Symbolic Predictive Analysis for Concurrent Programs

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

[1] NEC Laboratories America, Princeton, NJ, USA
[2] University of California, San Diego, La Jolla, 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 happens-before causality relations have been proposed for predictive analysis to improve the interleaving coverage while ensuring the absence of false alarms. However, these models are based on only the observed events, and typically do not utilize source code. Furthermore, the enumerative algorithms they use for verifying safety properties in the predicted execution traces often suffer from the interleaving explosion problem. In this paper, we introduce a new symbolic causal model based on source code and the observed events, and propose a symbolic algorithm to check whether a safety property holds in all feasible permutations of events in the given execution trace. Rather than explicitly enumerating the interleavings, our algorithm conducts the verification using a novel encoding of the causal model and symbolic reasoning with a satisfiability modulo theory (SMT) solver. Our algorithm has a larger interleaving coverage than known causal models in the literature. We also propose a method to *symbolically bound* the number of context switches allowed in an interleaving, to further improve the scalability of the algorithm.

1 Introduction

Predictive analysis aims at detecting concurrency errors by observing execution traces of a concurrent program which may be non-erroneous. Due to the inherent nondeterminism in scheduling concurrent processes/threads, executing a program with the same test input may lead to different program behaviors. This poses a significant challenge in testing concurrent programs—even if a test input may cause a failure, the erroneous 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. In predictive analysis, a concrete execution trace is given, together with a correctness property in the form of assertions embedded in the trace. The given execution trace need not violate the property, but there may exist an alternative trace, i.e., a feasible permutation of events of the given trace, that violates the property. The goal of predictive analysis is detecting such erroneous traces by *statically* analyzing the given execution trace without re-executing the program.

Existing predictive analysis algorithms can be classified into two categories based on the quality of reported bugs. The first category consists of methods that do not miss real errors but may report bogus errors. Historically, algorithms that are based on lockset analysis [1–3] fall into the first category. They strive to cover all possible interleavings that are feasible permutations of events of the given trace, but at the same time may introduce some interleavings that can never appear in the actual program execution.

The second category consists of methods that do not report bogus errors but may miss some real errors. Various causal models have been used by these methods [4–6], with some inspired by Lamport's happens-before causality relation [7]. They provide the *feasibility guarantee*–that all the reported erroneous interleavings are actual program executions, but they do not cover all interleavings allowed by the program source code.

This paper also focuses on predictive analysis algorithms with the feasibility guarantee. The given execution trace is regarded as a total order on the events appearing in the trace. Based on happens-before, one can derive a causal model—a partial order of events—which admits not only the given trace but also many alternative permutations. However, two significant problems need to be solved. First, checking all the feasible interleavings allowed by a causal model for property violations is still a bottleneck. Despite the long quest for more coverage in causal models, little has been done to improve the underlying checking algorithms. Existing methods [4–6] often rely on explicit enumeration of the predicted interleavings, which does not scale when the number of interleavings is large. In reality, the more general a causal model is, the larger the number of interleavings it admits. Second, these 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 code that produce the event is not known. Consequently, often unnecessarily strong happens-before causality is imposed to achieve the desired feasibility guarantee.

In this paper, we propose a *symbolic* predictive analysis algorithm to address these two problems. We assume that the source code is available for instrumentation to obtain *symbolic events* at runtime. We introduce a symbolic causal model based on program source code and observed events in a trace, to achieve the goal of covering more interleavings. This also facilitates a constraint-based modeling where various concurrency primitives or semantics (locks, semaphores, happens-before, sequential consistency, etc.) are handled easily and uniformly. More specifically, we make the following contributions:

- We introduce a *concurrent trace program* as a symbolic predictive model to capture feasible interleavings that can be predicted from a given execution trace.
- We propose a safety property checking algorithm using a concurrent static single assignment (CSSA) based encoding and symbolic reasoning with a SMT solver. The symbolic search automatically captures property- or goal-directed pruning, through conflict analysis and learning features in modern SMT solvers.
- We propose a simple method to symbolically bound the number of context switches in an interleaving, which further improves the scalability of our symbolic algorithm.

