

# Abstract Analysis of Symbolic Executions

Aws Albarghouthi<sup>1</sup>, Arie Gurfinkel<sup>2</sup>, Ou Wei<sup>1,3</sup>, and Marsha Chechik<sup>1</sup>

<sup>1</sup>Department of Computer Science, University of Toronto, Canada

<sup>2</sup>Software Engineering Institute, Carnegie Mellon University, USA

<sup>3</sup>Nanjing University of Aeronautics and Astronautics, China

**Abstract.** Multicore technology has moved concurrent programming to the forefront of computer science. In this paper, we look at the problem of reasoning about concurrent systems with infinite data domains and non-deterministic input, and develop a method for verification and falsification of safety properties of such systems. Novel characteristics of this method are (a) constructing under-approximating models via symbolic execution with abstract matching and (b) proving safety using under-approximating models.

## 1 Introduction

Concurrency has moved to the forefront of computer science due to the fact that future speedups of software rely on exploiting concurrent executions on multiple processor cores. Thus, the problem of creating correct concurrent programs is now paramount. Reasoning about such programs, i.e., determining whether properties of interest hold or fail in them, has always been difficult, especially if we consider “realistic” programs with infinite data domains (i.e., integer variables) and non-deterministic input. An example of such a program is the simple two-process mutual exclusion protocol shown in Fig. 1, where integer variables  $x$  and  $y$  are set non-deterministically (see Section 2 for more detail).

Approaches to reason about concurrent systems can be split into four categories. (1) “Classical” model-checking techniques, e.g., [19], were created to enumerate all reachable states of the program. Such techniques provide both verification and falsification information and are very effective when the state-space of the program is finite. However, they do not scale well for programs with large state-spaces and do not apply to those with infinite state-spaces. (2) Techniques like [3, 17, 8] build an *over-approximation* of program behaviours, via static analysis. These techniques can handle large/infinite state-spaces, are effective for verification purposes, but are not particularly well suited for finding bugs. (3) Techniques like [25, 24, 4, 22] explore an *under-approximation* of feasible program behaviours. These techniques are often inexpensive and very effective for finding bugs; they are, however, often unable to prove correctness of programs. (4) Recently, researchers have been exploring the combination of under- and over-approximation by combining dynamic and static analysis techniques, respectively. Examples of this approach include [24] and the YOGI project [23]. These techniques are effective both for verification and for falsification of safety

<b>Process 1</b>	<b>Process 2</b>
$t_1 : pc_1 = 1 \longrightarrow b := b + 1, pc_1 := 2$	$t_4 : pc_2 = 1 \longrightarrow b := b + 1, pc_2 := 2$
$t_2 : pc_1 = 2 \wedge x \leq y \wedge b = 2 \longrightarrow pc_1 := 3$	$t_5 : pc_2 = 2 \wedge x > y \wedge b = 2 \longrightarrow pc_2 := 3$
$t_3 : pc_1 = 3 \longrightarrow x := \text{nondet}, pc_1 := 2$	$t_6 : pc_2 = 3 \longrightarrow y := \text{nondet}, pc_2 := 2$

**Fig. 1.** A simple two-process mutual exclusion protocol with inputs  $x$  and  $y$ .

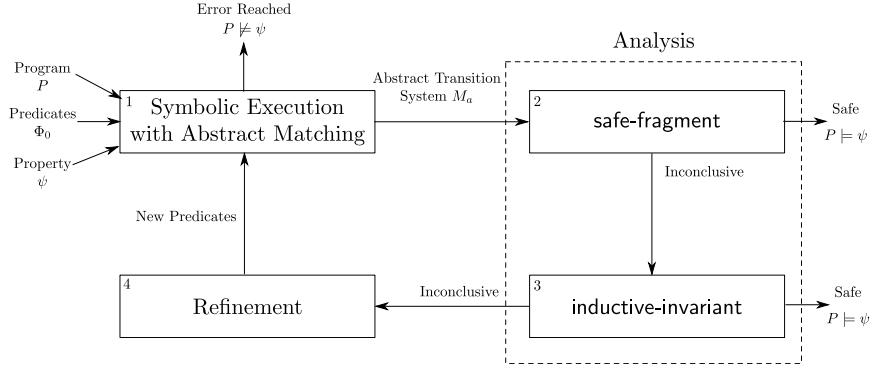
properties but, with the exception of [24], have been limited to sequential programs [27, 16, 20, 14]. Our work fits into this category.

In this paper, we propose a novel approach for automatically checking safety properties of *reactive* concurrent programs (over a finite number of threads) with *non-deterministic input* and *infinite data domains*. Handling these features allows us to target programs with infinite state-spaces, uninitialized variables, and communication with an external environment (e.g., user interaction). Our approach combines symbolic execution (to deal with non-deterministic input) and predicate abstraction (to deal with infinite data domains) in an abstraction-refinement cycle. Symbolic exploration proceeds along a path until it discovers two symbolic states that match to the same abstract state – the process is called *abstract matching* [18]. It produces an under-approximating abstract model that is more precise, in terms of feasible program behaviours it captures, than under-approximation techniques based on *must* transitions [25], concrete model checking and abstract matching [24], and weak reachability [4]. Since we only explore feasible program behaviours, all errors we encounter are real. We then analyse the abstract model to determine if it is also an over-approximation of the reachable concrete program states. If so, we conclude safety; otherwise, we refine the abstraction, adding predicates not to remove spurious counterexamples (as in the CEGAR framework [9]) but to enable us to explore more feasible program behaviours. To our knowledge, this is the first software verification algorithm combining symbolic execution with predicate abstraction and refinement. Our contributions are thus as follows: (i) a novel method for improving precision of under-approximating models by constructing them via a combination of symbolic execution and abstract matching; (ii) a novel technique for proving safety using under-approximating models; (iii) an implementation based on [24] and an empirical evaluation comparing the two approaches.

The rest of this paper is organized as follows. In Section 2, we give a general overview of the approach, illustrating it on the example in Fig. 1. We define the notation and provide background for the remainder of the paper in Section 3. Section 4 presents our approach in more detail, and Section 5 describes our implementation and experimental results. Section 6 compares our approach with related work. We conclude in Section 7 with the summary of our contributions and suggestions for future work.

