Trace-Based Symbolic Analysis for Atomicity Violations

Chao Wang[1], Rhishikesh Limaye[2], Malay Ganai[1], and Aarti Gupta[1]

[1] NEC Laboratories America, Princeton, NJ, USA
[2] University of California, Berkeley, CA, USA

Abstract. We propose a symbolic algorithm to accurately predict atomicity violations by analyzing a concrete execution trace of a concurrent program. We use both the execution trace and the program source code to construct a symbolic predictive model, which captures a large set of alternative interleavings of the events of the given trace. We use precise symbolic reasoning with a satisfiability modulo theory (SMT) solver to check the feasible interleavings for atomicity violations. Our algorithm differs from the existing methods in that all reported atomicity violations can appear in the actual program execution; and at the same time the feasible interleavings analyzed by our model are significantly more than other predictive models that guarantee the absence of false alarms.

1 Introduction

Atomicity, or *serializability*, is a semantic correctness condition for concurrent programs. Intuitively, a thread interleaving is serializable if it is equivalent to a serial execution, i.e. a thread interleaving which executes a transactional block without other threads interleaved in between. The transactional blocks are typically marked explicitly in the code. Much attention has recently been focused on *three-access* atomicity violations [1,2], which involves one shared variable and three consecutive accesses to the variable. Here we characterize consecutive accesses with respect to a shared variable; these accesses can be separated by events over possibly other shared variables. If two accesses in a local thread, which are inside a transactional block, are interleaved in between by an access in another thread, this interleaving may be unserializable if the remote access has data conflicts with the two local accesses. In practice, unserializable interleavings often indicate the presence of subtle concurrency bugs in the program.

Known techniques for detecting atomicity violations fall into the following three categories: static detection, runtime monitoring, and runtime prediction. Type-state or other static analysis based methods [3,4] try to identify potential violations at compile time. These methods typically ignore data and most of the synchronization primitives other than locks, and tend to report a large number of bogus errors. Runtime monitoring aims at identifying atomicity violations exposed by a given execution trace [5,1,6,7,8]. However, it is a challenging task during testing to trigger the erroneous thread schedule in the first place. In contrast, runtime prediction aims at detecting atomicity violations in all feasible interleavings of events of the given trace. In other words, even if no violation exists in that trace, but an alternative interleaving is erroneous, a predictive method [9,2,10,11,12,13] may be able to catch it without actually re-running the test.

Although there have been several predictive methods in the literature, they either suffer from imprecision as a result of conservative modeling (or no modeling at all) of the program data flow and consequently many false negatives [9,2,10], or suffer from a very

J. Esparza and R. Majumdar (Eds.): TACAS 2010, LNCS 6015, pp. 328–342, 2010.

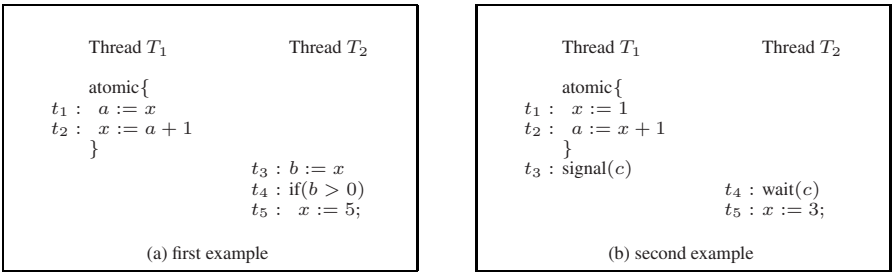

Fig. 1. Ignoring data/synchronizations may lead to bogus errors. All variables are initialized to 0.

limited coverage of interleavings due to trace-based under-approximations [11,12,13]. Previous efforts [4,2,10], for instance, focus on the control paths and model only locks provided that they obey the nested locking discipline. Their model can be viewed as abstracting other synchronization primitives into NOPs, including semaphores, barriers, POSIX condition variables, and Java's wait-notify[1]. Because of such approximations, the reported atomicity violations may not exist in the actual program. Although *potential* atomicity violations can serve as good hints for subsequent analysis, they are often not immediately useful to programmers, because manually deciding whether such violations exist in the actual program execution itself is a very challenging task.

Fig. 1 provides two examples in which the transactions, marked by keyword *atomic*, are indeed serializable, but *atomizer* [9] or methods in [2,10] would report them as atomicity violations. In each example, there are two concurrent threads T_1, T_2 and a shared variable x. Variables a, b are thread-local and variable c is a condition variable, accessible through POSIX-style signal/wait. The given trace is denoted by event sequence $t_1t_2t_3t_4t_5$ and is a serial execution. If one ignores data and synchronizations, there seems to be alternative interleavings, $t_1t_3t_4t_5t_2$ in (a) and $t_1t_4t_5t_2t_3$ in (b), that are unserializable. However, these interleavings cannot occur in the actual program execution, because of the initial value $x = 0$ and the if-condition in the first example and the signal/wait in the second example.

Methods using happens-before causalities [11,12] often guarantee no bogus errors, but tend to miss many real ones. Fig. 2 shows a model in this category—the maximal causal model [12]—for the examples in Fig. 1. This model has been shown in [12] to subsume many earlier happens-before causal models. Here events accessing the shared variable x are represented by the actual values read/written in the given trace, and events involving thread-local variables only are abstracted into NOPs. The model admits all interleavings in which these *concrete events* are sequentially consistent. In Fig. 2, for example, the alternative sequences are deemed as sequentially inconsistent in both programs, because consecutive reads t_1, t_3 in the first example return different values, and in the second example t_2 reads in 1 from x immediately after t_5 writing 3. Therefore, this model can avoid reporting these two bogus errors. However, consider modifying the programs in Fig. 1 by changing t_4 in the first example into `if(b≥0)`, and removing the signal/wait of t_3, t_4 in the second example. Now, the aforementioned alternative interleavings expose real atomicity violations, but in both examples, the concrete read/write events (Fig. 2) remain the same—these real violations will be missed.

[1] These synchronization primitives cannot be simulated using only nested locks.

