mathsf{l}_1 \colon \mathtt{assume}\ a_{\mathsf{tr}_{\mathsf{A}}} = \bot\ ;\ \mathtt{goto}\ \mathsf{l}_{x1};$
$\mathsf{l}_{x1} \colon x := e;\ \mathtt{goto}\ \mathsf{l}_2;$
// Write in the delayed transaction
$\mathsf{l}_1 \colon \mathtt{assume}\ a_{\mathsf{tr}_{\mathsf{A}}} \neq \bot \wedge p.a \neq \bot\ ;\ \mathtt{goto}\ \mathsf{l}_{x2};$
$\mathsf{l}_{x2} \colon x' := e;\ \mathtt{goto}\ \mathsf{l}_{x3};$ (3)
$\mathsf{l}_{x3} \colon x.event' := 1;\ \mathtt{goto}\ \mathsf{l}_2;$ (4)
// Special write in the delayed transaction
$\mathsf{l}_1 \colon \mathtt{assume}\ a_{\mathsf{st}_{\mathsf{A}}} = \bot \wedge a_{\mathsf{tr}_{\mathsf{A}}} \neq \bot \wedge x.event = \bot\ ;\ \mathtt{goto}\ \mathsf{l}_{x4};$
$\mathsf{l}_{x4} \colon x' := e;\ \mathtt{goto}\ \mathsf{l}_{x5};$
$\mathsf{l}_{x5} \colon a_{\mathsf{st}_{\mathsf{A}}} := \text{`}x\text{`};\ \mathtt{goto}\ \mathsf{l}_{x6};$ (5)
$\mathsf{l}_{x8} \colon x.event' := 1;\ \mathtt{goto}\ \mathsf{l}_2;$

Fig. 3: Instrumentation of the Attacker. We use ‘x‘ to denote the name of the shared variable x.

5.1 Instrumentation of the Attacker

Figure 3 lists the instrumentation of the write and read instructions of the attacker. Each process passes through an initial phase where it executes transactions that are visible immediately to all the other processes (i.e., they are not delayed), and then non-deterministically it can choose to delay a transaction at which point it sets the flag $a_{\mathsf{tr}_{\mathsf{A}}}$ to true. During the delayed transaction it chooses non-deterministically a write instruction to a variable x and stores the name of this variable in the flag $a_{\mathsf{st}_{\mathsf{A}}}$ (line (5)). The values written during the delayed transaction are stored in the primed variables and are visible only to the current transaction, in case the transaction reads its own writes. For example, given a variable z, all writes to z from the original program are transformed into writes to the primed version z' (line (3)). Each time, the attacker writes to z, it sets the flag $z.event' = 1$. This flag is used later by transactions from happens-before helpers to avoid writing to variables that the delayed transaction writes to.

A read on a variable, y, in the delayed transaction takes her value from the primed version, y'. In every read in the delayed transaction, we set the flag $y.event$ to ld (line (1)) to be used latter in order for a process to join the happens-before helpers. Afterward, the attacker starts the happens-before path, and it sets the variable HB to true (line (2)) to mark the start of the happens. When the flag HB is set to true the attacker stops executing new transactions.

5.2 Instrumentation of the Happens-Before Helpers

The remaining processes, which are not the attacker, can become a happens-before helper. Figure 4 lists the instrumentation of write and read instructions of a happens-before helper. In a first phase, each process executes the original code until the flag $a_{\mathsf{tr}_{\mathsf{A}}}$ is set to true by the attacker. This flag signals the “creation” of the secondary copy of the shared-variables, which can be observed only by the attacker. At this point, the flag HB is set to true, and the happens-before

helper process chooses non-deterministically a first transaction through which it wants to join the set of happens-before helpers, i.e., continue the happens-before dependency created by the existing happens-before helpers. When a process chooses a transaction, it makes a pledge (while executing the `begin` instruction) that during this transaction it will either read from a variable that was written to by another happens-before helper, write to a variable that was accessed (read or written) by another happens-before helper, or write to a variable that was read from in the delayed transaction. When the pledge is met, the process sets the flag $p.hbh$ to true (lines (7) and (11)). The execution is blocked if a process does not keep its pledge (i.e., the flag $p.hbh$ is null) at the end of the transaction. Note that the first process to join the happens-before helper has to execute a transaction t which writes to a variable that was read from in the delayed transaction since this is the only way to build a happens-before between t, and the delayed transaction (PO is not possible since t is not from the attacker, WR is not possible since t does not see the writes of the delayed transaction, and WW is not possible since t cannot write to a variable that the delayed transaction writes to). We use a flag $x.event$ for each variable x to record the type (read ld or write st) of the last access made by a happens-before helper (lines (8) and (10)). During the execution of a transaction that is part of the happens-before dependency, we must ensure that the transaction does not write to variable y where $y.even'$ is set to 1. Otherwise, the execution is blocked (line 9).