## 2 Overview

In this section, we illustrate our approach on a simple two-process mutex protocol shown in Fig. 1. The protocol is written in a simple guarded command language. Initially, variables  $x$  and  $y$  are undefined (i.e., they can have an arbitrary value),



**Fig. 2.** Abstract analysis of symbolic executions.

$b$  is 0,  $pc_1$  is 1, and  $pc_2$  is 1. Process 1 starts at  $pc_1 = 1$ , increments  $b$ , and moves to  $pc_1 = 2$  (transition  $t_1$ ). At  $pc_1 = 2$ , it waits until  $b$  becomes 2 and  $x$  is less than or equal to  $y$  and proceeds to its critical section at  $pc_1 = 3$  (transition  $t_2$ ). At  $pc_1 = 3$ , it sets  $x$  non-deterministically (modelling input) and returns to  $pc_1 = 2$  (transition  $t_3$ ). Process 2 behaves analogously but uses process counter  $pc_2$  and resets variable  $y$  in its critical section. We aim to show that this protocol satisfies the mutual exclusion property: a state where  $pc_1 = 3 \wedge pc_2 = 3$  is not reachable.

The high-level overview of our approach is shown in Fig. 2. To determine whether a safety property  $\psi$  holds in a program  $P$ , we compute an abstract transition system,  $M_a$ , of  $P$  w.r.t. some initial set of predicates  $\Phi_0$  using symbolic execution with abstract matching. The state-space of  $M_a$  is an *under-approximation* of reachable abstract states of  $P$ . If an error is found during the symbolic execution step, we report  $P$  as unsafe and terminate. Otherwise,  $M_a \models \psi$ , and  $M_a$  is passed to the analysis phase which checks, via two separate steps, whether the state-space of  $M_a$  is also an *over-approximation* of  $P$ . If so, we are able to conclude that  $P$  is safe. Otherwise, we refine the set of predicates and repeat the entire process.

Our approach follows an abstraction-refinement loop, but differs from the standard CEGAR framework [9] in two ways: (1) we compute an *under-approximating abstraction* of  $P$  (using symbolic execution); (2) we do not rely on counterexamples to perform the refinement. In the rest of this section, we discuss each step of our approach in turn.

**Symbolic Execution with Abstract Matching.** Fig. 3(a) shows a symbolic execution tree of the program in Fig. 1. The initial set of predicates,  $\Phi_0 = \{x \leq y, b = 2\}$ , consists of all the predicates from the guards of the program. A *symbolic state* consists of the current values of variables conjoined with the *path condition* that has to be satisfied in order to reach this state. In Fig. 3(a), each state is represented as a box, with values of variables in the order  $(pc_1, pc_2, x, y, b)$  appearing in the top and the path condition – in the bottom. For example, state  $s_1$  is  $(pc_1 = 1, pc_2 = 1, x = x_0, y = y_0, b = 0) \wedge (x_0 \leq y_0)$ , where  $x_0$  and  $y_0$  are symbolic constants representing the initial value of  $x$  and  $y$ , respectively.

We use traditional symbolic execution with one additional constraint: in each symbolic state, each predicate from  $\Phi_0$  must be either satisfied or refuted. If necessary, we split a symbolic state by strengthening its path condition. For example, the initial state of the program in Fig. 1,  $s_0 = (pc_1 = 1, pc_2 = 1, x = x_0, y = y_0, b = 0)$ , neither satisfies nor refutes the predicate  $x \leq y$ . Thus, it is split into states  $s_1$  and  $s_2$  that satisfy and refute  $x \leq y$ , respectively. They become the new initial states. Similarly, states  $s_5$  and  $s_6$  are obtained by splitting a symbolic successor of  $s_4$ . Our constraint may increase the number of symbolic states, but it ensures that each symbolic state corresponds to (or matches with) a unique valuation of all of the predicates in  $\Phi_0$ . We call such a valuation an *abstract state*, and define a function  $\alpha(s)$  mapping a symbolic state  $s$  into an abstract state.

The symbolic execution proceeds along a path until it discovers two states  $s$  and  $s'$  that match the same abstract state  $a$ , i.e.,  $\alpha(s) = \alpha(s') = a$ . For example, the symbolic path starting at  $s_1$  and passing through  $s_3$  is stopped at  $s_5$ . Following [24], we call this process *abstract matching*. Since the range of  $\alpha$  is finite, symbolic execution with abstract matching is guaranteed to terminate. Of course, execution also aborts whenever it encounters an error state.

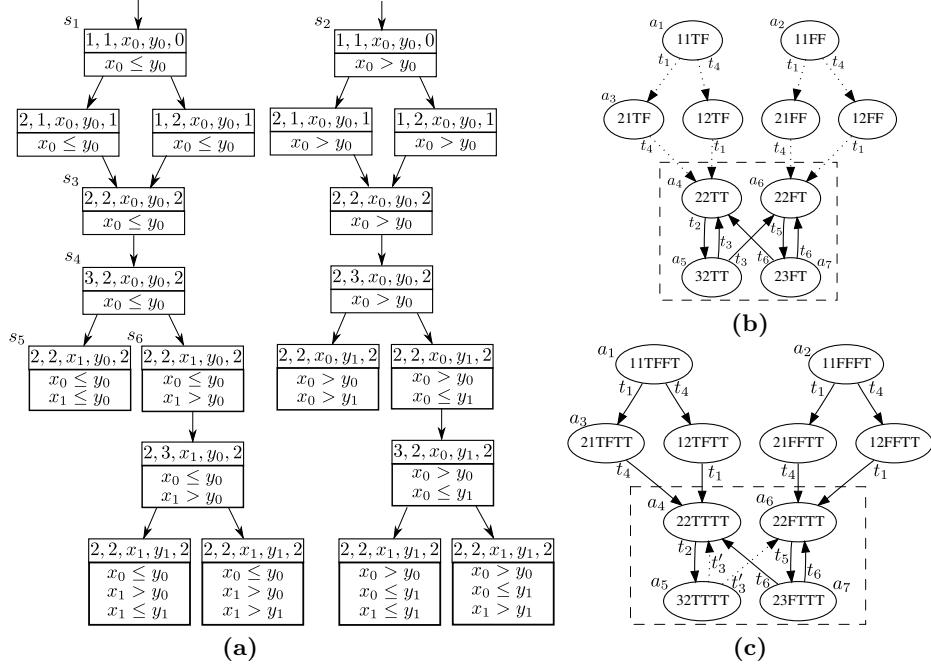
An abstract transition system  $M_a$  is obtained from the symbolic execution tree by adding a transition between two abstract states  $a$  and  $a'$  iff there is a transition between two states  $s$  and  $s'$  in the symbolic execution tree, and  $\alpha(s) = a$  and  $\alpha(s') = a'$ . The abstract transition system  $M_a$  for the execution tree in Fig. 3(a) is shown in Fig. 3(b). In the figure, each state is a valuation to  $(pc_1, pc_2, x \leq y, b = 2)$ . For example,  $\alpha(s_1) = a_1$  and  $\alpha(s_2) = a_2$ . An error state is unreachable in  $M_a$ , so it is passed to the analysis phase.