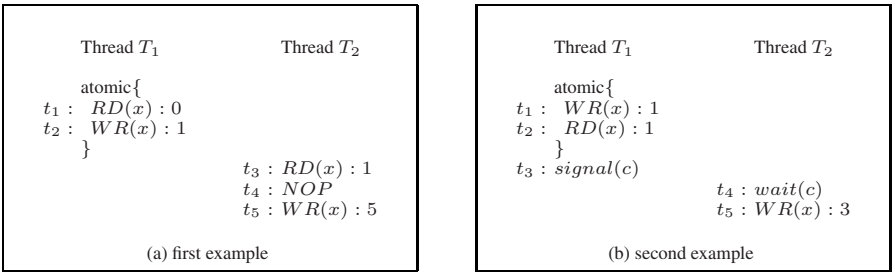

Fig. 2. Predictive models using under-approximations may miss real errors

In this paper, we propose a more precise algorithm for predicting atomicity violations. Given an execution trace on which transactional blocks are explicitly marked, we check all alternative interleavings of the *symbolic events* of that trace for three-access atomicity violations. The symbolic events are constructed from both the concrete trace and the program source code. Compared to existing causal models, for example, [12], our model covers more interleavings while guaranteeing no false alarms. Since the algorithm is more precise than the methods in [9,2], we envision the following procedure in which it may be applied:

1. Run a test of the concurrent program to obtain an execution trace.
2. Run a sound but over-approximate algorithm [9,2] to detect all *potential* atomicity violations. If no violation is found, return.
3. Build the precise predictive model, and for each potential violation, check whether it is feasible. If it is feasible, create a concrete and replayable witness trace.

More specifically, we formulate the checking in Step 3 as a satisfiability problem, by constructing a formula which is satisfiable iff there exists a feasible and yet unserializable interleaving of events of the given trace. The formula is in a quantifier-free first-order logic and is decided by a Satisfiability Modulo Theory (SMT) solver [14].

Our main contributions are applying the trace-based symbolic predictive model to analyzing atomicity and encoding the detection of three-access violations on its interleavings as an SMT problem, followed by the subsequent analysis using a SMT solver. Our model for predicting atomicity violations tracks the actual data flow and models all synchronization primitives precisely. The greater capability of covering interleavings by our method is due to the use of concrete trace as well as the program source code. Furthermore, using symbolic techniques rather than explicit enumeration makes the analysis less sensitive to the large number of interleavings.

The remainder of this paper is organized as follows. After establishing notation in Section 2 and Section 3, we present the SMT-based algorithm for detecting atomicity violations in Section 4. In Section 5, we explain how to search for an erroneous prefix as opposed to a complete interleaving. We present experimental results in Section 6, review related work in Section 7, and give our conclusions in Section 8.

2 Preliminaries

Programs and Traces. A *concurrent program* has a set of *threads* and a set SV of *shared variables*. Each thread T_i, where $1 \leq i \leq k$, has a set of *local variables* LV_i.

- Let $Tid = \{1, \ldots, k\}$ be the set of thread indices.
- Let $V_i = SV \cup LV_i$, where $1 \leq i \leq k$, be the set of variables accessible in T_i.

The remaining aspects of a concurrent program are left unspecified, to apply more generally to different programming languages. An *execution trace* is a sequence of events $\rho = t_1 \ldots t_n$. An *event* $t \in \rho$ is a tuple $\langle tid, action \rangle$, where $tid \in Tid$ and $action$ is a computation of the form $(\mathsf{assume}(c), asgn)$, i.e. a *guarded assignment*, where

- $asgn$ is a set of assignments, each of the form $v := exp$, where $v \in V_i$ is a variable and exp is an expression over V_i.
- $\mathsf{assume}(c)$ means the conditional expression c over V_i must be true for the assignments in $asgn$ to execute.

Each event t in ρ is a unique execution instance of a statement in the program. If a statement in the textual representation of the program is executed multiple times, e.g., in a loop or a recursive function, each execution instance is modeled as a separate event. By defining the expression syntax suitably, the trace representation can model executions of any multithreaded program[2].

The guarded assignment action has three variants: (1) when the guard $c = \mathsf{true}$, it models normal assignments in a basic block; (2) when the assignment set $asgn$ is empty, $\mathsf{assume}(c)$ models the execution of a branching statement `if(c)`; and (3) with both the guard and the assignment set, it can model the atomic *check-and-set* operation, which is the foundation of all concurrency/synchronization primitives.

Synchronization Primitives. We use the guarded assignments in our implementation to model all synchronization primitives in POSIX Threads (or *PThreads*). This includes locks, semaphores, condition variables, barriers, etc. For example, acquire of a mutex lock l in the thread T_i, where $i \in Tid$, is modeled as event $\langle i, (\mathsf{assume}(l = 0), \{l := i\}) \rangle$; here 0 means the lock is available and thread index i indicates the owner of the lock. Release of lock l is accurately modeled as $\langle i, (\mathsf{assume}(l = i), \{l := 0\}) \rangle$. Similarly, acquire of a counting semaphore cs is modeled using $(\mathsf{assume}(cs > 0), \{cs := cs - 1\})$, while release is modeled using $(\mathsf{assume}(cs \geq 0), \{cs := cs + 1\})$. Fig. 3 shows the symbolic representations of traces in Fig. 1. Note that signal/wait in the second example are modeled using guarded assignments as well. Specifically, `wait(c)` is split into two events t_4 and $t_{4'}$, which first resets c to 0, then waits for c to become non-zero and in the same atomic action resets c back to 0. This modeling conforms to the POSIX standard, allowing t_3 `:signal(c)` to be interleaved in between.

Concurrent Trace Programs. The semantics of an execution trace is defined using a state transition system. Let $V = SV \cup \bigcup_i LV_i$, $1 \leq i \leq k$, be the set of all program variables and Val be a set of values of variables in V. A *state* is a map $s : V \rightarrow Val$ assigning a value to each variable. We also use $s[v]$ and $s[exp]$ to denote the values of $v \in V$ and expression exp in state s. We say that a *state transition* $s \xrightarrow{t} s'$ exists, where s, s' are states and t is an event in thread T_i, $1 \leq i \leq k$, iff

- $t = \langle i, (\mathsf{assume}(c), asgn) \rangle$, $s[c]$ is true, and for each assignment $v := exp$ in $asgn$, $s'[v] = s[exp]$ holds; states s and s' agree on all other variables.

[2] Details on modeling generic language constructs, such as those in C/C++/Java, are not directly related to concurrency; for more information refer to recent efforts in [15,16].

