Abstract Analysis via Symbolic Executions

by

Aws Albarghouthi

A thesis submitted in conformity with the requirements
for the degree of Master's of Science
Graduate Department of Computer Science
University of Toronto

Abstract

Abstract Analysis via Symbolic Executions

Aws Albarghouthi

Master's of Science

Graduate Department of Computer Science

University of Toronto

2010

Multicore technology has moved concurrent programming to the forefront of computer science. In this thesis, 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. We compare our approach with recent related work from literature and show that our approach is generally superior both at verification and falsification.

To my Grandfather,

Acknowledgements

I would like to thank Ou Wei for helping me come up with the initial idea and kickstart the project. I would like to thank Arie Gurfinkel whose knowledge, insights, and patience were invaluable in every stage of this project. I would also like to thank my supervisor Marsha Chechik for her support, encouragement, and unparalleled excitement. Thanks to the many friends who made my master's experience, simply, amazing.

Finally, I would like to thank my parents for their infinite love and support.

Contents

List of Figures

Chapter 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. 1.1, where integer variables x and y are set non-deterministically (see Chapter 2 for more detail).

Approaches to reason about concurrent systems can be split into four categories. (1) "Classical" model-checking techniques, e.g., [17], 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, 15, 6] 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 [23, 22, 2, 20] explore an *under-approximation* of feasible program be-

haviours. 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 [22] and the Yogi project [21]. These techniques are effective both for verification and for falsification of safety properties but, with the exception of [22], have been limited to sequential programs [27, 14, 18, 13, 11]. Our work fits into this category.

In this thesis, we propose a novel approach for automatically checking safety properties of *reactive* concurrent programs 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* [16]. 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 [23], concrete model checking and abstract matching [22], and weak reachability [2]. 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 concrete program. If so, we conclude safety; otherwise, we refine the abstraction, adding predicates not to remove spurious counterexamples (as in the CEGAR framework [7]) 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

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

Figure 1.1: A simple two-process mutual exclusion protocol with inputs x and y.

technique for proving safety using under-approximating models; *(iii)* an implementation based on [22] and an empirical evaluation comparing the two approaches.

The rest of this thesis is organized as follows. In Chapter 2, we give a general overview of the approach, illustrating it on the example in Fig. 1.1. We define the notation and provide background for the remainder of the thesis in Chapter 3. Chapter 4 presents our approach in more detail, and Chapter 5 describes our implementation and experimental results. Chapter 6 compares our approach with related work. We conclude in Chapter 7 with the summary of our contributions and suggestions for future work.

Chapter 2

Overview

In this chapter, we illustrate our approach on a simple two-process mutex protocol shown in Fig. 1.1. 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), 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. 2.1. To determine whether a safety property ψ holds in a program P, we compute an abstract transition system, M_a, of P w.r.t. some initial set of predicates Φ_0 using symbolic execution with abstract matching. The state-space of M_a is an *under-approximation* of reachable states of P. If an error is found during this 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

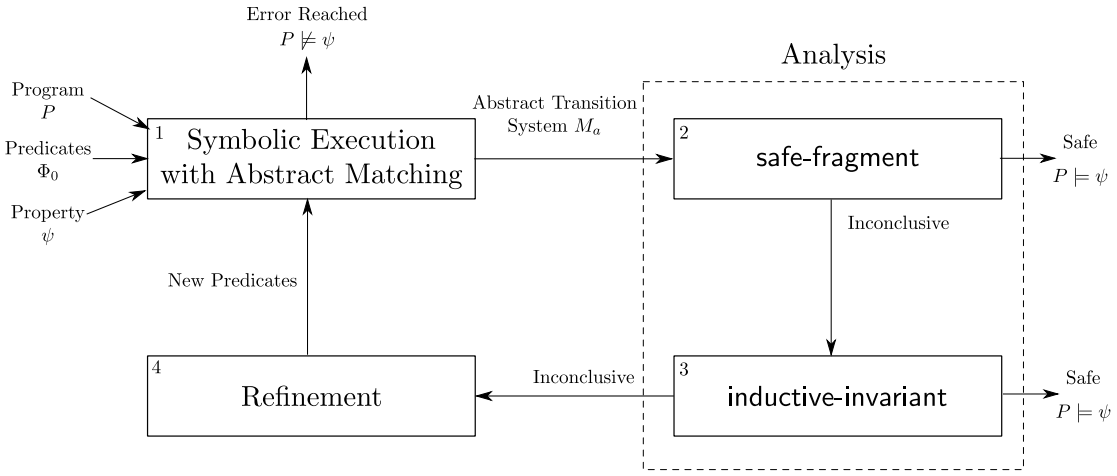

Figure 2.1: Abstract analysis of symbolic executions.

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 [7] 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 chapter, we discuss each step of our approach in turn.

Symbolic Execution with Abstract Matching.

