Checking Robustness Between Weak Transactional Consistency Models[⋆]

Sidi Mohamed Beillahi(✉), Ahmed Bouajjani, and Constantin Enea

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

Abstract. Concurrent accesses to databases are typically encapsulated in transactions in order to enable isolation from other concurrent computations and resilience to failures. Modern databases provide transactions with various semantics corresponding to different trade-offs between consistency and availability. Since a weaker consistency model provides better performance, an important issue is investigating the weakest level of consistency needed by a given program (to satisfy its specification). As a way of dealing with this issue, we investigate the problem of checking whether a given program has the same set of behaviors when replacing a consistency model with a weaker one. This property known as *robustness* generally implies that any specification of the program is preserved when weakening the consistency. We focus on the robustness problem for consistency models which are weaker than standard serializability, namely, causal consistency, prefix consistency, and snapshot isolation. We show that checking robustness between these models is polynomial time reducible to a state reachability problem under serializability. We use this reduction to also derive a pragmatic proof technique based on Lipton's reduction theory that allows to prove programs robust. We have applied our techniques to several challenging applications drawn from the literature of distributed systems and databases.

Keywords: Transactional databases · Weak consistency · Program verification

1 Introduction

Concurrent accesses to databases are typically encapsulated in transactions in order to enable isolation from other concurrent computations and resilience to failures. Modern databases provide transactions with various semantics corresponding to different tradeoffs between consistency and availability. The strongest consistency level is achieved with *serializable* transactions [42] whose outcome in concurrent executions is the same as if the transactions were executed atomically in some order. Since serializability (SER) carries a significant penalty on availability, modern databases often provide weaker consistency models, e.g.,

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

causal consistency (CC) [38], prefix consistency (PC) [22, 25], and snapshot isolation (SI) [12]. Causal consistency requires that if a transaction t_1 "affects" another transaction t_2, e.g., t_1 executes before t_2 in the same session or t_2 reads a value written by t_1, then the updates in these two transactions are observed by any other transaction in this order. Concurrent transactions, which are not causally related to each other, can be observed in different orders, leading to behaviors that are not possible under SER. Prefix consistency requires that there is a total commit order between all the transactions such that each transaction observes all the updates in a prefix of this sequence (PC is stronger than CC). Two transactions can observe the *same* prefix, which leads to behaviors that are not admitted by SER. Snapshot isolation further requires that two different transactions observe different prefixes if they both write to a common variable.

Since a weaker consistency model provides better performance, an important issue is identifying the *weakest* level of consistency needed by a program (to satisfy its specification). One way to tackle this issue is checking whether a program P designed under a consistency model S has the same behaviors when run under a weaker consistency model W. This property of a program is generally known as *robustness* against substituting S with W. It implies that any specification of P is preserved when weakening the consistency model (from S to W). Preserving any specification is convenient since specifications are rarely present in practice.

The problem of checking robustness for a given program has been investigated in several recent works, but only when the stronger model (S) is SER, e.g., [9, 10, 19, 26, 13, 40], or sequential consistency in the non-transactional case, e.g. [36, 15, 29]. However, there is a large class of specifications that can be implemented even in the presence of "anomalies", i.e., behaviors which are not admitted under SER (see [46] for a discussion). In this context, an important question is whether a certain implementation (program) is robust against substituting a weak consistency model, e.g., SI, with a weaker one, e.g., CC.

In this paper, we consider the sequence of increasingly strong consistency models mentioned above, CC, PC, and SI, and investigate the problem of checking robustness for a given program against weakening the consistency model to one in this range. We study the asymptotic complexity of this problem and propose effective techniques for establishing robustness based on abstraction. There are two important cases to consider: robustness against substituting SI with PC and PC with CC, respectively. Robustness against substituting SI with CC can be obtained as the conjunction of these two cases.

In the first case (SI vs PC), checking robustness for a program P is reduced to a reachability (assertion checking) problem in a composition of P under PC with a monitor that checks whether a PC behavior is an "anomaly", i.e., admitted by P under PC, but not under SI. This approach raises two non-trivial challenges: (1) defining a monitor for detecting PC vs SI anomalies that uses a minimal amount of auxiliary memory (to remember past events), and (2) determining the complexity of checking if the composition of P with the monitor reaches a specific control location[1] under the (weaker) model PC. Interestingly enough,

[1] We assume that the monitor goes to an error location when detecting an anomaly.

we address these two challenges by studying the relationship between these two weak consistency models, PC and SI, and *serializability.* The construction of the monitor is based on the fact that the PC vs SI anomalies can be defined as roughly, the difference between the PC vs SER and SI vs SER anomalies (investigated in previous work [13]), and we show that the reachability problem under PC can be reduced to a reachability problem under SER. These results lead to a polynomial-time reduction of this robustness problem (for arbitrary programs) to a reachability problem under SER, which is important from a practical point of view since the SER semantics (as opposed to the PC or SI semantics) can be encoded easily in existing verification tools (using locks to guard the isolation of transactions). These results also enable a precise characterization of the complexity class of this problem.

Checking robustness against substituting PC with CC is reduced to the problem of checking robustness against substituting SER with CC. The latter has been shown to be polynomial-time reducible to reachability under SER in [10]. This surprising result relies on the reduction from PC reachability to SER reachability mentioned above. This reduction shows that a given program P reaches a certain control location under PC iff a transformed program P', where essentially, each transaction is split in two parts, one part containing all the reads, and one part containing all the writes, reaches the same control location under SER. Since this reduction preserves the structure of the program, CC vs PC anomalies of a program P correspond to CC vs SER anomalies of the transformed program P'.

Beyond enabling these reductions, the characterization of classes of anomalies or the reduction from the PC semantics to the SER semantics are also important for a better understanding of these weak consistency models and the differences between them. We believe that these results can find applications beyond robustness checking, e.g., verifying conformance to given specifications.

As a more pragmatic approach for establishing robustness, which avoids a non-reachability proof under SER, we have introduced a proof methodology that builds on Lipton's reduction theory [39] and the concept of commutativity dependency graph introduced in [9], which represents mover type dependencies between the transactions in a program. We give sufficient conditions for robustness in all the cases mentioned above, which characterize the commutativity dependency graph associated to a given program.

We tested the applicability of these verification techniques on a benchmark containing seven challenging applications extracted from previous work [30, 34, 19]. These techniques are precise enough for proving or disproving the robustness of all these applications, for all combinations of the consistency models.

Complete proofs and more details can be found in [11].

2 Overview

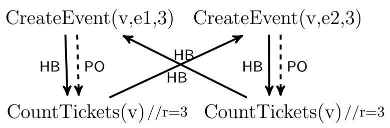
We give an overview of the robustness problems investigated in this paper, discussing first the case PC vs. CC, and then SI vs PC. We end with an example that illustrates the robustness checking technique based on commutativity arguments.

```
      Process 1                  Process 2

CreateEvent(v, e1, 3):     CreateEvent(v, e2, 3):
[ Tickets[v][e1] := 3 ]    [ Tickets[v][e2] := 3 ]

CountTickets(v):           CountTickets(v):
[ r := ∑Tickets[v][e] ]    [ r := ∑Tickets[v][e] ]
        e                          e
```

(a) FusionTicket.

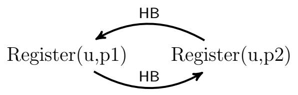


(b) A CC trace of FusionTicket.

```
      Process 1                  Process 2

Register(u, p1):              Register(u, p2):
[ r := RegisteredUsers[u]     [ r := RegisteredUsers[u]
  assume r == 0                 assume r == 0
  RegisteredUsers[u] := 1       RegisteredUsers[u] := 1
  Password[u] := p1 ]           Password[u] := p2 ]
```

(c) Twitter.



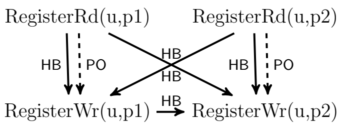
(d) A CC and PC trace of Twitter.

```
      Process 1                  Process 2

RegisterRd(u, p1):            RegisterRd(u, p2):
[ r := RegisteredUsers[u]     [ r := RegisteredUsers[u]
  assume r == 0 ]               assume r == 0 ]

RegisterWr(u, p1):            RegisterWr(u, p2):
[ RegisteredUsers[u] := 1     [ RegisteredUsers[u] := 1
  Password[u] := p1 ]           Password[u] := p2 ]
```

(e) Transformed Twitter.

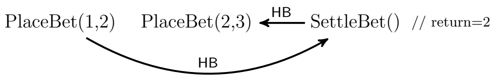


(f) A CC and SER trace of transformed Twitter.

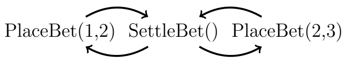
```
      Process 1                  Process 2                  Process 3

PlaceBet(1,2):               PlaceBet(2,3):               SettleBet():
[ assume time < TIMEOUT      [ assume time < TIMEOUT      [Bets' := Bets
    Bets[1] := 2 ]               Bets[2] := 3 ]             n := Bets'.Length
                                                            assume time > TIMEOUT & n > 0
                                                            select i s.t. Bets'[i] ≠ ⊥
                                                            return := Bets'[i] ]
```

(g) Betting.



(h) A PC and SI trace of Betting.



(i) Commutativity dependency graph of Betting.

Fig. 1: Transactional programs and traces under different consistency models.

Robustness PC vs CC. We illustrate the robustness against substituting PC with CC using the FusionTicket and the Twitter programs in Figure 1a and Figure 1c, respectively. FusionTicket manages tickets for a number of events, each event being associated with a venue. Its state consists of a two-dimensional map that stores the number of tickets for an event in a given venue (r is a local variable, and the assignment in `CountTickets` is interpreted as a read of the shared state). The program has two processes and each process contains two transactions. The first transaction creates an event e in a venue v with a number of tickets n, and the second transaction computes the total number of tickets for all the events in a venue v. A possible candidate for a specification of this program is that the values computed in `CountTickets` are monotonically increasing since each such value is computed after creating a new event. Twitter provides a transaction for registering a new user with a given username and password, which is executed by two parallel processes. Its state contains two maps that record whether a given username has been registered (0 and 1 stand for non-registered and registered, respectively) and the password for a given username. Each transaction first checks whether a given username is free (see the `assume` statement). The intended specification is that the user must be registered with the given password when the registration transaction succeeds.

A program is robust against substituting PC with CC if its set of behaviors under the two models coincide. We model behaviors of a given program as *traces*, which record standard control-flow and data-flow dependencies between transactions, e.g., the order between transactions in the same session and whether a transaction reads the value written by another (read-from). The transitive closure of the union of all these dependency relations is called *happens-before.* Figure 1b pictures a trace of FusionTicket where the concrete values which are read in a transaction are written under comments. In this trace, each process registers a different event but in the same venue and with the same number of tickets, and it ignores the event created by the other process when computing the sum of tickets in the venue.

Figure 1b pictures a trace of FusionTicket under CC, which is a witness that FusionTicket is *not* robust against substituting PC with CC. This trace is also a violation of the intended specification since the number of tickets is not increasing (the sum of tickets is 3 in both processes). The happens-before dependencies (pictured with HB labeled edges) include the program-order PO (the order between transactions in the same process), and read-write dependencies, since an instance of CountTickets(v) does not observe the value written by the CreateEvent transaction in the other process (the latter overwrites some value that the former reads). This trace is allowed under CC because the transaction CreateEvent(v, e1, 3) executes concurrently with the transaction CountTickets(v) in the other process, and similarly for CreateEvent(v, e2, 3). However, it is not allowed under PC since it is impossible to define a total commit order between CreateEvent(v, e1, 3) and CreateEvent(v, e2, 3) that justifies the reads of both CountTickets(v) transactions (these reads should correspond to the updates in a prefix of this order). For instance, assuming that CreateEvent(v, e1, 3) commits before CreateEvent(v, e2, 3), CountTickets(v) in the second process must observe the effect of CreateEvent(v, e1, 3) as well since it observes the effect of CreateEvent(v, e2, 3). However, this contradicts the fact that CountTickets(v) computes the sum of tickets as being 3.

On the other hand, Twitter is robust against substituting PC with CC. For instance, Figure 1d pictures a trace of Twitter under CC, where the `assume` in both transactions pass. In this trace, the transactions Register(u,p1) and Register(u,p2) execute concurrently and are unaware of each other's writes (they are not causally related). The HB dependencies include write-write dependencies since both transactions write on the same location (we consider the transaction in Process 2 to be the last one writing to the `Password` map), and read-write dependencies since each transaction reads `RegisteredUsers` that is written by the other. This trace is also allowed under `PC` since the commit order can be defined such that Register(u,p1) is ordered before Register(u,p2), and then both transactions read from the initial state (the empty prefix). Note that this trace has a cyclic happens-before which means that it is not allowed under serializability.

Checking robustness `PC` vs `CC`. We reduce the problem of checking robustness against substituting `PC` with `CC` to the robustness problem against substituting `SER` with `CC` (the latter reduces to a reachability problem under `SER` [10]). This reduction relies on a syntactic program transformation that rewrites `PC` behaviors of a given program P to `SER` behaviors of another program P'. The program P' is obtained by splitting each transaction t of P into two transactions: the first transaction performs all the reads in t and the second performs all the writes in t (the two are related by program order). Figure 1e shows this transformation applied on Twitter. The trace in Figure 1f is a serializable execution of the transformed Twitter which is "observationally" equivalent to the trace in Figure 1d of the original Twitter, i.e., each read of the shared state returns the same value and the writes on the shared state are applied in the same order (the acyclicity of the happens-before shows that this is a serializable trace). The transformed FusionTicket coincides with the original version because it contains no transaction that both reads and writes on the shared state.

We show that `PC` behaviors and `SER` behaviors of the original and transformed program, respectively, are related by a bijection. In particular, we show that any `PC` vs. `CC` robustness violation of the original program manifests as a `SER` vs. `CC` robustness violation of the transformed program, and vice-versa. For instance, the `CC` trace of the original Twitter in Figure 1d corresponds to the `CC` trace of the transformed Twitter in Figure 1f, and the acyclicity of the latter (the fact that it is admitted by `SER`) implies that the former is admitted by the original Twitter under `PC`. On the other hand, the trace in Figure 1b is also a `CC` of the transformed FusionTicket and its cyclicity implies that it is not admitted by FusionTicket under `PC`, and thus, it represents a robustness violation.

Robustness `SI` vs `PC`. We illustrate the robustness against substituting `SI` with `PC` using Twitter and the Betting program in Figure 1g. Twitter is *not* robust against substituting `SI` with `PC`, the trace in Figure 1d being a witness violation. This trace is also a violation of the intended specification since one of the users registers a password that is overwritten in a concurrent transaction. This `PC` trace is not possible under `SI` because Register(u,p1) and Register(u,p2) observe the same prefix of the commit order (i.e., an empty prefix), but they write to a common memory location Password[u] which is not allowed under `SI`.

On the other hand, the Betting program in Figure 1g, which manages a set of bets, is robust against substituting `SI` with `PC`. The first two processes execute one transaction that places a bet of a value v with a unique bet identifier id, assuming that the bet expiration time is not yet reached (bets are recorded in the map `Bets`). The third process contains a single transaction that settles the betting assuming that the bet expiration time was reached and at least one bet has been placed. This transaction starts by taking a snapshot of the `Bets` map into a local variable `Bets'`, and then selects a random non-null value (different from $\perp$) in the map to correspond to the winning bet. The intended specification of this program is that the winning bet corresponds to a genuine bet that was placed. Figure 1g pictures a PC trace of Betting where SettleBet observes only the bet of the first process PlaceBet(1,2). The HB dependency towards the second process denotes a read-write dependency (SettleBet reads a cell of the map `Bets` which is overwritten by the second process). This trace is allowed under SI because no two transactions write to the same location.

Checking robustness SI vs PC. We reduce robustness against substituting PC with CC to a reachability problem under SER. This reduction is based on a characterization of happens-before cycles[2] that are possible under PC but not SI, and the transformation described above that allows to simulate the PC semantics of a program on top of SER. The former is used to define an instrumentation (monitor) for the transformed program that reaches an error state iff the original program is not robust. Therefore, we show that the happens-before cycles in PC traces that are not admitted by SI must contain a transaction that (1) overwrites a value written by another transaction in the cycle and (2) reads a value overwritten by another transaction in the cycle. For instance, the trace of Twitter in Figure 1d is not allowed under SI because Register(u,p2) overwrites a value written by Register(u,p1) (the password) and reads a value overwritten by Register(u,p1) (checking whether the username u is registered). The trace of Betting in Figure 1g is allowed under SI because its happens-before is acyclic.

Checking robustness using commutativity arguments. Based on the reductions above, we propose an approximated method for proving robustness based on the concept of mover in Lipton's reduction theory [39]. A transaction is a left (resp., right) mover if it commutes to the left (resp., right) of another transaction (by a different process) while preserving the computation. We use the notion of mover to characterize the data-flow dependencies in the happens-before. Roughly, there exists a data-flow 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 transformed program (obtained by splitting the transactions of the original program as explained above), and derive a proof method for robustness which inspects paths in this graph. Two transactions t_1 and t_2 are linked by a directed edge iff t_1 *cannot* move to the right of t_2 (or t_2 cannot move to the left of t_1), or if they are related by the program order. Moreover, two transactions t_1 and t_2 are linked by an undirected edge iff they are the result of splitting the same transaction.

A program is robust against substituting PC with CC if roughly, its commutativity dependency graph does *not* contain a *simple* cycle of directed edges with two distinct transactions t_1 and t_2, such that t_1 does not commute left because of another transaction t_3 in the cycle that reads a variable that t_1 writes to,

[2] Traces with an acyclic happens-before are not robustness violations because they are admitted under serializability, which implies that they are admitted under the weaker model SI as well.

$\langle \mathit{prog} \rangle$ ::= `program` $\langle \mathit{process} \rangle^*$

$\langle \mathit{process} \rangle$ ::= `process` $\langle \mathit{pid} \rangle$ `regs` $\langle \mathit{reg} \rangle^*$ $\langle \mathit{txn} \rangle^*$

$\langle \mathit{txn} \rangle$ ::= `begin` $\langle \mathit{read} \rangle^*$ $\langle \mathit{test} \rangle^*$ $\langle \mathit{write} \rangle^*$ `commit`

$\langle \mathit{read} \rangle$ ::= $\langle \mathit{label} \rangle$`:` $\langle \mathit{reg} \rangle$ `:=` $\langle \mathit{var} \rangle$`;` `goto` $\langle \mathit{label} \rangle$`;`

$\langle \mathit{test} \rangle$ ::= $\langle \mathit{label} \rangle$`:` `assume` $\langle \mathit{bexpr} \rangle$`;` `goto` $\langle \mathit{label} \rangle$`;`

$\langle \mathit{write} \rangle$ ::= $\langle \mathit{label} \rangle$`:` $\langle \mathit{var} \rangle$ `:=` $\langle \mathit{reg\text{-}expr} \rangle$`;` `goto` $\langle \mathit{label} \rangle$`;`

Fig. 2: The syntax of our programming language. a^* indicates zero or more occurrences of a. $\langle \mathit{pid} \rangle$, $\langle \mathit{reg} \rangle$, $\langle \mathit{label} \rangle$, and $\langle \mathit{var} \rangle$ represent a process identifier, a register, a label, and a shared variable, respectively. $\langle \mathit{reg\text{-}expr} \rangle$ is an expression over registers while $\langle \mathit{bexpr} \rangle$ is a Boolean expression over registers, or the non-deterministic choice $*$.

and t_2 does not commute right because of another transaction t_4 in the cycle (t_3 and t_4 can coincide) that writes to a variable that t_2 either reads from or writes to[3]. For instance, Figure 1i shows the commutativity dependency graph of the transformed Betting program, which coincides with the original Betting because PlaceBet(1,2) and PlaceBet(2,3) are write-only transactions and SettleBet() is a read-only transaction. Both simple cycles in Figure 1i contain just two transactions and therefore do not meet the criterion above which requires at least 3 transactions. Therefore, Betting is robust against substituting `PC` with `CC`.

A program is robust against substituting `SI` with `PC`, if roughly, its commutativity dependency graph does *not* contain a *simple* cycle with two successive transactions t_1 and t_2 that are linked by an undirected edge, such that t_1 does not commute left because of another transaction t_3 in the cycle that writes to a variable that t_1 writes to, and t_2 does not commute right because of another transaction t_4 in the cycle (t_3 and t_4 can coincide) that writes to a variable that t_2 reads from[4]. Betting is also robust against substituting `SI` with `PC` for the same reason (simple cycles of size 2).

3 Consistency Models

Syntax. We present our results in the context of the simple programming language, defined in Figure 2, where a program is a parallel composition of *processes* distinguished using a set of identifiers $\mathbb{P}$. A process is a sequence of *transactions* and each transaction is a sequence of *labeled instructions*. A transaction starts with a `begin` instruction and finishes with a `commit` instruction. Instructions include assignments to a process-local *register* from a set $\mathbb{R}$ or to a *shared variable* from a set $\mathbb{V}$, or an `assume`. The assignments use values from a data domain

[3] The transactions t_1, t_2, t_3, and t_4 correspond to t_1, t_i, t_n, and t_{i+1}, respectively, in Theorem 6.

[4] The transactions t_1, t_2, t_3, and t_4 correspond to t_1, t_2, t_n, and t_3, respectively, in Theorem 7.

$\mathbb{D}$. 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 $\langle var \rangle := \langle reg \rangle$ is called a *write* to the shared-variable $\langle var \rangle$. The `assume` $\langle bexpr \rangle$ blocks the process if the Boolean expression $\langle bexpr \rangle$ over registers is false. It can be used to model conditionals. The `goto` statement transfers the control to the program location (instruction) specified by a given label. Since multiple instructions can have the same label, `goto` statements can be used to mimic imperative constructs like loops and conditionals inside transactions.

We assume w.l.o.g. that every transaction is written as a sequence of reads or `assume` statements followed by a sequence of writes (a single `goto` statement from the sequence of read/`assume` instructions transfers the control to the sequence of writes). In the context of the consistency models we study in this paper, every program can be equivalently rewritten as a set of transactions of this form.

To simplify the technical exposition, programs contain a bounded number of processes and each process executes a bounded number of transactions. A transaction may execute an unbounded number of instructions but these instructions concern a bounded number of variables, which makes it impossible to model SQL (select/update) queries that may access tables with a statically unknown number of rows. Our results can be extended beyond these restrictions as explained in Remark 1 and Remark 2.

Semantics. We describe the semantics of a program under four consistency models, i.e., causal consistency[5] (`CC`), prefix consistency (`PC`), snapshot isolation (`SI`), and serializability (`SER`).

In the semantics of a program under `CC`, shared variables are replicated across each process, each process maintaining its own local valuation of these variables. During the execution of a transaction in a process, its writes are stored in a *transaction log* that can be accessed only by the process executing the transaction and that is broadcasted to all the other processes at the end of the transaction. To read a shared variable x, a process p first accesses its transaction log and takes the last written value on x, if any, and then its own valuation of the shared variable, if x was not written during the current transaction. Transaction logs are delivered to every process in an order consistent with the *causal* relation between transactions, i.e., the transitive closure of the union of the *program order* (the order in which transactions are executed by a process), and the *read-from* relation (a transaction t_1 reads-from a transaction t_2 iff t_1 reads a value that was written by t_2). When a process receives a transaction log, it immediately applies it on its shared-variable valuation.

In the semantics of a program under `PC` and `SI`, shared variables are stored in a central memory and each process keeps a local valuation of these variables. When a process starts a new transaction, it fetches a consistent snapshot of the shared variables from the central memory and stores it in its local valuation of these variables. During the execution of a transaction in a process, writes to shared variables are stored in the local valuation of these variables, and in a transaction log. To read a shared variable, a process takes its own valuation of the

[5] We consider a variation known as causal convergence [20, 16]

shared variable. A process commits a transaction by applying the updates in the transaction log on the central memory in an atomic way (to make them visible to all processes). Under SI, when a process applies the writes in a transaction log on the central memory, it must ensure that there were no concurrent writes that occurred after the last fetch from the central memory to a shared variable that was written during the current transaction. Otherwise, the transaction is aborted and its effects discarded.

In the semantics of a program under SER, we adopt a simple operational model where we keep a single shared-variable valuation in a central memory (accessed by all processes) with the standard interpretation of read and write statements. Transactions execute serially, one after another.

We use a standard model of executions of a program called *trace*. A trace represents the order between transactions in the same process, and the data-flow in an execution using standard happens-before relations between transactions. We assume that each transaction in a program is identified uniquely using a transaction identifier from a set $\mathbb{T}$. Also, $f : \mathbb{T} \to 2^{\mathbb{S}}$ is a mapping that associates each transaction in $\mathbb{T}$ with a sequence of read and write events from the set

$$\mathbb{S} = \{\mathsf{re}(t, x, v), \mathsf{we}(t, x, v) : t \in \mathbb{T}, x \in \mathbb{V}, v \in \mathbb{D}\}$$

where $\mathsf{re}(t, x, v)$ is a read of x returning v, and $\mathsf{we}(t, x, v)$ is a write of v to x.

Definition 1. *A* trace *is a tuple* $\tau = (\rho, f, \mathsf{TO}, \mathsf{PO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$ *where* $\rho \subseteq \mathbb{T}$ *is a set of transaction identifiers, and*

- TO *is a mapping giving the order between events in each transaction, i.e., it associates each transaction* t *in* ρ *with a total order* $\mathsf{TO}(t)$ *on* $f(t) \times f(t)$.
- PO *is the program order relation, a strict partial order on* $\rho \times \rho$ *that orders every two transactions issued by the same process.*
- WR *is the read-from relation between distinct transactions* $(t1, t2) \in \rho \times \rho$ *representing the fact that* $t2$ *reads a value written by* $t1$.
- WW *is the store order relation on* $\rho \times \rho$ *between distinct transactions that write to the same shared variable.*
- RW *is the conflict order relation between distinct transactions, defined by* $\mathsf{RW} = \mathsf{WR}^{-1}; \mathsf{WW}$ *(; denotes the sequential composition of two relations).*

For simplicity, for a trace $\tau = (\rho, f, \mathsf{TO}, \mathsf{PO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$, we write $t \in \tau$ instead of $t \in \rho$. We also assume that each trace contains a fictitious transaction that writes the initial values of all shared variables, and which is ordered before any other transaction in program order. Also, $\mathbb{T}\mathsf{r}_{\mathsf{X}}(\mathcal{P})$ is the set of traces representing executions of program $\mathcal{P}$ under a consistency model X.

For each $\mathsf{X} \in \{\mathtt{CC}, \mathtt{PC}, \mathtt{SI}, \mathtt{SER}\}$, the set of traces $\mathbb{T}\mathsf{r}_{\mathsf{X}}(\mathcal{P})$ can be described using the set of properties in Table 1. A trace τ is possible under causal consistency iff there exist two relations CO a partial order (causal order) and ARB a total order (arbitration order) that includes CO, such that the properties $\mathsf{AxCausal}$, AxArb, and $\mathsf{AxRetVal}$ hold [27, 16]. $\mathsf{AxCausal}$ guarantees that the program order and the read-from relation are included in the causal order, and AxArb guarantees

AxCausal	$\mathsf{CO}_0^+ \subseteq \mathsf{CO}$
AxArb	$ARB_0^+ \subseteq ARB$
AxCC	$\mathsf{AxRetVal} \wedge \mathsf{AxCausal} \wedge \mathsf{AxArb}$
AxPrefix	$ARB; \mathsf{CO} \subseteq \mathsf{CO}$
AxPC	$\mathsf{AxPrefix} \wedge \mathsf{AxCC}$
AxConflict	$\mathsf{WW} \subseteq \mathsf{CO}$
AxSI	$\mathsf{AxConflict} \wedge \mathsf{AxPC}$
AxSer	$\mathsf{AxRetVal} \wedge \mathsf{AxCausal} \wedge \mathsf{AxArb} \wedge \mathsf{CO} = ARB$

where
$\mathsf{CO}_0 = \mathsf{PO} \cup \mathsf{WR}$ and $ARB_0 = \mathsf{PO} \cup \mathsf{WR} \cup \mathsf{WW}$
$\mathsf{AxRetVal} = \forall\, t \in \tau.\ \forall\ \mathsf{re}(t, x, v) \in f(t)$ we have that

- there exist a transaction $t_0 = Max_{ARB}(\{t' \in \tau \mid (t', t) \in \mathsf{CO} \wedge \exists\ \mathsf{we}(t', x, \cdot) \in f(t')\})$ and an event $\mathsf{we}(t_0, x, v) = Max_{\mathsf{TO}(t_0)}(\{\mathsf{we}(t_0, x, \cdot) \in f(t_0)\})$.

Table 1: Declarative definitions of consistency models. For an order relation $\leq$, $a = Max_{\leq}(A)$ iff $a \in A \wedge \forall\ b \in A.\ b \leq a$.

that the causal order and the store order are included in the arbitration order. AxRetVal guarantees that a read returns the value written by the last write in the last transaction that contains a write to the same variable and that is ordered by CO before the read's transaction. We use AxCC to denote the conjunction of these three properties. A trace τ is possible under prefix consistency iff there exist a causal order CO and an arbitration order ARB such that AxCC holds and the property AxPrefix holds as well [27]. AxPrefix guarantees that every transaction observes a prefix of transactions that are ordered by ARB before it. We use AxPC to denote the conjunction of AxCC and AxPrefix. A trace τ is possible under snapshot isolation iff there exist a causal order CO and an arbitration order ARB such that AxPC holds and the property AxConflict holds [27]. AxConflict guarantees that if two transactions write to the same variable then one of them must observe the other. We use AxSI to denote the conjunction of AxPC and AxConflict. A trace τ is serializable iff there exist a causal order CO and an arbitration order ARB such that the property AxSer holds which implies that the two relations CO and ARB coincide. Note that for any given program $\mathcal{P}$, $\mathbb{T}\mathsf{r}_{\mathtt{SER}}(\mathcal{P}) \subseteq \mathbb{T}\mathsf{r}_{\mathtt{SI}}(\mathcal{P}) \subseteq \mathbb{T}\mathsf{r}_{\mathtt{PC}}(\mathcal{P}) \subseteq \mathbb{T}\mathsf{r}_{\mathtt{CC}}(\mathcal{P})$. Also, the four consistency models we consider disallow anomalies such as dirty and phantom reads.

For a given trace $\tau = (\rho, f, \mathsf{TO}, \mathsf{PO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$, the happens before order is the transitive closure of the union of all the relations in the trace, i.e., $\mathsf{HB} = (\mathsf{PO} \cup \mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW})^+$. A classic result states that a trace τ is serializable iff HB is acyclic [2, 47]. Note that HB is acyclic implies that WW is a total order between transactions that write to the same variable, and $(\mathsf{PO} \cup \mathsf{WR})^+$ and $(\mathsf{PO} \cup \mathsf{WR} \cup \mathsf{WW})^+$ are acyclic.

3.1 Robustness

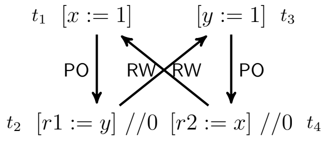
In this work, we investigate the problem of checking whether a program $\mathcal{P}$ under a semantics $\mathsf{Y} \in \{\mathtt{PC},\ \mathtt{SI}\}$ produces the same set of traces as under a weaker semantics $\mathsf{X} \in \{\mathtt{CC},\ \mathtt{PC}\}$. When this holds, we say that $\mathcal{P}$ is *robust* against X relative to Y.

Definition 2. *A program $\mathcal{P}$ is called* robust *against a semantics $\mathsf{X} \in \{\texttt{CC}, \texttt{PC}, \texttt{SI}\}$ relative to a semantics $\mathsf{Y} \in \{\texttt{PC}, \texttt{SI}, \texttt{SER}\}$ such that Y is stronger than X iff $\mathbb{T}r_\mathsf{X}(\mathcal{P}) = \mathbb{T}r_\mathsf{Y}(\mathcal{P})$.*

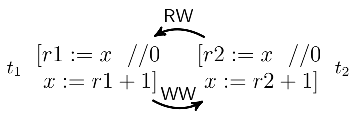
If $\mathcal{P}$ is not robust against X relative to Y then there must exist a trace $\tau \in \mathbb{T}r_\mathsf{X}(\mathcal{P}) \setminus \mathbb{T}r_\mathsf{Y}(\mathcal{P})$. We say that τ is a robustness violation trace.

We illustrate the notion of robustness on the programs in Figure 3, which are commonly used in the literature. In all programs, transactions of the same process are aligned vertically and ordered from top to bottom. Each read instruction is commented with the value it reads in some execution.

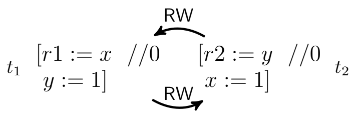
The store buffering (SB) program in Figure 3a contains four transactions that are issued by two distinct processes. We emphasize an execution where t_2 reads 0 from y and t_4 reads 0 from x. This execution is allowed under $\texttt{CC}$ since the two writes by t_1 and t_3 are not causally dependent. Thus, t_2 and t_4 are executed without seeing the writes from t_3 and t_1, respectively. However, this execution is not feasible under $\texttt{PC}$ (which implies that it is not feasible under both $\texttt{SI}$ and $\texttt{SER}$). In particular, we cannot have neither $(t_1, t_3) \in ARB$ nor $(t_3, t_1) \in ARB$ which contradicts the fact that ARB is total order. For example, if $(t_1, t_3) \in ARB$, then $(t_1, t_4) \in \mathsf{CO}$ (since $ARB; \mathsf{CO} \subset \mathsf{CO}$) which contradicts the fact that t_4 does not see t_1. Similarly, $(t_3, t_1) \in ARB$ implies that $(t_3, t_2) \in \mathsf{CO}$ which contradicts the fact that t_2 does not see t_3. Thus, SB is not robust against $\texttt{CC}$ relative to $\texttt{PC}$.



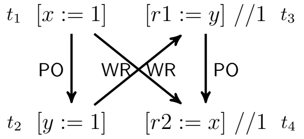
(a) Store Buffering (SB).



(b) Lost Update (LU).



(c) Write Skew (WS).



(d) Message Passing (MP).

Fig. 3: Litmus programs

The lost update (LU) program in Figure 3b has two transactions that are issued by two distinct processes. We highlight an execution where both transactions read 0 from x. This execution is allowed under $\texttt{PC}$ since both transactions are not causally dependent and can be executed in parallel by the two processes. However, it is not allowed under $\texttt{SI}$ since both transactions write to a common variable (i.e., x). Thus, they cannot be executed in parallel and one of them must see the write of the other. Thus, SB is not robust against $\texttt{PC}$ relative to $\texttt{SI}$.

The write skew (WS) program in Figure 3c has two transactions that are issued by two distinct processes. We highlight an execution where t_1 reads 0 from x and t_2 reads 0 from y. This execution is allowed under $\texttt{SI}$ since both transactions are not causally dependent, do not write to a common variable, and can be executed in parallel by the two processes. However, this execution is not allowed under $\texttt{SER}$ since one of the two transactions must see the write of the other. Thus, WS is not robust against $\texttt{SI}$ relative to $\texttt{SER}$.

The message passing (MP) program in Figure 3d has four transactions issued by two processes. Because t_1 and t_2 are causally dependent, under any semantics $\mathsf{X} \in \{\mathtt{CC}, \mathtt{PC}, \mathtt{SI}, \mathtt{SER}\}$ we only have three possible executions of MP, which correspond to either t_3 and t_4 not observing the writes of t_1 and t_2, or t_3 and t_4 observe the writes of both t_1 and t_2, or t_4 observes the write of t_1 (we highlight the values read in the second case in Figure 3d). Therefore, the executions of this program under the four consistency models coincide. Thus, MP is robust against CC relative to any other model.

4 Robustness Against CC Relative to PC

We show that checking robustness against CC relative to PC can be reduced to checking robustness against CC relative to SER. The crux of this reduction is a program transformation that allows to simulate the PC semantics of a program $\mathcal{P}$ using the SER semantics of a program $\mathcal{P}_\clubsuit$. Checking robustness against CC relative to SER can be reduced in polynomial time to reachability under SER [10].

Given a program $\mathcal{P}$ with a set of transactions $\mathsf{Tr}(\mathcal{P})$, we define a program $\mathcal{P}_\clubsuit$ such that every transaction $t \in \mathsf{Tr}(\mathcal{P})$ is split into a transaction $t[r]$ that contains all the read/`assume` statements in t (in the same order) and another transaction $t[w]$ that contains all the write statements in t (in the same order). In the following, we establish the following result:

Theorem 1. *A program $\mathcal{P}$ is robust against* CC *relative to* PC *iff $\mathcal{P}_\clubsuit$ is robust against* CC *relative to* SER.

Intuitively, under PC, processes can execute concurrent transactions that fetch the same consistent snapshot of the shared variables from the central memory and subsequently commit their writes. Decoupling the read part of a transaction from the write part allows to simulate such behaviors even under SER.

The proof of this theorem relies on several intermediate results concerning the relationship between traces of $\mathcal{P}$ and $\mathcal{P}_\clubsuit$. Let $\tau = (\rho, \mathsf{PO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}) \in \mathbb{T}\mathsf{r}_\mathsf{X}(\mathcal{P})$ be a trace of a program $\mathcal{P}$ under a semantics X. We define the trace $\tau_\clubsuit = (\rho_\clubsuit, \mathsf{PO}_\clubsuit, \mathsf{WR}_\clubsuit, \mathsf{WW}_\clubsuit, \mathsf{RW}_\clubsuit)$ where every transaction $t \in \tau$ is split into two transactions $t[r] \in \tau_\clubsuit$ and $t[w] \in \tau_\clubsuit$, and the dependency relations are straightforward adaptations, i.e.,

- $\mathsf{PO}_\clubsuit$ is the smallest transitive relation that includes $(t[r], t[w])$ for every t, and $(t[w], t'[r])$ if $(t, t') \in \mathsf{PO}$,
- $(t'[w], t[r]) \in \mathsf{WR}_\clubsuit$, $(t'[w], t[w]) \in \mathsf{WW}_\clubsuit$, and $(t'[r], t[w]) \in \mathsf{RW}_\clubsuit$ if $(t', t) \in \mathsf{WR}$, $(t', t) \in \mathsf{WW}$, and $(t', t) \in \mathsf{RW}$, respectively.

For instance, Figure 4 pictures the trace $\tau_\clubsuit$ for the LU trace τ given in Figure 3b. For traces τ of programs that contain singleton transactions, e.g., SB in Figure 3a, $\tau_\clubsuit$ coincides with τ.

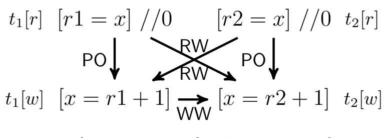


Fig. 4: A trace of the transformed LU program ($\mathsf{LU}_\clubsuit$).

Conversely, for a given trace $\tau_\clubsuit = (\rho_\clubsuit, \mathsf{PO}_\clubsuit, \mathsf{WR}_\clubsuit, \mathsf{WW}_\clubsuit, \mathsf{RW}_\clubsuit) \in \mathbb{T}\mathsf{r}_\mathsf{X}(\mathcal{P}_\clubsuit)$

of a program $\mathcal{P}_\clubsuit$ under a semantics X, we define the trace $\tau = (\rho, \mathsf{PO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$ where every two components $t[r]$ and $t[w]$ are merged into a transaction $t \in \tau$. The dependency relations are defined in a straightforward way, e.g., if $(t'[w], t[w]) \in \mathsf{WW}_\clubsuit$ then $(t', t) \in \mathsf{WW}$.

The following lemma shows that for any semantics $\mathsf{X} \in \{\mathtt{CC}, \mathtt{PC}, \mathtt{SI}\}$, if $\tau \in \mathbb{T}\mathsf{r}_\mathsf{X}(\mathcal{P})$ for a program $\mathcal{P}$, then $\tau_\clubsuit$ is a valid trace of $\mathcal{P}_\clubsuit$ under X, i.e., $\tau_\clubsuit \in \mathbb{T}\mathsf{r}_\mathsf{X}(\mathcal{P}_\clubsuit)$. Intuitively, this lemma shows that splitting transactions in a trace and defining dependency relations appropriately cannot introduce cycles in these relations and preserves the validity of the different consistency axioms.

The proof of this lemma relies on constructing a causal order $\mathsf{CO}_\clubsuit$ and an arbitration order $ARB_\clubsuit$ for the trace $\tau_\clubsuit$ starting from the analogous relations in τ. In the case of $\mathtt{CC}$, these are the smallest transitive relations such that:

- $\mathsf{PO}_\clubsuit \subseteq \mathsf{CO}_\clubsuit \subseteq ARB_\clubsuit$, and
- if $(t_1, t_2) \in \mathsf{CO}$ then $(t_1[w], t_2[r]) \in \mathsf{CO}_\clubsuit$, and if $(t_1, t_2) \in ARB$ then $(t_1[w], t_2[r]) \in ARB_\clubsuit$.

For $\mathtt{PC}$ and $\mathtt{SI}$, $\mathsf{CO}_\clubsuit$ must additionally satisfy: if $(t_1, t_2) \in ARB$, then $(t_1[w], t_2[w]) \in \mathsf{CO}_\clubsuit$. This is required in order to satisfy the axiom $\mathsf{AxPrefix}$, i.e., $ARB_\clubsuit; \mathsf{CO}_\clubsuit \subset \mathsf{CO}_\clubsuit$, when $(t_1[w], t_2[r]) \in ARB_\clubsuit$ and $(t_2[r], t_2[w]) \in \mathsf{CO}_\clubsuit$.

This construction ensures that $\mathsf{CO}_\clubsuit$ is a partial order and $ARB_\clubsuit$ is a total order because CO is a partial order and ARB is a total order. Also, based on the above rules, we have that: if $(t_1[w], t_2[r]) \in \mathsf{CO}_\clubsuit$ then $(t_1, t_2) \in \mathsf{CO}$, and similarly, if $(t_1[w], t_2[r]) \in ARB_\clubsuit$ then $(t_1, t_2) \in ARB$.

Lemma 1. *If $\tau \in \mathbb{T}r_\mathsf{X}(\mathcal{P})$, then $\tau_\clubsuit \in \mathbb{T}r_\mathsf{X}(\mathcal{P}_\clubsuit)$.*

Before presenting a strengthening of Lemma 1 when X is $\mathtt{CC}$, we give an important characterization of $\mathtt{CC}$ traces. This characterization is stated in terms of acyclicity properties.

Lemma 2. *τ is a trace under $\mathtt{CC}$ iff ARB_0^+ and $\mathsf{CO}_0^+; \mathsf{RW}$ are acyclic (ARB_0 and CO_0 are defined in Table 1).*

Next we show that a trace τ of a program $\mathcal{P}$ is $\mathtt{CC}$ iff the corresponding trace $\tau_\clubsuit$ of $\mathcal{P}_\clubsuit$ is $\mathtt{CC}$ as well. This result is based on the observation that cycles in ARB_0^+ or $\mathsf{CO}_0^+; \mathsf{RW}$ cannot be broken by splitting transactions.

Lemma 3. *A trace τ of $\mathcal{P}$ is $\mathtt{CC}$ iff the corresponding trace $\tau_\clubsuit$ of $\mathcal{P}_\clubsuit$ is $\mathtt{CC}$.*

The following lemma shows that a trace τ is $\mathtt{PC}$ iff the corresponding trace $\tau_\clubsuit$ is $\mathtt{SER}$. The if direction in the proof is based on constructing a causal order CO and an arbitration order ARB for the trace τ from the arbitration order $ARB_\clubsuit$ in $\tau_\clubsuit$ (since $\tau_\clubsuit$ is a trace under serializability $\mathsf{CO}_\clubsuit$ and $ARB_\clubsuit$ coincide). These are the smallest transitive relations such that:

- if $(t_1[w], t_2[r]) \in ARB_\clubsuit$ then $(t_1, t_2) \in \mathsf{CO}$,
- if $(t_1[w], t_2[w]) \in ARB_\clubsuit$ then $(t_1, t_2) \in ARB$[6].

[6] If $t_1[w]$ is empty (t_1 is read-only), then we set $(t_1, t_2) \in ARB$ if $(t_1[r], t_2[w]) \in \mathsf{CO}_\clubsuit$. If $t_2[w]$ is empty, then $(t_1, t_2) \in ARB$ if $(t_1[w], t_2[r]) \in \mathsf{CO}_\clubsuit$. If both $t_1[w]$ and $t_2[w]$ are empty, then $(t_1, t_2) \in ARB$ if $(t_1[r], t_2[r]) \in \mathsf{CO}_\clubsuit$.

The only-if direction is based on the fact that any cycle in the dependency relations of τ that is admitted under PC (characterized in Lemma 7) is "broken" by splitting transactions. Also, splitting transactions cannot introduce new cycles that do not originate in τ.

Lemma 4. *A trace τ is* PC *iff $\tau_{\clubsuit}$ is* SER

The lemmas above are used to prove Theorem 1 as follows:

Proof of Theorem 1: For the if direction, assume by contradiction that $\mathcal{P}$ is not robust against CC relative to PC. Then, there must exist a trace $\tau \in \mathbb{T}\mathsf{r}_{\mathtt{CC}}(\mathcal{P}) \setminus \mathbb{T}\mathsf{r}_{\mathtt{PC}}(\mathcal{P})$. Lemmas 3 and 4 imply that the corresponding trace $\tau_{\clubsuit}$ of $\mathcal{P}_{\clubsuit}$ is CC and not SER. Thus, $\mathcal{P}_{\clubsuit}$ is not robust against CC relative to SER. The only-if direction is proved similarly. □

Robustness against CC relative to SER has been shown to be reducible in polynomial time to the reachability problem under SER [10]. Given a program $\mathcal{P}$ and a control location ℓ, the reachability problem under SER asks whether there exists an execution of $\mathcal{P}$ under SER that reaches ℓ. Therefore, as a corollary of Theorem 1, we obtain the following:

Corollary 1. *Checking robustness against* CC *relative to* PC *is reducible to the reachability problem under* SER *in polynomial time.*

In the following we discuss the complexity of this problem in the case of finite-state programs (bounded data domain). The upper bound follows from Corollary 1 and standard results about the complexity of the reachability problem under sequential consistency, which extend to SER, with a bounded [35] or parametric number of processes [45]. For the lower bound, given an instance $(\mathcal{P}, \ell)$ of the reachability problem under sequential consistency, we construct a program $\mathcal{P}'$ where each statement s of $\mathcal{P}$ is executed in a different transaction that guards[7] the execution of s using a global lock (the lock can be implemented in our programming language as usual, e.g., using a busy wait loop for locking), and where reaching the location ℓ enables the execution of a "gadget" that corresponds to the SB program in Figure 3a. Executing each statement under a global lock ensures that every execution of $\mathcal{P}'$ under CC is serializable, and faithfully represents an execution of $\mathcal{P}$ under sequential consistency. Moreover, $\mathcal{P}$ reaches ℓ iff $\mathcal{P}'$ contains a robustness violation, which is due to the SB execution.

Corollary 2. *Checking robustness of a program with a fixed number of variables and bounded data domain against* CC *relative to* PC *is PSPACE-complete when the number of processes is bounded and EXPSPACE-complete, otherwise.*

5 Robustness Against PC Relative to SI

In this section, we show that checking robustness against PC relative to SI can be reduced in polynomial time to a reachability problem under the SER semantics. We reuse the program transformation from the previous section that allows to simulate PC behaviors on top of SER, and additionally, we provide a characterization of traces that distinguish the PC semantics from SI. We use this

[7] That is, the transaction is of the form [lock; s; unlock]

characterization to define an instrumentation (monitor) that is able to detect if a program under PC admits such traces.

We show that the happens-before cycles in a robustness violation (against PC relative to SI) must contain a WW dependency followed by a RW dependency, and they should not contain two successive RW dependencies. This follows from the fact that every happens-before cycle in a PC trace must contain either two successive RW dependencies, or a WW dependency followed by a RW dependency. Otherwise, the happens-before cycle would imply a cycle in the arbitration order. Then, any trace under PC where all its simple happens-before cycles contain two successive RW dependencies is possible under SI. For instance, the trace of the non-robust LU execution in Figure 3b contains WW dependency followed by a RW dependency and does not contain two successive RW dependencies which is disallowed SI, while the trace of the robust WS execution in Figure 3c contains two successive RW dependencies. As a first step, we prove the following theorem characterizing traces that are allowed under both PC and SI.

Theorem 2. *A program $\mathcal{P}$ is robust against* PC *relative to* SI *iff every happens-before cycle in a trace of $\mathcal{P}$ under* PC *contains two successive* RW *dependencies.*

Before giving the proof of the above theorem, we state several intermediate results that characterize cycles in PC or SI traces. First, we show that every PC trace in which all simple happens-before cycles contain two successive RW is also a SI trace.

Lemma 5. *If a trace τ is* PC *and all happens-before cycles in τ contain two successive* RW *dependencies, then τ is* SI*.*

The proof of Theorem 2 also relies on the following lemma that characterizes happens-before cycles permissible under SI.

Lemma 6. *[23, 13] If a trace τ is* SI*, then all its happens-before cycles must contain two successive* RW *dependencies.*

Proof of Theorem 2: For the only-if direction, if $\mathcal{P}$ is robust against PC relative to SI then every trace τ of $\mathcal{P}$ under PC is SI as well. Therefore, by Lemma 6, all cycles in τ contain two successive RW which concludes the proof of this direction. For the reverse, let τ be a trace of $\mathcal{P}$ under PC such that all its happens-before cycles contain two successive RW. Then, by Lemma 5, we have that τ is SI. Thus, every trace τ of $\mathcal{P}$ under PC is SI. $\square$

Next, we present an important lemma that characterizes happens before cycles possible under the PC semantics. This is a strengthening of a result in [13] which shows that all happens before cycles under PC must have two successive dependencies in $\{\mathsf{RW}, \mathsf{WW}\}$ and at least one RW. We show that the two successive dependencies cannot be RW followed WW, or two successive WW.

Lemma 7. *If a trace τ is* PC *then all happens-before cycles in τ must contain either two successive* RW *dependencies or a* WW *dependency followed by a* RW *dependency.*

Combining the results of Theorem 2 and Lemmas 4 and 7, we obtain the following characterization of traces which violate robustness against PC relative to SI.

Theorem 3. *A program $\mathcal{P}$ is not robust against* PC *relative to* SI *iff there exists a trace $\tau_\clubsuit$ of $\mathcal{P}_\clubsuit$ under* SER *such that the trace τ obtained by merging*[8] *read and write transactions in $\tau_\clubsuit$ contains a happens-before cycle that does not contain two successive* RW *dependencies, and it contains a* WW *dependency followed by a* RW *dependency.*

The results above enable a reduction from checking robustness against PC relative to SI to a reachability problem under the SER semantics. For a program $\mathcal{P}$, we define an instrumentation denoted by $[\![\mathcal{P}]\!]$, such that $\mathcal{P}$ is not robust against PC relative to SI iff $[\![\mathcal{P}]\!]$ violates an assertion under SER. The instrumentation consists in rewriting every transaction of $\mathcal{P}$ as shown in Figure 6.

$$\begin{tikzcd}
\alpha & t_\# \arrow[r, bend left, "\mathsf{RW}", "\beta"'] & t_0 \arrow[r, bend left, "\mathsf{HB}", "\gamma"'] & t \arrow[ll, bend left, "\mathsf{WW}"]
\end{tikzcd}$$

Fig. 5: Execution simulating a violation to robustness against PC relative to SI.

The instrumentation $[\![\mathcal{P}]\!]$ running under SER simulates the PC semantics of $\mathcal{P}$ using the same idea of decoupling the execution of the read part of a transaction from the write part. It violates an assertion when it simulates a PC trace containing a happens-before cycle as in Theorem 3. The execution corresponding to this trace has the shape given in Figure 5, where $t_\#$ is the transaction that occurs between the WW and the RW dependencies, and every transaction executed after $t_\#$ (this can be a full transaction in $\mathcal{P}$, or only the read or write part of a transaction in $\mathcal{P}$) is related by a happens-before path to $t_\#$ (otherwise, the execution of this transaction can be reordered to occur before $t_\#$). A transaction in $\mathcal{P}$ can have its read part included in α and the write part included in β or γ. Also, β and γ may contain transactions in $\mathcal{P}$ that executed only their read part. It is possible that $t_0 = t$, $\beta = \gamma = \epsilon$, and $\alpha = \epsilon$ (the LU program shown in Figure 3b is an example where this can happen). The instrumentation uses auxiliary variables to track happens-before dependencies, which are explained below.

The instrumentation executes (incomplete) transactions without affecting the auxiliary variables (without tracking happens-before dependencies) (lines 3 and 5) until a non-deterministically chosen point in time when it declares the current transaction as the candidate for $t_\#$ (line 9). Only one candidate for $t_\#$ can be chosen during the execution. This transaction executes only its reads and it chooses non-deterministically a variable that it could write as a witness for the WW dependency (see lines 16-22). The name of this variable is stored in a global variable `varW` (see the definition of $\mathcal{I}_\#($ x := e $)$). The writes are *not* applied on the shared memory. Intuitively, $t_\#$ should be thought as a transaction whose writes are delayed for later, after transaction t in Figure 5 executed. The instrumentation checks that $t_\#$ and t can be connected by some happens-before path that includes the RW and WW dependencies, and that does not contain two consecutive RW dependencies. If it is the case, it violates an assertion at the commit point of t. Since the write part of $t_\#$ is intuitively delayed to execute after t, the process executing $t_\#$ is disabled all along the execution (see the `assume false`).

[8] This transformation has been defined at the beginning of Section 4.

Transaction "**begin** ⟨read⟩* ⟨test⟩* ⟨write⟩* **commit**" is rewritten to:

```
 1 if ( !done# )
 2  if ( * )
 3   begin <read>* <test>* commit
 4   if ( !done# )
 5    begin <write>* commit
 6   else
 7    𝓘(begin) (𝓘(<write>))* 𝓘(commit)
 8  else
 9   begin (𝓘#(<read>))* <test>* (𝓘#(<write>))* 𝓘#(commit)
10   assume false;
11 else if ( * )
12  rdSet' := ∅;
13  wrSet' := ∅;
14  𝓘(begin) (𝓘(<read>))* <test>* 𝓘(commit)
15  𝓘(begin) (𝓘(<write>))* 𝓘(commit)
```

$\mathcal{I}_\#$(r := x):

```
16 r := x;
17 hbR['x'] := 0;
18 rdSet := rdSet ∪ { 'x' };
```

$\mathcal{I}_\#$(x := e):

```
19 if ( varW == ⊥ and * )
20   varW := 'x';
```

$\mathcal{I}_\#$(commit):

```
21 assume ( varW != ⊥ )
22 done# := true
```

$\mathcal{I}$(begin):

```
23 begin
24 hb := ⊥
25 if ( hbP != ⊥ and hbP < 2 )
26   hb := 0;
27 else if ( hbP = 2 )
28   hb := 2;
```

$\mathcal{I}$(commit):

```
29 assume ( hb != ⊥ )
30 assert ( hb == 2 or varW ∉ wrSet' );
31 if ( hbP == ⊥ or hbP > hb )
32   hbP = hb;
33 for each 'x' ∈ wrSet'
34   if ( hbW['x'] == ⊥ or hbW['x'] > hb )
35     hbW['x'] = hb;
36 for each 'x' ∈ rdSet'
37   if ( hbR['x'] == ⊥ or hbR['x'] > hb )
38     hbR['x'] = hb;
39 rdSet := rdSet ∪ rdSet';
40 wrSet := wrSet ∪ wrSet';
41 commit
```

$\mathcal{I}$(r := x):

```
42 r := x;
43 rdSet' := rdSet' ∪ { 'x' };
44 if ( 'x' ∈ wrSet )
45   if ( hbW['x'] != 2 )
46     hb := 0
47   else if ( hb == ⊥ )
48     hb := hbW['x']
```

$\mathcal{I}$(x := e):

```
49 x := e;
50 wrSet' := wrSet' ∪ { 'x' };
51 if ( 'x' ∈ wrSet )
52   if ( hbW['x'] != 2 )
53     hb := 0
54   else if ( hb == ⊥ )
55     hb := hbW['x']
56 if ( 'x' ∈ rdSet )
57   if ( hb = ⊥ or hb > hbR['x'] + 1 )
58     hb := min(hbR['x'] + 1,2)
```

Fig. 6: A program instrumentation for checking robustness against PC relative to SI. The auxiliary variables used by the instrumentation are shared variables, except for `hbP`, `rdSet'`, and `wrSet'`, which are process-local variables, and they are initially set to ⊥. This instrumentation uses program constructs which can be defined as syntactic sugar from the syntax presented in Section 3, e.g., if-then-else statements (outside transactions).

After choosing the candidate for $t_\#$, the instrumentation uses the auxiliary variables for tracking happens-before dependencies. Therefore, `rdSet` and `wrSet` record variables read and written, respectively, by transactions that are connected by a happens-before path to $t_\#$ (in a trace of $\mathcal{P}$). This is ensured by the assume at line 29. During the execution, the variables read or written by a transaction[9] that writes a variable in `rdSet` (see line 56), or reads or writes a variable in `wrSet` (see lines 44 and 51), will be added to these sets (see lines 39

[9] These are stored in the local variables `rdSet'` and `wrSet'` while the transaction is running.

and 40). Since the variables that $t_{\#}$ writes in $\mathcal{P}$ are not recorded in `wrSet`, these happens-before paths must necessarily start with a RW dependency (from $t_{\#}$). When the assertion fails (line 30), the condition `varW` $\in$ `wrSet`' ensures that the current transaction has a WW dependency towards the write part of $t_{\#}$ (the current transaction plays the role of t in Figure 5).

The rest of the instrumentation checks that there exists a happens-before path from $t_{\#}$ to t that does not include two consecutive RW dependencies, called a $\texttt{SI}_{\neg}$ path. This check is based on the auxiliary variables whose name is prefixed by `hb` and which take values in the domain $\{\bot, 0, 1, 2\}$ ($\bot$ represents the initial value). Therefore,

– `hbR['x']` (resp., `hbW['x']`) is 0 iff there exists a transaction t' that reads `x` (resp., writes to `x`), such that there exists a $\texttt{SI}_{\neg}$ path from $t_{\#}$ to t' that ends with a dependency which is *not* RW,
– `hbR['x']` (resp., `hbW['x']`) is 1 iff there exists a transaction t' that reads `x` (resp., writes to `x`) that is connected to $t_{\#}$ by a $\texttt{SI}_{\neg}$ path, and *every* $\texttt{SI}_{\neg}$ path from $t_{\#}$ to a transaction t'' that reads `x` (resp., writes to `x`) ends with an RW dependency,
– `hbR['x']` (resp., `hbW['x']`) is 2 iff there exists no $\texttt{SI}_{\neg}$ path from $t_{\#}$ to a transaction t' that reads `x` (resp., writes to `x`).

The local variable `hbP` has the same interpretation, except that t' and t'' are instantiated over transactions in the same process (that already executed) instead of transactions that read or write a certain variable. Similarly, the variable `hb` is a particular case where t' and t'' are instantiated to the current transaction. The violation of the assertion at line 30 implies that `hb` is 0 or 1, which means that there exists a $\texttt{SI}_{\neg}$ path from $t_{\#}$ to t.

During each transaction that executes after $t_{\#}$, the variable `hb` characterizing happens-before paths that end in this transaction is updated every time a new happens-before dependency is witnessed (using the values of the other variables). For instance, when witnessing a WR dependency (line 44), if there exists a $\texttt{SI}_{\neg}$ path to a transaction that writes to `x`, then the path that continues with the WR dependency towards the current transaction is also a $\texttt{SI}_{\neg}$ path, and the last dependency of this path is not RW. Therefore, `hb` is set to 0 (see line 46). Otherwise, if every path to a transaction that writes to `x` is not a $\texttt{SI}_{\neg}$ path, then every path that continues to the current transaction (by taking the WR dependency) remains a non $\texttt{SI}_{\neg}$ path, and `hb` is set to the value of `hbW['x']`, which is 2 in this case (see line 48). Before ending a transaction, the value of `hb` can be used to modify the `hbR`, `hbW`, and `hbP` variables, but only if those variables contain bigger values (see lines 31–38).

The correctness of the instrumentation is stated in the following theorem.

Theorem 4. *A program $\mathcal{P}$ is robust against* `PC` *relative to* `SI` *iff the instrumentation in Figure 6 does not violate an assertion when executed under* `SER`.

Theorem 4 implies the following complexity result for finite-state programs. The lower bound is proved similarly to the case `CC` vs `PC`.

Corollary 3. *Checking robustness of a program with a fixed number of variables and bounded data domain against* PC *relative to* SI *is PSPACE-complete when the number of processes is bounded and EXPSPACE-complete, otherwise.*

Checking robustness against CC relative to SI can be also shown to be reducible (in polynomial time) to a reachability problem under SER by combining the results of checking robustness against CC relative to PC and PC relative to SI.

Theorem 5. *A program $\mathcal{P}$ is robust against* CC *relative to* SI *iff $\mathcal{P}$ is robust against* CC *relative to* PC *and $\mathcal{P}$ is robust against* PC *relative to* SI*.*

Remark 1. Our reductions of robustness checking to reachability apply to an extension of our programming language where the number of processes is unbounded and each process can execute an arbitrary number of times a statically known set of transactions. This holds because the instrumentation in Figure 6 and the one in [10] (for the case CC vs. SER) consist in adding a set of instructions that manipulate a fixed set of process-local or shared variables, which do not store process or transaction identifiers. These reductions extend also to SQL queries that access unbounded size tables. Rows in a table can be interpreted as memory locations (identified by primary keys in unbounded domains, e.g., integers), and SQL queries can be interpreted as instructions that read/write a set of locations in one shot. These possibly unbounded sets of locations can be represented symbolically using the conditions in the SQL queries (e.g., the condition in the WHERE part of a SELECT). The instrumentation in Figure 6 needs to be adapted so that read and write sets are updated by adding sets of locations for a given instruction (represented symbolically as mentioned above).

6 Proving Robustness Using Commutativity Dependency Graphs

We describe an approximated technique for proving robustness, which leverages the concept of left/right mover in Lipton's reduction theory [39]. This technique reasons on the *commutativity dependency graph* [9] associated to the transformation $\mathcal{P}_\clubsuit$ of an input program $\mathcal{P}$ that allows to simulate the PC semantics under serializability (we use a slight variation of the original definition of this class of graphs). We characterize robustness against CC relative to PC and PC relative to SI in terms of certain properties that (simple) cycles in this graph must satisfy.

We recall the concept of movers and the definition of commutativity dependency graphs. Given a program $\mathcal{P}$ and a trace $\tau = t_1 \cdot \ldots \cdot t_n \in \mathbb{T}\mathsf{r}_{\mathtt{SER}}(\mathcal{P})$ of $\mathcal{P}$ under serializability, we say that $t_i \in \tau$ *moves right (resp., left)* in τ if $t_1 \cdot \ldots \cdot t_{i-1} \cdot t_{i+1} \cdot t_i \cdot t_{i+2} \cdot \ldots \cdot t_n$ (resp., $t_1 \cdot \ldots \cdot t_{i-2} \cdot t_i \cdot t_{i-1} \cdot t_{i+1} \cdot \ldots \cdot t_n$) is also a valid execution of $\mathcal{P}$, t_i and t_{i+1} (resp., t_{i-1}) are executed by distinct processes, and both traces reach the same end state. A transaction $t \in \mathsf{Tr}(\mathcal{P})$ is not a right (resp., left) mover iff there exists a trace $\tau \in \mathbb{T}\mathsf{r}_{\mathtt{SER}}(\mathcal{P})$ such that $t \in \tau$ and t doesn't move right (resp., left) in τ. Thus, when a transaction t is *not* a right mover then there must exist another transaction $t' \in \tau$ which caused t to not be permutable to the right (while preserving the end state). Since t and t' do not commute, then this must be because of either a write-read, write-write, or a read-write dependency relation between the two transactions. We say that t is not a right mover because of t' and a dependency relation 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 (t' reads some value written by t). We define the relations $\mathsf{M_{WW}}$ and $\mathsf{M_{RW}}$ corresponding to write-write and read-write dependencies in a similar way. We call $\mathsf{M_{WR}}$, $\mathsf{M_{WW}}$, and $\mathsf{M_{RW}}$, *non-mover* relations.

The *commutativity dependency graph* of a program $\mathcal{P}$ is a graph where vertices represent transactions in $\mathcal{P}$. Two vertices are linked by a program order edge if the two transactions are executed by the same process. The other edges in this graph represent the "non-mover" relations $\mathsf{M_{WR}}$, $\mathsf{M_{WW}}$, and $\mathsf{M_{RW}}$. Two vertices that represent the two components $t[w]$ and $t[r]$ of the same transaction t (already linked by PO edge) are also linked by an undirected edge labeled by STO (same-transaction relation).

Our results about the robustness of a program $\mathcal{P}$ are stated over a slight variation of the commutativity dependency graph of $\mathcal{P}_♣$ (where a transaction is either read-only or write-only). This graph contains additional undirected edges that link every pair of transactions $t[r]$ and $t[w]$ of $\mathcal{P}_♣$ that were originally components of the same transaction t in $\mathcal{P}$. Given such a commutativity dependency graph, the robustness of $\mathcal{P}$ is implied by the absence of cycles of specific shapes. These cycles can be seen as an abstraction of potential robustness violations for the respective semantics (see Theorem 6 and Theorem 7). Figure 7 pictures the commutativity dependency graph for the MP program. Since every transaction in MP is singleton, the two programs MP and $\mathsf{MP}_♣$ coincide.

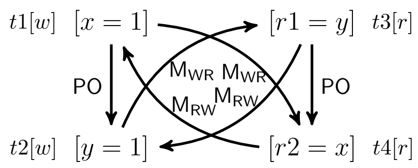


Fig. 7: The commutativity dependency graph of the $\mathsf{MP}_♣$ program.

Using the characterization of robustness violations against CC relative to SER from [10] and the reduction in Theorem 1, we obtain the following result concerning the robustness against CC relative to PC.

Theorem 6. *Given a program $\mathcal{P}$, if the commutativity dependency graph of the program $\mathcal{P}_♣$ does not contain a simple cycle formed by $t_1 \cdots t_i \cdots t_n$ such that:*

- $(t_n, t_1) \in \mathsf{M_{RW}}$*;*
- $(t_j, t_{j+1}) \in (\mathsf{PO} \cup \mathsf{WR})^*$*, for* $j \in [1, i-1]$*;*
- $(t_i, t_{i+1}) \in (\mathsf{M_{RW}} \cup \mathsf{M_{WW}})$*;*
- $(t_j, t_{j+1}) \in (\mathsf{M_{RW}} \cup \mathsf{M_{WW}} \cup \mathsf{M_{WR}} \cup \mathsf{PO})$*, for* $j \in [i+1, n-1]$*.*

then $\mathcal{P}$ is robust against CC *relative to* PC*.*

Next we give the characterization of commutativity dependency graphs required for proving robustness against PC relative to SI.

Theorem 7. *Given a program $\mathcal{P}$, if the commutativity dependency graph of the program $\mathcal{P}_{\clubsuit}$ does not contain a simple cycle formed by $t_1 \cdots t_n$ such that:*

- $(t_n, t_1) \in \mathsf{M_{WW}}$, $(t_1, t_2) \in \mathsf{STO}$, *and* $(t_2, t_3) \in \mathsf{M_{RW}}$*;*
- $(t_j, t_{j+1}) \in (\mathsf{M_{RW}} \cup \mathsf{M_{WW}} \cup \mathsf{M_{WR}} \cup \mathsf{PO} \cup \mathsf{STO})^*$*, for* $j \in [3, n-1]$*;*
- $\forall\ j \in [2, n-2]$.
  - *if* $(t_j, t_{j+1}) \in \mathsf{M_{RW}}$ *then* $(t_{j+1}, t_{j+2}) \in (\mathsf{M_{WR}} \cup \mathsf{PO} \cup \mathsf{M_{WW}})$*;*
  - *if* $(t_{j+1}, t_{j+2}) \in \mathsf{M_{RW}}$ *then* $(t_j, t_{j+1}) \in (\mathsf{M_{WR}} \cup \mathsf{PO})$.
- $\forall\ j \in [3, n-3]$. *if* $(t_{j+1}, t_{j+2}) \in \mathsf{STO}$ *and* $(t_{j+2}, t_{j+3}) \in \mathsf{M_{RW}}$ *then* $(t_j, t_{j+1}) \in \mathsf{M_{WW}}$.

then $\mathcal{P}$ is robust against `PC` *relative to* `SI`*.*

In Figure 7, we have three simple cycles in the graph:

- $(t1[w], t4[r]) \in \mathsf{M_{WR}}$ and $(t4[r], t1[w]) \in \mathsf{M_{RW}}$,
- $(t2[w], t3[r]) \in \mathsf{M_{WR}}$ and $(t3[r], t2[w]) \in \mathsf{M_{RW}}$,
- $(t1[w], t2[w]) \in \mathsf{PO}$, $(t2[w], t3[r]) \in \mathsf{M_{WR}}$, $(t3[r], t4[r]) \in \mathsf{PO}$, and $(t4[r], t1[w]) \in \mathsf{M_{RW}}$.

Notice that none of the cycles satisfies the properties in Theorems 6 and 7. Therefore, MP is robust against `CC` relative to `PC` and against `PC` relative to `SI`.

Remark 2. For programs that contain an unbounded number of processes, an unbounded number of instantiations of a fixed number of process "templates", or unbounded loops with bodies that contain entire transactions, a sound robustness check consists in applying Theorem 6 and Theorem 7 to (bounded) programs that contain two copies of each process template, and where each loop is unfolded exactly two times. This holds because the mover relations are "static", they do not depend on the context in which the transactions execute, and each cycle requiring more than two process instances or more than two loop iterations can be short-circuited to a cycle that exists also in the bounded program. Every outgoing edge from a third instance/iteration can also be taken from the second instance/iteration. Two copies/iterations are necessary in order to discover cycles between instances of the same transaction (the cycles in Theorem 6 and Theorem 7 are simple and cannot contain the same transaction twice). These results extend easily to SQL queries as well because the notion of mover is independent of particular classes of programs or instructions.

7 Experimental Evaluation

We evaluated our approach for checking robustness on 7 applications extracted from the literature on databases and distributed systems, and an application Betting designed by ourselves. Two applications were extracted from the OLTP-Bench benchmark [30]: a vote recording application (Vote) and a consumer review application (Epinions). Three applications were obtained from Github projects (used also in [9, 19]): a distributed lock application for the Cassandra database (CassandraLock [24]), an application for recording trade activities

(SimpleCurrencyExchange [48]), and a micro social media application (Twitter [49]). The last two applications are a movie ticketing application (FusionTicket) [34], and a user subscription application inspired by the Twitter application (Subscription). Each application consists of a set of SQL transactions that can be called an arbitrary number of times from an arbitrary number of processes. For instance, Subscription provides an AddUser transaction for adding a new user with a given username and password, and a RemoveUser transaction for removing an existing user. (The examples in Figure 1 are particular variations of FusionTicket, Twitter, and Betting.) We considered five variations of the robustness problem: the three robustness problems we studied in this paper along with robustness against SI relative to SER and against CC relative to SER. The artifacts are available in a GitHub repository [31].

Table 2: Results of the experiments. The columns titled X-Y stand for the result of applications robustness against X relative to Y.

Application	Transactions	Robustness				
		CC-PC	PC-SI	CC-SI	SI-SER	CC-SER
Betting	2	yes	yes	yes	yes	yes
CassandraLock	3	yes	yes	yes	yes	yes
Epinions	8	no	yes	no	yes	no
FusionTicket	3	no	no	no	yes	no
SimpleCurrencyExchange	4	yes	yes	yes	yes	yes
Subscription	2	yes	no	no	yes	no
Twitter	3	no	no	no	yes	no
Vote	1	yes	yes	yes	no	no

In the first part of the experiments, we check for robustness violations in bounded-size executions of a given application. For each application, we have constructed a client program with a fixed number of processes (2) and a fixed number of transactions of the corresponding application (at most 2 transactions per process). For each program and pair of consistency models, we check for robustness violations using the reductions to reachability under SER presented in Section 4 and Section 5 in the case of pairs of weak consistency models, and the reductions in [9, 10] when checking for robustness relative to SER.

We check for reachability (assertion violations) using the Boogie program verifier [8]. We model tables as unbounded maps in Boogie and SQL queries as first-order formulas over these maps (that may contain existential or universal quantifiers). To model the uniqueness of primary keys we use Boogie linear types.

Table 2 reports the results of this experiment (cells filled with "no")[10]. Five applications are not robust against at least one of the semantics relative to some other stronger semantics. The runtimes (wall-clock times) for the robustness checks are all under one second, and the memory consumption is around 50 Megabytes. Concerning scalability, the reductions to reachability presented in Section 4 and Section 5 show that checking robustness is as hard as checking

[10] The Twitter client in Table 2, which is not PC vs CC robust, is different from the one described in Section 2. This client program consists of two processes, each executing FollowUser and AddTweet.

reachability (the size of the instrumented program is only linear in the size of the original program). Therefore, checking robustness will also suffer from the classic state explosion problem when increasing the number of processes. On the other hand, increasing the number of transactions in a process does not seem to introduce a large overhead. Increasing the number of transactions per process in the clients of Epinions, FusionTicket, and SimpleCurrencyExchange from 2 to 5 introduces a running time overhead of at most 25%.

All the robustness violations we report correspond to violations of the intended specifications. For instance: (1) the robustness violation of Epinions against CC relative to PC allows two users to update their ratings for a given product and then when each user queries the overall rating of this product they do not observe the latest rating that was given by the other user, (2) the robustness violation of Subscription against PC relative to SI allows two users to register new accounts with the same identifier, and (3) the robustness violation of Vote against SI relative to SER allows the same user to vote twice. The specification violation in Twitter was reported in [19]. However, it was reported as violation of a different robustness property (CC relative to SER) while our work shows that the violation persists when replacing a weak consistency model (e.g., SI) with a weaker one (e.g. CC). This implies that this specification violation is not present under SI (since it appears in the difference between CC and SI behaviors), which cannot be deduced from previous work.

In the second part of the experiments, we used the technique described in Section 6, based on commutativity dependency graphs, to prove robustness. For each application (set of transactions) we considered a program that for each ordered pair of (possibly identical) transactions in the application, contains two processes executing that pair of transactions. Following Remark 2, the robustness of such a program implies the robustness of a *most general client* of the application that executes each transaction an arbitrary number of times and from an arbitrary number of processes. We focused on the cases where we could not find robustness violations in the first part. To build the "non-mover" relations M_{WR}, M_{WW}, and M_{RW} for the commutativity dependency graph, we use the left/right mover check provided by the CIVL verifier [33]. The results are reported in Table 2, the cells filled with "yes". We showed that the three applications Betting, CassandraLock and SimpleCurrencyExchange are robust against any semantics relative to some other stronger semantics. As mentioned earlier, all these robustness results are established for arbitrarily large executions and clients with an arbitrary number of processes. For instance, the robustness of SimpleCurrencyExchange ensures that when the exchange market owner observes a trade registered by a user, they observe also all the other trades that were done by this user in the past.

In conclusion, our experiments show that the robustness checking techniques we present are effective in proving or disproving robustness of concrete applications. Moreover, it shows that the robustness property for different combinations of consistency models is a relevant design principle, that can help in choosing the right consistency model for realistic applications, i.e., navigating the tradeoff between consistency and performance (in general, weakening the consistency leads to better performance).

8 Related Work

The consistency models in this paper were studied in several recent works [21, 20, 25, 43, 16, 44, 14]. Most of them focused on their operational and axiomatic formalizations. The formal definitions we use in this paper are based on those given in [25, 16]. Biswas and Enea [14] shows that checking whether an execution is CC is polynomial time while checking whether it is PC or SI is NP-complete.

The robustness problem we study in this paper has been investigated in the context of weak memory models, but only relative to sequential consistency, against Release/Aquire (RA), TSO and Power [36, 17, 15, 29]. Checking robustness against CC and SI relative to SER has been investigated in [9, 10]. In this work, we study the robustness problem between two weak consistency models, which poses different non-trivial challenges. In particular, previous work proposed reductions to reachability under sequential consistency (or SER) that relied on a concept of minimal robustness violations (w.r.t. an operational semantics), which does not apply in our case. The relationship between PC and SER is similar in spirit to the one given by Biswas and Enea [14] in the context of checking whether an execution is PC. However, that relationship was proven in the context of a "weaker" notion of trace (containing only program order and read-from), and it does not extend to our notion of trace. For instance, that result does not imply preserving WW dependencies which is crucial in our case.

Some works describe various over- or under-approximate analyses for checking robustness relative to SER. The works in [13, 18, 19, 26, 40] propose static analysis techniques based on computing an abstraction of the set of computations, which is used for proving robustness. In particular, [19, 40] encode program executions under the weak consistency model using FOL formulas to describe the dependency relations between actions in the executions. These approaches may return false alarms due to the abstractions they consider in their encoding. Note that in this paper, we prove a strengthening of the results of [13] with regard to the shape of happens before cycles allowed under PC.

An alternative to *trace-based* robustness, is *state-based* robustness which requires that a program is robust if the sets of reachable states under two semantics coincide. While state-robustness is the necessary and sufficient concept for preserving state-invariants, its verification, which amounts in computing the set of reachable states under the weak semantics models is in general a hard problem. The decidability and the complexity of this problem has been investigated in the context of relaxed memory models such as TSO and Power, and it has been shown that it is either decidable but highly complex (non-primitive recursive), or undecidable [5, 6]. Automatic procedures for approximate reachability/invariant checking have been proposed using either abstractions or bounded analyses, e.g., [7, 4, 28, 1]. Proof methods have also been developed for verifying invariants in the context of weakly consistent models such as [37, 32, 41, 3]. These methods, however, do not provide decision procedures.

References

1. Abdulla, P.A., Atig, M.F., Bouajjani, A., Ngo, T.P.: Context-bounded analysis for POWER. In: Legay, A., Margaria, T. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10206, pp. 56–74 (2017). https://doi.org/10.1007/978-3-662-54580-5_4, https://doi.org/10.1007/978-3-662-54580-5_4
2. Adya, A.: Weak consistency: A generalized theory and optimistic implementations for distributed transactions. Ph.D. thesis (1999)
3. Alglave, J., Cousot, P.: Ogre and pythia: an invariance proof method for weak consistency models. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 3–18. ACM (2017), http://dl.acm.org/citation.cfm?id=3009883
4. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 141–157. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_9, https://doi.org/10.1007/978-3-642-39799-8_9
5. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In: Hermenegildo, M.V., Palsberg, J. (eds.) Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010. pp. 7–18. ACM (2010). https://doi.org/10.1145/1706299.1706303, https://doi.org/10.1145/1706299.1706303
6. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: What's decidable about weak memory models? In: Seidl, H. (ed.) Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7211, pp. 26–46. Springer (2012). https://doi.org/10.1007/978-3-642-28869-2_2, https://doi.org/10.1007/978-3-642-28869-2_2
7. Atig, M.F., Bouajjani, A., Parlato, G.: Getting rid of store-buffers in TSO analysis. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 99–115. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_9, https://doi.org/10.1007/978-3-642-22110-1_9
8. 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). https://doi.org/10.1007/11804192_17, https://doi.org/10.1007/11804192_17

9. Beillahi, S.M., Bouajjani, A., Enea, C.: Checking robustness against snapshot isolation. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II. Lecture Notes in Computer Science, vol. 11562, pp. 286–304. Springer (2019). https://doi.org/10.1007/978-3-030-25543-5_17, https://doi.org/10.1007/978-3-030-25543-5_17
10. Beillahi, S.M., Bouajjani, A., Enea, C.: Robustness against transactional causal consistency. In: Fokkink, W.J., van Glabbeek, R. (eds.) 30th International Conference on Concurrency Theory, CONCUR 2019, August 27-30, 2019, Amsterdam, the Netherlands. LIPIcs, vol. 140, pp. 30:1–30:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.CONCUR.2019.30, https://doi.org/10.4230/LIPIcs.CONCUR.2019.30
11. Beillahi, S.M., Bouajjani, A., Enea, C.: Checking robustness between weak transactional consistency models. CoRR **abs/2101.09032** (2021), http://arxiv.org/abs/2101.09032
12. 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). https://doi.org/10.1145/223784.223785, https://doi.org/10.1145/223784.223785
13. 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 für Informatik (2016). https://doi.org/10.4230/LIPIcs.CONCUR.2016.7, https://doi.org/10.4230/LIPIcs.CONCUR.2016.7
14. Biswas, R., Enea, C.: On the complexity of checking transactional consistency. Proc. ACM Program. Lang. **3**(OOPSLA), 165:1–165:28 (2019). https://doi.org/10.1145/3360591, https://doi.org/10.1145/3360591
15. 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). https://doi.org/10.1007/978-3-642-37036-6_29, https://doi.org/10.1007/978-3-642-37036-6_29
16. Bouajjani, A., Enea, C., Guerraoui, R., Hamza, J.: On verifying causal consistency. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 626–638. ACM (2017), http://dl.acm.org/citation.cfm?id=3009888
17. 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). https://doi.org/10.1007/978-3-642-22012-8_34, https://doi.org/10.1007/978-3-642-22012-8_34
18. Brutschy, L., Dimitrov, D., Müller, P., Vechev, M.T.: Serializability for eventual consistency: criterion, analysis, and applications. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Pro-

gramming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 458–472. ACM (2017), http://dl.acm.org/citation.cfm?id=3009895

19. 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). https://doi.org/10.1145/3192366.3192415, https://doi.org/10.1145/3192366.3192415
20. Burckhardt, S.: Principles of eventual consistency. Found. Trends Program. Lang. **1**(1-2), 1–150 (2014). https://doi.org/10.1561/2500000011, https://doi.org/10.1561/2500000011
21. Burckhardt, S., Gotsman, A., Yang, H., Zawirski, M.: Replicated data types: specification, verification, optimality. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014. pp. 271–284. ACM (2014). https://doi.org/10.1145/2535838.2535848, https://doi.org/10.1145/2535838.2535848
22. Burckhardt, S., Leijen, D., Protzenko, J., Fähndrich, M.: Global sequence protocol: A robust abstraction for replicated shared state. In: Boyland, J.T. (ed.) 29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic. LIPIcs, vol. 37, pp. 568–590. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2015). https://doi.org/10.4230/LIPIcs.ECOOP.2015.568, https://doi.org/10.4230/LIPIcs.ECOOP.2015.568
23. Cahill, M.J., Röhm, U., Fekete, A.D.: Serializable isolation for snapshot databases. ACM Trans. Database Syst. **34**(4), 20:1–20:42 (2009). https://doi.org/10.1145/1620585.1620587, https://doi.org/10.1145/1620585.1620587
24. Cassandra-lock: https://github.com/dekses/cassandra-lock
25. Cerone, A., Bernardi, G., Gotsman, A.: A framework for transactional consistency models with atomic visibility. In: Aceto, L., de Frutos-Escrig, D. (eds.) 26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015. LIPIcs, vol. 42, pp. 58–71. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2015). https://doi.org/10.4230/LIPIcs.CONCUR.2015.58, https://doi.org/10.4230/LIPIcs.CONCUR.2015.58
26. Cerone, A., Gotsman, A.: Analysing snapshot isolation. J. ACM **65**(2), 11:1–11:41 (2018). https://doi.org/10.1145/3152396, https://doi.org/10.1145/3152396
27. Cerone, A., Gotsman, A., Yang, H.: Algebraic laws for weak consistency. In: Meyer, R., Nestmann, U. (eds.) 28th International Conference on Concurrency Theory, CONCUR 2017, September 5-8, 2017, Berlin, Germany. LIPIcs, vol. 85, pp. 26:1–26:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPIcs.CONCUR.2017.26, https://doi.org/10.4230/LIPIcs.CONCUR.2017.26
28. Dan, A.M., Meshman, Y., Vechev, M.T., Yahav, E.: Effective abstractions for verification under relaxed memory models. Comput. Lang. Syst. Struct. **47**, 62–76 (2017). https://doi.org/10.1016/j.cl.2016.02.003, https://doi.org/10.1016/j.cl.2016.02.003
29. 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). https://doi.org/10.1007/978-3-662-43951-7_14, https://doi.org/10.1007/978-3-662-43951-7_14
30. Difallah, D.E., Pavlo, A., Curino, C., Cudré-Mauroux, P.: Oltp-bench: An extensible testbed for benchmarking relational databases. Proc. VLDB Endow. **7**(4), 277–288 (2013). https://doi.org/10.14778/2732240.2732246, http://www.vldb.org/pvldb/vol7/p277-difallah.pdf
31. Experiments: https://github.com/relative-robustness/artifact
32. 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). https://doi.org/10.1145/2837614.2837625, https://doi.org/10.1145/2837614.2837625
33. 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). https://doi.org/10.1007/978-3-319-21668-3_26, https://doi.org/10.1007/978-3-319-21668-3_26
34. 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). https://doi.org/10.1145/2987550.2987559, https://doi.org/10.1145/2987550.2987559
35. 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). https://doi.org/10.1109/SFCS.1977.16, https://doi.org/10.1109/SFCS.1977.16
36. Lahav, O., Margalit, R.: Robustness against release/acquire semantics. In: McKinley, K.S., Fisher, K. (eds.) Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019. pp. 126–141. ACM (2019). https://doi.org/10.1145/3314221.3314604, https://doi.org/10.1145/3314221.3314604
37. Lahav, O., Vafeiadis, V.: Owicki-gries reasoning for weak memory models. In: Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B. (eds.) Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9135, pp. 311–323. Springer (2015). https://doi.org/10.1007/978-3-662-47666-6_25, https://doi.org/10.1007/978-3-662-47666-6_25
38. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978). https://doi.org/10.1145/359545.359563, https://doi.org/10.1145/359545.359563
39. Lipton, R.J.: Reduction: A method of proving properties of parallel programs. Commun. ACM **18**(12), 717–721 (1975). https://doi.org/10.1145/361227.361234, https://doi.org/10.1145/361227.361234
40. 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 für Informatik (2018). https://doi.org/10.4230/LIPIcs.CONCUR.2018.41, https://doi.org/10.4230/LIPIcs.CONCUR.2018.41

41. Najafzadeh, M., Gotsman, A., Yang, H., Ferreira, C., Shapiro, M.: The CISE tool: proving weakly-consistent applications correct. In: Alvaro, P., Bessani, A. (eds.) Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2016, London, United Kingdom, April 18, 2016. pp. 2:1–2:3. ACM (2016). https://doi.org/10.1145/2911151.2911160, https://doi.org/10.1145/2911151.2911160
42. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26**(4), 631–653 (1979). https://doi.org/10.1145/322154.322158, https://doi.org/10.1145/322154.322158
43. Perrin, M., Mostéfaoui, A., Jard, C.: Causal consistency: beyond memory. In: Asenjo, R., Harris, T. (eds.) Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016. pp. 26:1–26:12. ACM (2016). https://doi.org/10.1145/2851141.2851170, https://doi.org/10.1145/2851141.2851170
44. Raad, A., Lahav, O., Vafeiadis, V.: On the semantics of snapshot isolation. In: Enea, C., Piskac, R. (eds.) Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11388, pp. 1–23. Springer (2019). https://doi.org/10.1007/978-3-030-11245-5_1, https://doi.org/10.1007/978-3-030-11245-5_1
45. Rackoff, C.: The covering and boundedness problems for vector addition systems. Theor. Comput. Sci. **6**, 223–231 (1978). https://doi.org/10.1016/0304-3975(78)90036-1, https://doi.org/10.1016/0304-3975(78)90036-1
46. Shapiro, M., Ardekani, M.S., Petri, G.: Consistency in 3d. 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. 3:1–3:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/LIPIcs.CONCUR.2016.3, https://doi.org/10.4230/LIPIcs.CONCUR.2016.3
47. 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). https://doi.org/10.1145/42190.42277, https://doi.org/10.1145/42190.42277
48. Trade: https://github.com/Haiyan2/Trade
49. Twitter: https://github.com/edmundophie/cassandra-twitter