Symbolic Predictive Analysis for Concurrent Programs

Chao Wang[1], Sudipta Kundu[2], Rhishikesh Limaye[3], Malay Ganai[1], and Aarti Gupta[1]

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

Abstract. Predictive analysis aims at detecting concurrency errors during runtime by monitoring a concrete execution trace of a concurrent program. In recent years, various models based on the happens-before causality relations have been proposed for predictive analysis. However, these models often rely on only the observed runtime events and typically do not utilize the program source code. Furthermore, the enumerative algorithms they use for verifying safety properties in the predicted traces often suffer from the interleaving explosion problem. In this paper, we introduce a precise predictive model based on both the program source code and the observed execution events, and propose a symbolic algorithm to check whether a safety property holds in all feasible permutations of events of the given trace. Rather than explicitly enumerating and checking the interleavings, our method conducts the search using a novel encoding and symbolic reasoning with a satisfiability modulo theory solver. We also propose a technique to bound the number of context switches allowed in the interleavings during the symbolic search, to further improve the scalability of the algorithm.

Keywords: Concurrent trace program; Predictive analysis; Happens-before; Context bounding; SMT; SAT

1. Introduction

Concurrent programs are becoming pervasive in the era of multi-processor and multi-core systems. However, our ability to effectively harness the power of parallelism is predicated upon advances in tools for verifying and debugging concurrent programs. Due to the inherent nondeterminism in scheduling concurrent events, executing a program with the same test data input may lead to different behaviors. This poses a significant challenge in testing – even if a test input may cause a failure, the erroneous thread interleaving manifesting the failure may not be executed during testing. Furthermore, merely executing the same test multiple times does not always increase the interleaving coverage. Predictive analysis aims at detecting concurrency errors by observing execution traces of a concurrent program which themselves may be non-erroneous. In predictive analysis, a concrete execution trace is given together with a correctness property, e.g. in the form of embedded assertions or generic correctness properties such as atomicity and data-race freedom. Sometimes the given execution trace does not violate the property, but there exists an alternative trace, i.e., a

Correspondence and offprint requests to: Chao Wang, NEC Laboratories America, Princeton, NJ 08540, USA. E-mail: chaowang@nec-labs.com

feasible permutation of events of the given trace, that violates the property. The goal of predictive analysis is to detect such erroneous traces by *statically* analyzing the given execution trace without re-executing the program.

Predictive analysis complements model checking and other static program analysis techniques. Due to the extremely large number of possible thread interleavings of most realistic concurrent programs, even for a given test input, it is practically infeasible to inspect all possible execution traces precisely as in model checking. While over-approximated static analysis may be effective in inferring invariants and proving correctness, when being used to detect bugs in real world applications, they often suffer from overwhelmingly many bogus warnings. In predictive analysis, the number of false alarms can be significantly reduced or even eliminated because the analysis is now focused on a concrete execution trace generated by running the program in its target environment.

Existing predictive analysis methods in the literature can be classified broadly into two categories based on the quality of their reported bugs. The first category consists of methods that do not miss real errors but may report bogus errors. Historically, methods based on the lockset analysis (e.g. [SBN+97, FF04, WS06]) fall into the first category. These methods strive to cover all the feasible interleavings of the given trace, but may also introduce many bogus interleavings. (We note that some of the lockset based methods can be both unsound and incomplete.) The second category consists of methods that rely on the various causal models (e.g. [SRA05, CR07, SCR08]), with many of them inspired by the happens-before causality relation first introduced by Lamport [Lam78]. These methods often provide a *feasibility guarantee* – that all the reported bugs can actually happen, but they may not cover all the detectable errors.

In this paper, we focus on predictive analysis with the feasibility guarantee. In this context, we view the given trace as a total order of the executed events and the causal model as a partial order, admitting not only the given trace but also many alternative interleavings. There are two challenging problems. First, the existing causal models often do not assume that source code is available, and therefore rely on observing only the *concrete events* during execution. In a concrete event, typically the values read from or written to shared memory locations are available, whereas the actual program statement that produces the event is not known. Consequently, unnecessarily strong causality constraints may be imposed to achieve the desired feasibility guarantee, leading to many missed errors. Second, enumerating all the feasible interleavings allowed by a given causal model for property checking is computationally expensive. Despite the long quest for more precise causal models, little progress has been made to improve the underlying verification algorithms. The currently dominating method of explicitly enumerating interleavings, e.g. in [SRA05, CR07, SCR08], does not scale well since the number of interleavings can be astronomically large.

We propose a *symbolic* predictive analysis to address these challenging problems. First, we assume that the program source code is available in addition to the sequence of events observed at runtime. We define a predictive model, called the Concurrent Trace Program (CTP), based on information from both the program source code and the observed execution. This new model covers provably more interleavings than the existing causal models in the literature (e.g. [SRA05, CR07, SCR08]), and facilitates an efficient constraint based analysis. That is, various synchronization primitives and concurrency semantics, such as locks, semaphores, happens-before, and sequential consistency, are all handled easily and uniformly. More specifically, we make the following contributions:

- We introduce a predictive model, called CTP, to capture all the feasible interleavings that can be predicted from a given execution trace.
- We propose a verification algorithm based on encoding the program semantics and the properties as a set of first-order logic formulas and solving these formulas using a satisfiability modulo theory (SMT) solver. This SMT-based symbolic search is often more efficient than explicit enumeration of interleavings since it performs property-driven pruning by leveraging conflict analysis and other learning features in the modern SMT solvers.
- To further improve the efficiency of symbolic search, we propose an encoding based method to bound the number of context switches allowed in any predicted interleavings.

If desired, our predictive model can be constrained or relaxed to match known (both sound and unsound) methods in the literature. This is because CTP is the precise model in the context of trace-based predictive analysis, whereas these other models are either over-approximated or under-approximated. For example, by constraining the CTP and hence removing some of its allowed interleavings, we can degenerate it to any sound causal models such as the one in [SCR08]. This also demonstrates the fact our model covers more interleavings and therefore can predict more bugs.

Finally, although our SMT-based symbolic analysis is designed for checking properties in the CTP, it can easily be extended to the more general application settings. In fact, the symbolic encoding is applicable to any structurally bounded concurrent program, where each thread is a loop-free, recursion-free, sequential program, but with possibly many branching statements and execution paths.

The remainder of this paper is organized as follows. In Section 2, we use a motivating example to illustrate the idea of symbolic predictive analysis. In Section 3, we formally define execution traces and our CTP model. In Sec-

tion 4, we present our symbolic algorithm for checking assertion properties. In Section 5, we extend it for checking atomicity violations, data races, and other concurrency errors. In Section 6, we show that the idea of context-bounding to (unsoundly) reduce the computational overhead can be implemented during symbolic encoding. In Section 7, we demonstrate how to degenerate the CTP to match some known models in the literature. In Section 8, we extend our encoding to the more general setting of structurally bounded concurrent programs. We present the experimental results in Section 9, review the related work in Section 10, and conclude in Section 11.

2. Motivating Example

Fig. 1 shows an execution trace of a multithreaded C program, modified from an example in [SCR08]. There are two concurrent threads T_1 and T_2, three shared variables x, y and z, two local variables a and b, and a semaphore l. The counting semaphore l can be viewed as an integer variable initialized to value 1. Executing $acq(l)$ allows a thread to acquire the semaphore when $(l > 0)$ and decrease l by one, while executing $rel(l)$ allows a thread to increase l by one. The initial program state is $x = y = 0$. The sequence $\rho = t_1$–$t_{11}t_{13}$ of statements denotes the execution order of the given trace. The property in this case is specified in t_{12} as an assertion. The given trace ρ does not violate this assertion. However, a *feasible* permutation of this trace, $\rho' = (t_1$–$t_4)t_9t_{10}t_{11}t_{12}t_{13}(t_5$–$t_8)$, exposes the error.

To our knowledge, none of the *sound* causal models in the literature, including [Lam78, SRA05, CR07, SCR08], can predict this error. By sound, what we mean is that the model does not produce bogus errors (most lockset based algorithms are not sound). For instance, if Lamport's happens-before causality is used to define the feasible trace permutations of ρ, the execution order of all *read-after-write* event pairs in ρ must be respected. In particular, it means that event t_8 must be executed before t_{10} and event t_7 must be executed before t_{11}. This is a sufficient, but often not necessary, condition to ensure that all the admitted trace permutations are feasible – many other feasible interleavings are left out. Although the subsequent models (e.g. [SRA05, CR07, SCR08]) lifted some of these constraints without jeopardizing the feasibility guarantee, they still could not predict the error in ρ'. Consider the *maximal causal model* in [SCR08], which relies on the axioms of semaphore and sequential consistency and is general enough to subsume the other causal models. This model allows all the classic happens-before constraints to be lifted, except the one stating that t_7 must happen before t_{11}. This is because changing the order of two data conflicting events may lead to different program states. Unfortunately, in order to predict trace ρ', one must lift this remaining constraint.

The fundamental reason why existing causal models cannot predict the error in Fig. 1 is that, they model events in ρ as concrete values read from, or written to, shared variables. Such *concrete* events are tied closely to a particular execution. Consider t_{11} `:if(x>b)`, for instance; it is regarded as an event that *reads value* 1 *from variable* x. This is a partial interpretation because other program statements, such as `if(b>x)` or `if(x>1)` or even `b:=x`, may produce the same event. If the model needs to guarantee that all reported bugs are real, then unnecessarily strong happens-before constraints may need to be imposed over concrete events like t_{11} to ensure the feasibility of all the admitted traces, regardless of what statement produces the event.

In contrast, we model the execution trace as a sequence of *symbolic events* by considering the program statements that produce ρ and by capturing the abstract values (e.g. relevant predicates). For instance, we model event t_{11} as $\mathsf{assume}(x > b)$, where $\mathsf{assume}(c)$ means condition c holds when the event is executed, indicating that t_{11} is produced by a branching statement and $(x > b)$ is the condition taken. Furthermore, we do not use the classic happens-before causality relations to define the set of predicted traces. Instead, we require that all interleavings of these symbolic events must satisfy the sequential consistency semantics of the concrete program. Essentially, sequential consistency requires that each read of a shared variable gets the value set by the most recent write. In the running example, it is possible to move symbolic events t_9–t_{12} ahead of t_5–t_8 while still maintaining the sequential consistency. As a result, our new algorithm, while maintaining the feasibility guarantee, is capable of predicting the erroneous behavior in ρ'.

3. Preliminaries

In this section, we define programs, execution traces, and concurrent trace programs. Concurrent trace programs are our models for symbolic predictive analysis.