Fig. 2.2(a) shows a symbolic execution tree of the program in Fig. 1.1. 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. 2.2(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 Φ_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. 1.1, $s_0 = (pc_1 = 1, pc_2 = 1, x = x_0, y = y_0, b = 0)$, 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 ensures that each symbolic state corresponds to (or matches with) a unique valuation of all of the predicates in Φ_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 [22], we call this process *abstract matching*. Since the domain of α 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. 2.2(a) is shown in Fig. 2.2(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 Chapter 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. 2.2(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. 1.1 satisfies the mutual exclusion property.

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 Chapter 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. 2.2(b). First, we compute the preimage of a_3 w.r.t. transition t_1, resulting in $(pc_2 = 2 \wedge x \leq y \wedge b \neq 1)$.

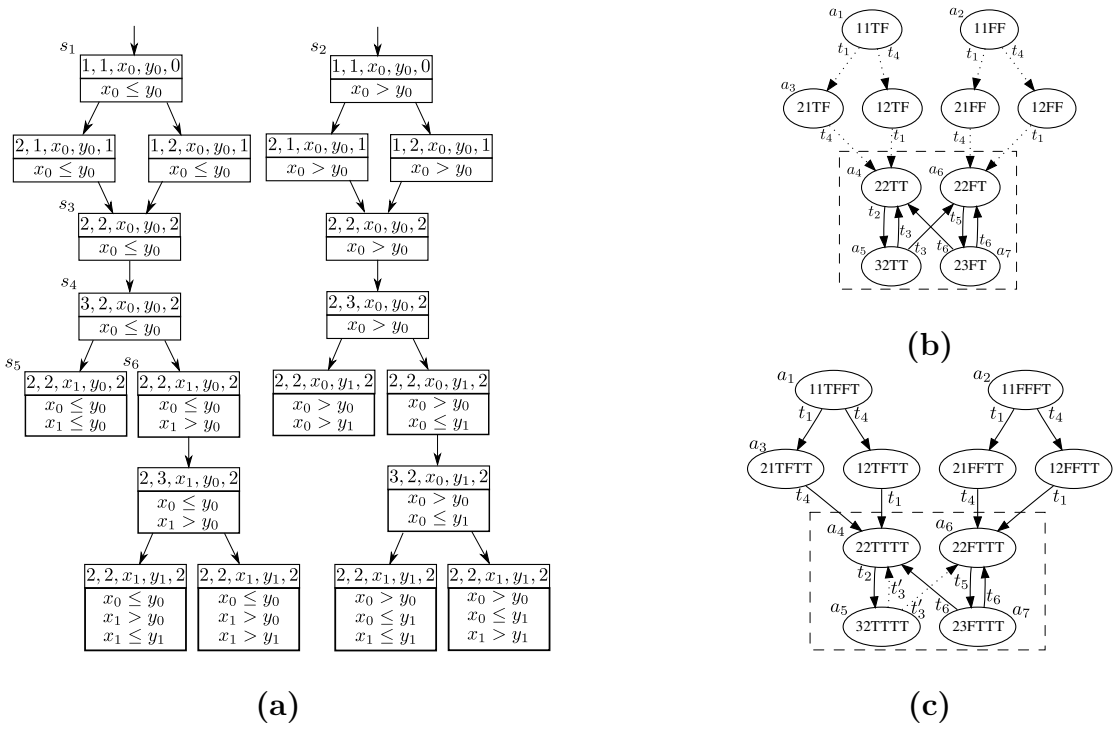

Figure 2.2: (a) Symbolic execution of the program in Fig. 1.1; (b) its corresponding abstract transition system M_a; (c) a modified abstract transition system M'_a.

Second, we add only the predicate $b \neq 1$ to Φ_0 since program counter pc_2 is represented explicitly, and we already have $x \leq y$.

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

Chapter 3

Preliminaries

This chapter outlines the definitions and notation used in this thesis.

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 $g_t \longrightarrow e_t$, 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.

Preimage and Strongest Postcondition

Let ϕ be a formula over program variables. The *preimage* of ϕ 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 ϕ via t. The *strongest postcondition* of ϕ 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 ϕ.

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 $L : S \to 2^{AP}$ 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 $R(s, t, s')$. A *maximal path* is either an infinite sequence of states $s_1, s_2, \cdots$, where $s_1 \in S_0$ and for every $i \geq 1$, $R(s_i, t, s_{i+1})$ holds for some $t \in T$; or a finite sequence of states $s_1, \cdots, s_n$ where $s_1 \in S_0$, for every $1 \leq i < n$, $R(s_i, t, s_{i+1})$ holds for some $t \in T$, and there do not exist $s' \in S$ and $t \in T$ such that $R(s_n, t, s')$.

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 $S = 2^{V \to \mathbb{Z}}$, $S_0 = \{s \in S \mid s \models I\}$, 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 $e_t : (V \to \mathbb{Z}) \to 2^{(V \to \mathbb{Z})}$ is a function which computes all possible states resulting from applying the assignments to some state. Finally, $L(s) = \{\phi \in AP \mid s \models \phi\}$.