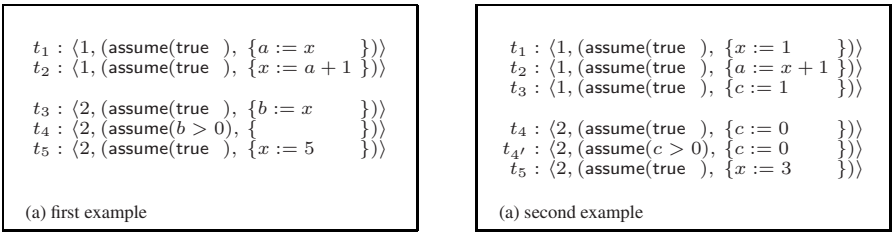

Fig. 3. The symbolic representations of concurrent execution traces

Let $\rho = t_1 \dots t_n$ be an execution trace of program P. Then ρ can be viewed as a total order on the set of symbolic events in ρ. From ρ one can derive a partial order called the concurrent trace program (CTP). Previously, we have used CTPs [17,18] to predict assertion failures and to prune redundant interleavings in stateless model checking.

Definition 1. *The* concurrent trace program *with respect to ρ, denoted CTP_ρ, is a partially ordered set $(T, \sqsubseteq)$ such that,*

- *$T = \{t \mid t \in \rho\}$ is the set of events, and*
- *$\sqsubseteq$ is a partial order such that, for any $t_i, t_j \in T$, $t_i \sqsubseteq t_j$ iff $tid(t_i) = tid(t_j)$ and $i < j$ (in ρ, event t_i appears before t_j).*

Intuitively, CTP_ρ orders events from the same thread by their execution order in ρ; events from different threads are not *explicitly* ordered with each other. In the sequel, we will say $t \in CTP_\rho$ to mean that $t \in T$ is associated with the CTP.

We now define *feasible linearizations* of CTP_ρ. Let $\rho' = t'_1 \dots t'_n$ be a linearization of CTP_ρ, i.e. an interleaving of events of ρ. We say that ρ' is *feasible* iff there exist states $s_0, \dots, s_n$ such that, s_0 is the initial state of the program and for all $i = 1, \dots, n$, there exists a transition $s_{i-1} \xrightarrow{t'_i} s_i$. This definition captures the standard sequential consistency semantics for concurrent programs, where we modeled concurrency primitives such as locks by using auxiliary shared variables.

3 Three-Access Atomicity Violations

An execution trace ρ is *serializable* iff it is equivalent to a feasible linearization ρ' which executes the transactions without other threads interleaved in between. Informally, two traces are equivalent iff we can transform one into another by repeatedly swapping adjacent independent events. Here two events are considered as *independent* iff swapping their execution order always leads to the same program state.

Atomicity Violations. Three-access atomicity violation is a special case of serializability violations, involving an event sequence $t_c \dots t_r \dots t_{c'}$ such that:

1. t_c and $t_{c'}$ are in a transactional block of one thread, and t_r is in another thread;
2. t_c and t_r are data dependent; and t_r and $t_{c'}$ are data dependent.

The recent study in [1] shows that in practice atomicity violations account for a very large number of concurrency errors. Depending on whether each event is a *read* or

write, there are eight combinations of the triplet $t_c, t_r, t_{c'}$. While R-R-R, R-R-W, and W-R-R are serializable, the remaining five may indicate atomicity violations.

Given the CTP_ρ and a transaction $trans = t_i \dots t_j$, where $t_i \dots t_j$ are events from a thread in ρ, we use the set PAV to denote all these potential atomicity violations. Conceptually, the set PAV can be computed by scanning the trace ρ once, and for each remote event $t_r \in CTP_\rho$, finding the two local events $t_c, t_{c'} \in trans$ such that $\langle t_c, t_r, t_{c'} \rangle$ forms a non-serializable pattern.

The crucial problem of deciding whether an event sequence $t_c \dots t_r \dots t_{c'}$ exists in the actual program execution is difficult. However, over-approximate algorithms, such as those based on Lipton's reduction theory [9] or [10,2], can be used to weed out event triplets in PAV that are definitely infeasible. For example, the method in [2] reduces the problem of checking (the existence of) $t_c \dots t_r \dots t_{c'}$ to *simultaneous reachability* under nested locking. That is, does there exist an event $t_{c''}$ such that (1) $t_{c''}$ is within the same thread and is located between t_c and $t_{c'}$ and (2) $t_{c''}, t_r$ are simultaneously reachable? Under nested locking, simultaneous reachability can be decided by a compositional analysis based on locksets and *acquisition histories* [19]. However, the analysis in [2] is over-approximate in that it ignores the data flow and synchronizations other than nested locks[3].

Guarded Independence. Sometimes, two events with data conflict may still be independent with each other, although they are *conflict-dependent*. A data conflict occurs when two events access the same variable and at least one of them is a *write*. In the literature, conflict-independence between two events is defined as: (1) executing one does not enable/disable another, and (2) they do not have data conflict. These conditions are sufficient but not necessary for two events to be *independent*. Consider event t_1:`x=5` and event t_2:`x=5`, for example. They have a data conflict but are semantically independent. Here, we use a more precise *guarded independence* relation as follows (c.f. [20]).

Definition 2. *Two events t_1, t_2 are guarded independent with respect to a condition c_G, denoted $\langle t_1, t_2, c_G \rangle$, iff the guard $c_G(t_1, t_2)$ implies that the following properties:*

1. *if t_1 is enabled in s and $s \xrightarrow{t_1} s'$, then t_2 is enabled in s iff t_2 is enabled in s'; and*
2. *if t_1, t_2 are enabled in s, there is a unique state s' such that $s \overset{t_1 t_2}{\Rightarrow} s'$ and $s \overset{t_2 t_1}{\Rightarrow} s'$.*

The guard c_G is computed by a static traversal of the control flow structure [20]. For each event t, let $V_{RD}(t)$ be the set of variables read by t, and $V_{WR}(t)$ be the set of variables written by t. We define the *potential conflict set* between $t_1, t_2 \in CTP_\rho$ as

$$\mathcal{C}_{t_1,t_2} = V_{RD}(t_1) \cap V_{WR}(t_2) \cup V_{RD}(t_2) \cap V_{WR}(t_1) \cup V_{WR}(t_1) \cap V_{WR}(t_2) \ .$$