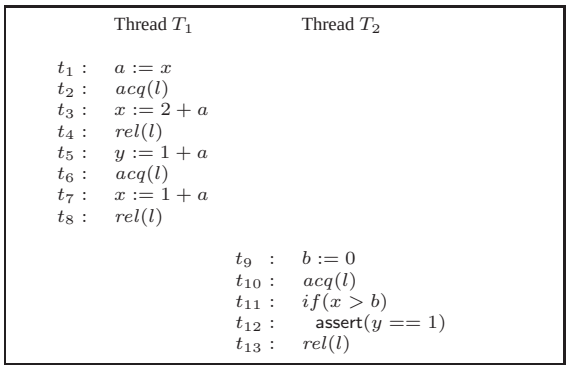


Fig. 1. A sequence of executed statements (x=y=0 initially)

t_1 : $\langle 1, (\mathsf{assume}(\mathsf{true}\ ), \{a := x\})\rangle$
t_2 : $\langle 1, (\mathsf{assume}(l > 0), \{l := l - 1\})\rangle$
t_3 : $\langle 1, (\mathsf{assume}(\mathsf{true}\ ), \{x := 2 + a\})\rangle$
t_4 : $\langle 1, (\mathsf{assume}(\mathsf{true}\ ), \{l := l + 1\})\rangle$
t_5 : $\langle 1, (\mathsf{assume}(\mathsf{true}\ ), \{y := 1 + a\})\rangle$
t_6 : $\langle 1, (\mathsf{assume}(l > 0), \{l := l - 1\})\rangle$
t_7 : $\langle 1, (\mathsf{assume}(\mathsf{true}\ ), \{x := 1 + a\})\rangle$
t_8 : $\langle 1, (\mathsf{assume}(\mathsf{true}\ ), \{l := l + 1\})\rangle$

t_9 : $\langle 2, (\mathsf{assume}(\mathsf{true}\ ), \{b := 0\})\rangle$
t_{10} : $\langle 2, (\mathsf{assume}(l > 0), \{l := l - 1\})\rangle$
t_{11} : $\langle 2, (\mathsf{assume}(x > b), \{\ \})\rangle$
t_{12} : $\langle 2, (\mathsf{assert}(y = 1)\ )\rangle$
t_{13} : $\langle 2, (\mathsf{assume}(\mathsf{true}\ ), \{l := l + 1\})\rangle$

Fig. 2. The symbolic events of the given trace (x=y=0 initially)

3.1. Programs and Execution Traces

A concurrent program has a finite set of *threads* and a finite set SV of *shared* variables. Each thread T_i, where $1 \leq i \leq k$, has a finite 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 the concurrent program, including the control flow and the expression syntax, are intentionally left unspecified in order to be more general. Instead, we directly define the symbolic execution traces.

A symbolic execution trace of a program is a finite sequence of events $\rho = t_1 \ldots t_n$. An *event* t is a tuple $\langle tid, action\rangle$ where $tid \in Tid$ is a thread index and $action$ is an atomic computation. An action in thread T_i may have one of the following types:

- $(\mathsf{assume}(c), asgn)$ is the atomic *guarded assignment* action, 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.
- $\mathsf{assert}(c)$ is the assertion action. The conditional expression c over V_i must be true when the event is executed; otherwise, an error is raised.

Each event in the execution trace is unique. If a statement in the textual representation of the program is executed again, e.g., when it is inside a loop or a recursive function or a routine executed by multiple threads, a new event will be created to represent this new execution instance. By defining the expression syntax suitably, this symbolic trace representation can model the execution of any shared-memory multithreaded program. Details on modeling generic language constructs in C or Java are omitted since they are not directly related to concurrency; for more information on language modeling please refer to recent efforts in [WCGY09, IYS+05] and [CKL04, LQ08].

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

Synchronization Primitives. We use the guarded assignments in our implementation to model synchronization primitives in POSIX threads (and Java). This includes mutex locks, semaphores, condition variables, barriers, etc. For example, the acquire of 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. The release of lock l is modeled as $\langle i, (\mathsf{assume}(l = i), \{l := 0\})\rangle$. Similarly, the acquire of counting semaphore cs is modeled as $(\mathsf{assume}(cs > 0), \{cs := cs - 1\})$, and the release is modeled as $(\mathsf{assume}(cs \geq 0), \{cs := cs + 1\})$. Action $\mathsf{assert}(c)$ specifies the correctness property, and it corresponds to the assertion function in the standard C library.

Example. Fig. 2 shows an example symbolic execution trace, which corresponds to trace ρ in Fig. 1. Note that the synchronization primitive $acq(l)$ in t_2 is modeled as an atomic guarded assignment action. The normal assignment in

t_1 is modeled as a guarded assignment with $\mathsf{assume}(\mathsf{true})$. The if-statement in t_{11} is modeled as a guarded assignment with $asgn$ being an empty set.

3.2. Concurrent Trace Programs

The semantics of a symbolic 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 $s : V \to Val$ 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 iff one of the following conditions holds:

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

Let $\rho = t_1 \dots t_n$ be an execution trace of program P. It defines a total order on the set of events. From this total order, we can derive a partial order called the concurrent trace program.

Definition 3.1. The *Concurrent Trace Program (CTP)* derived from ρ, denoted CTP_ρ, is a partial order $(T, \sqsubseteq)$ such that,

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

In the sequel, we say $t \in CTP_\rho$ if $t \in T$ and T is associated with that CTP. 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. Keeping events symbolic and events from different threads un-ordered is the crucial difference between CTP and the existing causal models [Lam78, SRA05, CR07, SCR08]. Consider events t_7 and t_{11} in Fig. 1 again. Our CTP model allows t_{11} to be executed before t_7 since the new interleavings can still obey the sequential consistency semantics, but none of the existing causal models would allow this movement, merely because t_7 conflicts with t_{11} and is also executed before t_{11} in ρ.

We guarantee the feasibility of all the predicted traces in CTP_ρ through the notion of *feasible* linearization. In this context, a linearization of the partial order ($\sqsubseteq$) represents an interleaving of the events in ρ. Some of these linearizations may not correspond to any actual program execution; these are called the *infeasible* linearizations. During symbolic predictive analysis, we want to consider only the feasible linearizations. Let $\rho' = t'_1 \dots t'_n$ be a linearization of CTP_ρ. We say that ρ' is a *feasible* linearization 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 a concurrent program. Note that at this stage, all the synchronization primitives, such as those on mutex locks, are modeled using auxiliary variables and guarded assignments.

4. Symbolic Predictive Analysis

Given an execution trace ρ, we derive the model CTP_ρ and *symbolically* check all its feasible linearizations for property violations. For this, we create a formula Φ_{CTP_ρ} such that Φ_{CTP_ρ} is satisfiable iff there exists a feasible linearization of CTP_ρ that violates the property. Specifically, we use a novel encoding to create a satisfiability formula in a subset of quantifier-free first-order logic to facilitate the application of an off-the-shelf SMT solver.

4.1. Concurrent Static Single Assignment

Our encoding is based on transforming the CTP into a concurrent static single assignment (CSSA) form, inspired by [LPM99]. The CSSA form 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 an expression. In our case, an event defines v iff v appears in the left-hand side of an assignment; an event uses v iff v appears in a condition (of assume or assert) or the right-hand side of an assignment. Unlike in the classic sequential SSA form, we need not add ϕ-functions

t_0 : $\langle 1, (\mathsf{assume}(\mathsf{true}\ ), \{x_0 := 0, y_0 := 0, l_0 := 1\}) \rangle$
t_1 : $\langle 1, (\mathsf{assume}(\mathsf{true}\ ), \{a_1 := \pi^1\}) \rangle$ where $\pi^1 \leftarrow \pi(x_0)$
t_2 : $\langle 1, (\mathsf{assume}(\pi^2 > 0\,), \{l_1 := \pi^2 - 1\}) \rangle$ where $\pi^2 \leftarrow \pi(l_0, l_5, l_6)$
t_3 : $\langle 1, (\mathsf{assume}(\mathsf{true}\ ), \{x_1 := 2 + a_1\}) \rangle$
t_4 : $\langle 1, (\mathsf{assume}(\mathsf{true}\ ), \{l_2 := \pi^3 + 1\}) \rangle$ where $\pi^3 \leftarrow \pi(l_1, l_5, l_6)$
t_5 : $\langle 1, (\mathsf{assume}(\mathsf{true}\ ), \{y_1 := 1 + a_1\}) \rangle$
t_6 : $\langle 1, (\mathsf{assume}(\pi^4 > 0\,), \{l_3 := \pi^4 - 1\}) \rangle$ where $\pi^4 \leftarrow \pi(l_2, l_5, l_6)$
t_7 : $\langle 1, (\mathsf{assume}(\mathsf{true}\ ), \{x_2 := 1 + a_1\}) \rangle$
t_8 : $\langle 1, (\mathsf{assume}(\mathsf{true}\ ), \{l_4 := \pi^5 + 1\}) \rangle$ where $\pi^5 \leftarrow \pi(l_3, l_5, l_6)$

t_9 : $\langle 2, (\mathsf{assume}(\mathsf{true}\ ), \{b_1 := 0\}) \rangle$
t_{10} : $\langle 2, (\mathsf{assume}(\pi^6 > 0\,), \{l_5 := \pi^6 - 1\}) \rangle$ where $\pi^6 \leftarrow \pi(l_0, l_1, l_2, l_3, l_4)$
t_{11} : $\langle 2, (\mathsf{assume}(\pi^7 > b_1), \{\ \}) \rangle$ where $\pi^7 \leftarrow \pi(x_0, x_1, x_2)$
t_{12} : $\langle 2, (\mathsf{assert}(\pi^8 = 1)\,) \rangle$ where $\pi^8 \leftarrow \pi(y_0, y_1)$
t_{13} : $\langle 2, (\mathsf{assume}(\mathsf{true}\ ), \{l_6 := \pi^9 + 1\}) \rangle$ where $\pi^9 \leftarrow \pi(l_0, l_1, l_2, l_3, l_4, l_5)$

Fig. 3. The CSSA form of the concurrent trace program in Fig. 2

to model the confluence of multiple if-else branches because in a concurrent trace program, each thread has a single control path. The branching decisions have already been made during program execution resulting in the trace ρ.

We differentiate shared variables in SV from local variables in LV_i, $1 \leq i \leq k$. Each use of variable $v \in LV_i$ corresponds to a unique definition, i.e. a preceding event in the same thread T_i that defines v. For shared variables, however, each use of variable $v \in SV$ may map to multiple definitions due to thread interleaving. A π-function is added to model the confluence of these possible definitions.

Definition 4.1. A π-function, introduced for a shared variable v immediately 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.

Therefore, the construction of CSSA 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'.

Example. Fig. 3 shows the CSSA form of the CTP in Fig. 2. We add new names π^1–π^9 and π-functions for the shared variable uses. The condition $(x > b)$ in t_{11} becomes $(\pi^7 > b_1)$ where $\pi^7 \leftarrow \pi(x_0, x_1, x_2)$ denotes the current value of shared variable x and b_1 denotes the value of local variable b defined in t_9. The names x_0, x_1, x_2 denote the values of x defined in t_0, t_3 and t_7, respectively. Event t_0 is added to model the initial values of the variables.

Semantics of π-Functions. Let $v' \leftarrow \pi(v_1, \ldots, v_l)$ be defined in event t, and each v_i, $1 \leq i \leq l$, be defined in event t_i. The π-function may return any of the parameters as the result depending on the write-read consistency in a particular interleaving. Intuitively, $(v' = v_i)$ in an interleaving iff v_i is the most recent definition before event t. More formally, $(v' = v_i)$, $1 \leq i \leq l$, holds iff the following conditions hold,

- event t_i, which defines v_i, is executed before event t; and
- any other event that writes to the same variable, i.e. event t_j that defines v_j, for all $1 \leq j \leq l$ and $j \neq i$, should be executed either before t_i or after t.

4.2. CSSA-based SAT Encoding

We construct the quantifier-free first-order logic formula Φ_{CTP}[1] based on the notion of feasible linearizations of CTP (in Section 3.2) and the π-function semantics (in Section 4.1). The construction is straightforward and follows their definitions. The entire formula Φ_{CTP} consists of the following four subformulas:

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

[1] We omit the subscript ρ in CTP_ρ where it is understood from the context.

where Φ_{PO} encodes the program order, Φ_{VD} encodes the variable definitions, Φ_{PI} encodes the π-functions to enforce sequential consistency of shared variable accesses, and Φ_{PRP} encodes the violation of the given property.