**Analysis: safe-fragment.** This check is based on a notion of an *exact* transition. A transition between two abstract states  $a$  and  $b$  is *exact* iff every concrete state corresponding to  $a$  can transition to a concrete state corresponding to  $b$ . For example, transition  $a_4 \rightarrow a_5$  in  $M_a$  is exact (denoted by a solid line) whereas transition  $a_1 \rightarrow a_3$  in  $M_a$  is inexact (denoted by a dotted line).

We say that a set of states  $Q$ , called a *fragment*, of an abstract transition system  $M_a$  is *exact* iff (a) there is no outgoing transition from  $Q$  to other states in  $M_a$ , and (b) all internal transitions within  $Q$  are exact. Intuitively, all executions from concrete states corresponding to an exact fragment  $Q$  are trapped in it. We say that an exact fragment  $Q$  is *safe* iff it does not contain error states, i.e., it approximates a part of the state-space of  $P$  that cannot reach an error.

**safe-fragment** determines whether all paths in  $M_a$  are eventually trapped in a safe exact fragment. This is reduced to checking whether the transitions inside and between all nontrivial strongly connected components of  $M_a$  are exact. If so,  $M_a$  is an over-approximation of  $P$  (see Section 4.2); therefore, none of the executions of  $P$  can reach error and thus  $P$  is safe.

The check succeeds in our example. This is easily verified by looking at Fig. 3(b), where all paths are trapped in the safe exact fragment consisting of the states  $a_4, a_5, a_6$ , and  $a_7$ . Thus, the program in Fig. 1 satisfies the mutual exclusion property.



**Fig. 3.** (a) Symbolic execution of the program in Fig. 1; (b) its corresponding abstract transition system  $M_a$ ; (c) a modified abstract transition system  $M'_a$ .

**Analysis: inductive-invariant.** This check determines whether the state-space of  $M_a$  is an inductive invariant: i.e., it is closed under applying transitions of  $P$ . If so, the state-space of  $M_a$  over-approximates that of  $P$ , and thus  $P$  is safe. This check is complimentary to safe-fragment described above (see Section 4.2). If it fails, we move to the refinement phase.

**Refinement.** In this phase, we generate new predicates to refine inexact transitions of  $M_a$ . The refinement is based on computing preimage and is similar to the commonly used weakest precondition-based refinement. Although not needed in our running example, we illustrate refinement using the inexact transition  $a_1 \xrightarrow{t_1} a_3$  of  $M_a$  in Fig. 3(b). First, we compute the preimage of  $a_3$  w.r.t. transition  $t_1$ , resulting in  $(pc_2 = 2 \wedge x \leq y \wedge b \neq 1)$ . Second, we add only the predicate  $b \neq 1$  to  $\Phi_0$  since program counter  $pc_2$  is represented explicitly, and we already have  $x \leq y$ .

In the remainder of the paper, we formalize the above notions and evaluate the efficiency of our approach.

### 3 Preliminaries

This section outlines the definitions and notation used in this paper.

**Program.** We use a guarded command language to specify programs. A *program*  $P$  is a tuple  $(V, I, T)$ , where  $V$  is a finite set of integer variables,  $I(V)$  is an initial condition, and  $T$  is a finite set of transitions. Each transition  $t \in T$  is of the form  $g_t \rightarrow e_t$ , where  $g_t$  is a Boolean expression over the variables  $V$ , and  $e_t$  is a set of concurrent assignments. Each assignment is of the form  $x := linExp$  or  $x := nondet$ , where  $x$  is a variable in  $V$ ,  $linExp$  is an expression from linear arithmetic over variables in  $V$ , and  $nondet$  is a special expression used to denote non-deterministic input.

**Transition System.** A *transition system* over a finite set of atomic propositions  $AP$  and a set of transition labels  $T$  is a tuple  $(S, R, S_0, L)$ , where  $S$  is a (possibly infinite) set of states,  $R \subseteq S \times T \times S$  is the transition relation,  $S_0 \subseteq S$  is the set of initial states, and  $L : S \rightarrow 2^{AP}$  is a labelling function, mapping each state to the set of atomic propositions that hold in it. For clarity, we write  $s \xrightarrow{t} s'$  to denote  $R(s, t, s')$ .

The *concrete semantics* of a program  $P = (V, I, T)$  is a transition system  $C(P) = (S, R, S_0, L)$  over some atomic propositions  $AP$  and the set of program transitions  $T$ , where  $S = 2^{V \rightarrow \mathbb{Z}}$ ,  $S_0 = \{s \in S \mid s \models I\}$ , and  $s \xrightarrow{t} s'$  for some  $t \in T$  iff  $s \models g_t$  and  $s' \in e_t(s)$ . By  $s \models g_t$ , we mean that the valuation of variables in  $s$  satisfies the Boolean expression  $g_t$ , and  $e_t : (V \rightarrow \mathbb{Z}) \rightarrow 2^{(V \rightarrow \mathbb{Z})}$  is a function which computes all possible states resulting from applying the assignments to some state. Finally,  $L(s) = \{\phi \in AP \mid s \models \phi\}$ .

**Preimage and Strongest Postcondition.** Let  $\phi$  be a formula over program variables. The *preimage* of  $\phi$  w.r.t. a transition  $t$ ,  $pre(\phi, t) = \exists s' \cdot (s \xrightarrow{t} s' \wedge s' \models \phi)$ , is a formula describing the set of all states which can reach a state satisfying  $\phi$  via  $t$ . The *strongest postcondition* of  $\phi$  w.r.t. a transition  $t$ ,  $sp(\phi, t) = \exists s' \cdot (s' \xrightarrow{t} s \wedge s' \models \phi)$ , is a formula describing the set of all states that are reachable via  $t$  from a state satisfying  $\phi$ .

**Predicate Abstraction.** Let  $\Phi = \{\phi_1, \dots, \phi_n\}$  be a set of predicates over program variables. The *predicate abstraction*  $\alpha_\Phi$  is a function from concrete states to Boolean formulae (abstract states) over predicates in  $\Phi$ . Given a concrete state  $s$ ,  $\alpha_\Phi(s) = \bigwedge_{\phi \in \Phi_s} \phi \wedge \bigwedge_{\phi \in \bar{\Phi}_s} \neg\phi$ , where  $\Phi_s = \{\phi \in \Phi \mid s \models \phi\}$  and  $\bar{\Phi}_s = \Phi \setminus \Phi_s$ . A *concretization function*  $\gamma_\Phi$  takes a Boolean formula over  $\Phi$  and returns the set of concrete states satisfying the formula. Given a Boolean formula  $\psi$  over  $\Phi$ ,  $\gamma_\Phi(\psi) = \{s \in S \mid s \models \psi\}$ . For a set of states  $X$ , we write  $\alpha_\Phi(X)$  to mean  $\bigvee \{\alpha_\Phi(s) \mid s \in X\}$ .

