

FINDING TERMINATION AND TIME IMPROVEMENT IN  
PREDICATE ABSTRACTION WITH UNDER-APPROXIMATION  
AND ABSTRACT MATCHING

by

Dritan Kudra

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

August 2007

Copyright © 2007 Dritan Kudra  
All Rights Reserved

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Dritan Kudra

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

\_\_\_\_\_  
Date

\_\_\_\_\_  
Eric Mercer, Chair

\_\_\_\_\_  
Date

\_\_\_\_\_  
Michael Jones

\_\_\_\_\_  
Date

\_\_\_\_\_  
Robert Burton

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Dritan Kudra in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

---

Date

---

Eric Mercer  
Chair, Graduate Committee

Accepted for the  
Department

---

Parris K. Egbert  
Graduate Coordinator

Accepted for the  
College

---

Thomas W. Sederberg  
Associate Dean, College of Physical and Mathematical  
Sciences

## ABSTRACT

### FINDING TERMINATION AND TIME IMPROVEMENT IN PREDICATE ABSTRACTION WITH UNDER-APPROXIMATION AND ABSTRACT MATCHING

Dritan Kudra

Department of Computer Science

Master of Science

The focus of current formal verification methods is mitigating the state explosion problem. One of these formal methods is predicate abstraction, which reduces concrete states of a system to bitvectors of true/false valuations of a set of predicates. Predicate abstraction comes in two flavors, over-approximation and under-approximation. A drawback of over-approximation is that it produces too many spurious errors for data-intensive applications. A more recent under-approximation technique which does not produce spurious errors, does abstract matching on concrete states (AMCS). AMCS adds behaviors to an abstract system by augmenting the set of initial predicates, making use of a theorem prover. The logic behind this approach is that if an error is found in the early coarse abstractions of the system, we save space and time. Our research improves AMCS by providing a refinement technique which guarantees termination. Our technique finds errors in less time and space by

using an abstract state splitting algorithm based on intervals, which does not require a theorem prover.

## ACKNOWLEDGMENTS

Professionally, I would like to thank my committee chair, Eric Mercer, for giving me invaluable counsel and guidance in my research. I would also like to thank Michael Jones, for giving thoughtful ideas and insights, during different stages of this research.

Personally, I would like to thank my parents, Fatos and Vjollca, and my sister Teuta, for their love and sacrifice during the years that I have been in the university.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Background</b>	<b>4</b>
2.1	Precise Abstraction . . . . .	4
2.2	AMCS . . . . .	4
2.2.1	The AMCS Model . . . . .	5
2.2.2	The AMCS algorithm . . . . .	5
<b>3</b>	<b>Solution</b>	<b>11</b>
3.1	Splitting Technique . . . . .	11
<b>4</b>	<b>Our Techniques</b>	<b>14</b>
4.1	MaxOverlap . . . . .	15
4.2	MinSet . . . . .	16
4.3	MinOnly . . . . .	17
<b>5</b>	<b>Experiment and Results</b>	<b>20</b>
5.1	Implementations . . . . .	20
5.2	Hardware . . . . .	20
5.3	Independent Variables . . . . .	21
5.4	Dependent Variables . . . . .	21
5.5	Results . . . . .	21
5.5.1	Table Legend . . . . .	22

5.5.2	Data Interpretation . . . . .	23
5.5.3	Threats to Validity . . . . .	27
<b>6</b>	<b>Conclusions and Future Work</b>	<b>29</b>

## List of Figures

2.1	AMCS non-terminating example . . . . .	8
4.1	MaxOverlap example . . . . .	16
4.2	MinSet example . . . . .	17
4.3	MinOnly example . . . . .	18

## List of Tables

2.1	AMCS error-free search . . . . .	10
5.1	Normalized time results . . . . .	23
5.2	Per model percentage of errors found over total errors . . . . .	23
5.3	Normalized states results . . . . .	24
5.4	Normalized error depth results . . . . .	24
5.5	Normalized predicate results . . . . .	25

# Chapter 1

## Introduction

Explicit model checking shows the relationship between a system and a formula using state enumeration. The system in our research is represented by a piece of software, and for our purposes, the goal of model checking is to find errors inside this software. The traditional approach to finding errors is through extensive testing of different scenarios in the system; however, traditional testing with sampling does not work for large systems where correctness is essential because often there are bugs in the untested areas of the system. Systems such as vehicle embedded software, transaction protocols, or critical parts of operating systems, need almost absolute correctness because of financial and safety reasons.

A way to model check a system is a method called *explicit state enumeration* (ESE). In ESE we verify a system by creating a connected graph of all the reachable states. This technique is effective but not feasible for solving most real-life problems due to memory and time limitations in current hardware. As systems get more complex, their state graph becomes exponentially bigger causing a problem called *state explosion*.

One of the major areas of research in verification and validation is mitigating the state explosion problem. Some ways to deal with this problem are: symbolic model checking, dead variable analysis, symmetry reduction, and predicate abstraction. Symbolic model checking addresses the space explosion problem by using binary decision diagrams to store the possible states in a system [12]. Dead variable analysis

traverses a program looking for dead variables in order to avoid storing them inside states [11]. Symmetry reductions find symmetries in a state graph in order to store only one of the symmetric sides [10]. Predicate abstraction, which is the general area of our research, stores valuations of predicates, rather than concrete states, in order to minimize the stored information and still preserve error-detection capacity [8].

In predicate abstraction, we map a concrete state to a bitvector of true/false valuations of a set of predicates. More formally, this mapping is done by an abstraction function  $\alpha_{\Phi}(s)$ , where  $\Phi$  is a predicate set,  $s$  is a concrete state, and  $\alpha$  builds the bitvector by evaluating each predicate in  $\Phi$  on  $s$ . An abstract state is formed by AND-ing all the predicate valuations denoted by the bitvector.

Predicate abstraction can over-approximate or under-approximate [13] a system. Over-approximation happens when the abstraction has additional behaviors compared with the concrete system. In this case, if an error is found in the abstraction, we know that there might be an error in the concrete system, but we are not sure because the abstraction has extra-behaviors. On the other hand, in under-approximation, the abstraction has only behaviors that exist in the concrete system. If an error is found in the abstraction, we are guaranteed an error in the concrete system [15, 4, 8, 7]. Our research is in this latter sub-field of predicate abstraction.