To help present the encoding algorithm, we use the following notations:

- **first event t_{first}:** we add a dummy event t_{first} to be the first executed event in the CTP. That is, $\forall t \in CTP$ and $t \neq t_{\mathsf{first}}$, event t must be executed after t_{first};
- **last event t_{last}:** we add a dummy event t_{last} to be the last executed event in the CTP. That is, $\forall t \in CTP$ and $t \neq t_{\mathsf{last}}$, event t must be executed before t_{last};
- **first event t^i_{first} of thread T_i:** for each $i \in Tid$, this is the first event of the thread;
- **last event t^i_{last} of thread T_i:** for each $i \in Tid$, this is the last event of the thread;
- **thread-local 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 event t'. The actual constraint comprising $HB(t, t')$ is described in the next section.

Path Conditions. For each event $t \in CTP$, we define path condition $g(t)$ such that t is executed iff $g(t)$ is true. The path conditions are computed as follows:

1. If $t = t_{\mathsf{first}}$, or $t = t^i_{\mathsf{first}}$ where $i \in Tid$, let $g(t) := \mathsf{true}$.
2. Otherwise, t has a thread-local preceding event t'.
   - if t' has action $(\mathsf{assume}(c), asgn)$, let $g(t) := c \wedge g(t')$;
   - if t' has action $\mathsf{assert}(c)$, let $g(t) := g(t')$.

   Note that an assert event does not contribute to the path condition.

Program Order (Φ_{PO}). Formula Φ_{PO} captures the event order within each thread. *It does not impose any inter-thread constraint.* Let $\Phi_{PO} := \mathsf{true}$ initially. For each event $t \in CTP$,

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. If $t = t_{\mathsf{last}}$, let $\Phi_{PO} := \Phi_{PO} \wedge \bigwedge_{\forall i \in Tid} HB(t^i_{\mathsf{last}}, t_{\mathsf{last}})$;
4. Otherwise, t has a thread-local preceding event t'; let $\Phi_{PO} := \Phi_{PO} \wedge HB(t', t)$.

Variable Definition (Φ_{VD}). Formula Φ_{VD} is the conjunction of all variable definitions. Let $\Phi_{VD} := \mathsf{true}$ initially. For each event $t \in CTP$,

1. If t has action $(\mathsf{assume}(c), asgn)$, for each assignment $v := exp$ in $asgn$, let $\Phi_{VD} := \Phi_{VD} \wedge (v = exp)$;
2. Otherwise, do nothing.

The π-Function (Φ_{PI}). Each π-function defines a new variable v', and Φ_{PI} is a conjunction of all these variable definitions. Let $\Phi_{PI} := \mathsf{true}$ initially. For each $v' \leftarrow \pi(v_1, \ldots, v_l)$ defined in event t, where v' is used, also assume that each v_i, $1 \leq i \leq l$, is defined in event t_i. Let

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

Intuitively, 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, $1 \leq j \leq l$ and $j \neq i$, is either before t_i, or after this use of v_i in t.

Assertion Property (Φ_{PRP}). Let $t \in CTP$ be the event with $\mathsf{assert}(c)$, which specifies the property.

$$\Phi_{PRP} := g(t) \wedge \neg c$$

Intuitively, condition c must hold if t is executed. We negate this requirement to search for a property violation.

Example. Fig. 4 illustrates the CSSA-based encoding of the example in Fig. 3, where subformulas in Φ_{PO} and Φ_{VD}

Path Conditions:

t_0 :
t_1 : $g_1 = \mathsf{true}$
t_2 : $g_2 = g_1 \wedge (\pi^2 > 0)$
t_3 : $g_3 = g_2$
t_4 : $g_4 = g_3$
t_5 : $g_5 = g_4$
t_6 : $g_6 = g_5 \wedge (\pi^4 > 0)$
t_7 : $g_7 = g_6$
t_8 : $g_8 = g_7$

t_9 : $g_9 = \mathsf{true}$
t_{10} : $g_{10} = g_9 \wedge (\pi^6 > 0)$
t_{11} : $g_{11} = g_{10} \wedge (\pi^7 > b_1)$
t_{12} : $g_{12} = g_{11}$
t_{13} : $g_{13} = g_{13}$
t_{14} :

Program Order:

$HB(t_0, t_1)$
$HB(t_1, t_2)$
$HB(t_2, t_3)$
$HB(t_3, t_4)$
$HB(t_4, t_5)$
$HB(t_5, t_6)$
$HB(t_6, t_7)$
$HB(t_7, t_8)$

$HB(t_0, t_9)$
$HB(t_9, t_{10})$
$HB(t_{10}, t_{11})$
$HB(t_{11}, t_{12})$
$HB(t_{12}, t_{13})$
$HB(t_8, t_{14}) \wedge HB(t_{13}, t_{14})$

Variable Definitions:

$x_0 = 0 \wedge y_0 = 0 \wedge l_0 = 1$
$a_1 = \pi^1$
$l_1 = \pi^2 - 1$
$x_1 = 2 + a_1$
$l_2 = \pi^3 + 1$
$y_1 = 1 + a_1$
$l_3 = \pi^4 - 1$
$x_2 = 1 + a_1$
$l_4 = \pi^5 + 1$

$b_1 = 0$
$l_5 = \pi^6 - 1$

$l_6 = \pi^9 + 1$

Fig. 4. The CSSA-based symbolic encoding of the CTP in Fig. 3

are listed. In the figure, t_0, t_{14} are the entry and exit events. Φ_{PRP} (at t_{12}) is defined as $\neg g_{12} \vee (\pi^8 = 1)$. The subformula in Φ_{PI} for $\pi^7 \leftarrow \pi(x_0, x_1, x_2)$ in t_{11} is defined as follows:

$$\begin{array}{lllll} t_{11}: & (\ \pi^7 = x_0 \wedge (\mathsf{true}) & \wedge HB(t_{11}, t_3) & \wedge HB(t_{11}, t_7) \\ & \vee \pi^7 = x_1 \wedge g_3 \wedge HB(t_3, t_{11}) & \wedge \mathsf{true} & \wedge HB(t_{11}, t_7) \\ & \vee \pi^7 = x_2 \wedge g_7 \wedge HB(t_7, t_{11}) & \wedge \mathsf{true} & \wedge \mathsf{true}) \end{array}$$

Some HB-constraints in the above example evaluate to false and true statically – such cases are frequent and therefore we perform simplification to reduce the formula size. Consider t_{10}, for instance, when $(\pi^6 = l_0)$, the constraint $HB(t_{10}, t_4) \wedge HB(t_{10}, t_6) \wedge HB(t_{10}, t_8)$ is implied by the existing constraint $HB(t_{10}, t_2)$, and therefore is omitted. 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.

4.3. Correctness and Complexity

Recall that for two arbitrary events t and t', the constraint $HB(t, t')$ denotes 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 execution time[2]. A satisfiable solution for Φ_{CTP_ρ} therefore induces values of $\mathcal{O}(t)$, i.e., times of all events in the linearization. The constraint $HB(t, t')$ is captured using the less-than operator as follows:

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

We now state the correctness of our encoding.

Theorem 4.1. Formula Φ_{CTP} is satisfiable iff there exists a feasible linearization of the CTP that violates the assertion property.

Proof. The symbolic encoding closely follows the semantics of CTP, the feasible linearizations, and the π-functions. If formula Φ_{CTP} is satisfiable, by following the values of the $\mathcal{O}(t)$ variables in this satisfying assignment, we can construct a total order of the CTP events, which corresponds to a serial execution of these events. Since this serial execution satisfies both the program order and the sequential consistency semantics ($\Phi_{VD} \wedge \Phi_{PI}$) and at same time violates the assertion (Φ_{PRP}), it is a real bug. On the other hand, if formula Φ_{CTP} is unsatisfiable, there cannot be a real bug because, if there were a feasible and erroneous program execution, i.e. a total order of these CTP events, then from this total order we can create a satisfying assignment to all the $\mathcal{O}(t)$ variables – simply assign i to variable $\mathcal{O}(t)$ if event t is the i-th executed event. Furthermore, the read/write concrete values in this real program execution is by

[2] The execution time is an integer value denoting its position in the linearization.

definition sequentially consistent, therefore satisfying ($\Phi_{VD} \wedge \Phi_{PI}$). As a result Φ_{CTP} should be satisfiable, leading to the contradiction. □

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 $trans$. The size of $(\Phi_{PO} \wedge \Phi_{VD} \wedge \Phi_{PI} \wedge \Phi_{PRP})$ in the worst case is $O(n + n + n_\pi \times l_\pi^2 + n_\pi \times l_{trans})$. We also assume that each event in ρ accesses at most one shared variable; this is not a loss of generality because, otherwise, we can always rewrite the composite event into a sequence of simpler events. 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[3]. In contrast, in conventional bounded model checking (BMC) algorithms for verifying concurrent programs, e.g. [WYKG08, KWG09], 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.

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 $\mathcal{O}(t) - \mathcal{O}(t') \leq -1$. It is special in that the integer constant c in the general IDL constraint $(x - y \leq c)$ is always -1 in this case. Deciding this fragment of IDL is easier because consistency can be reduced to cycle detection in a so-called constraint graph, which has a linear complexity, rather than the more expensive negative-cycle detection [WIGG05, WGG06] required by the general IDL constraints.

5. Predicting Other Concurrency Errors

In addition to assertion failures, our algorithm can be used to predict concurrency errors such as atomicity violations and data races. In this case, the overall symbolic encoding remains the same as in Section 4 and the only difference is in subformula Φ_{PRP}.

5.1. Three-Access Atomicity Violations

In this section, we use three-access atomicity violations as an example to illustrate the encoding of Φ_{PRP}. Three-access atomicity violation is a special case of serializability violations, involving a sequence $t_c \ldots t_r \ldots t_{c'}$ 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.

In this case, t_c and $t_{c'}$ are meant to be executed atomically, but due to programming errors, this design intent is not correctly enforced by proper use of synchronizations. As a result, some of the interleaved executions are not serializable. Recent study in [LTQZ06] shows that in practice such atomicity violations account for a very large number of concurrency errors.

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 the adjacent and independent events. Here two events are considered as *independent* iff swapping their execution order always leads to the same program state.

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. We use the set PAV of event triplets to denote all potential three-access atomicity violations in a CTP. Given CTP_ρ and a clearly marked user transaction $trans = t_i \ldots t_j$, where $t_i \ldots t_j$ are events in ρ, the set PAV can be computed by scanning ρ once, and for each remote event $t_r \in CTP_\rho$, finding 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 is deciding whether an event triplet $t_c \ldots t_r \ldots t_{c'}$ can show up in an actual program execution. This is a difficult problem because it essentially is model checking the CTP. However, over-approximate algorithms, such as those based on Lipton's reduction theory [FF04] or [FM09a, FM09b], can be used to weed out event triplets in PAV that are definitely infeasible. For example, the method in [FM09b] reduces the problem of checking the existence of $t_c \ldots t_r \ldots t_{c'}$ to *simultaneous reachability* under nested locking. That is, does there exist

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

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 an analysis based on computing the lock *acquisition histories* [KIG05]. However, this is an over-approximated analysis that ignores both the data and all the synchronizations other than nested locks. In practice, programs with only nested locking can enforce mutual exclusion, but cannot coordinate thread interactions because nested locks cannot simulate powerful primitives such as semaphores. In a more recent paper [KW10], this analysis is made more precise by using a Universal Causality Graph (UCG) which also considers synchronizations other than the nested locks.