A transition  $a_1 \xrightarrow{t} a_2$ , where  $a_1$  and  $a_2$  are abstract states is a *must* transition iff  $\forall s \in \gamma_\Phi(a_1) \cdot \exists s' \in \gamma_\Phi(a_2)$  s.t.  $s \xrightarrow{t} s'$ . A transition is a *may* transition iff  $\exists s \in \gamma_\Phi(a_1) \cdot \exists s' \in \gamma_\Phi(a_2)$  s.t.  $s \xrightarrow{t} s'$ . In this paper, we call *must* transitions *exact*, and transitions that are *may* but not *must - inexact*. A transition  $a_1 \xrightarrow{t} a_2$  is exact iff  $a_1 \Rightarrow pre(a_2, t)$ .

```

1: function REFINE( $P, \psi$ )
2:    $\Phi \leftarrow$  predicates from guards in  $P$  and  $\psi$ 
3:   while true do
4:      $inductive \leftarrow true$ 
5:      $(fin, inf, A_0) \leftarrow$  SYMBOLICEXEC( $P, \Phi$ ) ▷ symbolic execution
6:     if a state in  $(fin, inf, A_0)$  satisfies  $\neg\psi$  then return false
7:     if SAFEFRAGMENT( $fin, inf$ ) then return true ▷ safe-fragment
8:      $A \leftarrow$  all states in  $(inf, fin, A_0)$ 
9:     for all  $(a_1, t, a_2) \in (fin \cup inf)$  do ▷ inductive-invariant
10:      if  $\neg(a_1 \Rightarrow pre(a_2, t))$  then
11:        add predicates in  $pre(a_2, t)$  to  $\Phi$  ▷ refinement
12:        if  $\neg(sp(a_1, t) \Rightarrow \bigvee A)$  then  $inductive \leftarrow false$ 
13:      if  $inductive$  then return true

```

**Fig. 4.** Refinement loop (main function).

## 4 Abstract Analysis of Symbolic Executions

In this section, we describe our algorithm in detail and discuss its properties.

### 4.1 Algorithm

Our abstraction-refinement based verification algorithm is implemented by the function REFINE (Fig. 4) which does symbolic execution followed by **safe-fragment** and **inductive-invariant** checks and refinement (see Fig. 2). It uses two helper functions: SYMBOLICEXEC (Fig. 5), to do symbolic execution with abstract matching and to compute the explored abstract transition system, and SAFEFRAGMENT (Fig. 6), to prove safety of the abstract transition system.

REFINE initializes the set  $\Phi$  with all the predicates in the program’s guards and in the safety property  $\psi$  (line 2), and enters the execute-analyse-refine loop (lines 3–13). It uses SYMBOLICEXEC (line 5) to compute the abstract transition system. It terminates with *false* if an error state is found (line 6); otherwise, it performs the **safe-fragment** check (line 7) followed, if needed, by the **inductive-invariant** check (lines 9–13). The Boolean variable *inductive* holds the result of **inductive-invariant**. If it is *false* after **inductive-invariant** (line 13), then REFINE repeats symbolic execution with new predicates added to  $\Phi$ ; otherwise, it returns *true*.

SYMBOLICEXEC performs depth-first symbolic execution with abstract matching. It uses the stack *symbStack* of sets of symbolic states to keep track of the current path. A *symbolic state*  $s$  over a set of variables  $V$  is a tuple  $(f, PC)$ , where  $f$  is a function mapping each program variable to an integer or symbolic constant, and a *path condition*  $PC$  is a set of constraints over symbolic and integer constants. A concrete state  $c$  is represented by a symbolic state  $s = (f, PC)$  iff  $c \models \exists V_s \cdot \bigwedge_{y \in V} y = f(y) \wedge \bigwedge_{p \in PC} p$ , where  $V_s = \{z \mid \exists y \cdot f(y) = z \wedge z \text{ is a symbolic constant}\}$ . For example, the state  $(\{x \mapsto x_0, y \mapsto y_0\}, \{y_0 > 0, x_0 > y_0\})$  denotes the set of concrete states where  $y$  is strictly greater than 0 and  $x$  is strictly greater than  $y$ .

Let  $x$  be a variable in  $V$  and  $s = (f, PC)$  a symbolic state. The symbolic execution is done by the function EXEC (line 14 of Fig. 5) using the following rules: if  $e_t$  is  $x := nondet$  then the result is the state  $s' = (f[x \rightarrow z], PC)$ ,

```

1: function SYMBOLICEXEC( $P, \Phi$ )
2:    $symbStack \leftarrow$  empty stack            $\triangleright$  Each item on the stack is a set of states
3:   push SPLITSTATE( $s_0, \Phi$ ) on  $symbStack$ 
4:    $A_0 \leftarrow \{\alpha_\Phi(s) \mid s \in \text{SPLITSTATE}(s_0, \Phi)\}$ 
5:    $(fin, inf, trans) \leftarrow (\emptyset, \emptyset, \emptyset)$ 
6:   while  $symbStack$  is not empty do
7:      $S \leftarrow$  top of  $symbStack$ 
8:     choose  $s \in S$  s.t. for some  $t$ ,  $s \models g_t$ 
9:     if no such transition exists or transitions exhausted then
10:     $fin \leftarrow fin \cup \text{ALLPATHS}(S, trans)$ 
11:     $trans \leftarrow$  tail of  $trans$ 
12:    pop  $symbStack$ 
13:    continue
14:     $result \leftarrow \text{EXEC}(s, e_t)$ 
15:     $S' \leftarrow \text{SPLITSTATE}(result, \Phi)$ 
16:    for all  $s' \in S' \mid \alpha_\Phi(s') = \alpha_\Phi(s)$  or  $\exists t \cdot (\alpha_\Phi(s'), t') \in trans$  do
17:       $fin \leftarrow fin \cup \text{STEM}((\alpha_\Phi(s), t, \alpha_\Phi(s')), trans)$ 
18:       $inf \leftarrow inf \cup \text{LOOP}((\alpha_\Phi(s), t, \alpha_\Phi(s')), trans)$ 
19:       $S' \leftarrow S' \setminus \{s'\}$ 
20:    if  $S' \neq \emptyset$  then
21:      push  $S'$  on  $symbStack$ 
22:       $trans \leftarrow$  prepend  $(\alpha_\Phi(s), t)$  to  $trans$ 
23:   return  $(fin, inf, A_0)$ 