For programs with pointers ($*p$) and arrays ($a[i]$), we compute the guarded independence relation R_G as follows:

1. when $\mathcal{C}_{t_1,t_2} = \emptyset$, add $\langle t_1, t_2, true \rangle$ to R_G;
2. when $\mathcal{C}_{t_1,t_2} = \{a[i], a[j]\}$, add $\langle t_1, t_2, i \neq j \rangle$ to R_G;
3. when $\mathcal{C}_{t_1,t_2} = \{*p_i, *p_j\}$, add $\langle t_1, t_2, p_i \neq p_j \rangle$ to R_G;
4. when $\mathcal{C}_{t_1,t_2} = \{x\}$, consider the following cases:

[3] Programs with only nested locking can enforce mutual exclusion, but cannot coordinate thread interactions because nested locks cannot simulate powerful primitives such as semaphores.

a. **RD-WR:** if $x \in V_{RD}(t_1)$ and $x := e$ is in t_2, add $\langle t_1, t_2, x = e\rangle$ to R_G;
b. **WR-WR:** if $x := e_1$ is in t_1 and $x := e_2$ is in t_2, add $\langle t_1, t_2, e_1 = e_2\rangle$ to R_G;
c. **WR-C:** if x is in assume condition $cond$ of t_1, and $x := e$ is in t_2, add $\langle t_1, t_2, cond = cond[x \rightarrow e]\rangle$ to R_G, in which $cond[x \rightarrow e]$ denotes the replacement of x with e.

This set of rules can be easily extended to handle a richer set of language constructs. Note that among these patterns, the syntactic conditions based on data conflict (conflict-independence) is able to catch the first pattern only. Also note that methods in [1,2,10] use conflict-independence (hence *conflict-serializable*), whereas our method is based on guarded independence. In symbolic search based on SMT/SAT solvers, the guarded independence relation can be compactly encoded as constraints in the problem formulation, as described in the next section.

4 Capturing the Feasible Interleavings

Given the CTP_ρ and a set PAV of event triplets as potential atomicity violations, we check whether a violation exists in any feasible linearization of CTP_ρ. For this, we create a formula Φ which is satisfiable iff there exists a feasible linearization of CTP_ρ that exposes the violation. Let $\Phi := \Phi_{CTP_\rho} \wedge \Phi_{AV}$, where Φ_{CTP_ρ} captures all feasible linearizations of CTP_ρ and Φ_{AV} encodes the condition that one event triplet exists.

4.1 Concurrent Static Single Assignment

Our encoding is based on transforming CTP_ρ into a concurrent static single assignment (CSSA) form. Our CSSA form, inspired by [21], has the property that each variable is defined exactly once. Here a *definition* of variable $v \in V$ is an event that modifies v, and a *use* of v is an event where it appears in a condition or in the right-hand side of an assignment. Unlike in the classic sequential SSA form, we need not add ϕ-functions to model the confluence of multiple if-else branches, because in CTP_ρ, each thread has a single control path. All the branching decisions in the program have already been made during the execution that generates the trace ρ in the first place.

We differentiate the shared variables in SV from the thread-local variables in LV_i, $1 \leq i \leq k$. Each use of $v \in LV_i$ corresponds to a unique preceding event in the same thread T_i that defines v. Each use of $v \in SV$, in contrast, may map to multiple definitions in the same or other threads, and a π-function is added to model these definitions.

Definition 3. *A π-function, added for a shared variable v before its use, has the form $\pi(v_1, \ldots, v_l)$, where each v_i, $1 \leq i \leq l$, is either the most recent definition of v in the same thread as the use, or a definition of v in another concurrent thread.*

The construction of the CSSA form consists of the following steps:

1. Create unique names for local/shared variables in their definitions.
2. For each use of a local variable $v \in LV_i$, $1 \leq i \leq k$, replace v with the most recent (unique) definition v'.
3. For each use of a shared variable $v \in SV$, create a unique name v' and add the definition $v' \leftarrow \pi(v_1, \ldots, v_l)$. Then replace v with the new definition v'.

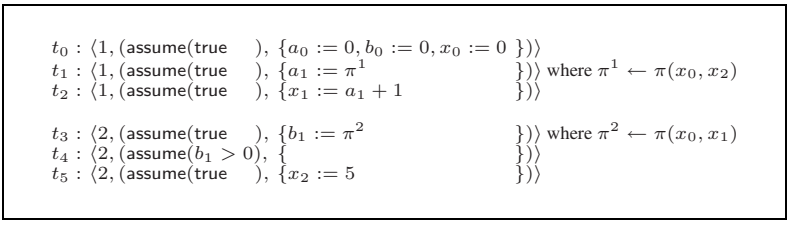

Fig. 4. The CSSA form of the concurrent trace program

Fig. 4 shows the CSSA form of the CTP in Fig. 3(a). Note that event t_0 is added to model the initial values of all variables. We add names π^1 and π^2 for the shared variable uses. The assignment in t_1 becomes $a_1 := \pi^1$ because the value read from x can be defined as either x_0 or x_2, depending on the thread interleaving. The local variable a_1 in t_2, on the other hand, is uniquely defined as in t_1.

The semantics of π-functions are defined as follows. Let $v' \leftarrow \pi(v_1, \ldots, v_l)$ be defined in event t, and let each parameter v_i, $1 \leq i \leq l$, be defined in event t_i. The evaluation of π-function depends on the write-read consistency in a particular interleaving. Intuitively, $(v' = v_i)$ iff v_i is the most recent definition before the use in event t. More formally, $(v' = v_i)$, $1 \leq i \leq l$, iff the following conditions hold,

- event t_i, which defines v_i, is executed before event t; and
- any event t_j that defines v_j, $1 \leq j \leq l$ and $j \neq i$, is executed either before the definition in t_i or after the use in t.

4.2 Encoding Feasible Linearizations

We construct Φ_{CTP_ρ} based on the notion of feasible linearizations (defined in Section 2). It consists of the following subformulas:

$$\Phi_{CTP} := \Phi_{PO} \wedge \Phi_{VD} \wedge \Phi_{PI}\ ,$$

where Φ_{PO} encodes the program order, Φ_{VD} encodes the variable definitions, and Φ_{PI} encodes the π-functions.