Encoding Atomicity Violations. Although the analysis such as [FM09b, KW10] can be used to quickly weed out the apparently bogus violations in PAV, many of the remaining ones are still bogus. Whether an event triplet in PAV exists in a feasible linearization of the CTP can be decided symbolically. Given a set PAV of potential violations, we build formula Φ_{PRP} as follows: For each $\langle t_c, t_r, t_{c'}\rangle \in PAV$, where t_c and t_r are data dependent and t_r and $t_{c'}$ are data dependent, let

$$\Phi_{PRP} := g(t_c) \wedge g(t_r) \wedge g(t_{c'}) \wedge HB(t_c, t_r) \wedge HB(t_r, t_{c'})$$

Recall that for two events t and t', the constraint $HB(t, t')$ denote that t must be executed before t'. As in Section 4, the entire formula Φ_{CTP} consists of the following four subformulas:

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

5.2. Data Races and Other Violations

We can use Φ_{PRP} to encode data races, as well as the existence of other generic programming errors, as a set of happens-before constraints. Although in principle data races can be modeled using embedded assertions, it is often more convenient to express them directly using happen-before constraints. For example, if we want to check data race between two events t_i and t_j, and assuming that t_i' (or t_j') is the thread-local preceding event of t_i (or t_j), the property formula is defined as follows,

$$\Phi_{PRP} := g(t_i) \wedge g(t_j) \wedge HB(t_i', t_j) \wedge HB(t_j', t_i)$$

That is, t_i and t_j are simultaneously enabled.

The capability of checking embedded assertions can also be leveraged to encode other programming errors. For example, if we are interested in checking double locking/unlocking errors of a non-recursive mutex lock, i.e. the lock is acquired multiple times without being released in between, or is released multiple times without being acquired in between. For checking the double locking error, we first insert an assertion $assert(A \neq i)$ for each $lock(A)$ executed by Thread T_i. Here the value of A indicates the lock owner's thread index (recall that 0 means the lock is free). Then we transform the lock event into a guarded assignment $\langle \mathsf{assume}(A = 0)\{A := i\}\rangle$. For checking the double unlocking error, we first insert an assertion $assert(A = i)$ for each $unlock(A)$ executed by Thread T_i, and then transform the unlock event into a regular assignment $A := 0$.

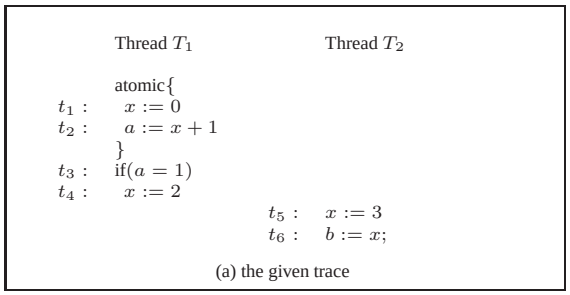
Similarly, for recursive locks, one can use embedded assertions to check locking errors caused by a mismatched number of lock/unlock events. In practice, embedded assertions can be inserted into the CTPs automatically to check such generic programming errors.

When there are multiple potential errors, they can be either combined together to form a single property formula, or separated into independent subproblems. Since these subproblems are independent from each other (embarrassingly parallel), it is often advantageous to separate them and solve them individually using parallel machines.

5.3. Detecting Erroneous Prefixes

The algorithm presented so far aims at detecting bugs in all feasible linearizations of a CTP. Therefore, a bug is reported iff (1) a three-access atomicity violation occurs in an interleaving, and (2) the interleaving is a feasible linearization of CTP_ρ. Sometimes this may become too restrictive, since the existence of an atomicity violation, for example, can lead 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. 5. 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



(a) the given trace

t_1 : $\langle 1, (\mathsf{assume}(\mathsf{true}\ ), \{x_1 := 0\ \})\rangle$
t_2 : $\langle 1, (\mathsf{assume}(\mathsf{true}\ ), \{a_1 := \pi^1 + 1\ \})\rangle$
t_3 : $\langle 1, (\mathsf{assume}(a_1 = 1), \{\ \})\rangle$
t_4 : $\langle 1, (\mathsf{assume}(\mathsf{true}\ ), \{x_2 := 2\ \})\rangle$
t_5 : $\langle 2, (\mathsf{assume}(\mathsf{true}\ ), \{x_3 := 3\ \})\rangle$
t_6 : $\langle 2, (\mathsf{assume}(\mathsf{true}\ ), \{b_1 := \pi^2\ \})\rangle$

(b) erroneous prefix

Fig. 5. The atomicity violation leads to a previously untaken branch.

Path Conditions:

t_1 : $g_1 = \mathsf{true}$
t_2 : $g_2 = g_1$
t_3 : $g_3 = g_2$
t_4 : $g_4 = g_3 \wedge (a_1 = 1)$
t_5 : $g_5 = \mathsf{true}$
t_6 : $g_6 = \mathsf{true}$

Program Order:

$HB(t_1, t_2)$
$HB(t_2, t_3)$
$HB(t_3, t_4)$
$HB(t_5, t_6)$

Variable Definitions:

$x_1 = 0$
$a_1 = \pi^1 + 1$
$x_2 = 2$
$x_3 = 3$
$b_1 = \pi^2$

The π-Functions:

$$
\begin{array}{lll}
t_2: & (\pi^1 = x_1) \wedge g_1 \wedge HB(t_1,t_2) & \wedge(HB(t_5,t_1) \vee HB(t_2,t_5)) \\
\vee & (\pi^1 = x_3) \wedge g_5 \wedge HB(t_5,t_2) & \wedge(HB(t_1,t_5) \vee HB(t_2,t_1)) \\
t_6: & (\pi^2 = x_1) \wedge g_1 \wedge HB(t_1,t_6) & \wedge(HB(t_4,t_1) \vee HB(t_6,t_4)) \wedge (HB(t_5,t_1) \vee HB(t_6,t_5)) \\
\vee & (\pi^2 = x_2) \wedge g_4 \wedge HB(t_4,t_6) & \wedge(HB(t_1,t_4) \vee HB(t_6,t_1)) \wedge (HB(t_5,t_4) \vee HB(t_6,t_5)) \\
\vee & (\pi^2 = x_3) \wedge g_5 \wedge HB(t_5,t_6) & \wedge(HB(t_1,t_5) \vee HB(t_6,t_1)) \wedge (HB(t_4,t_5) \vee HB(t_6,t_4))
\end{array}
$$

Fig. 6. The CSSA-based encoding of CTP_ρ in Fig. 5

encoding, the π-constraint in t_6 will become invalid.

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

Note that in the interleaving $t_1 t_5 t_2 \ldots$, 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.

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 encoding algorithm in Section 4 as follows. Let event triplet $\langle t_c, t_r, t_{c'}\rangle \in PAV$ be a potential violation. We modify the construction of Φ_{PI} (for the π-function in event t) as follows:

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

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.

6. Symbolic Context Bounding

In this section, we present a symbolic encoding that effectively bounds the number of context switches allowed by an interleaving. Traditionally, a *context switch* is defined as the computing process of storing and restoring the CPU state (context) when executing a concurrent program, such that multiple processes or threads can share a single CPU resource. The idea of using context bounding to reduce complexity in verifying concurrent programs was introduced by Qadeer and Rehof [QR05]. Several subsequent studies [MQ06, LR08] have confirmed that concurrency bugs in practice can often be exposed in interleavings with a surprisingly small number of context switches. Therefore, during verification, imposing a small context bound to the admitted interleavings is a good strategy to control the computational complexity; in practice this is often enough to uncover most of the subtle concurrency bugs.

Example. Consider the running example in Fig. 1. If we restrict the number of context switches of an interleaving to 1, there are only two possibilities:

$$\rho' = (t_1 t_2 \dots t_8)(t_9 t_{10} \dots t_{13})$$
$$\rho'' = (t_9 t_{10} \dots t_{13})(t_1 t_2 \dots t_8)$$

In both cases the context switch happens when one thread completes its execution. However, none of the two traces is erroneous; and ρ'' is not even feasible. When we increase the context bound to 2, the number of admitted interleavings remains small but now the following trace is included:

$$\rho''' = (t_1 t_2 t_3)(t_9 t_{10} t_{11} t_{12})(t_4 \dots t_8)$$

The trace has two context switches and exposes the error in t_{12} (where $y = 0$).

6.1. Revisiting the HB-Constraints

We defined $HB(t, t')$ as $\mathcal{O}(t) < \mathcal{O}(t')$ earlier. However, the *strictly-less-than* constraint is sufficient, but not necessary, to ensure the correctness of our encoding. To facilitate context bounding, we modify the definition of $HB(t, t')$ as follows:

1. $HB(t, t') := \mathcal{O}(t) \leq \mathcal{O}(t')$ if one of the following conditions hold: $tid(t) = tid(t')$, or $t = t_{\mathsf{first}}$, or $t' = t_{\mathsf{last}}$.
2. $HB(t, t') := \mathcal{O}(t) < \mathcal{O}(t')$ otherwise.

Note first that, if two events t, t' are from the same thread, the execution time $\mathcal{O}(t)$ need not be strictly less than $\mathcal{O}(t')$ to enforce $HB(t, t')$. This is because the CSSA form, through the renaming of definitions and uses of thread-local variables, already guarantees the *flow-sensitivity* within each thread; that is, implicitly, a definition always happens before the subsequent uses. Therefore, when $tid(t) = tid(t')$, we relax the definition of $HB(t, t')$ by using *less than or equal to*[4].

Second, if events t, t' are from two different threads (and $t \neq t_{\mathsf{first}}$, $t \neq t_{\mathsf{last}}$), according to our encoding rules, the constraint $HB(t, t')$ must have been introduced by the subformula Φ_{PI} encoding π-functions. In such case, $HB(t, t')$ means that there is *at least one context switch* between the execution of t and t'. Therefore, when $tid(t) \neq tid(t')$, we force event t to happen *strictly before* event t' in time.

6.2. Adding the Context Bound

Let b be the maximal number of context switches allowed in an interleaving. Given the formula Φ_{CTP_ρ} as defined in the previous section, we construct the context-bounded formula $\Phi_{CTP_\rho}(b)$ as follows:

$$\Phi_{CTP_\rho}(b) := \Phi_{CTP_\rho} \wedge (\mathcal{O}(t_{\mathsf{last}}) - \mathcal{O}(t_{\mathsf{first}}) \leq b)$$

The additional constraint states that t_{last}, the unique exit event, must be executed no more than b steps later than t_{first}, the unique entry event.

The execution times of the events in a trace always form a non-decreasing sequence. Furthermore, the execution time is forced to increase whenever a context switch happens, i.e., as a result of $HB(t, t')$ when $tid(t) \neq tid(t')$. In the above constraint, such increases of execution time is limited to less than or equal to b[5].

[4] When $HB(t, t')$ is a constant, we replace it with true or false.
[5] In CHESS [MQ06], which uses an explicit rather than symbolic search, a variant is used to count only the *preemptive* context switches.

Theorem 6.1. Let ρ' be a feasible linearization of CTP_ρ. Let $CB(\rho')$ be the number of context switches in ρ'. If $CB(\rho') \leq b$ and ρ' violates the correctness property, then $\Phi_{CTP_\rho}(b)$ is satisfiable.