Predicate Abstraction

Let $\Phi = \{\phi_1, \cdots, \phi_n\}$ be a set of predicates over program variables. The *predicate abstraction* α_Φ is a function from concrete states to Boolean formulae (abstract states) over predicates in Φ. Given a concrete state s, $\alpha_\Phi(s) = \bigwedge_{\phi \in \Phi_s} \phi \wedge \bigwedge_{\phi \in \overline{\Phi}_s} \neg\phi$, where $\Phi_s = \{\phi \in \Phi \mid s \models \phi\}$ and $\overline{\Phi}_s = \Phi \setminus \Phi_s$. A *concretization function* γ_Φ takes a Boolean formula over Φ and returns the set of concrete states satisfying the formula. Given a Boolean formula ψ over Φ, $\gamma_\Phi(\psi) = \{s \in S \mid s \models \psi\}$. 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 thesis, 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 $a_1 \Rightarrow pre(a_2, t)$.

Simulation Relation

Given two transition systems $M = (S, R, S_0, L)$ over AP and $M' = (S', R', S'_0, L')$ over $AP' \subseteq AP$, we call $\rho \subseteq S \times S'$ a *simulation relation* between M and M' iff for all $s \in S$ and $s' \in S'$, if $\rho(s, s')0$ then: (1) $L(s) \cap AP' = L(s')$, and (2) $\forall r \in S$ s.t. $R(s, t, r)$ holds for some label t, $\exists r' \in S'$ s.t. $R(s', t', r')$ holds for some label t', and r, r' are related by ρ. M' *simulates* M iff exists a simulation relation ρ s.t. $\forall s \in S_0 \cdot \exists s' \in S'_0$ where $\rho(s, s')$ holds.

Φ-maximal path

Let C be a concrete path $c_1, \cdots, c_n$ such that for all $1 \leq i \leq n$, $c_i \xrightarrow{t} c_{i+1}$ for some transition t. C is Φ-maximal, where Φ is a set of predicates, iff $\exists i \in [1 \cdots n)$. $\alpha_\Phi(c_n) = \alpha_\Phi(c_i)$ and $\forall k, j \in [1 \cdots n]$. $(k > j \wedge \alpha_\Phi(c_j) = \alpha_\Phi(c_k)) \Rightarrow k = n$ and $j = i$

Chapter 4

Abstract Analysis of Symbolic Executions

In this chapter, we describe our algorithm in detail, discuss its properties, and comapre the precision of our under-approximating models with other under-approximating models.

4.1 Algorithm

Our abstraction-refinement based verification algorithm is implemented by the function REFINE (Fig. 4.1) which does symbolic execution followed by safe-fragment and inductive-invariant checks and refinement (see Fig. 2.1). It uses two helper functions: SYMBOLICEXEC (Fig. 4.2), to do symbolic execution with abstract matching and to compute the explored abstract transition system, and SAFEFRAGMENT (Fig. 4.3), to prove safety of the abstract transition system.

REFINE initializes the set Φ with all the predicates in the program's guards and in the safety property ψ (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

1: **function** Refine(P, ψ)
2: $\Phi \leftarrow$ predicates from guards in P and ψ
3: **while** *true* **do**
4: *inductive* $\leftarrow$ *true*
5: $(\mathit{fin}, \mathit{inf}, A_0) \leftarrow$ symbolicExec(P, Φ) ▷ 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 $(\mathit{inf}, \mathit{fin}, A_0)$
9: **for all** $(a_1, t, a_2) \in (\mathit{fin} \cup \mathit{inf})$ **do** ▷ inductive-invariant
10: **if** $\neg(a_1 \Rightarrow pre(a_2, t))$ **then**
11: add predicates in $pre(a_2, t)$ to Φ ▷ refinement
12: **if** $\neg(sp(a_1, t) \Rightarrow \bigvee A)$ **then** *inductive* $\leftarrow$ *false*
13: **if** *inductive* **then** return *true*

Figure 4.1: Refinement loop (main function).

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 Φ; otherwise, it returns *true*.

1: **function** SYMBOLICEXEC(P, Φ)
2: $symbStack \leftarrow$ empty stack ▷ Each item on the stack is a set of states
3: push SPLITSTATE(s_0, Φ) 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)

Figure 4.2: Symbolic execution with abstract matching.

