Lightweight Reasoning About Program Correctness

Marsha Chechik Wei Ding
Department of Computer Science
University of Toronto
Toronto, Ontario, Canada M5S 3G4
{chechik,wei}@cs.toronto.edu

Abstract

Automated verification tools vary widely in the types of properties they are able to analyze, the complexity of their algorithms, and the amount of necessary user involvement. In this paper we propose a framework for step-wise automatic verification and describe a lightweight scalable program analysis tool that combines abstraction and model checking. The tool guarantees that its *True* and *False* answers are sound with respect to the original system. We also check the effectiveness of the tool on an implementation of the Safety-Injection System.

Key Words: Program analysis, abstract interpretation, model checking, CTL.

1 Introduction

Recent years have seen an increasing interest in computer-supported techniques for analyzing correctness of software artifacts. In particular, this interest is caused by the potential effectiveness of lightweight formal methods [Jackson and Wing, 1996]. In this approach, verification consists of automated checking of an artifact against some critical properties (e.g., deadlock-freedom, security, fairness), often concentrating on debugging instead of assurance. Most often, lightweight methods include *model checking* [Clarke et al., 1986] – a technique for automatically verifying properties of a system. 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 [Holzmann, 1997], SMV [McMillan, 1993], and Murϕ [Dill, 1996]. Although model checking started as a technique for verifying hardware, it has been effectively applied in a variety of software projects. For example, SMV was used to verify correctness of mode logic in A-7 aircraft [Sreemani and Atlee, 1996] and TCAS specifications [Chan et al., 1999]; SPIN was applied to the validation of the remote object invocation in CORBA GIOP [Kamel and Leue, 1998], checking Java programs [Havelund and Pressburger, 1999], and many others. Model checking became part of the routine V&V process during the development of Lucent's new server product [Holzmann, 1999], and has been applied to reasoning about user interfaces [Dwyer et al., 1997] and business processes [Janssen et al., 1999].

Model checking offers a potential for 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 support reasoning about infinite-state systems by "executing" all paths of the system up to a certain depth [Godefroid, 1997, Holzmann, 1997]. 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 [Holzmann, 1999, Cousot and Cousot, 1999, Visser et al., 2000]. Although it is highly-desirable that properties hold on the abstracted model if and only if they hold in the original model [Clarke et al., 1994], such assurance is difficult to obtain: a different abstraction has to be built for each class of properties under analysis [Dams et al., 1997].

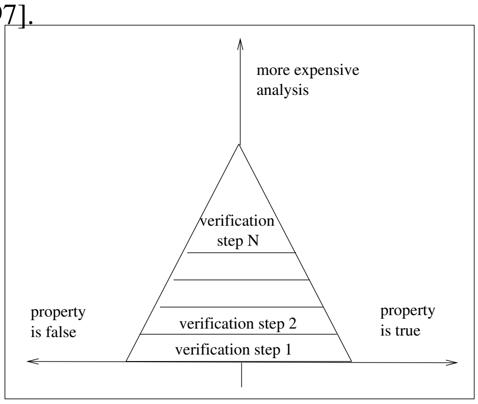

Figure 1: Framework for automatic verification.

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 1. 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 results 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 latter case, the analyst applies 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 (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.

In this paper, 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 is true". However, our approach is to build a "level 1" verifier. First, we 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. Then we model check the abstracted system. If our analysis yields *True*, the property holds in the original system; if it yields *False*, the property does not hold, and if it yields *Maybe*, the analysis is inconclusive. For such cases, properties can be verified using more expensive techniques.

1.1 Related Work

The idea of verification with the presence of abstraction has been explored by several researchers. Most of the approaches, e.g., [Jackson, 1994, Clarke et al., 1995, Kelb et al., 1995, Colon and Uribe, 1998, Saidi, 1999] are based on computing an over-approximation of the behaviour of the program and using it for reasoning about universally-quantified properties, and computing an under-approximation and using it for existentially-quantified properties.