Proof. Let $m = CB(\rho')$. We partition ρ' into $m+1$ segments $seg_0\ seg_1 \ldots seg_m$ such that each segment is a subsequence of events without context switch. Now we assign an execution time (integer) for all $t \in \rho'$ as follows: $\mathcal{O}(t) = i$ iff $t \in seg_i$, where $0 \leq i \leq m$. In our encoding, only the HB-constraints in Φ_{PO} and Φ_{PI} and the context-bound constraint refer to the $\mathcal{O}(t)$ variables. The above variable assignment is guaranteed to satisfy these constraints. Therefore, if ρ' violates the correctness property, then $\Phi_{CTP_\rho}(b)$ is satisfiable. □

By the same reasoning, if $CB(\rho') > b$, trace ρ' is excluded by formula $\Phi_{CTP_\rho}(b)$.

6.3. Lifting the CB Constraint

In this context bounded analysis, one can empirically choose a bound b_{max} and check the satisfiability of formula $\Phi_{CTP_\rho}(b_{max})$. Alternatively, one can iteratively set $b = 1, 2, \ldots, b_{max}$; and for each b, check the satisfiability of the formula

$$\Phi_{CTP_\rho} \wedge (\mathcal{O}(t_{\mathsf{last}}) - \mathcal{O}(t_{\mathsf{first}}) = b)$$

In both cases, if the formula is satisfiable, an error has been found. Otherwise, the SMT solver used to decide the formula can return a subset of the given formula as a *proof of unsatisfiability*. More formally, the proof of unsatisfiability of a formula f, which is unsatisfiable, is a subformula f_{unsat} of f such that f_{unsat} itself is also unsatisfiable.

The proof of unsatisfiability f_{unsat} can be viewed as a generalization of the given formula f; it is more general because some of the constraints of f may not be needed to prove unsatisfiability. In our method, we can check whether the context-bound constraint appears in f_{unsat}. If the context-bound constraint does not appear in f_{unsat}, it means that, even without context bounding, the formula Φ_{CTP_ρ} itself is unsatisfiable. In other words, we have generalized the context-bounded proof into a proof of the general case – that the property holds in all the feasible interleavings.

7. Relating to Other Causal Models

In this section, we show that our symbolic algorithm can be further constrained to match many causal models in the literature. Recall that these models fall into two categories: under-approximated models and over-approximated models. The under-approximated models (e.g. [SRA05, CR07, SCR08, SFF09]) often use more stringent causality constraints to ensure that they do not admit any bogus interleaving. Through the exercise of constraining CTP to match these models, we can demonstrate that our model covers more interleavings. The over-approximated methods (e.g. [FF04, FM09b, FM09a]) typically focus on only the control paths while ignoring the data; therefore many of the reported violations cannot appear in the actual program execution. We demonstrate that our algorithm can mimic these methods by relaxing CTP to make it less precise (but potentially more scalable).

7.1. Under-Approximated Models

We use the maximal causal model [SCR08] as an example, because it subsumes most of the other under-approximated models. In the maximal causal model, only events involving shared objects are recorded. Events involving local objects are ignored. Furthermore, for each shared variable read and write, only the concrete value of the variable is recorded. There is no knowledge as to what kind of source code statement produces a certain event. Therefore, an event has one of the following forms:

- A concurrency synchronization primitive;
- Reading value val from a shared variable $v \in SV$;
- Writing value val to a shared variable $v \in SV$.
- An assertion event (the property);

Constraining CTP. Let $\rho = t_1 \ldots t_n$ be the given execution trace and $\mathsf{pfx} = t_1 \ldots t_i$, where $1 \leq i < n$, be a prefix of ρ. Let s be the program state after executing pfx from s_0. Let $s[v]$ be the concrete value of variable $v \in V$ in state

	Thread T_1	Thread T_2
t_1 :	reading 0 from x	
t_2 :	$acq(l)$	
t'_3 :	writing 1 to x	
t_4 :	$rel(l)$	
t_5 :	writing 1 to y	
t_6 :	$acq(l)$	
t_7 :	writing 1 to x	
t_8 :	$rel(l)$	
t_9 :		nop
t_{10} :		$acq(l)$
t_{11} :		reading 1 from x
t_{12} :		$\mathsf{assert}(y == 1)$
t_{13} :		$rel(l)$

Fig. 7. The concrete event sequence

$$
\begin{array}{rlllll}
\star\star t_1: & \langle & 1, (\mathsf{assume}(x=0), & \{ & \}) & \rangle \\
t_2: & \langle & 1, (\mathsf{assume}(l>0), & \{l := l-1 & \}) & \rangle \\
\star\star t'_3: & \langle & 1, (\mathsf{assume}(\mathsf{true}), & \{x := 1 & \}) & \rangle \\
t_4: & \langle & 1, (\mathsf{assume}(\mathsf{true}), & \{l := l+1 & \}) & \rangle \\
\star\star t_5: & \langle & 1, (\mathsf{assume}(\mathsf{true}), & \{y := 1 & \}) & \rangle \\
t_6: & \langle & 1, (\mathsf{assume}(l>0), & \{l := l-1 & \}) & \rangle \\
\star\star t_7: & \langle & 1, (\mathsf{assume}(\mathsf{true}), & \{x := 1 & \}) & \rangle \\
t_8: & \langle & 1, (\mathsf{assume}(\mathsf{true}), & \{l := l+1 & \}) & \rangle \\
\star\star t_9: & \langle & 2, (\mathsf{assume}(\mathsf{true}), & \{ & \}) & \rangle \\
t_{10}: & \langle & 2, (\mathsf{assume}(l>0), & \{l := l-1 & \}) & \rangle \\
\star\star t_{11}: & \langle & 2, (\mathsf{assume}(x=1), & \{ & \}) & \rangle \\
t_{12}: & \langle & 2, (\mathsf{assert}(y=1) & &) & \rangle \\
t_{13}: & \langle & 2, (\mathsf{assume}(\mathsf{true}), & \{l := l+1 & \}) & \rangle
\end{array}
$$

Fig. 8. The constrained CTP

s. For each event t in ρ, let $RDsv(t)$ be the set of shared variables used in t – appearing either in a condition or in the right-hand side of an assignment.

Let $CTP_\rho = (T, \sqsubseteq)$ as in Definition 3.1, we compute the constrained model $CM_\rho = (T', \sqsubseteq')$ as follows:

1. **Deriving T':** For each event $t \in \rho$,
   - if t is a synchronization primitive or $t = (tid, \mathsf{assert}(c))$, add t to T';
   - otherwise, $t = (tid, (\mathsf{assume}(c), asgn))$. Add event $t' = (tid, (\mathsf{assume}(c'), asgn'))$ to T', where
     - $c' = \bigwedge_{\forall v \in RDsv(t)} (v = s_i[v])$,
     - $asgn' = \{v := s_i[exp] \mid v \in RDsv(t) \text{ and } v := exp \in asgn\}$.
2. **Deriving $\sqsubseteq'$:** For two arbitrary events $t'_i, t'_j \in T'$, we have $t'_i \sqsubseteq t'_j$ iff there exist $t_i \sqsubseteq t_j \in T$ and events t'_i, t'_j are derived from t_i, t_j respectively.

The condition c' in Rule 1 needs further explanation. Here $v = s_i[v]$ means that in the constrained causal model, we only admit those trace permutations where, when the execution reaches event t, the shared variables in $RDsv(t)$ will have exactly the same set of values as they are in the given trace ρ. Otherwise, the trace permutation will not be admitted by CM_ρ. Clearly, this is an under-approximation of CTP_ρ.

As an example, we slightly modify the program in Fig. 1 as follows: *we replace* $t_3 : x := 2+a$ *with* $t'_3 : x := 1+a$. The sequence of concrete events in ρ is shown in Fig. 7. There still exists an erroneous trace that violates the assertion in t_{12}. The difference between the two examples is subtle: in the original example, the erroneous trace ρ' in Section 2 cannot be predicted by the maximal causal model; whereas in the modified example, the erroneous trace can be predicted by the maximal causal model. The reason is that in the modified example, the program code in t'_3 and t_7 produce identical events in ρ: *Writing value 1 to the shared variable* x. Therefore, t_{11} can be moved ahead of t_5 but after t_4 (the permutation satisfies the sequential consistency axioms used in the maximal causal model).

We derive the constrained model CM_ρ as shown in Fig. 8. Whenever an event has a different form in CM_ρ from the one in CTP_ρ (Fig. 2), we mark it with the symbol $\star\star$. Note that all the semaphore events remain symbolic, whereas the rest are under-approximated into concrete values. For instance, event t_1 is reduced from $\langle 1, (\mathsf{assume}(\mathsf{true}), \{a := x\})\rangle$ to $\langle 1, (\mathsf{assume}(x = 0), \{\ \})\rangle$, because value 0 is being read from the shared variable x in the given trace ρ. Similarly, event t'_3 is reduced from $\langle 1, (\mathsf{assume}(\mathsf{true}), \{x := 1 + a\})\rangle$ to $\langle 1, (\mathsf{assume}(\mathsf{true}), \{x := 1\})\rangle$, because the right-hand-side expression evaluates to 1 in ρ. These events are no longer symbolic. Note that concrete events correspond to constant values, which can be propagated to further simplify the constraints in our encoding. However, these also result in less coverage in CM_ρ than CTP_ρ.

Since CM_ρ shares the same symbolic representation as CTP_ρ, the notion of *feasible linearizations* of a CTP, defined in Section 3.2, and the symbolic algorithm in Section 4 remain applicable. In the running example, the erroneous trace $\rho' = (t_1 t_2 t'_3 t_4) t_9 t_{10} t_{11} t_{12} t_{13} (t_5 – t_8)$ is admitted by CM_ρ. Since CM_ρ admits less interleavings than CTP_ρ, there can be significantly more opportunities to statically simplify subformulas in Φ_{CM_ρ} before submitting it to a SMT solver. Consider the example in Fig. 8. The new formula Φ_{CM_ρ} is similar to the one in Fig. 4. However, the following

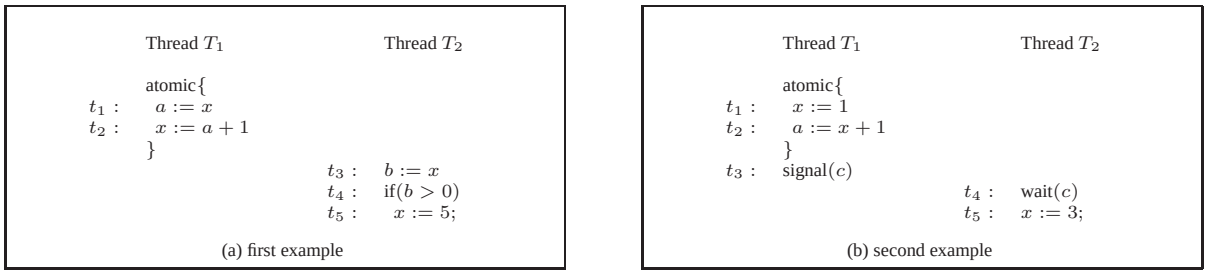


Fig. 9. Ignoring data/synchronizations leads to bogus errors. All variables are initialized to 0.

variables become constants.

$t_3 : \quad x_1 = 1$
$t_6 : \quad y_1 = 1$
$t_7 : \quad x_2 = 1$

These constant values can be propagated to drastically simplify the constraints of path conditions and π-functions, potentially resulting in improved performance.

7.2. Over-Approximated Models

We use the predictive method in [FM09b, FM09a] as an example. It focuses on modeling the control paths under nested locking, while ignoring the data and synchronizations other than nested locks. Therefore, an $if(c)...else$ statement in the program can be viewed as being abstracted into $if(*)...else$ to allow the execution of both branches; an assignment $x := expr$, where $x \in SV$ is shared, can be viewed as being abstracted into WR(x); and an assignment a := x, where a is thread-local and $x \in SV$ is shared, can be viewed as being abstracted into RD(x). The actual values read/written are not modeled.