To ease the presentation, we use the following notations.

- **Event t_{first}:** we add a dummy event t_{first} to be the first event executed in the CTP.
- **Event t^i_{first}:** for each $i \in Tid$, this is the first event of the thread T_i;
- **Preceding event:** for each event t, we define its thread-local preceding event t' as follows: $tid(t') = tid(t)$ and for any other event $t'' \in CTP$ such that $tid(t'') = tid(t)$, either $t'' \sqsubseteq t'$ or $t \sqsubseteq t''$.
- **HB-constraint:** we use $HB(t, t')$ to denote that event t is executed before t'.

The detailed encoding algorithm is given as follows:

- *Path Conditions.* For each event $t \in CTP_\rho$, we define the path condition $g(t)$ which is true iff t is executed.
  1. If $t = t_{\mathsf{first}}$, or $t = t^i_{\mathsf{first}}$ where $i \in Tid$, let $g(t) := \mathsf{true}$.
  2. Otherwise, let $g(t) := c \wedge g(t')$, where $t' : (\mathsf{assume}(c), asgn)$ is the thread-local preceding event.

- *Program Order (Φ_{PO}).* Φ_{PO} captures the event order within threads. Let $\Phi_{PO} :=$ true initially. For each event $t \in CTP_\rho$,
  1. if $t = t_{\mathsf{first}}$, do nothing;
  2. if $t = t^i_{\mathsf{first}}$, where $i \in Tid$, let $\Phi_{PO} := \Phi_{PO} \wedge HB(t_{\mathsf{first}}, t^i_{\mathsf{first}})$;
  3. otherwise, t has a thread-local preceding event t'; let $\Phi_{PO} := \Phi_{PO} \wedge HB(t', t)$.
- *Variable Definition (Φ_{VD}).* Let $\Phi_{VD} :=$ true initially. For each event $t \in CTP_\rho$,
  1. if t has action $(\mathsf{assume}(c), asgn)$, for each assignment $v := exp$ in $asgn$, let $\Phi_{VD} := \Phi_{VD} \wedge (v = exp)$;
- *The π-Function (Φ_{PI}).* Let $\Phi_{PI} :=$ true initially. For each assignment $v' \leftarrow \pi(v_1, \ldots, v_l)$, where v' is used in event t, and each v_i, $1 \le i \le l$, is defined in event t_i; let

$$\Phi_{PI} := \Phi_{PI} \wedge \bigvee_{i=1}^{l} (v' = v_i) \wedge g(t_i) \wedge HB(t_i, t) \wedge \bigwedge_{j=1, j\neq i}^{l} (HB(t_j, t_i) \vee HB(t, t_j))$$

  This encodes that the π-function evaluates to v_i iff it chooses the i-th definition in the π-set (indicated by $g(t_i) \wedge HB(t_i, t)$), such that any other definition v_j, $1 \le j \le l$ and $j \neq i$, is defined either before t_i, or after this use of v_i in t.

4.3 Encoding Atomicity Violations

Given a set PAV of potential violations, we build formula Φ_{AV} as follows: Initialize $\Phi_{AV} :=$ false. Then for each event triplet $\langle t_c, t_r, t_{c'} \rangle \in PAV$, where t_c and t_r are guarded independent under $c_G(t_c, t_r)$, and t_r and $t_{c'}$ are guarded independent under $c_G(t_r, t_{c'})$, as defined in Section 3, let

$$\begin{aligned}\Phi_{AV} := \Phi_{AV} \vee (\ & g(t_c) \wedge g(t_r) \wedge g(t_{c'}) \wedge \neg c_G(t_c, t_r) \wedge \neg c_G(t_r, t_{c'}) \\ & \wedge HB(t_c, t_r) \wedge HB(t_r, t_{c'})\ )\end{aligned}$$

Recall that for two events t and t', the constraint $HB(t, t')$ denote that t must be executed before t'. Consider a model where we introduce for each event $t \in CTP$ a fresh integer variable $\mathcal{O}(t)$ denoting its position in the linearization (execution time). A satisfiable assignment to Φ_{CTP_ρ} therefore induces values of $\mathcal{O}(t)$, i.e., positions of all events in the linearization. $HB(t, t')$ is defined as follows:

$$HB(t, t') := \mathcal{O}(t) < \mathcal{O}(t')$$

In satisfiability modulo theory, $HB(t, t')$ corresponds to a special subset of *Integer Difference Logic (IDL)*, i.e. $\mathcal{O}(t) < \mathcal{O}(t')$, or simply $\mathcal{O}(t) - \mathcal{O}(t') \le -1$. It is special in that the integer constant c in the IDL constraint $(x - y \le c)$ is always -1. Deciding this fragment of IDL is easier because consistency can be reduced to cycle detection in the constraint graph, which has a linear complexity, rather than the more expensive negative-cycle detection [22].

Fig. 5 illustrates the CSSA-based encoding of CTP in Fig. 4. Note that it is common for many path conditions, variable definitions, and HB-constraints to be constants. For example, $HB(t_0, t_1)$ and $HB(t_0, t_5)$ in Fig. 4 are always true, while $HB(t_5, t_0)$ and $HB(t_1, t_0)$ are always false—such simplifications are frequent and will lead to significant reduction in formula size.

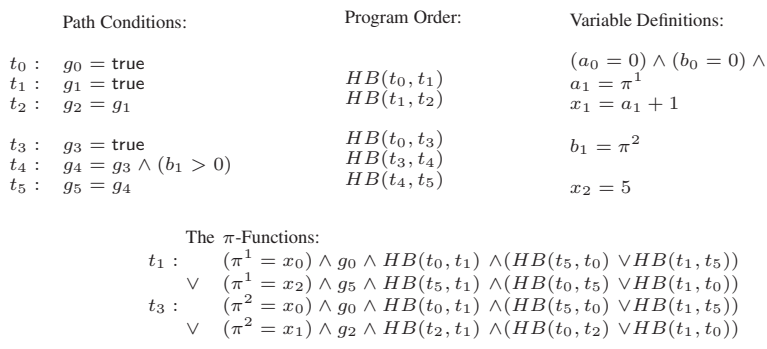

Fig. 5. The CSSA-based encoding of CTP_ρ in Fig. 4

For synchronization primitives such as locks, there are even more opportunities to simplify the formula. For example, if $\pi^1 \leftarrow \pi(l_1, \ldots, l_n)$ denotes the value read from a lock variable l during lock acquire, then we know that $(\pi^1 = 0)$ must hold, since the lock need to be available. This means for non-zero π-parameters, the constraint $(\pi^1 = l_i)$, where $1 \leq i \leq n$, always evaluates to false. And due to the mutex lock semantics, for all $1 \leq i \leq n$, we know $l_i = 0$ iff l_i is defined by a lock release.