Bultan [Bultan et al., 2000, Bultan et al., 1999, Bultan et al., 1998] and his colleagues built an infinite-state symbolic model-checker. The model-checker is composite: boolean and enumerated type variables are represented using binary decision diagrams (BDDs), whereas integer variables are represented using Presburger constraints. In addition to the standard BDD library CUDD [Somenzi, 1999], this model-checker also uses the Omega library [Kelly et al., 1996, Pugh, 1992] for efficient manipulation of symbolic encodings of transition relations and sets of states using affine constraints on integer variables, logic connectives and quantifiers. This approach, if it converges, guarantees that *True* and *False* answers are sound. However, the procedure is partial, with the convergence dependent on the structure of the program and the formula to be verified. The authors also propose an approximation technique based on *widening* (see Section 2.1 below) that allows them to guarantee convergence, but this results in a conservative analyzer (it always terminates and never yields a spurious result, but might not give a definite answer).

Dams [Dams et al., 1994, Dams et al., 1997] demonstrated how to abstract reactive systems so that the abstracted transition systems preserve certain forms of combined existential/universal properties. The properties are specified using a version of μ-calculus [Kozen, 1983] which 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. However, this approach requires a different abstraction for each property.

Pardo [Pardo and Hachtel, 1997] showed how to build the abstract and the concrete models of the system and conservatively verify properties expressed in μ-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.

Our work is probably the closest to that of Bruns and Godefroid [Bruns and Godefroid, 1999], [Bruns and Godefroid, 2000]. They have introduced Kripke structures with three-valued state variables and transitions representing *partial state spaces*, referring to them as *partial Kripke structures*. They extended temporal logic, both linear and branching-time, to this case, proposing both two-valued and three-valued logics for expressing properties of partial models. Model-checking in these logics reduces to two questions to a classical model-checker. Since reasoning about partial models is, in effect, reasoning about all *completions* of those models – it potentially answers the question "Is there a completion of this partial model making the property *True*", they describe 3-valued model-checking problem as related to satisfiability.

Huth et al. [Huth et al., 2001] apply the *modal transition systems* (MTS) of Larsen and Thomsen [Larsen and Thomsen, 1988] to problems which have been treated with three-valued logic, such as the partial specifications of Bruns and Godefroid, above, and the three-valued program analysis of Sagiv, Reps, and Wilhelm [Sagiv et al., 1999]. These systems have "must", "may", and "must not" type transitions. The authors define a 3-valued extension of the modal μ-calculus for MTS and describe an algorithm for model-checking queries in a fragment of this language using classical model-checking.

1.2 Organization of the Paper

The rest of this paper is organized as follows: Section 2 gives an overview of model checking and abstract interpretation. Section 3 discusses the theoretical goals of this work and introduces new algorithms for our program verification system. The design of this system is described in Section 4. Section 5 demonstrates the results of using our abstract model checker to analyze the Safety-Injection System. We conclude the paper with the summary of this work and the outline of future research directions.

2 Background

In this section we recall basic definitions of abstract interpretation and model checking.

2.1 Abstract Interpretation

Given a finite or an infinite system and a desired abstraction, *abstract interpretation* [Cousot and Cousot, 1976, Cousot and Cousot, 1977] provides a method for symbolically executing systems using the abstract instead of the concrete domain. Familiar data-flow analysis algorithms, e.g. constant propagation or live variables, are examples of abstract interpretation. In particular, abstract interpretation can be used for building the abstract state-space of the system. The abstraction is provided by the user and need not be dependent on the choice of properties to verify.

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

For abstract interpretation, Galois connection is defined as follows:

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

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,

$$(\{-1345\} \times \{17\}) \stackrel{\alpha}{\to} -(+) \times (+) = (-) \times (+) = (-)$$

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

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

$\begin{pmatrix} 0 \\ - \\ + \end{pmatrix}$ 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 interpretor 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 is converging:

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 from 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, [Cousot and Cousot, 1976] introduced an abstract binary operator *widening*, denoted as $\overline{\nabla}$, which represents a "jump". For any abstract values i_0 and i_1, $i_0 \cup i_1 \subseteq i_0 \overline{\nabla} i_1$. [Cousot and Cousot, 1976] defined widening as follows:

$[a_1, b_1]$ $\overline{\nabla}$ $[a_2, b_2]$ =
    [if $a_2 < a_1$ then $-\infty$ else a_1 fi,
      if $b_2 > b_1$ then $+\infty$ else b_1 fi]

For example,

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

2.2 Model Checking

In this paper we concern ourselves with *CTL model checking* – an automatic technique for verifying properties expressed in a propositional branching-time temporal logic called *Computational Tree Logic* (CTL) [Clarke et al., 1986]. 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 $R \subseteq S \times S$, an initial state I, a set of atomic propositions P, and a labeling function $L : S \rightarrow 2^P$. R must be a total function, i.e, $\forall s \in S \cdot \exists t \in S \cdot (s, t) \in R$. 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.