Relaxing CTP. We relax CTP_ρ, by ignoring the data flow, to mimic this over-approximated model, denoted CAM_ρ, .

- For events in CTP_ρ involving no synchronization primitives,
  1. assume(c) becomes assume$(*)$, meaning that the condition always holds;
  2. $sv := expr$ becomes $sv := *$, meaning that the actual value written to shared variable $sv \in SV$ is ignored;
  3. $lv := sv$ becomes $* := sv$, meaning that the actual value read from shared variable $sv \in SV$ (and then written to local variable $lv \in LV$) is ignored.
- For events in CTP_ρ involving synchronization primitives, such as locks, they remain the same in CAM_ρ.

Since CAM_ρ shares the same form of symbolic representation as CTP_ρ, the notion of *feasible linearizations* and the SMT-based symbolic algorithm remain applicable. In this case, due to the over-approximations in CAM_ρ, our algorithm may report bogus violations same as in the method of [FM09b, FM09a].

Fig. 9 provides two examples in which the transactions, marked by keyword *atomic*, are indeed serializable, but over-approximated methods [FF04, FM09b, FM09a] would report 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.

Fig. 10 shows the relaxed models for CTPs in Fig. 9. Note that both assume$(*)$ and $x := *$ would lead to SMT constraints that are constant true, since $*$ means arbitrary value. They do not add cost to the SMT encoding. Our symbolic analysis would report a bogus violation in the first example (data is ignored), but would not report any violation in the second example (signal/wait are modeled precisely). In this sense, CAM_ρ remains more accurate than the model in [FM09b, FM09a]. Their model would correspond to a further over-approximation of CAM_ρ, where all synchronizations other than nested mutex locks are ignored, leading the report of bogus errors in both examples in Fig. 9.

t_1 : $\langle 1, (\mathsf{assume}(\mathsf{true}\ ), \quad \{* := x\}) \quad \rangle$
t_2 : $\langle 1, (\mathsf{assume}(\mathsf{true}\ ), \quad \{x := *\}) \quad \rangle$

t_3 : $\langle 2, (\mathsf{assume}(\mathsf{true}\ ), \quad \{* := x\}) \quad \rangle$
t_4 : $\langle 2, (\mathsf{assume}(*), \quad \{\ \}) \quad \rangle$
t_5 : $\langle 2, (\mathsf{assume}(\mathsf{true}\ ), \quad \{x := *\}) \quad \rangle$

(a) first example

t_1 : $\langle 1, (\mathsf{assume}(\mathsf{true}\ ), \quad \{x := *\}) \quad \rangle$
t_2 : $\langle 1, (\mathsf{assume}(\mathsf{true}\ ), \quad \{* := x + 1\}) \quad \rangle$
t_3 : $\langle 1, (\mathsf{assume}(\mathsf{true}\ ), \quad \{c := 1\}) \quad \rangle$

t_4 : $\langle 2, (\mathsf{assume}(\mathsf{true}\ ), \quad \{c := 0\}) \quad \rangle$
t_4 : $\langle 2, (\mathsf{assume}(c > 0), \quad \{c := 0\}) \quad \rangle$
t_5 : $\langle 2, (\mathsf{assume}(\mathsf{true}\ ), \quad \{x := *\}) \quad \rangle$

(b) second example

Fig. 10. The relaxed CTP after ignoring data flow

Thread T_0

```
    int x = 0;
    int y = 0;
    pthread_t t1, t2;
    main() {
t1      pthread_create(&t1,0,foo,0);
t2      pthread_create(&t2,0,bar,0);
t3      pthread_join(t2,0);
t4      pthread_join(t1,0);
t5      assert( x != y);
    }
```

Thread T_1

```
        foo() {
            int a;
t11         a=y;
t12         if (a==0) {
t13             x=1;
t14             a=x+1;
t15             x=a;
t16         }else
t17             x=0;
t18     }
```

Thread T_2

```
        bar() {
            int b;
t21         b=x;
t22         if (b==0) {
t23             y=1;
t24             b=y+1;
t25             y=b;
t26         }else
t27             y=0;
t28     }
```

Fig. 11. A multithreaded C program using POSIX threads.

We note that the purpose of this exercise is to highlight the differences between CTP_ρ and the existing predictive models, rather than posing them as competing methods. In practice, we suggest the use of both collaboratively: one can use over-approximate models to quickly weed out spurious violations (see Section 5.1), and use the more precise CTP_ρ to check the remaining ones.

8. Encoding Threads with Branches

The symbolic algorithm presented up to this point is tuned for concurrent trace programs derived from a single execution trace ρ. An important property of CTPs derived from concrete execution traces is that each thread has a single control path since all branching decisions have already been made at runtime. Therefore, each thread is in fact a bounded straight-line sequential program. Although our CSSA based encoding has been designed with CTP in mind, it can easily be extended to the more general application settings.

In this section, we show how the encoding in Section 4 can be extended to handle concurrent threads with branches. In this case, we assume that each thread is a loop-free, recursion-free, bounded sequential program, but with possibly many control paths. Such models are useful in various program analysis and verification settings. For example, in bounded model checking [BCCZ99], one can construct such model by unrolling the loops/recursions of the threads for a fixed number of times. In the context of predictive analysis, one can also construct such model by merging multiple CTPs together and later checking them together.

Example. Figure 11 shows an example of a multithreaded C program with two shared variables x and y. The main thread T_0 creates T_1 and T_2, which in turn start running `foo` and `bar`, respectively. Thread T_0 waits for T_1, T_2 to terminate and join back, before checking the assertion condition $(x \neq y)$. Here `pthread_create` and `pthread_join` are routines in *PThreads* library, directly corresponding to *fork/join* in our model. Figure 12 shows the bounded execution model of the program. (Since this particular example does not have loops and recursions, its bounded execution model happens to be the same as the original program.)

In this graph representation, nodes denote control locations and edges denote events. We use $\triangle$ to indicate the start of $fork$ and ∇ to indicate the end of $join$. Note that $fork$ results in simultaneously executing all the outgoing edges, while $join$ results in simultaneously executing all the incoming edges. The assertion at t_5 defines the correctness property, which holds in some, but not in all, execution traces of the program. In particular, the execution trace $\rho = t_1 t_2 \{t_{11}\text{–}t_{15}\} t_{18} t_{21} t_{26} t_{27} t_{28} \{t_3\text{–}t_5\}$ does not violate the assertion, whereas the execution trace $\rho' = t_1 t_2 \{t_{11}\text{-}t_{14}\} t_{21} t_{26} t_{27} t_{28} t_{15} t_{18} \{t_3\text{-}t_5\}$ violates the assertion.

The key to modeling branches within a thread is precisely correlating multiple variable definitions (in different

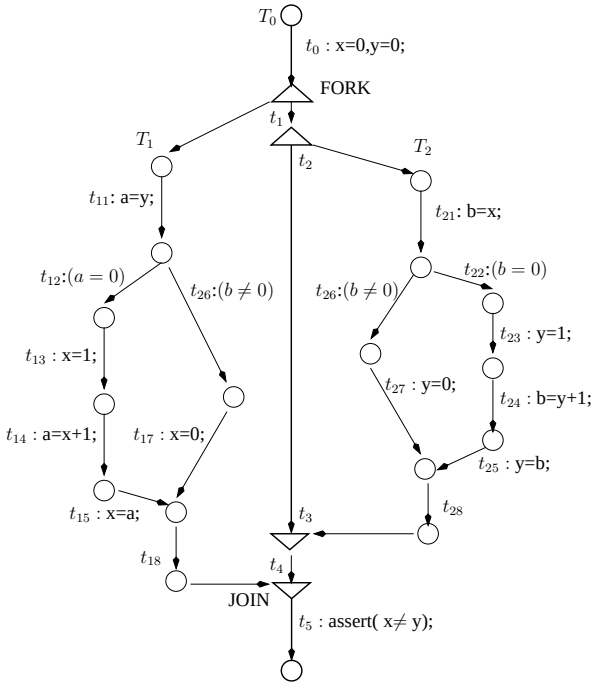


Fig. 12. The control flow graph of the program in Figure 11.

branches) to the subsequent uses of that variable. Unlike in the CTP case, here we need the ϕ-functions as in the classic SSA form to model the confluence of multiple if-else branches.

Definition 8.1. A ϕ-function, for a local variable v at the merging node of multiple branches of the same thread, has the form $w \leftarrow \phi(w_1, \ldots, w_l)$, where each w_j ($1 \leq j \leq l$) is the value written to v along the j-th incoming branch.

Intuitively, the value of a variable at the merging node, denoted by w, is mapped to a preceding write w_j iff the control path p is executed and w_j is defined along path p.

During the construction of the CSSA form, we add the following step for threads with multiple control paths. For each local variable $v \in LV_i$, at each merging point of if-else branches, we add a ϕ-function as defined above to represent the possible definitions. During the symbolic encoding, if a local variable has a ϕ-function $w \leftarrow \phi(w_1, \ldots, w_l)$, where w_j is last defined along the path ended with edge t_j, we add the following constraint

- $\Phi_{VD} := \Phi_{VD} \wedge \bigvee_{j=1}^{l}(w = w_j) \wedge g(t_j)$.

Here $g(t_j)$ is the guard of the incoming edge t_j (incoming to the merging node).

Figure 13 shows the symbolic encoding of individual threads for the running example in Figure 12. In this example, fresh variables $x_1, x_2, x_3, y_1, y_2, y_3$ are added to denote the values of the six global writes, while $r_x^1, r_x^2, r_x^3, r_y^1, r_y^2, r_y^3$ are added to denote the values of the six global reads. The property subformula is defined as

$$\Phi_{PRP} := \ g(t_5) \wedge \neg(r_x^3 \neq r_y^3)$$

The ϕ-functions for variables a_3 and b_3 at merging points t_{18} and t_{28} are interpreted as follows,

$$(a_3 = a_1) \wedge g(t_{17}) \vee (a_3 = a_2) \wedge g(t_{15})$$
$$(b_3 = b_1) \wedge g(t_{27}) \vee (b_3 = b_2) \wedge g(t_{25})$$

That is, when the control path along event t_{17} is taken, a_3 should be the same as a_1; otherwise, a_3 should be the same as a_2 which is defined on the path of event t_{15}.