A recently published under-approximating technique uses *abstract matching on concrete states* [13]. For simplicity, we call this method *AMCS*. Abstract matching on concrete states means that during the search we still explore concrete states, just like in ESE, but the matching for existing states is done on abstract states. In order to save time and space, AMCS progressively includes behaviors of the real system by augmenting the predicate-set on successive iterations. As the predicate-set is augmented, new behavior is added to the abstract system in an effort to find errors. This process is called *refinement*. Refinement terminates when the abstraction is a precise representation of the real system or when an error is discovered. Since the

check for errors in the abstract system is done after each iteration, it is hoped to detect an error during the early refinement steps without having to store or explore a lot of behaviors from the real system.

AMCS suffers from a few drawbacks. Its process of refinement through augmentation of the predicate-set can potentially not terminate because in most cases, AMCS cannot detect that the abstract system is a bisimulation of the real system. When the algorithm terminates without finding errors, most often it has explored the whole concrete state graph. In addition, AMCS employs a theorem prover to detect the need for refinement, but this external tool consumes the majority of the search time and makes error discovery very slow.

In our research, we address the non-termination and the large theorem prover overhead of AMCS. We provide a comparison study on three new techniques that use the same under-approximating search pattern as AMCS but guarantee termination and need no theorem prover. The main idea behind our new techniques is that they augment the abstract state space in iterative steps until either an error is found or we have a 1:1 relation between the concrete and abstract state spaces. The abstract state space is augmented by splitting those abstract states that have more than one concrete state match. This multiple-match condition is detected by storing inside each abstract state, max/min values from the concrete states matching to it. These new techniques outperform AMCS in error discovery and each has different strengths and weaknesses, which we explore in latter sections of this paper.

# Chapter 2

## Background

In this section we give a detailed explanation of the AMCS algorithm. In order for the reader to better understand this algorithm, we need to introduce the concept of *precise abstraction*.

### 2.1 Precise Abstraction

Consider a transition  $s_1 \xrightarrow{i} s_2$  between two concrete states and an abstraction function  $\alpha_\Phi$ . An abstraction is considered precise if every concrete state in  $\alpha_\Phi(s_1)$  transitions under  $i$  to at least one concrete state in  $\alpha_\Phi(s_2)$  [1, 5, 6]. The abstraction would not be precise if at least one concrete state in  $\alpha_\Phi(s_1)$  does not transition to any concrete states in  $\alpha_\Phi(s_2)$  [2, 3, 8, 16].

### 2.2 AMCS

AMCS makes use of weakest precondition to detect precise abstraction in order to stop its refinement process. The idea behind the AMCS algorithm is to do as little refinement as possible before finding errors, since more refinement usually means more states to store. The interested reader is referred to [13] for more details.

### 2.2.1 The AMCS Model

The program model that AMCS uses is just a list of guarded transitions. More formally, a model  $M$  is defined as:  $M = \{(V, T) \mid T = \{t_1, \dots, t_k\}\}$  where each  $t_i$  is a transition such that  $t_i = (g_i(x) \rightarrow x = e_i(x))$ . In this setup,  $x$  is a set of variables and  $e_i(x)$  is the set of assignments to these variables under transition  $i$ .  $V$  represents a set of integer variables that is finite. The guard  $g_i$  is the conjunction of a mandatory program location and an optional predicate  $p$ . The concrete state space generated by a model  $M$  is a transition system  $\|M\|$  defined as:  $\|M\| = (S, \{\xrightarrow{i}\}, s_0, L)$ , where  $S$  is a set of concrete states,  $\{\xrightarrow{i}\}$  is a transition relation between concrete states,  $s_0$  is a start concrete state, and  $L$  is a labeling function where  $L(s) = \{p \in AP \mid s \models p\}$  for a set of atomic propositions  $AP$ . The notation  $s \models p$  means that  $p$  holds in  $s$ . The set of reachable labelings in a transition system  $\|M\|$  is:  $RL(\|M\|) = \{L(s) \mid s \in S : s_0 \xrightarrow{*} s\}$

### 2.2.2 The AMCS algorithm

AMCS has two parts which execute in turns: the *control* shown in Algorithm 1 and the *search* shown in Algorithm 2. At the start, AMCS has a set of initial predicates that is augmented during each call to the *search* algorithm. Each of these calls to *search* constitutes a refinement step. The *control* gets an abstract system from the *search* at line 3 and checks it for errors. More formally, the abstract system created by *search* is defined as:  $\alpha_\Phi(\|M\|) = (States, Transitions)$  returned by  $search(M, \Phi)$ . The *control* also gets a new set of predicates from the *search* at line 3, and at line 7, checks it against the old set of predicates to see if it is refined. In the *control*, if the predicate-set is refined and no error is found, we call the *search* again to build another abstract system from the new set of predicates. Also in the *control*, if the predicate-set is not refined, we know that the abstraction is precise, and we terminate because there are no more behaviors left to consider.

---

**Algorithm 1** control(*error\_condition*)

---

```
1:  $i = 1$ ;  $\Phi_i =$  initial predicate set;  
2: while true do  
3:   (States, Transitions,  $\Phi_{i+1}$ ) = search( $M$ ,  $\Phi_i$ )  
4:   if error_condition is reachable in (States, Transitions) then  
5:     return counter-example  
6:   end if  
7:   if  $\Phi_{i+1} = \Phi_i$  then  
8:     return unreachable  
9:   end if  
10: end while
```

---

Algorithm 2 is responsible for building the abstract state-space from the current set of predicates and also for creating the new set of predicates for a possible next iteration. The concrete-state search is breadth-first (BFS) and there is a BFS *Wait* queue introduced at line 1, where we put only those concrete states that map to new abstract states. Matching for states already seen is done on abstract states; this check is at line 13. At the end of the *search*, we return  $\alpha_\Phi(\|M\|)$  and  $\Phi_{new}$  to the *control* on line 24.

Refinement is the augmentation of our set of predicates  $\Phi$  to create a new set of predicates  $\Phi_{new}$ , which should be one step closer in making the abstraction in the abstract transition graph precise with respect to all transitions; though, not guaranteed. Most of the time, refinement is done by computing the weakest precondition (*wp*) of the abstraction of a successor of a concrete state and checking to see if the weakest precondition includes the concrete state. Based on set theory, this inclusion would mean that our abstraction is still precise; the first violation of this inclusion would mean an imprecise abstraction. The main part of refinement is  $\alpha_\Phi(s) \Rightarrow \alpha_\Phi(s')[e_i(x)/x]$  on line 10, which is equivalent to  $\alpha_\Phi(s) \Rightarrow wp(\alpha_\Phi(s'), i)$ . Note that the weakest precondition  $wp(\alpha_\Phi(s'), i)$  of an abstract state  $\alpha_\Phi(s')$ , is a predicate-defined set that represents all possible concrete states that end up inside  $\alpha_\Phi(s')$  after taking transition  $i$ . Intuitively, the weakest precondition of an abstract state under transition  $i$  is the abstract state