The happens-before helpers continue executing their instructions, until one of them reads from the shared variable x whose name was stored in $a_{\mathsf{st_A}}$. This establishes a happens-before dependency between the delayed transaction and a "fictitious" store event corresponding to the delayed transaction that could be executed just after this read of x. The execution doesn't have to contain this store event explicitly since it is always enabled. Therefore, at the end of every transaction, the instrumentation checks whether the transaction read x. If it is the case, then the execution stops and goes to an error state to indicate that this is a robustness violation. Notice that after the attacker stops, the only processes that are executing transactions are happens-before helpers, which is justified since when a process is not from a happens-before helper it implies that we cannot construct a happens-before dependency between a transaction of this process and the delayed transaction which means that the two transactions commute which in turn implies that this process's transactions can be executed before executing the delayed transaction of the attacker.

5.3 Correctness

The role of a process in an execution is chosen non-deterministically at runtime. Therefore, the final instrumentation of a given program $\mathcal{P}$, denoted by $[\![\mathcal{P}]\!]$, is obtained by replacing each labeled instruction $\langle linst \rangle$ with the concatenation of the instrumentations corresponding to the attacker and the happens-before helpers, i.e., $[\![\langle linst \rangle]\!] ::= [\![\langle linst \rangle]\!]_{\mathsf{A}}\ [\![\langle linst \rangle]\!]_{\mathsf{HbH}}$

The following theorem states the correctness of the instrumentation.

$[\![\mathsf{l}_1 : r := x;\ \mathsf{goto}\ \mathsf{l}_2 ;]\!]_{\mathsf{HbH}} =$

// Read before the delayed transaction

$\mathsf{l}_1 :$ assume HB $= \bot \wedge p.a = \bot$; goto l_{x1} ;

$\mathsf{l}_{x1} : r := x;$ goto l_2 ;

// Read after the delayed transaction

$\mathsf{l}_1 :$ assume HB $\neq \bot$; goto l_{x2} ;

$\mathsf{l}_{x2} : r := x;$ goto l_{x3} ;

$\mathsf{l}_{x3} :$ assume $x.eventI = \mathsf{st} \wedge p.hbh = \bot$; goto l_{x4} ; (7)

$\mathsf{l}_{x4} : p.hbh := \mathsf{true};$ goto l_2 ;

$\mathsf{l}_{x3} :$ assume $x.event = \bot$; goto l_{x5} ;

$\mathsf{l}_{x5} : x.event := \mathsf{ld};$ goto l_2 ; (8)

$\mathsf{l}_{x3} :$ assume $x.event \neq \bot \vee p.hbh \neq \bot$; goto l_2 ;

$[\![\mathsf{l}_1 : x := e;\ \mathsf{goto}\ \mathsf{l}_2 ;]\!]_{\mathsf{HbH}} =$

// Write before the delayed transaction

$\mathsf{l}_1 :$ assume HB $= \bot \wedge a_{\mathsf{tr_A}} = \bot$; goto l_{x1} ;

$\mathsf{l}_{x1} : x := e;$ goto l_2 ; (6)

// Write after the delayed transaction

$\mathsf{l}_1 :$ assume HB $\neq \bot \wedge p.a = \bot$; goto l_{x2} ;

$\mathsf{l}_{x2} :$ assume $x.event' \neq \bot$; assume false; (9)

$\mathsf{l}_{x2} :$ assume $x.event' = \bot$; goto l_{x3} ;

$\mathsf{l}_{x3} : x := e;$ goto l_{x4} ;

$\mathsf{l}_{x4} : x.event := \mathsf{st};$ goto l_{x5} ; (10)

$\mathsf{l}_{x5} :$ assume $x.eventI \neq \bot \wedge p.hbh = \bot$; goto l_{x6} ;

$\mathsf{l}_{x6} : p.hbh := \mathsf{true};$ goto l_2 ; (11)

$\mathsf{l}_{x5} :$ assume $x.eventI = \bot \vee p.hbh \neq \bot$; goto l_2 ;

Fig. 4: Instrumentation of Happens-Before Helpers.