The π-functions for modeling shared variable reads stay the same as in Section 4. The semantics of π-functions, as defined in Section 4.1, is already general enough to handle the impact of branches over shared variable accesses. In particular, in the symbolic encoding of $v' \leftarrow \pi(v_1, \ldots, v_l)$ where the use (read) of the variable (v') is in event t and each definition (v_i) is written in event t_i, we say $(v' = v_i)$ only if $g(t_i)$ is true. Here the path condition $g(t_i)$ being

Variable Definitions (Φ_{VD}):

$t_0 : x_0 = 0 \wedge y_0 = 0$	
$t_1 :$	
$t_2 :$	
$t_3 :$	
$t_4 :$	
$t_5 :$	
$t_{11} : a_1 = r_y^1$	$t_{21} : b_1 = r_x^2$
$t_{12} :$	$t_{22} :$
$t_{13} : x_1 = 1$	$t_{23} : y_1 = 1$
$t_{14} : a_2 = r_x^1 + 1$	$t_{24} : b_2 = r_y^2 - 1$
$t_{15} : x_2 = a_1$	$t_{25} : y_2 = b_2$
$t_{16} :$	$t_{26} :$
$t_{17} : x_3 = 0$	$t_{27} : y_3 = 0$
$t_{18} : a_3 = \phi(a_1, a_2)$	$t_{28} : b_3 = \phi(b_1, b_2)$

Program Order (Φ_{PO}):

$HB(t_0, t_1)$	
$HB(t_1, t_2)$	$HB(t_1, t_{11})$
$HB(t_2, t_3)$	$HB(t_2, t_{21})$
$HB(t_3, t_4)$	
$HB(t_4, t_5)$	
$HB(t_{11}, t_{12})$	$HB(t_{21}, t_{22})$
$HB(t_{11}, t_{16})$	$HB(t_{21}, t_{26})$
$HB(t_{12}, t_{13})$	$HB(t_{22}, t_{23})$
$HB(t_{13}, t_{14})$	$HB(t_{23}, t_{24})$
$HB(t_{14}, t_{15})$	$HB(t_{24}, t_{25})$
$HB(t_{16}, t_{17})$	$HB(t_{26}, t_{27})$
$HB(t_{15}, t_{18})$	$HB(t_{25}, t_{28})$
$HB(t_{17}, t_{18})$	$HB(t_{27}, t_{28})$
$HB(t_{18}, t_4)$	$HB(t_{28}, t_3)$

Path Conditions of All Events:

$g(t_1) = \mathsf{true}$	
$g(t_2) = g(t_1)$	
$g(t_3) = g(t_2)$	
$g(t_4) = g(t_3)$	
$g(t_5) = g(t_4)$	
$g(t_{11}) = g(t_1)$	$g(t_{21}) = g(t_2)$
$g(t_{12}) = g(t_{11}) \wedge (a_1 = 0)$	$g(t_{22}) = g(t_{21} \wedge (b_1 = 0)$
$g(t_{13}) = g(t_{12})$	$g(t_{23}) = g(t_{22})$
$g(t_{14}) = g(t_{13})$	$g(t_{24}) = g(t_{23})$
$g(t_{15}) = g(t_{14})$	$g(t_{25}) = g(t_{24})$
$g(t_{16}) = g(t_{11}) \wedge (a_1 \neq 0)$	$g(t_{26}) = g(t_{21}) \wedge (b_1 \neq 0)$
$g(t_{17}) = g(t_{16})$	$g(t_{27}) = g(t_{26})$
$g(t_{18}) = g(t_{15}) \vee g(t_{17})$	$g(t_{28}) = g(t_{25}) \vee g(t_{27})$

Fig. 13. The symbolic encoding of individual threads of the bounded execution model in Figure 12.

true means that the thread-local control path along event t_i is executed. Therefore, if there are multiple control paths in a thread, they will be distinguished by their distinct path conditions. In the example of Figure 13, we have omitted the detailed encoding of the following π-functions: $r_y^1 \leftarrow \pi(y_0, y_1, y_2, y_3)$, $r_x^1 \leftarrow \pi(x_1)$, $r_x^2 \leftarrow \pi(x_0, x_1, x_2, x_3)$, and $r_y^2 \leftarrow \pi(y_1)$, since the encoding follows the same algorithm as in Section 4 and the example of Figure 6.

9. Experiments

We have implemented the proposed symbolic predictive analysis in a tool called *Fusion*. Our tool is capable of handling symbolic execution traces generated by Java programs and multi-threaded C programs using the Linux *PThreads* library. For C programs, we use *CIL* [NMRW02] to instrument the program source code, so that at runtime, the program itself can log the executed statements as a sequence of symbolic events. For Java programs, our tool can parse event traces logged at runtime by a modified Java virtual machine, kindly provided by Mahmoud Said and Zijiang Yang [SWYS11]. For symbolic analysis, we use the *Yices* SMT solver [DdM06] to solve the quantifier-free first-order logic formulas. Our experiments were conducted on a PC with 1.6 GHz Intel processor and 2GB memory running Linux.

9.1. Predicting Assertion Failures

We have conducted experiments using the following benchmarks: The first set consists of C variants of the *banking* example [FNU03] with known bugs due to atomicity violations. Unlike previous work [WS06, CR07, SCR08], we directly check the functional correctness property, stating the consistency of all bank accounts at the end of the execution; this is a significantly harder problem than detecting data races [CR07, SCR08] or atomicity violations [WS06] (which may not cause a violation of the functional property). The second set of benchmarks are the *indexer* examples from [FG05], which we implemented using C and the Linux *PThreads* library. In these examples, multiple threads share a hash table with 128 entries. With less than 12 threads, there is no hash table collision among different threads – although this fact cannot be easily inferred by purely static analysis. With more than 12 threads, the number of irredundant interleavings (after partial order reduction) quickly explodes. In our experiments, we set the number of threads to 15, 20, and 25, respectively. Our properties are assertions stating that no collision has happened on a particular hash table entry.

Table 1 shows the results. The first three columns show the statistics of the test cases, including the name, the number of threads, and the number of shared and total variables (that are accessed in the trace). The next two columns show whether the given (non-erroneous) trace has an erroneous permutation, and the trace length after slicing. The next three columns show the run times of trace capturing and slicing, our symbolic analysis, and our context-bounded symbolic analysis (with bound 2). The runtime is the total time spent in deciding all the property: either the time to find an error or the time to prove that there is no error. The final two columns show the run times of a BMC algorithm [WYKG08] applied to the same CTPs with the BMC unrolling depth set to the trace length, and an explicit search algorithm enhanced by DPOR [FG05] applied to the same CTPs. Therefore, it is indeed a fair performance comparison of the proposed symbolic analysis with existing algorithms (on the same CTP models). However, we also

Table 1. Experimental results of predicting assertion failures (MO – memory out 800 MB)

The Test Program				The Given Trace		The Analysis Time (s)			Run Time (s)	
program name	threads	shared /	vars	property	length	slicing	predict	predict-cb	BMC	Explicit
banking-2	2	97 /	264	passed	843	1.4	0.1	0.1	0.3	36.5
banking-2a	2	97 /	264	error	843	1.4	0.1	0.1	7.2	1.2
banking-5	5	104 /	331	passed	1622	1.7	0.3	0.1	2.7	>600
banking-5a	5	104 /	331	error	1622	1.7	0.1	0.1	>600	1.8
banking-10	10	114 /	441	passed	2725	7.0	1.6	0.6	31.8	>600
banking-10a	10	114 /	441	error	2725	7.0	0.1	0.1	MO	2.8
indexer-10	10	285 /	539	passed	3000	1.1	0.1	0.1	0.1	12.8
indexer-15	15	305 /	669	passed	4277	2.3	0.1	0.1	>600	>600
indexer-15a	15	305 /	669	error	4277	2.2	0.4	0.2	>600	>600
indexer-20	20	325 /	799	passed	5647	4.0	0.4	0.1	MO	>600
indexer-20a	20	325 /	799	error	5647	4.1	3.2	0.7	MO	>600
indexer-25	25	345 /	829	passed	7482	6.0	0.9	0.1	MO	>600
indexer-25a	25	345 /	829	error	7482	6.1	26.1	9.8	MO	>600

would like to note that both BMC and DPOR are more general algorithms in that they are capable of checking more generic properties (e.g. LTL formulas).

The slicing in our experiments is thread-sensitive and the traces after slicing consist of mostly irreducible shared variable accesses – for each access, there exists at least one conflicting access from a concurrent thread. The number of equivalence classes of interleavings is directly related to the number of such shared accesses (worst-case double-exponential [QR05]). In the *indexer* examples, for instance, since there is no hash table collision with fewer than 12 threads, the problem is easier to solve. (In [FG05], such cases were used to showcase the power of the DPOR algorithm in dynamically detecting these non-conflicting variable accesses). However, when the number of threads is set to 15, 20, and 25, the number of collisions increases rapidly. Our results show that purely explicit algorithms, even with DPOR, do not scale well in such cases. This is likely a bottleneck for other explicit enumeration based approaches as well. The BMC algorithm did not perform well because of its large formula sizes as a result of explicitly unrolling the transition relation of the CTP. In contrast, our symbolic algorithm remains efficient in navigating the large search space.

9.2. Predicting Atomicity Violations

We have conducted experiments using the following benchmarks: The first set of examples mimic two known concurrency bug patterns (c.f. [LTQZ06]). 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 the parameterized *bank* examples [FNU03]. 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 2 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 (*pavs*), the number of violations after pruning using a simple static must-happen-before analysis (*hb-*

Table 2. Experimental results of predicting atomicity violations

The Test Program			The Given Trace			The Symbolic Analysis				w/o Data
name	thrds	svars	simplify/	original	regions	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

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 *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* [FM09b], 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.

9.3. Predicting Errors in Java Traces

For these experiments, our tool parses a set of execution traces logged at runtime by a modified Java virtual machine. The Java programs are all from the public domain [hi, hr, HP00, vPG04]. Note that most basic synchronization primitives in Java are very similar to their counterparts in *PThreads*; therefore both kinds of traces can be analyzed in a unified symbolic encoding framework. In particular, the Java `synchronized` block is modeled explicitly as a pair of lock/unlock statements over this block's intrinsic lock, which is a reentrant mutex lock.

Table 3 shows the experimental results on predicting atomicity violations in execution traces of Java programs. The first two columns show the test program name and the number of threads. The next five columns show the statistics of the given trace, including the number of all events, the number of lock events, the number of wait-notify events, the number of lock variables, and the number of condition variables. The next five columns show the results of predicting three-access atomicity violations, using a combination of conservative static analysis (pavs, ls-pavs, hb-pavs) and symbolic analysis (sym-pavs). The first three columns are the total number of warnings (potential atomicity violations), the warnings remained after a lockset based analysis (ls-pavs), and the warnings remained after a must-happens-before analysis (hb-pavs). Our symbolic analysis is applied to the potential violations remained after both hb-pavs and ls-pavs. The last two columns show the results of this symbolic analysis, including the number of concrete witnesses and the symbolic analysis time in seconds.

Note that in order to detect atomicity violations, the user-intended transactions need to be marked explicitly in the traces. For these Java traces, we have assumed that all the `synchronized` blocks are intended to be atomic, unless there is a *wait* in the block (in which case it is clearly not atomic).

The Test Program			The Given Trace				The Symbolic Analysis				
name	thrds	events	lk-v	wn-v	lk-evs	wn-evs	pavs	ls-pavs	hb-pavs	sym-avs	sym-time(s)
ex.race	3	29	4	0	2	0	2	2	0	0	0.0
ex.norace	3	37	8	0	2	0	2	2	0	0	0.0
ra.Main	3	55	12	5	3	4	2	2	0	0	0.0
connectionpool	4	97	16	5	1	3	30	6	4	0	0.0
liveness.BugG	7	285	39	6	9	6	280	60	220	0	0.0
s1.JGFBarrier	10	649	62	21	2	7	852	102	612	0	1.6
s1.JGFBarrier	13	799	77	28	2	7	950	87	709	0	3.7
account.Main	11	902	146	12	21	10	140	140	60	2	1.3
philo.Philo	6	1141	126	41	6	22	413	81	177	0	0.0
s1.JGFSyncB	16	1510	237	0	2	0	13578	186	11532	0	0.0
account.Main	21	1747	282	20	41	20	280	280	120	3	5.9
elevator.E	4	3000	368	0	11	0	6	4	2	0	0.0
elevator.E	4	4998	587	0	11	0	12	8	4	0	0.0
elevator.E	4	8000	1126	0	11	0	18	12	6	0	0.0
tsp.Tsp	4	45653	20	5	5	3	0	0	0	0	0.0

Table 3. Predicting atomicity violations in traces of Java programs.