---

**Algorithm 2** search  $(M, \Phi)$ 

---

```
1:  $\Phi_{new} = \Phi$ ; add  $s_0$  to Wait; add  $\alpha_\Phi(s_0)$  to States
2: while Wait  $\neq 0$  do
3:    $s = \text{Wait.dequeue}()$ 
4:   for all possible transitions  $i$  from  $s$  do
5:     if  $s \models g_i$  then
6:       if  $\alpha_\Phi(s) \Rightarrow (\text{guard of } i)$  then
7:         add  $g_i$  to  $\Phi_{new}$ 
8:       end if
9:        $s' = \text{successor of } s \text{ after taking transition } i$ 
10:      if  $\alpha_\Phi(s) \Rightarrow \alpha_\Phi(s')[e_i(x)/x]$  is not valid then
11:        add predicates in  $\alpha_\Phi(s')[e_i(x)/x]$  to  $\Phi_{new}$ 
12:      end if
13:      if  $\alpha_\Phi(s') \notin \text{States}$  then
14:         $\text{Wait.enqueue}(s')$ ; add  $\alpha_\Phi(s')$  to States
15:      end if
16:      add  $(\alpha_\Phi(s), i, \alpha_\Phi(s'))$  to Transitions
17:    else
18:      if  $\alpha_\Phi(s) \Rightarrow (\text{complement of guard of } i)$  then
19:        add  $g_i$  to  $\Phi_{new}$ 
20:      end if
21:    end if
22:  end for
23: end while
24: return  $(\text{States}, \text{Transitions}, \Phi_{new})$ 
```

---

created by applying the inverse of transition  $i$  to  $\alpha_\Phi(s')$ . The implication in line 10 of Algorithm 2 is interpreted as the set inclusion of the left side into the right side. The evaluation of this implication is done with a theorem prover which is implicitly incorporated into the algorithm. The order of operation on the right side of the implication is: we first evaluate  $\alpha_\Phi(s')$  and then substitute every occurrence of variable  $x$  with its valuation under transition  $i$ . To form  $\Phi_{new}$ , we add to  $\Phi$  the predicates from the weakest precondition. Adding the wp-predicates, is an attempt to assure that our precision check  $\alpha_\Phi(s) \Rightarrow wp(\alpha_\Phi(s'), i)$  does not fail on the next iteration on this same abstract state and transition. Note that there are cases where  $\alpha_\Phi$  is precise with respect to  $s \xrightarrow{i} s'$ , but the weakest precondition does not detect that precision. One such case happens when there are no states  $s'$  in the concrete state space such that

```

l0:   x:=0; y:=0;
l1:   while(y≥0){
l2:       y:=x+y;
          }

```

Figure 2.1: AMCS non-terminating example

$\alpha_{\Phi}(s) = \alpha_{\Phi}(s')$  for  $s \neq s'$ , and the check  $\alpha_{\Phi}(s) \Rightarrow wp(\alpha_{\Phi}(s'), i)$  fails. This drawback makes weakest precondition an uncertain technique for detecting precise abstraction.

It is to be noted that adding more predicates using weakest precondition does not necessarily add more refinement in the abstract state space. It could happen that after augmenting the predicate set, the abstract system graph does not change shape and all the abstract states include the same concrete states that they had before the augmentation [7, 15]. So refining with weakest precondition is just a heuristic that potentially adds precision during each refinement step, but it does not guarantee a bounded number of refinements.

An example where the weakest precondition fails to refine the system is given in Figure 2.1. This is a pathological example taken from [13] (although, our experiments show that similar constructs occur frequently). The initial predicate set is  $\Phi = (y \geq 0)$ . When reaching transition  $y = x + y$ , after substituting and abstracting on line 10, we get the implication  $((y \geq 0) = T) \rightarrow ((x + y) \geq 0) = T$ . The implication fails, and we add predicate  $(x + y) \geq 0$  to  $\Phi_{new}$ . On the successive iterations, we augment our predicate set with predicates of the form  $(ax + y)$ , where  $a$  is always increasing [9, 13]. The algorithm does not terminate because new predicates, which do not alter the abstract state space, are infinitely added to our predicate-set on each call to *search*. These new predicates add no relevant information that the theorem prover can use to resolve the implication.

Even in the cases when AMCS does terminate, most often it ends up exploring the entire concrete state space of a model. We discover this in implementing AMCS

inside the Bogor model checker using BIR as the input language for the models. In this preliminary experiment we wanted to see how often AMCS terminates and how often it finds an abstract bisimulation that has less states than the concrete system. Our results from this experiment are shown in Table 2.1. In 6 out of 10 models, the search does not terminate, which is indicated by *NT*. When it terminates, three out of four times we have no reduction in the final number of states compared with the concrete state graph. In addition, the theorem prover takes about 76% of the search time on the Table 2.1 models.

Two techniques are mentioned in [13] to somewhat mitigate non-termination and theorem prover overhead. The first is *Add-All AMCS*. The main idea in Add-All AMCS is that whenever we have an imprecise transition, we add to our global set of predicates the predicates describing the concrete state where this transition originated. The addition can be done after a transition has shown to be imprecise in a number of consecutive iterations of *search*. Termination is guaranteed by this variation of AMCS, but one can imagine how big our predicate set gets after adding predicates to describe full concrete states every time the search encounters an imprecise transition. In addition, our experiments show that Add-all AMCS is too slow in error-discovery due to the theorem prover.

The second technique is *No-TP AMCS*. The main idea here is that we do not employ a theorem prover. The technique considers all transitions as imprecise and adds predicates with weakest precondition on all of them. No-TP AMCS does not incur the overhead of the theorem prover, but if the search does not exit on error, non-termination is unavoidable since transitions are never checked for precision in the abstraction. Treating all transitions as imprecise means that we increment our set of predicates on all of them, making this set grow large quickly. The results of our experiments show that No-TP AMCS is also slow because of the overhead required in computing new predicates on every transition. Each solution that we provide in this

Table 2.1: AMCS error-free search

Models	States Explored
Barbershop	4621 states (no gain)
Branches	NT
CustWorkers	5014 states (no gain)
DiningPhil	2847 states (no gain)
Fibonacci	NT
Factorial	NT
Handshake	79/298 bissimulation
Hyman	NT
Stack	NT
SumToN	NT

paper, in itself, mitigates the termination and theorem-prover-overhead problems. In addition, our solutions prove to be more effective in error-discovery than AMCS, Add-All AMCS, and No-TP AMCS.