Theorem 2. *$\mathcal{P}$ is not robust against* SI *iff $[\![\mathcal{P}]\!]$ reaches the error state.*

If a program is not robust, this implies that the execution of the program under SI results in a trace where the happens-before is cyclic. Which is possible only if the program contains at least one delayed transaction. In the proof of this theorem, we show that is sufficient to search for executions that contain a single delayed transaction. The proofs are discussed in the Appendix.

Notice that in the instrumentation of the attacker, the delayed transaction must contain a read and write instructions on different variables. Also, the transactions of the happens-before helpers must not contain a write to a variable that the delayed transaction writes to. The following corollary states the complexity of checking robustness for finite-state programs [1] against snapshot isolation. It is a direct consequence of Theorem 2 and of previous results concerning the reachability problem in concurrent programs running over a sequentially-consistent memory, with a fixed [16] or parametric number of processes [21].

Corollary 1. *Checking robustness of finite-state programs against snapshot isolation is PSPACE-complete when the number of processes is fixed and EXPSPACE-complete, otherwise.*

The instrumentation can be extended to SQL (select/update) queries where a statement may include expressions over a finite/infinite set of variables, e.g., by manipulating a set of flags x.event for each statement instead of only one.

6 Proving Program Robustness

As a more pragmatic alternative to the reduction in the previous section, we define an approximated method for proving robustness which is inspired by Lipton's reduction theory [17].

[1] Programs with a bounded number of variables taking values from a bounded domain.

Movers. Given an execution $\tau = ev_1 \cdot \ldots \cdot ev_n$ of a program $\mathcal{P}$ under serializability (where each event ev_i corresponds to executing an entire transaction), we say that the event ev_i *moves right (resp., left)* in τ if $ev_1 \cdot \ldots \cdot ev_{i-1} \cdot ev_{i+1} \cdot ev_i \cdot ev_{i+2} \cdot \ldots \cdot ev_n$ (resp., $ev_1 \cdot \ldots \cdot ev_{i-2} \cdot ev_i \cdot ev_{i-1} \cdot ev_{i+1} \cdot \ldots \cdot ev_n$) is also a valid execution of $\mathcal{P}$, the process of ev_i is different from the process of ev_{i+1} (resp., ev_{i-1}), and both executions reach to the same end state σ_n. For an execution τ, let $\mathsf{instOf}_\tau(ev_i)$ denote the transaction that generated the event ev_i. A transaction t of a program $\mathcal{P}$ is a *right (resp., left) mover* if for all executions τ of $\mathcal{P}$ under serializability, the event ev_i with $\mathsf{instOf}(ev_i) = t$ moves right (resp., left) in τ.

If a transaction t is not a right mover, then there must exist an execution τ of $\mathcal{P}$ under serializability and an event ev_i of τ with $\mathsf{instOf}(ev_i) = t$ that does not move right. This implies that there must exist another ev_{i+1} of τ which caused ev_i to not be a right mover. Since ev_i and ev_{i+1} do not commute, then this must be because of either a write-read, write-write, or a read-write dependency. If $t' = \mathsf{instOf}(ev_{i+1})$, we say that t is not a right mover because of t' and some dependency that is either write-read, write-write, or read-write. Notice that when t is not a right mover because of t' then t' is not a left mover because of t.

We define $\mathsf{M_{WR}}$ as a binary relation between transactions such that $(t, t') \in \mathsf{M_{WR}}$ when t is *not* a right mover because of t' and a write-read dependency. We define the relations $\mathsf{M_{WW}}$ and $\mathsf{M_{RW}}$ corresponding to write-write and read-write dependencies in a similar way.

Read/Write-free transactions. Given a transaction t, we define $t \setminus \{r\}$ as a variation of t where all the reads from shared variables are replaced with non-deterministic reads, i.e., $\langle reg \rangle := \langle var \rangle$ statements are replaced with $\langle reg \rangle := \star$ where $\star$ denotes non-deterministic choice. We also define $t \setminus \{w\}$ as a variation of t where all the writes to shared variables in t are disabled. Intuitively, recalling the reduction to SC reachability in Section 5, $t \setminus \{w\}$ simulates the delay of a transaction by the Attacker, i.e., the writes are not made visible to other processes, and $t \setminus \{r\}$ approximates the commit of the delayed transaction which only applies a set of writes.