In all cases, the time spent in conservative static analysis is negligible in comparison to the symbolic analysis time. The results also show that, if one relies on conservative static analysis only, whether it is based on lockset analysis or happen-before analysis, the number of reported warnings (atomicity violations) would be large. In this table, we have noticed that in some certain cases (ex.race, ex.norace, ra.Main, and tsp.Tsp), the conservative static analysis results are as good as symbolic analysis, while in many other cases, the number of bogus warnings is unacceptably large. In contrast, the reported (real) violations in the symbolic predictive analysis are significantly fewer.

10. Related Work

The fundamental concept used in this paper is the partial order over the events of an execution trace. This is related to the *happens-before* causality introduced by Lamport in [Lam78]. However, the original form of Lamport's happens-before causality is fairly restricted for the purpose of predicting bugs, e.g. if in the given trace, one lock-unlock region is executed before another lock-unlock region, then swapping the execution order of these two critical sections is not allowed. Although various sound causal models, including [SRA05, CR07, SCR08], have been proposed to relax the original happens-before causality, they may still miss real bugs that would have been caught by our method – this has been illustrated in Section 2. In terms of implementation, our encoding of the HB constraints to specify the execution order among events is related to, but is more abstract than, the logical clocks [Lam78] and the vector clocks [Fid91].

For predicting concurrency errors, there have been many efforts including static analysis (e.g. [FQ03, FM06]), runtime monitoring (e.g. [XBH05, LTQZ06, WS06, FM08, FFY08]), and runtime prediction (e.g. [FF04, FM09b, FM09a, CR07, SCR08, SFF09]). In addition, Lu *et al.* [LTQZ06] used access interleaving invariants to capture patterns of test runs and then monitor production runs for detecting three-access atomicity violations. Xu *et al.* [XBH05] 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. Wang and Stoller [WS06] 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. In [FM06], Farzan and Madhusudan introduced the notion of *causal atomicity*; subsequently they used lockset analysis and lock acquisition histories over execution traces to predict causal atomicity violations [FM09a, FM09b]. In a more recent paper [KW10],

this analysis was generalized using a Universal Causality Graph (UCG) which considers not only arbitrary locks but also other synchronization primitives.

Our CSSA-based symbolic encoding (earlier versions appeared in [WKGG09, WLGG10]) is significantly different from the standard SSA-based SAT encoding in [CKL04], which is popular for verifying *sequential* programs. Our context-bounded analysis differs from the context-bounded analysis in [RG05, LR08, LQR09] 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 [BAM07], which explicitly encodes ordering between all pairs of events in pure Boolean logic. In [JHN05], 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 symbolic methods were applied to whole programs and not to concurrent trace programs (CTPs).

Using SMT solvers in our method do not impose any theoretical limitation because 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 [BNRS08], 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.

At a higher level, our work also relates to the various dynamic model checking algorithms [God05, MQ06, YCG08, WYGG08, YCGW09, WSG11]. However, these algorithms need to re-execute the program when exploring different interleavings, and except for [WYGG08] they are often not property-directed. Here our goal is to detect errors *without* re-executing the program. In our previous work [WCGY09], we have used the notion of concurrent trace program but the goal was to prune the search space in dynamic model checking. In this work, we use the CTP and the CSSA-based encoding for predictive analysis. To our knowledge, this is the first attempt at symbolic predictive analysis.

11. Conclusions

In this paper, we propose a symbolic algorithm for detecting concurrency errors in all feasible permutations of events of a give execution trace. The algorithm uses a succinct concurrent static single assignment (CSSA) based encoding to generate an SMT formula such that the violation of a property exists iff the SMT formula is satisfiable. It facilitates a constraint-based modeling where various synchronization primitives or concurrency semantics are handled easily and uniformly. We also propose a symbolic method to bound the number of context switches in an interleaving. Besides predictive analysis, our CSSA-based encoding can be very useful in many other contexts since it is general enough to handle any structurally bounded concurrent program.

12. Acknowledgments

We thank Fang Yu for his help with implementing the CSSA-based encoding. We also thank Sriram Sankaranarayanan, Rajeev Alur, Nishant Sinha and the annonymous reviewers for their critique of the draft.

References

[BAM07] S. Burckhardt, R. Alur, and M. Martin. CheckFence: checking consistency of concurrent data types on relaxed memory models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 12–21, 2007.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207. Springer, 1999. LNCS 1579.

[BNRS08] Nels Beckman, Aditya V. Nori, Sriram K. Rajamani, and Robert J. Simmons. Proofs from tests. In *International Symposium on Software Testing and Analysis*, pages 3–14, 2008.

[CKL04] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 168–176, 2004. LNCS 2988.

[CR07] F. Chen and G. Rosu. Parametric and sliced causality. In *International Conference on Computer Aided Verification*, pages 240–253. Springer, 2007. LNCS 4590.

[DdM06] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *International Conference on Computer Aided Verification*, pages 81–94. Springer, 2006. LNCS 4144.

[FF04] Cormac Flanagan and Stephen N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Parallel and Distributed Processing Symposium*, 2004.

[FFY08] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 293–303, 2008.

[FG05] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 110–121, 2005.

[Fid91] Colin J. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, 1991.

[FM06] Azadeh Farzan and P. Madhusudan. Causal atomicity. In *International Conference on Computer Aided Verification*, pages 315–328, 2006. LNCS 4144.

[FM08] Azadeh Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In *International Conference on Computer Aided Verification*, pages 52–65, 2008. LNCS 5123.

[FM09a] Azadeh Farzan and P. Madhusudan. The complexity of predicting atomicity violations. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 155–169, 2009.

[FM09b] Azadeh Farzan and P. Madhusudan. Meta-analysis for atomicity violations under nested locking. In *International Conference on Computer Aided Verification*, pages 248–262, 2009. LNCS.

[FNU03] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. In *Parallel and Distributed Processing Symposium*, page 286, 2003.

[FQ03] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 338–349, 2003.

[God05] P. Godefroid. Software model checking: The VeriSoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005.

[hi] http://research.microsoft.com/qadeer/cav issta.htm. Joint cav/issta special even on specification, verification, and testing of concurrent software.

[HP00] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Software Tools for Technology Transfer*, 2(4), 2000.

[hr] http://www2.epcc.ed.ac.uk/computing/ research_activities/java_grande/index_1.html. The java grande forum benchmark suite.

[IYS+05] F. Ivančić, Z. Yang, I. Shlyakhter, M.K. Ganai, A. Gupta, and P. Ashar. F-SOFT: Software verification platform. In *Computer-Aided Verification*, pages 301–306. Springer, 2005. LNCS 3576.

[JHN05] T. Jussila, K. Heljanlo, and I. Niemelä. BMC via on-the-fly determinization. *International Journal on Software Tools for Technology Transfer*, 7(2):89–101, 2005.

[KIG05] Vineet Kahlon, Franjo Ivancic, and Aarti Gupta. Reasoning about threads communicating via locks. In *International Conference on Computer Aided Verification*, pages 505–518, 2005. LNCS 3576.

[KW10] Vineet Kahlon and Chao Wang. Universal Causality Graphs: A precise happens-before model for detecting bugs in concurrent programs. In *International Conference on Computer Aided Verification*, pages 434–449. Springer, 2010. LNCS 6174.

[KWG09] V. Kahlon, C. Wang, and A. Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *International Conference on Computer Aided Verification*, pages 398–413, 2009.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[LPM99] J. Lee, D. Padua, and S. Midkiff. Basic compiler algorithms for parallel programs. In *Principles and Practice of Parallel Programming*, pages 1–12, 1999.

[LQ08] S. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 171–182, 2008.

[LQR09] Shuvendu Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. Static and precise detection of concurrency errors in systems code using SMT solvers. In *International Conference on Computer Aided Verification*, pages 509–524, 2009.

[LR08] Akash Lal and Thomas W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. In *International Conference on Computer Aided Verification*, pages 37–53, 2008. LNCS 5123.

[LTQZ06] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2006.

[MQ06] M. Musuvathi and S. Qadeer. CHESS: Systematic stress testing of concurrent software. In *Logic-Based Program Synthesis and Transformation*, pages 15–16. Springer, 2006. LNCS 4407.

[NMRW02] G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of c programs. In *International Conference on Compiler Construction*, pages 213–228, 2002. LNCS 2304.

[QR05] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 93–107. Springer, 2005.

[RG05] I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs. In *International Conference on Computer Aided Verification*, pages 82–97, 2005. LNCS 2988.

[SBN+97] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[SCR08] Traian Florin Serbănută, Feng Chen, and Grigore Rosu. Maximal causal models for multithreaded systems. Technical Report UIUCDCS-R-2008-3017, University of Illinois at Urbana-Champaign, 2008.

[SFF09] Caitlin Sadowski, Stephen N. Freund, and Cormac Flanagan. Singletrack: A dynamic determinism checker for multithreaded programs. In *European Symposium on Programming*, pages 394–409, 2009.

[SRA05] K. Sen, G. Rosu, and G. Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *Formal Methods for Open Object-Based Distributed Systems*, pages 211–226, 2005.

[SWYS11] M. Said, C. Wang, Z. Yang, and K. Sakallah. Generating data race witnesses by an SMT-based analysis. In *NASA Formal Methods Symposium*, 2011.

[vPG04] C. von Praun and T. R. Gross. Static detection of atomicity violations in object-oriented programs. *Object Technology*, 3(6), 2004.

[WCGY09] C. Wang, S. Chaudhuri, A. Gupta, and Y. Yang. Symbolic pruning of concurrent program executions. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 23–32, 2009.

[WGG06] C. Wang, A. Gupta, and M. Ganai. Predicate learning and selective theory deduction for a difference logic solver. In *Design Automation Conference*, pages 235–240. ACM, 2006.

[WIGG05] C. Wang, F. Ivančić, M. Ganai, and A. Gupta. Deciding separation logic formulae by SAT and incremental negative cycle elimination. In *Logic for Programming Artificial Intelligence and Reasoning*, pages 322–336. Springer-Verlag, 2005. LNCS 3835.

[WKGG09] C. Wang, S. Kundu, M. Ganai, and A. Gupta. Symbolic predictive analysis for concurrent programs. In *International Symposium on Formal Methods*, pages 256–272, 2009.

[WLGG10] C. Wang, R. Limaye, M. Ganai, and A. Gupta. Trace-based symbolic analysis for atomicity violations. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2010.

[WS06] Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Software Eng.*, 32(2):93–110, 2006.

[WSG11] C. Wang, M. Said, and A. Gupta. Coverage driven systematic concurrency testing. In *International Conference on Software Engineering*, 2011.

[WYGG08] C. Wang, Y. Yang, A. Gupta, and G. Gopalakrishnan. Dynamic model checking with property driven pruning to detect race conditions. In *Automated Technology for Verification and Analysis*, 2008.

[WYKG08] C. Wang, Z. Yang, V. Kahlon, and A. Gupta. Peephole partial order reduction. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 382–396, 2008.

[XBH05] Min Xu, Rastislav Bodík, and Mark D. Hill. A serializability violation detector for shared-memory server programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–14, 2005.

[YCG08] Y. Yang, X. Chen, and G. Gopalakrishnan. Inspect: A runtime model checker for multithreaded C programs. Technical Report UUCS-08-004, University of Utah, 2008.

[YCGW09] Y. Yang, X. Chen, G. Gopalakrishnan, and C. Wang. Automatic discovery of transition symmetry in multithreaded programs using dynamic analysis. In *SPIN workshop on Software Model Checking*, 2009.