Our specification language is an extension of CTL that allows us to specify and verify properties involving arithmetic ($+$, $/$, $\times$, `exp`, `mod`) and logical operations ($=$, $\neq$, $>$, $\geq$, $<$, $\leq$) on the values of global variables of the program. $x > 5$ and $(x + 2)/3 = y$ are some of the atomic propositions in this version of CTL. CTL is then defined as follows:

1. Every atomic proposition $p \in P$ is a CTL formula.
2. If φ and ψ 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 φ holds in some (every) immediate successor of the current program state; $EF\varphi$ ($AF\varphi$) means that φ holds in the future along some (every)

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

Figure 2: Formal definition of CTL.

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, φ continuously holds until ψ becomes true. The formal definition is given in Figure 2, where the remaining operators are defined as follows:

$$
\begin{array}{rcl}
AF(\varphi) & \equiv & A[True\, U\varphi] \\
AG(\varphi) & \equiv & \neg EF(\neg\varphi) \\
EF(\varphi) & \equiv & E[True\, U\varphi] \\
EG(\varphi) & \equiv & \neg AF(\neg\varphi)
\end{array}
$$

Definitions of AF and EF indicate that we are using a "strong until", that is, $E[\varphi U \psi]$ and $A[\varphi U \psi]$ are true only if ψ eventually occurs.

2.3 The Logic

The logic we use has been first defined by Kleene [Kleene, 1952]. The values are arranged as the total order *False* $\sqsubseteq$ *Maybe* $\sqsubseteq$ *True*, with conjunction defined as $a \wedge b \equiv$ if $a \sqsubseteq b$ then a else b. For example, *False* $\wedge$ *Maybe* $=$ *False*. Disjunction is defined as $a \vee b \equiv$ if $b \sqsubseteq a$ then a else b. Negation is: $\neg$*False* $=$ *True*, $\neg$*True* $=$ *False*, $\neg$*Maybe* $=$ *Maybe*, so the law of excluded middle ($a \wedge \neg a =$ *False*) does not hold when $a =$ *Maybe*. This logic is *quasi-boolean*, and is discussed in detail in [Chechik et al., 2001].

```
int b;
int xy;
int main ( ) {
  int a;
  int c;
  b = 13;
  c = 2;
  xy = -20;
  while ( 1 ) {
    xy = xy + 4;
    if (xy == 0)
      b = 5;
    else
      b = b * c;
    if ( (a != 0) && (a >= -3) )
      if ( (a != 2) && (a != 4) && (a !=7) )
        if (a != -2)
          c = 2;
    printf(``xy is %d'', xy);
    printf(``b is %d'', b);
  }
}
```

Figure 3: A program fragment in C–.

Finally, material implication is defined as

$$a \Rightarrow b \equiv \neg a \vee b$$

For example, $\mathit{Maybe} \Rightarrow \mathit{False} = \neg \mathit{Maybe} \vee \mathit{False} = \mathit{Maybe}$.

3 Lightweight Model Checking

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 *True, False* and *Maybe*, such that the analysis that results in *True* and in *False* is sound. Using static analysis, we build the abstract system by associating each state of the program with an abstraction of the set of values that program variables can attain when the control reaches this point along any execution path. This abstraction, which reduces the state-space for finite-state and for infinite-state systems, is computed completely automatically.

In this section we introduce the language for constructing programs, describe the process of building the labeled transition machine, and present a model checking algorithm on the three-valued logic.

3.1 The Input Language

Our input language, called C–, is a simple language with C-like syntax. 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 this language. It also does not support any user-defined (compound) data structures. A complete grammar of the language is available in [Ding, 2000]. Figure 3 gives an example program written in C–.