Commutativity dependency graph. Given a program $\mathcal{P}$, we define the commutativity dependency graph as a graph where vertices represent transactions and their read/write-free variations. Two vertices which correspond to the original transactions in $\mathcal{P}$ are related by a program order edge, if they belong to the same process. The other edges in this graph represent the "non-mover" relations $\mathsf{M_{WR}}$, $\mathsf{M_{WW}}$, and $\mathsf{M_{RW}}$.

Given a program $\mathcal{P}$, we say that the commutativity dependency graph of $\mathcal{P}$ contains a *non-mover cycle* if there exist a set of transactions $t_0, t_1, \ldots, t_n$ of $\mathcal{P}$ such that the following hold:

(a) $(t_0'', t_1) \in \mathsf{M_{RW}}$ where t_0'' is the write-free variation of t_0 and t_1 does not write to a variable that t_0 writes to;
(b) for all $i \in [1, n]$, $(t_i, t_{i+1}) \in (\mathsf{PO} \cup \mathsf{M_{WR}} \cup \mathsf{M_{WW}} \cup \mathsf{M_{RW}})$, t_i and t_{i+1} do not write to a shared variable that t_0 writes to;
(c) $(t_n, t_0') \in \mathsf{M_{RW}}$ where t_0' is the read-free variation of t_0 and t_n does not write to a variable that t_0 writes to.

A non-mover cycle approximates an execution of the instrumentation defined in Section 5 in between the moment that the Attacker delays a transaction t_0 (which here corresponds to the write-free variation t_0'') and the moment where t_0 gets committed (the read-free variation t_0').

The following theorem shows that the acyclicity of the commutativity dependency graph of a program implies the robustness of the program. Actually, the notion of robustness in this theorem relies on a slightly different notion of trace where store-order and write-order dependencies take into account values, i.e., store-order relates only writes writing different values and the write-order relates a read to the oldest write (w.r.t. execution order) writing its value. This relaxation helps in avoiding some harmless robustness violations due to for instance, two transactions writing the same value to some variable.

Theorem 3. *For a program $\mathcal{P}$, if the commutativity dependency graph of $\mathcal{P}$ does not contain non-mover cycles, then $\mathcal{P}$ is robust.*

7 Experiments

To test the applicability of our robustness checking algorithms, we have considered a benchmark of 10 applications extracted from the literature related to weakly consistent databases in general. A first set of applications are open source projects that were implemented to be run over the Cassandra database, extracted from [10]. The second set of applications is composed of: TPC-C [23], an on-line transaction processing benchmark widely used in the database community, SmallBank, a simplified representation of a banking application [2], FusionTicket, a movie ticketing application [15], Auction, an online auction application [5], and Courseware, a course registration service extracted from [13, 18].

A first experiment concerns the reduction of robustness checking to SC reachability. For each application, we have constructed a client (i.e., a program composed of transactions defined within that application) with a fixed number of processes (at most 3) and a fixed number of transactions (between 3 and 7 transactions per process). We have encoded the instrumentation of this client, defined in Section 5, in the Boogie programming language [3] and used the Civl verifier [14] in order to check whether the assertions introduced by the instrumentation are violated (which would represent a robustness violation). Note that since clients are of fixed size, this requires no additional assertions/invariants (it is an instance of bounded model checking). The results are reported in Table 1. We have found two of the applications, Courseware and SmallBank, to *not* be robust against snapshot isolation. The violation in Courseware is caused by transactions RemoveCourse and EnrollStudent that execute concurrently, RemoveCourse removing a course that has no registered student and EnrollStudent registering a new student to the same course. We get an invalid state where a student is registered for a course that was removed. SmallBank's violation contains transactions Balance, TransactSaving, and WriteCheck. One process executes WriteCheck where it withdraws an amount from the checking account after checking that the sum of the checking and savings accounts is bigger than this amount. Concurrently, a second process executes TransactSaving where it

Table 1: An overview of the analysis results. CDG stands for commutativity dependency graph. The columns PO and PT show the number of proof obligations and proof time in second, respectively. T stands for trivial when the application has only read-only transactions.