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 symbolic state s denotes the set of concrete states it represents. 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. 4.2) using the following rules: if e_t is $x := \mathit{nondet}$ then the result is the state $s' = (f[x \to z], PC)$, where z is fresh symbolic constant (i.e., is not used in any symbolic state so far) and $f[x \to z](a)$ 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 $s' = (f[x \to z], PC')$, where z is fresh and $PC' = PC \cup (z = u[f(v_1)/v_1, \cdots, f(v_n)/v_n])$; 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 Φ. 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. 4.4. Basically, we compute a predicate abstraction of a symbolic state s and then split s using all the minterms in the result[1]. For example, consider the symbolic state $s = (\{x \mapsto x_0, y \mapsto y_0\}, \{y_0 = 0\})$ over $V = \{x, y\}$, and let $\Phi = \{x > 0, y > 0\}$. Predicate ab-

[1]A minterm is is a conjunction of literals containing all predicates in Φ.

```
1: function safeFragment(fin,inf)
2:     worklist ← inf
3:     while worklist ≠ ∅ do
4:         (a, t, b) ← remove element from worklist
5:         if ¬(a ⇒ pre(b, t)) then return false
6:         T ← {(a′, t′, b′) ∈ fin | a′ ∈ {a, b}}
7:         fin ← fin \ T
8:         worklist ← worklist ∪ T
9:     return true
```

Figure 4.3: safe-fragment check

```
function splitState(s)
    S ← ∅, X ← α_Φ(s)
    for all minterms x ∈ X do
        S ← S ∪ (f, PC ∪ x[f(v_1)/v_1, ··· , f(v_n)/v_n])
    return S
```

Figure 4.4: Splitting symbolic states.

straction of s over Φ 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. 2.2(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. 2.2(b)) it discovers a loop. Then, $trans = [(a_4, t_2), (a_3, t_4), (a_1, t_1)]$, $\text{STEM}(trans, (a_5, t_3, a_4)) = \{(a_1, t_1, a_3), (a_3, t_4, a_4)\}$, and $\text{LOOP}(trans, (a_5, t_3, a_4)) = \{(a_4, t_2, a_5), (a_5, t_3, a_4)\}$.

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, $\text{ALLPATHS}(S, trans) = \{(a_1, t_1, a_2), (a_2, t_2, a_3), (a_2, t_2, a_4)\}$.

SAFEFRAGMENT 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 4.2.2 states

that the algorithm is also sound for safety properties Of course, since property checking is undecidable, the algorithm is incomplete.

In the rest of this chapter, 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 $x \in S^a, L^a(x) = \{\phi \in \Phi \mid x \models \phi\}$.

Lemma 4.2.1. *Let* Π *be the set of all maximal paths of some program* P*. Let* Φ *be some set of predicates over the variables in* P*. Then, for every* $\pi \in \Pi$ *that has a* Φ*-maximal prefix,* SYMBOLICEXEC *only traverses the prefix of* π *that is* Φ*-maximal. For every* $\pi \in \Pi$ *that has no* Φ*-maximal prefix,* SYMBOLICEXEC *traverses the whole path.*

Proof. Choose a $\pi \in \Pi$ that has a Φ-maximal prefix. Let $c_1, \cdots, c_n$ be the Φ-maximal prefix of π. SYMBOLICEXEC must start from a symbolic state s_1 that includes c_1 (line 3). Through SPLITSTATE, $\alpha_\Phi(s_1) \Leftrightarrow \alpha_\Phi(c_1)$. Let t be the transition between c_1 and c_2. We know that $s_1 \models g_t$ and therefore, SYMBOLICEXEC executes e_t from s_1 gets to a state s_2 (after splitting the resulting state) where $\alpha_\Phi(s_2) \Leftrightarrow \alpha_\Phi(c_2)$. This process continues until we reach the symbolic state s_n. SYMBOLICEXEC backtracks only when (1) no transition can be taken (line 9), (2) or two symbolic states along the path correspond to the same abstract state (line 16). From $c_1, \cdots, c_n$, we know that the first condition does not occur on any point upto s_n. Therefore, backtracking only occurs when SYMBOLICEXEC reaches s_n, because for all $1 \leq i \leq n$, $\alpha_\Phi(s_i) \Leftrightarrow \alpha_\Phi(c_i)$ and, by definition of Φ-maximal path, no two states c_i and c_j, where $i, j \in \{1, \cdots, n-1\}$, and $i \neq j$, exist such that $\alpha_\Phi(c_i) = \alpha_\Phi(c_j)$. It follows trivially from above that if a path has no Φ-maximal prefix, SYMBOLICEXEC traverses the whole path. If a path has no Φ-maximal prefix, then it is finite, since there is a finite number of abstract states and if it was infinite, then at least two states along the path match must to the same abstract state. Therefore, SYMBOLICEXEC only backtracks when it reaches the end of the path (line 9). □

Theorem 4.2.2 (Soundness)**.** *Let ψ be a safety property, P be a program satisfying ψ, 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. Assume that* REFINE *declares M_a as safe (i.e., terminates on line 7 or 13). 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., $\forall s \in S \cdot \alpha_\Phi(s) \in S^a$).*

Proof. First, assume termination due to safe-fragment*.*

Then the explored abstract system M_a simulates the concrete system M_c. To show that, let $\rho \subseteq S \times S^a$ be the simulation relation such that $\rho = \{(s, \alpha_\Phi(s)) | s \in S\}$. We have to show the following:

1. For every $s_0 \in S_0$, $(s_0, \alpha_\Phi(s_0)) \in \rho$: This is trivial as the algorithm explores every $\alpha_\Phi(s_0)$ for every $s_0 \in S_0$ (line 4 of SYMBOLICEXEC).