```

**Fig. 5.** Symbolic execution with abstract matching.

<pre> 1: <b>function</b> SAFEFRAGMENT(<math>fin, inf</math>) 2:   <math>worklist \leftarrow inf</math> 3:   <b>while</b> <math>worklist \neq \emptyset</math> <b>do</b> 4:     <math>(a, t, b) \leftarrow</math> remove element from <math>worklist</math> 5:     <b>if</b> <math>\neg(a \Rightarrow pre(b, t))</math> <b>then return</b> <i>false</i> 6:     <math>T \leftarrow \{(a', t', b') \in fin \mid a' \in \{a, b\}\}</math> 7:     <math>fin \leftarrow fin \setminus T</math> 8:     <math>worklist \leftarrow worklist \cup T</math> 9:   <b>return</b> <i>true</i> </pre>	<pre> <b>function</b> SPLITSTATE(<math>s</math>) <math>S \leftarrow \emptyset, X \leftarrow \alpha_\Phi(s)</math> <b>for all</b> minterms <math>x \in X</math> <b>do</b>   <math>S \leftarrow S \cup (f, PC \cup x[f(v_1)/v_1, \dots, f(v_n)/v_n])</math> <b>return</b> <math>S</math> </pre>
--	---

**Fig. 6.** Algorithms for safe-fragment check and splitting symbolic states.

where  $z$  is fresh symbolic constant (i.e., is not used in any symbolic state so far) and  $f[x \rightarrow z](a)$  is  $z$  if  $x = a$  and  $f(a)$  otherwise; if  $e_t$  is  $x := u$  for some expression  $u$ , then the result is the state  $s' = (f[x \rightarrow z], PC')$ , where  $z$  is fresh and  $PC' = PC \cup (z = u[f(v_1)/v_1, \dots, f(v_n)/v_n])$ ; if  $e_t$  is a concurrent assignment, the result is obtained by the obvious generalization of the base rules.

Recall that we require that a symbolic state satisfies or refutes each predicate in  $\Phi$ . We enforce this using SPLITSTATE that takes a symbolic state as input and returns a set of states that satisfy our constraint. The naïve implementation is to recursively split a state  $s$  by taking a predicate from  $p \in \Phi$  that is neither satisfied nor refuted by  $s$  and creating two new states by adding  $p$  and  $\neg p$  to the path constraint of  $s$ , respectively. This is highly inefficient.

Instead, we reduce the problem to predicate abstraction as shown in Fig. 6. Basically, we compute a predicate abstraction of a symbolic state  $s$  and then split  $s$  using all the minterms in the result<sup>1</sup>. For example, consider the symbolic state  $s = (\{x \mapsto x_0, y \mapsto y_0\}, \{y_0 = 0\})$  over  $V = \{x, y\}$ , and let  $\Phi = \{x >$

<sup>1</sup> A minterm is a conjunction of literals containing all predicates in  $\Phi$ .

$0, y > 0\}.$  Predicate abstraction of  $s$  over  $\Phi$  has two minterms:  $\{x > 0 \wedge y \leq 0, x < 0 \wedge y \leq 0\}.$  This leads to two new symbolic states  $\{(f', PC'), (f'', PC'')\},$  where  $f'' = f' = \{x \mapsto x_0, y \mapsto y_0\},$   $PC' = \{y_0 = 0, x_0 > 0, 0 \leq 0\},$  and  $PC'' = \{y_0 = 0, x_0 \leq 0, 0 \leq 0\}.$  Of course, tautologies like  $0 \leq 0$  are discarded when the expression is simplified. This has the same worst-case complexity as the naïve approach, but allows us to use recent advances in predicate abstraction (such an AllSAT SMT solver).

In SYMBOLICEXEC, the variable  $trans$  keeps an abstraction of the path from the initial state to states at the top of  $symbStack$  as a list of tuples  $(a, t)$ , where  $a$  is an abstract state and  $t$  a transition. Whenever SYMBOLICEXEC reaches a state whose abstraction has been seen before on the current path (i.e., either it is the same as a predecessor or it appears in  $trans$  – line 16), it stops current exploration and backtracks. At this point,  $trans$  is a lasso-shaped abstract path. Functions STEM and LOOP are used to extract the transitions that occur on the stem of the path, stored in set  $fin$  and the loop of the path, stored in  $inf.$  For example, consider the symbolic execution tree shown in Fig. 3(a). When SYMBOLICEXEC takes the transition  $s_4 \xrightarrow{t_3} s_5$  (corresponding to the abstract transition  $a_5 \xrightarrow{t_3} a_4$  in the abstract model in Fig. 3(b)) it discovers a loop. Then,  $trans = [(a_4, t_2), (a_3, t_4), (a_1, t_1)], STEM(trans, (a_5, t_3, a_4)) = \{(a_1, t_1, a_3), (a_3, t_4, a_4)\},$  and  $LOOP(trans, (a_5, t_3, a_4)) = \{(a_4, t_2, a_5), (a_5, t_3, a_4)\}.$  Note that transitions in the list  $trans$  are stored in reverse order from which they appear on the abstract path.

When SYMBOLICEXEC reaches a set of symbolic states where no transition can be taken (line 9), the transitions leading to that set of states are added to  $fin$  using the ALLPATHS function. For example, assume  $trans = [(a_2, t_2), (a_1, t_1)]$  and  $S = \{a_3, a_4\}.$  Then,  $ALLPATHS(S, trans) = \{(a_1, t_1, a_2), (a_2, t_2, a_3), (a_2, t_2, a_4)\}.$

SAFEMFRAGMENT works by locating the fragment of the abstract model that is reached by all execution paths. This is done by finding all transitions in or reachable from  $inf.$  If all of these transitions are exact, then we conclude safety. Otherwise, we proceed to inductive-invariant.

In inductive-invariant (lines 9–13 of REFINE), for every state which is the source of an inexact transition  $t$ , we check if its strongest postcondition w.r.t.  $t$  is a subset of the set of abstract states explored (line 12). If so, the explored abstract states over-approximate the set of reachable concrete states, and thus we can conclude safety. Otherwise, we go back to the symbolic execution stage, but now with new predicates added from preimages of destination states of inexact transitions (line 11 of REFINE).

## 4.2 Soundness and Monotonicity

Our algorithm only reports real errors. This is ensured by restricting symbolic execution to explore only symbolic states with satisfiable path constraints. Theorem 1 states that the algorithm is also sound for safety properties<sup>2</sup>. Of course, since property checking is undecidable, the algorithm is incomplete.

---

<sup>2</sup> Proofs of all theorems can be found in [1].

In the rest of this section, we represent the abstract state-space explored by SYMBOLICEXEC by a transition system  $M_a = (S^a, R^a, S_0^a, L^a)$ , where  $AP = \Phi$ ,  $S^a$  is the set of all states appearing in  $(fin, inf, A_0)$ ,  $R^a$  is the set of all transitions appearing in  $fin \cup inf$ ,  $S_0^a = A_0$ , and for  $x \in S^a$ ,  $L^a(x) = \{\phi \in \Phi \mid x \models \phi\}$ .