If desired, our symbolic algorithm can be further constrained to match the interleaving coverage of known causal models in the literature. In effect, our new model has a larger interleaving coverage than the existing models.

The remainder of this paper is organized as follows. In Section 2, we provide a motivating example and illustrate our ideas. In Section 3, we define execution traces and our predictive model. In Section 4, we present the SMT-based symbolic property checking algorithm. In Section 5, we present the symbolic encoding to enforce context-bounding. In Section 6, we demonstrate how our algorithm can be constrained to match a more restrictive causal model [6]. We present our experimental results in Section 7. We review related work in Section 8 and give our conclusions in Section 9.

2 Motivating Example

Fig. 1 shows a multithreaded program execution trace, modified from an example in [6]. There are two concurrent threads T_1 and T_2, three shared variables x, y and z, two thread-local variables a and b, and a counting semaphore l. The semaphore l can be viewed as an integer variable initialized to 1: $acq(l)$ acquires the semaphore when $(l > 0)$ and decreases l by one, while $rel(l)$ releases the semaphore and increases 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 correctness property is specified as an assertion in t_{12}. 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 [7, 4–6], can predict this error. By sound, we mean that the predictive technique does not generate false alarms (most of the 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, which are over the same shared variable, must be respected. It means that event t_8 must be executed before t_{10} and event t_7 must be executed before t_{11}. These *happens-before* constraints are sufficient but often not necessary to ensure that the admitted traces are feasible—many other feasible interleavings are left out.

Various causal models proposed subsequently aimed at lifting some of these happens-before constraints without jeopardizing the feasibility guarantee [4–6]. However, when applied to the example in Fig. 1, none of them can predict the erroneous trace $\rho' = (t_1$–$t_4)t_9t_{10}t_{11}t_{12}t_{13}(t_5$–$t_8)$. Consider, for example, the *maximal causal model* in [6]. The model relies on the axioms of semaphore and sequential consistency and is general enough to subsume other known causal models. This model allows all the classic happens-before constraints to be lifted, except for the constraint stating that event t_7 must happen before t_{11}. As a result, the model in [6] cannot be used to predict the error in ρ'.

The reason these sound models cannot predict the error in Fig. 1 is that they model events in ρ as the concrete values read from or written to shared variables. Such *concrete events* are tied closely to the given trace. 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)`, `if(x>1)`, and even assignment `b:=x`, may produce the same event. Consequently, unnecessarily strong happens-before constraints are imposed over event t_{11} to ensure the feasibility of all 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 capturing abstract values (e.g. relevant predicates). For instance, we model event t_{11} in Fig. 1 as $\mathsf{assume}(x > b)$, where $\mathsf{assume}(c)$ means the 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. We do not use the happens-before causality to define the set of admitted traces. Instead, we allow all possible interleavings of these symbolic events as long as the sequential consistency semantics of a concurrent program execution is respected. 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'.

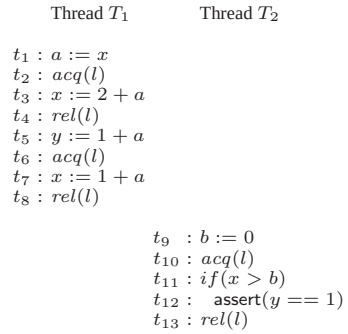

Fig. 1. The sequence of executed program statements (x=y=0 initially)

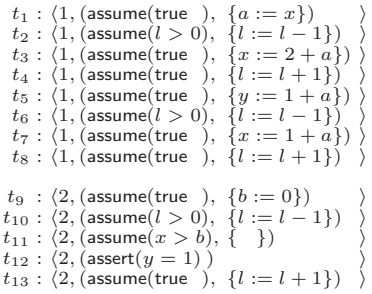

Fig. 2. The symbolic representation of the execution trace (x=y=0 initially)

3 Preliminaries

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

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 a 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.