Application	#Transactions	Robustness	Reachability Analysis		CDG Analysis	
			PO	PT	PO	PT
Auction	4	✓	70	0.3	20	0.5
Courseware	5	✗	59	0.37	na	na
FusionTicket	4	✓	72	0.3	34	0.5
SmallBank	5	✗	48	0.28	na	na
TPC-C	5	✓	54	0.7	82	3.7
Cassieq-Core	8	✓	173	0.55	104	2.9
Currency-Exchange	6	✓	88	0.35	26	3.5
PlayList	14	✓	99	4.63	236	7.3
RoomStore	5	✓	85	0.3	22	0.5
Shopping-Cart	4	✓	58	0.25	T	T

withdraws an amount from the saving account after checking that it is smaller than the amount in the savings account. Afterwards, the second process checks the contents of both the checking and saving accounts. We get an invalid state where the sum of the checking and savings accounts is negative.

Since in the first experiment we consider fixed clients, the lack of assertion violations doesn't imply that the application is robust (this instantiation of our reduction can only be used to reveal robustness violations). Thus, a second experiment concerns the robustness proof method based on commutativity dependency graphs (Section 6). For the applications that were not identified as non-robust by the previous method, we have used Civl to construct their commutativity dependency graphs, i.e., identify the "non-mover" relations M_{WR}, M_{WW}, and M_{RW} (Civl allows to check whether some code fragment is a left-/right mover). In all cases, the graph didn't contain non-mover cycles, which allows to conclude that the applications are robust.

The experiments show that our results can be used for finding violations and proving robustness, and that they apply to a large set of interesting examples. Note that the reduction to SC and the proof method based on commutativity dependency graphs are valid for programs with SQL (select/update) queries.

8 Related Work

Decidability and complexity of robustness has been investigated in the context of relaxed memory models such as TSO and Power [6, 8, 12]. Our work borrows some high-level principles from [6] which addresses the robustness against TSO. We reuse the high-level methodology of characterizing minimal violations according to some measure and defining reductions to SC reachability using a program instrumentation. Instantiating this methodology in our context is however very different, several fundamental differences being:

– `SI` and TSO admit different sets of relaxations and `SI` is a model of transactional databases.
– We use a different notion of measure: the measure in [6] counts the number of events between a write issue and a write commit while our notion of measure counts the number of delayed transactions. This is a first reason for which the proof techniques in [6] don't extend to our context.
– Transactions induce more complex traces: two transactions might be related by several dependency relations since each transaction may contain multiple reads and writes to different locations. In TSO, each action is a read or a write to some location, and two events are related by a single dependency relation. Also, the number of dependencies between two transactions depends on the execution since the set of reads/writes in a transaction evolves dynamically.

Other works [8, 12] define decision procedures which are based on the theory of regular languages and do not extend to infinite-state programs like in our case.

As far as we know, our work provides the first results concerning the decidability and the complexity of robustness checking in the context of transactions. The existing work on the verification of robustness for transactional programs provide either over- or under-approximate analyses. Our commutativity dependency graphs are similar to the static dependency graphs used in [5, 9, 10, 11], but they are more precise, i.e., reducing the number of false alarms. The static dependency graphs record happens-before dependencies between transactions based on a syntactic approximation of the variables accessed by a transaction. For example, our techniques are able to prove that the program in Figure 5 is robust, while this is not possible using static dependency graphs. The latter would contain a dependency from transaction t_1 to t_2 and one from t_2 to t_1 just because syntactically, each of the two transactions reads both variables and may write to one of them. Our dependency graphs take into account the semantics of these transactions and do not include this happens-before cycle. Other over- and under-approximate analyses have been proposed in [19]. They are based on encoding executions into first order logic, bounded-model checking for the under-approximate analysis, and a sound check for proving a cut-off bound on the size of the happens-before cycles possible in the executions of a program, for the over-approximate analysis. The latter is strictly less precise than our method based on commutativity dependency graphs. For instance, extending the TPC-C application with additional transactions will make the method in [19] fail while our method will succeed in proving robustness (the three transactions are for adding a new product, adding a new warehouse based on the number of customers and warehouses, and adding a new customer, respectively).

```
     p1:                          p2:
t1: [ if (x > y)     ||  t2: [ if (y > x)
      r1 = x - y                   r2 = y - x
      x  = y ]                     y  = x ]
```

Fig. 5: A robust program.

Finally, the idea of using Lipton's reduction theory for checking robustness has been also used in the context of the TSO memory model [7], but the techniques are completely different, e.g., the TSO technique considers each update in isolation and doesn't consider non-mover cycles like in our commutativity dependency graphs.