# Chapter 3

## Solution

In essence, AMCS splits abstract states until it either finds an error or a precise representation of the concrete system. Its splitting technique guarantees that it will eventually reach a bisimilar structure to the concrete system, but the technique could also keep adding predicates with weakest precondition even after we have reached a bisimulation. So in our solution, we are looking for splitting techniques that guarantee termination and find errors faster. Below we introduce a general technique for achieving this goal.

### 3.1 Splitting Technique

In order to split abstract states, we need a splitting condition that indicates which states to split. The splitting condition is satisfied when an abstract state has at least two distinct concrete states mapped to it. More formally this is expressed as:

Having  $\|M\| = (S, \{\overset{i}{\rightarrow}\}, s_0, L)$ , two states  $s_1, s_2 \in S$  meet the splitting condition if  $\alpha_\Phi(s_1) = \alpha_\Phi(s_2) \wedge s_1 \neq s_2$ . We denote this splitting condition as a relation  $SplitCond(s_1, s_2)$ , that returns true when  $s_1$  and  $s_2$  meet the condition.

To illustrate a situation where  $SplitCond(s_1, s_2)$  holds, consider concrete states  $s_1 = (x = 3)$  and  $s_2 = (x = 4)$ . These states match under the same abstract state  $\alpha = (x < 5)$ . In this case,  $SplitCond(s_1, s_2)$  returns true because  $\alpha_\Phi(s_1) = \alpha_\Phi(s_2)$  and  $s_1 \neq s_2$ .

---

**Algorithm 3** GeneralSplitting( $\|M\|, \Phi$ )

---

```
1: while (true) do
2:   if error found in  $\alpha_\Phi(\|M\|)$  then
3:     return counter – example
4:   end if
5:   add predicates  $\Phi_{new}$  to split all members of  $Eligible(\Phi)$ 
6:   if ( $\Phi = \Phi_{new}$ ) then
7:     return unreachable
8:   end if
9:    $\Phi = \Phi_{new}$ 
10: end while
```

---

Now we need to define the set of abstract states that satisfy the splitting condition. These are the states that we can split. We formally call this set  $Eligible(\Phi)$ . For a transition system  $\|M\|$ :  $Eligible(\Phi) = \{\alpha \mid \exists s_1, s_2 \in S. SplitCond(s_1, s_2) \wedge \alpha = \alpha_\Phi(s_1)\}$

Now our general technique for splitting becomes: refine  $\Phi$  such that each member of  $Eligible(\Phi)$  is split in  $\Phi_{new}$ . This technique is implemented in Algorithm 3. We pass to the algorithm a concrete transition system  $\|M\|$  and an initial set of predicates  $\Phi$ . The output is either a counter-example to the error or *unreachable*, which means that no error was found. Each iteration of the outside *while* loop represents a *refinement step*. On each refinement step, we first abstract the concrete system  $\|M\|$  creating  $\alpha_\Phi(\|M\|)$ , then search this abstraction for errors. The search happens in line 2. If no error is found, we need to add more behaviors to the abstract system for the next refinement step. The behaviors are added by augmenting  $\Phi$  to create  $\Phi_{new}$ , which is the set of predicates for the next iteration. The set is augmented on line 5, where we add predicates that split all members of  $Eligible(\Phi)$ . If  $Eligible(\Phi)$  is empty then  $\Phi = \Phi_{new}$ , and we return *unreachable* in line 7.

The main difference between AMCS and GeneralSplitting is that AMCS splits abstract states until it creates a bisimulation of the concrete system, which it often cannot detect. On the other hand, GeneralSplitting splits abstract states until it

creates a 1:1 relationship between the concrete and abstract systems. Now let's prove that *GeneralSplitting* fulfills our expectations; meaning that it always terminates and always finds valid errors.

**Theorem 1:** *GeneralSplitting*( $\|M\|, \Phi$ ) always terminates if  $\|M\|$  is finite.

**Proof:** The only way in which *GeneralSplitting* could not terminate is if *Eligible*( $\Phi$ ) never becomes empty. We proceed by contradiction. Assume that *Eligible*( $\Phi$ ) never becomes empty. This means that when running *GeneralSplitting*( $\|M\|, \Phi$ ) for a finite  $\|M\|$ , every refinement step, *SplitCond*( $s_1, s_2$ ) is true for some pair of states  $s_1, s_2$ . If *SplitCond*( $s_1, s_2$ ) is always true, we always keep splitting concrete states from falling under the same abstract state, and there are always abstract states to divide. Continuous splitting could only arise if  $\|M\|$  is infinite, which contradicts the original assumption.

**Theorem 2:** If *GeneralSplitting*( $\|M\|, \Phi$ ) returns unreachable, then the error is indeed unreachable in  $\|M\|$ .

**Proof:** *GeneralSplitting*( $\|M\|, \Phi$ ) returns unreachable when *Eligible*( $\Phi$ ) is empty, which by definition happens when no two different concrete states match under the same abstract state. In this case there is a 1:1 relationship between abstract states in  $\alpha_\Phi(\|M\|)$  and concrete states in  $\|M\|$ . Based on this 1:1 relationship, we can say that if there is no error in  $\alpha_\Phi(\|M\|)$ , then there is no error in  $\|M\|$  either.

**Theorem 3:** If *GeneralSplitting*( $\|M\|, \Phi$ ) returns an error, this error exists in  $\|M\|$ .

**Proof:** All errors found inside  $\alpha_\Phi(\|M\|)$  exist in  $\|M\|$  because by definition,  $\alpha_\Phi(\|M\|)$  is an under-approximation of  $\|M\|$ .

## Chapter 4

### Our Techniques

There are two things missing from *GeneralSplitting* to make it implementable. First, we need a search algorithm that produces  $\alpha_{\Phi}(\|M\|)$  as an under-approximation of  $\|M\|$ . The second missing item is a way to detect a member of  $Eligible(\Phi)$  and add a predicate to split it. To deal with the first problem we borrow the BFS concrete search with abstract matching from AMCS, which always produces an under-approximation of  $\|M\|$ . The search is shown in Algorithm 4, which we can call at line 2 of Algorithm 3 when we create  $\alpha_{\Phi}(\|M\|)$ . The search is similar to Algorithm 2 with the exception that it does not employ the theorem prover. So lines 6, 10, 11, and 18 of Algorithm 2 are omitted in Algorithm 4. To solve the second problem of computing the eligible set, we store inside an abstract state the maximum and minimum values for each variable of the concrete states matching to this abstract state. Intuitively, each max-min pair forms a segment in space. Adding a predicate that splits one of these segments also splits its owning abstract state. Mathematically, if there is a variable  $x$  for which  $max(x) \neq min(x)$ , we know that this abstract state has more than one concrete state match to it, and we can add a splitting predicate.

During the design of our different error-search solutions, we are interested in keeping the abstract state space small to avoid state explosion at the abstraction level. It is true that *GeneralSplitting* explores the whole concrete state space in its worst case, but usually we hope to discover the error in the earlier and coarser abstractions of the concrete system. So an abstraction technique that explores the

---

**Algorithm 4** BFS-AMCS( $M, \Phi$ )

---

```
1:  $\Phi_{new} := \Phi$ ; add  $s_0$  to Wait; add  $\alpha_\Phi(s_0)$  to States
2: while Wait  $\neq 0$  do
3:    $s = \text{Wait.dequeue}()$ 
4:   for all possible transitions  $i$  from  $s$  do
5:      $s' :=$  successor of  $s$  after taking transition  $i$ 
6:     if  $\alpha_\Phi(s') \notin \text{States}$  then
7:        $\text{Wait.enqueue}(s')$ ; add  $\alpha_\Phi(s')$  to States
8:     end if
9:     add  $(\alpha_\Phi(s), i, \alpha_\Phi(s'))$  to Transitions
10:  end for
11: end while
12: return (States, Transitions,  $\Phi_{new}$ )
```

---

whole concrete state space on the second or third refinement step is not ideal because it does not give much room for early error discovery in a small abstract state space. On the other hand, we also have to consider not increasing the abstract state space too slowly since this incurs unnecessary time overhead until an error is found. With these ideas in mind, we present three different under-approximation search heuristics in the following sections.

## 4.1 MaxOverlap

Being frivolous in adding predicates is not a good idea because we might hit state explosion fast and not find any errors. We present a technique in which each refinement step of Algorithm 3 adds to  $\Phi_{new}$  the predicate that divides the most abstract states. We compute this predicate by plotting in an axis, for each variable, the max-min segments stored inside each abstract state. Then for each plot we add a predicate at a point of maximum overlap between these segments. From all these computed predicates, we pick the one that splits the most states. If there are ties, we choose randomly between the candidates.

An example that illustrates MaxOverlap is shown in Figure 4.1. The figure shows three abstract states  $\alpha_1, \alpha_2$ , and  $\alpha_3$ , which hold their max-min segments for a

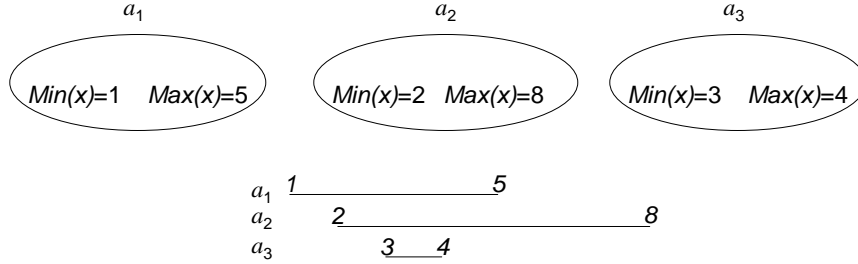


Figure 4.1: MaxOverlap example

variable  $x$ . In order to add a predicate that splits the most of these states, we plot all max-min segments and compute the points of maximum overlap. In this case, any point in the segment  $[3, 4]$  is a maximum overlap point. If we pick the point 3 and add predicate  $x > 3$ , that predicate effectively splits all three abstract states.

## 4.2 MinSet

Adding one maximum-overlapping predicate every time we abstract the concrete system, as in MaxOverlap, is good for keeping the abstract state space small, but it may be slow in error-discovery because the abstract state space increases in very small increments. To speed up error discovery, we present a heuristic called *MinSet*. In MinSet, over all the variables in the program, we find a minimal set of predicates that splits all states in  $Eligible(\Phi)$ .

An example that illustrates MinSet is shown in Figure 4.2. In the figure we again have three abstract states  $\alpha_1, \alpha_2$ , and  $\alpha_3$  which we want to split; however, now we have stored max-min segments for variables  $x$  and  $y$ . Just as in MaxOverlap, we plot the max-min segments for each variable. Now we traverse the plotted graph looking for a minimal set of predicates that splits all states. In this case, for variable  $x$ , one such set is  $\{(x > 2), (x > 6)\}$ ; for variable  $y$  a possible set is  $\{(y > 2), (y > 6)\}$ . Now we put all these predicates and all our abstract states in a table as shown in Figure 4.2, assigning a 1 where the predicate splits a state, and a 0 when it does not.

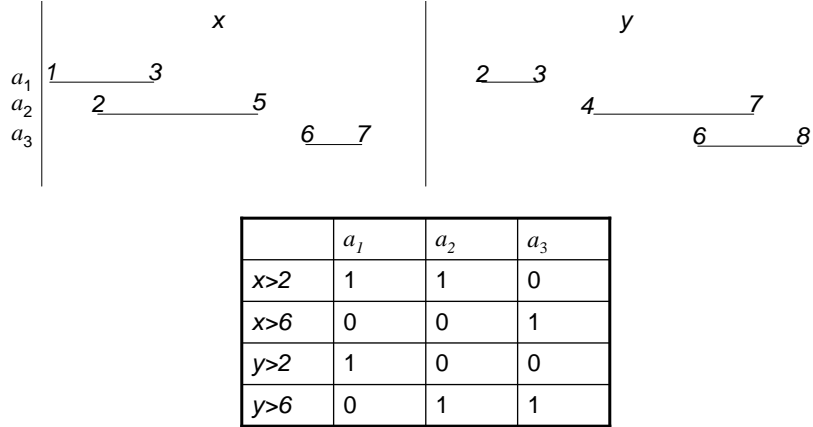


Figure 4.2: MinSet example

The next step is to search the table and compute a minimum set of predicates that splits all states, which in essence, is the minimum covering problem. We solve this problem with a standard greedy heuristic that iterates through the table, finding and storing the predicate that splits the most abstract states. The process continues until we have predicates for splitting all abstract states. The complexity of this algorithm is  $N^2 \log(N)$ . After we have found this minimal set of predicates, we add it to our global set of predicates to be used in the next iteration. Since we use a greedy algorithm, the final set found is not guaranteed to be the optimal answer, but a more comprehensive algorithm might also require more time and memory.

### 4.3 MinOnly

The techniques that we have used so far for detecting members of  $Eligible(\Phi)$  store inside an abstract state both the maximum and minimum values for each variable. This is necessary when we are computing overlapping of max-min segments, but what if there is a more efficient way to detect and split members of  $Eligible(\Phi)$ . If we could improve this operation in terms of time and memory, we would be able to find more errors faster, since the whole search would be faster and more abstract states could be

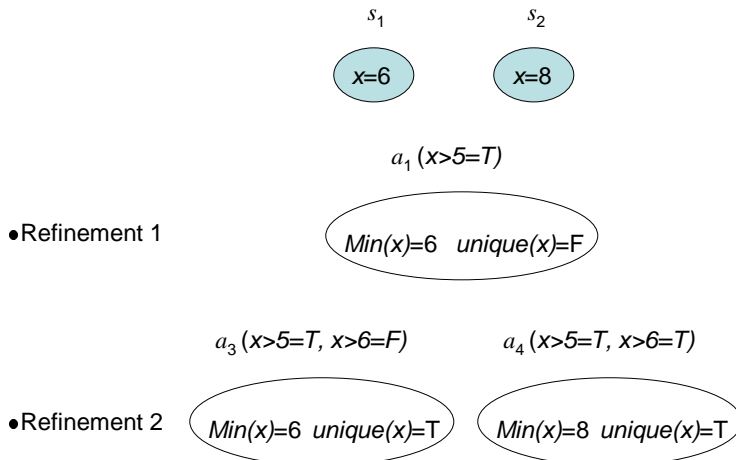


Figure 4.3: MinOnly example

stored. One such technique is accomplished by storing inside an abstract state for each variable only the minimum values and a boolean *unique*. The variable *unique* tells us whether two or more distinct values are used to compute its corresponding minimum, which happens only if this abstract state is a member of  $Eligible(\Phi)$ . We call this method *MinOnly*. Just like in *MinSet*, in every refinement step of Algorithm 3, we add predicates to split all members of  $Eligible(\Phi)$ . This time, however, we do not incur the computational cost of finding a minimal set to accomplish this task, but instead, for each member of  $Eligible(\Phi)$ , we add a predicate of the form  $var > min(var)$ . Since we store minimum values for every variable in the program, there could be more than one variable that we could use to compute this predicate. To break these ties, we randomly pick between the variables that have minimums that can split the abstract state, which by design, are the variables that have their *unique* set to false <sup>1</sup>.

An example that illustrates *MinOnly* is shown in Figure 4.3. In the figure we have two concrete states  $s_1$  and  $s_2$  storing values for variable  $x$ . Both concrete states

<sup>1</sup>We randomly break ties in all algorithms to control for default search order

match under the same abstract state  $\alpha_1$  for predicate  $(x > 5)$ . The abstract state  $\alpha_1$  is a member of  $Eligible(\Phi)$  because more than one distinct state matches to it, but in order to detect this fact, we have stored inside it the minimum value for  $x$  and have set *unique* to false to indicate that at least two distinct values are used to compute this minimum. Now if we want to split  $\alpha_1$ , all we need to do is add the new predicate  $(x > \min(x))$  to our set of predicates. In this case, the new predicate is  $(x > 6)$ . A predicate of the form  $(x \neq \min(x))$  can also accomplish the same purpose, but since it is less general than  $(x > \min(x))$ , it might increase the number of predicates before the error is found. As it can be seen in the figure, in the second refinement,  $\alpha_1$  is split into  $\alpha_3$  and  $\alpha_4$ , which are not members of  $Eligible(\Phi)$  and can be split no further.

# Chapter 5

## Experiment and Results

In our experiment we wanted to see which of the following approaches finds errors faster using fewer resources. The approaches we consider are: MaxOverlap, MinSet, MinOnly, RandomDFS, AMCS, AMCSNoTP, and AddAll-AMCS.

### 5.1 Implementations

In order to analyze AMCS and our techniques, we implement them in java as extensions to *Bogor*, which is an extensible software model checking framework developed at Kansas State University [14]. In our implementations we use the *Simplify* theorem prover, which is also used in the AMCS paper [13]. The input-models are written in *BIR*, which is the Bogor modeling language. We also implement a random depth-first concrete search (RandomDFS), to see how our heuristics perform against it. RandomDFS has the same general structure as normal DFS, with the exception that when it decides which node to visit next, it chooses randomly among all unvisited child nodes of the node where the search is at that point in time.

### 5.2 Hardware

All experiments are run on a Pentium-4 machine with Windows-XP and 2GB of RAM.

## 5.3 Independent Variables

### The Models

We create a few models of well-known concurrency problems, such as dining-philosophers, barbershop, hyman, customers-workers, etc. These models are in the ranges of 500000 – 1000000 states. To avoid being bias in the selection of our properties to be verified, we randomly choose 40 states in each model and mark these states as errors. Then we run each implementation on all models to see how it performs in finding these errors. In order to measure efficiency in error discovery, we introduce a time bound of five minutes to search each model. This creates a situation where time-wise it is hard for the implementations to explore the whole concrete state space in any of the models. In an industry scenario, this time bound would scale and represent how long we are willing to wait for an answer from the model checker.

## 5.4 Dependent Variables

### Measured Categories

Every time an implementations finds an error, we store: the time it took to find the error, the number of states, the number of predicates, the number of refinements, and counter-example depth. The counter-example depth is the length from the initial state to our error.

## 5.5 Results

In this section we present the results on how each implementation performs on all the models in terms of error-discovery. The tables in this section represent the average results over 12 runs of our experiment. We stop at this number of trials after running

a student’s t-test, which shows that all of our results are statistically significant at the 0.05 confidence level or below.

### 5.5.1 Table Legend

In order to quantify our results, we use a scoring mechanism based on points. We could just report the raw data that we get from the experiment, but due to the size of the data set, we need a way to summarize the data so that results are easier to understand. To this end, we give a point to each win on each measured category. If there are ties, we give a point to all winners. For example, a win on time is given to the implementation that finds a particular error faster than all other implementations. In this case, the winning implementation gets a point. A win on states is given to the implementation that has less states than all other implementations at the moment of error discovery. Similarly, a win on predicates is given to the implementation having less predicates than all other implementations at the moment of error discovery. In terms of depth, a win is given to the implementation that has the shortest path from the initial state to the error at the moment of error discovery.

The result tables show the winner implementations in terms of: time, states, predicates, and error-depth. The data is normalized for ease of reading. In the tables, the implementation that wins on most errors for a particular model, gets a 1; the other implementations get a relative score compared to the winner. For example, looking at Table 5.1, MinOnly on model *SumToNBig* finds the quickest the most errors, so we assign it a 1. The rest of the techniques are assigned a score which is the ratio of how many errors they find the quickest over how many errors MinOnly finds the quickest.

Here is an illustrating example on our scoring mechanism. Assume our only implementations are MinOnly and MaxOverlap. There are 40 errors total and MinOnly finds the quickest 20 of them. Max-Overlap finds the quickest 10 errors, and

Table 5.1: Normalized time results

Models	AMCS	RandDFS	AMCSNoTP	MinOnly	MaxOverlap	AddAll	MinSet
Branches	0.00	0.22	0.00	0.00	1.00	0.00	0.33
SumToNBig	0.00	0.00	0.00	1.00	0.80	0.00	0.00
CustWorkers	0.00	0.00	0.00	1.00	0.36	0.00	0.00
MultiplyToN	0.00	0.11	0.00	1.00	0.44	0.00	0.00
Hyman	0.00	0.19	0.00	1.00	0.00	0.00	0.00
DiningPhil	0.00	1.00	0.00	0.25	0.00	0.00	0.00
Barbershop	0.00	0.00	0.00	1.00	0.15	0.00	0.00

Table 5.2: Per model percentage of errors found over total errors

Models	AMCS	RandDFS	AMCSNoTP	MinOnly	MaxOverlap	AddAll	MinSet
Branches	0.02	0.30	0.05	0.32	0.35	0.02	0.15
SumToNBig	0.20	0.00	0.20	0.45	0.30	0.20	0.20
CustWorkers	0.15	0.32	0.00	0.48	0.22	0.15	0.15
MultiplyToN	0.18	0.05	0.20	0.70	0.35	0.18	0.15
Hyman	0.00	0.32	0.00	0.48	0.02	0.00	0.15
DiningPhil	0.20	1.00	0.02	1.00	0.72	0.20	0.72
Barbershop	0.10	0.00	0.20	0.58	0.40	0.10	0.38

the remaining 10 errors are not found by either implementation. In this case on our *time* table, MinOnly gets a 1 because it finds the quickest the most errors. MaxOverlap gets a 0.5, which is the ratio of the 10 errors that it finds the quickest over the 20 errors that MinOnly finds the quickest. There is the rare case where both implementations find an error at the same time, but this does not significantly affect the comparison, since both implementations would get a point in this case. The same methodology is used to create all the other tables.

### 5.5.2 Data Interpretation

Looking at the tables in this section, one can see that MinOnly wins on most models over the other techniques in all measured categories. AMCS and its variations never win in terms of time, and in the other categories, on most models they compete with

Table 5.3: Normalized states results

Models	AMCS	RandDFS	AMCSNoTP	MinOnly	MaxOverlap	AddAll	MinSet
Branches	0.00	1.00	0.09	0.09	0.36	0.00	0.00
SumToNBig	0.47	0.00	0.47	1.00	0.71	0.47	0.47
CustWorkers	0.00	0.14	0.00	1.00	0.36	0.21	0.21
MultiplyToN	0.37	0.00	0.42	1.00	0.68	0.37	0.32
Hyman	0.00	0.08	0.00	1.00	0.08	0.00	0.25
DiningPhil	0.30	0.74	0.04	1.00	0.78	0.30	0.78
Barbershop	0.17	0.00	0.42	1.00	0.71	0.17	0.67

Table 5.4: Normalized error depth results

Models	AMCS	RandDFS	AMCSNoTP	MinOnly	MaxOverlap	AddAll	MinSet
Branches	0.07	0.29	0.14	0.64	1.00	0.07	0.43
SumToNBig	0.44	0.00	0.44	1.00	0.67	0.44	0.44
CustWorkers	0.32	0.00	0.00	1.00	0.47	0.32	0.32
MultiplyToN	0.25	0.00	0.29	1.00	0.50	0.25	0.21
Hyman	0.00	0.00	0.00	1.00	0.06	0.00	0.22
DiningPhil	0.20	0.00	0.02	1.00	0.72	0.20	0.72
Barbershop	0.17	0.00	0.35	1.00	0.70	0.17	0.65

each-other for the last places. MaxOverlap is the second best on most models on all categories.

### Time and Errors Found

The *time* results are shown in Table 5.1. Looking at the MinOnly column, one can notice that it has the most 1-s, showing its superiority in error discovery speed. We attribute this superiority to its relatively low computational overhead, which gives it a time and memory advantage during the search. MinOnly loses to MaxOverlap on *Branches* and to RandomDFS on *DiningPhil*. We attribute these loses to the fact that the winner implementations are a better match for these particular models.

Table 5.2 shows the fraction of total errors that each implementation finds on each model. Looking at Table 5.2, we can see that both MinOnly and RandomDFS find all errors on *DiningPhil*; however looking at Table 5.1 we see that RandomDFS

Table 5.5: Normalized predicate results

Models	AMCS	RandDFS	AMCSNoTP	MinOnly	MaxOverlap	AddAll	MinSet
Branches	0.08	0.00	0.08	0.08	1.00	0.08	0.31
SumToNBig	0.57	0.00	0.57	1.00	0.86	0.57	0.57
CustWorkers	0.38	0.00	0.00	1.00	0.56	0.38	0.38
MultiplyToN	0.37	0.00	0.42	1.00	0.74	0.37	0.32
Hyman	0.00	0.00	0.00	1.00	0.06	0.00	0.24
DiningPhil	0.20	0.00	0.02	1.00	0.72	0.20	0.72
Barbershop	0.00	0.00	0.00	1.00	0.00	0.00	0.65

wins on time on four times more errors. Since both techniques find all errors and the errors are randomly spread throughout the whole state space, we can deduce that both implementations come close to exploring the whole state space of *DiningPhil*. The data suggest that in situations where state explosion is not present, RandomDFS may be more effective for certain types of models. So MinOnly does not guarantee to perform better on all models and all situations; however, based on our experiment, it is hard to deny its superiority on these models. This superiority makes it a good choice for real life situations, where in most cases, we do not know the size of the input models or whether we are able to explore their entire state space. Because MinOnly is fast, Table 5.2 shows that it finds the most errors on all models except *Branches*. Finding more errors also gives MinOnly an advantage in terms of *states*, *predicates*, and *error depth*.

The MaxOverlap column also shows interesting results. On *Branches*, MaxOverlap wins over all other models, and on *SumToN*, it performs comparably with MinOnly. It shows relatively poor results on all other models compared with MinOnly, which could be attributed to the fact that it has more computational overhead in computing predicates for the next refinement step. Another reason why MaxOverlap is slower than MinOnly is that it adds only one predicate at a time on each refinement step, thus expanding the abstract state space too slowly. As a consequence, it ends up not finding many of the errors, as shown in Table 5.2.

MinSet shows poor performance in both Table 5.1 and Table 5.2. We attribute this poor performance to the computational cost that MinSet incurs when computing its minimal set of predicates that splits all abstract states. In addition, we find that MinSet incurs a significant memory cost to build the table for the min-cover problem.

RandomDFS also shows good performance in Table 5.2. It comes first on *DiningPhil* and second on *CustWorkers* and *Hyman*. RandomDFS is fast because it does not have the computational overhead of abstracting states and finding predicates in refinement steps.

In terms of memory, we focus mostly on MaxOverlap, MinOnly, and RandomDFS since they are the top three performers on percentage of errors found. We are safe to make this simplification because the rest of the implementations are very slow in finding errors. It is to be noted that MaxOverlap is the expected winner in terms of states and predicates due to the fact that it has the most conservative predicate adding technique.

## States and Predicates

Looking at the MinOnly column in Table 5.3, it is evident that it has the most 1-s; however, looking at the percentage of errors found, MaxOverlap shows promising results in terms of states. On *Branches* MaxOverlap and MinOnly find roughly the same percentage of errors. Now moving to Table 5.3, MaxOverlap performs four times better on states than MinOnly. On *MultiplyToN* MinOnly finds twice as many errors as MaxOverlap, so half of the errors that MinOnly wins on *states* are by default because MaxOverlap does not find them. Looking at Table 5.3, MaxOverlap wins on states on 68% as many errors as MinOnly. So if we do not count the errors where MinOnly wins by default, MaxOverlap has a higher score on states. Following a similar logic, one can see that MaxOverlap performs comparably with MinOnly on

*SumToN*, *DiningPhil*, and *Barbershop*. This trend of MaxOverlap performing well in terms of states can be attributed to its conservative predicate-adding technique.

MinSet also performs well in terms of *states*. This result is because MinSet also has a conservative technique for adding predicates on each refinement step. Considering the relatively low percentage of errors that it finds, we are induced to think that MinSet is a promising technique assuming that a less costly method for computing predicates could be found.

The results on predicates are shown in Table 5.5. The data in this table reflects the pattern that we see on *states*, with MaxOverlap performing better than MinOnly if we do not consider the errors where MinOnly wins by default. MinSet also shows good results in terms of predicates because it also has a conservative predicate-generation technique.

## Depth

The result for error depth are shown in Table 5.4. Looking also at Table 5.2, there is no clear winner among the under-approximating techniques, which is explained by the fact that they all use a similar search order. An interesting result is shown by RandomDFS which gets 0.00 on all models, indicating that if we are looking for short counter-example traces, RandomDFS is not ideal because it tends to go deep and find the longer paths to the error.

### 5.5.3 Threats to Validity

A threat to the validity of our experiment is the diversity of our models. It is true that most of our models are well-known concurrency problems in computer science, but our study lacks comprehensiveness due to their limited number. Theoretically, we cannot exclude the possibility that different models might provide different results, but this threat is very hard to avoid in any empirical analysis. Another threat to

validity is the five minute time-bound that we allow for searching each model. It is hard to determine how this time bound would scale in industry, when searching models in the orders of billions of states.

# Chapter 6

## Conclusions and Future Work

We present three new under-approximation techniques that outperform AMCS in error discovery in terms of: time, states, predicates, and error depth. In addition, all our approaches guarantee termination. Out of these techniques, MinOnly finds errors in significantly less *time* on five out of seven models, due to its low computational overhead. Because it operates fast, Min-Only also finds the largest number of errors on six out of seven models.

Max-Overlap shows promising signs in keeping the abstract state space small and might be a good choice for applications where space is very critical. However, its slow approach in adding predicates is a serious drawback. MinSet on the other hand, shows serious flaws due to its large computational overhead. However, the idea of finding a minimum set of predicates that splits all members of  $Elibible(\Phi)$ , might show better results if a different algorithm is used for its implementation.

A future venue is to look for a better bisimulation-detecting technique. AMCS detects bisimulation through precision and weakest precondition, which are the roots of its non-terminating drawback. Another promising idea is in finding a way to add predicates so that a bisimulation is created before exploring the whole concrete state space. However, until we have faster theorem provers, techniques similar to the ones that we propose will have the advantage.

## Bibliography

- [1] T. Ball. A theory of predicate-complete test coverage and generation. In *Microsoft Research Technical Report*, pages 28+, 2004.
- [2] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstraction for model checking c programs. In *Lecture Notes in Computer Science*, volume 2031, pages 268+, 2001.
- [3] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Proceedings of the 25th International Conference on Software Engineering*, pages 385–395, May 2003.
- [4] P. Cousot and R. Cousot. Abstract interpretation frameworks. In *Logic and Computation*, volume 4, pages 511–547, 1992.
- [5] D. Dams and K. S. Namjoshi. The existence of finite abstractions for branching time model checking. In *Proceedings of the 19th Conference on Logic in Computer Science(LICS)*, pages 335–344, 2004.
- [6] L. de Alfaro, P. Godefroid, and R. Jagadeesan. Three-valued abstractions of games: Uncertainty, but with precision. In *Proceedings of the 19th Conference on Logic in Computer Science(LICS)*, pages 170–179, 2004.
- [7] P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based model checking using modal transition systems. In *Lecture Notes in Computer Science*, volume 2154, pages 426+, 2001.
- [8] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV 97)*, volume 1254, pages 72–83. Springer Verlag, 1997.
- [9] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: A new algorithm for property checking. In *Proceedings of the 14th Annual Symposium on Foundations of Software Engineering (FSE)*, pages 117–127, 2006.

- [10] R. Iosif. Symmetry reduction criteria for software model checking. In *Proceedings of 9th International SPIN Workshop*, volume 2318 of *LNCS*, pages 22–41. Springer, April 2002.
- [11] M. Lewis. Interrupt handlers in dead variable analysis. Technical Report SMC-BYU-0105, Software Model Checking Laboratory, Brigham Young University, October 2005.
- [12] K. L. McMillan. *Symbolic Model Checking: An approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, 1992.
- [13] C. Pasareanu, R. Pelanek, and W. Visser. Concrete model checking with abstract matching and refinement. In *Proceedings of the 17th Conference on Computer Aided Verification (CAV)*, pages 52–66, 2005.
- [14] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 267–276, New York, NY, USA, 2003. ACM Press.
- [15] S. Shoham and O. Grumberg. Monotonic abstraction-refinement for CTL. In *Proceedings of the 10th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988, pages 546–560, 2004.
- [16] W. Visser, S. Park, and J. Penix. Applying predicate abstraction to model check object-oriented programs. In *Proceedings of the 3rd ACM SIGSOFT Conference*, pages 364–377, 2000.