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

- $(\mathsf{assume}(c), asgn)$ 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 multiple times, e.g., when it is inside a loop or a routine executed by multiple threads, each execution instance is modeled as a separate event.

By defining the expression syntax suitably, the symbolic trace representation can model the execution of any shared-memory multithreaded program. Details on modeling generic C/C++ language constructs are not directly related to concurrency; for more information refer to recent efforts in [8–10].

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 is the foundation of all types of concurrency primitives. For example, acquiring a counting semaphore l can be modeled as the action $(\mathsf{assume}(l > 0), \{l := l - 1\})$.

Example. Fig. 2 shows an example symbolic execution trace representation, which corresponds to 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 with $\mathsf{assume}(\mathsf{true})$. The if-statement in t_{11} is modeled 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, $1 \leq i \leq k$, iff one of the following conditions holds:

- $t = \langle i, (\mathsf{assume}(c), asgn)\rangle$, $s[c]$ is true, and for each assignment $lval := exp$ in $asgn$, $s'[lval] = s[exp]$ 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 a symbolic execution trace of a concurrent program P. It defines a total order on the symbolic events. From ρ we can derive a partial order called the concurrent trace program (CTP).

Definition 1. *The* concurrent trace program *of ρ is a partially ordered set $CTP_\rho = (T, \sqsubseteq)$ such that,*

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

In the sequel, we will say a transition $t \in CTP_\rho$ to mean that $t \in T$ is associated with the 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 allowing events from different threads to remain un-ordered with each other is the crucial difference from existing sound causal models [7, 4–6].

We guarantee the feasibility of predicted traces through the notion of *feasible linearizations* of CTP_ρ. A linearization of this partial order is an alternative interleaving

of events in ρ. 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 $s_{i-1} \xrightarrow{t'_i} s_i$. Note that this definition captures the standard sequential consistency semantics for concurrent programs, where we modeled concurrency primitives such as locks by using auxiliary shared variables in atomic guarded assignment events.

4 Symbolic Predictive Analysis Algorithm

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 an encoding that creates the formula in a quantifier-free first-order logic to facilitate the application of off-the-shelf SMT solvers [11].

4.1 Concurrent Static Single Assignment

Our encoding is based on transforming the concurrent trace program into a concurrent static single assignment (CSSA) form, inspired by [12]. 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 (an `assume` or the `assert`) or the right-hand-side of an assignment.

Unlike in the classic sequential SSA form, we need not add ϕ-functions to model the confluence of multiple if-else branches because in 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, 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 2. *A π-function, introduced for a shared variable v immediately before its use, has the form $\pi(v_1, \dots, v_l)$, 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 $v' \leftarrow \pi(v_1, \dots, v_l)$. Then replace v with the new definition v'.

Example. Fig. 3 shows the CSSA form of the CTP in Fig. 2. We add 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 denotes 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.