The encoding of $\Phi = \Phi_{CTP_\rho} \wedge \Phi_{AV}$ closely follows our definitions of CTP, feasible linearizations, and the semantics of π-functions. We now state its correctness. The proof is straightforward and is omitted for brevity.

Theorem 1. *Formula $\Phi = \Phi_{CTP_\rho} \wedge \Phi_{AV}$ is satisfiable iff there exists a feasible linearization of the CTP that violates the given atomicity property.*

Let n be the number of events in CTP_ρ, let n_π be the number of shared variable uses, let l_π be the maximal number of parameters in any π-function, and let l_{trans} be the number of shared variable accesses in $trans$. We also assume that each event in ρ accesses at most one shared variable. The size of $(\Phi_{PO} \wedge \Phi_{VD} \wedge \Phi_{PI} \wedge \Phi_{AV})$ in the worst case is $O(n + n + n_\pi \times l_\pi^2 + n_\pi \times l_{trans})$. We note that shared variable accesses in typical concurrent programs are often few and far in between, especially when compared to computations within threads, to minimize the synchronization overhead. This means that l_π, n_π, and l_{trans} are typically much smaller than n, which significantly reduces the formula size[4]. In contrast, in conventional bounded model checking (BMC) algorithms for verifying concurrent programs, e.g. [20], which employ an explicit *scheduler variable* at each time frame, the BMC formula size quadratically depends on n, and cannot be easily reduced even if l_π, n_π, and l_{trans} are significantly smaller than n.

5 Capturing Erroneous Trace Prefixes

The algorithm presented so far aims at detecting atomicity violations in all feasible linearizations of a CTP. Therefore, a violation is reported iff (1) a three-access atomicity violation occurs in an interleaving, and (2) the interleaving is a feasible linearization

[4] Our experiments show that l_π is typically in the lower single-digit range (the average is 4).

of CTP_ρ. Sometimes, this may become too restrictive, because the existence of an atomicity violation often leads to the subsequent execution of a branch that is not taken by the given trace ρ (hence the branch is not in CTP_ρ).

Consider the example in Fig. 6. In this trace, event t_4 is guarded by $(a = 1)$. There is a real atomicity violation under thread schedule $t_1t_5t_2\ldots$. However, this trace prefix invalidates the condition $(a = 1)$ in t_3—event t_4 will be skipped. In this sense, the trace $t_1t_5t_2\ldots$ does not qualify as a linearization of CTP_ρ. In our aforementioned symbolic encoding, the π-constraint in t_6 will become invalid.

$$\begin{array}{rl} t_6: & (\pi^2 = x_1) \wedge g_1 \wedge HB(t_1,t_6) \wedge (HB(t_4,t_1) \vee HB(t_6,t_4)) \wedge (HB(t_5,t_1) \vee HB(t_6,t_5)) \\ \vee & (\pi^2 = x_2) \wedge g_4 \wedge HB(t_4,t_6) \wedge (HB(t_1,t_4) \vee HB(t_6,t_1)) \wedge (HB(t_5,t_4) \vee HB(t_6,t_5)) \\ \vee & (\pi^2 = x_3) \wedge g_5 \wedge HB(t_5,t_6) \wedge (HB(t_1,t_5) \vee HB(t_6,t_1)) \wedge (HB(t_4,t_5) \vee HB(t_6,t_4)) \end{array}$$

Note that in the interleaving $t_1t_5t_2\ldots$, we have g_4, $HB(t_4,t_1)$, $HB(t_6,t_4)$, $HB(t_4,t_5)$, $HB(t_6,t_4)$ all evaluated to false. This rules out the interleaving as a feasible linearization of CTP_ρ, although it has exposed a real atomicity violation.

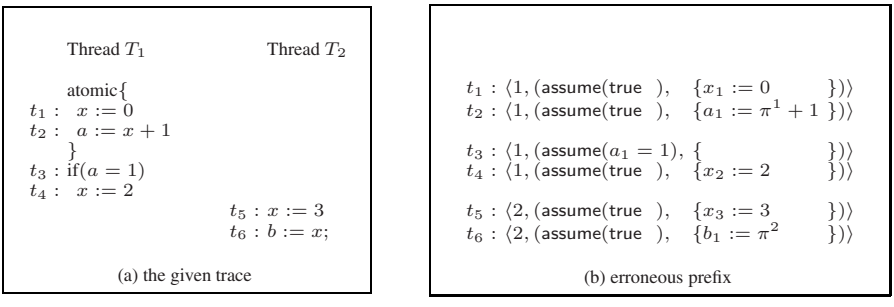

Fig. 6. The atomicity violation leads to a previously untaken branch

We now extend our notion of feasible linearizations of a CTP to all prefixes of its feasible linearizations, or the *feasible linearization prefixes*. The extension is straightforward. Let $\mathsf{FeaLin}(CTP_\rho)$ be the set of feasible linearizations of CTP_ρ. We define the set $\mathsf{FeaPfx}(CTP_\rho)$ of feasible linearization prefixes as follows:

$$\mathsf{FeaPfx}(CTP_\rho) := \{w \mid w \text{ is a prefix of } \rho' \in \mathsf{FeaLin}(CTP_\rho)\}$$

We extend our symbolic encoding to capture these erroneous trace prefixes (as opposed to entire erroneous traces). We extend the symbolic encoding in Section 4 as follows. Let event triplet $\langle t_c, t_r, t_{c'}\rangle \in PAV$ be the potential violation. We modify the construction of Φ_{PI} (for the π-function in event t) as follows:

$$\begin{array}{l} \Phi_{PI} := \Phi_{PI} \wedge (\ HB(t_{c'}, t) \vee \\ \qquad \bigvee_{i=1}^{l} (v' = v_i) \wedge g(t_i) \wedge HB(t_i, t) \wedge \bigwedge_{j=1, j\neq i}^{l} (HB(t_j, t_i) \vee HB(t, t_j))) \end{array}$$

That is, if the atomicity violation has already happened in some prefix, as indicated by $HB(t_{c'}, t)$, i.e. when the event t associated with this π-function happens after $t_{c'}$, then we do not enforce any read-after-write consistency. Otherwise, read-after-write consistency is enforced as before, as shown in the second line in the formula above. The rest of the encoding algorithm remains the same. We now state the correctness of this encoding extension. The proof is straightforward and is omitted for brevity.

Theorem 2. *Formula $\Phi = \Phi_{CTP_\rho} \wedge \Phi_{AV}$ is satisfiable iff there exists a feasible linearization prefix of the CTP that violates the given atomicity property.*

6 Experiments

We have implemented the proposed algorithm in a tool called *Fusion*. Our tool is capable of handling execution traces generated by multi-threaded C programs using the Linux *PThreads* library. We use CIL [23] for instrumenting the C source code and use the *Yices* SMT solver [14] to solve the satisfiability formulas. Our experiments were conducted on a PC with 1.6 GHz Intel processor and 2GB memory running Fedora 8.

We have conducted preliminary experiments using the following benchmarks[5]: The first set of examples mimic two concurrency bug patterns from the Apache web server code (c.f. [1]). The original programs, *atom001* and *atom002*, have atomicity violations. We generated two additional programs, *atom001a* and *atom002a*, by adding code to the original programs to remove the violations. The second set of examples are Linux/Pthreads/C implementation of the parameterized *bank* example [24]. We instantiate the program with the number of threads being 2,3,. . .. The original programs (*bank-av*) have nested locks as well as shared variables, and have known bugs due to atomicity violations. We provided two different fixes, one of which (*bank-nav*) removes all atomicity violations while another (*bank-sav*) removes some of them. We used both condition variables and additional shared variables in our fixes. Although the original programs (*bank-av*) does not show the difference in the quality of various prediction methods (because violations detected by ignoring data and synchronizations are actually feasible), the precision differences show up on the programs with fixes. In these cases, some atomicity violations no longer exist, and yet methods based on over-approximate predictive models would still report violations.

Table 1. Experimental results of predicting atomicity violations

Test Program			The Given Trace		Symbolic Analysis				w/o Data [2]
name	thrds	svars	simplify/ original	regions	orig-pavs	hb-pavs	sym-avs	sym-time (s)	pavs
atom001	3	14	50 / 88	1	8	2	1	0.03	1
atom001a	3	16	58 / 100	1	8	2	**0**	0.03	1
atom002	3	24	349 / 462	1	212	34	33	20.4	33
atom002a	3	26	359 / 462	1	212	34	**0**	17.6	33
bank-av-2	3	109	278 / 748	2	24	8	8	0.1	8
bank-av-4	5	113	527 / 1213	4	48	16	16	0.6	16
bank-av-6	7	117	770 / 1672	6	72	24	24	2.3	24
bank-av-8	9	121	1016 / 2134	8	96	32	32	2.5	32
bank-sav-2	3	119	337 / 852	2	24	8	**4**	0.2	8
bank-sav-4	5	123	642 / 1410	4	48	16	**8**	0.9	16
bank-sav-6	7	127	941 / 1960	6	72	24	**12**	3.8	24
bank-sav-8	9	131	1243 / 2517	8	96	32	**16**	4.6	32
bank-nav-2	3	119	341 / 856	2	24	8	**0**	0.2	8
bank-nav-4	5	123	647 / 1414	4	48	16	**0**	0.2	16
bank-nav-6	7	127	953 / 1972	6	72	24	**0**	3.7	24
bank-nav-8	9	131	1163 / 2362	8	96	32	**0**	140.6	32

[5] Examples are available at http://www.nec-labs.com/∼chaowang/pubDOC/atom.tar.gz

Table 1 shows the experimental results. The first three columns show the statistics of test cases, including the program name, the number of threads, and the number of shared variables that are accessed in the given trace. The next two columns show the length of the trace, in both the original and the simplified versions, and the number of transactions (*regions*). Our simplification consists of trace-based program slicing, dead variable removal, and constant folding; furthermore, variables defined as global, but not accessed by more than one thread in the given trace, are not counted as shared in the table (*svars*). The next four columns show the statistics of our symbolic analysis, including the size of PAV (*orig-pavs*), the number of violations after pruning using a simple static must-happen-before analysis (*hb-pavs*), the number of real violations (*sym-avs*) reported by our symbolic analysis, and the runtime in seconds. In the last column, we provide the number of (potential) atomicity violations if we ignore the data flow and synchronizations other than nested locking.

The results show that, if one relies on only static analysis, the number of reported violations (in *orig-pavs*) is often large, even for a prediction based on a single trace. Our simple must-happen-before analysis utilizes the semantics of thread *create* and *join*, and seems effective in pruning away event triplets that are definitely infeasible. In addition, if one utilizes the nested locking semantics, as in *w/o Data* [2], more spurious event triplets can be pruned away. However, note that the number of remaining violations can still be large. In contrast, our symbolic analysis prunes away all the spurious violations and reports much fewer atomicity violations. For each violation that we report, we also produce a concrete execution trace exposing the violation. This *witness* trace can be used by the thread scheduler in *Fusion*, to re-run the program and replay the actual violation. We also note that the runtime overhead of our symbolic analysis is modest. The algorithm can be used in the context of a post-mortem analysis.

7 Related Work

We have mentioned in Section 1 some of the static methods [3,4], runtime monitoring [5,1,6,7,8], and runtime prediction [9,2,10,11,12,13] for detecting atomicity violations. Lu *et al.* [1] used access interleaving invariants to capture patterns of test runs and then monitor production runs for detecting three-access atomicity violations. Xu *et al.* [5] used a variant of the two-phase locking algorithm to monitor and detect serializability violations. Both methods were aimed at detecting, not predicting, errors in the given trace. In [4], Farzan and Madhusudan introduced the notion of *causal atomicity* in a static program analysis focusing on the control paths; subsequently they used execution traces for predicting atomicity violations [10,2]. Wang and Stoller [6] also studied the prediction of serializability violations under the assumptions of deadlock-freedom and nested locking; their algorithms are precise for checking violations involving one or two transactions but incomplete for checking arbitrary runs.

Our symbolic encoding for detecting atomicity violations is related to, but is different from, the SSA-based SAT encoding [15], which is popular for *sequential* programs. Our analysis differs from the context-bounded analysis in [25,26,16] since they *a priori* fix the number of context switches in order to reduce concurrent programs to sequential programs. In contrast, our method in Section 4 is for the unbounded case, although context-bounding constraints may be added to further improve performance. We directly capture the partial order in *difference logic*, therefore differing from

CheckFence [27], which explicitly encodes ordering between all pairs of events in pure Boolean logic. In [28], a non-standard synchronous execution model is used to schedule multiple events simultaneously whenever possible instead of using the standard interleaving model. Furthermore, all the aforementioned methods were applied to whole programs and not to concurrent trace programs (CTPs). In previous works [17,18] we have used the notion of CTP, but the context was stateless model checking to prune redundant interleavings in the former, and predicting assertion failures in the later.

The quantifier-free formulas produced by our encoding are decidable due to the finite size of the CTP. When non-linear arithmetic operations appear in the symbolic execution trace, they are treated as bit-vector operations. This way, the rapid progress in SMT solvers can be directly utilized to improve performance in practice. In the presence of unknown functions, trace-based abstraction techniques as in [29], which uses concrete parameter/return values to model library functions, are employed to derive the predictive model, while ensuring that the analysis results remain precise.

8 Conclusions

In this paper, we propose a symbolic algorithm for detecting three-access atomicity violations in all feasible interleavings of events in a given execution trace. The new algorithm uses a succinct encoding to generate an SMT formula such that the violation of an atomicity property exists iff the SMT formula is satisfiable. It does not report bogus errors and at the same time achieves a better interleaving coverage than existing methods for predictive analysis.

References

1. Lu, S., Tucek, J., Qin, F., Zhou, Y.: AVIO: detecting atomicity violations via access interleaving invariants. In: Architectural Support for Programming Languages and Operating Systems, pp. 37–48 (2006)
2. Farzan, A., Madhusudan, P.: Meta-analysis for atomicity violations under nested locking. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification. LNCS, vol. 5643, pp. 248–262. Springer, Heidelberg (2009)
3. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: Programming Language Design and Implementation, pp. 338–349 (2003)
4. Farzan, A., Madhusudan, P.: Causal atomicity. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 315–328. Springer, Heidelberg (2006)
5. Xu, M., Bodík, R., Hill, M.D.: A serializability violation detector for shared-memory server programs. In: Programming Language Design and Implementation, pp. 1–14 (2005)
6. Wang, L., Stoller, S.D.: Runtime analysis of atomicity for multithreaded programs. IEEE Trans. Software Eng. 32(2), 93–110 (2006)
7. Farzan, A., Madhusudan, P.: Monitoring atomicity in concurrent programs. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 52–65. Springer, Heidelberg (2008)
8. Flanagan, C., Freund, S.N., Yi, J.: Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In: PLDI, pp. 293–303 (2008)
9. Flanagan, C., Freund, S.N.: Atomizer: A dynamic atomicity checker for multithreaded programs. In: Parallel and Distributed Processing Symposium (IPDPS). IEEE, Los Alamitos (2004)
10. Farzan, A., Madhusudan, P.: The complexity of predicting atomicity violations. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 155–169. Springer, Heidelberg (2009)

11. Chen, F., Rosu, G.: Parametric and sliced causality. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 240–253. Springer, Heidelberg (2007)
12. Serbănuţă, T.F., Chen, F., Rosu, G.: Maximal causal models for multithreaded systems. Technical Report UIUCDCS-R-2008-3017, University of Illinois at Urbana-Champaign (2008)
13. Sadowski, C., Freund, S.N., Flanagan, C.: Singletrack: A dynamic determinism checker for multithreaded programs. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 394–409. Springer, Heidelberg (2009)
14. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for dpll(t). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
15. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
16. Lahiri, S., Qadeer, S., Rakamaric, Z.: Static and precise detection of concurrency errors in systems code using SMT solvers. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification. LNCS, vol. 5643, pp. 509–524. Springer, Heidelberg (2009)
17. Wang, C., Chaudhuri, S., Gupta, A., Yang, Y.: Symbolic pruning of concurrent program executions. In: Foundations of Software Engineering, pp. 23–32. ACM, New York (2009)
18. Wang, C., Kundu, S., Ganai, M., Gupta, A.: Symbolic predictive analysis for concurrent programs. In: International Symposium on Formal Methods, pp. 256–272. ACM, New York (2009)
19. Kahlon, V., Ivancic, F., Gupta, A.: Reasoning about threads communicating via locks. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 505–518. Springer, Heidelberg (2005)
20. Wang, C., Yang, Z., Kahlon, V., Gupta, A.: Peephole partial order reduction. In: Tools and Algorithms for Construction and Analysis of Systems, pp. 382–396. Springer, Heidelberg (2008)
21. Lee, J., Padua, D., Midkiff, S.: Basic compiler algorithms for parallel programs. In: Principles and Practice of Parallel Programming, pp. 1–12 (1999)
22. Wang, C., Gupta, A., Ganai, M.: Predicate learning and selective theory deduction for a difference logic solver. In: Design Automation Conference, pp. 235–240. ACM, New York (2006)
23. Necula, G., McPeak, S., Rahul, S., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of c programs. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
24. Farchi, E., Nir, Y., Ur, S.: Concurrent bug patterns and how to test them. In: Parallel and Distributed Processing Symposium, p. 286 (2003)
25. Rabinovitz, I., Grumberg, O.: Bounded model checking of concurrent programs. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 82–97. Springer, Heidelberg (2005)
26. Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 37–51. Springer, Heidelberg (2008)
27. Burckhardt, S., Alur, R., Martin, M.: CheckFence: checking consistency of concurrent data types on relaxed memory models. In: PLDI, pp. 12–21. ACM, New York (2007)
28. Jussila, T., Heljanlo, K., Niemelä, I.: BMC via on-the-fly determinization. STTT 7(2), 89–101 (2005)
29. Beckman, N., Nori, A.V., Rajamani, S.K., Simmons, R.J.: Proofs from tests. In: International Symposium on Software Testing and Analysis, pp. 3–14. ACM, New York (2008)