3.2 Construction of a Labeled Transition System

Here, we describe the transformation of the program representation into a labeled transition system.

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 ($L_T, L_F : W \to 2^P$).

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" [Norrish, 1997].

Definition 1 *A* state *(otherwise referred to as* program state*) is a mapping between the set of global variables and their values. A* state change *occurs when at least one of the global variables changes its value.*

Once the abstract finite Kripke structure is constructed, we would like to ask questions about it. Since questions are asked of the entire program, it makes sense to limit them to only global variables. This treatment is standard for reasoning about structured programs (e.g., as implemented in the Promela/SPIN framework [Holzmann, 1997]). However, for object-oriented programs, the concept of global variables is not defined, and some recent work addressed the ability to phrase and reason about properties of object methods and instance variables [Demartini et al., 1999].

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 define a set of variables V and let $V_w \subseteq V$ be the set of variables which are accessible in the lexical scope associated with a state $w \in W$, and $G \subseteq V$ be the set of variables which are accessible at every point of the program. Thus, G is the set of *global variables*. Each state w in the program is an $n+1$-tuple, $w = (ln, (v_1, d_1), (v_2, d_2), ..., (v_n, d_n))$, in which ln corresponds to the line number of the state in the program, and $\forall i,\ 1 \leq i \leq n, v_i \in V_w, d_i \in 2^{D_i}$; here, d_i is a subset of D_i – the values of the concrete domain of v_i.

We start the analysis by parsing the program and building an Abstract Syntax Tree (AST). AST is an intermediate representation for the structure of the program under interpretation. 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 mapping a concrete state w onto an abstract state w^α, where $w^\alpha = \alpha(w)$. This process maps each $d_i \in 2^{D_i}$ onto an abstract value ${D^\alpha}_i$. The abstract value for boolean variables of C– programs is a set of values they can attain when the control reaches this point. For integer variables, such a set can be infinite, and we abstract it further. The abstraction function is introduced and discussed in Section 4. The above process results in an abstract state space W^α, in which each $w^\alpha \in W^\alpha$ is an $n+1$-tuple $w^\alpha = (ln, (v_1, {D^\alpha}_1), (v_2, {D^\alpha}_2), ...(v_n, {D^\alpha}_n))$. Notice that line numbers and the set of variables are the same in the concrete and the abstract state space. α is chosen so that W^α is finite, and an abstract state w^α can represent one or more or even an infinite number of concrete states due to the abstraction.

The labeling functions become ${L^\alpha}_T, {L^\alpha}_F : W^\alpha \to 2^P$. In the concrete domain, $\forall w \in W, L_T \cap L_F = \emptyset$, and $L_T \cup L_F = 2^P$. 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 latter 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. The goal of our work is to ensure that we get "reliable positive" and "reliable negative" answers, i.e., ${L^\alpha}_T \subseteq L_T$ and ${L^\alpha}_F \subseteq L_F$. So, in our case, ${L^\alpha}_T \cap {L^\alpha}_F = \emptyset$, but ${L^\alpha}_T \cup {L^\alpha}_F \subseteq 2^P$.

Further, we want to ensure that all transitions of the concrete system are preserved in the abstract, but the concretization of abstract transitions does not result in spurious transitions. To ensure that, we build a tuple (R^α_M, R^α_T) of transition relations over the abstract state space. $R^\alpha_T : W^\alpha \times W^\alpha$ captures *definite* transitions, and is defined as follows:

$$(s^\alpha, t^\alpha) \in R_T^\alpha \text{ iff } \forall s, t \in W \cdot (s^\alpha = \alpha(s) \ \wedge \ t^\alpha = \alpha(t)) \Rightarrow (s,t) \in R$$

$R_M^\alpha : W^\alpha \times W^\alpha$ captures *possible* transitions and is defined as follows:

$$(s^\alpha, t^\alpha) \in R_M^\alpha \text{ iff } \exists s, t \in W \cdot s^\alpha = \alpha(s) \ \wedge \ t^\alpha = \alpha(t) \ \wedge \ (s,t) \in R$$

Clearly, $R_T^\alpha \subseteq R_M^\alpha$. The resulting abstract finite-state program is $PG^\alpha = (W^\alpha, (R_T^\alpha, R_M^\alpha), I^\alpha, P, ({L^\alpha}_T, {L^\alpha}_F))$.