$$
\begin{array}{lll}
t_0 : \langle 1, (\mathsf{assume}(\mathsf{true}\quad), & \{x_0 := 0, y_0 := 0, l_0 := 1\}) \rangle & \\
t_1 : \langle 1, (\mathsf{assume}(\mathsf{true}\quad), & \{a_1 := \pi^1\}) & \rangle \text{ where } \pi^1 \leftarrow \pi(x_0) \\
t_2 : \langle 1, (\mathsf{assume}(\pi^2 > 0\,), & \{l_1 := \pi^2 - 1\}) & \rangle \text{ where } \pi^2 \leftarrow \pi(l_0, l_5, l_6) \\
t_3 : \langle 1, (\mathsf{assume}(\mathsf{true}\quad), & \{x_1 := 2 + a_1\}) & \rangle \\
t_4 : \langle 1, (\mathsf{assume}(\mathsf{true}\quad), & \{l_2 := \pi^3 + 1\}) & \rangle \text{ where } \pi^3 \leftarrow \pi(l_1, l_5, l_6) \\
t_5 : \langle 1, (\mathsf{assume}(\mathsf{true}\quad), & \{y_1 := 1 + a_1\}) & \rangle \\
t_6 : \langle 1, (\mathsf{assume}(\pi^4 > 0\,), & \{l_3 := \pi^4 - 1\}) & \rangle \text{ where } \pi^4 \leftarrow \pi(l_2, l_5, l_6) \\
t_7 : \langle 1, (\mathsf{assume}(\mathsf{true}\quad), & \{x_2 := 1 + a_1\}) & \rangle \\
t_8 : \langle 1, (\mathsf{assume}(\mathsf{true}\quad), & \{l_4 := \pi^5 + 1\}) & \rangle \text{ where } \pi^5 \leftarrow \pi(l_3, l_5, l_6) \\
 & & \\
t_9 : \langle 2, (\mathsf{assume}(\mathsf{true}\quad), & \{b_1 := 0\}) & \rangle \\
t_{10} : \langle 2, (\mathsf{assume}(\pi^6 > 0\,), & \{l_5 := \pi^6 - 1\}) & \rangle \text{ where } \pi^6 \leftarrow \pi(l_0, l_1, l_2, l_3, l_4) \\
t_{11} : \langle 2, (\mathsf{assume}(\pi^7 > b_1), & \{\quad\}) & \rangle \text{ where } \pi^7 \leftarrow \pi(x_0, x_1, x_2) \\
t_{12} : \langle 2, (\mathsf{assert}(\pi^8 = 1)\,) & & \rangle \text{ where } \pi^8 \leftarrow \pi(y_0, y_1) \\
t_{13} : \langle 2, (\mathsf{assume}(\mathsf{true}\quad), & \{l_6 := \pi^9 + 1\}) & \rangle \text{ where } \pi^9 \leftarrow \pi(l_0, l_1, l_2, l_3, l_4, l_5)
\end{array}
$$

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

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 event t_j that defines v_j, $1 \leq i \leq l$ and $j \neq i$, is executed either before the definition t_i or after the use 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:

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

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

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

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

– **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 action assert(c), which specifies the correctness property.

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

Intuitively, the assertion condition c must hold if t is executed. Recall that Φ_{PRP} is negated in Φ_{CTP_ρ} to search for property violations.

Example Fig. 4 illustrates the CSSA-based encoding of the example in Fig. 3, where the subformulas that form Φ_{PO} and Φ_{VD} are listed. In the figure, t_0, t_{14} are the dummy 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:

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

Note that some HB-constraints evaluate to constant false and true—such simplification is frequent and is performed in our implementation to reduce the formula size.

Path Conditions:		Program Order:	Variable Definitions:
t_0 :			$x_0 = 0 \wedge y_0 = 0 \wedge l_0 = 1$
t_1 :	$g_1 = \mathsf{true}$	$HB(t_0, t_1)$	$a_1 = \pi^1$
t_2 :	$g_2 = g_1 \wedge (\pi^2 > 0)$	$HB(t_1, t_2)$	$l_1 = \pi^2 - 1$
t_3 :	$g_3 = g_2$	$HB(t_2, t_3)$	$x_1 = 2 + a_1$
t_4 :	$g_4 = g_3$	$HB(t_3, t_4)$	$l_2 = \pi^3 + 1$
t_5 :	$g_5 = g_4$	$HB(t_4, t_5)$	$y_1 = 1 + a_1$
t_6 :	$g_6 = g_5 \wedge (\pi^4 > 0)$	$HB(t_5, t_6)$	$l_3 = \pi^4 - 1$
t_7 :	$g_7 = g_6$	$HB(t_6, t_7)$	$x_2 = 1 + a_1$
t_8 :	$g_8 = g_7$	$HB(t_7, t_8)$	$l_4 = \pi^5 + 1$
t_9 :	$g_9 = \mathsf{true}$	$HB(t_0, t_9)$	$b_1 = 0$
t_{10} :	$g_{10} = g_9 \wedge (\pi^6 > 0)$	$HB(t_9, t_{10})$	$l_5 = \pi^6 - 1$
t_{11} :	$g_{11} = g_{10} \wedge (\pi^7 > b_1)$	$HB(t_{10}, t_{11})$	
t_{12} :	$g_{12} = g_{11}$	$HB(t_{11}, t_{12})$	$l_6 = \pi^9 + 1$
t_{13} :	$g_{13} = g_{13}$	$HB(t_{12}, t_{13})$	
t_{14} :		$HB(t_8, t_{14}) \wedge HB(t_{13}, t_{14})$	