2. For any two states $(s_1, a_1) \in \rho$, if $R(s_1, t, s_2)$ for some $s_2 \in S$ and some transition t, then $(s_2, \alpha_\Phi(s_2))$ is in ρ: Let $\rho(s_1, a_1)$ where $s_1 \in S$ and $a_1 \in S^a$. Assume $R(s_1, t, s_2)$, but not $R_a(a_1, t, a_2)$, where $a_2 = \alpha_\Phi(s_2)$. From SYMBOLICEXEC, we know that some $s \subseteq \gamma_\Phi(a_1)$ has been reached where s is a symbolic state. Since $a_1 \models g_i$, SYMBOLICEXEC must have applied the transition t on the state s. But $a_2 \notin S^a$, so the application of e_t on s must have resulted in an abstract state other than a_2. Therefore, by definition of inexact transitions, the transition t from a_1 is inexact. If a_1 is in *inf* or reachable from states in *inf*, then the algorithm should not have terminated with "program is safe" (because of line 5 of SAFEFRAGMENT). Otherwise, a_1 must be outside the safe fragment (i.e., in *fin* when SAFEFRAGMENT reaches line 9). From our assumption that the algorithm terminated with safe-fragment, we know that all transitions in the *worklist* when SAFEFRAGMENT terminates are exact. Therefore, if we continue execution on an any symbolic execu-

tion path, we will never visit states outside the safe fragment. From Lemma 4.2.1, this means that for every $\pi \in \Pi$, all abstract states a, such that $\alpha_\Phi(s) = a$ and s is a concrete state in π, must have been discovered by SYMBOLICEXEC. Therefore, $a_1 \xrightarrow{t} a_2$ is discovered by SYMBOLICEXEC.

Second, assume termination due to inductive-invariant.
This means that SAFE-FRAGMENT did not prove safety. Assume that $s \in S$ and $\alpha_\Phi(s) \notin S^a$. Without loss of generality, assume that there exists $s' \in S$ such that $\alpha_\Phi(s') \in S^a$ and $s' \xrightarrow{t} s$ for some transition t. Then $sp(\alpha_\Phi(s'), t_i) \Rightarrow \bigvee S^a$ is not valid, since s is in S, but not in $\bigcup_{a \in S^a} \gamma_\Phi(a)$. This contradicts the assumption that the algorithm terminated with "program is safe" in line 18 of REFINE. Thus, $\forall s \in S.\ \alpha_\Phi(s) \in S^a$. □

In contrast with other under-approximating approaches, e.g., [22, 2], 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 [22].

Theorem 4.2.3 (Monotonicity)**.** *Let Φ and Φ' 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 Φ and Φ', respectively. Then, $C \subseteq C'$.*

Proof. Let Π be the set of all maximal paths of P. From Lemma 4.2.1, for every $\pi \in \Pi$, SYMBOLICEXEC will traverse a prefix of π that is Φ-maximal. Without loss of generality, choose some $\pi \in \Pi$ such that π has a Φ-maximal prefix. Let the Φ-maximal prefix of π be $c_1, \cdots, c_m$. Then, there exists a state c_k other than c_m such that $\alpha_\Phi(c_m) = \alpha_\Phi(c_k)$. When running SYMBOLICEXEC with Φ', if $\alpha_{\Phi'}(c_m) = \alpha_{\Phi'}(c_k)$ still holds, then the same prefix $c_1, \cdots, c_m$ of π will be traversed by SYMBOLICEXEC. Otherwise, $c_1, \cdots, c_m$ will not be Φ'-maximal, and therefore symbolic execution will continue beyond c_m, exploring more concrete states. This is because simply adding new conjuncts to two unequal conjunctions

of predicates can not make them equal. And this means that no two unequal concrete states c_i and c_j can match to the same abstract state over Φ' if they do not match to the same abstract state over Φ. Similarly, in the case of finite paths that do not have a Φ-maximal prefix, adding extra predicates can not create a Φ'-maximal prefix.
Therefore, for every $\pi \in \Pi$, if π_Φ is the prefix of π explored for the predicates Φ, then, for Φ', SYMBOLICEXEC explores a prefix $\pi_{\Phi'}$ of π such that $\pi_{\Phi'}$ equals π_Φ or π_Φ is a prefix of $\pi_{\Phi'}$. □

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 [10, 22, 2]. In the future, we hope to solve this problem by using an abstract domain of tri-vectors.

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

Theorem 4.2.4. *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. 2.2(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. 2.2(c). It is obtained from symbolically executing a program obtained by replacing transition t_3 by $t_3 : pc_1 = 3 \longrightarrow pc_1 := 2,\ x := x+1$ in the protocol in Fig. 1.1, and assuming that Φ 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. □ □

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 required.