In order to construct an abstract Kripke structure in which every transition corresponds to a change to a global variable, we define a "global variable changed" predicate c on a state $y \in W^\alpha$:

$$c(y) \equiv \exists x \in W^\alpha \cdot ((x,y) \in R_M^\alpha) \Rightarrow (\exists g \in G \cdot (g, D^\alpha(x)) \neq (g, D^\alpha(y)))$$

In the above definition, $(g, D^\alpha(x))$ and $(g, D^\alpha(y))$ represent g's abstract value in states x and y, respectively. This definition indicates that at least one global variable changes its value in state y. Now we construct the abstract aggregate state space S^α. In this construction, every element $s^\alpha \in 2^{W^\alpha}$ contains one state w^α 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^α via the transitive closure of R_M^α (denoted $R_M^{\alpha *}$). S^α is defined recursively as follows:

$$\forall w^\alpha \in W^\alpha \cdot c(w^\alpha) \Rightarrow (\exists! s^\alpha \in S^\alpha \cdot w^\alpha \in s^\alpha)$$
$$\forall w_1^\alpha, w_2^\alpha \in W^\alpha \cdot \exists s^\alpha \in S^\alpha \cdot (c(w_1^\alpha) \ \wedge \ \neg c(w_2^\alpha) \ \wedge \ (w_1^\alpha, w_2^\alpha) \in R_M^{\alpha *} \ \wedge \ w_1^\alpha \in s^\alpha) \Rightarrow (w_2^\alpha \in s^\alpha)$$

We use $\exists!$ to indicate existence and uniqueness. Note that values of global variables within s^α are the same. We refer to L_T^α and L_F^α as the labeling functions that map each $s^\alpha \in S^\alpha$ to a set of atomic propositions *on global variables* that are true (false) in that state. The transitions between states in S^α are again split into *definite* (E_T^α) and *possible* (E_M^α), defined as follows:

$$(s^\alpha, t^\alpha) \in E_M^\alpha \quad \text{iff} \quad \exists i, j \cdot {w_i}^\alpha \in s^\alpha \ \wedge \ {w_j}^\alpha \in t^\alpha \ \wedge \ ({w_i}^\alpha, {w_j}^\alpha) \in R_M^\alpha$$
$$(s^\alpha, t^\alpha) \in E_T^\alpha \quad \text{iff} \quad \forall i, j \cdot ({w_i}^\alpha \in s^\alpha \ \wedge \ {w_j}^\alpha \in t^\alpha) \Rightarrow ({w_i}^\alpha, {w_j}^\alpha) \in R_T^\alpha$$

Our abstract Kripke structure $K^\alpha = (S^\alpha, (E_T^\alpha, E_M^\alpha), I^\alpha, P, ({L^\alpha}_T, {L^\alpha}_F))$ is now ready.

3.3 Model Checking Algorithm

We now present the algorithm that receives a Kripke structure K^α constructed above and a correctness property expressed in the version of CTL described in Section 2, and determines whether or not the property holds in the system. As mentioned in the previous section, we want to ensure that our analysis yields "reliable positive" and "reliable negative" answers, i.e., if the analysis concluded that a property is *True*, then it holds in the original system, and if the analysis concluded that a property is *False*, then it does not hold in the original system. In order to do so, 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.

The algorithm recursively goes through the structure of the property under analysis, associating each subproperty φ with a pair of sets of states (Yes(φ), No(φ)). Yes(φ) $\subseteq S^\alpha$ is a set of states in which φ 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 φ is *False*, is defined similarly. In all states which are in neither Yes(φ) nor No(φ), φ has a value *Maybe*. These states are not explicitly computed. We also define two *predecessor* functions. The first one, pred$_T$: $2^{S^\alpha} \rightarrow 2^{S^\alpha}$, takes a set of states Q and returns all states that can reach some state in Q in one *True* transition:

$$s^\alpha \in \text{pred}_T(Q) \text{ iff } \exists t^\alpha \in Q \cdot (s^\alpha, t^\alpha) \in E_T^\alpha$$

pred$_M$ is the same as pred$_T$ except that it returns all states that can reach some state in Q in one *Maybe* transition:

$$s^\alpha \in \text{pred}_M(Q) \text{ iff } \exists t^\alpha \in Q \cdot (s^\alpha, t^\alpha) \in E_M^\alpha$$

The algorithm, inspired by Bultan's symbolic model checker for infinite-state systems [Bultan et al., 1999], is given in Figure 4. For example, a property $\varphi \wedge \psi$ holds in state s^α if s^α is in Yes sets of both φ and ψ. The same property does not hold in state s^α if s^α is in the No set of either φ or ψ. When verifying $EX\varphi$, we note that if φ holds in some immediate successor of state s^α, then $EX\varphi$ holds in s^α; any immediate successor in which φ may hold ($S^\alpha-$ No(φ)) should be excluded from No($EX\varphi$). $A[\varphi U \psi]$ is computed recursively as follows: $A[\varphi U \psi]$ is *True* in all states S_0 in which ψ holds; it is also *True* in predecessors of S_0 in which φ holds and all of which successors are in S_0. $A[\varphi U \psi]$ is *False* in a state s^α iff ψ does not hold in s^α and either φ does not hold in s^α or one of its successors does not lead to ψ.

Theorem 1 *The abstract model-checking algorithm in Figure 4 is correct. Let K be a concrete model, K^α be its abstraction, and p be a property under analysis. Further, let function* `Check`(p) *be run on K^α, returning a tuple (Yes(p), No(p)). Finally, let I and I^α be the concrete and the abstract initial states,* Procedure CHECK(p)

CASE

$p \in A$	:	Return (Yes(p), No(p))
$p = \neg\varphi$	:	Return (No(φ), Yes(φ))
$p = \varphi \wedge \psi$	:	Return (Yes(φ) $\cap$ Yes(ψ), No(φ) $\cup$ No(ψ))
$p = \varphi \vee \psi$	:	Return (Yes(φ) $\cup$ Yes(ψ), No(φ) $\cap$ No(ψ))
$p = EX\varphi$	:	Return (pred$_T$(Yes(φ)), $S^\alpha -$ pred$_M$($S^\alpha -$ No(φ)))
$p = AX\varphi$	:	Return ($S^\alpha -$ pred$_M$($S^\alpha -$ Yes(φ)), pred$_T$(No(φ)))
$p = E[\varphi U \psi]$	:	1. Y_0 = Yes(ψ) Y_{i+1} = Y_i $\cup$ (pred$_T$(Y_i) $\cap$ Yes(φ)) Until Y_m = Y_{m+1} 2. N_0 = No(ψ) N_{i+1} = N_i $\cap$ (($S^\alpha -$ pred$_M$($S^\alpha - N_i$)) $\cup$ No(φ)) Until N_n = N_{n+1} 3. Return (Y_m, N_n)
$p = A[\varphi U \psi]$	:	1. Y_0 = Yes(ψ) Y_{i+1} = Y_i $\cup$ ((pred$_T$(Y_i) $-$ pred$_M$($S^\alpha - Y_i$)) $\cap$ Yes(φ)) Until Y_m = Y_{m+1} 2. N_0 = No(ψ) N_{i+1} = N_i $\cap$ (pred$_T$(N_i) $\cup$ No(φ)) Until N_n = N_{n+1} 3. Return (Y_m, N_n)

Figure 4: Model Checking Algorithm.

respectively. Then,

$$I^\alpha \in \mathrm{Yes}(p) \quad \Rightarrow \quad K, I \models p$$
$$I^\alpha \in \mathrm{No}(p) \quad \Rightarrow \quad K, I \not\models p$$

For the proof of this theorem, please refer to [Ding, 2000].