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

Let n be the number of events in a CTP, n_π be the number of shared variable uses, and l_π be the maximal number of parameters in any π-function. Our encoding produces a formula of size $O(n + n_\pi \times l_\pi^2)$. Although in the worst case—when each event *reads and writes* shared variables—the size becomes $O(n^3)$, it is rare in realistic applications. The reason is that shared variable accesses in a concurrent program are often kept few and far in between, especially when compared to computations within threads, to minimize the synchronization overhead. In contrast, conventional bounded model checking (BMC) algorithms, e.g. [13], would generate significantly larger formulas. To cover all feasible interleavings in a CTP, the BMC unrolling depth needs to be n, which results in the same $O(n^3)$ complexity. However, the BMC formula size cannot be easily reduced even if l_π and n_π are significantly smaller than n. In Section 7, we will present experimental comparison of our CSSA-based encoding with the BMC algorithm in [13].

4.3 Proof of Correctness

Recall that for two arbitrary events t and t', the constraint $HB(t, t')$ denote that t must be executed before t'. Consider a model where we introduce for each event $t \in CTP$ a fresh integer variable $\mathcal{O}(t)$ denoting its 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 as follows:

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

We now state the correctness of our encoding.

Theorem 1. *Formula Φ_{CTP} is satisfiable iff there exists a feasible linearization of the CTP that violates the assertion property.*

Proof: The encoding closely follows our definitions of CTP, feasible linearizations, and the semantics of π-functions. The proof is straightforward and is omitted for brevity.

5 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 [14]. Several subsequent studies have confirmed [15, 16] that concurrency bugs in practice can often be exposed in interleavings with a surprisingly small number of context switches.

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 $y = 0$).

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

5.1 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*[3].

Second, if events t, t' are from two different threads (and $t \neq t_{\mathsf{first}}$ and $t \neq t_{\mathsf{last}}$), according to our encoding rules, the constraint $HB(t,t')$ must be 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.

5.2 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[4].

Theorem 2. *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 $\Phi_{CTP_\rho}(b)$.

[3] When $HB(t,t')$ is a constant, we replace it with true or false.

[4] In CHESS [15], whose exploration algorithm is purely explicit rather than symbolic, a variant is used to count only the *preemptive* context switches.

5.3 Lifting the CB Constraint

In the 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.

6 Relating to Other Causal Models

In this section, we show that our symbolic algorithm can be further constrained to match known causal models in the literature. By doing this exercise, we also demonstrate that our algorithm has a larger interleaving coverage. Since the maximal causal model [6], proposed recently by Serbănută, Chen and Rosu, has the capability of capturing more feasible interleavings than prior sound causal models, we will use it as an example. In this case, our algorithm provides a symbolic property checking algorithm, in contrast to their model checking algorithm based on explicit enumeration.

We assume that during the program execution, only events involving shared objects are monitored, and except for synchronization primitives, the program code that produces the events are not available. Therefore, an event is in one of the following forms:

- A concurrency synchronization/communication 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);

Example. We slightly modify the example 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. 5. 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).

