

ANALYZING INFINITE-STATE PROGRAMS WITH ABSTRACT  
INTERPRETATION

by

Wei Ding

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

Copyright © 1999 by Wei Ding



# Analyzing Infinite-state Programs with Abstract Interpretation

Wei Ding

Master of Science

Graduate Department of Computer Science

University of Toronto

1999

## Abstract

Recent years have seen an increasing interest in computer-supported techniques for analyzing correctness of software artifacts. Two main approaches used for improving the quality of systems are *testing* and *verification*. We are interested in applying *abstract interpretation* to testing and model checking. Abstract interpretation is a way of symbolically executing programs using abstract instead of concrete domain. In this thesis, we propose an abstract-interpretation based symbolic tester and an abstract model checker to analyze infinite-state programs. In our approach, information specified in the *abstract testcases* are propagated and the abstraction of behaviors of the program written in a subset of C under analysis is computed automatically using abstract interpretation. Then we model check the abstracted system: if the model checker yields *Yes*, the property holds in the original system, and if the model checker yields *No*, the property does not hold. Our tool guarantees convergence and is demonstrated to be sound with respect to both *Yes* and *No* answers. Abstract model checking in conjunction with environmental assumptions enables us to improve the quality and possibly accelerate the convergence of our analysis. Finally, we check the effectiveness of the tool on an implementation of a reactive model — the Safety-Injection System. Our tool is an initial attempt for building a framework for step-wise automatic verification and scalable program analysis technique.

# Acknowledgements

First of all, I would like to thank my supervisor, Marsha Chechik, for her excellent guidance. Her breadth of knowledge and enthusiasm have been a constant inspiration for me. This thesis would not have been possible without her unfailing patience and encouragement.

I would like to express my gratitude to E.C.R. Hehner, who was my second reviewer. His valuable suggestions have helped to improve this thesis to a great extent.

Special thanks go to Mark Pichora, Albert Yu and Daniel House for many interesting discussions. Tefvik Bultan has helped me a lot in understanding his work more thoroughly.

I would like to acknowledge the financial support provided by the Natural Sciences and Engineering Research Council of Canada.

Finally, thanks to my parents, for their endless love and selfishless support. I would not be where I am without you.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Related Work . . . . .	4
1.3	Thesis Overview . . . . .	6
<b>2</b>	<b>Building an Abstract Interpreter</b>	<b>9</b>
2.1	Background . . . . .	9
2.2	Why Build Our Own Abstract Interpreter? . . . . .	12
2.3	The Input Language . . . . .	13
2.4	Construction of an Abstract Finite-state Program . . . . .	13
2.5	Implementation . . . . .	16
2.5.1	Data Structure . . . . .	16
2.5.2	Algorithms . . . . .	17
2.5.3	Operations On Sets of Intervals . . . . .	23
2.6	Correctness . . . . .	24
2.7	Performance . . . . .	26
2.8	Testing . . . . .	26
2.9	Summary . . . . .	29
<b>3</b>	<b>Light-weight Model Checking</b>	<b>35</b>
3.1	Background . . . . .	35
3.2	Construction of a Labeled Transition System . . . . .	38

3.3	Model Checking Algorithm . . . . .	40
3.4	Performance . . . . .	43
3.5	Combining Model Checking with Environmental Assumptions . . . . .	43
3.6	Summary . . . . .	45
<b>4</b>	<b>Case Study</b>	<b>47</b>
4.1	The Application . . . . .	47
4.2	Verification . . . . .	50
4.3	Assumptions on Environment . . . . .	51
4.4	Summary . . . . .	52
<b>5</b>	<b>Conclusion</b>	<b>57</b>
5.1	Summary . . . . .	57
5.2	Contributions . . . . .	58
5.3	Limitations . . . . .	60
5.4	Future Work . . . . .	61
<b>A</b>	<b>Proofs</b>	<b>63</b>
<b>B</b>	<b>Grammar Of Input Language</b>	<b>75</b>
<b>C</b>	<b>Grammar Of Property Specification Language</b>	<b>81</b>
<b>D</b>	<b>Grammar Of TestCase Specification Language</b>	<b>85</b>
	<b>Bibliography</b>	<b>86</b>

# List of Tables

2.1	Execution of the While Loop of the Program in Figure 2.2. . . . .	21
4.1	Mode Transition Table. . . . .	48
4.2	Event Table. . . . .	49
4.3	Condition Table. . . . .	49
4.4	Running Time for Verifying Different Properties. . . . .	51



# List of Figures

1.1	Architecture of the Abstract Model Checker. . . . .	7
2.1	An Example of Cousot's Abstract Interpreter. . . . .	12
2.2	A Program Fragment. . . . .	13
2.3	A Program and its AST Representation. . . . .	15
2.4	An Example of Data Structure for Each Variable. . . . .	17
2.5	Algorithm for Analyzing Conditional Statements. . . . .	18
2.6	Control-Flow Graph of the Program in Figure 2.2. . . . .	19
2.7	An Example of Using Omega Calculator. . . . .	20
2.8	Procedure Widening. . . . .	21
2.9	Algorithm for Analyzing Loops. . . . .	22
2.10	<i>Union, Difference, Widening and Merge</i> on Sets of Intervals. . . . .	23
2.11	Arithmetic on Intervals. . . . .	25
2.12	Comparison on Intervals. . . . .	30
2.13	Arithmetic on Sets of Intervals. . . . .	31
2.14	Comparison on Sets of Intervals. . . . .	32
2.15	A Testcases File. . . . .	33
2.16	A Program Fragment. . . . .	33
3.1	A Program Fragment (same as in Figure 2.2). . . . .	40
3.2	Kripke Structure $K^\alpha$ Built from the Program Fragment in Figure 3.1. . .	40
3.3	Model Checking Algorithm. . . . .	41
3.4	A Program Fragment. . . . .	44

4.1	The Environmental Assumptions for Safety-Injection System. . . . .	52
4.2	Safety-Injection Implementation. . . . .	53
4.2	Safety-Injection Implementation (cont'd). . . . .	54
4.2	Safety-Injection Implementation (cont'd). . . . .	55
4.2	Safety-Injection Implementation (cont'd). . . . .	56
5.1	Framework for Automatic Verification. . . . .	59

# Chapter 1

## Introduction

### 1.1 Motivation

Recent years have seen an increasing interest in computer-supported techniques for analyzing correctness of software artifacts. Two main approaches used for improving the quality of systems are *testing* and *verification*.

In testing [5, 35, 62, 55], one usually checks if a given implementation conforms with the intended system behavior by providing a set of inputs to the system. During testing, users experiment with the system by providing a finite number of testcases, and observing how the output of the system changes based on the input. It has been argued that testing specifications at the early stage of software development cycle is important to reduce the cost of software construction [55]. Unfortunately, testing has the drawbacks that the number of testcases needed to cover the system behavior is significant. Completely testing a moderate-size system can require tens or hundreds of thousands or possibly even an infinite number of testcases. Instead, people try to break the input space into a finite number of equivalence classes. However, generating equivalence classes for input is notably difficult.

Common verification techniques can be divided into two broad classes: *theorem proving* and *model checking*. Theorem proving is a technique where logical rules of deduction are used to formally prove that a system satisfies certain desired properties or that one

system specification is equivalent to another. The logical rules are given by a formal system which defines a set of axioms and inference rules, and a theorem prover tries to find a proof of a property from the axioms of the system. Theorem provers can deal directly with infinite state spaces. Notable applications of theorem provers are: the Boyer and Moore theorem prover [10] was used to prove the correctness of many binary machine code programs produced from high-level languages (such as C); the Verity verification tool [57] was widely used in the design of processors like System/390 and the PowerPc; the ACL2 mechanical theorem prover was used to prove the correctness of the floating point square root microcode [65], and errors in the microcode were found. Recently, the number and kinds of theorem provers have increased due to the growing interest in them. The theorem provers have three broad categories according to their degree of automation: the Larch Prover (LP) [34, 39] and ACL2 [60] are user-guided automatic deduction tools; Coq [23] and HOL [37] are examples of proof checkers, and they have been used in program verification; PVS [66] and STeP [8] combine interactive proofs with powerful decision procedures and model checking. These systems perform an increasing number of proofs automatically. However, most non-trivial proofs require users guidance, and users are expected to have considerable mathematical skills and training.

Model checkers [20, 4] automatically verify temporal properties of reactive and concurrent systems. Given a system and a property, a model checker builds the reachability graph by exhaustively exploring the state-space of the system. A number of industrial model checkers have been developed, including SPIN [48], SMV [59], and Mur $\phi$  [31]. *Symbolic model checking* [16] has been a successful technique in which large finite-state machines are compactly represented by *binary decision diagrams* (BDDs) [12]. Binary decision diagram is a canonical form for a Boolean formula. The resulting state space is much less than the number of states in the original system. For example, HyTech [3, 2] is a symbolic model checker for encoding real domains using affine constraints on real variables. Although model checking started as a technique for verifying hardware, it has been effectively applied in a variety of software projects. For example, SMV has been used to verify correctness of mode logic in A-7 aircraft [67] and TCAS specifications [17];

SPIN has been applied to the validation of the remote object invocation in CORBA GIOP [52], checking Java programs [40], and many others. Model checking became part of the routine V&V process during the development of Lucent’s new server product [47], and has been applied to reasoning about user interfaces [33] and business processes [51].

Model checking offers a potential of push-button verification. However, this potential is not easily realizable, especially for checking correctness of programs, as opposed to specifications, protocols, or other software artifacts. First of all, model checking is mostly limited to finite-state systems (i.e., every variable in the system should have a finite domain). Several model checkers allow reasoning about infinite-state systems by “executing” all paths of the system up to a certain depth [36, 48]. However, such systems cannot guarantee that the system satisfies the desired property.

To check programs, an analyst has to utilize abstractions, computed either automatically or by hand [47]. And, although it is highly-desirable that properties hold in the abstracted model if and only if they hold in the original model [18], such assurance is difficult to obtain: a different abstraction has to be built for each class of properties under analysis [29]. Verification by abstraction was applied to infinite state systems as shown in [32, 38, 41, 61]. In these systems, verifiers operate on the abstract system, and users have to decide which level of abstraction to use. A method to compute an abstract state space for a given finite/infinite system is needed.

In this thesis, we explore applications of abstract interpretation for analyzing program correctness. In particular, we are interested in applying this technique to testing and model checking. In our approach, we first compute the abstraction of behaviors of the program under analysis using abstract interpretation. This abstraction is not dependent on a choice of properties to verify and is computed automatically, even though the program may not be finite-state.

We then present *abstract interpretation-based testing* which enables formal testing of programs by giving environmental assumptions and applying static analysis. With *abstract testing*, we can have a finite representation of an infinite number of inputs: *abstract testcases*; and we can specify information about the environment to achieve

more precise analysis.

Afterwards, we present the model checking algorithm that works on the abstracted system: if the model checker yields *Yes*, the property holds in the original system, and if the model checker yields *No*, the property does not hold. The properties for which the analysis is inconclusive (*Maybe*) can then be verified using more expensive techniques. We discuss verification of sequential programs against fairly complex properties, involving temporal logic and arithmetic on values of variables, e.g. “*a* is never less than *b*”, “immediately after  $2a + b > 5$ , *c* will be true”.

## 1.2 Related Work

The major limitation of model checking is the state space explosion problem, i.e., when the system has large number of variables, the state space grows exponentially. Abstraction techniques [18, 22, 30, 58] offer a possible solution to the state-explosion problem. Such techniques allow us to abstract the model under analysis to a smaller one, in a property-preserving way, such that if a property holds in the abstracted model, it also holds in the original model. Abstractions are essential for effective model-checking [46], although constructing a good abstraction is often very difficult.

The idea of verification with the presence of abstraction has been explored by several researchers. In particular, Bharadwaj and Heitmeyer [6, 42] analyzed SCR specifications using the SPIN [48] model checker. The size of the concrete state space is reduced by three abstraction methods: eliminating variables which are not relevant to the property being verified (SCR ensures that dependencies between variables form a partial order), removing detailed monitored variables, and replacing input variables by predicates. The last approach makes the verification conservative, allowing false negatives. Properties are specified as logical formulae, known as state invariants, which assert the truth of predicate formulae in all reachable states of the system.

Bultan [14, 15] created a symbolic model checker with three main phases: first, an input program (written in a simple event-action language) is translated into a set of Pres-

burger relations which symbolically encode the program’s underlying transition relation. Afterwards, the program’s state space and transitions are partitioned into equivalence classes to reduce the complexity of verification. Finally, the verification algorithm interacts with the Omega library [54] to solve linear inequalities. At this phase, user interaction might be needed. With a combination of exact-backward, exact-forward, approximate-backward, approximate-forward and reachability analysis, this approach gives both conservative “yes” and “no” answers. However, this procedure is partial, with the convergence dependent on the structure of the program and the formula to be verified.

Clarke [21] presented a conservative model checker. This approach is to symbolically encode the state space and transitions into BDDs [11], and check properties on the abstracted state space. The abstraction is formed by first abstracting the program states using a surjective (onto) function, and then mapping transitions from a concrete system to an abstract system. The resulting state space is finite. The specification language used in this work is CTL\* which combines both branching-time and linear-time operators. Dingel and Filkorn [32] extended this method by combining data abstraction and assumption-commitment-style model checking with theorem proving to check against properties expressed in LTL. This approach is not fully automatic, i.e., the proof may require user intervention to choose an appropriate abstraction method.

Jackson [49] proposed a model checking method to analyze infinite specifications expressed in Z or VDM. His approach defines an abstract state space where each abstract state represents a (possibly infinite) equivalence class of concrete states. This method sometimes fails to prove a property that is true, but does not prove false properties and hence is sound.

Dams [29] demonstrated how to abstract reactive systems so that the abstracted transition systems preserve certain forms of combined safety/liveness properties. The properties are specified using  $L_\mu$  [56], which is a modal  $\mu$ -calculus that can express safety, liveness and fairness properties of real-time systems. This approach provides a method for computing the abstract model directly from a program text. Furthermore,

Pardo [64] showed how to build the abstract and the concrete models of the system and conservatively verify properties expressed in  $\mu$ -calculus on the abstracted model. If the formula is proved false, related states are successively refined, until the given formula is verified or computational resources are exhausted. Kelb [53] applied abstract interpretation to model check full propositional  $\mu$ -calculus over parallel processes. The abstractions are computed compositionally.

Bienmüller’s and Brockmeyer [7] verified an avionics application using abstraction and symbolic model checking. Their technique allows focusing verification on aspects of the model relevant to the property under investigation.

In [28], Cousot extended the idea of abstract interpretation based model checking with a new combination of forward and backward abstract fixpoint computation. The result is more precise compared to the methodology of merely conjuncting forward analysis with backward analysis. Furthermore, for unsound abstractions, it is proposed to use the classical forward and backward abstract interpretation analysis to reduce the concrete state space and verify universal safety properties on-the-fly.

Recently, there has been some progress in applying abstract interpretation to testing. In [25], Cousot briefly introduced the idea of automatically computing approximate invariant assertions for programs, and applying such approximate information about the program behavior in program testing systems. In [9], a mathematical framework called *abstract debugging* was proposed. This framework is a combination of forward and backward propagation, and the least and the greatest fixed point computation. In this framework, programmers can use invariant assertions and intermittent assertions (such as termination) to formally check the validity of a program statically, and debug the program during run-time.

### 1.3 Thesis Overview

We have implemented our *Abstract Model Checker* (AMC) as a 26,000-line C program. Figure 1.1 shows the architecture of our software. The *Abstract Interpreter* (AI) receives

the program under analysis and an optional testcases file and builds the abstract finite-state program  $PG = (W, s_0, R, L_T, L_F)$ . This structure, together with a set of CTL formulas, can become the input to the *Model Checker* which checks each property and returns *Yes* if the formula holds in the program, *No* if the formula does not hold, or *Maybe* if the validity of the formula cannot be established. In the later two cases, the model checker also returns a counter-example.

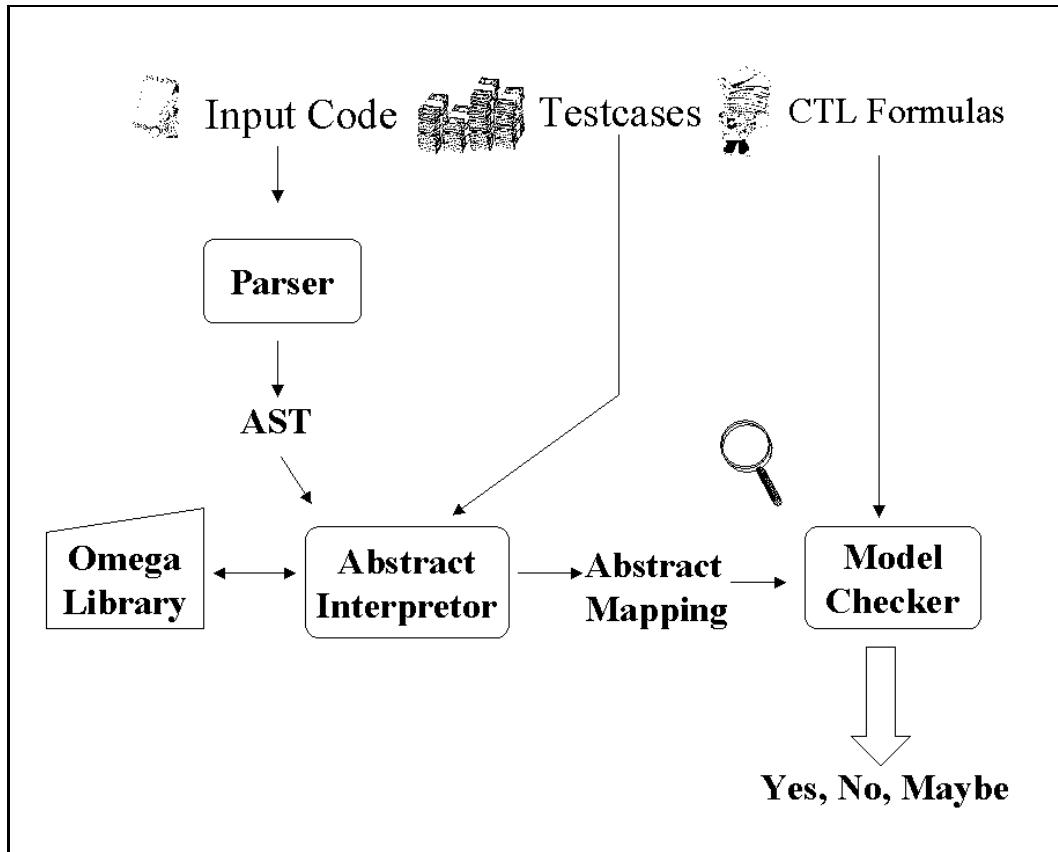


Figure 1.1: Architecture of the Abstract Model Checker.

The rest of this thesis is organized as follows: Chapter 2 describes the process of building the abstract interpreter. Chapter 3 discusses a light-weight model checker and introduces new algorithms for our program verification system. Proofs of correctness of these algorithms are given in Appendix A. Chapter 4 demonstrates the results of using our abstract model checker to analyze the Safety-Injection System. We conclude the thesis with the summary of this work and the outline of the future research directions

in Chapter 5. Finally, Appendix B, Appendix C and Appendix D give the complete grammar of our input language, property specification language and testcase specification language, respectively.

# Chapter 2

## Building an Abstract Interpreter

This chapter describes an abstract interpreter (*AI*) which takes a real program as input and builds an abstract finite-state program, in which every state corresponds to a line number and a list of variables associated with their abstract values.

The rest of this chapter is organized as follows: in Section 2.1, we recall the basic definitions of abstract interpretation. Section 2.2 points out the motivation for building our own abstract interpreter. Section 2.3 gives a description of our input language. Section 2.4 discusses the transformation of the program representation into an abstract finite-state program. Section 2.5 demonstrates some implementation details including the data structure for storing information about variables, algorithms for analyzing conditional statements and loops, and formal definitions of operations on sets of intervals. Proofs of soundness and the computational complexity of our AI are shown in Section 2.6 and Section 2.7, respectively. Finally, we present an application of abstract interpretation to testing in Section 2.8, and summarize this chapter in Section 2.9.

### 2.1 Background

Abstract interpretation [27, 29] is a way of symbolically executing programs using abstract instead of concrete domain. Familiar data-flow analysis algorithms, e.g., constant propagation or live variables, are examples of abstract interpretation. Let  $D_c$  and

$D_a$  be the concrete and the abstract domain, respectively. The *abstraction* function  $\alpha : 2^{D_c} \rightarrow D_a$  maps a set of concrete values into an abstract value.  $\alpha$  has an inverse, the *concretization* function  $\gamma : D_a \rightarrow 2^{D_c}$ . An abstraction is *valid* if the pair of functions ( $\alpha$ ,  $\gamma$ ) forms a Galois connection:

$$\begin{aligned} \forall s \in 2^{D_c}, \quad s \subseteq \gamma(\alpha(s)) \\ \forall t \in D_a, \quad t = \alpha(\gamma(t)) \end{aligned} .$$

For example, we can perform a “sign analysis” by replacing a set of integers ( $D_c = \mathbb{Z}$ ) by their signs ( $(-)$ ,  $(+)$ , or  $(0)$ ). Here,  $\alpha(\{17\}) = (+)$ , and  $\gamma((+)) = \mathbb{Z}^+$ . We can execute the program on the abstract values. For example,

$$(\{-1345\} \times \{17\}) \xrightarrow{\alpha} -(+) \times (+) = (-) \times (+) = (-).$$

However, the abstract values cannot always be determined exactly. Consider the following example:

$$(\{-1345\} \times \{17\} + \{22\}) \xrightarrow{\alpha} (-) + (+) = \begin{pmatrix} 0 \\ + \end{pmatrix}.$$

$\begin{pmatrix} 0 \\ + \end{pmatrix}$  can be represented as a set  $\{(-), (+), (0)\}$  with the interpretation that the result can be *any* of these values. When the abstract domain is finite, the abstract interpreter acts as a data-flow analyzer. However, we may also want to use abstract interpretation to reason about infinite-domain variables. In order to achieve tractability, we need to ensure that the abstraction has the following properties:

1. we have a finite representation of the infinite set of values. One way is to abstract from a set to an interval by taking the minimum (maximum) value of the set as the left (right) bound of the interval. For example,
  - $\alpha(\{-1, 5, 3\}) = [-1, 5]$
  - $\alpha(\{0.5, 1.3, 23\}) = [0.5, 23]$
2. we ensure convergence in a finite number of steps. With a finite-domain abstraction, convergence is guaranteed. To achieve convergence for the infinite-domain

abstraction, [27] introduced an abstract binary operator *widening*, denoted as  $\bar{\nabla}$ , which represents a “jump”. For all abstract values  $i_0$  and  $i_1$ ,  $i_0 \cup i_1 \subseteq i_0 \bar{\nabla} i_1$ . [27] defined widening as follows:

$$[a_1, b_1] \bar{\nabla} [a_2, b_2] =$$

$$\left[ \begin{array}{l} \text{if } a_2 < a_1 \text{ then } -\infty \text{ else } a_1 \text{ fi,} \\ \text{if } b_2 > b_1 \text{ then } +\infty \text{ else } b_1 \text{ fi} \end{array} \right]$$

For example,

- $[-1.5, 10] \bar{\nabla} [2, 44] = [-1.5, +\infty]$
- $[22, -0.1] \bar{\nabla} [-10, -0.4] = [-\infty, -0.1]$

Figure 2.1 gives an example of analyzing a short program using Cousot’s Abstract Interpreter [26]. The program body contains an assignment followed by a while loop: `x:=0; while x>0 do x:=x+1 od`. Here, we define an *abstract context* as a set of pairs  $(v, d_\alpha)$  in which  $d_\alpha$  is the abstract value of a variable  $v$ . An *input (output) abstract context* at a program point is the abstract context before (after) the execution of this point. Consider an example in Figure 2.1. Here, we begin a line with a line number to indicate a program statement. In this case, the preceding line includes the input context of that statement, and the succeeding line includes the output context of that statement. In the initial state, every variable is undefined, denoted as  $\_0\_ (\mathbf{x} : \_0\_)$ . A variable’s value is *undefined* means that this variable has only been declared but not assigned any values. When the execution reaches line 1, the input abstract context is  $(\mathbf{x} : \_0\_)$  ( $\mathbf{x}$  is undefined), and the output abstract context is  $(\mathbf{x} : [0, 0])$  ( $\mathbf{x}$ ’s value is 0). When the input context  $C$  encounters a test node, it results in two output contexts  $C_T$  and  $C_F$  associated with the true and false edges, respectively. Hence the input context  $C = (\mathbf{x}, [0, 0])$  results in  $C_T = (\mathbf{x}, \_|\_)$  and  $C_F = (\mathbf{x}, [0, 0])$  for the test node `x>0` ( $\_|\_$  represents the empty set). Therefore, in the input and the output contexts of line 3,  $\mathbf{x}$ ’s value is an empty set. The loop terminates immediately. In the final output context of this program, we have  $(\mathbf{x}, [0, 0])$ .

```

x:=0; while x>0 do x:=x+1 od
{ x:_0_ }
1:   x := 0;
{ x:[0,0] }
2:   while (0 < x) do
           { x:_|_ }
           x := (x + 1)
3:       { x:_|_ }
4:   od { ((x < 0) | (x = 0)) }
9: { x:[0,0] }

```

---

Figure 2.1: An Example of Cousot’s Abstract Interpreter.

## 2.2 Why Build Our Own Abstract Interpreter?

Cousot developed a static analyzer for an imperative sequential programming language. The language includes `boolean` and `integer` types; unary operators `+` and `-`; binary operators `+`, `-`, `*`, `/` and `mod`; assignment; conditional statement `if B then A else C fi`; iteration `while B do S od`. It is a simple terminating language, and it does not contain any input/output constructs. The program starts with a given input context and, upon termination, generates an output context. The goal of our work, on the other hand, is to enable reasoning about reactive systems. Such systems are harder to reason about because they interact with the environment continuously. Hence, Cousot’s static analyzer is not sufficient for our purpose, and we were motivated to develop our own abstract interpreter. Our software is implemented in C, as opposed to Objective CAML, like Cousot’s, and therefore is significantly faster. However, Cousot’s interpreter is much more general than ours. It can be instantiated for different abstractions, whereas we only have one built-in abstraction level where a set of values is abstracted into an interval representing a potentially infinite number of values.

```

1:  int b;
2:  int xy;
3:  int main ( ) {
4:      int a;
5:      int c;
6:      b = 13;
7:      c = 2;
8:      xy = -20;
9:      while ( 1 ) {
10:         xy = xy + 4;
11:         if (xy == 0)
12:             b = 5;
13:         else
14:             b = b * c;
15:         if ((a != 0)&&(a >= -3))
16:             if ((a != 2)&&(a != 4)&&(a !=7))
17:                 if (a != -2)
18:                     c = 2;
19:         print('xy is ', xy);
20:         print('b is ', b);
21:     }
22: }

```

---

Figure 2.2: A Program Fragment.

## 2.3 The Input Language

Our input language, called C-, is a sequential language with the syntax similar to C. The language includes the following constructs: `boolean` and `integer` types; conditional control structures (`if`, `else`); loops (`while`); input and output (`print`, `fprint`, `scan`, `fscan`); assignments; functions and procedures. Dynamic features such as recursion or pointers are not provided in the language. It also does not allow any user-defined (compound) data structures. A complete grammar of the language is available in Appendix B.

Consider the program given in Figure 2.2. The only difference between our language and a subset of C is the input/output commands (lines 19 and 20 of Figure 2.2). These use call by reference and are similar to I/O statements in Pascal.

## 2.4 Construction of an Abstract Finite-state Program

Here, we describe the transformation of the program representation into an abstract finite-state program. We start with a (infinite-state) program  $PG = (W, s_0, R, L_T, L_F)$ , where  $W$  is a (infinite) set of states,  $s_0 \in W$  is the initial state,  $R \subseteq W \times W$  is the total

accessibility relation, and  $L_T$  and  $L_F$  are *truth* and *falsity* labeling functions, mapping each state to the set of propositions that are *True* and *False*, respectively, in this state ( $L_T, L_F : W \rightarrow 2^P$ ).

Our goal is to construct an abstract finite-state program, in which every state corresponds to a line number in the program and a list of accessible variables associated with their abstract values. In order to do that, we define a set of variables  $V$  and let  $V_w \subseteq V$  be the set of variables which are accessible in the scope associated with a state  $w \in W$ . Each state  $w$  in the program is an  $n + 1$ -tuple,  $w = (ln, (v_1, d_1), (v_2, d_2), \dots, (v_n, d_n))$ , in which  $ln$  corresponds to the line number of the state in the program, and  $\forall i, 1 \leq i \leq n, v_i \in V_w, d_i \in 2^{D_i}$ ; where  $D_i$  is the concrete domain of  $v_i$ .

We start by parsing the program and building an *Abstract Syntax Tree* (AST). The Abstract Syntax Tree is an intermediate representation for the structure of the program under interpretation. An example of AST is given in Figure 2.3. Next, we propagate information about all variables (global and local) in the current scope throughout the AST, until we reach a fixpoint. Abstractions are formed by giving surjections  $\alpha = (\alpha_1, \alpha_2, \dots, \alpha_n)$  which map a concrete state  $w$  onto an abstract state  $w^\alpha$  ( $w^\alpha = \alpha(w)$ ) by mapping each  $d_i \in 2^{D_i}$  onto an abstract value  $D^{\alpha_i}$ . The result is an abstract state space  $W^\alpha$ , in which each  $w^\alpha \in W^\alpha$  is an  $n + 1$ -tuple  $w^\alpha = (ln, (v_1, D^{\alpha_1}), (v_2, D^{\alpha_2}), \dots, (v_n, D^{\alpha_n}))$ . Notice that line numbers and the set of variables are the same in the concrete and the abstract state space.  $\alpha$  is chosen so that  $W^\alpha$  is finite, and an abstract state  $w^\alpha$  can represent one or more or even an infinite number of concrete states due to the abstraction (this process is described in the following section). Moreover, there is only one state associated with each line of code. Let  $R^\alpha$  be a transition relation over the abstract state space,  $R^\alpha \subseteq W^\alpha \times W^\alpha$ .  $R^\alpha$  is constructed as follows:  $(s^\alpha, t^\alpha) \in R^\alpha$  iff  $\exists s, t$  s.t.  $s^\alpha = \alpha(s) \wedge t^\alpha = \alpha(t) \wedge (s, t) \in R$ . Our labeling functions then become  $L^\alpha_T, L^\alpha_F : W^\alpha \rightarrow 2^P$ . The abstract finite-state program  $PG^\alpha = (W^\alpha, R^\alpha, I^\alpha, P, L^\alpha_T, L^\alpha_F)$  is constructed.

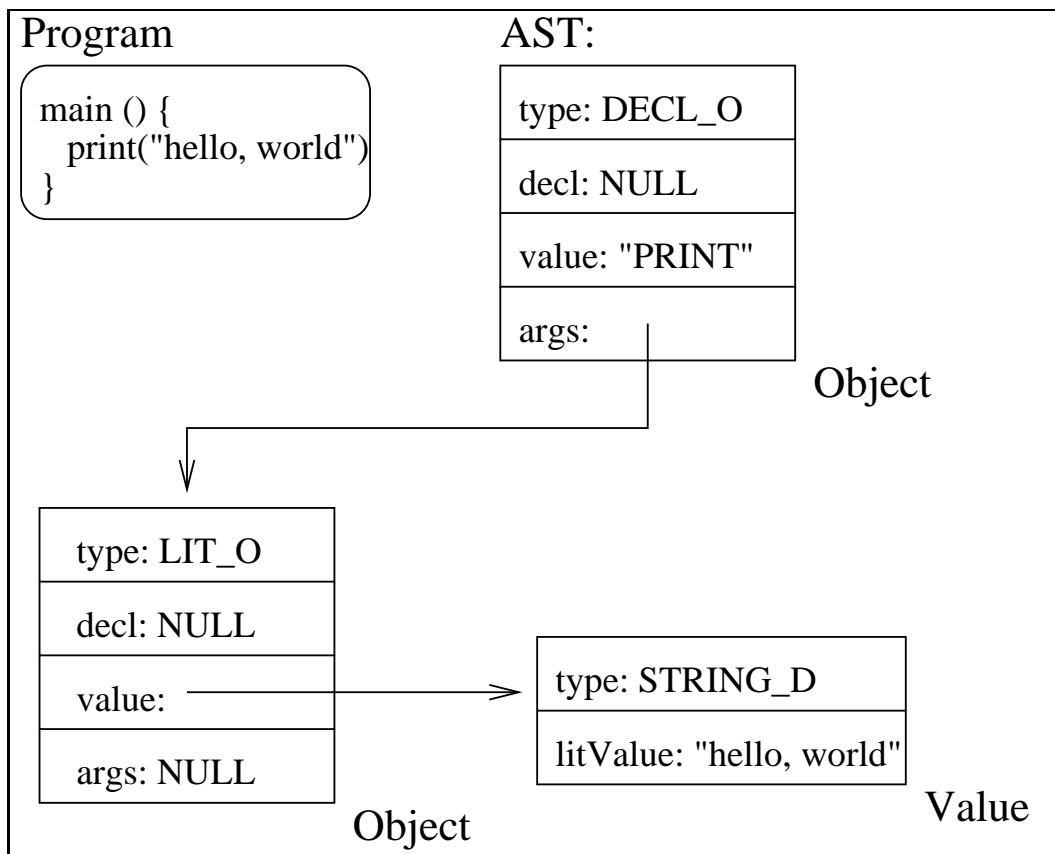


Figure 2.3: A Program and its AST Representation.

## 2.5 Implementation

In Section 2.4, we introduced the theoretical formulation for building the AI. In this section, we explain the detailed design of the system. In particular, the implementation of the data structure, algorithms for analyzing statements and the implementation of operations on sets of intervals are discussed.

### 2.5.1 Data Structure

The AI receives a program and “interprets” it by starting with an *input context* that consists of a set of values that variables have before a program statement, executing the statement, and producing an *output context*. The output context is then stored as part of the state. The abstract values of finite-domain variables (boolean or enumerated types) consist of sets of (concrete) values these variables can attain or *UNDEF* (undefined). However, values of infinite-domain variables such as integers should be abstracted further. In Section 2.1, we have briefly discussed how an abstraction function  $\alpha$  can be applied to a set to get an interval. However, for better precision, we will associate each infinite-domain variable with a (finite) set of intervals, with the following interpretation:

$$\gamma(\{a_1, a_2, \dots, a_n\}) = \gamma(a_1) \cup \gamma(a_2) \cup \dots \cup \gamma(a_n).$$

In the current implementation of AI, each variable is associated with the following data structure: the variable’s *name* and *type*, a boolean field *touch* for indicating whether or not the variable has been assigned a value, an array for storing the abstract values of the variable (an array of integers or of floats or of booleans) consisting of up to 5 intervals, and a link (*next*) to the next variable. If a variable’s *touch* is *False*, then the variable has only one value which is *UNDEF*. Figure 2.4 shows an example of a variable’s data structure. In particular, *xy* is an integer variable, and `touch = 1` indicates that *xy* has been assigned to a set of values  $\{[-3, -1], [1, 1], [3, 7]\}$ , and there are no other variables after *xy* in the linked list.

name	xy				
type	INTEGER_D				
minInt	-1	1	3		
	0	1	2	3	4
maxInt	-1	1	7		
link	X				

---

Figure 2.4: An Example of Data Structure for Each Variable.

### 2.5.2 Algorithms

As discussed in Section 2.3, statements in our input language C- fall into four major categories: (1) conditional statement; (2) iteration statement; (3) input/output statement; (4) assignment statement. Input/output statement and assignment statement in C- have the similar semantics to C. In this subsection, we present the algorithms for abstractly interpreting conditional and iteration statements.

The algorithm used in our AI for analyzing conditional statements is depicted in Figure 2.5. Given an input abstract context  $S_i$ , a conditional expression  $iepr$  and statements to execute when  $iepr$  is *True* or *False* ( $stmt_t$  and  $stmt_f$ , respectively), we either execute  $stmt_t$  ( $stmt_f$ ) based on  $S_i$  and then return the resulting abstract state, or call the Omega calculator to get abstract states that correspond to taking the If and the Else part ( $S_i^t$  and  $S_i^f$ , respectively), execute the statements, and compute the union of the resulting output contexts. The Omega calculator will be discussed in detail shortly after, and we use it for symbolically executing conditional expressions involving intervals.

For example, suppose we are running our AI on the program fragment depicted in Figure 2.2 (Figure 2.6 shows the control-flow graph for this fragment, with each state associated with the program line number). Let the input context before executing state 11 be  $((xy, \{[-20, 52]\}), (a, \{[-5, 8]\}), (b, \{[13, 13]\}), (c, \{[2, 2]\}))$ . The condition  $xy == 0$  evaluates to *Maybe*; therefore, we call the Omega calculator to determine that the value of

```

Procedure EVAL-IF (iepr, stmtt, stmtf, Si)
Evaluate iepr
IF iepr is True
    Execute stmtt starting with Si to get So
    Return So
ELSE IF iepr is False
    Execute stmtf starting with Si to get So
    Return So
ELSE IF iepr is Maybe
    Call Omega calculator to get  $S_i^t, S_i^f = S_i \stackrel{\alpha}{\triangleright} S_i^t$ .
    Execute stmtt starting with  $S_i^t$  to get  $S_o^t$ 
    Execute stmtf starting with  $S_i^f$  to get  $S_o^f$ 
    Return  $S_o^t \stackrel{\alpha}{\cup} S_o^f$ 

```

---

Figure 2.5: Algorithm for Analyzing Conditional Statements.

**xy** in input contexts for states 12 and 14 should be  $\{[0, 0]\}$  and  $\{[-20, -1], [1, 52]\}$ , respectively. The values of **b** in output contexts of these states are  $\{[5, 5]\}$  and  $\{[26, 26]\}$ ; these are unioned to obtain  $\{[5, 25]\}$  in the input context to state 15. The values of **a** after executing state 15 and state 16 are  $\{[-3, -1], [1, 8]\}$  and  $\{[-3, -1], [1, 1], [3, 3], [5, 6], [8, 8]\}$ , respectively. At this point, **a** has reached its limit of five intervals, and further splitting cannot be done; instead, we merge **a**'s intervals to get  $\{[-3, 8]\}$  and proceed with the execution. This introduces a loss of information and precision, but it is strictly conservative. The output value for **a** after state 17 is  $\{[-3, -3], [-1, 8]\}$ .

As shown in Figure 2.5, our software has an interaction with the Omega Calculator [54]. The Omega library is a set of C++ classes for manipulating integer tuple relations and sets, described by Presburger formulas (possibly with the limited use of uninterpreted function symbols). Figure 2.7 illustrates an example of using Omega calculator. Suppose, we are given an input abstract context  $((\mathbf{a}, \{[-\infty, -1], [3, +\infty]\}), (\mathbf{b}, \{[1, 4], [11, +\infty]\}), (\mathbf{c}, \{[2, 7]\}))$ , and we are now about to execute the conditional statement `if (a>b+c)`. As shown in Figure 2.5, it is required that the conditional expression be evaluated and  $S_i^t$  and  $S_i^f$  be generated if the expression is evaluated to *Maybe*. Since

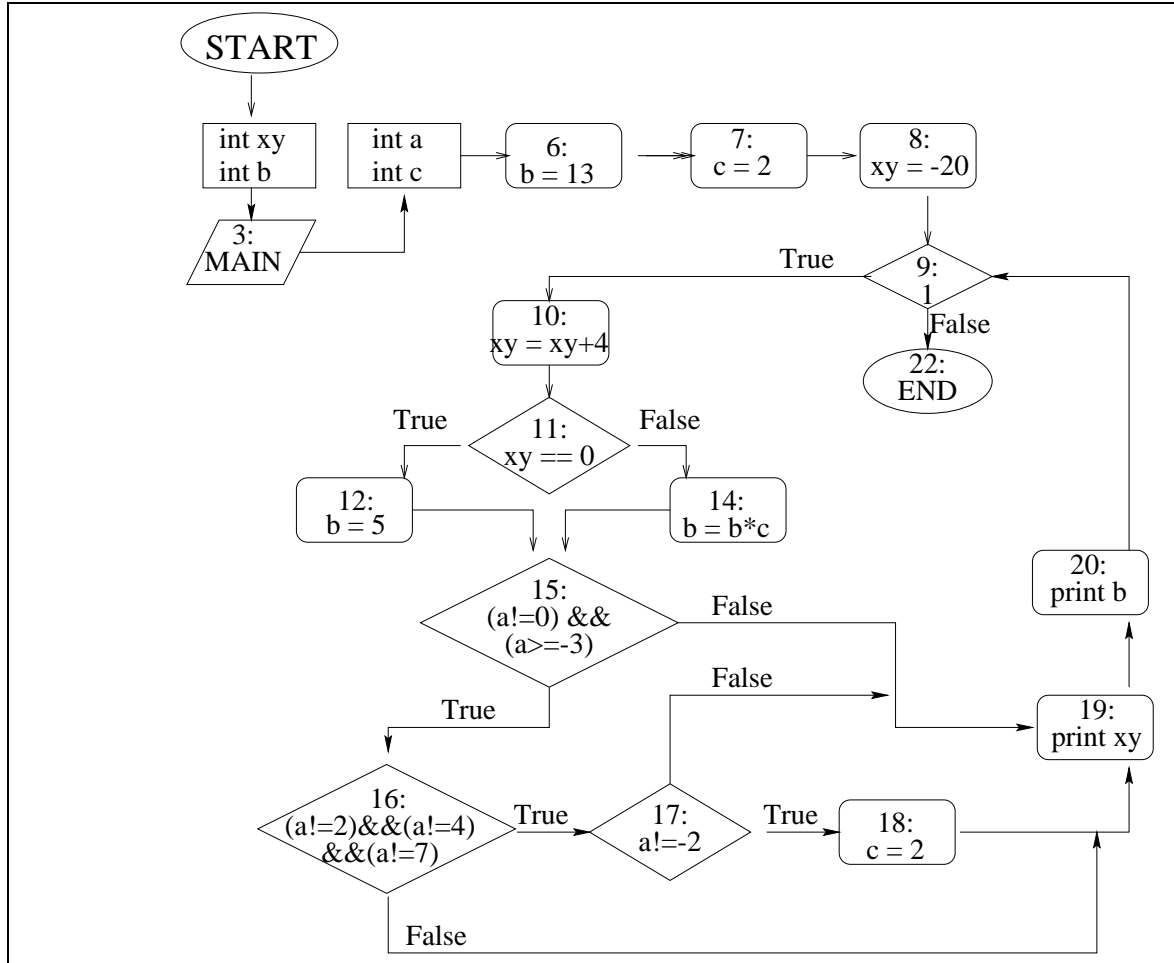


Figure 2.6: Control-Flow Graph of the Program in Figure 2.2.

$\mathbf{b+c} = \{[3, 11], [13, +\infty]\}$ ,  $\mathbf{a > b+c}$  evaluates to *Maybe*. We then send the input abstract context  $S$  and the conditional statement  $T$  to the Omega calculator to get  $S_i^t$ , and  $S_i^f = S_i \overset{\alpha}{\triangleright} S_i^t$ .  $\overset{\alpha}{\triangleright}$  is a *difference* operator defined on sets of intervals (see Section 2.5.3 for details).

Loops are executed until a fixpoint on values of all variables has been achieved. For better precision, we redefine Cousot’s widening function as in Figure 2.8. For example,  $\mathbf{Widen}([2, 5], [4, 10], 19) = [1, 10]$ , whereas  $\mathbf{Widen}([2, 5], [4, 10], 20) = [2, +\infty]$ . In order to ensure that this fixpoint occurs in a finite number of steps, we execute each loop at most 20 times, keeping track of whether values of each variable decrease or increase between iterations. At the end of each iteration, we widen variables that have changed

```

/* Input */
S:={[a,b,c]: (a<=-1 || a>=3) &&
    (1<=b<=4 || b>10) && 2<=c<=7}
T:={[a,b,c]->[a]: a>b+c};
Answer:=S join T;
Answer;
T:={[a,b,c]->[b]: a>b+c};
Answer:=S join T;
Answer;
T:={[a,b,c]->[c]: a>b+c};
Answer:=S join T;
Answer;

/* Output, a line starts with '#' is a comment */
# Omega Calculator [v1.1, Nov 96]:
# S:={[a,b,c]: (a<=-1 || a>=3) &&
    (1<=b<=4 || b > 10) && 2<=c<=7};
#   T:={[a,b,c]->[a]: a>b+c};
#   Answer:=S join T;
#   Answer;

{[a]: 4 <= a}

#   T:={[a,b,c]->[b]: a>b+c};
#   Answer:=S join T;
#   Answer;

{[b]: 1 <= b <= 4} union
{[b]: 11 <= b}

#   T:={[a,b,c]->[c]: a>b+c};
#   Answer:=S join T;
#   Answer;

{[c]: 2 <= c <= 7}

```

---

Figure 2.7: An Example of Using Omega Calculator.

values between the current and the previous iterations. If a fixpoint was not achieved, we widen values of non-converged variables, with the increment and the decrement leading

```

Procedure WIDEN ( $[a_1, b_1]$ ,  $[a_2, b_2]$ ,  $count$ )
IF  $count < 20$ 
  Return  $[\min(a_1, a_2), \max(b_1, b_2)]$ 
ELSE
  IF  $a_2 < a_1$ 
     $a = -\infty$ 
  ELSE
     $a = a_1$ 
  IF  $b_2 > b_1$ 
     $b = +\infty$ 
  ELSE
     $b = b_1$ 
  Return  $[a, b]$ 

```

---

Figure 2.8: Procedure Widening.

to values of  $+\infty$  and  $-\infty$ , respectively. The algorithm used to analyze loops is depicted in Figure 2.9. Given an input abstract context  $S_i$ , a conditional expression  $wexpr$  and statements  $stmt$  to execute when  $wexpr$  is *True*, we initialize a counter  $count$  to 0, execute  $stmt$  based on  $(S_i)_{count}$ , and generate the output context  $(S_o)_{count}$ . Then we widen  $(S_i)_{count}$  with  $(S_o)_{count}$  to get  $(S_i)_{count+1}$ . If  $(S_i)_{count}$  and  $(S_i)_{count+1}$  are equal, the output context of  $count$ th iteration is returned; otherwise, the execution carries on until a fixpoint is reached.

<i>iteration</i>	<b>b</b>	<b>xy</b>
1	{[13, 13]}	{[-20, -20]}
6	{[5, 416]}	{[-20, 0]}
7	{[5, 832]}	{[-20, 4]}
19	{[5, 3407872]}	{[-20, 52]}
21	{[5, $+\infty$ ]}	{[-20, $+\infty$ ]}

Table 2.1: Execution of the While Loop of the Program in Figure 2.2.

Table 2.1 lists several values that variables **b** and **xy** attain in the input context to state 9 as we execute the main **while** loop of the program in Figure 2.2. At the first iteration, these values are {[13, 13]} and {[-20, -20]}, respectively. In the following 19

```

Procedure EVAL-WHILE (wexpr, stmt,  $S_i$ )
  Evaluate wexpr
  int count = 0
   $(S_i)_{count} = S_i$ 
  WHILE wexpr is True
    Execute stmt starting with  $(S_i)_{count}$  to get  $(S_o)_{count}$ 
    Call WIDEN ( $(S_i)_{count}$ ,  $(S_o)_{count}$ , count) to get  $(S_i)_{count+1}$ 
    IF  $(S_i)_{count} = (S_i)_{count+1}$ 
      Return  $(S_i)_{count}$ 
    ELSE
      Evaluate wexpr
      count ++

```

---

Figure 2.9: Algorithm for Analyzing Loops.

iterations, we note that the maximum values  $\mathbf{b}$  and  $\mathbf{xy}$  can attain are increasing, whereas their minimum values stay the same. Thus, the widening which occurs after the 20th iteration changes only the maximum values of these variables.

**Theorem 1** *The Widen function is sound:  $[a_1, b_1] \cup [a_2, b_2] \subseteq \text{Widen}([a_1, b_1], [a_2, b_2], \text{count})$ , where  $0 \leq \text{count} \leq 20$ .*

**Proof:**

Let  $\text{Widen}([a_1, b_1], [a_2, b_2], \text{count}) = [a, b]$ .

The proof is constructed by cases determined by the value of *count*.

Case (1): IF *count* < 20 THEN  $a = \min(a_1, b_1) \wedge b = \max(a_2, b_2)$

$$\Rightarrow ([a_1, b_1] \cup [a_2, b_2]) = \text{Widen}([a_1, b_1], [a_2, b_2], \text{count})$$

Case (2): IF *count* = 20 THEN

$$\text{IF } a_1 \leq a_2 \text{ THEN } a = a_1 \text{ ELSE } a = -\infty$$

$$\text{IF } b_1 \geq b_2 \text{ THEN } b = b_1 \text{ ELSE } b = +\infty$$

$$\Rightarrow a \leq a_1 \wedge a \leq a_2 \wedge b \geq b_1 \wedge b \geq b_2$$

$$\Rightarrow [a_1, b_1] \cup [a_2, b_2] \subseteq [a, b] = \text{Widen}([a_1, b_1], [a_2, b_2], \text{count})$$

Thus  $[a_1, b_1] \cup [a_2, b_2] \subseteq \text{Widen}([a_1, b_1], [a_2, b_2], \text{count})$ , where  $0 \leq \text{count} \leq 20$ .

□

### 2.5.3 Operations On Sets of Intervals

In AI, a variable's value is either a set of intervals or *UNDEF*. We define  $\overset{\alpha}{\sqcup}$ ,  $\overset{\alpha}{\cup}$ ,  $\overset{\alpha}{\triangleright}$ , and  $\overset{\alpha}{\nabla}$  (merge, union, difference and widening on sets of intervals) in Figure 2.10.

Let  $a_i, b_i$  be intervals and assume, without a loss of generality, that  $m \leq n$ :

$$\begin{aligned}
\overset{\alpha}{\sqcup} UNDEF &\equiv UNDEF \\
\overset{\alpha}{\sqcup} \{a\} &\equiv \{a\} \\
\overset{\alpha}{\sqcup} \{a_1, \dots, a_m\} &\equiv \{a_1 \cup a_2 \cup \dots \cup a_m\} \\
\{a\} \overset{\alpha}{\cup} UNDEF &\equiv \{a\} \\
\{a\} \overset{\alpha}{\cup} \{b_1, \dots, b_n\} &\equiv \{a \cup b_1, \dots, a \cup b_n\} \\
\{a_1, \dots, a_m\} \overset{\alpha}{\cup} \{b_1, \dots, b_n\} &\equiv (\overset{\alpha}{\sqcup} \{a_1, \dots, a_m\}) \overset{\alpha}{\cup} \{b_1, \dots, b_n\} \\
\{a\} \overset{\alpha}{\cup} \{b\} &\equiv \{b\} \overset{\alpha}{\cup} \{a\} \\
\{a\} \overset{\alpha}{\triangleright} UNDEF &\equiv \{a\} \\
UNDEF \overset{\alpha}{\triangleright} \{a\} &\equiv UNDEF \\
\{a\} \overset{\alpha}{\triangleright} \{b_1, \dots, b_n\} &\equiv \{a \triangleright b_1, \dots, a \triangleright b_n\} \\
\{a_1, \dots, a_m\} \overset{\alpha}{\triangleright} \{b_1, \dots, b_n\} &\equiv (\overset{\alpha}{\sqcup} \{a_1, \dots, a_m\}) \overset{\alpha}{\triangleright} \{b_1, \dots, b_n\} \\
UNDEF \overset{\alpha}{\nabla} \{a\} &\equiv \{a\} \\
\{a\} \overset{\alpha}{\nabla} \{b_1, \dots, b_n\} &\equiv \{a \nabla b_1, \dots, a \nabla b_n\} \\
\{a_1, \dots, a_m\} \overset{\alpha}{\nabla} \{b_1, \dots, b_n\} &\equiv (\overset{\alpha}{\sqcup} \{a_1, \dots, a_m\}) \overset{\alpha}{\nabla} \{b_1, \dots, b_n\} \\
\{a\} \overset{\alpha}{\nabla} \{b\} &\equiv \{b\} \overset{\alpha}{\nabla} \{a\}
\end{aligned}$$


---

Figure 2.10: *Union, Difference, Widening and Merge* on Sets of Intervals.

When we encounter two sets, each containing more than one interval, we first union elements of the set with less intervals (in this case,  $\{a_1, \dots, a_m\}$ ) into one interval, and then union the result with each interval of the other set.  $\overset{\alpha}{\triangleright}$  (difference) and  $\overset{\alpha}{\nabla}$  (widening) are similar. Interval operations *union* and *difference* have their usual meaning, and our *widening* on intervals was defined in Section 2.5.2. Arithmetic and comparison on

intervals are defined formally in Figure 2.11 and Figure 2.12 respectively. For example, addition on two intervals is defined as taking the addition of left (right) bounds of the two intervals as the left (right) bound of the resulting interval. Figure 2.13 and Figure 2.14 define arithmetics and comparisons on sets of intervals. They are defined in a similar fashion to *union* on sets of intervals.

## 2.6 Correctness

In this section, we demonstrate the correctness of AI, i.e., we show that our abstract analysis satisfies the definition of a Galois connection: given a set of values  $s$ , the concretization of an abstraction of  $s$  contains no less information than  $s$  :

$$\begin{aligned} \forall s \in 2^{D_c}, \quad s &\subseteq \gamma(\alpha(s)) \\ \forall t \in D_a, \quad t &= \alpha(\gamma(t)) \end{aligned}$$

In order to prove that, we need to show that *widening* on sets of intervals is sound: for all sets of intervals  $\{a_1, a_2, \dots, a_m\} \overset{\alpha}{\cup} \{b_1, b_2, \dots, b_n\} \overset{\alpha}{\subseteq} \{a_1, a_2, \dots, a_m\} \overset{\alpha}{\nabla} \{b_1, b_2, \dots, b_n\}$ . We have proved Theorem 1 which asserts that our Widen function is sound. As depicted in Figure 2.10, we define  $\overset{\alpha}{\cup}$  and  $\overset{\alpha}{\nabla}$  using *union* and *widening* on intervals. Therefore, it is possible to say that if widening on intervals is sound, widening on sets of intervals is also sound.

Each statement has an input context and an output context. In order to prove that the analysis of a statement is sound, we need to show that the  $j$ th input context of the statement is a subset of its  $j + 1$ th input context ( $j$  is a natural number).

**Theorem 2** *Our analysis of a statement is sound: for any input context  $S_i$  of a statement,  $(S_i)_j \overset{\alpha}{\subseteq} (S_i)_{j+1}$ , where  $0 \leq j < +\infty$  .*

**Proof:**

The proof is constructed by cases:

We assume that  $-\infty < c < +\infty$ , i.e.,  $c$  is any real number except minus and plus infinity. Let us refer  $a_1, a_2, b_1$  and  $b_2$  as  $c$ , if they are neither  $+\infty$  nor  $-\infty$ .

$[a_1, b_1] + [a_2, b_2] \equiv [a_1 + a_2, b_1 + b_2]$ , where

$$\begin{aligned} -\infty + c &\equiv -\infty \\ +\infty + c &\equiv +\infty \\ -\infty + +\infty &\equiv \text{ERROR} \end{aligned}$$

$[a_1, b_1] - [a_2, b_2] \equiv [a_1 - b_2, b_1 - a_2]$ , where

$$\begin{aligned} -\infty - c &\equiv -\infty \\ +\infty - c &\equiv +\infty \\ +\infty - -\infty &\equiv +\infty \\ -\infty - +\infty &\equiv -\infty \end{aligned}$$

$[a_1, b_1] \times [a_2, b_2] \equiv [\min(a_1 \times b_1, a_1 \times b_2, a_2 \times b_1, a_2 \times b_2), \max(a_1 \times b_1, a_1 \times b_2, a_2 \times b_1, a_2 \times b_2)]$ , where

$$\begin{aligned} -\infty \times c &\equiv \text{if } c \text{ is } 0 \text{ then } 0 \text{ if } c < 0 \text{ then } +\infty \text{ else } -\infty \text{ fi} \\ +\infty \times c &\equiv \text{if } c \text{ is } 0 \text{ then } 0 \text{ if } c < 0 \text{ then } -\infty \text{ else } +\infty \text{ fi} \\ -\infty \times +\infty &\equiv \text{ERROR} \end{aligned}$$

$[a_1, b_1] \div [a_2, b_2] \equiv$

if  $0 \in [a_2, b_2]$  then ERROR

else  $[\min(a_1 \div b_1, a_1 \div b_2, a_2 \div b_1, a_2 \div b_2), \max(a_1 \div b_1, a_1 \div b_2, a_2 \div b_1, a_2 \div b_2)]$  fi, where

$$\begin{aligned} -\infty \div c &\equiv \\ &\quad \text{if } c \text{ is } 0 \text{ then ERROR if } c < 0 \text{ then } +\infty \text{ else } -\infty \text{ fi} \\ +\infty \div c &\equiv \\ &\quad \text{if } c \text{ is } 0 \text{ then ERROR if } c < 0 \text{ then } -\infty \text{ else } +\infty \text{ fi} \\ -\infty \div +\infty &= \text{ERROR} \\ +\infty \div -\infty &= \text{ERROR} \end{aligned}$$

$[a_1, b_1] \bmod [a_2, b_2] \equiv$  if  $0 \in [a_2, b_2]$  then ERROR else  $[0, \max(|a_2| - 1, |b_2| - 1)]$  fi, where

$$\begin{aligned} -\infty \bmod c &\equiv \text{if } c \text{ is } 0 \text{ then ERROR else } [0, c - 1] \text{ fi} \\ +\infty \bmod c &\equiv \text{if } c \text{ is } 0 \text{ then ERROR else } [0, c - 1] \text{ fi} \\ -\infty \bmod +\infty &\equiv \text{ERROR} \\ +\infty \bmod -\infty &\equiv \text{ERROR} \end{aligned}$$

$[a_1, b_1] \exp [a_2, b_2] \equiv$

if  $0 \leq a_1 \leq b_1$  and  $0 \leq a_2 \leq b_2$  then  $[a_1^{a_2}, b_1^{b_2}]$

else  $[-\infty, +\infty]$  fi, where

$$\begin{aligned} -\infty \exp c &\equiv \text{if } c \text{ is } 0 \text{ then } 1 \text{ else } [-\infty, +\infty] \text{ fi} \\ +\infty \exp c &\equiv \text{if } c \text{ is } 0 \text{ then } 1 \text{ else } [-\infty, +\infty] \text{ fi} \\ -\infty \exp +\infty &\equiv \text{ERROR} \\ -\infty \exp -\infty &\equiv \text{ERROR} \end{aligned}$$

---

Figure 2.11: Arithmetic on Intervals.

A statement can be either (1) inside or (2) outside the scope of a loop.

Case (1) A statement outside the scope of a loop can be executed at most once, and has only one input context. Therefore, analysis of such a statement is always conservative.

Case (2) For a statement inside the scope of a loop, on  $j$ th and  $j + 1$ th iterations, the statement has input contexts  $(S_i)_j$  and  $(S_i)_{j+1}$ , respectively. As depicted in Figure 2.9, for each loop, we first execute the statement, and widen  $j$ th output context of that statement  $(S_o)_j$  with  $(S_i)_j$  to get  $(S_i)_{j+1}$ , i.e.,  $(S_i)_{j+1} = (S_i)_j \overset{\alpha}{\nabla} (S_o)_j$ . According to the definition of widening, we know that  $((S_i)_j \overset{\alpha}{\cup} (S_o)_j) \overset{\alpha}{\subseteq} ((S_i)_j \overset{\alpha}{\nabla} (S_o)_j)$ . Therefore,  $(S_i)_j \overset{\alpha}{\subseteq} (S_i)_{j+1}$ . Hence, analysis of a statement is sound if this statement is inside a loop. Thus the analysis of a statement is sound, i.e., for any input contexts,  $(S_i)_j \overset{\alpha}{\subseteq} (S_i)_{j+1}$ , where  $j$  is a natural number. □

## 2.7 Performance

Given a program  $PG$ , let  $|V|$  be the total number of variables, global and local, and  $n$  be the number of statements in  $PG$ . The worst case of the AI algorithm occurs when the program has  $|V|$  loops, and each loop widens exactly one variable. We go through each loop at most 20 times; therefore, each statement in  $PG$  can be changed at most  $20 \times |V|$  times, and there are  $n \times 20 \times |V|$  changes altogether. Furthermore, every state has at most  $n - 1$  predecessors. For each change of state, we union abstract values of variables of all the predecessors, which takes  $(n - 1) \times |V| \times m$  steps ( $m$  is a constant proportional to the number of intervals associated with each variable). Therefore, the entire computation of the abstract interpreter takes  $20 \times |V| \times n \times m \times (n - 1) \times |V|$  steps, which is  $O(|V|^2 \times n^2)$ .

## 2.8 Testing

Given an implementation whose internal structure is unknown, one usually wants to check if the implementation satisfies certain properties by providing a finite number of

testcases. However, this technique is very expensive because the number of testcases needed to cover the system behavior is unmanageably large. It is also possible that the program takes an infinitely long time for each run because termination is not guaranteed. We propose abstract testing as a solution to the above problems. With abstract testing, we are able to have a finite representation of an infinite number of testcases. The reason is that we can abstract a set of input values to an interval, and the abstraction function is chosen such that the resulted number of intervals is finite. Also, since the abstract state space is finite and much smaller than the concrete state space, termination is guaranteed.

Ultimately, we want the abstract interpreter to act as a symbolic tester, where we are allowed to specify abstract testcases. Testcases are finite sequences of values that will be read when the system requests input from a certain monitor. The values for a variable may be singletons (i.e.,  $[3, 3]$ ), all possible values (i.e.,  $[-\infty, +\infty]$ ) or other combinations. This allows us to combine “analysis” of some variables with “testing” of others. The result is an interpreter which is able to reason about program correctness. The formal grammar of the testcase specification language is given in Appendix D. On each line, the user specifies a *line number*, and a *variable*, followed by its *intervals*, for example:

LINE 5 xy 4 TO 5, 6

The interpretation of above example is: on line 5 in the program, two intervals  $[4, 5]$  and  $[6, 6]$  will be read if the program requests a value for variable **xy**.

As depicted in Figure 1.1, AI has an optional input: a file of testcases. We start by parsing the testcases and building a table in which each row is associated with a line number of the program and a list of variables with their intervals that we wish to read on this line. An example of a *testcases file* is given in Figure 2.15. MINF and INF represent  $-\infty$  and  $+\infty$ , respectively. We keep a global pointer which indicates what is the currently active row in the table, and set it to the first row of the table at the beginning of the execution. Next, we start executing AI on the input program. Whenever there is an input request, we compare the line numbers in the current row with the one

in the program, and if they do not match, we give values  $[-\infty, +\infty]$  to the variable and proceed; otherwise, we fetch the row from the table, move the pointer to the next row and proceed. If the testcases file includes more testcases than what the program actually requests, the rest of the testcases are ignored. If, on the opposite, there are less testcases provided than what the program requests, we repeatedly feed to the program the last inputs with the matching line number, if they are available. We assume the input values to be  $[-\infty, +\infty]$  in all other cases.

In order to understand how our system analyzes testcases, let us consider some concrete examples. The program depicted in Figure 2.16 requests input on lines 6 and 10. We start executing the program and set the pointer `pt` to point to the first row of Figure 2.15. When the control reaches line 6, the program asks for input. Then `a` gets  $\{[0, 0]\}$  and `b` gets  $\{[-\infty, -1], [0, 0], [1, +\infty]\}$ . `pt` moves to the next row. The statement on line 10 is inside a loop. `xy` gets values  $\{[0, 0]\}$  on the first iteration,  $\{[-5, -1], [4, 8]\}$  on the second iteration, and  $\{[-10, -5], [-1, -1], [1, +\infty]\}$  on the third iteration. For the following iterations, testcases are no longer provided. We then keep reading the same last input which is  $\{[-10, -5], [-1, -1], [1, +\infty]\}$ , until a fixpoint is reached. Notice that we do the usual widening for input values. For example, in the input context to line 8 (the beginning of the while loop), the values of `xy` are: *UNDEF* on the first iteration,  $\{[0, 0]\}$  on the second iteration,  $\{[-5, 0], [0, 8]\}$  on the third iteration,  $\{[-10, 8], [-5, 8], [-5, +\infty]\}$  on the fourth iteration, and remains the same in the subsequent steps. This program gives the following output:

```
a min 0, max 0
```

```
b min 12, max 12
```

```
c min -10, max 8, min -5, max 9, min -5, max INF
```

Usually, it is difficult to generate equivalence classes for input values, because it is unclear which answer each testcase can generate. With abstract testcases, we can refine input values from large ranges into smaller ranges. By symbolically executing the program, we know the answer that each testcase yields, and from these answers, we may easily judge what good equivalence classes are.

## 2.9 Summary

In this Chapter, we developed an abstract interpreter. Input to the AI is a sequential imperative program written in a subset of C and an optional testcases file. Thus, a possible application of our AI is symbolic abstract tester. The AI executes the program under analysis symbolically, and at the end of analysis produces an abstract finite-state program  $PG^\alpha = (W^\alpha, R^\alpha, I^\alpha, P, L^\alpha_T, L^\alpha_F)$ . The abstraction function abstracts a set to an interval which may represent an infinite number of values. The process takes  $O(|V|^2 \times n^2)$  steps, where  $|V|$  is the total number of variables in the program, and  $n$  is the number of statements in  $PG$ . We also demonstrated the soundness of our analysis, that is, given a set of values  $s$ , the concretization of our abstract analysis of  $s$  contains no less information than  $s$ .

$$\begin{aligned}
[a_1, b_1] = [a_2, b_2] &\equiv \\
&\text{if } a_1 = b_1 = a_2 = b_2 \text{ then } [True, True] \\
&\text{else if } a_1 < b_1 < a_2 < b_2 \text{ then } [False, False] \\
&\text{else if } a_2 < b_2 < a_1 < b_1 \text{ then } [False, False] \\
&\text{else } [False, True] \text{ fi, where} \\
&\quad -\infty = -\infty \equiv [False, True] \\
&\quad +\infty = +\infty \equiv [False, True] \\
\\
[a_1, b_1] > [a_2, b_2] &\equiv \\
&\text{if } a_2 < b_2 < a_1 < b_1 \text{ then } [True, True] \\
&\text{else if } a_1 < b_1 < a_2 < b_2 \text{ then } [False, False] \\
&\text{else } [False, True] \text{ fi, where} \\
&\quad -\infty > -\infty \equiv [False, True] \\
&\quad +\infty > +\infty \equiv [False, True] \\
\\
[a_1, b_1] < [a_2, b_2] &\equiv \\
&\text{if } a_2 < b_2 < a_1 < b_1 \text{ then } [False, False] \\
&\text{else if } a_1 < b_1 < a_2 < b_2 \text{ then } [True, True] \\
&\text{else } [False, True] \text{ fi, where} \\
&\quad -\infty < -\infty \equiv [False, True] \\
&\quad +\infty < +\infty \equiv [False, True] \\
\\
[a_1, b_1] \geq [a_2, b_2] &\equiv \\
&\text{if } [a_1, b_1] > [a_2, b_2] \text{ is } [True, True] \text{ or } [a_1, b_1] = [a_2, b_2] \text{ is } [True, True] \\
&\quad \text{then } [True, True] \\
&\text{else if } [a_1, b_1] > [a_2, b_2] \text{ is } [False, False] \text{ and } [a_1, b_1] = [a_2, b_2] \text{ is } [False, False] \\
&\quad \text{then } [False, False] \\
&\text{else } [False, True] \text{ fi} \\
\\
[a_1, b_1] \leq [a_2, b_2] &\equiv \\
&\text{if } [a_1, b_1] < [a_2, b_2] \text{ is } [True, True] \text{ or } [a_1, b_1] = [a_2, b_2] \text{ is } [True, True] \\
&\quad \text{then } [True, True] \\
&\text{else if } [a_1, b_1] < [a_2, b_2] \text{ is } [False, False] \text{ and } [a_1, b_1] = [a_2, b_2] \text{ is } [False, False] \\
&\quad \text{then } [False, False] \\
&\text{else } [False, True] \text{ fi} \\
\\
[a_1, b_1] \neq [a_2, b_2] &\equiv \\
&\text{if } [a_1, b_1] = [a_2, b_2] \text{ is } [True, True] \text{ then } [False, False] \\
&\text{else if } [a_1, b_1] = [a_2, b_2] \text{ is } [False, False] \text{ then } [True, True] \\
&\text{else } [False, True] \text{ fi}
\end{aligned}$$

---

Figure 2.12: Comparison on Intervals.

$$\begin{aligned}
\{a\} \overset{\alpha}{+} UNDEF &\equiv UNDEF \\
\{a\} \overset{\alpha}{+} \{b_1, \dots, b_n\} &\equiv \{a + b_1, \dots, a + b_n\} \\
\{a_1, \dots, a_m\} \overset{\alpha}{+} \{b_1, \dots, b_n\} &\equiv \{a_1 \cup \dots \cup a_m\} \overset{\alpha}{+} \{b_1, \dots, b_n\} \\
\{a\} \overset{\alpha}{+} \{b\} &\equiv \{b\} \overset{\alpha}{+} \{a\} \\
\\
\{a\} \overset{\alpha}{-} UNDEF &\equiv UNDEF \\
UNDEF \overset{\alpha}{-} \{a\} &\equiv UNDEF \\
\{a\} \overset{\alpha}{-} \{b_1, \dots, b_n\} &\equiv \{a - b_1, \dots, a - b_n\} \\
\{a_1, \dots, a_m\} \overset{\alpha}{-} \{b_1, \dots, b_n\} &\equiv \{a_1 \cup \dots \cup a_m\} \overset{\alpha}{-} \{b_1, \dots, b_n\} \\
\\
\{a\} \overset{\alpha}{\times} UNDEF &\equiv UNDEF \\
\{a\} \overset{\alpha}{\times} \{b_1, \dots, b_n\} &\equiv \{a \times b_1, \dots, a \times b_n\} \\
\{a_1, \dots, a_m\} \overset{\alpha}{\times} \{b_1, \dots, b_n\} &\equiv \{a_1 \cup \dots \cup a_m\} \overset{\alpha}{\times} \{b_1, \dots, b_n\} \\
\{a\} \overset{\alpha}{\times} \{b\} &\equiv \{b\} \overset{\alpha}{\times} \{a\} \\
\\
\{a\} \overset{\alpha}{\div} UNDEF &\equiv UNDEF \\
UNDEF \overset{\alpha}{\div} \{a\} &\equiv UNDEF \\
\{a\} \overset{\alpha}{\div} \{b_1, \dots, b_n\} &\equiv \{a \div b_1, \dots, a \div b_n\} \\
\{a_1, \dots, a_m\} \overset{\alpha}{\div} \{b_1, \dots, b_n\} &\equiv \{a_1 \cup \dots \cup a_m\} \overset{\alpha}{\div} \{b_1, \dots, b_n\} \\
\\
\{a\} \overset{\alpha}{\text{mod}} UNDEF &\equiv UNDEF \\
UNDEF \overset{\alpha}{\text{mod}} \{a\} &\equiv UNDEF \\
\{a\} \overset{\alpha}{\text{mod}} \{b_1, \dots, b_n\} &\equiv \{a \text{ mod } b_1, \dots, a \text{ mod } b_n\} \\
\{a_1, \dots, a_m\} \overset{\alpha}{\text{mod}} \{b_1, \dots, b_n\} &\equiv \{a_1 \cup \dots \cup a_m\} \overset{\alpha}{\text{mod}} \{b_1, \dots, b_n\} \\
\\
\{a\} \overset{\alpha}{\text{exp}} UNDEF &\equiv UNDEF \\
UNDEF \overset{\alpha}{\text{exp}} \{a\} &\equiv UNDEF \\
\{a\} \overset{\alpha}{\text{exp}} \{b_1, \dots, b_n\} &\equiv \{a \text{ exp } b_1, \dots, a \text{ exp } b_n\} \\
\{a_1, \dots, a_m\} \overset{\alpha}{\text{exp}} \{b_1, \dots, b_n\} &\equiv \{a_1 \cup \dots \cup a_m\} \overset{\alpha}{\text{exp}} \{b_1, \dots, b_n\}
\end{aligned}$$

---

Figure 2.13: Arithmetic on Sets of Intervals.

$$\begin{aligned}
\{a\} &\stackrel{\alpha}{=} UNDEF \equiv UNDEF \\
\{a\} &\stackrel{\alpha}{=} \{b_1, \dots, b_n\} \equiv \{a = b_1, \dots, a = b_n\} \\
\{a_1, \dots, a_m\} &\stackrel{\alpha}{=} \{b_1, \dots, b_n\} \equiv \{a_1 \cup \dots \cup a_m\} \stackrel{\alpha}{=} \{b_1, \dots, b_n\} \\
\{a\} &\stackrel{\alpha}{=} \{b\} \equiv \{b\} \stackrel{\alpha}{=} \{a\} \\
\\
\{a\} &\stackrel{\alpha}{>} UNDEF \equiv UNDEF \\
UNDEF &\stackrel{\alpha}{>} \{a\} \equiv UNDEF \\
\{a\} &\stackrel{\alpha}{>} \{b_1, \dots, b_n\} \equiv \{a > b_1, \dots, a > b_n\} \\
\{a_1, \dots, a_m\} &\stackrel{\alpha}{>} \{b_1, \dots, b_n\} \equiv \{a_1 \cup \dots \cup a_m\} \stackrel{\alpha}{>} \{b_1, \dots, b_n\} \\
\\
\{a\} &\stackrel{\alpha}{<} UNDEF \equiv UNDEF \\
UNDEF &\stackrel{\alpha}{<} \{a\} \equiv UNDEF \\
\{a\} &\stackrel{\alpha}{<} \{b_1, \dots, b_n\} \equiv \{a < b_1, \dots, a < b_n\} \\
\{a_1, \dots, a_m\} &\stackrel{\alpha}{<} \{b_1, \dots, b_n\} \equiv \{a_1 \cup \dots \cup a_m\} \stackrel{\alpha}{<} \{b_1, \dots, b_n\} \\
\\
\{a\} &\stackrel{\alpha}{\geq} UNDEF \equiv UNDEF \\
UNDEF &\stackrel{\alpha}{\geq} \{a\} \equiv UNDEF \\
\{a\} &\stackrel{\alpha}{\geq} \{b_1, \dots, b_n\} \equiv \{a \geq b_1, \dots, a \geq b_n\} \\
\{a_1, \dots, a_m\} &\stackrel{\alpha}{\geq} \{b_1, \dots, b_n\} \equiv \{a_1 \cup \dots \cup a_m\} \stackrel{\alpha}{\geq} \{b_1, \dots, b_n\} \\
\\
\{a\} &\stackrel{\alpha}{\leq} UNDEF \equiv UNDEF \\
UNDEF &\stackrel{\alpha}{\leq} \{a\} \equiv UNDEF \\
\{a\} &\stackrel{\alpha}{\leq} \{b_1, \dots, b_n\} \equiv \{a \leq b_1, \dots, a \leq b_n\} \\
\{a_1, \dots, a_m\} &\stackrel{\alpha}{\leq} \{b_1, \dots, b_n\} \equiv \{a_1 \cup \dots \cup a_m\} \stackrel{\alpha}{\leq} \{b_1, \dots, b_n\} \\
\\
\{a\} &\stackrel{\alpha}{\neq} UNDEF \equiv UNDEF \\
\{a\} &\stackrel{\alpha}{\neq} \{b_1, \dots, b_n\} \equiv \{a \neq b_1, \dots, a \neq b_n\} \\
\{a_1, \dots, a_m\} &\stackrel{\alpha}{\neq} \{b_1, \dots, b_n\} \equiv \{a_1 \cup \dots \cup a_m\} \stackrel{\alpha}{\neq} \{b_1, \dots, b_n\}
\end{aligned}$$

---

Figure 2.14: Comparison on Sets of Intervals.

```
6 a [0, 0] b [MINF, -1] [0, 0] [1, INF]
10 xy [0, 0]}
10 xy [-5, -1] [4, 8]
10 xy [-10, -5] [-1, -1] [1, INF]
```

---

Figure 2.15: A Testcases File.

```
1: int a;
2: int b;
3: int xy;
4: int main() {
5:     int c;
6:     scan(b, a);
7:     c = 2;
8:     while (1) {
9:         if (a == 0) {
10:             scan(xy);
11:             b = 12;
12:         }
13:         else
14:             c = c * 2;
15:     }
16: print(a, b, xy);
17: }
```

---

Figure 2.16: A Program Fragment.



# Chapter 3

## Light-weight Model Checking

In this chapter, we consider combining abstract interpretation with CTL model checking. This combination enables us to analyze infinite-state programs and reason about values of variables in CTL formulae combined with arithmetic. The goal of this work is to use abstract interpretation to alleviate the state explosion problem of model checking while ensuring that the properties verified on the abstract system can be properly interpreted in the original system. This goal is achieved by constructing an abstract model checker on our three-valued logic that returns values *Yes*, *No* and *Maybe*, such that the analysis that results in *Yes* and in *No* are sound.

### 3.1 Background

Model checking is a technique for automatically verifying properties of a system. Examples of such properties include liveness, deadlock freedom, etc. Given a system and a property, a model checker builds the reachability graph by exhaustively exploring the state space of the system. Thus, model-checking is typically restricted to finite-state systems, with a few notable exceptions, e.g. [36, 14]. Properties are usually presented as formulas in some temporal logic. In this chapter we will concern ourselves with *CTL model checking* – an automatic technique for verifying properties expressed in a propositional branching-time temporal logic called *Computation Tree Logic* (CTL) [20]. The

system is defined by a Kripke structure, and properties are evaluated on a tree of infinite computations produced by the model of the system. The standard notation  $M, s \models f$  indicates that a formula  $f$  holds in a state  $s$  of a model  $M$ . If a formula holds in the initial state, it is considered to hold in the model.

A Kripke structure consists of a set of states  $S$ , a transition relation  $R \subseteq S \times S$ , a set of initial states  $I$ , a set of atomic propositions  $P$ , and a labeling function  $L : S \rightarrow 2^P$ .  $R$  must be a total function, i.e.,  $\forall s \in S, \exists t \in S, \text{ s.t. } (s, t) \in R$ . If a state  $s_n$  has no successors, we add a self-loop to it, so that  $(s_n, s_n) \in R$ . Intuitively, for each  $s \in S$ , the labeling function provides a list of atomic propositions which are *True* in this state.

We use CTL on boolean expressions that include variables and arithmetic operations on them. The set  $P$  of atomic propositions is defined as follows:

If  $P_1$  and  $P_2$  are terms, then  $P_1 = P_2, P_1 \neq P_2, P_1 > P_2, P_1 \geq P_2, P_1 < P_2, P_1 \leq P_2$  are atomic propositions.

Terms are defined recursively as

1. Every identifier and every constant is a term.
2. If  $t$  is a term, so is  $(t)$ .
3. If  $t_1$  and  $t_2$  are terms, so are  $t_1 + t_2, t_1 - t_2, t_1/t_2, t_1 \times t_2, t_1 \bmod t_2, t_1 \exp t_2$ .

$\bmod$  and  $\exp$  are the mod and the exponentiation functions, respectively. For example, the following are some atomic propositions in this version of CTL:

- $x > 5$
- $(x + 2)/3 = y$

CTL is then defined as follows:

1. Every atomic proposition  $p \in P$  is a CTL formula.
2. If  $\varphi$  and  $\psi$  are CTL formulas, then so are  $\neg\varphi, \varphi \wedge \psi, \varphi \vee \psi, EX\varphi, AX\varphi, EF\varphi, AF\varphi, E[\varphi U \psi], A[\varphi U \psi]$

The logic connectives  $\neg$ ,  $\wedge$  and  $\vee$  have their usual meanings. The existential (universal) quantifier  $E$  ( $A$ ) is used to quantify over paths. The operator  $X$  means “at the next step”,  $F$  represents “sometime in the future”, and  $U$  is “until”. Therefore,  $EX\varphi$  ( $AX\varphi$ ) means that  $\varphi$  holds in some (every) immediate successor of the current program state;  $EF\varphi$  ( $AF\varphi$ ) means that  $\varphi$  holds in the future along some (every) path emanating from the current state;  $E[\varphi U \psi]$  ( $A[\varphi U \psi]$ ) means that for some (every) computation path starting from the current state,  $\varphi$  continuously holds until  $\psi$  becomes true.

Formally,

$$\begin{aligned}
M, s_0 \models \varphi & \text{ iff } \varphi \in L(s_0) \\
M, s_0 \models \neg\varphi & \text{ iff } M, s_0 \not\models \varphi \\
M, s_0 \models \varphi \wedge \psi & \text{ iff } M, s_0 \models \varphi \wedge M, s_0 \models \psi \\
M, s_0 \models \varphi \vee \psi & \text{ iff } M, s_0 \models \varphi \vee M, s_0 \models \psi \\
M, s_0 \models EX\varphi & \text{ iff } \exists t \in S, (s_0, t) \in R \wedge M, t \models \varphi \\
M, s_0 \models AX\varphi & \text{ iff } \forall t \in S, (s_0, t) \in R \rightarrow M, t \models \varphi \\
M, s_0 \models E[\varphi U \psi] & \text{ iff there exists some path } s_0, s_1, \dots, \\
& \exists i, i \geq 0 \wedge M, s_i \models \psi \wedge \\
& \forall j, 0 \leq j < i \rightarrow M, s_j \models \varphi \\
M, s_0 \models A[\varphi U \psi] & \text{ iff for every path } s_0, s_1, \dots, \\
& \exists i, i \geq 0 \wedge M, s_i \models \psi \wedge \\
& \forall j, 0 \leq j < i \rightarrow M, s_j \models \varphi.
\end{aligned}$$

Note that  $AF(\varphi) \equiv A[True U \varphi]$ ; and  $EF(\varphi) \equiv E[True U \varphi]$ , that is, we are using a “strong until”. We also use the abbreviations  $EG(\varphi)$  and  $AG(\varphi)$  to represent the property that  $\varphi$  holds at every state for some (every) path emanating from  $s_0$ .  $EG$  and  $AG$  are defined as:

- $EG(\varphi) \equiv \neg AF(\neg\varphi)$
- $AG(\varphi) \equiv \neg EF(\neg\varphi)$

## 3.2 Construction of a Labeled Transition System

The AI receives the program under analysis and builds the abstract finite-state program  $PG^\alpha = (W^\alpha, R^\alpha, I^\alpha, P, L^\alpha_T, L^\alpha_F)$  using the process described in Section 2.4. Here, we describe the transformation of the abstract finite-state program representation into a labeled transition system.

In C-, as in C, there is no one-to-one correspondence between assignments to variables and lines of code. In fact, before attempting to verify programs expressed in C-, we need to give it a well-defined formal semantics, i.e., describe the way in which each construct transforms the “program state” [63]. We define a *state* to be a mapping between a set of global variables and their values, and thus a *state change* occurs when at least one of the global variables changes its value.

Our goal is to construct an abstract finite Kripke structure, in which every edge represents a state change in the program. In order to do that, we let  $G \subseteq V$  be the set of variables which are accessible at every point of the program. Thus,  $G$  is the set of *global variables*. In addition, we define a “global variable changed” predicate  $p$  on a state such that  $p(y)$  is *True* iff  $\exists x \in \text{pred}(y), \exists g \in G$ , s.t.  $(g, D^\alpha(x)) \neq (g, D^\alpha(y))$ , where  $g, D^\alpha(x)$  ( $g, D^\alpha(y)$ ) indicates  $g$ 's abstract value in state  $x$  ( $y$ ). Now we construct an abstract aggregate state space  $S^\alpha$  in which every element  $s^\alpha \in 2^{S^\alpha}$  contains one state  $w^\alpha$  which involves a change to a global variable, and other states that do not involve changes to global variables and can be reached from  $w^\alpha$  via the transitive closure of  $R^\alpha$  (denoted  $R^{\alpha*}$ ).  $s^\alpha$  is defined recursively as follows:

- if  $p(w^\alpha)$  is true, then  $w^\alpha \in s^\alpha$ .
- $\forall t \in W^\alpha$ , if  $R^{\alpha*}(w, t) \wedge \neg p(t)$ , then  $t \in s^\alpha$ .
- $w_1^\alpha \in s^\alpha \wedge w_2^\alpha \in s^\alpha \rightarrow \neg(p(w_1^\alpha) \wedge p(w_2^\alpha))$

The transitions between states in  $S^\alpha$ ,  $E^\alpha \subseteq S^\alpha \times S^\alpha$ , are thus defined as:

$$(s^\alpha, t^\alpha) \in E^\alpha \text{ iff } \exists i, j \text{ s.t.}, w_i^\alpha \in s^\alpha \wedge w_j^\alpha \in t^\alpha \wedge (w_i^\alpha, w_j^\alpha) \in R^\alpha$$

Our abstract Kripke structure  $K^\alpha = (S^\alpha, E^\alpha, I^\alpha, P, L^\alpha_T, L^\alpha_F)$  is now ready.

Note that each concrete state  $w \in W$  is characterized by a list of global variables associated with their concrete values and its line number in the program. In the concrete domain,  $\forall w \in W, L_T \cap L_F = \emptyset$ , and  $L_T \cup L_F = P$ . Under the assumptions of the Galois connection framework, an abstract system has at least as many behaviors as the corresponding concrete one. Typically, verification on abstracted systems is done either *conservatively* or *optimistically*. The former case provides “reliable negative” answers, with  $L^\alpha_T \supseteq L_T$ , and  $L^\alpha_F \subseteq L_F$ . The later case provides “reliable positive” answers, with  $L^\alpha_T \subseteq L_T$ , and  $L^\alpha_F \supseteq L_F$ . In either case, one side of the answer cannot be trusted. Our ultimate goal is to ensure that the analysis is done in the way that both the positive and the negative analyses are sound.

**Definition 1** *The analysis is sound with respect to Yes answer: the property holds in the original system if the analysis yields Yes for the property; the analysis is sound with respect to No answer: the property does not hold in the original system if the analysis yields No for the property.*

In order to build a system to satisfy the above definition, we introduce a third logical value *Maybe*. Thus, if the analysis concluded that a property *Maybe* holds in the system, then it is unknown whether or not the property holds in the concrete system. In our abstract Kripke Structure,  $L^\alpha_T \cap L^\alpha_F = \emptyset$ , but  $L^\alpha_T \cup L^\alpha_F \subseteq P$ .

Let us revisit the program fragment in Figure 2.2 (we present it again in Figure 3.1). Figure 3.2 shows the final Kripke structure of this program built from the control flow graph of Figure 2.6. Each state is associated with a line number of the statement that changes a global variable in the original program, and with the abstract values that global variables have after the execution of this state. For example, state 10 of Figure 3.2 is an aggregation of states 10 and 11 of Figure 2.6.

```

1:  int b;
2:  int xy;
3:  int main ( ) {
4:      int a;
5:      int c;
6:      b = 13;
7:      c = 2;
8:      xy = -20;
9:      while ( 1 ) {
10:         xy = xy + 4;
11:         if (xy == 0)
12:             b = 5;
13:         else
14:             b = b * c;
15:         if ((a != 0)&&(a >= -3))
16:             if ((a != 2)&&(a != 4)&&(a !=7))
17:                 if (a != -2)
18:                     c = 2;
19:         print('xy is ', xy);
20:         print('b is ', b);
21:     }
22: }

```

Figure 3.1: A Program Fragment (same as in Figure 2.2).

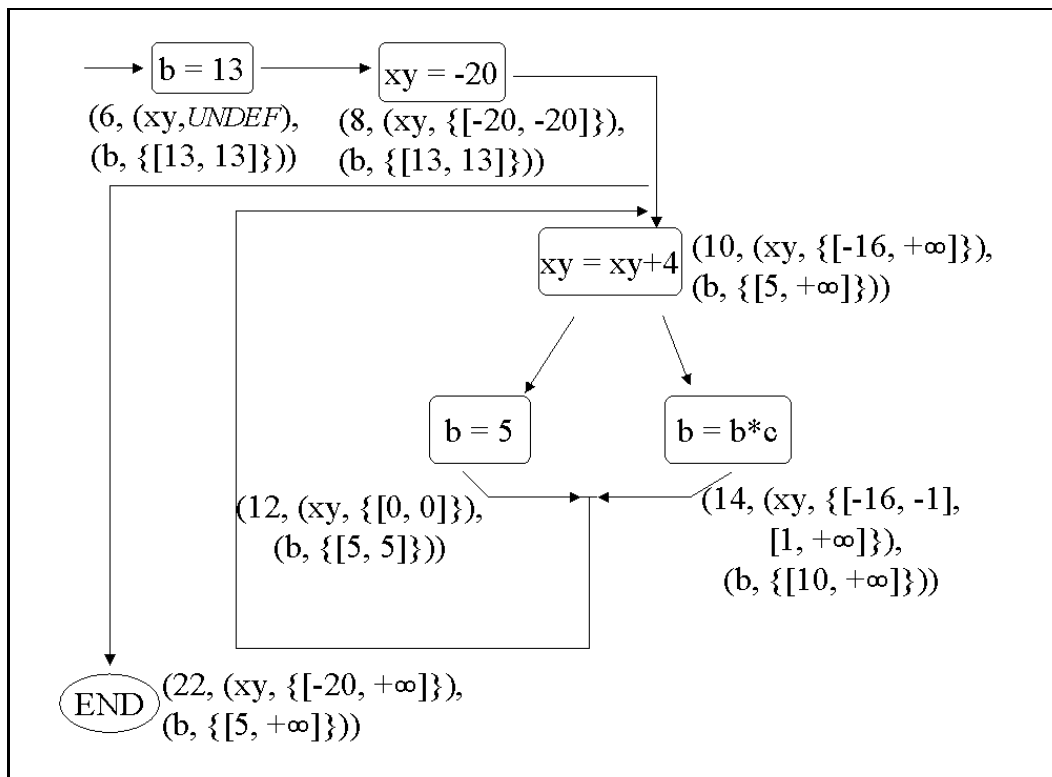


Figure 3.2: Kripke Structure  $K^\alpha$  Built from the Program Fragment in Figure 3.1.

### 3.3 Model Checking Algorithm

We now present the algorithm that receives a Kripke structure  $K^\alpha$  constructed above and a correctness property expressed in the version of CTL described in Section 3.1, and

```

Procedure CHECK( $\varphi$ )
CASE
 $\varphi \in P$       : Return (Yes( $\varphi$ ), No( $\varphi$ ))
 $\varphi = \neg p$    : Return (No( $p$ ), Yes( $p$ ))
 $\varphi = p \wedge q$  : Return (Yes( $p$ )  $\cap$  Yes( $q$ ), No( $p$ )  $\cup$  No( $q$ ))
 $\varphi = p \vee q$  : Return (Yes( $p$ )  $\cup$  Yes( $q$ ), No( $p$ )  $\cap$  No( $q$ ))
 $\varphi = EXp$     : Return (pred(Yes( $p$ )),  $S^\alpha - \text{pred}(S^\alpha - \text{No}(p))$ )
 $\varphi = AXp$     : Return ( $S^\alpha - \text{pred}(S^\alpha - \text{Yes}(p))$ , pred(No( $p$ )))
 $\varphi = E[pUq]$  : 1.  $Q_0 = \text{Yes}(q)$ 
                    $Q_{i+1} = Q_i \cup (\text{pred}(Q_i) \cap \text{Yes}(p))$ 
                   Until  $Q_m = Q_{m+1}$ 
                   2.  $T_0 = \text{No}(q)$ 
                    $T_{i+1} = T_i \cap ((S^\alpha - \text{pred}(S^\alpha - T_i)) \cup \text{No}(p))$ 
                   Until  $T_n = T_{n+1}$ 
                   3. Return ( $Q_m, T_n$ )
 $\varphi = A[pUq]$  : 1.  $Q_0 = \text{Yes}(q)$ 
                    $Q_{i+1} = Q_i \cup ((\text{pred}(Q_i) - \text{pred}(S^\alpha - Q_i)) \cap \text{Yes}(p))$ 
                   Until  $Q_m = Q_{m+1}$ 
                   2.  $T_0 = \text{No}(q)$ 
                    $T_{i+1} = T_i \cap (\text{pred}(T_i) \cup \text{No}(p))$ 
                   Until  $T_n = T_{n+1}$ 
                   3. Return ( $Q_m, T_n$ )

```

---

Figure 3.3: Model Checking Algorithm.

determines whether or not the property holds in the system.

The algorithm recursively goes through the structure of the property under analysis, associating each subproperty  $\varphi$  with a pair of sets of states (Yes( $\varphi$ ), No( $\varphi$ )). Yes( $\varphi$ )  $\subseteq S^\alpha$  is a set of states in which  $\varphi$  is *True*, or, more formally,  $s^\alpha \in \text{Yes}(\varphi)$  iff  $\varphi \in L^\alpha_T(s^\alpha)$ . No( $\varphi$ ) which represents a set of states in which  $\varphi$  is *False*, is defined similarly. We also define a *predecessor* function  $\text{pred} : 2^{S^\alpha} \rightarrow 2^{S^\alpha}$  which, given a set of states  $Q$ , returns all the states that can reach some state in  $Q$  in one transition:

$$s^\alpha \in \text{pred}(Q) \text{ iff } \exists t^\alpha \in Q \wedge (s^\alpha, t^\alpha) \in E^\alpha$$

The algorithm, inspired by Bultan's symbolic model checker for infinite-state systems [14], is given in Figure 3.3. For example, a property  $p \wedge q$  holds in state  $s^\alpha$  if  $s^\alpha$  is in Yes sets of both  $p$  and  $q$ . The same property does not hold in state  $s^\alpha$  if  $s^\alpha$  is in the No set of either  $p$  or  $q$ . When verifying  $EXp$ , we note that if  $p$  holds in some immediate

successor of state  $s^\alpha$ , then  $EXp$  holds in  $s^\alpha$ ; any immediate successor in which  $p$  may hold ( $S^\alpha - \text{No}(p)$ ) should be excluded from  $\text{No}(EXp)$ .  $A[pUq]$  is computed recursively as follows:  $A[pUq]$  is *Yes* in all states  $S_0$  in which  $q$  holds; it is also *Yes* in predecessors of  $S_0$  in which  $p$  holds and all of which successors are in  $S_0$ .  $A[pUq]$  is *No* in a state  $s^\alpha$  iff  $q$  does not hold in  $s^\alpha$  and either  $p$  does not hold in  $s^\alpha$  or one of its successors does not lead to  $q$ .

**Theorem 3** *Our model checker is sound with respect to both Yes and No answers.*

The proof of this theorem is given in Appendix A.

As discussed in Section 2.5.1, a variable's value is *UNDEF* if it has not been assigned any values. For an atomic proposition  $p$ , if there exists a variable appearing in  $p$  and whose value is *UNDEF* in some state  $s^\alpha$ , we put  $s^\alpha$  in both  $\text{Yes}(p)$  and  $\text{No}(p)$ . Moreover, as we check the properties recursively, if  $p$  is quantified by the universal quantifier  $A$ , we remove  $s^\alpha$  from  $\text{No}(p)$ ; if  $p$  is quantified by the existential quantifier  $E$ , we remove  $s^\alpha$  from  $\text{Yes}(p)$ , and proceed. Finally, a state is said to be *reachable* if it is the initial state, or at least one global variable is not *UNDEF* in it.

**Theorem 4** *The analysis of UNDEF is sound with respect to both Yes and No answers.*

The proof of this theorem is given in Appendix A.

The AI receives the program fragment given in Figure 3.1 and builds the abstract finite-state program  $PG^\alpha$ . We then transform this  $PG^\alpha$  into the final Kripke structure depicted in Figure 3.2 using the process described in Section 2.4. This structure becomes input to the Model Checker whose algorithm is described in Figure 3.3. For example, we can model check the structure depicted in Figure 3.2 against CTL properties  $AG((\mathbf{xy} + \mathbf{b}) \leq 0)$ ,  $EF(\mathbf{b} = 5)$ , and  $EF(\mathbf{b} = 12)$ . AMC returns *No* for the first property because it is violated in the state corresponding to line 12 of the program. The second property is determined to be *Yes* because it is satisfied in the state corresponding to line 12. The third property is determined to be *Maybe*: it *Maybe* holds in state corresponding to line 14 and does not definitely hold in any state. In addition, AMC returns information about

the states (abstract values of the global variables) corresponding to line 12 for the first property and line 14 for the third property.

## 3.4 Performance

To compute the performance of our model checker, we let  $|P|$  be the length of a property  $P$  under verification, and  $n$  be the number of states in  $K^\alpha$ . Among all the CTL formulas,  $A[\varphi U \psi]$  is the most complicated one. For this algorithm,  $Q_i$  can change value at most  $n$  times before a fixpoint is reached, and it takes  $n - 1$  steps to compute  $Q_i$ 's predecessors each time. Verification of this property takes  $O(n \times (n - 1))$  steps. Therefore, the total running time for our model checker to check a formula  $P$  is  $O(|P| \times n^2)$ . Notice that the verification process builds upon the abstract interpreter, i.e., this time complexity does not include the effort that our abstract interpreter takes to generate the abstract finite-state program.

## 3.5 Combining Model Checking with Environmental Assumptions

Model checking often results in *Maybe* answers, because inputs are assumed to include all possible values. We can use testcases to specify information about environment. Environmental assumptions may make the model checker more precise by eliminating some “meaningless” inputs.

An *environmental assumption* specifies constraints on the values of variables imposed either by laws of nature or by users in the system. Examples of such assumptions are: a person's age cannot be less than 0,  $a$ 's value is always greater than 5, etc. An environmental assumption is specified in the same format as a testcase. In the line number field, we either use 0 to indicate that this line is a specification of environmental assumption, or a real line number in the program to indicate that this line is a specification of abstract testcase. Testcases interact with the program continuously as long as the program

keeps requesting input, whereas environmental assumptions remain fixed throughout the program run. Thus we first process the assumptions, store the values for variables as part of the input context to the initial state, and execute the program with this input context.

Combining testing with model checking lets us limit the range of input domain, and thus possibly accelerate convergence of the analysis. For example, assume that the environmental assumption for Figure 3.4 is that  $b$  is always equal to 0. We process the assumption, and generate the input context  $((a, UNDEF), (b, \{[0, 0]\}), (xy, UNDEF))$ . The loop terminates immediately because  $b \neq 0$  evaluates to *False*.

```
1: int a;
2: int b;
3: int xy;
4: int main() {
5:     int c;
6:     scan(b, a);
7:     c = 2;
8:     while (b != 0) {
9:         if (a == 0) {
10:             scan(xy);
11:             b = 12;
12:         }
13:         else
14:             b = a * 2;
15:     }
16: }
```

---

Figure 3.4: A Program Fragment.

If we model check the program presented in Figure 3.4 against the property  $EF(b = 2)$ , without any environmental assumptions, AMC returns *Maybe*. On line 6, we assume that both  $a$  and  $b$  receive values  $[-\infty, +\infty]$ . Therefore, the condition  $a == 0$  *Maybe* holds, and lines 10-11 and 14 are both executed; in consequence,  $b = 2$  *Maybe* holds in the state corresponding to line 14 and does not definitely hold in any state. With the environmen-

tal assumption that  $a$  is always equal to 1,  $a == 0$  definitely does not hold,  $b = 2$  holds in the state corresponding to line 14, and AMC returns *Yes*.

## 3.6 Summary

In this chapter, we presented a light-weight model checker which receives an abstract finite-state program and a set of CTL formulae as input, checks each CTL formula and returns *Yes* (*No*) if the formula does (does not) hold in the system, or *Maybe* if the validity of the formula cannot be established. We have proposed a model-checking algorithm which recursively goes through the structure of the property under analysis, associating each subproperty with a pair of sets of states in which the subproperty holds and does not hold. The algorithm is provably sound with respect to both *Yes* and *No* answers. The process takes low-order polynomial time which is proportional to the size of the formula under analysis and the square of number of states in the final Kripke structure. In addition, we also demonstrated that model checking combined with environmental assumptions may help improve the precision of our analysis.



# Chapter 4

## Case Study

To determine the effectiveness of our abstract model checker, we analyzed the simplified version of a *Safety-Injection System* [24], specified using the SCR [1, 45] requirements notation.

### 4.1 The Application

Safety-Injection is an embedded system that monitors the water pressure and injects the coolant into the reactor core when the pressure falls below a certain threshold. There is a manual control that the operator can use to prevent the system from injecting the coolant, which causes the system to be overridden. A reset switch prevents the system from being overridden. The system inputs the value of the water pressure and outputs a boolean condition signifying whether to inject the coolant. In addition, it maintains the internal state reflecting the water pressure. If the water pressure falls below a threshold **Low**, the system’s pressure level becomes too low; if the water pressure rises above **Permit**, the system’s pressure level becomes high; otherwise, this level is “within the permitted range”.

The *software cost reduction* (SCR) requirements notation was developed by researchers at the Naval Research Laboratory for specifying requirements of reactive systems [1, 44, 45]. In SCR requirements specification, a system’s environmental behavior is specified as

a set of *monitored variables*, which represent inputs of the system, and a set of *controlled variables*, which represent outputs of the system. The system behavior is modeled by three entities: modes, terms and conditions and events. A *mode class* defines a set of states, called *modes*, described behaviorally in a table denoting current-state/next-state transitions. A *term* is a function defined on any entities. A *condition* is a predicate defined on monitored or controlled variables, mode classes, or terms. An *event* occurs when the value of any condition changes. A *conditioned event* [43] is defined as

$$@T(a) \text{ WHEN } [b] \equiv \neg a \wedge a' \wedge b,$$

where the unprimed conditions  $a$  and  $b$  are evaluated in the previous state, and the primed condition  $a'$  is evaluated in the current state.

Table 4.1 is the mode transition table of the Safety-Injection system. A mode class defines transitions between its modes, based on environmental changes — events. The mode class `Pressure` has three modes: `TooLow`, `Permitted` and `High`. At any given time, the system must be in exactly one of the above modes. A drop in `WaterPres` below the level `Low` will cause the system to enter mode `TooLow`, and a raise in `WaterPres` above the level `Permit` will cause the system to enter the mode `High`. In this table, each row specifies a conditioned event that triggers the transition from the source mode to the destination mode. A conditioned event can trigger one or more transitions between pairs of modes.

Source Mode	Events	Destination Mode
TooLow	@T(WaterPres >= Low)	Permitted
Permitted	@T(WaterPres < Low)	TooLow
Permitted	@T(WaterPres >= Permit)	High
High	@T(WaterPres < Permit)	Permitted

Table 4.1: Mode Transition Table.

Table 4.2 is an event table for defining the term `Overriden`. `Overriden` is *true* whenever safety injection is blocked, and *false* otherwise. Each row in this Table specifies

that when the system is in a certain mode (first column), if an event depicted in the second column occurs, the value of `Overriden` will become *true*, and if an event depicted in the third column occurs, the value of `Overriden` will become *false*. Notice that `@T(Inmode)` denotes that the system enters the corresponding mode. For instance, the first row of Table 4.2 says that if the system is currently in mode `High`, `Overriden` cannot become *true*, and `Overriden` will become *false* if the system enters the mode `TooLow`.

Modes	Events	
High	Never	@T(Inmode)
TooLow, Permitted	@T(Block = On) When Reset = Off	@T(Inmode) OR @T(Reset = On)
Overriden' =	true	false

Table 4.2: Event Table.

We describe the behavior of `SafetyInjection` (the controlled variable) using Table 4.3. The first row reads: if `Pressure` is `High` or `Permitted`, `SafetyInjection` will be turned off in the next state.

Modes	Conditions	
High, Permitted	true	false
TooLow	Overriden	NOT Overriden
SafetyInjection'	Off	On

Table 4.3: Condition Table.

We have implemented the Safety-Injection system as a 116-line C- program with 8 global variables closely reflecting those of the specification: `WaterPres` of type integer, `Block` and `Reset` of type boolean, `Injection` of type boolean, `Overriden` of type boolean, constants `Low` and `Permit`, and `Pressure` of type integer (our system does not allow enumerated types). The implementation also includes 7 functions and 8 local variables. The complete implementation of the Safety-Injection System is given in Figure 4.2.

## 4.2 Verification

The specification language of our system is expressive enough to capture complex properties of the Safety-Injection system, such as:

1.  $AG((\text{Reset} \wedge \text{Pressure} \neq \text{High}) \rightarrow \neg \text{Overriden})$

*Overriden will become false if the system has been reset and pressure is not high.*

This property corresponds to the *event table* — Table 4.2.

2.  $AG((\text{Reset} \wedge \text{Pressure} = \text{TooLow}) \rightarrow \text{Injection})$

*The system will inject the coolant if the pressure is too low and the reset button is pressed.* This property corresponds to the *condition table* — Table 4.3.

3.  $AG((\text{Block} \wedge \neg \text{Reset}) \rightarrow \neg \text{Overriden})$

*If the block button is pressed and the reset button is not pressed, Overriden will become false.* This property also corresponds to the *event table* — Table 4.2.

4.  $AG((\text{Pressure} = \text{TooLow} \wedge \text{WaterPres} \leq \text{Permit})$   
 $\rightarrow AX(\text{WaterPres} \geq \text{Permit} \rightarrow AF(\text{Pressure} = \text{High})))$

*Whenever the pressure is too low and the water pressure raises above the allowed threshold, then the system will eventually transit into a state where the pressure is high.* This property corresponds to the *mode transition table* — Table 4.1.

5.  $AG(\text{WaterPres} \geq 0)$

*The water pressure should always be greater or equal to 0.*

We verified the above properties on Sun UltraSPARC-II with 4 X 400 MHz processors and 4 GB of RAM. Table 4.4 lists running times for building the abstract Kripke structure and verifying the properties. In particular, building an abstract finite-state program for the Safety-Injection system took 3.43 seconds (user) and 4.88 seconds (system); the entire verification effort, including building the abstract finite-state program and checking all properties, took 3.82 seconds (user), 5.69 seconds (system). Our model-checker yielded

Properties	User (Seconds)	System (Seconds)	Answer
	3.43	4.88	
(1)	3.36	5.24	<i>Yes</i>
(2)	3.32	5.20	<i>Yes</i>
(3)	3.40	5.32	<i>Yes</i>
(4)	3.64	5.44	<i>Yes</i>
(5)	3.05	4.50	<i>Maybe</i>
(1)-(5)	3.82	5.69	

Table 4.4: Running Time for Verifying Different Properties.

*True* for properties (1)-(4), and *Maybe* for the fifth property. The final Kripke structure consisted of only 30 abstract states.

Safety-Injection has been verified by two other research groups. Bultan [13] has verified Properties 1 and 2, although we were unable to determine the exact size of his models. Bharadwaj and Heitmeyer [6] also verified Properties 1 and 2. The unabstracted system (SPIN performs on-the-fly verification, without building a complete state space) consists of over 1.7 million states, whereas with the combination of the abstraction methods we described in Section 1.2, it brings the state space down to 650 states.

### 4.3 Assumptions on Environment

Assumptions are invariant constraints that must hold in all system states. Environmental assumptions are the inputs of the system specified as testcases, and they were designed to help improve accuracy of our analysis. We can specify all the assumptions that the system can have, either by laws of nature or by other parts of the system.

Notice that Property 5 evaluated to *Maybe*, because we assumed that `WaterPres` receives values  $[-\infty, +\infty]$  whenever it requests input. If we run the system with the environmental assumptions given in Figure 4.1 (`INF` represents  $+\infty$ ), the model-checker

```
LINE 0 WaterPres [0, 5] [10, INF]
```

---

Figure 4.1: The Environmental Assumptions for Safety-Injection System.

will yield *Yes* for all five properties this time. Different runs under various environmental assumptions may help characterize the relationship between input to the system and the validity of the problem.

## 4.4 Summary

In this chapter, we demonstrated our analysis technique described in previous chapters on a more realistic application — the Safety-Injection system. We have implemented the system as a 116-line C- program, and verified it against 5 properties. Finally, we imposed certain environmental assumptions to the system, and showed that the resulting analysis is more accurate.

```
1: boolean Block;
2: boolean Reset;
3: boolean Exit;
4: int WaterPres; /* MONITORED VARIABLES */
5: boolean Injection; /* CONTROLLED VARIABLES */
6: boolean Overriden; /* TERMS */
7: boolean buttonBPressed;
8: boolean buttonRPressed;
9: boolean buttonEPressed;
10: int next;
11: int Pressure; /* MODE CLASS */

12: int Initialize () {
13:     WaterPres = 4;
14:     Pressure = 0;
15:     Overriden = 0;
16:     Injection = 0;
17:     Block = 0;
18:     Reset = 0;
19:     buttonBPressed = 0;
20:     buttonRPressed = 0;
21:     buttonEPressed = 0;
22:     next = 2;
23:     return 1;
24: }

25: int Get_Event (int sem) {
26:     int temp;
27:     temp = 1;
28:     if (sem == 1) {
29:         if (buttonBPressed == 0) {
30:             Block = 0;
31:             buttonBPressed = 1;
32:         }
33:         else {
34:             Block = 1;
35:             buttonBPressed = 0;
36:         }
37:     }
```

---

Figure 4.2: Safety-Injection Implementation.

```
38:         if (sem == 2) {
39:             if (buttonRPressed == 0) {
40:                 Reset = 0;
41:                 buttonRPressed = 1;
42:             }
43:             else
44:                 {
45:                 Reset = 1;
46:                 buttonRPressed = 0;
47:             }
48:         }
49:         if (sem == 3) {
50:             if (buttonEPressed == 0) {
51:                 Exit = 0;
52:                 buttonEPressed = 1;
53:             }
54:             else {
55:                 Exit = 1;
56:                 buttonEPressed = 0;
57:             }
58:         }
59:         return 1;
60:     }

61: int Get_Mode () {
62:     int temp;
63:     temp = 1;
64:     if (Pressure == 0) {
65:         if (WaterPres >= 5)
66:             Pressure = 1;
67:     }
```

---

Figure 4.2: Safety-Injection Implementation (cont'd).

```
68:         if (Pressure == 1) {
69:             if (WaterPres >= 15)
70:                 Pressure = 2;
71:             if (WaterPres < 5)
72:                 Pressure = 0;
73:         }
74:         if (Pressure == 2) {
75:             if (WaterPres < 15)
76:                 Pressure = 1;
77:         }
78:         return 1;
79:     }

80: int Get_Term() {
81:     if ((Reset == 0) && (Pressure == 0))
82:         if (Block == 1)
83:             Overriden = 1;
84:     if ((Pressure == 1) && (Reset == 0))
85:         if (Block == 1)
86:             Overriden = 1;
87:     if (Pressure == 2)
88:         Overriden = 0;
89:     return 1;
90: }

91: int Get_Control () {
92:     if (Overriden == 0)
93:         Injection = 1;
94:     if (Pressure == 2)
95:         Injection = 0;
96:     if (Pressure == 1)
97:         Injection = 0;
98:     if ((Pressure == 0) && (Overriden == 1))
99:         Injection = 0;
100:     return 1;
101: }
```

---

Figure 4.2: Safety-Injection Implementation (cont'd).

```
102: main() {
103:     int flag;
104:     int semo;
105:     int flag1;

106:     fopen("safeinput", "r");
107:     flag = Initialize();
108:     while (1) {
109:         scanf(WaterPres);
110:         fscanf("safeinput", semo);
111:         flag1 = Get_Event(semo);
112:         flag = Get_Mode();
113:         flag1 = Get_Term();
114:         flag1 = Get_Control();
115:     }
116: }
```

---

Figure 4.2: Safety-Injection Implementation (cont'd).

# Chapter 5

## Conclusion

In this chapter, we conclude this thesis with a summary of our work and its main contributions, a discussion of limitations of our current work and an outline of future research directions.

### 5.1 Summary

In the first part of this thesis, we developed an abstract interpreter with the running time of  $O(|V|^2 \times n^2)$ , where  $|V|$  is the total number of variables, global and local, and  $n$  is the number of statements in the program. The AI inputs the program under analysis written in a subset of C and an optional testcases file. Hence, the AI also acts as a symbolic tester, where users are allowed to specify abstract testcases. This enables one to do one-path analysis, all-paths analysis, or a combination of these two. No user-created abstractions are necessary, and at the end of the analysis, AI produces an abstract finite-state program, in which every state corresponds to a line number and a list of variables associated with their abstract values.

In the second part, we presented a light-weight model checker which receives an abstract finite-state program produced by the AI and a set of CTL formulae as its input, checks each CTL formula and yields answers. The verification always converges and is guaranteed to be sound: if the model checker yields *Yes*, the property holds in the concrete

system, and if it yields *No*, the property does not hold. In addition, we also demonstrated that model checking combined with environmental assumptions may help improve the precision of our analysis, reduce the number of inconclusive answers *Maybe*, and accelerate the convergence of the analysis. The process converges in low-order polynomial time  $O(|P| \times n^2)$ , where  $P$  is the property under analysis, and  $n$  is the number of statements in the program.

Finally, we demonstrated our analysis technique on the Safety-Injection System. The Safety-Injection system has been implemented in C-, and subsequently verified. We also showed that by imposing environmental assumptions to the system, our abstract model checker becomes more effective.

Our approach is not limited to the analysis of programs; it can be applied to finite-state and infinite-state specifications equally well. We also believe that tightening up the code of our model checker and making the state encoding symbolic will further improve its running time.

## 5.2 Contributions

Major contributions of our work are:

1. We created a symbolic abstract tester that guarantees convergence, and allows to specify abstract testcases.
2. We have developed an abstract model checker (AMC)
  - that can reason about values of infinite-domain variables and arithmetic;
  - that guarantees convergence;
  - whose algorithm is proved sound with respect to both *Yes* and *No* answers: a property holds in the original system if our AMC yields *Yes* for the property; a property does not hold in the original system if our AMC yields *No* for the property;

- in which abstraction is computed automatically, independent of the property under verification;
- whose algorithm is very cheap: the entire verification effort, including computing the abstract finite-state program and checking the property under investigation, takes low-order polynomial time;
- which allows to explicitly specify assumptions on the environment.

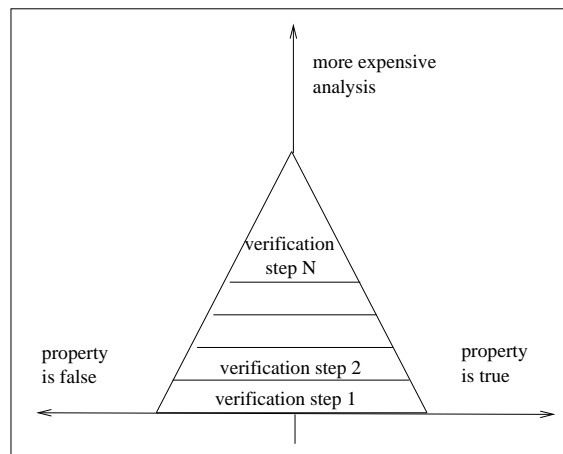


Figure 5.1: Framework for Automatic Verification.

Our model checker can be used as a level 1 verifier in the step-wise automatic verification framework discussed below. Given a large number of available verification techniques and a potential complexity and expense of their application and interpretation of results, we propose a “layered” approach to automatic verification, depicted in Figure 5.1. This technique is particularly inspired by the effectiveness and potential of lightweight formal methods [50]. Given a system  $S$  and a property  $P$ , we would like to know if  $P$  holds in  $S$ . We would like to start at verification level 1, which is fairly inexpensive, both in terms of the work required of the user, and in terms of computing resources required. This step will result in one of three conclusions:  $P$  is definitely true or definitely false in  $S$ , at which point the verification stops, or the analysis cannot yield any information. In the later case, the analyst will apply a technique on verification level 2. This technique is more expensive than that of verification level 1, but may help in determining whether

or not  $P$  holds. If it does not, the analyst proceeds in applying more and more complex and expensive techniques until (1)  $P$  is definitely proved or definitely disproved or (2) all levels are exhausted or (3) all resources are exhausted. Note that no precision is lost at each level. All properties that have not been concluded to definitely hold or definitely not hold on  $S$  during the verification level  $k - 1$  have to proceed to level  $k$ .

What is the benefit of the step-wise verification framework outlined above? It allows us to categorize existing tools based on their effectiveness in verifying properties and the complexity of application (Note that this complexity metric includes the effort needed by a human and the effort needed by a computer). This also allows one to utilize verification efforts more effectively.

### 5.3 Limitations

This thesis is an initial attempt to establish a framework for step-wise automatic verification. The results of our work are limited in several ways:

1. The implementation of the tool cannot handle complex constructs of the input language. These include recursion, user-defined data types, dynamic memory allocation, pointers, etc. We also currently limit our verification to sequential programs.
2. Our tool interacts with the Omega library, which can only handle operations on integer-valued variables. Thus, reasoning about floating-point numbers is currently not supported.
3. There is only one built-in level of abstraction provided in our system.
4. The input language, being a subset of C, does not have formal semantics; in particular, the notion of a *state transition* is poorly-defined. We chose to associate a state with values of global variables, and a state transition with changes of values of global variables. Perhaps a more flexible way to determine the granularity of state transitions is more appropriate.

5. Our model checker returns *Maybe* if it cannot determine whether a property holds in the system. Tightening up reasoning about abstract values and/or choosing property-specific abstractions should reduce the number of cases for which verification is inconclusive.
6. When a property is determined to be *No* or *Maybe*, the AMC also returns information about values of global variables in the states where property is violated or *Maybe* violated. This is not sufficient for the users to understand what the counter example is in the original system.

## 5.4 Future Work

In short-term future work:

- We hope to extend our model-checker to reasoning about CTL\* [19] which combines branching-time and linear-time operators and is strictly more expressive than CTL.
- We would like to address the issue of state granularity. We can do so by either asking users to specify which global variables constitute a “state” or to add language constructs for explicitly stating the beginning and the end of each state, either via *begin-state/end-state* or via adding the notion of time (*time-tick*), where each state occurs between consecutive time-ticks.
- We would like to develop more case studies, i.e., to demonstrate our analysis technique on some industrial-size non-trivial systems.

In long-term future work:

- We would like to extend our input language to handle concurrent systems.
- We want to produce more useful counter examples when the AMC detects a (possible) violation of the property being verified. We can do so by translating the counter examples in the abstract system into ones corresponding to the original

system. In this case, the user's understanding of the verification will be in terms of the original system rather than the abstract model.

- We expect to build different levels of abstractions where if the analysis is inconclusive, then more expensive techniques can be applied until the property is verified or all resources are exhausted.
- We would like to measure the effectiveness of our framework, that is, to classify the existing verification techniques for analyzing reactive systems, assigning these techniques levels from 1 and on (depicted in Figure 5.1) based on the effectiveness and the cost of their verification techniques.

# Appendix A

## Proofs

In this Appendix, we give proofs of correctness of Theorem 3 and Theorem 4.

**Theorem 3** *Our model checker is sound with respect to both Yes and No answers.*

Let  $(AM)_T(\phi, M^\alpha, s^\alpha)$  ( $(AM)_F(\phi, M^\alpha, s^\alpha)$ ) indicate that our model checker returns *Yes* (*No*) when checking a formula  $\phi$  in state  $s^\alpha$  of the abstract model  $M^\alpha$ .

Assume:

$$(AM)_T(\varphi, M^\alpha, s^\alpha) \Rightarrow M, s \models \varphi \tag{A.1}$$

$$(AM)_F(\varphi, M^\alpha, s^\alpha) \Rightarrow M, s \not\models \varphi \tag{A.2}$$

$$(AM)_T(\psi, M^\alpha, s^\alpha) \Rightarrow M, s \models \psi \tag{A.3}$$

$$(AM)_F(\psi, M^\alpha, s^\alpha) \Rightarrow M, s \not\models \psi \tag{A.4}$$

where  $M, s$  are the model and the state of the concrete program, respectively. The above expressions state that our model checker is sound with respect to *Yes* and *No* answers for  $\varphi$  and  $\psi$ .

Prove:

$$(AM)_T(\neg\varphi, M^\alpha, s^\alpha) \Rightarrow M, s \models \neg\varphi \quad (\text{A.5})$$

$$(AM)_F(\neg\varphi, M^\alpha, s^\alpha) \Rightarrow M, s \not\models \neg\varphi \quad (\text{A.6})$$

$$(AM)_T(\varphi \wedge \psi, M^\alpha, s^\alpha) \Rightarrow M, s \models \varphi \wedge \psi \quad (\text{A.7})$$

$$(AM)_F(\varphi \wedge \psi, M^\alpha, s^\alpha) \Rightarrow M, s \not\models \varphi \wedge \psi \quad (\text{A.8})$$

$$(AM)_T(\varphi \vee \psi, M^\alpha, s^\alpha) \Rightarrow M, s \models \varphi \vee \psi \quad (\text{A.9})$$

$$(AM)_F(\varphi \vee \psi, M^\alpha, s^\alpha) \Rightarrow M, s \not\models \varphi \vee \psi \quad (\text{A.10})$$

$$(AM)_T(EX\varphi, M^\alpha, s^\alpha) \Rightarrow M, s \models EX\varphi \quad (\text{A.11})$$

$$(AM)_F(EX\varphi, M^\alpha, s^\alpha) \Rightarrow M, s \not\models EX\varphi \quad (\text{A.12})$$

$$(AM)_T(AX\varphi, M^\alpha, s^\alpha) \Rightarrow M, s \models AX\varphi \quad (\text{A.13})$$

$$(AM)_F(AX\varphi, M^\alpha, s^\alpha) \Rightarrow M, s \not\models AX\varphi \quad (\text{A.14})$$

$$(AM)_T(E[\varphi U \psi], M^\alpha, s^\alpha) \Rightarrow M, s \models E[\varphi U \psi] \quad (\text{A.15})$$

$$(AM)_F(E[\varphi U \psi], M^\alpha, s^\alpha) \Rightarrow M, s \not\models E[\varphi U \psi] \quad (\text{A.16})$$

$$(AM)_T(A[\varphi U \psi], M^\alpha, s^\alpha) \Rightarrow M, s \models A[\varphi U \psi] \quad (\text{A.17})$$

$$(AM)_F(A[\varphi U \psi], M^\alpha, s^\alpha) \Rightarrow M, s \not\models A[\varphi U \psi] \quad (\text{A.18})$$

The proof is by induction on the structure of  $\varphi$  and  $\psi$ :

**A.5**  $(AM)_T(\neg\varphi, M^\alpha, s^\alpha) \Rightarrow M, s \models \neg\varphi$

$$\begin{aligned} & (AM)_T(\neg\varphi, M^\alpha, s^\alpha) && \text{;algorithm} \\ \Rightarrow & (AM)_F(\varphi, M^\alpha, s^\alpha) && \text{; (A.2)} \\ \Rightarrow & M, s \not\models \varphi \\ \Rightarrow & M, s \models \neg\varphi \end{aligned}$$

**A.6**  $(AM)_F(\neg\varphi, M^\alpha, s^\alpha) \Rightarrow M, s \not\models \neg\varphi$

$$\begin{aligned} & (AM)_F(\neg\varphi, M^\alpha, s^\alpha) && \text{;algorithm} \\ \Rightarrow & (AM)_T(\varphi, M^\alpha, s^\alpha) && \text{; (A.1)} \\ \Rightarrow & M, s \models \varphi \end{aligned}$$

$$\Rightarrow M, s \not\models \neg\varphi$$

$$\begin{aligned} \mathbf{A.7} \quad (AM)_T(\varphi \wedge \psi, M^\alpha, s^\alpha) &\Rightarrow M, s \models \varphi \wedge \psi \\ (AM)_T(\varphi \wedge \psi, M^\alpha, s^\alpha) & \qquad \qquad \qquad ; \text{ algorithm} \\ \Rightarrow (AM)_T(\varphi, M^\alpha, s^\alpha) \wedge (AM)_T(\psi, M^\alpha, s^\alpha) & \qquad \qquad \qquad ; (\mathbf{A.1}) (\mathbf{A.3}) \\ \Rightarrow M, s \models \varphi \wedge M, s \models \psi & \\ \Rightarrow M, s \models \varphi \wedge \psi & \end{aligned}$$

$$\begin{aligned} \mathbf{A.8} \quad (AM)_F(\varphi \wedge \psi, M^\alpha, s^\alpha) &\Rightarrow M, s \not\models \varphi \wedge \psi \\ (AM)_F(\varphi \wedge \psi, M^\alpha, s^\alpha) & \qquad \qquad \qquad ; \text{ algorithm} \\ \Rightarrow (AM)_F(\varphi, M^\alpha, s^\alpha) \vee (AM)_F(\psi, M^\alpha, s^\alpha) & \qquad \qquad \qquad ; (\mathbf{A.2}) (\mathbf{A.4}) \\ \Rightarrow M, s \not\models \varphi \vee M, s \not\models \psi & \\ \Rightarrow M, s \not\models \varphi \wedge \psi & \end{aligned}$$

$$\begin{aligned} \mathbf{A.9} \quad (AM)_T(\varphi \vee \psi, M^\alpha, s^\alpha) &\Rightarrow M, s \models \varphi \vee \psi \\ (AM)_T(\varphi \vee \psi, M^\alpha, s^\alpha) & \qquad \qquad \qquad ; \text{ algorithm} \\ \Rightarrow (AM)_T(\varphi, M^\alpha, s^\alpha) \vee (AM)_T(\psi, M^\alpha, s^\alpha) & \qquad \qquad \qquad ; (\mathbf{A.1}) (\mathbf{A.3}) \\ \Rightarrow M, s \models \varphi \vee M, s \models \psi & \\ \Rightarrow M, s \models \varphi \vee \psi & \end{aligned}$$

$$\begin{aligned} \mathbf{A.10} \quad (AM)_F(\varphi \vee \psi, M^\alpha, s^\alpha) &\Rightarrow M, s \not\models \varphi \vee \psi \\ (AM)_F(\varphi \vee \psi, M^\alpha, s^\alpha) & \qquad \qquad \qquad ; \text{ algorithm} \\ \Rightarrow (AM)_F(\varphi, M^\alpha, s^\alpha) \wedge (AM)_F(\psi, M^\alpha, s^\alpha) & \qquad \qquad \qquad ; (\mathbf{A.2}) (\mathbf{A.4}) \\ \Rightarrow M, s \not\models \varphi \wedge M, s \not\models \psi & \\ \Rightarrow M, s \not\models \varphi \vee \psi & \end{aligned}$$

$$\begin{aligned} \mathbf{A.11} \quad (AM)_T(EX\varphi, M^\alpha, s^\alpha) &\Rightarrow M, s \models EX\varphi \\ (AM)_T(EX\varphi, M^\alpha, s^\alpha) & \qquad \qquad \qquad ; \text{ algorithm} \\ \Rightarrow \exists t^\alpha, s^\alpha = \text{pred}(t^\alpha) \wedge (AM)_T(\varphi, M^\alpha, t^\alpha) & \qquad \qquad \qquad ; (\mathbf{A.1}) \end{aligned}$$

$$\begin{aligned} &\Rightarrow \exists t, s = \text{pred}(t) \wedge M, t \models \varphi \\ &\Rightarrow M, s \models EX\varphi \end{aligned}$$

**A.12**  $(AM)_F(EX\varphi, M^\alpha, s^\alpha) \Rightarrow M, s \not\models EX\varphi$

**DEF A.12:** Let  $T^\alpha$  be the set of states such that  $\forall t^\alpha \in T^\alpha, (AM)_F(\varphi, M^\alpha, t^\alpha)$ .

$$\begin{aligned} &(AM)_F(EX\varphi, M^\alpha, s^\alpha) && \text{; algorithm} \\ &\Rightarrow s^\alpha \in S^\alpha \wedge s^\alpha \notin \text{pred}(S^\alpha - T^\alpha) \\ &\Rightarrow \forall t^\alpha, s^\alpha \in \text{pred}(t^\alpha) \Rightarrow t^\alpha \in T^\alpha && \text{; (DEF A.12)} \\ &\Rightarrow \forall t^\alpha, s^\alpha \in \text{pred}(t^\alpha) \Rightarrow (AM)_F(\varphi, M^\alpha, t^\alpha) && \text{; (A.2)} \\ &\Rightarrow \forall t, s \in \text{pred}(t) \Rightarrow M, t \not\models \varphi \\ &\Rightarrow M, s \not\models EX\varphi \end{aligned}$$

**A.13**  $(AM)_T(AX\varphi, M^\alpha, s^\alpha) \Rightarrow M, s \models AX\varphi$

**DEF A.13:** Let  $T^\alpha$  be the set of states such that  $\forall t^\alpha \in T^\alpha, (AM)_T(\varphi, M^\alpha, t^\alpha)$ .

$$\begin{aligned} &(AM)_T(AX\varphi, M^\alpha, s^\alpha) && \text{; algorithm} \\ &\Rightarrow s^\alpha \in S^\alpha \wedge s^\alpha \notin \text{pred}(S^\alpha - T^\alpha) \\ &\Rightarrow \forall t^\alpha, s^\alpha \in \text{pred}(t^\alpha) \Rightarrow t^\alpha \in T^\alpha && \text{; (DEF A.13)} \\ &\Rightarrow \forall t^\alpha, s^\alpha \in \text{pred}(t^\alpha) \Rightarrow (AM)_T(\varphi, M^\alpha, t^\alpha) && \text{; (A.1)} \\ &\Rightarrow \forall t, s \in \text{pred}(t) \Rightarrow M, t \models \varphi \\ &\Rightarrow M, s \models AX\varphi \end{aligned}$$

**A.14**  $(AM)_F(AX\varphi, M^\alpha, s^\alpha) \Rightarrow M, s \not\models AX\varphi$

**DEF A.14:** Let  $T^\alpha$  be the set of states such that  $\forall t^\alpha \in T^\alpha, (AM)_F(\varphi, M^\alpha, t^\alpha)$ .

$$\begin{aligned} &(AM)_F(AX\varphi, M^\alpha, s^\alpha) && \text{; algorithm} \\ &\Rightarrow s^\alpha \in \text{pred}(T^\alpha) \\ &\Rightarrow \exists t^\alpha, t^\alpha \in T^\alpha \wedge s^\alpha \in \text{pred}(t^\alpha) && \text{; (DEF A.14)} \\ &\Rightarrow \exists t^\alpha, s^\alpha \in \text{pred}(t^\alpha) \wedge (AM)_F(\varphi, M^\alpha, t^\alpha) && \text{; (A.2)} \\ &\Rightarrow \exists t, s \in \text{pred}(t) \wedge M, t \not\models \varphi \\ &\Rightarrow M, s \not\models AX\varphi \end{aligned}$$

**A.15**  $(AM)_T(E[\varphi U\psi], M^\alpha, s^\alpha) \Rightarrow M, s \models E[\varphi U\psi]$

The proof is by induction on the length of the path from  $s^\alpha$  to a state in which  $\psi$  holds.

**Base Case:** length = 0.

$$\begin{aligned}
& (AM)_T(E[\varphi U\psi], M^\alpha, s^\alpha) \\
& \Rightarrow (AM)_T(\psi, M^\alpha, s^\alpha) && \text{; (A.3)} \\
& \Rightarrow M, s \models \psi \\
& \Rightarrow M, s \models E[\varphi U\psi]
\end{aligned}$$

**IH:** Let  $T^\alpha$  be the set of states from which  $\psi$  can be reached in  $< n$  steps.

$$\text{Assume } \forall t^\alpha \in T^\alpha, (AM)_T(E[\varphi U\psi], M^\alpha, t^\alpha) \Rightarrow M, t \models E[\varphi U\psi]$$

**Prove:** The formula holds for paths of length  $\leq n$ .

$$\begin{aligned}
& (AM)_T(E[\varphi U\psi], M^\alpha, s^\alpha) && \text{; algorithm} \\
& \Rightarrow (AM)_T(\psi, M^\alpha, s^\alpha) \vee ((AM)_T(\varphi, M^\alpha, s^\alpha) \wedge s^\alpha \in \text{pred}(T^\alpha)) && \text{;(BASE CASE)} \\
& \Rightarrow M, s \models E[\varphi U\psi] \vee ((AM)_T(\varphi, M^\alpha, s^\alpha) \wedge (\exists t^\alpha \in T^\alpha, s^\alpha \in \text{pred}(T^\alpha))) && \text{; (IH), (A.1)} \\
& \Rightarrow M, s \models E[\varphi U\psi] \vee (M, s \models \varphi \wedge s \in \text{pred}(t) \wedge M, t \models E[\varphi U\psi]) \\
& \Rightarrow M, s \models E[\varphi U\psi]
\end{aligned}$$

**A.16**  $(AM)_F(E[\varphi U\psi], M^\alpha, s^\alpha) \Rightarrow M, s \not\models E[\varphi U\psi]$

$E[\varphi U\psi]$  does not hold in state  $s^\alpha$  if on every path emanating from  $s^\alpha$  either (1)  $\psi$  does not occur or (2) before the first occurrence of  $\psi$ , there is a state in which  $\varphi$  does not hold.

The algorithm can be expanded as follows:

$$\begin{aligned}
& T_0 = \text{No}(\psi) \\
& \text{Case (1): } T_{i+1} = T_i \cap (S^\alpha - \text{pred}(S^\alpha - T_i)) \\
& \text{Until } T_n = T_{n+1} \\
& \text{Case (2): } Q = T_0 \cap \text{No}(\varphi) \\
& \text{No}(E[\varphi U\psi]) = T_n \cup Q
\end{aligned}$$

Case (1) starts from the set of states where  $\neg\psi$  is true and

recursively intersects it with those states that do not have successors in which  $\psi$  holds. The result is the set of states that can never reach  $\psi$ , i.e.  $M, s \models AG(\neg\psi) \Rightarrow M, s \not\models E[\varphi U \psi]$ .

Case (2) results in the set of states in which neither  $\varphi$  nor  $\psi$  hold, i.e.  $M, s \models (\neg\varphi \wedge \neg\psi) \Rightarrow M, s \not\models E[\varphi U \psi]$ .

**A.17**  $(AM)_T(A[\varphi U \psi], M^\alpha, s^\alpha) \Rightarrow M, s \models A[\varphi U \psi]$

The proof is by induction on the length of the path from  $s^\alpha$  to a state in which  $\psi$  holds.

**Base Case:** length = 0.

$$\begin{aligned} & (AM)_T(A[\varphi U \psi], M^\alpha, s^\alpha) \\ & \Rightarrow (AM)_T(\psi, M^\alpha, s^\alpha) && ; \text{(A.3)} \\ & \Rightarrow M, s \models \psi \\ & \Rightarrow M, s \models A[\varphi U \psi] \end{aligned}$$

**IH:** Let  $T^\alpha$  be the set of states from which  $\psi$  can be reached in  $< n$  steps.

$$\text{Assume } \forall t^\alpha \in T^\alpha, (AM)_T(A[\varphi U \psi], M^\alpha, t^\alpha) \Rightarrow M, t \models A[\varphi U \psi]$$

**Prove:** The formula holds for paths of length  $\leq n$ .

$$\begin{aligned} & (AM)_T(A[\varphi U \psi], M^\alpha, s^\alpha) && ; \text{algorithm} \\ & \Rightarrow (AM)_T(\psi, M^\alpha, s^\alpha) \vee \\ & \quad ((AM)_T(\varphi, M^\alpha, s^\alpha) \wedge s^\alpha \in \text{pred}(T^\alpha) \wedge s^\alpha \notin \text{pred}(S^\alpha - T^\alpha)) && ; \text{(BASE CASE)} \\ & \Rightarrow M, s \models A[\varphi U \psi] \vee \\ & \quad ((AM)_T(\varphi, M^\alpha, s^\alpha) \wedge s^\alpha \in \text{pred}(T^\alpha) \wedge s^\alpha \notin \text{pred}(S^\alpha - T^\alpha)) \\ & \Rightarrow M, s \models A[\varphi U \psi] \vee \\ & \quad ((AM)_T(\varphi, M^\alpha, s^\alpha) \wedge (\forall t^\alpha, s^\alpha \in \text{pred}(t^\alpha) \Rightarrow (t^\alpha \in T^\alpha))) && ; \text{(IH), (A.1)} \\ & \Rightarrow M, s \models A[\varphi U \psi] \vee \\ & \quad (M, s \models \varphi \wedge (\forall t, s \in \text{pred}(t) \Rightarrow M, t \models A[\varphi U \psi])) \\ & \Rightarrow M, s \models A[\varphi U \psi] \end{aligned}$$

**A.18**  $(AM)_F(A[\varphi U \psi], M^\alpha, s^\alpha) \Rightarrow M, s \not\models A[\varphi U \psi]$

$A[\varphi U \psi]$  does not hold in state  $s^\alpha$  if either (1)  $\psi$  does not occur on some path emanating from  $s^\alpha$ , or (2) on some path emanating from  $s^\alpha$ , before the first occurrence of  $\psi$ , there is a state in which  $\varphi$  does not hold.

The algorithm can be expanded as follows:

$$T_0 = \text{No}(\psi)$$

$$\text{Case (1): } T_{i+1} = T_i \cap \text{pred}(T_i)$$

$$\text{Until } T_n = T_{n+1}$$

$$\text{Case (2): } Q = T_0 \cap \text{No}(\varphi)$$

$$\text{No}(A[\varphi U \psi]) = T_n \cup Q$$

Case (1) starts from the set of states where  $\neg\psi$  is true and recursively intersects them with those states that have successors in which  $\neg\psi$  holds. The result is the set of states that can lead a path which never reaches  $\psi$ , i.e.  $M, s \models EG(\neg\psi) \Rightarrow M, s \not\models A[\varphi U \psi]$ .

Case (2) results in the set of states in which neither  $\varphi$  nor  $\psi$  hold, i.e.

$$M, s \models (\neg\varphi \wedge \neg\psi) \Rightarrow M, s \not\models A[\varphi U \psi]. \quad \square$$

**Theorem 4** *The analysis of UNDEF is sound with respect to both Yes and No answers.*

Assume  $p(q)$  is an atomic proposition in which every variable is defined, and  $\varphi(\psi)$  is the same atomic proposition except that at least one of its variables is undefined. *UNDEF* should not effect our analysis, and the appropriate way of treatment is to “ignore” it. This means that  $p(q)$  holds in state  $s^\alpha$  implies that  $\varphi(\psi)$  also holds in state  $s^\alpha$ ;  $p(q)$  does not hold in state  $s^\alpha$  implies that  $\varphi(\psi)$  also does not hold in state  $s^\alpha$ . Moreover,  $EXp$  ( $E[pUq]$ ) does not hold in state  $s^\alpha$  implies that  $EX\varphi$  ( $E[\varphi U \psi]$ ) does not hold in  $s^\alpha$ ; and  $AXp$  ( $A[pUq]$ ) holds in state  $s^\alpha$  implies that  $AX\varphi$  ( $A[\varphi U \psi]$ ) holds in  $s^\alpha$ . Intuitively, for  $EXp$ , we need a “witness” to the formula  $p$  in a next state, but  $\varphi$  cannot be a witnesses since  $\varphi$  is *UNDEF*.  $AXp$  is dual to  $EXp$ . For  $E[pUq]$ , we need a “witness” to a path in which  $p$  holds until  $q$  becomes *True*, but  $\varphi$  and  $\psi$  cannot be the witness since they are undefined, and  $A[pUq]$  is dual. The proofs of (A.23)-(A.32) are sufficient to show that

our analysis of *UNDEF* is sound because all other CTL formulas can be defined using  $p$  ( $q$ ),  $\neg p$  ( $\neg q$ ),  $EXp$  ( $EXq$ ),  $AXp$  ( $AXq$ ),  $E[pUq]$  and  $A[pUq]$ .

Assume:

$$(AM)_T(p, M^\alpha, s^\alpha) \Rightarrow (AM)_T(\varphi, M^\alpha, s^\alpha) \quad (\text{A.19})$$

$$(AM)_F(p, M^\alpha, s^\alpha) \Rightarrow (AM)_F(\varphi, M^\alpha, s^\alpha) \quad (\text{A.20})$$

$$(AM)_T(q, M^\alpha, s^\alpha) \Rightarrow (AM)_T(\psi, M^\alpha, s^\alpha) \quad (\text{A.21})$$

$$(AM)_F(q, M^\alpha, s^\alpha) \Rightarrow (AM)_F(\psi, M^\alpha, s^\alpha) \quad (\text{A.22})$$

The above expressions state that our analysis of *UNDEF* is sound with respect to *Yes* and *No* answers for atomic propositions.

Prove:

$$(AM)_T(\neg p, M^\alpha, s^\alpha) \Rightarrow (AM)_F(\neg\varphi, M^\alpha, s^\alpha) \quad (\text{A.23})$$

$$(AM)_F(\neg p, M^\alpha, s^\alpha) \Rightarrow (AM)_F(\neg\varphi, M^\alpha, s^\alpha) \quad (\text{A.24})$$

$$(AM)_T(p \wedge q, M^\alpha, s^\alpha) \Rightarrow (AM)_T(\varphi \wedge \psi, M^\alpha, s^\alpha) \quad (\text{A.25})$$

$$(AM)_F(p \wedge q, M^\alpha, s^\alpha) \Rightarrow (AM)_F(\varphi \wedge \psi, M^\alpha, s^\alpha) \quad (\text{A.26})$$

$$(AM)_T(p \vee q, M^\alpha, s^\alpha) \Rightarrow (AM)_T(\varphi \vee \psi, M^\alpha, s^\alpha) \quad (\text{A.27})$$

$$(AM)_F(p \vee q, M^\alpha, s^\alpha) \Rightarrow (AM)_F(\varphi \vee \psi, M^\alpha, s^\alpha) \quad (\text{A.28})$$

$$(AM)_F(EXp, M^\alpha, s^\alpha) \Rightarrow (AM)_F(EX\varphi, M^\alpha, s^\alpha) \quad (\text{A.29})$$

$$(AM)_T(AXp, M^\alpha, s^\alpha) \Rightarrow (AM)_T(AX\varphi, M^\alpha, s^\alpha) \quad (\text{A.30})$$

$$(AM)_F(E[pUq], M^\alpha, s^\alpha) \Rightarrow (AM)_F(E[\varphi U\psi], M^\alpha, s^\alpha) \quad (\text{A.31})$$

$$(AM)_T(A[pUq], M^\alpha, s^\alpha) \Rightarrow (AM)_T(A[\varphi U\psi], M^\alpha, s^\alpha) \quad (\text{A.32})$$

The proof is by induction on the structure of  $\varphi$  and  $\psi$ :

**A.23**  $(AM)_T(\neg p, M^\alpha, s^\alpha) \Rightarrow (AM)_F(\neg\varphi, M^\alpha, s^\alpha)$

$$(AM)_T(\neg p, M^\alpha, s^\alpha)$$

$$\Rightarrow (AM)_F(p, M^\alpha, s^\alpha)$$

;algorithm

;(A.20)

$$\begin{aligned} &\Rightarrow (AM)_F(\varphi, M^\alpha, s^\alpha) \\ &\Rightarrow (AM)_T(\neg\varphi, M^\alpha, s^\alpha) \end{aligned}$$

$$\mathbf{A.24} \quad (AM)_F(\neg p, M^\alpha, s^\alpha) \Rightarrow (AM)_F(\neg\varphi M^\alpha, s^\alpha)$$

$$\begin{aligned} &(AM)_F(\neg p, M^\alpha, s^\alpha) \\ &\Rightarrow (AM)_T(p, M^\alpha, s^\alpha) \\ &\Rightarrow (AM)_T(\varphi, M^\alpha, s^\alpha) \\ &\Rightarrow (AM)_F(\neg\varphi, M^\alpha, s^\alpha) \end{aligned}$$

;algorithm  
;(A.19)

$$\mathbf{A.25} \quad (AM)_T(p \wedge q, M^\alpha, s^\alpha) \Rightarrow (AM)_T(\varphi \wedge \psi, M^\alpha, s^\alpha)$$

$$\begin{aligned} &(AM)_T(p \wedge q, M^\alpha, s^\alpha) \\ &\Rightarrow (AM)_T(p, M^\alpha, s^\alpha) \wedge (AM)_T(q, M^\alpha, s^\alpha) \\ &\Rightarrow (AM)_T(\varphi, M^\alpha, s^\alpha) \wedge (AM)_T(\psi, M^\alpha, s^\alpha) \\ &\Rightarrow (AM)_T(\varphi \wedge \psi, M^\alpha, s^\alpha) \end{aligned}$$

;algorithm  
;(A.19) (A.21)

$$\mathbf{A.26} \quad (AM)_F(p \wedge q, M^\alpha, s^\alpha) \Rightarrow (AM)_F(\varphi \wedge \psi, M^\alpha, s^\alpha)$$

$$\begin{aligned} &(AM)_F(p \wedge q, M^\alpha, s^\alpha) \\ &\Rightarrow (AM)_F(p, M^\alpha, s^\alpha) \vee (AM)_F(q, M^\alpha, s^\alpha) \\ &\Rightarrow (AM)_F(\varphi, M^\alpha, s^\alpha) \vee (AM)_F(\psi, M^\alpha, s^\alpha) \\ &\Rightarrow (AM)_F(\varphi \wedge \psi, M^\alpha, s^\alpha) \end{aligned}$$

;algorithm  
;(A.20) (A.22)

$$\mathbf{A.27} \quad (AM)_T(p \vee q, M^\alpha, s^\alpha) \Rightarrow (AM)_T(\varphi \vee \psi, M^\alpha, s^\alpha)$$

$$\begin{aligned} &(AM)_T(p \vee q, M^\alpha, s^\alpha) \\ &\Rightarrow (AM)_T(p, M^\alpha, s^\alpha) \vee (AM)_T(q, M^\alpha, s^\alpha) \\ &\Rightarrow (AM)_T(\varphi, M^\alpha, s^\alpha) \vee (AM)_T(\psi, M^\alpha, s^\alpha) \\ &\Rightarrow (AM)_T(\varphi \vee \psi, M^\alpha, s^\alpha) \end{aligned}$$

;algorithm  
;(A.19) (A.21)

$$\mathbf{A.28} \quad (AM)_F(p \vee q, M^\alpha, s^\alpha) \Rightarrow (AM)_F(\varphi \vee \psi, M^\alpha, s^\alpha)$$

$$(AM)_F(p \vee q, M^\alpha, s^\alpha)$$

;algorithm

$$\begin{aligned}
&\Rightarrow (AM)_F(p, M^\alpha, s^\alpha) \wedge (AM)_F(q, M^\alpha, s^\alpha) && ;(\mathbf{A.20}) \ (\mathbf{A.22}) \\
&\Rightarrow (AM)_F(\varphi, M^\alpha, s^\alpha) \wedge (AM)_F(\psi, M^\alpha, s^\alpha) \\
&\Rightarrow (AM)_F(\varphi \vee \psi, M^\alpha, s^\alpha)
\end{aligned}$$

**A.29**  $(AM)_F(EXp, M^\alpha, s^\alpha) \Rightarrow (AM)_F(EX\varphi, M^\alpha, s^\alpha)$

**DEF A.29:** Let  $T^\alpha$  be the set of states such that  $\forall t^\alpha \in T^\alpha, (AM)_F(p, M^\alpha, t^\alpha)$ .

$$\begin{aligned}
&(AM)_F(EXp, M^\alpha, s^\alpha) && ;\text{algorithm} \\
&\Rightarrow s^\alpha \in S^\alpha \wedge s^\alpha \notin \text{pred}(S^\alpha - T^\alpha) \\
&\Rightarrow \forall t^\alpha, s^\alpha \in \text{pred}(t^\alpha) \Rightarrow t^\alpha \in T^\alpha && ; (\mathbf{DEF A.29}) \\
&\Rightarrow \forall t^\alpha, s^\alpha \in \text{pred}(t^\alpha) \Rightarrow (AM)_F(p, M^\alpha, t^\alpha) && ; (\mathbf{A.20}) \\
&\Rightarrow \forall t^\alpha, s^\alpha \in \text{pred}(t^\alpha) \Rightarrow (AM)_F(\varphi, M^\alpha, t^\alpha) \\
&\Rightarrow (AM)_F(EX\varphi, M^\alpha, s^\alpha)
\end{aligned}$$

**A.30**  $(AM)_T(AXp, M^\alpha, s^\alpha) \Rightarrow (AM)_T(AX\varphi, M^\alpha, s^\alpha)$

**DEF A.30** Let  $T^\alpha$  be the set of states such that  $\forall t^\alpha \in T^\alpha, (AM)_T(p, M^\alpha, t^\alpha)$ .

$$\begin{aligned}
&(AM)_T(AXp, M^\alpha, s^\alpha) && ;\text{algorithm} \\
&\Rightarrow s^\alpha \in S^\alpha \wedge s^\alpha \notin \text{pred}(S^\alpha - T^\alpha) \\
&\Rightarrow \forall t^\alpha, s^\alpha \in \text{pred}(t^\alpha) \Rightarrow t^\alpha \in T^\alpha && ; (\mathbf{DEF A.30}) \\
&\Rightarrow \forall t^\alpha, s^\alpha \in \text{pred}(t^\alpha) \Rightarrow (AM)_T(p, M^\alpha, t^\alpha) && ; (\mathbf{A.19}) \\
&\Rightarrow \forall t^\alpha, s^\alpha \in \text{pred}(t^\alpha) \Rightarrow (AM)_T(\varphi, M^\alpha, t^\alpha) \\
&\Rightarrow (AM)_T(AX\varphi, M^\alpha, s^\alpha)
\end{aligned}$$

**A.31**  $(AM)_F(E[pUq], M^\alpha, s^\alpha) \Rightarrow (AM)_F(E[\varphi U\psi], M^\alpha, s^\alpha)$

$E[pUq]$  does not hold in state  $s^\alpha$  if on every path emanating from  $s^\alpha$  either (1)  $q$  does not occur or (2) before the first occurrence of  $q$ , there is a state in which  $p$  does not hold. The algorithm can be expanded as follows:

$$T_0 = \text{No}(q)$$

$$\text{Case (1): } T_{i+1} = T_i \cap (S^\alpha - \text{pred}(S^\alpha - T_i))$$

$$\text{Until } T_n = T_{n+1}$$

Case (2):  $Q = T_0 \cap \text{No}(p)$

$$\text{No}(E[pUq]) = T_n \cup Q$$

Case (1) starts from the set of states where  $\neg q$  is *True* and recursively intersects it with those states that do not have successors in which  $q$  holds.

The result is the set of states that can never reach  $q$ . From **(A.22)**,

we know that the resulting set of states also satisfy the property that they can never reach  $\psi$ , i.e.,  $(AM)_F(E[\varphi U \psi], M^\alpha, s^\alpha)$

Case (2) results in the set of states in which neither  $p$  nor  $q$  hold.

From **(A.20)** **(A.22)**, we know that in the resulting states, neither  $\varphi$  nor  $\psi$  hold, i.e.,  $(AM)_T(\neg\varphi \wedge \neg\psi, M^\alpha, s^\alpha) \Rightarrow (AM)_F(E[\varphi U \psi], M^\alpha, s^\alpha)$ .

$$\mathbf{A.32} \quad (AM)_T(A[pUq], M^\alpha, s^\alpha) \Rightarrow (AM)_T(A[\varphi U \psi], M^\alpha, s^\alpha)$$

The proof is by induction on the length of the path from  $s^\alpha$  to a state in which  $q$  holds.

**Base Case:** length = 0.

$$\begin{aligned} & (AM)_T(A[pUq], M^\alpha, s^\alpha) \\ & \Rightarrow (AM)_T(q, M^\alpha, s^\alpha) && \text{; (A.21)} \\ & \Rightarrow (AM)_T(\psi, M^\alpha, s^\alpha) \\ & \Rightarrow (AM)_T(A[\varphi U \psi], M^\alpha, s^\alpha) \end{aligned}$$

**IH:** Let  $T^\alpha$  be the set of states from which  $q$  can be reached in  $< n$  steps.

$$\text{Assume } \forall t^\alpha \in T^\alpha, (AM)_T(A[pUq], M^\alpha, t^\alpha) \Rightarrow (AM)_T(A[\varphi U \psi], M^\alpha, t^\alpha)$$

**Prove:** The formula holds for paths of length  $\leq n$ .

$$\begin{aligned} & (AM)_T(A[pUq], M^\alpha, s^\alpha) && \text{; algorithm} \\ & \Rightarrow (AM)_T(q, M^\alpha, s^\alpha) \vee \\ & \quad ((AM)_T(p, M^\alpha, s^\alpha) \wedge s^\alpha \in \text{pred}(T) \wedge s^\alpha \notin \text{pred}(S^\alpha - T)) && \text{;(BASE CASE)} \\ & \Rightarrow (AM)_T(A[\varphi U \psi], M^\alpha, s^\alpha) \vee \\ & \quad ((AM)_T(p, M^\alpha, s^\alpha) \wedge s^\alpha \in \text{pred}(T) \wedge s^\alpha \notin \text{pred}(S^\alpha - T)) && \text{(A.19)} \\ & \Rightarrow (AM)_T(A[\varphi U \psi], M^\alpha, s^\alpha) \vee \\ & \quad ((AM)_T(\varphi, M^\alpha, s^\alpha) \wedge s^\alpha \in \text{pred}(T) \wedge s^\alpha \notin \text{pred}(S^\alpha - T)) && \text{;(IH)} \end{aligned}$$

$$\Rightarrow (AM)_T(A[\varphi U\psi], M^\alpha, s^\alpha)$$

□

# Appendix B

## Grammar Of Input Language

Below is the complete grammar of our input language. The grammar is LALR(1) which enables the use of Lex and Yacc tools for automatic parsing. The grammar has undefined terminals symbols *integer-constant*, *character-constant*, *floating-constant* and *identifier*. The `typewriter` style words and symbols are terminals given literally. It is necessary to duplicate each production with an *opt* symbol depending on the rules of the parser generator.

*translation-unit:*

*external-declaration*

| *translation-unit external-declaration*

*external-declaration:*

*function-definition*

| *declaration*

*function-definition:*

*declaration-specifiers<sub>opt</sub> declarator declaration-list<sub>opt</sub> compound-statement*

*declaration:*

*declaration-specifiers declarator-list*

*declaration-list:*

*declaration*

| *declaration-list declaration*

*declaration-specifiers:*

*type-specifier declaration-specifiers*<sub>opt</sub>

*type-specifier:*

`void | char | int | float`

*declarator-list:*

*declarator*

| *declarator-list, declarator*

*declarator:*

*direct-declarator*

*direct-declarator:*

*identifier*

| (*declarator*)

| *direct-declarator* (*parameter-type-list*)

| *direct-declarator* (*identifier-list*<sub>opt</sub>)

*parameter-type-list:*

*parameter-list*

| *parameter-list, parameter-type-list*

*parameter-list:*

*parameter-declaration*

| *parameter-list, parameter-declaration*

*parameter-declaration:*

*declaration-specifiers declarator*

*identifier-list:*

*identifier*

| *identifier-list, identifier*

*statement:*

*expression-statement*

| *compound-statement*

| *selection-statement*

$|$ *iteration-statement*  
*expression-statement*:  
 $expression_{opt};$   
*compound-statement*:  
 $\{ declaration-list_{opt} statement-list_{opt} \}$   
*statement-list*:  
 $statement$   
 $| statement-list statement$   
*selection-statement*:  
 $if (expression) statement$   
 $| if (expression) statement \mathbf{else} statement$   
*iteration-statement*:  
 $while (expression) statement$   
*jump-statement*:  
 $break ;$   
 $| return expression_{opt};$   
*expression*:  
 $assignment-expression$   
 $| expression, assignment-expression$   
*assignment-expression*:  
 $conditional-expression$   
 $unary-expression assignment-operator assignment-expression$   
*assignment-operator*:  
 $=$   
*conditional-expression*:  
 $logical-OR-expression$   
*constant-expression*:  
 $conditional-expression$   
*logical-OR-expression*:

*logical-AND-expression*

| *logical-AND-expression* || *logical-AND-expression*

*logical-AND-expression:*

*logical-AND-expression* && *AND-expression*

*AND-expression:*

*equality-expression*

| *AND-expression* & *equality-expression*

*equality-expression:*

*relational-expression*

| *equality-expression* == *relational-expression*

| *equality-expression* != *relational-expression*

*relational-expression:*

*relational-expression* < *additive-expression*

| *relational-expression* > *additive-expression*

| *relational-expression* <= *additive-expression*

| *relational-expression* >= *additive-expression*

*additive-expression:*

*multiplicative-expression*

| *additive-expression* + *multiplicative-expression*

| *additive-expression* - *multiplicative-expression*

*multiplicative-expression:*

*multiplicative-expression* \* *unary-expression*

| *multiplicative-expression* / *unary-expression*

| *multiplicative-expression* % *unary-expression*

*unary-expression:*

*primary-expression*

| *unary-operator* *unary-expression*

| *unary-expression* (*argument-expression-list*<sub>opt</sub>)

*unary-operator:*

+ | - | !

*primary-expression:*

*identifier*

| *constant*

| *string*

| *(expression)*

*constant:*

*integer-constant*

| *character-constant*

| *floating-constant*



# Appendix C

## Grammar Of Property Specification Language

Below is a recapitulation of the grammar we use for specifying CTL properties. The grammar has undefined terminals symbols *integer-constant*, *character-constant*, *floating-constant* and *identifier*. The typewriter style words and symbols are terminals given literally.

*property:*

*expression*  
| *expression* ; *property*

*expression:*

*assignment-expression*  
| *additive-expression*  
| *relational-expression*  
| *logical-expression*  
| *ctl-expression*  
| *unary-expression*  
| *primary-expression*

*relational-expression:*

*expression* < *expression*

$| \textit{expression} > \textit{expression}$   
 $| \textit{expression} <= \textit{expression}$   
 $| \textit{expression} >= \textit{expression}$   
 $| \textit{expression} = \textit{expression}$   
 $| \textit{expression} != \textit{expression}$

*additive-expression:*

$\textit{expression} + \textit{expression}$   
 $| \textit{expression} - \textit{expression}$

*unary-operator: one of*

$+ \quad | \quad - \quad | \quad !$

*ctl-expression:*

$\text{AX } \textit{expression}$   
 $| \text{EX } \textit{expression}$   
 $| \textit{expression} \text{ UNTIL } \textit{expression}$   
 $| \text{AU}[\textit{expression}, \textit{expression}]$   
 $| \text{EU}[\textit{expression}, \textit{expression}]$   
 $| \textit{expression} \text{ IMP } \textit{expression}$   
 $| \text{AF } \textit{expression}$   
 $| \text{EF } \textit{expression}$   
 $| \text{AG } \textit{expression}$

*unary-expression:*

$\textit{primary-expression}$   
 $| \textit{unary-operator } \textit{expression}$

*logical-relation:*

$\textit{expression} \text{ | } \textit{expression}$   
 $| \textit{expression} \text{ \& } \textit{expression}$

*primary-expression:*

$\textit{identifier}$   
 $| \textit{constant}$

| (*expression*)

*IMP:*

->

*constant:*

| *integer-constant*

| *character-constant*

| *floating-constant*



# Appendix D

## Grammar Of TestCase Specification Language

Here is the complete grammar of input testcases language. The grammar has undefined terminals symbols *ICONST*, *FCONST* and *ID*. The typewriter style words and symbols are terminals given literally.

*inputs:*

*input*

| *input inputs*

*input:*

LINE *ICONST variables*

*variables:*

*variable*

| *variable variables*

*variable:*

*ID intervals*

*intervals:*

*interval*

| *intervals interval*

*interval:*

*ICONST*

| *ICONST* TO *ICONST*

| *FCONST*

| *FCONST* TO *FCONST*

# Bibliography

- [1] T. Alspaugh, S. Faulk, K. Britton, R. Parker, D. Parnas, and J. Shore. “Software Requirements for the A-7E Aircraft”. Technical report, Naval Research Laboratory, March 1988.
- [2] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. “The Algorithmic Analysis of Hybrid Systems”. *Theoretical Computer Science*, 138:3–34, 1995.
- [3] Rajeev Alur, Costas Courcoubetis, and Thomas A. Henzinger. “Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems”. In *Hybrid Systems, Lecture Notes in Computer Science #736*, pages 209–229, may 1993.
- [4] Richard J. Anderson, Paul Beame, Steve Burns, William Chan, Francesmary Modugno, David Notkin, and Jon Reese. “Model Checking Large Software Specifications”. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE’96)*, October 1996.
- [5] M. Balcer, W. Hasling, and T. Ostrand. “Automatic Generation of Test Scripts from Formal Test Specifications”. In *Proceedings of TAV3: the Symposium on Testing, Analysis and Verification*, pages 210–218, Key West Florida, December 1989. ACM SIGSOFT 89.

- [6] Ramesh Bharadwaj and Connie Heitmeyer. “Model Checking Complete Requirements Specifications Using Abstraction”. *Journal of Automated Software Engineering*, 6(1), January 1999.
- [7] Tom Bienmüller, Udo Brockmeyer, Werner Damm, Gert Döhmen, Claus Eßmann, Hans-Jürgen Holberg, Hardi Hungar, Bernhard Josko, Rainer Schlor, Gunnar Wittich, Hartmut Wittke, Geoffrey Clements, John Rowlands, and Eric Sefton. “Formal Verification of an Avionics Application using Abstraction and Symbolic Model Checking”. In Felix Redmill and Tom Anderson, editors, *Towards System Safety: Proceedings of the Seventh Safety-Critical Systems Symposium, Huntingdon, UK 1999*, pages 150–173. Springer, 1999.
- [8] N. Bjoerner, A. Browne, E. Chang, and M. Colon. “STeP: Deductive-Algorithmic Verification of Reactive and Real-time Systems”. *Lecture Notes in Computer Science*, 1102:415–418, 1996.
- [9] Francois Bourdoncle. “Abstract Debugging of Higher-Order Imperative Languages”. In *Proceedings of ACM SIGPLAN-PLDI*, pages 46–55, June 1993.
- [10] R. Boyer and J. Moore. *A Computational Logic Handbook*. Academic Press, New York, 1988.
- [11] R. E. Bryant. “Graph-based Algorithms for Boolean Function Manipulation.”. *Transactions on Computers*, 8(C-35):677–691, 1986.
- [12] Randal E. Bryant. “Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams”. *Computing Surveys*, 24(3):293–318, September 1992.
- [13] Tevfik Bultan, Richard Gerber, and Christopher League. “Verifying Systems with Integer Constraints and Boolean Predicates: A Composite Approach”. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA '98)*, pages 113–123, March 1998.

- [14] Tevfik Bultan, Richard Gerber, and William Pugh. “Symbolic Model Checking of Infinite State Programs Using Presburger Arithmetic”. In *Proceedings of International Conference on Computer-Aided Verification*, Haifa, Israel, 1997.
- [15] Tevfik Bultan, Richard Gerber, and William Pugh. “Model Checking Concurrent Systems with Unbounded Integer Variables: Symbolic Representations, Approximations and Experimental Results.”. *ACM Transactions on Programming Languages and Systems*, 1999.
- [16] J.R. Burch, E.M. Clarke, K.L. McMillan, D.J. Dill, and L.J. Hwang. “Symbolic Model Checking:  $10^{20}$  States and Beyond”. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, pages 428–439, June 1990.
- [17] William Chan, Richard J. Anderson, Paul Beame, David H. Jones, David Notkin, and William E. Warner. “Decoupling Synchronization from Local Control for Efficient Symbolic Model Checking of StateCharts”. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE’99)*, pages 142–151, May 1999.
- [18] Edmund M. Clarke, Orna Grumberg, and David E. Long. “Model Checking and Abstraction”. *IEEE Transactions on Programming Languages and Systems*, 19(2), 1994.
- [19] E.M. Clarke, E.A. Emerson, and A.P. Sistla. “Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications”. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, January 1983.
- [20] E.M. Clarke, E.A. Emerson, and A.P. Sistla. “Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications”. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.
- [21] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. Long, K.L. McMillan, and L.A. Ness. “Verification of the Futurebus+ Cache Coherence Protocol”. In *Formal Methods in System Design*, volume 6, pages 217–232, 1995.

- [22] R. Cleaveland and J. Riely. “Testing-Based Abstractions for Value-Passing Systems”. *Lecture Notes in Computer Science*, 836:417–432, 1994.
- [23] Cristina Cornes, Judicael Courant, Jean-Christophe Filliatre, Gerard Huet, Pascal Manoury, Cesar Munoz, Chetan Murthy, Christine Parent, Catherine Paulin-Mohring, Amokrane Saibi, and Benjamin Werner. “The Coq Proof Assistant, Reference Manual, Version 5.10”. Technical Report RT-0177, INRIA, Institut National de Recherche en Informatique et en Automatique, 1995.
- [24] P.-J. Courtois and D. L. Parnas. “Documentation for Safety Critical Software”. In *Proceedings of the 15th International Conference on Software Engineering*, pages 315–323, May 1993.
- [25] Patrick Cousot. *Semantic Foundations of Program Analysis, Chapter 10*. Prentice Hall, Englewood Cliffs, New Jersey, U.S.A., 1981.
- [26] Patrick Cousot. “The Calculational Design of a Generic Abstract Interpreter”. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, August 1999.
- [27] Patrick Cousot and Radhia Cousot. “Static Determination of Dynamic Properties of Programs”. In *Proceedings of the "Colloque sur la Programmation"*, April 1976.
- [28] Patrick Cousot and Radhia Cousot. “Refining Model Checking by Abstract Interpretation”. *Automated Software Engineering, special issue on Automated Software Analysis*, 6:69–95, 1999.
- [29] D. Dams, R. Gerth, and O. Grumberg. “Abstract Interpretation of Reactive Systems”. *ACM Transactions on Programming Languages and Systems*, 2(19):253–291, March 1997.
- [30] D. Dams, O. Grumberg, and R. Gerth. “Abstract Interpretation of Reactive System: Abstraction-preserving  $\forall CTL^*$ ,  $\exists CTL^*$  and  $C^*$ ”, pages 573–592. North-Holland, 1994.

- [31] David L. Dill. “The Mur $\phi$  Verification System”. In R. Alur and T.A. Henzinger, editors, *Computer-Aided Verification Computer*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393, New York, N.Y., 1996. Springer-Verlag.
- [32] J. Dingel and T. Filkorn. “Model Checking for Infinite State Systems using Data Abstraction, Assumption-commitment Style Reasoning and Theorem Proving”. In *Proceedings of 7th International Conference on Computer Aided Verification (CAV’95)*, *LNCS 939*, pages 54–69. Springer-Verlag, July 1995.
- [33] Matthew Dwyer, Vicki Carr, and Laura Hines. “Model Checking Graphical User Interfaces Using Abstractions”. In *Proceedings of Foundations of Software Engineering*, Zurich, Switzerland, September 1997.
- [34] Stephen J. Garland and John V. Guttag. “A Guide to LP, The Larch Prover”. Technical Report SRC Research Report 82, December 1991.
- [35] A. Ghosh, S. Devadas, and A. Newton. “*Sequential Logic Testing and Verification*”. Kluwer, Boston, 1992.
- [36] Patrice Godefroid. “VeriSoft: A Tool for the Automatic Analysis of Concurrent Reactive Software”. In *Proceedings of CAV’97*, pages 476–479, 1997.
- [37] M. Gordon. “Mechanizing Programming Logics in Higher Order Logic”. Technical Report 145, University of Cambridge, Cambridge, UK, 1988.
- [38] S. Graf. “Characterization of a Sequentially Consistent Memory and Verification of a Cache Memory by Abstraction”. 1995.
- [39] J.V. Guttag and J.J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.
- [40] K. Havelund and T. Pressburger. “Model Checking Java Programs Using Java Pathfinder”. *International Journal on Software Tools for Technology Transfer*, 1999.

- [41] K. Havelund and N. Shankar. “Experiments in Theorem Proving and Model Checking for Protocol Verification”. In *FME’96: Industrial Benefit and Advances in Formal Methods*, volume 1051, page P. 662. Springer-Verlag, 1996.
- [42] C.L. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. “Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications”. *IEEE Transactions on Software Engineering*, 24(11), November 1998.
- [43] Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. “SCR\*: A Toolset for Specifying and Analyzing Requirements”. In *Proceedings of 10th Annual Conference on Computer Assurance*, pages 109–122, June 1995.
- [44] K. Heninger. “Software Requirements for the A-7E Aircraft”. Technical Report NRL Report 3876, Naval Research Laboratory, Washington, DC, 1978.
- [45] K. Heninger. “Specifying Software Requirements for Complex Systems: New Techniques and Their Applications”. *IEEE Transactions on Software Engineering*, SE-6(1):2–12, January 1980.
- [46] Gerard Holzmann. Keynote address: “Designing Executable Abstractions”. In *Proceedings of 2nd Workshop on Formal Methods in Software Practice*, March 1998.
- [47] Gerard Holzmann. “A Practical Method for Verifying Event-Driven Software”. In *Proceedings of the 21st International Conference on Software Engineering (ICSE’99)*, pages 597–607, May 1999.
- [48] G.J. Holzmann. “The Model Checker SPIN”. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
- [49] Daniel Jackson. “Abstract Model Checking of Infinite Specifications”. In *Proceedings of FME’94: Industrial Benefit of Formal Methods, Second International Symposium of Formal Methods Europe*, pages 519–531, October 1994.
- [50] Daniel Jackson and Jeannette Wing. “Lightweight Formal Methods”. *IEEE Computer*, April 1996.

- [51] Wil Janssen, Radu Mateescu, Sjouke Mauw, Peter Fennema, and Petra van der Stappen. “Model Checking for Managers”. In *Theoretical and Practical Aspects of SPIN Model Checking, LNCS 1680*, pages 92–107, September 1999.
- [52] Moataz Kamel and Stefan Leue. “Validation of Remote Object Invocation and Object Migration in CORBA GIOP using Promela/Spin”. In *Proceedings of the 4th International SPIN Workshop (SPIN'94)*, Paris, France, November 1998.
- [53] P. Kelb, D. Dams, and R. Gerth. “Practical Symbolic Model Checking of the Full  $\mu$ -calculus using Compositional Abstractions”. Technical Report 95-31, Department of Computer Science, Eindhoven University of Technology, 1995.
- [54] Wayne Kelly and William Pugh. “The Omega Calculator and Library”. Technical report, University of Maryland, November 1996.
- [55] R.A. Kemmerer. “Testing Formal Specifications to Detect Design Errors”. *IEEE Transactions on Software Engineering*, SE-11(1):32–43, January 1985.
- [56] D Kozen. “Results on the Propositional  $\mu$ -calculus”. *Theoretical Computer Science*, 27:334–354, 1983.
- [57] A. Kuehlmann, A. Srinivasan, and D. P. LaPotin. “Verity — A Formal Verification Program for Custom CMOS Circuits”. *IBM Journal of Research and Development*, 39(1/2):149–165, January 1995.
- [58] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. “Property Preserving Abstractions for the Verification of Concurrent Systems”. *Formal Methods in System Design*, 6:1–35, 1995.
- [59] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993.
- [60] J S. Moore and M. Kaufmann. “A Precise Description of the ACL2 Logic”. <http://www.cs.utexas.edu/users/moore/publications/km97a.ps.Z>, 1997.

- [61] O. Mueller and T. Nipkow. “Combining Model Checking and Deduction for I/O-Automata”. In *Tools and Algorithms for the Analysis and Construction of Systems (TACAS’95)*, LNCS 1019, page 1, 1995.
- [62] G. J. Myers. *The Art of Software Testing*. Wiley International, 1979.
- [63] Michael Norrish. “An Abstract Dynamic Semantics for C”. Technical Report TR421-mn200, University of Cambridge Computer Laboratory, May 1997.
- [64] Abelardo Pardo and Gary D. Hachtel. “Automatic Abstraction Techniques for Propositional  $\mu$ -calculus Model Checking”. In *Proceedings of 9th International Conference on Computer Aided Verification (CAV’97)*, LNCS 1254, pages 12–23. Springer-Verlag, June 1997.
- [65] David Russinoff. “A Mechanically Checked Proof of Correctness of the AMD K5 Floating-Point Square Root Microcode”. In *Formal Methods in System Design, Special Issue on Arithmetic Circuits*, 1997.
- [66] N. Shankar, S. Owre, and J. Rushby. “The PVS Proof Checker: A Reference Manual (Beta Release)”. Technical report, Computer Science Lab, SRI International, Menlo Park, CA, March 1993.
- [67] Tirumale Sreemani and Joanne M. Atlee. “Feasibility of Model Checking Software Requirements: A Case Study”. In *Proceedings of COMPASS’96*, Gaithersburg, Maryland, June 1996.