Bibliography

[1] Adya, A.: Weak consistency: A generalized theory and optimistic implementations for distributed transactions. Ph.D. thesis (1999)

[2] Alomari, M., Cahill, M.J., Fekete, A., Röhm, U.: The cost of serializability on platforms that use snapshot isolation. In: Alonso, G., Blakeley, J.A., Chen, A.L.P. (eds.) Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico. pp. 576–585. IEEE Computer Society (2008)

[3] Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures. Lecture Notes in Computer Science, vol. 4111, pp. 364–387. Springer (2005)

[4] Berenson, H., Bernstein, P.A., Gray, J., Melton, J., O'Neil, E.J., O'Neil, P.E.: A critique of ANSI SQL isolation levels. In: Carey, M.J., Schneider, D.A. (eds.) Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995. pp. 1–10. ACM Press (1995)

[5] Bernardi, G., Gotsman, A.: Robustness against consistency models with atomic visibility. In: Desharnais, J., Jagadeesan, R. (eds.) 27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada. LIPIcs, vol. 59, pp. 7:1–7:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016)

[6] Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7792, pp. 533–553. Springer (2013)

[7] Bouajjani, A., Enea, C., Mutluergil, S.O., Tasiran, S.: Reasoning about TSO programs using reduction and abstraction. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10982, pp. 336–353. Springer (2018)

[8] Bouajjani, A., Meyer, R., Möhlmann, E.: Deciding robustness against total store ordering. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II. Lecture Notes in Computer Science, vol. 6756, pp. 428–440. Springer (2011)

[9] Brutschy, L., Dimitrov, D., Müller, P., Vechev, M.T.: Serializability for eventual consistency: criterion, analysis, and applications. In: Castagna, G., Gor-

don, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 458–472. ACM (2017)

[10] Brutschy, L., Dimitrov, D., Müller, P., Vechev, M.T.: Static serializability analysis for causal consistency. In: Foster, J.S., Grossman, D. (eds.) Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018. pp. 90–104. ACM (2018)

[11] Cerone, A., Gotsman, A.: Analysing snapshot isolation. J. ACM **65**(2), 11:1–11:41 (2018)

[12] Derevenetc, E., Meyer, R.: Robustness against power is pspace-complete. In: Esparza, J., Fraigniaud, P., Husfeldt, T., Koutsoupias, E. (eds.) Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II. Lecture Notes in Computer Science, vol. 8573, pp. 158–170. Springer (2014)

[13] Gotsman, A., Yang, H., Ferreira, C., Najafzadeh, M., Shapiro, M.: 'cause i'm strong enough: reasoning about consistency choices in distributed systems. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 371–384. ACM (2016)

[14] Hawblitzel, C., Petrank, E., Qadeer, S., Tasiran, S.: Automated and modular refinement reasoning for concurrent programs. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9207, pp. 449–465. Springer (2015)

[15] Holt, B., Bornholt, J., Zhang, I., Ports, D.R.K., Oskin, M., Ceze, L.: Disciplined inconsistency with consistency types. In: Aguilera, M.K., Cooper, B., Diao, Y. (eds.) Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016. pp. 279–293. ACM (2016)

[16] Kozen, D.: Lower bounds for natural proof systems. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977. pp. 254–266. IEEE Computer Society (1977)

[17] Lipton, R.J.: Reduction: A method of proving properties of parallel programs. Commun. ACM **18**(12), 717–721 (1975)

[18] Nagar, K., Jagannathan, S.: Automated detection of serializability violations under weak consistency. In: Schewe, S., Zhang, L. (eds.) 29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China. LIPIcs, vol. 118, pp. 41:1–41:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2018)

[19] Nagar, K., Jagannathan, S.: Automatic detection of serializability violations under weak consistency. In: 29th Intern. Conf. on Concurrency Theory (CONCUR'18) (September 2018), to appear
[20] Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26**(4), 631–653 (1979)
[21] Rackoff, C.: The covering and boundedness problems for vector addition systems. Theor. Comput. Sci. **6**, 223–231 (1978)
[22] Shasha, D.E., Snir, M.: Efficient and correct execution of parallel programs that share memory. ACM Trans. Program. Lang. Syst. **10**(2), 282–312 (1988)
[23] TPC: Tech. rep., Transaction Processing Performance Council (February 2010), `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf`