Verifying Weakly Consistent Transactional Programs using Symbolic Execution

Burcu Kulahcioglu Ozkan

Max Planck Institute for Software Systems (MPI-SWS)

Abstract. We present a method for verifying whether all executions of a set of transactions satisfy a given invariant when run on weakly consistent systems. Existing approaches check that all executions under weak consistency are equivalent to some serial execution of the transactions, and separately that the serial executions satisfy the invariant. While sound, this can be overly strict. Programs running on systems with weak guarantees are usually designed to tolerate some anomalies w.r.t. the serial semantics and yet maintain some expected program invariants even on executions that are not serializable. In contrast, our technique does not restrict possible executions to be serializable, but directly checks whether given program properties hold w.r.t. all executions allowed under varying consistency models.
Our approach uses symbolic execution techniques and satisfiability checkers. We summarize the effects of transactions using symbolic execution and build a satisfiability formula that precisely captures all possible valuations of the data variables under a given consistency model. Then, we check whether the program invariants hold on the resulting symbolic set of behaviors. Our encoding is parameterized over the underlying consistency specification. Hence, the programmer can check the correctness of a program under several consistency models—eventual consistency, causal consistency, (parallel) snapshot isolation, serializability— and identify the level of consistency needed to satisfy the application-level invariants.

Keywords: Weak consistency, Transactions, Symbolic execution, Satisfiability

1 Introduction

Large-scale distributed systems rely on replicated databases that maintain data across a large number of nodes, potentially over a wide geographical span. Clients of the system can perform transactions at any node; the database is responsible to synchronize the data across the many nodes and maintain "consistency." Traditionally, consistency implied that the database was serializable [8]: the result of concurrently executing a set of transactions should be equivalent to executing the transactions serially in some order. Unfortunately, the synchronization cost of maintaining consistency is high; moreover, the CAP theorem [9] states that a distributed system cannot simultaneously guarantee consistency, availability, and partition tolerance. Thus, many modern systems sacrifice serializability in favor of weaker guarantees which allow executions that cannot be explained by any serial execution. A generic weaker guarantee is *eventual consistency* [31, 13], which states that all replicas reach a consistent state if no more user updates arrive to the data centers.

Generally, eventual consistency guarantee is too weak by itself to satisfy the specifications of many applications: indeed, user updates never stop arriving in these systems. Hence, systems provide additional consistency guarantees which pose some restrictions on executions and specify which subset of anomalous (non-serializable) behaviors are allowed by a system, and which are not. Such weak consistency models include causal consistency [27], prefix consistency [14], parallel snapshot isolation [32], and snapshot isolation [6].

While weaker consistency models offer more availability and performance, they also make reasoning about programs more difficult. Under serializability, a programmer could argue about invariants one transaction at a time, disregarding concurrent interactions. Under weaker models, this is no longer possible.

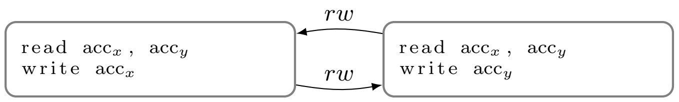
Our goal in this work is to propose a method to verify safety properties of a program running under a weak consistency model. Existing work for analyzing safety properties of weakly consistent programs [10, 11, 28, 5, 4] take serializability as a reference model for correctness and decompose the safety verification problem into two steps: (1) show that the safety property holds under serializability, and separately, (2) show that a program is robust against a weak consistency model. Robustness against a weak consistency model [7] means that the program has exactly the same observable behaviors as with serializability guarantees. To show robustness, these methods build a dependency graph from the program executions and check for cycles in the graph which violate serializability.

While sound, this method is often too strict. Most programs designed for weak consistency are expected to tolerate some anomalies and yet satisfy application-level safety properties. Consider the two programs in Figure 1. Neither program is robust against snapshot isolation, which allows concurrent transactions to commit if they write into a disjoint set of data variables. However, one of the programs exhibit buggy behavior with respect to the application-level invariant while the other one satisfies its application invariants under snapshot isolation.

The program in Figure 1(a) considers a simple bank application. The example has two concurrent transactions operating on two bank accounts with an invariant that the total amount in both accounts is nonnegative. While the serial executions of the transactions satisfy the assertion, their execution under weaker consistency models do not. Under snapshot isolation (SI), both transactions can read from the initial snapshot and be unaware of each other's updates. Both transactions can successfully commit under SI since they update disjoint sets of variables.

On the other hand, the non-serializable program in Figure 1(b) is correct. This program considers a simple course scheduling application with an invariant that only a single course is assigned to a time slot. The example has two concurrent transactions to schedule the given courses into time slots which operate on a timetable database. The transactions concurrently read the timetable

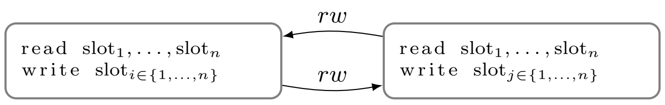
```
transaction T1()                          transaction T2()
  x = read acc_x in Accounts                 x = read acc_x in Accounts
  y = read acc_y in Accounts                 y = read acc_y in Accounts
  if (x + y) > 100                           if (x + y) > 100
    write acc_x (x-100) in Accounts            write acc_y (y-100) in Accounts
                  acc_x = acc_y = 60
                  T1() || T2()
                  assert(Accounts[acc_x] + Accounts[acc_y]) > 0
```


rw
read acc_x , acc_y
write acc_x
rw
read acc_x , acc_y
write acc_y

(a) Not robust against SI. Some executions under SI result in incorrect behavior.

```
transaction scheduleCourse(courseId)
   slots = read slot_1, ..., slot_n in TimeTable
   index = findAvailable(slots, courseId)
   if index >= 0
      write slot_index courseId in TimeTable

  scheduleCourse(courseId_1) || scheduleCourse(courseId_2)
  assert(TimeTable[courseId_1] != TimeTable[courseId_2])
```


rw
read $\mathrm{slot}_1, \ldots, \mathrm{slot}_n$
write $\mathrm{slot}_{i\in\{1,\ldots,n\}}$
rw
read $\mathrm{slot}_1, \ldots, \mathrm{slot}_n$
write $\mathrm{slot}_{j\in\{1,\ldots,n\}}$

(b) Not robust against SI. However, its executions under SI are correct.

Fig. 1: Two programs both of which are not robust against SI. While the first example fails to provide its specifications, the second example satisfies them.

slots, check for a slot that is available and that satisfies the course requirements, and commit the assignment of course to the time slot by marking the allocated slot. Both transactions can successfully commit under snapshot isolation if they write to disjoint slots, e.g., if $i \neq j$ in Figure 1(b). All executions under snapshot isolation satisfy the assertion.

In this paper, we describe a method to verify application-specific assertions in a transactional program running under a weak consistency model. Our method is parameterized over the underlying consistency model. Given a set of transactions on an underlying database, an assertion on the program state, and a consistency model, our method proceeds as follows. First, we use symbolic execution to construct a summary for each transaction. The summary describes the relation between the before- and after-states for each transaction. Second, we symbolically encode an ordering of the transactions in the program, and compose the transaction summaries according to this ordering. The ordering specifies the data flow relationships between the transactions. Third, we use the axiomatic approach of [16] to encode constraints on valid executions under the weak consistency model. Altogether, this reduces the problem of assertion verification to a satisfiability checking question for the conjunction of all these constraints.

We show the applicability of our approach on a set of benchmarks written in the Boogie programming language [3] and used Symbooglix [26] for symbolic execution of transactions. Our approach allows the use of existing symbolic execution and satisfiability checking tools for the problem of verifying programs running on weakly consistent systems with complicated sets of behaviors.

2 Transactions on Weakly Consistent Systems

2.1 Abstract Executions

We formalize weakly consistent transactions in an axiomatic way, based on the framework presented by Cerone et al. [16]. We consider a database which keeps a set of *variables* $\texttt{Vars} = \{x, y, \ldots\}$, replicated among a set of nodes in the distributed system. Clients interact with the database variables by running *transactions* $T \in \mathbb{T}$, which are programs issuing some read and write operations atomically on the database variables. For simplicity, we assume all the variables are integer valued and we define the operations on the variables as the set $\texttt{Op} = \{\mathrm{rd}(x, n), \mathrm{wr}(x, n) \mid x \in \texttt{Vars}, n \in \mathbb{Z}\}$. An *event* over $\texttt{Op}$ is a labeled invocation $\texttt{op}^{\ell}(x, n)$ of an operation. It consists of a unique identifier ℓ and an operation $\texttt{op} \in \texttt{Op}$ on a variable $x \in \texttt{Vars}$ and a value $n \in \mathbb{Z}$. For example, the event $\mathrm{rd}^{\ell}(x, 0)$ represents an event with the (unique) label ℓ that reads the value 0 from variable x and $\mathrm{wr}^{\ell'}(x, 1)$ represents a write of value 1 to variable x. When the label is not important, we omit it and write $\texttt{op}^{(\cdot)}(x, n)$.

Definition 1 (Transaction Trace and History). *A* transaction trace *is a pair* $(E, <_{po})$ *where* E *is a finite set of events over* $\texttt{Op}$ *and* program order $<_{po}$ *is a total order over* E*. A* history $H = \langle \texttt{Vars}, \{T_1, \ldots, T_n\}\rangle$ *consists of a set of variables* $\texttt{Vars}$ *and a finite set of transaction traces with pairwise disjoint sets of identifiers. We assume all transactions are potentially concurrent to each other.*

Intuitively, a transaction trace records a successful sequence of operations on a database atomically executed by a client in a transaction and the order in which the operations were performed. A history records a concurrent set of transactions. On weakly consistent systems, the distributed nodes are not immediately synchronized after commiting a transaction. Therefore, the updates of a transaction on a replicated variable may not immediately be visible to all the nodes. Weakly consistent systems allow executing client transactions without the necessity of receiving all the updates committed on different replicas.

Systems providing weak consistency define a conflict resolution policy on the set of its operations to resolve conflicting updates made by concurrent transactions, such as last writer wins (LWW) for a register data type or add wins for a set data type [31]. For example, Cassandra [24] attaches a timestamp to each data update and applies LWW conflict resolution, i.e., it chooses the data with the most recent timestamp in case of concurrent updates to a data variable.

An abstract execution of a weakly consistent system is formally defined by the binary relations *visibility vis* and the *arbitration ar* between the transactions in a history. We write $T_1 \xrightarrow{vis} T_2$ if $(T_1, T2) \in vis$ and similarly $T_1 \xrightarrow{ar} T_2$

if $(T_1, T_2) \in ar$. The visibility relation is a strict pre-order (i.e., irreflexive and transitive), and models the delivery of updates between the replicas of a variable: $T_1 \xrightarrow{vis} T_2$ means that the updates of transaction T_1 are delivered to the node executing the transaction T_2 and therefore T_2 operates on variables that have been updated based on the operations in T_1. Two transactions are *concurrent* if neither of them sees the effects of the other, i.e., $T_1 \not\xrightarrow{vis} T_2$ and $T_2 \not\xrightarrow{vis} T_1$. The arbitration relation is a total order; $T_1 \xrightarrow{ar} T_2$ intuitively means that the version of variables written by T_2 supersede the versions written by T_1. The arbitration relation can be computed by Lamport timestamps [25].

Definition 2 (Abstract Execution). *An (abstract)* execution *of a history H is a tuple $A = \langle H, vis, ar \rangle$ of the history H with a visibility relation $vis \subseteq H \times H$ and an arbitration relation $ar \subseteq H \times H$ such that $vis \subseteq ar$.*

The constraint $vis \subseteq ar$ in an execution ensures that if T_2 is aware of T_1 (i.e., $T_1 \xrightarrow{vis} T_2$), then T_2's writes supersede T_1's writes (i.e., $T_1 \xrightarrow{ar} T_2$).

The weakest consistency specification is *eventual consistency* [31, 13], which provides the basic guarantee that in a state where clients stop submitting transactions (which is called *quiescent state*) (i) all update transactions will eventually be visible to each node and (ii) the value of all the copies of the database variables will be the same. Eventual consistency is too weak by itself to satisfy the specifications of many applications. Hence, systems provide a spectrum of weak consistency models which provide additional guarantees on the system execution by requiring synchronization to some extent.

2.2 Axioms for Weak Consistency

In this section we recall a set of axioms summarized in [16, 7] whose combination can be used to define weak consistency models.

We need some notation before we can formally describe the axioms. For a total order $<\subseteq A \times A$ on a set A and a non-empty set $B \subseteq A$, we define $\max(B, <)$ (respectively, $\min(B, <)$) as the unique event $b \in B$ such that, for all $a \in B$, we have $a < b$ or $a = b$ (respectively, $b < a$ or $b = a$). The operations max and min are undefined if B is empty. For an event $a \in A$, we write $\mathsf{bf}(a, <)$ for the set $\{b \in A \mid b < a\}$ of events preceding a and write $\mathsf{bf}(a, <\mid B)$ for $\mathsf{bf}(a, <) \cap B$. For a set of events E and $x \in \texttt{Vars}$, we write E_x (resp. E^r_x, E^w_x) for the restriction of E to operations (resp. read, write operations) on variable x: $E_x = \{\mathsf{op}^\ell(\hat{x}, n) \in E \mid x = \hat{x}\}$, $E^r_x = \{\mathrm{rd}^\ell(\hat{x}, n) \in E \mid \hat{x} = x\}$, $E^w_x = \{\mathrm{wr}^\ell(\hat{x}, n) \in E \mid \hat{x} = x\}$.

The axiom Int is the *internal consistency axiom* which ensures that, within a transaction, the database provides sequential semantics: in a transaction $(E, <_{po})$, a read event e on a variable x returns the value of the last event on x preceding e. Formally,

$$\forall (E, <_{po}) \in H. \forall \mathrm{rd}^\ell(x, n)) \in E.$$
$$\mathsf{bf}(e, <_{po}\mid E_x) = \emptyset \ \vee\ \max(\mathsf{bf}(e, <_{po}\mid E_x)) \equiv \mathsf{op}^{(\cdot)}(x, n) \qquad (\textsc{Int})$$

The axiom Ext is the *external consistency axiom* which ensures that, if in $(E, <_{po})$ a read e on x is not preceded by an operation on the same variable, then its value is determined in terms of writes by other transactions visible to it, if no transaction has written to x, by the initial value 0. For a transaction $T = (E, <_{po})$, we define the predicate $\langle T \text{ writes } (x,n)\rangle$ as $\max(E^w_x, <_{po}) \equiv \mathrm{wr}^{(\cdot)}(x,n)$ and the predicate $\langle T \text{ reads } (x,n)\rangle$ as $\min(E^r_x) \equiv \mathrm{rd}^{(\cdot)}(x,n)$. We also define $\langle T \text{ writes } x\rangle$ as $\exists n \in \mathbb{Z}.\langle T \text{ writes } (x,n)\rangle$. Formally, the Ext axiom states:

$$\forall (E, <_{po}) \in H. \forall x \in \texttt{Vars}, \forall n \in \mathbb{Z}. \langle (E, <_{po}) \text{ reads } (x,n)\rangle \Rightarrow$$
$$(\mathsf{bf}(T, \xrightarrow{vis} | \ \mathrm{wr}(x)) = \emptyset \wedge n = 0) \vee \max(\mathsf{bf}(T, \xrightarrow{vis} | \ \mathrm{wr}(x)), \xrightarrow{ar}) \equiv \mathsf{op}^{(\cdot)}(x,n) \qquad (\text{Ext})$$

where $\mathrm{wr}(x) = \{T \in H \mid \langle T \text{ writes } x\rangle\}$.

The NoConflict axiom states that updates to the same variable by different transactions must be ordered by the visibility relation:

$$\forall T_1, T_2 \in H. (\exists x \in \texttt{Vars}. T_1 \text{ writes } x \wedge T_2 \text{ writes } x) \Rightarrow$$
$$T_1 = T_2 \vee T_1 \xrightarrow{vis} T_2 \vee T_2 \xrightarrow{vis} T_1 \qquad (\text{NoConflict})$$

The axiom TransVis states the transitivity of the *vis* relation:

$$\forall T_1, T_2, T_3 \in H. T_1 \xrightarrow{vis} T_2 \wedge T_2 \xrightarrow{vis} T_3 \implies T_1 \xrightarrow{vis} T_3 \qquad (\text{TransVis})$$

The Prefix axiom states that if T_3 observes T_2, then it also observes any transaction before T_2 in the arbitration order and hence it is stricter than TransVis:

$$\forall T_1, T_2, T_3 \in H : T_1 \xrightarrow{\text{ar}} T_2 \wedge T_2 \xrightarrow{\text{vis}} T_3 \implies T_1 \xrightarrow{\text{vis}} T_3 \qquad (\text{Prefix})$$

The axiom TotalVis states that *vis* is a total order, i.e., $vis = ar$, and hence it is stricter than Prefix:

$$\forall T_1, T_2 \in H. T_1 = T_2 \vee T_1 \xrightarrow{vis} T_2 \vee T_2 \xrightarrow{vis} T_1 \qquad (\text{TotalVis})$$

2.3 Weak Consistency Models

We now recall weak consistency models based on the axioms in Section 2.2 for which we provide symbolic encodings in Section 3. The definitions of the consistency models are summarized in Table 1.

Serializability (SER) [8] is a strong consistency model that guarantees the transactions to be executed serially and in the same order on every node. Formally, serializability allows executions which satisfy internal and external consistency for which the visibility relation is totally ordered.

Snapshot Isolation (SI) [6, 19] weakens the serializability guarantee by allowing concurrent execution of two transactions that do not write to the same data variable. A transaction may not see all committed transactions in the system but it sees a prefix of the total order of transactions. However, it cannot commit if

$$\begin{aligned}
\text{SER} &= \textsc{Int} \wedge \textsc{Ext} \wedge \textsc{TotalVis}\\
\text{SI} &= \textsc{Int} \wedge \textsc{Ext} \wedge \textsc{Prefix} \wedge \textsc{NoConflict}\\
\text{PSI} &= \textsc{Int} \wedge \textsc{Ext} \wedge \textsc{TransVis} \wedge \textsc{NoConflict}\\
\text{PC} &= \textsc{Int} \wedge \textsc{Ext} \wedge \textsc{Prefix}\\
\text{CC} &= \textsc{Int} \wedge \textsc{Ext} \wedge \textsc{TransVis}
\end{aligned}$$

Table 1: Definitions of the consistency models

it updates an intersecting set of data variables with the set of updated variables of a concurrent transaction (formalized by NoConflict axiom).
Parallel Snapshot Isolation (PSI) [32] relaxes SI by weakening the Prefix requirement which enforces a global ordering of transactions to causal delivery of transactions (TransVis). Causal delivery ensures the ordered delivery of causally related updates. If a transaction T_i was visible to the execution of the transaction T_j, i.e., T_j operates on the effects produced by T_i, then T_j is causally related to T_j. In causally consistent systems, all the replicas see the transactions T_i and T_j in that order. This is formalized by the axiom TransVis.
Prefix Consistency (PC) [14] is also a relaxation of SI which is not strictly stronger or weaker than PSI. PC is strict in the sense that it requires a transaction to see the updates of some prefix of all the updates w.r.t., *ar* relation, enforcing Prefix. On the other hand, it is weak on its guarantees for committing transactions. It allows conflicting updates of concurrent transactions, not enforcing NoConflict.
Causal Consistency (CC) [27] requires causally related transactions to be visible to other replicas in the causal order (TransVis). Some variants of causal consistency are defined in the context of both memory and distributed systems (e.g., causal memory [1], causal convergence [12, 13]). All these definitions are based on the requirement of causal delivery. Causal consistency guarantees are weaker than PSI as CC allows the transactions with conflicting set of updates to commit concurrently.

Verification Problem In this work, we study the program verification problem parametrized over the consistency model. Given a history with a set of transactions on the database variables, a consistency model, and an assertion on the database variable, we ask whether there is an abstract execution of the system allowed by the consistency model which violates the program assertion.

Definition 3 (Verification Problem). *Given a history $H = \langle \texttt{Vars}, \{T_1, \ldots, T_n\}\rangle$, a consistency model $cm \in \{\text{SER}, \text{SI}, \text{PSI}, \text{PC}, \text{CC}\}$, and a program assertion $\phi_{\textsc{Prog}}$ on the variables* `Vars`*, the verification problem asks whether there is an abstract execution $A = \langle H, vis, ar\rangle$ satisfying cm that violates $\phi_{\textsc{Prog}}$ after executing $\{T_1, \ldots, T_n\}$.*

In the next section, we present our method for answering the verification problem by using symbolic execution and encoding the possible set of program behaviors into a satisfiability formula.

3 Encoding Weakly Consistent Executions

Our method encodes the possible set of executions of a set of concurrent transactions under a given consistency model into a satisfiability formula Φ. Our encoding has three steps:

1. Symbolically executing each transaction to summarize its effects into symbolic valuation of variables (Section 3.1),
2. Connecting the symbolic valuations of the transactions together so that the composition captures only causally consistent sets of executions (Section 3.2)
3. Constraining the sets of executions w.r.t. a consistency model (Section 3.3)

3.1 Symbolic Execution of Transactions

In transactional programs, the effects of a transaction are made visible to other transactions *atomically*. The intermediate state of in-progress or rolling back transactions are not seen by any other transaction. This property allows for a modular encoding for each transaction independently of others.

Given a set of transactions $T_1, \ldots, T_n$, we execute each transaction on a symbolic state and obtain their symbolic summaries. The summary of the transaction is the relation between the initial symbolic snapshot and the final symbolic expressions for the variables. In order to track different sets of variables, for each transaction, we introduce two arrays of symbolic variables $\mathtt{X}$ and $\mathtt{X}'$. These arrays keep the symbolic values for each variable before and after the execution of a transaction respectively. The contents of $\mathtt{X'}_i$ with $1 \leq i \leq n$ keeps the updates made by the transaction in the arbitration order i on state $\mathtt{X}_i$ in a symbolic way.

For each transaction $T_i \in \{T_1, \ldots, T_n\}$, the formula Transaction-Summary represents the relationship between its input valuation $\mathtt{X}$ and the output valuation $\mathtt{X}'$ along with a map $\mathtt{Wr} : \mathtt{Vars} \mapsto \mathbb{B}$. $\mathtt{Wr}$ maps a variable to a Boolean value such that $\mathtt{Wr}.x$ is $\mathtt{true}$ if T_i modifies the data variable x. The valuation of the map $\mathtt{Wr}$ for a transaction is computed during its symbolic execution. This information is used later for detecting conflicts between concurrent transactions.

$$T_i(\mathtt{X}, \mathtt{X}', \mathtt{Wr}) \qquad \text{(Transaction-Summary)}$$

In order to represent valid executions, we have to "tie together" different symbolic states. The symbolic states of transactions are tied together in their arbitration order. We model the arbitration as a symbolic permutation of transactions. We introduce variables $ar_1, \ldots, ar_n$ which has distinct values from $\{1, \ldots, n\}$. The variable ar_i represents the identifier of the transaction in the arbitration order i, that is $ar_i = j$ iff T_j is the ith transaction in the arbitration.

Figure 2 illustrates the execution of transactions on the symbolic states. Each transaction T_{ar_i} in the arbitration order i, reads a symbolic state $\mathtt{X}_i$ and updates it to $\mathtt{X}'_i$. The first transaction is executed on $\mathtt{X}_0$ which contains the initial variable valuation. The next transactions can operate on either the initial symbolic state or another state produced by an earlier transaction.

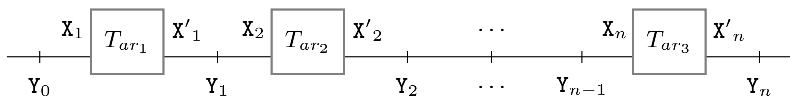


Fig. 2: The execution of transactions $T_{ar_1}, \ldots, T_{ar_n}$ in the arbitration order on the symbolic states $X_1, \ldots, X_n$ respectively.

Our encoding nondeterministically picks a symbolic state $\mathtt{X}_i$ with $1 \leq i \leq n$ from the set of states produced by the earlier transactions $\mathtt{X'}_j$ with $i < j \leq n$. We use an additional array of symbolic states $\mathtt{Y}_i$ to keep the effects of the first i transactions in the arbitration order. Initially, $\mathtt{Y}_0 = \mathtt{X}_0$. As we explain in the next subsection, the later values of $\mathtt{Y}$ are calculated by applying the transactions' effects in the arbitration order.

3.2 Encoding the Executions

In this subsection, we build a logical formula Φ which brings together the symbolic execution summaries of $T_1, \ldots, T_n$. The resulting formula models the possible executions of transactions as illustrated in Figure 2.

The set of symbolic valuations which satisfy Φ models all possible variable valuations that can be obtained after the execution of the transactions. All possible executions of $T_1, \ldots, T_n$ satisfy the program properties iff the intersection of Φ and the negation of the program properties is not satisfiable.

The encoding has three main components:

- The arbitration order of the transactions $T_1, \ldots, T_n$ (Arbitration)
- The input symbolic states transactions read from (Input-States)
- The symbolic states after the synchronization of updates (Output-States)

(Arbitration) encodes the arbitration order of the transaction summaries using the variables $ar_1, \ldots, ar_n$. The formula requires the variables $ar_1, \ldots, ar_n$ to have distinct values in $\{1, \ldots, n\}$. It also encodes that ar_i keeps the ith transaction T_j in the arbitration order.

$$\bigwedge_{1 \leq i \leq n} \bigvee_{1 \leq j \leq n} (ar_i = j) \wedge \bigwedge_{1 \leq i, j \leq n} (i \neq j \implies ar_i \neq ar_j)$$
$$\wedge \bigwedge_{1 \leq j \leq n} \bigvee_{1 \leq i \leq n} (ar_i = j \implies T_j(\mathtt{X}_i, \mathtt{X'}_i, \mathtt{Wr}_i)) \qquad \text{(Arbitration)}$$

(Input-States) encodes the possible sets of symbolic states $\mathtt{X}_i$ that a transaction in the arbitration order i can read from. A transaction in ith order can either read from the output snapshot of a transaction earlier than itself in the arbitration order or a symbolic state that has the effects of first $j < i$ transactions in the arbitration order. The formula $\phi_{\text{Read}}(cm)$ further restricts the set of symbolic values w.r.t. a consistency model $cm \in \{\text{CC}, \text{PSI}, \text{PC}, \text{SI}, \text{SER}\}$ as we explain in the next subsection.

$$\bigwedge_{0<i\leq n} \bigvee_{0\leq j<i} \left(\bigwedge_{x\in\texttt{Vars}} (\texttt{X}_i.x = \texttt{X}'_j.x) \vee \bigwedge_{x\in\texttt{Vars}} (\texttt{X}_i.x = \texttt{Y}_j.x) \right) \wedge \phi_{\textsc{Read}(cm)} \tag{\textsc{Input-States}}$$

(Output-States) encodes the valuation of the variables $\texttt{Y}_{1\leq i\leq n}$ which summarizes the effects of first i transactions in the arbitration order. For each variable $x \in \texttt{Vars}$, if the ith transaction writes to x, $\texttt{Y}_i.x$ is equal to the output value of ith transaction $\texttt{X}'_i.x$. Otherwise, $\texttt{Y}_i.x$ keeps the existing value of x, i.e., $\texttt{Y}_{i-1}.x$.[1] The formula $\phi_{\textsc{Write}}(cm)$ further restricts the set of symbolic values w.r.t. a consistency model $cm \in \{\mathrm{CC}, \mathrm{PSI}, \mathrm{PC}, \mathrm{SI}, \mathrm{SER}\}$.

$$\bigwedge_{0<i\leq n} \bigwedge_{x\in\texttt{Vars}} ((\texttt{Wr}_i(x) \implies \texttt{Y}_i = \texttt{X}'_i.x) \wedge (\neg\texttt{Wr}_i(x) \implies \texttt{Y}_i.x = \texttt{Y}_{i-1}.x)) \wedge \phi_{\textsc{Write}(cm)} \tag{\textsc{Output-States}}$$

(Initial) encodes the initial valuation of the variables. Initially, $\texttt{Y}_0$ is equal to the initial variable valuation.

$$\bigwedge_{x\in\texttt{Vars}} (\texttt{Y}_0.x = \texttt{X}'_0.x \ \wedge \ \texttt{X}_0.x = \texttt{X}'_0.x) \tag{\textsc{Initial}}$$

The formula Φ is the intersection of the formulas above, which encodes all possible executions of $T_1, \ldots, T_n$ satisfying Int, Ext and TransVis:

$$\Phi = (\textsc{Arbitration}) \ \wedge \ (\textsc{Input-States}) \ \wedge \ (\textsc{Output-States}) \ \wedge \ (\textsc{Initial})$$

We check whether some property $\phi_{\textsc{Prog}}$ on $x \in \texttt{Vars}$ holds for all possible executions of the program by using (Program-Prop). We obtain (Program-Prop) by replacing the accesses of $x \in \texttt{Vars}$ to the symbolic valuation $\texttt{Y}_n.x$, so that we evaluates ϕ on the symbolic valuation of the variables obtained after executing transactions $T_1, \ldots, T_n$.

$$\phi_{\textsc{Prog}}[\forall x \in \texttt{Vars}.\ \texttt{Y}_n.x/x] \tag{\textsc{Program-Prop}}$$

Theorem 1. *The encoded set of variable valuations satisfies the property ϕ under causal consistency iff the formula $\Phi \wedge \neg$(Program-Prop) is not satisfiable with $\phi_{\textsc{Read}}(cm) = \texttt{true}$ and $\phi_{\textsc{Write}}(cm) = \texttt{true}$.*

The theorem follows from the fact that the encoded set of executions satisfies Int, Ext and TransVis axioms. The axiom Int is trivially satisfied by the Transaction-Summary which is obtained by symbolically executing a transaction. Ext is satisfied by restricting the input symbolic valuation $\texttt{X}_i.x$ to the last visible value to the transaction in the arbitration order i. The axiom TransVis is satisfied by the relation between the symbolic input/output states of transactions. For any three transactions T_{ar_i}, T_{ar_j} and T_{ar_k} such that $\texttt{X}_j = \texttt{X}'_i$ and $\texttt{X}_k = \texttt{X}'_j$, T_{ar_k} operating on the symbolic output state of T_{ar_j} sees the effect of T_{ar_j} as well as the effect of T_{ar_i} on whose output state T_{ar_j} operates. Therefore, for all $T_{ar_i} \xrightarrow{vis} T_{ar_j}$ and $T_{ar_i} \xrightarrow{vis} T_{ar_k}$, we have $T_{ar_i} \xrightarrow{vis} T_{ar_k}$.

[1] Our encoding follows Last Writer Wins (LWW) policy.

CC	$\phi_{\text{READ}}(\text{CC}) = \mathsf{true}$	$\phi_{\text{WRITE}}(\text{CC}) = \mathsf{true}$
PSI	$\phi_{\text{READ}}(\text{PSI}) = \mathsf{true}$	$\phi_{\text{WRITE}}(\text{PSI}) = \phi\text{-NoConflict}$
PC	$\phi_{\text{READ}}(\text{PC}) = \phi\text{-Prefix}$	$\phi_{\text{WRITE}}(\text{PC}) = \mathsf{true}$
SI	$\phi_{\text{READ}}(\text{SI}) = \phi\text{-Prefix}$	$\phi_{\text{WRITE}}(\text{SI}) = \phi\text{-NoConflict}$
SER	$\phi_{\text{READ}}(\text{SER}) = \phi\text{-TotalVis}$	$\phi_{\text{WRITE}}(\text{SER}) = \mathsf{true}$

Fig. 3: Encodings for ϕ_{READ} and ϕ_{WRITE} to constrain the set of executions.

Notice that all the program behaviors encoded by the formula—without any further restrictions on which symbolic values to read or which transactions to commit—are not allowed by all weak consistency models. In the next subsection, we restrict the executions for different consistency models.

3.3 Encoding the Consistency Model

We model the allowed set of executions $A = \langle H, vis, ar \rangle$ under the given consistency model $cm \in \{$ PSI, PC, SI, SER$\}$ by incorporating the restrictions of these consistency models into two additional constraints in the formula Φ:

$\phi_{\text{READ}}(cm)$	Constrains the visibility relation $vis \subseteq ar$, i.e., the effects of which transactions can be visible for a transaction by restricting the set of symbolic states a transaction can read from
$\phi_{\text{WRITE}}(cm)$	Constrains which transactions can commit concurrently based on the set of variables they write to

In order to satisfy Prefix, TotalVis and NoConflict, we restrict the possible set of symbolic valuations which satisfy Φ. Figure 3 lists the formulas for constraining the executions to satisfy the axioms defined in Section 2.3.

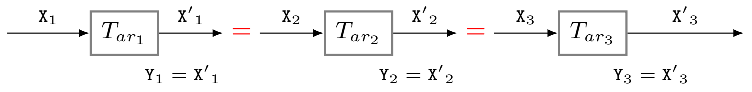
(ϕ-Prefix) requires the symbolic variable valuation $\mathtt{X}_i$ read by a transaction to be a prefix state, i.e., a valuation of variables obtained after the effect of a prefix of transactions in the arbitration order.

$$\bigwedge_{1\leq i\leq n} \bigvee_{0\leq j<i} \bigwedge_{x\in\mathtt{Vars}} (\mathtt{X}_i.x = \mathtt{Y}_j.x) \qquad (\phi\text{−Prefix})$$

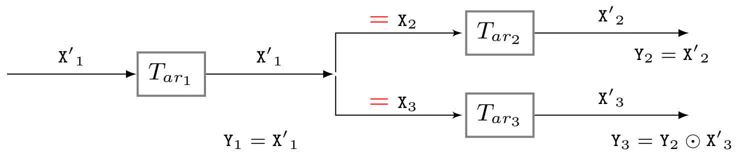
(ϕ-TotalVis) requires the visibility relation to be a total order. It requires the input state $\mathtt{X}_i$ of the transaction in the ith arbitration order to be the output state of the transaction in the $(i-1)$th arbitration order, i.e., $\mathtt{X'}_{i-1}$. In this case, $\mathtt{X'}_{i-1}$ is equal to $\mathtt{Y}_{i-1}$ since there are no concurrent transactions.

$$\bigwedge_{1\leq i\leq n} \bigwedge_{x\in\mathtt{Vars}} (\mathtt{X}_i.x = \mathtt{Y}_{i-1}.x) \qquad (\phi\text{-TotalVis})$$

(ϕ-NoConflict) requires that if the transaction T_{ar_i} reads from the symbolic state of an earlier transaction T_{ar_k}, there are not any transactions T_{ar_j} in between T_{ar_k} and T_{ar_i} (where $T_{ar_j} \overset{vis}{\not\longrightarrow} T_{ar_i}$ and $T_{ar_i} \overset{vis}{\not\longrightarrow} T_{ar_j}$) which update the same variable with T_{ar_i}.



(a) The symbolic execution produces a serial execution of T_{ar_1}, T_{ar_2} and T_{ar_3}.



(b) The symbolic execution produces an execution under snapshot isolation. Both transactions T_{ar_2} and T_{ar_3} operate on the snapshot valuation $\mathtt{X'}_1$ produced by T_{ar_1}.

Fig. 4: Two different executions of transactions $T_{ar_1}, T_{ar_2}, T_{ar_3}$. Initially, $\mathtt{Y}_0 = \mathtt{X}_0$.

$$\bigwedge_{0\le k<i\le n}\ \bigwedge_{x\in\mathtt{Vars}} \left((\mathtt{X}_i = \mathtt{X'}_k \wedge \mathtt{Wr}_{ar_i}.x) \implies \bigwedge_{k<j<i} \neg\mathtt{Wr}_{ar_j}.x \right) \quad (\phi\text{-NoConflict})$$

Theorem 2. *The answer to the verification problem in Definition 3 for a history $H = \langle \mathtt{Vars}, \{T_1, \ldots, T_n\}\rangle$, consistency model $cm \in \{\mathrm{CC}, \mathrm{PSI}, \mathrm{PC}, \mathrm{SI}, \mathrm{SER}\}$, and program assertion ϕ_{PROG} is* Yes *iff $\Phi \wedge \neg$*Program-Prop *is not satisfiable.*

Example 1. Figure 4(a) encodes a serializable execution by sequencing the symbolic states in the arbitration order. It feeds the output state of a transaction T_{ar_i} to the input transaction of $T_{ar_{i+1}}$ where $T_i \xrightarrow{ar} T_{ar_{i+1}}$. In a serializable execution, the prefix state which summarize the effects of first i transactions w.r.t. the arbitration is equal to the output valuation of the ith transaction.

Example 2. Figure 4(b) encodes an execution of transactions under snapshot isolation. In that particular execution, both transactions T_{ar_2} and T_{ar_3} operate on the output snapshot of T_{ar_1} (i.e., $T_{ar_1} \xrightarrow{vis} T_{ar_2}$ and $T_{ar_1} \xrightarrow{vis} T_{ar_3}$) and they are concurrent to each other (i.e., $T_{ar_2} \not\xrightarrow{vis} T_{ar_3}$ and $T_{ar_3} \not\xrightarrow{vis} T_{ar_2}$). This is a valid execution under SI iff T_{ar_2} and T_{ar_3} update a disjoint set of variables. Similar to the serializable case, $\mathtt{Y}_0$ keeps the initial values of the data variables which is read by the transaction with the smallest arbitration, $\mathtt{Y}_1 = \mathtt{X}[1]$ keeps the updates of T_{ar_1}. The snapshots $\mathtt{Y}_2$ and $\mathtt{Y}_3$ keep the effects up to the second and third transactions in the arbitration respectively. In this example, the symbolic valuations $\mathtt{Y}$ and $\mathtt{X}$ differ from each other. Consider $T_{ar_2} \xrightarrow{ar} T_{ar_3}$ where T_{ar_2} writes to $x \in \mathtt{Vars}$ and T_{ar_3} writes to $y \in \mathtt{Vars}$. Then, the symbolic states $\mathtt{X}_2$ and $\mathtt{X}_3$ would be respectively be aware of only the updates on x and y respectively. On the other hand, $\mathtt{Y}_3$ would summarize the effects of all three transactions, incorporating the updates on both x and y. Consider an alternative execution of the same example in Figure 4(b) under causal consistency, which allows conflicting updates. In a case where both T_{ar_2} and T_{ar_3} write to the same variable, the value in the snapshot $\mathtt{Y}_3$ would be determined by the conflict resolution policy (denoted with $\odot$ in the figure). For LWW policy, the final value is the value written by the transaction with the highest arbitration among the concurrent transactions (as given in (Output-States) formula), resulting in *lost update* in this example.

4 Experiments

We show the applicability of our approach on a set of benchmarks extracted from the literature related to weakly consistent databases. We encoded our formulas in Boogie language [3] using its support for symbolic variables and symbolically executed the transactions using Symbooglix [26] symbolic execution engine.

We performed our experiments on `Auction`, an online auction application from [29], `Courseware`, a course registration service adapted from [20, 28, 29], `FusionTicket`, a ticket sales application adapted from [21], and a simple banking system, `SimpleBank`, extracted from [16], all of which operate on key-value data stores. We implemented these systems in Boogie language and instrumented with the encoding of our formula. Then, we symbolically executed the instrumented transactions using Symbooglix and checked whether concurrent execution of some benchmark set of transactions satisfy the applications' properties.

Table 2 lists the number of transactions (#T) in each benchmark and whether the application properties are satisfied (ϕ) under different consistency models. We also report the number of satisfiability queries solved (#q) and run time in seconds (t) for computing different paths of the symbolic execution of the instrumented transactions. In addition to the consistency model, the number of satisfiability queries and hence the run time depend on the number of transactions, the number of variables read/written by the transactions, and variable accesses of concurrent transactions (i.e., the paths with certain concurrent accesses are infeasible for some consistency models, pruning further exploration of certain paths). We collected the results on a machine with a 2.6 GHz Intel Core i7 processor and 16 GB memory running macOS Catalina.

`Auction` models an auction system with transactions to start an auction, place a bid, and close an auction. The application requires that when an auction is closed, the declared winner is the bidder with the highest bid. Two different versions of the application are given in [29]. In the first version, the application property can be violated in the concurrent execution of transaction for placing bids, and close of an auction. The second version aims to satisfy the property by introducing tokens to replicas and closing an auction only after all tokens are collected. We implemented and verified both versions. As shown in Table 2, concurrent execution of start/close auction and two bidding transactions may fail to satisfy the application property under consistency models weaker than serializability in the first version (V1), while the second version (V2) satisfies it.

`Courseware` application provides transactions to add courses, add students, enroll students to courses, and schedule courses to timetable slots. In the first benchmark (B1), we check whether the property of having unique names for each

Table 2: Experimental results for the benchmarks for varying consistency models.

Benchmark			SER			SI			PSI			PC			CC		
		T	#q	t	ϕ	#q	t	ϕ	#q	t	ϕ	#q	t	ϕ	#q	t	ϕ
`Auction`	V1	4	14000	360	✓	15134	490	✗	14414	418	✗	13982	381	✗	13262	300	✗
	V2	4	17993	1578	✓	18001	1393	✓	18349	1653	✓	17945	1495	✓	17897	1533	✓
`Courseware`	B1	2	1073	23	✓	833	19	✗	593	13	✗	739	15	✗	499	9	✗
	B2	2	1514	353	✓	949	95	✓	589	62	✓	843	94	✗	483	34	✗
	B3	3	1745	509	✓	1325	298	✓	1257	323	✓	977	160	✓	881	82	✓
`FusionTicket`	B1	3	1340	35	✓	1329	35	✓	1322	34	✓	1318	35	✗	1311	31	✗
	B2	2	956	31	✓	825	24	✓	753	21	✓	765	22	✗	693	17	✗
`SimpleBank`	B1	2	446	26	✓	348	16	✗	314	12	✗	280	12	✗	246	8	✗

course holds if two transactions concurrently add a course with different ids but the same name. While the property is satisfied under serializability, transactions writing to different keys of the courses table can commit the same course name in the weaker consistency models. In (B2), we check whether the property of assigning each slot to a single course holds if two transactions concurrently schedule a course. As explained in the example in Figure 1(b), this property holds under weaker consistency models SI and PSI as well as SER. In (B3), we run transactions for adding a student, adding a course and enrolling the newly added student to the newly added course concurrently. We verify that the application property that requires each enrolled student and course exist in students and courses respectively is satisfied in all consistency models.

`FusionTicket` application provides transactions to add events and purchase tickets. The application updates the price of a ticket based on the sold number of tickets and has an application property on the expected amount to be collected from the tickets. Concurrent execution of multiple purchase transactions (B1) may violate this property under weak consistency models. The application also requires each event to be assigned to a different venue. Concurrent execution of multiple transactions for adding events (B2), does not violate that property under SI and PSI as well as SER. Because, the transactions writing to the same venue cannot commit concurrently under SI and PSI.

`SimpleBank` is the implementation of the example in Figure 1(a). Concurrent transactions to withdraw some amount may violate the property of nonnegative balance under consistency models weaker than SER.

Our experiments show that our approach can be used for verifying whether an application's properties hold when a set of transactions are run concurrently on a weakly consistent database. As the method is parametric to the consistency model, it is easy to check for the properties for a spectrum of consistency models.

5 Related Work

A vast amount of work is devoted to relaxing the consistency in the context of both databases and weak memory. Here we limit our focus to the correctness of weakly consistent programs assuming the correctness of the underlying system.

A line of existing work reason about the correctness of weakly consistent programs based on the serializability of the transactions [15, 33, 10, 11, 28]. The notion of *robustness* against consisteny models [7] is introduced to characterize whether the program produces the same behavior on a weakly consistent or serializable system. The serializability of weakly consistent transactions is analyzed using both dynamic and static methods. The work in [10] builds a dynamic analyzer which incorporates commutativity and absorption properties of operations, and [11] presents a static analysis tool for detecting non-serializable behaviors. The work in [28] reduces serializability checking to a satisfiability problem for automated detection of serializability violations. Focusing on widely used consistency models, recent work presents algorithms for verifying robustness against SI [4] and CC [5]. The robustness notion is also extended for different consistency models (e.g., robustness against PSI towards SI) [17]. While these definitions relax the serializability requirement, robustness towards a consistency model is still restrictive for checking the correctness of applications.

Some works verify the application properties of the specifications of weakly consistent programs. The work in [20, 30] propose a proof system for showing the application invariants hold under some choice of consistency guarantees of distributed operations. While this work requires low level operational reasoning, the work in [22] presents a system for compositional rely-guarantee style proof system for concurrent transactions running on weakly consistent systems. The work in [23] presents a program transformation based technique for verifying transactional programs with relaxed operations. With a motivation of preserving program invariants, *Explicit Consistency* [2] is proposed as a variant of weak consistency that exploits static analysis techniques to infer conflicting operations. In a recent work, [29] presents a proof rule to verify specifications of distributed objects. Differently, our method verifies implementations of transactional programs by modeling the behavior of underlying weakly consistent system.

In the context of exploring program behaviors, Repliss tool [34] exercises executions of an application with randomized invocations. Commander [18] explores the execution of a weakly consistent program using a bounded scheduler parameterized in both the schedule exploration strategy and also the consistency model. Different from execution based approaches which runs the system for different possible executions, we present an encoding which symbolically captures all behaviors of the program implementation under a consistency specification.

6 Conclusion

We presented a satisfiability based method for the verification of transactional programs running on weak consistency models. Our method summarizes the transactions by using symbolic execution, encodes the set of possible program executions under a consistency model into a satisfiability formula and checks the program assertions in the symbolic set of program states satisfying the formula. To the best of our knowledge, our work is the first to utilize symbolic execution techniques for the analysis of weakly consistent transactions.

Acknowledgements. We would like to thank Rupak Majumdar for his feedback and contribution during various stages of this work.

References

1. Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal memory: Definitions, implementation, and programming. Distributed Comput. **9**(1), 37–49 (1995)
2. Balegas, V., Duarte, S., Ferreira, C., Rodrigues, R., Preguiça, N.M., Najafzadeh, M., Shapiro, M.: Putting consistency back into eventual consistency. In: The 10th European Conf. on Computer Systems, EuroSys. pp. 6:1–6:16. ACM (2015)
3. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: Formal Methods for Components and Objects, FMCO. LNCS, vol. 4111, pp. 364–387. Springer (2005)
4. Beillahi, S.M., Bouajjani, A., Enea, C.: Checking robustness against snapshot isolation. In: Computer Aided Verification - 31st Int. Conf., CAV. LNCS, vol. 11562, pp. 286–304. Springer (2019)
5. Beillahi, S.M., Bouajjani, A., Enea, C.: Robustness against transactional causal consistency. In: 30th Int. Conf. on Concurrency Theory, CONCUR. LIPIcs, vol. 140, pp. 30:1–30:18 (2019)
6. 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: ACM SIGMOD Int. Conf. on Management of Data. pp. 1–10. ACM Press (1995)
7. Bernardi, G., Gotsman, A.: Robustness against consistency models with atomic visibility. In: 27th Int. Conf. on Concurrency Theory, CONCUR. LIPIcs, vol. 59, pp. 7:1–7:15 (2016)
8. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley (1987)
9. Brewer, E.A.: Towards robust distributed systems (abstract). In: The 9th Annual ACM Symp. on Principles of Distributed Computing. p. 7. ACM (2000)
10. Brutschy, L., Dimitrov, D., Müller, P., Vechev, M.T.: Serializability for eventual consistency: criterion, analysis, and applications. In: The 44th ACM SIGPLAN Symp. on Principles of Programming Languages, POPL. pp. 458–472. ACM (2017)
11. Brutschy, L., Dimitrov, D., Müller, P., Vechev, M.T.: Static serializability analysis for causal consistency. In: The 39th ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI. pp. 90–104. ACM (2018)
12. Burckhardt, S.: Principles of eventual consistency. Foundations and Trends in Programming Languages **1**(1-2), 1–150 (2014)
13. Burckhardt, S., Gotsman, A., Yang, H., Zawirski, M.: Replicated data types: specification, verification, optimality. In: The 41st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL. pp. 271–284. ACM (2014)
14. Burckhardt, S., Leijen, D., Protzenko, J., Fähndrich, M.: Global sequence protocol: A robust abstraction for replicated shared state. In: 29th European Conf. on Object-Oriented Programming, ECOOP. LIPIcs, vol. 37, pp. 568–590 (2015)
15. Cahill, M.J., Röhm, U., Fekete, A.D.: Serializable isolation for snapshot databases. ACM Trans. Database Syst. **34**(4), 20:1–20:42 (2009)
16. Cerone, A., Bernardi, G., Gotsman, A.: A framework for transactional consistency models with atomic visibility. In: 26th Int. Conf. on Concurrency Theory, CONCUR. LIPIcs, vol. 42, pp. 58–71 (2015)

17. Cerone, A., Gotsman, A.: Analysing snapshot isolation. J. ACM **65**(2), 11:1–11:41 (2018)
18. Dabaghchian, M., Rakamaric, Z., Kulahcioglu Ozkan, B., Mutlu, E., Tasiran, S.: Consistency-aware scheduling for weakly consistent programs. ACM SIGSOFT Software Engineering Notes **42**(4), 1–5 (2017)
19. Fekete, A., Liarokapis, D., O'Neil, E.J., O'Neil, P.E., Shasha, D.E.: Making snapshot isolation serializable. ACM Trans. Database Syst. **30**(2), 492–528 (2005)
20. Gotsman, A., Yang, H., Ferreira, C., Najafzadeh, M., Shapiro, M.: 'cause i'm strong enough: reasoning about consistency choices in distributed systems. In: The 43rd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL. pp. 371–384. ACM (2016)
21. Holt, B., Bornholt, J., Zhang, I., Ports, D.R.K., Oskin, M., Ceze, L.: Disciplined inconsistency with consistency types. In: The 7th ACM Symp. on Cloud Comp. pp. 279–293. ACM (2016)
22. Kaki, G., Nagar, K., Najafzadeh, M., Jagannathan, S.: Alone together: compositional reasoning and inference for weak isolation. Proc. ACM Program. Lang. **2**(POPL), 27:1–27:34 (2018)
23. Kuru, I., Kulahcioglu Ozkan, B., Mutluergil, S.O., Tasiran, S., Elmas, T., Cohen, E.: Verifying programs under snapshot isolation and similar relaxed consistency models. In: The 9th ACM SIGPLAN Workshop on Transactional Comp. (2014)
24. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. Operating Systems Review **44**(2), 35–40 (2010)
25. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21**(7), 558–565 (1978)
26. Liew, D., Cadar, C., Donaldson, A.F.: Symbooglix: A symbolic execution engine for boogie programs. In: 2016 IEEE Int. Conf. on Software Testing, Verification and Validation, ICST. pp. 45–56. IEEE Computer Society (2016)
27. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In: The 23rd ACM Symp. on Op. Sys. Principles 2011, SOSP. pp. 401–416. ACM (2011)
28. Nagar, K., Jagannathan, S.: Automated detection of serializability violations under weak consistency. In: 29th Int. Conf. on Concurrency Theory, CONCUR. LIPIcs, vol. 118, pp. 41:1–41:18 (2018)
29. Nair, S.S., Petri, G., Shapiro, M.: Proving the safety of highly-available distributed objects. In: Programming Languages and Systems - 29th European Symp. on Programming, ESOP. LNCS, vol. 12075, pp. 544–571. Springer (2020)
30. Najafzadeh, M., Gotsman, A., Yang, H., Ferreira, C., Shapiro, M.: The CISE tool: proving weakly-consistent applications correct. In: The 2nd Workshop on the Principles and Practice of Consistency for Distributed Data. pp. 2:1–2:3. ACM (2016)
31. Shapiro, M., Preguiça, N.M., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Stabilization, Safety, and Security of Distributed Systems - 13th Int. Symp., SSS. LNCS, vol. 6976, pp. 386–400. Springer (2011)
32. Sovran, Y., Power, R., Aguilera, M.K., Li, J.: Transactional storage for geo-replicated systems. In: The 23rd ACM Symp. on Op. Sys. Principles 2011, SOSP. pp. 385–400. ACM (2011)
33. Zellag, K., Kemme, B.: How *consistent* is your cloud application? In: ACM Symp. on Cloud Comp., SOCC. p. 6 (2012)
34. Zeller, P.: Testing properties of weakly consistent programs with repliss. In: The 3rd Int. Workshop on Principles and Practice of Consistency for Distributed Data. pp. 3:1–3:5 (2017)