**Theorem 1 (Soundness).** *Let  $\psi$  be a safety property,  $P$  be a program satisfying  $\psi$ , and  $M_c = (S, R, S_0, L)$  be a transition system of  $P$ . W.l.o.g., assume that every state in  $S$  is reachable from  $S_0$ . Let  $M_a = (S^a, R^a, S_0^a, L^a)$  be the abstract transition system constructed by SYMBOLICEXEC in the last iteration of REFINE. Then, (i) if REFINE terminates after safe-fragment (line 7), then  $M_a$  simulates  $M_c$ ; (ii) if REFINE terminates after inductive-invariant (line 13), then  $S^a$  over-approximates  $S$  (i.e.,  $\forall s \in S \cdot \alpha_\Phi(s) \in S^a$ ).*

As in SYNERGY (see Sec. 4 in [16]), when our algorithm terminates with **safe-fragment**, the current abstraction simulates, but is not necessarily bisimilar to, the concrete program. Moreover, if it terminates with **inductive-invariant** then the abstraction may not even simulate the concrete program.

In contrast with other under-approximating approaches, e.g., [24, 4], our algorithm explores more states in each successive iteration than in a previous one. That is, the exploration is monotonically increasing. This ensures steady progress towards an error state (if one exists). Intuitively, we get this by keeping an abstract visited table per each path, as opposed to a unique global table as in [24].

**Theorem 2 (Monotonicity).** *Let  $\Phi$  and  $\Phi'$  be two sets of predicates s.t.  $\Phi \subseteq \Phi'$ . Let  $P$  be a program, and  $C$  and  $C'$  be the concrete states of  $P$  explored by SYMBOLICEXEC under  $\Phi$  and  $\Phi'$ , respectively. Then,  $C \subseteq C'$ .*

In contrast, our approach is not monotonic for proving safety: adding new predicates may cause an exact transition used by **safe-region** check to become inexact [13, 24, 4]. In the future, we hope to solve this problem by using an abstract domain of tri-vectors.

As discussed in Section 2, the two checks, **safe-fragment** and **inductive-invariant**, are incomparable. We prove this below.

**Theorem 3.** *There is an abstract model  $M_a$  constructed by SYMBOLICEXEC that passes exactly one of **safe-fragment** and **inductive-invariant** checks.*

*Proof.* First, we give an example where **safe-fragment** holds but **inductive-invariant** fails. Consider  $M_a$  in Fig. 3(b). Recall that it passes **safe-fragment** check. It fails **inductive-invariant** since it is not closed under strongest postcondition:  $sp(a_1, t_1) = (pc_1 = 2 \wedge pc_2 = 1 \wedge x \leq y \wedge b \neq 2) \vee (pc_1 = 2 \wedge pc_2 = 1 \wedge x \leq y \wedge b = 2)$ ; the second disjunct is not covered by an explored abstract state.

Second, we give an example where **inductive-invariant** holds but **safe-fragment** fails. Consider  $M'_a$  shown in Fig. 3(c). It is obtained from symbolically executing a program obtained by replacing transition  $t_3$  by  $t_3 : pc_1 = 3 \rightarrow pc_1 := 2$ ,  $x := x + 1$  in the protocol in Fig. 1, and assuming that  $\Phi$  includes predicates  $b = 0$ ,  $b = 1$ , and predicates from the guards. All transitions of  $M'_a$ , with the exceptions of the two transitions from  $a_5$ , are exact. **safe-fragment** fails on  $M'_a$ . **inductive-invariant** does not: the only interesting case is that  $sp(a_5, t'_3) = (pc_1 = 2 \wedge pc_2 = 2 \wedge b = 2)$  is covered by explored abstract states.  $\square$

Program	$\psi$	Iter		Prvr.		Qurs.		Preds.		Time(s)		Con/Abs States		Check
		ASE	UR	ASE	UR	ASE	UR	ASE	UR	ASE	UR	ASE	UR	
bakery <sub>2</sub>	t	3	4	141	367	8	10	0.347	0.452	52/33	49/36	-	II	
RAX	t	1	-	6	-	2	-	0.261	-	81/44	-	-	SF	
elev <sub>4</sub>	t	1	4	418	5789	13	19	1.013	8.146	468/378	468/456	-	SF	
elev <sub>5</sub>	t	1	5	1169	26252	15	23	3.459	44	1256/910	1253/1204	-	SF	
elev <sub>6</sub>	t	1	6	3156	105830	17	27	12.275	220.633	3248/2126	3224/3060	-	SF	
elev <sub>7</sub>	t	1	-	7116	-	19	-	40.867	-	8160/4862	-	-	SF	
elev <sub>8</sub>	t	1	-	15036	-	21	-	185.717	-	15200/9422	-	-	SF	
ticket <sub>2</sub>	t	4	4	135	120	8	8	0.609	0.404	22/9	12 / 9	-	SF	
ticket <sub>3</sub>	t	5	5	672	661	14	14	1.413	0.923	182/31	41/31	-	SF	
ticket <sub>4</sub>	t	6	6	4088	4061	23	23	33.51	5.143	5011/129	170/129	-	SF	
mesi	t	16	16	6893	12172	47	47	36.61	49.627	18/18	18/18	-	SF	
berkley	t	11	11	3113	4623	38	38	15.729	17.605	13/12	13/12	-	SF	
b.bakery <sub>2-e</sub>	f	1	2	0	74	2	5	0.178	1.188	80/80	193/193	-		
ticket <sub>2-e</sub>	f	1	2	0	11	2	5	0.073	0.155	12 / 9	26 / 17	-		
ticket <sub>3-5</sub>	t	1	3	0	145	3	14	0.058	0.341	14/12	93/81	-		
ticket <sub>3-10</sub>	t	3	8	152	1218	14	21	0.525	2.229	30/27	302/240	-		
ticket <sub>3-15</sub>	t	8	13	1225	2090	21	26	3.107	5.15	47/44	507/395	-		
ticket <sub>3-20</sub>	t	13	18	2500	3918	26	31	6.869	9.501	62/59	712/550	-		
ticket <sub>3-25</sub>	t	18	23	3925	5493	31	36	13.038	15.821	77/74	917/705	-		
ticket <sub>3-30</sub>	t	23	28	5500	7219	36	41	20.762	34.701	92/89	1112/860	-		
ticket <sub>3-35</sub>	t	28	33	7225	9093	41	46	46.379	51.579	107/104	1327/1015	-		
ticket <sub>3-40</sub>	t	33	38	9100	11118	46	51	71.462	82.974	122/119	1532/1170	-		
RAX-5	t	5	5	46	123	12	20	0.373	0.363	50/49	170/170	-		
RAX-10	t	10	10	146	483	17	35	0.988	1.528	90/89	350/350	-		
RAX-15	t	15	15	296	1063	22	50	2.031	4.341	130/129	530/530	-		
RAX-20	t	20	20	496	1878	27	65	3.675	9.934	170/169	710/710	-		
RAX-25	t	25	25	746	2913	32	80	6.442	19.578	210/209	890/890	-		
RAX-30	t	30	30	1046	4173	37	95	9.94	35.03	250/249	1070/1070	-		
RAX-35	t	35	35	1396	5658	42	110	15.155	57.315	290/289	1250/1250	-		
RAX-40	t	40	40	1796	7368	47	125	22.104	89.332	320/319	1430/1430	-		
RAX-45	t	45	45	2246	9303	52	140	30.821	133.063	370/369	1610/1610	-		

**Fig. 7.** Experimental results: ASE vs. UR [24].

We have shown that our algorithm is sound and explores the concrete state-space monotonically. We have also shown that the two safety checks, **safe-fragment** and **inductive-invariant**, are incomparable. Hence, both are useful.

## 5 Implementation and Experimental Results

We have implemented our algorithm in OCaml on top of the implementation of Pasareanu et al. [24]. We used GiNaC [5] for symbolic execution, MATHSAT4 [7] for computing predicate abstraction, SIMPLIFY [11] for checking exactness of transitions and computing inductive invariants, and Bradley’s implementation of Cooper’s method for quantifier elimination<sup>3</sup>. In all of our experiments, we added predicates only from those inexact transitions that are in the set *inf* (returned by SYMBOLICEXEC) or reachable from it.

In Fig. 7, we compare effectiveness of our *abstract analysis of symbolic executions* approach (referred to as ASE) with that of the *under-approximation refinement* algorithm of [24] (referred to as UR). We indicate whether the safety property of interest ( $\psi$ ) is true (t) or false (f) and report the number of iterations (Iter.), the number of theorem prover queries (Prvr. Qurs.), the total number

<sup>3</sup> Available at <http://theory.stanford.edu/~arbrad/sware.html>.

of predicates used (Preds.), the total amount of time needed, the number of concrete and abstract states explored in the final iteration, and the check with which ASE concluded safety (II for inductive-invariant, SF for safe-fragment, and “–” when a counterexample is returned). In cases where the experiment did not finish after 15 minutes, the table entries are “–”.

Since UR can only handle a single concrete initial state and no non-deterministic input, these are the characteristics of all programs in Fig. 7. We began by checking the mutual exclusion property of the `bakery` protocol with two processors, where our performance is a bit better than UR. On the other hand, ASE can prove that the Remote Agent Experiment (RAX), as presented in [24], is deadlock-free in a single iteration, while UR refines indefinitely. We then verified the elevator program, `elevi`, increasing the number of floors  $i$ , against the property that the elevator cannot be on two separate floors at the same time. We checked mutual exclusion of the `ticketi` protocol, increasing the number of processes  $i$ , as well as correctness cache coherence protocols `mesi` and `berkley` (these, along with their correctness properties, are taken from [10], restricting the number of initial states to one). Our results show that ASE generally outperforms UR in terms of the number of iterations and time it takes to prove safety. In the case of `ticketi` where ASE requires the same number of iterations and predicates, ASE takes more time as it explores more concrete states per iteration.

To illustrate the power of our approach at finding errors, we analysed defective versions, i.e., not satisfying mutual exclusion, of the bounded bakery (`b_bakery2-e`) and ticket (`ticket2-e`) protocols. We also checked whether a given ticket number  $X$  in the `ticketi` protocol (`ticketi-X`) and a given counter value  $X$  in the RAX example (`RAX-X`) are reachable. ASE terminates in fewer iterations than UR in the former case and in the same number of iterations but significantly fewer predicates in the latter.

In Fig. 8, we report on the results of ASE for checking properties of programs with unspecified initial states and/or non-deterministic input. Specifically, we verified mutual exclusion of `ticket`, where the initial ticket number is set non-deterministically, and `bakery` and `peterson` protocols, where each process stays in the critical section for a non-deterministic amount of time. We also verified correctness of cache coherence protocols, `mesi` and `synapse`, with undefined initial states.

In summary, ASE can analyse a wide range of programs that manipulate arbitrary integers and use non-deterministic input. And it can do so in less time, and considerably fewer iterations or with significantly fewer predicates than UR.

## 6 Related Work

The work by Pasareanu et al. [24] is the closest to ours. However, there are several key differences. First, our approach explores the state-space monotonically. Second, we use symbolic execution to deal with programs with arbitrary initial states and non-deterministic input. Third, we use over-approximation much

Program	$\psi$	Iter.	Prvr. Qurs.	Preds.	Symb/Abs States	Time(s)	Check
ticket <sub>2</sub>	t	4	523	12	5516/62	281.134	II
peterson <sub>2</sub>	t	1	24	4	700/38	424	II
bakery <sub>2</sub>	t	3	301	11	807/50	73.965	II
mesi	t	2	260	13	112/14	3.56	II
synapse	t	2	62	7	34/9	0.88	II
ticket <sub>2-e</sub>	f	1	0	2	12/10	0.104	-
ticket <sub>3-e</sub>	f	1	0	3	10/10	0.112	-

**Fig. 8.** Experimental results: programs with unspecified initial states and non-deterministic input.

more aggressively leading to a much faster convergence with fewer predicates. Comparison with other work is given below.

Over-approximation based techniques (e.g., [3, 17, 8]) build an abstraction that has more behaviours than the concrete system and prune infeasible computations via refinement. In contrast, our refinement is based on extending the frontier of feasible program behaviours. Most of such techniques, with the exception of [8], deal with sequential programs only.

Under-approximation based techniques [24, 4, 21, 25] build an abstraction that has fewer program behaviours than the concrete system. Our approach includes both reachable *must* and *may* transitions making the abstract models more precise than those that have just *must* transitions (e.g., [25]) and *must* and *reverse must* transitions (e.g., [4]<sup>4</sup>). The algorithm in [21] builds a finite bisimulation quotient of the program under analysis, but unlike the global refinement employed by us and [24], uses a local refinement instead. We leave a comparison of the efficiency of local and global refinements for future work.

Most recent automated software verification techniques that combine dynamic analysis for detecting bugs and static analysis for proving correctness (e.g., [27, 16, 14, 20]) concentrate on analysis of sequential programs, and unlike our approach which bounds program executions, assume terminating program executions. For example, [27] uses tests cases to explore an under-approximating abstract state-space with the hope of exploring all reachable abstract states but has no notion of refinement and thus the analysis may return false positives. Like our work, [2] uses abstraction to bound symbolic execution of programs. While this approach can handle programs with recursive data structures and arrays, its goal is debugging rather than verification, and it does not involve refinement.

[20] improves error detection capabilities of the CEGAR framework [9] by using program execution to drive abstraction-refinement. However, it does so by refining an over-approximation and is restricted to sequential programs.