4.3 Comparison with Weak Reachability

Our under-approximating models produced by SYMBOLICEXEC are more precise compared to models produced by concrete model checking with abstract matching [22]. This is a result of our search strategy: our algorithm backtracks when it finds two states that match to the same abstract state on the same path. On the other hand, [22] backtracks when it reaches a state which matches to an abstract state that has been visisted on any path (i.e. in a globally visisted abstract table). Compared to approaches based on only *must* transitions, e.g., [23], our algorithm is more precise as it does not only explore *must* transitions, but *may* transitions as well.

We now show that an under-approximating abstract model built using symbolic execution is at least as precise as an under-approximation defined by *weak reachability* [2] (Theorem 4.3.1 below). By precision, we mean the set of feasible behaviours in the abstract model. A state a_n is *weakly reachable* from a_1 if $a_1 \rightarrow^*_{must-} a_k \rightarrow^*_{must} a_n$. This means that a_2 is reachable from a_1 via zero or more *must-* transitions, and a_3 is reachable from a_2 via zero or more *must* transitions. A transition i between two abstract states a_1 and a_2 is *must-* iff for all $s \in \gamma_\phi(a_2)$, there exists a state $s' \in \gamma_\phi(a_1)$ such that $s' \xrightarrow{i} s$.

Theorem 4.3.1. *Let Φ be a set of predicates which includes guards of a program P. If a state a' is weakly reachable from an abstract initial state a in P under Φ, then* SYMBOLICEXEC *explores a' as well. Furthermore, for some programs,* SYMBOLICEXEC *explores abstract states that are not weakly reachable.*

Proof. Let a_n be weakly reachable from an initial state a_1, s.t. $a_1 \rightarrow^*_{must-} a_k \rightarrow^*_{must} a_n$. By definition of weak reachability, there exists a non-spurious concrete path $c_1, \cdots, c_n$

s.t. for all $1 \leq i \leq n$, $\alpha_\Phi(c_i) = a_i$. We first prove that our algorithm visits every weakly reachable state, breaking the proof into two cases:

1. Assume that $a_i \neq a_j$ for any i, j s.t. $i \neq j$. Then $c_i \neq c_j$ for any i, j. Thus, $c_1, \cdots, c_n$ does not have a Φ-maximal prefix, and therefore by Lemma 4.2.1, all states in a_i will be visited.

2. Assume existence of loops, i.e., $a_i = a_j$ for some i, j s.t. $i \neq j$. Until there are no loops in the path, we perform the following: (1) Find two states a_i and a_j where $a_i = a_j$ and $i \neq j$. (2) Remove all states a_l from the path, where $i < l \leq j$. This process ensures that a_1 and a_n are not removed. By the definition of the regular expression $a_1 \rightarrow^*_{must-} a_k \rightarrow^*_{must} a_n$, no matter how many states we remove, we still have a path $a_1 \rightarrow^*_{must-} a_k \rightarrow^*_{must} a_n$ such that a_n is weakly reachable from a_1 and no state a_i occurs twice. By condition 1, our algorithm will visit state a_n.

We now prove that there are cases where our algorithm finds abstract states that are not weakly reachable. Consider the program P:

$$pc = 1 \wedge x > 0 \rightarrow pc := 2, x := x - 1$$

Assume that the initial states are $(pc = 1 \wedge x > 0)$ and we have the set of predicates $\{pc = 1, pc = 2, x > 0\}$. symbolicExec takes the only transition in the program and discovers all the reachable abstract states of P: $(pc = 1 \wedge x > 0)$, $(pc = 2 \wedge x > 0)$, and $(pc = 2 \wedge x \leq 0)$. On the other hand, weak reachability does not find any transitions from $(pc = 1 \wedge x > 0)$ since the abstract transitions between states $(pc = 1 \wedge x > 0)$ and $(pc = 2 \wedge x > 0)$ and between states $(pc = 1 \wedge x > 0)$ and $(pc = 2 \wedge x \leq 0)$ are not *must* or *must-* transitions. □

Chapter 5

Implementation and Experimental Results

We've implemented our algorithm in OCaml on top of the implementation of Pasareanu et al. [22]. We used GiNaC [4] for symbolic execution, MathSAT4 [5] for computing predicate abstraction, Simplify [9] for checking exactness of transitions and computing inductive invariants, and Bradley's implementation of Cooper's method for quantifier elimination[1]. 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. 5.1, we compare effectiveness of our *abstract analysis of symbolic executions* approach (referred to as ASE) with that of the *under-approximation refinement* algorithm of [22] (referred to as UR). We indicate whether the safety property of interest (ψ) is true (t) or false (f) and report the number of iterations (Iter.), the number of theorem prover queries (Prvr. Qurs.), the total number of predicates used (Preds.), the total amount of time needed, and the number of concrete and abstract states explored in the final iteration. In cases where the experiment did not finish after 15 minutes, the table entries are "–".

[1] Available at `http://theory.stanford.edu/~arbrad/sware.html`.

		Iter		Prvr. Qurs.		Preds.		Time(s)		Con/Abs States	
Program	ψ	ASE	UR	ASE	UR	ASE	UR	ASE	UR	ASE	UR
bakery$_2$	t	3	4	141	367	8	10	0.347	0.452	52/33	49/36
RAX	t	1	-	6	-	2	-	0.261	-	81/44	-
elev$_4$	t	1	4	418	5789	13	19	1.013	8.146	468/378	468/456
elev$_5$	t	1	5	1169	26252	15	23	3.459	44	1256/910	1253/1204
elev$_6$	t	1	6	3156	105830	17	27	12.275	220.633	3248/2126	3224/3060
elev$_7$	t	1	-	7116	-	19	-	40.867	-	8160/4862	-
elev$_8$	t	1	-	15036	-	21	-	185.717	-	15200/9422	-
ticket$_2$	t	4	4	135	120	8	8	0.609	0.404	22/9	12 / 9
ticket$_3$	t	5	5	672	661	14	14	1.413	0.923	182/31	41/31
ticket$_4$	t	6	6	4088	4061	23	23	33.51	5.143	5011/129	170/129
mesi	t	16	16	6893	12172	47	47	36.61	49.627	18/18	18/18
berkley	t	11	11	3113	4623	38	38	15.729	17.605	13/12	13/12
b_bakery$_2$-e	f	1	2	0	74	2	5	0.178	1.188	80/80	193/193
ticket$_2$-e	f	1	2	0	11	2	5	0.073	0.155	12 / 9	26 / 17
ticket$_3$-5	t	1	3	0	145	3	14	0.058	0.341	14/12	93/81
ticket$_3$-10	t	3	8	152	1218	14	21	0.525	2.229	30/27	302/240
ticket$_3$-15	t	8	13	1225	2090	21	26	3.107	5.15	47/44	507/395
ticket$_3$-20	t	13	18	2500	3918	26	31	6.869	9.501	62/59	712/550
ticket$_3$-25	t	18	23	3925	5493	31	36	13.038	15.821	77/74	917/705
ticket$_3$-30	t	23	28	5500	7219	36	41	20.762	34.701	92/89	1112/860
ticket$_3$-35	t	28	33	7225	9093	41	46	46.379	51.579	107/104	1327/1015
ticket$_3$-40	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	1068	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

Figure 5.1: Experimental results: ASE vs. UR [22].

Since UR can only handle a single concrete initial state and no non-deterministic input, these are the characteristics of all programs in Fig. 5.1 [2]. 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 [22], is deadlock-free in a single iteration, while UR refines indefinitely. We then verified the elevator program, $\texttt{elev}_i$, 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 $\texttt{ticket}_i$ 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 [8], 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 $\texttt{ticket}_i$ 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 ($\texttt{b\_bakery}_2\texttt{-e}$) and ticket ($\texttt{ticket}_2\texttt{-e}$) protocols. We also checked whether a given ticket number X in the $\texttt{ticket}_i$ protocol ($\texttt{ticket}_i\texttt{-}X$) 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. 5.2, 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

[2] `ticket`, `bakery`, and `RAX` are from [24]. `elev` is from [25]

protocols, `mesi` and `synapse`, with undefined initial states[3].

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.

Program	ψ	**Iter.**	**Prvr. Qurs.**	**Preds.**	**Symb/Abs States**	**Time**(s)
$\texttt{ticket}_2$	t	4	523	12	5516/62	281.134
$\texttt{peterson}_2$	t	1	24	4	700/38	424
$\texttt{bakery}_2$	t	3	301	11	807/50	73.965
`mesi`	t	2	260	13	112/14	3.56
`synapse`	t	2	62	7	34/9	0.88
$\texttt{ticket}_2$	f	1	0	2	12/10	0.104
$\texttt{ticket}_3$	f	1	0	3	10/10	0.112

Figure 5.2: Experimental results: programs with unspecified initial states and non-deterministic input.

[3] `peterson` and `synapse` is from [25]

Chapter 6

Related Work

The work by Pasareanu et al. [22] 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 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, 15, 6]) 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 [6], deal with sequential programs only.

Under-approximation based techniques [22, 2, 19, 23] 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., [23]) and *must* and *reverse must* transitions (e.g., [2][1]). The algorithm in [19] builds a finite bisimiluation quotient of the program under analysis,

[1]See Chapter 4.3

but unlike the global refinement employed by us and [22], 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, 13, 14, 11, 18]) concentrate on analysis of sequential programs. 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, [1] 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.

[18] improves error detection capabilities of the CEGAR framework [7] 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) [13] 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 [14] 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 attempt to cover all program paths, our approach and [22, 27] attempt to cover all reachable abstract states. [11] 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.

Chapter 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.

Bibliography

[1] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. Symbolic execution with abstraction. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(1):53–67, 2009.

[2] Thomas Ball, Orna Kupferman, and Greta Yorsh. Abstraction for Falsification. In *CAV '05: Proceedings of the 17th International Conference on Computer Aided Verification*, pages 67–81, 2005.

[3] Thomas Ball and Sriram K. Rajamani. The slam toolkit. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 260–264, London, UK, 2001. Springer-Verlag.

[4] C. Bauer, A. Frink, and R. Kreckel. Introduction to the GiNaC Framework for Symbolic Computation with the C++ Programming Language. *Journal of Symbolic Computation*, 33:1–12, 2002.

[5] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4 SMT Solver. In *CAV '08: Proceedings of the 20th International Conference on Computer Aided Verification*, pages 299–303, 2008.

[6] S. Chaki, E.M. Clarke, A. Groce, S. Jha, and H. Veith. Modular Verification of Software Components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, June 2004.

[7] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proceedings of 12th International Conference on Computer-Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 154–169, 2000.

[8] Giorgio Delzanno. Automatic Verification of Parameterized Cache Coherence Protocols. In *Proceedings of 12th International Conference on Computer-Aided Verification (CAV'00)*, pages 53–68, 2000.

[9] David Detlefs, Greg Nelson, and James Saxe. Simplify: a Theorem Prover for Program Checking. *Journal of the ACM (JACM)*, 52(3):365–473, 2005.

[10] P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based model checking using modal transition systems. In *CONCUR '01: Proceedings of the 12th International Conference on Concurrency Theory*, pages 426–440, London, UK, 2001. Springer-Verlag.

[11] P. Godefroid, A.V. Nori, S.K. Rajamani, and S.D. Tetali. Compositional May-Must Program Analysis: Unleashing the Power of Alternation. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, year = 2010.*

[12] Patrice Godefroid. Compositional dynamic test generation. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–54, New York, NY, USA, 2007. ACM.

[13] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation.*, pages 213–223, 2005.

[14] Bhargav Gulavani, Thomas Henzinger, Yamini Kannan, Aditya Nori, and Sriram Rajamani. Synergy: a new algorithm for property checking. In *SIGSOFT '06/FSE-14:*

Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, pages 117–127, New York, NY, USA, 2006. ACM.

[15] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *POPL'02: Proceedings of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 58–70, January 2002.

[16] Gerard Holzmann and R. Joshi. Model-Driven Software Verification. In *Proceedings of the SPIN Workshop (SPIN'04)*, volume 2989 of *LNCS*, pages 76–91, 2004.

[17] G.J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), 1997.

[18] D. Kroening, A. Groce, and E. Clarke. Counterexample Guided Abstraction Refinement via Program Execution. In *ICFEM'04:Proceedings of the 6th International Conference on Formal Engineering Methods*, pages 224–238, 2004.

[19] David Lee and Mihalis Yannakakis. Online Minimization of Transition Systems. In *STOC'92: Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 264–274, 1992.

[20] M. Musuvathi and S. Qadeer. CHESS: Systematic Stress Testing of Concurrent Software. In *LOPSTR'06: Proceedings of 16th International Symposium on Logic-Based Program Synthesis and Transformation*, volume 4407 of *LNCS*, pages 15–16, July 2006.

[21] Aditya Nori, Sriram Rajamani, SaiDeep Tetali, and Aditya Thakur. The YOGI Project: Software Property Checking via Static Analysis and Testing. In *TACAS'09: Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 5505 of *LNCS*, pages 178–181, 2009.

[22] C. Pasareanu, R. Pelanek, and W. Visser. Concrete Model Checking with Abstract Matching and Refinement. In *CAV '05: Proceedings of the 17th International Conference on Computer Aided Verification*, volume 3576 of *LNCS*, pages 52–66, 2005.

[23] Corina Pasareanu, Matthew Dwyer, and Willem Visser. Finding Feasible Counter-examples when Model Checking Abstracted Java Programs. In *TACAS'01: Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 284–298, April 2001.

[24] Corina S. Pasareanu, Radek Pelánek, and Willem Visser. Predicate abstraction with under-approximation refinement. *Logical Methods in Computer Science*, 3(1), 2007.

[25] Radek Pelánek. Beem: Benchmarks for explicit model checkers. In Dragan Bosnacki and Stefan Edelkamp, editors, *Proceedings of the SPIN workshop (SPIN'07)*, volume 4595 of *Lecture Notes in Computer Science*, pages 263–267. Springer, 2007.

[26] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272, New York, NY, USA, 2005. ACM.

[27] Greta Yorsh, Thomas Ball, and Mooly Sagiv. Testing, Abstraction, Theorem Proving: Better Together! In *ISSTA'06: Proceedings of International Symposium on Software Testing and Anlaysis*, pages 145–156, 2006.