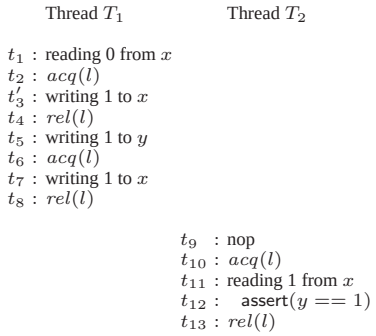

Fig. 5. The concrete event sequence

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

Fig. 6. The reduced causal model

Let $CTP_\rho = (T, \sqsubseteq)$ be the model as in Definition 1. We derive the constrained model CM_ρ as shown in Fig. 6. 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 underapproximated 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.

Semantics of the Constrained CTP. 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.

7 Experiments

We have implemented the proposed symbolic predictive analysis algorithm in a tool called *Fusion*. Our tool is capable of handling symbolic execution traces generated by arbitrary multi-threaded C programs using the Linux *PThreads* library. We use the *Yices* SMT solver [11] to solve the satisfiability formulas.

We have conducted preliminary experiments using the following benchmarks. The first set consists of C variants of the *banking* example [17] with known bugs due to atomicity violations. Unlike previous work [3, 5, 6], 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 [5, 6] or atomicity violations [3] (which may not cause a violation of the functional property). The second set of benchmarks are the *indexer* examples from [18], 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 irredudant 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. The experiments[5] were conducted on a PC with 1.6 GHz Intel processor and 2GB memory running Fedora 8.

Table 1. Experimental results of symbolic predictive analysis (MO–memory out 800 MB)

The Test Program			The Given Trace		Run Time (s)			Run Time (s)	
program name	threads	shared / vars	property	length	slicing	predict	predict-cb	BMC[13]	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

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 final two columns show the run times of a BMC algorithm [13] with the unrolling depth set to the trace length and an explicit search algorithm enhanced by DPOR [18].

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 [14]). 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 [18], 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, does 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. In contrast, our symbolic algorithm remains efficient in navigating the large search space.

[5] Examples're available at http://www.nec-labs.com/∼chaowang/pubDOC/predict-example.tar.

8 Related Work

The fundamental concept used in this paper is the partial order over the events in an execution trace. This is related to the *happens-before* causality introduced by Lamport in [7]. However, Lamport's happens-before causality, as well as the various subsequent causal models [4–6], has a strictly less interleaving coverage than our model. Our use of the HB constraints to specify the execution order among events is related to, but is more abstract than, the logical clocks [7] and the vector clocks [19, 20].

Our symbolic encoding is related to, but is different from, the SSA-based SAT encoding [8], which is popular for *sequential* programs. We use *difference logic* to directly capture the partial order. This differs from CheckFence [21], which explicitly encodes ordering between all pairs of relevant events (shared variable accesses) in pure Boolean logic. Furthermore, it does not create a causal mode, but directly enumerates the feasible interleavings (with respect to a given memory semantics). TCBMC [22] also uses context-bounding in their symbolic encoding. However, it has to *a priori* fix the number of bounded context switches. In contrast, our method in Section 4 is for the unbounded case—the context-bounding constraint in Section 5 is optional and is used to further improve performance. Furthermore, all the aforementioned methods were applied to whole programs and not to trace programs.

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. Although the formulas may be hard to solve in some cases, the rapid progress in SMT solvers can be directly utilized to improve performance in practice.

At a high level, our work also relates to dynamic model checking [23, 15, 24, 13]. However, these algorithms need to re-execute the program when exploring different interleavings, and in general, they are not property-directed. Our goal is to detect errors *without re-executing the program*. In our previous work [25], 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.

9 Conclusions

In this paper, we propose a symbolic algorithm for detecting concurrency errors in all feasible permutations of events in a give execution trace. The new algorithm uses a succinct concurrent static single assignment (CSSA) based encoding to generate an SMT formula such that the violation of an assertion property exists iff the SMT formula is satisfiable. We also propose a symbolic method to bound the number of context switches in an interleaving. The new algorithm can achieve a better interleaving coverage, and at the same time is more scalable than the explicit enumeration algorithms used by the various existing methods for predictive analysis.