Directed automated random testing (DART) [15] and its successors, [26, 12], run the program with random input, using path constraints to discover input that would exercise alternative program paths. The SYNERGY algorithm [16] combines DART-like testing with over-approximating abstractions, using results of tests to refine the abstract model and using the abstract model to drive test case generation. The end result is either a test case that reaches an error state, or an abstract model that simulates the program. Whereas DART-like approaches

---

<sup>4</sup> see [1] for a detailed comparison with [4].

attempt to cover all program paths, our approach and [24, 27] attempt to cover all reachable abstract states. [14] presents a compositional algorithm that combines DART and over-approximating techniques. DART-like testing is used to create under-approximating (*must*) summaries of functions, and techniques based on [3] are used to create over-approximating (*may*) summaries. The authors show that alternating *must* and *may* summaries yields better results than *must* only or *may* only summaries. However, these techniques are restricted to sequential programs.

## 7 Conclusion and Future Work

We presented a novel verification algorithm that combines symbolic execution and predicate abstraction in an abstraction-refinement cycle. Our approach applies to concurrent programs with infinite data domain and non-deterministic input. Given a program and a safety property, our algorithm executes the program symbolically, while building an under-approximating abstract model. If an error is reached by symbolic execution, we terminate and report it. Otherwise, we check whether the state-space of the abstract model over-approximates all concretely reachable states. If the analysis fails, we refine with new predicates and repeat the process. Not only do we handle a much wider range of programs than related approaches, we also improve on the number of iterations and the number of predicates used, whether the property of interest is true or false.

Our current implementation is a proof of concept – more work is needed to turn it into robust verification tool that is applicable to a real programming language (such as C) with complex features (e.g., structured and recursive data types, pointers, recursion, etc.). It is also interesting to see whether the approach extends to termination (and non-termination) properties. A promising direction is to use the under-approximation to derive either a ranking function or a counterexample to termination. We leave exploring these for future work.

**Acknowledgements.** We would like to thank Corina Pasareanu and Radek Pelanek for giving us access to their code, the anonymous referees for their helpful comments, and the formal methods group at the University of Toronto for the fruitful discussions.

## References

1. Albargouthi, A.: Abstract Analysis via Symbolic Executions. Master's thesis, Univ. of Toronto, Dept. of Comp. Sci. (February 2010), (in preparation)
2. Anand, S., Pasareanu, C.S., Visser, W.: Symbolic Execution with Abstraction. STTT 11(1), 53–67 (2009)
3. Ball, T., Rajamani, S.: The SLAM Toolkit. In: Proc. of CAV'01. LNCS, vol. 2102, pp. 260–264 (July 2001)
4. Ball, T., Kupferman, O., Yorsh, G.: Abstraction for Falsification. In: Proc. of CAV'05. pp. 67–81 (2005)
5. Bauer, C., Frink, A., Kreckel, R.: Introduction to the GiNaC Framework for Symbolic Computation with the C++ Programming Language. J. Symbolic Computation 33, 1–12 (2002)

6. Beyer, D., Henzinger, T., Théoduloz, G.: Program Analysis with Dynamic Precision Adjustment. In: Proc. of ASE'08. pp. 29–38 (2008)
7. Bruttmesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The Math-SAT 4 SMT Solver. In: Proc. of CAV'08. pp. 299–303 (2008)
8. Chaki, S., Clarke, E., Groce, A., Jha, S., Veith, H.: Modular Verification of Software Components in C. IEEE Tran. on Soft. Eng. 30(6), 388–402 (June 2004)
9. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement. In: Proc. of CAV'00. LNCS, vol. 1855, pp. 154–169 (2000)
10. Delzanno, G.: Automatic Verification of Parameterized Cache Coherence Protocols. In: Proc. of CAV'00. pp. 53–68 (2000)
11. Detlefs, D., Nelson, G., Saxe, J.: Simplify: a Theorem Prover for Program Checking. J. of the ACM 52(3), 365–473 (2005)
12. Godefroid, P.: Compositional Dynamic Test Generation. In: Proc. of POPL'07. pp. 47–54 (2007)
13. Godefroid, P., Huth, M., Jagadeesan, R.: Abstraction-based Model Checking using Modal Transition Systems. In: Proc. of CONCUR'01. LNCS, vol. 2154 (2001)
14. Godefroid, P., Nori, A., Rajamani, S., Tetali, S.: Compositional May-Must Program Analysis: Unleashing the Power of Alternation. In: Proc. of POPL'10 (2010)
15. Godefroid, P., Karlund, N., Sen, K.: DART: Directed Automated Random Testing. In: Proc. of PLDI'05. pp. 213–223 (2005)
16. Gulavani, B., Henzinger, T., Kannan, Y., Nori, A., Rajamani, S.: SYNERGY: a New Algorithm for Property Checking. In: Proc. of FSE'06. pp. 117–127 (2006)
17. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy Abstraction. In: Proc. of POPL'02. pp. 58–70 (January 2002)
18. Holzmann, G., Joshi, R.: Model-Driven Software Verification. In: Proc. of SPIN'04. LNCS, vol. 2989, pp. 76–91 (2004)
19. Holzmann, G.: The Model Checker SPIN. IEEE Tran. on Soft. Eng. 23(5) (1997)
20. Kroening, D., Groce, A., Clarke, E.: Counterexample Guided Abstraction Refinement via Program Execution. In: Proc. of ICFEM'04. pp. 224–238 (2004)
21. Lee, D., Yannakakis, M.: Online Minimization of Transition Systems. In: Proc. of STOC'92. pp. 264–274 (1992)
22. Musuvathi, M., Qadeer, S.: CHESS: Systematic Stress Testing of Concurrent Software. In: Proc. of LOPSTR'06. LNCS, vol. 4407, pp. 15–16 (July 2006)
23. Nori, A., Rajamani, S., Tetali, S., Thakur, A.: The YOGI Project: Software Property Checking via Static Analysis and Testing. In: Proc. of TACAS'09. LNCS, vol. 5505, pp. 178–181 (2009)
24. Pasareanu, C., Pelanek, R., Visser, W.: Concrete Model Checking with Abstract Matching and Refinement. In: Proc. of CAV'05. LNCS, vol. 3576, pp. 52–66 (2005)
25. Pasareanu, C., Dwyer, M., Visser, W.: Finding Feasible Counter-examples when Model Checking Abstracted Java Programs. In: Proc. of TACAS'01. LNCS, vol. 2031, pp. 284–298 (April 2001)
26. Sen, K., Marinov, D., Agha, G.: CUTE: A Concolic Unit Testing Engine for C. In: Proc. of ESEC/FSE'05. pp. 263–272 (2005)
27. Yorsh, G., Ball, T., Sagiv, M.: Testing, Abstraction, Theorem Proving: Better Together! In: Proc. of ISSTA'06. pp. 145–156 (2006)