References

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

2. Flanagan, C., Freund, S.N.: Atomizer: A dynamic atomicity checker for multithreaded programs. In: Parallel and Distributed Processing Symposium (IPDPS), IEEE (2004)
3. Wang, L., Stoller, S.D.: Runtime analysis of atomicity for multithreaded programs. IEEE Trans. Software Eng. **32**(2) (2006) 93–110
4. Sen, K., Rosu, G., Agha, G.: Detecting errors in multithreaded programs by generalized predictive analysis of executions. In: Formal Methods for Open Object-Based Distributed Systems, Springer (2005) 211–226 LNCS 3535.
5. Chen, F., Rosu, G.: Parametric and sliced causality. In: Computer Aided Verification, Springer (2007) 240–253 LNCS 4590.
6. Serbănută, T.F., Chen, F., Rosu, G.: Maximal causal models for multithreaded systems. Technical Report UIUCDCS-R-2008-3017, University of Illinois at Urbana-Champaign (2008)
7. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7) (1978) 558–565
8. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Tools and Algorithms for Construction and Analysis of Systems, Springer (2004) 168–176 LNCS 2988.
9. Ivančić, F., Shlyakhter, I., Gupta, A., Ganai, M., Kahlon, V., Wang, C., Yang, Z.: Model checking C program using F-Soft. In: International Conference on Computer Design. (October 2005) 297–308
10. Lahiri, S., Qadeer, S.: Back to the future: revisiting precise program verification using SMT solvers. In: Principles of Programming Languages, ACM (2008) 171–182
11. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for dpll(t). In: Computer Aided Verification, Springer (2006) 81–94 LNCS 4144.
12. Lee, J., Padua, D., Midkiff, S.: Basic compiler algorithms for parallel programs. In: Principles and Practice of Parallel Programming. (1999) 1–12
13. Wang, C., Yang, Z., Kahlon, V., Gupta, A.: Peephole partial order reduction. In: Tools and Algorithms for Construction and Analysis of Systems, Springer (2008) 382–396 LNCS 4963.
14. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Tools and Algorithms for Construction and Analysis of Systems, Springer (2005) 93–107 LNCS 3440.
15. Musuvathi, M., Qadeer, S.: CHESS: Systematic stress testing of concurrent software. In: Logic-Based Program Synthesis and Transformation, Springer (2006) 15–16 LNCS 4407.
16. Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. In: Computer Aided Verification, Springer (2008) 37–53 LNCS 5123.
17. Farchi, E., Nir, Y., Ur, S.: Concurrent bug patterns and how to test them. In: Parallel and Distributed Processing Symposium. (2003) 286
18. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Principles of programming languages. (2005) 110–121
19. Fidge, C.J.: Logical time in distributed computing systems. IEEE Computer **24**(8) (1991) 28–33
20. Mattern, F.: Efficient algorithms for distributed snapshots and global virtual time approximation. J. Parallel Distrib. Comput. **18**(4) (1993) 423–434
21. Burckhardt, S., Alur, R., Martin, M.: CheckFence: checking consistency of concurrent data types on relaxed memory models. In: Programming Language Design and Implementation, ACM (2007) 12–21
22. Rabinovitz, I., Grumberg, O.: Bounded model checking of concurrent programs. In: Computer Aided Verification, Springer (2005) 82–97 LNCS 2988.
23. Godefroid, P.: Software model checking: The VeriSoft approach. Formal Methods in System Design **26**(2) (2005) 77–101
24. Yang, Y., Chen, X., Gopalakrishnan, G.: Inspect: A Runtime Model Checker for Multithreaded C Programs. Technical Report UUCS-08-004, University of Utah (2008)
25. Wang, C., Chaudhuri, S., Gupta, A., Yang, Y.: Symbolic reduction of dynamic executions of concurrent programs. In: (submission)