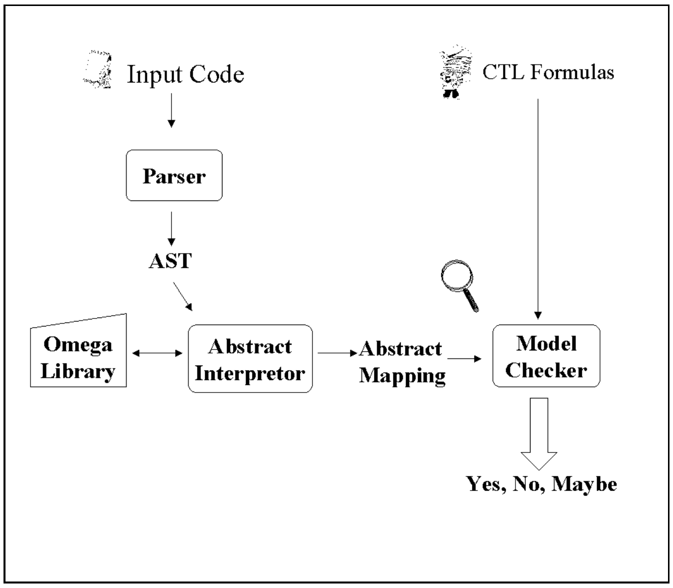

Figure 5: Architecture of the Abstract Model Checker.

4 Implementation

Our Abstract Model Checker (AMC), implemented in C, has the architecture as shown in Figure 5. The CTL formulas and the input language have been described in Sections 2 and 3, respectively. The *Abstract Interpretor* (AI) receives the program under analysis and builds the Kripke structure $K^\alpha = (S^\alpha, (E^\alpha_T, E^\alpha_M), I^\alpha, P, (L^\alpha{}_T, L^\alpha{}_F))$ using the process described in Section 3. This structure, together with a set of CTL formulas, becomes the input to the *Model Checker* which checks each property and returns *True* if the formula holds in the program, *False* if the formula does not hold, or *Maybe* if the validity of the formula cannot be established. In the latter two cases, the model checker also returns a counter-example. At the moment, the counter-example facility includes just the line number and the variable-value mappings of the states where the formula is not *True*.

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)[1]. At the beginning of a C– program, all variables are *UNDEF*, giving rise to the initial input

[1]For brevity, we do not discuss the treatment of *UNDEF* here. For details, please refer to [Ding, 2000].

context. Values of infinite-domain (or infinite-domain for practical purposes) variables such as integers should be abstracted further. In Section 2, we have briefly discussed how an abstraction function α can be applied to a set to get an interval. However, for better precision, we associate each infinite-domain variable with a (finite) set of intervals, with the following interpretation

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

For practical reasons, each set can consist of a finite number of intervals, referred to here as `MAX_INTERVAL` and set to 5 in the current implementation of AI. We define $\overset{\alpha}{\cup}$ (union on the set of intervals) below. Let a_i, b_j be intervals and assume, without a loss of generality, that $m \leq n$:

$$\begin{aligned}
&\{a\} \overset{\alpha}{\cup} \emptyset = \{a\} \\
&\{a\} \overset{\alpha}{\cup} \{b_1, ..., b_n\} = \{a \cup b_1, ..., a \cup b_n\} \\
&\{a_1, ..., a_m\} \overset{\alpha}{\cup} \{b_1, ..., b_n\} = \{a_1 \cup ... \cup a_m\} \overset{\alpha}{\cup} \{b_1, ..., b_n\}
\end{aligned}$$

When we encounter two sets, each containing more than one interval, we first union elements of the set that has the smaller number of intervals (in this case, $\{a_1, ..., a_m\}$) into one interval, and then union the result with each interval of the other set. Interval operations *union* and *difference* have their usual meaning, and *widening* on intervals is defined in Section 2. Other operations on sets of intervals, $\overset{\alpha}{-}$ (difference) and $\overset{\alpha}{\overline{\nabla}}$ (widening) are similar. Additional operations, including comparison and arithmetic, are defined formally in [Ding, 2000].

The algorithm used in our AI for analyzing conditional statements is depicted in Figure 6. Given an input abstract context S_i, a conditional expression *iexpr* and statements to execute when *iexpr* 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 manipulates sets of integer tuples and relations between integer tuples. Some examples include:

$\{[i,j] : 1 \leq i, j \leq 10\}$	A set of all intervals with left and right bounds between 1 and 10 inclusive.
$\{[i,j] \to [j,j'] : 1 \leq i < j < j' \leq n\}$	A set of relations between intervals with bounds between 1 and n, where the upper bound of the interval in the domain is the same as the lower bound of the interval in the range. Note that n is a free variable in this expression.

Tuple relations and sets are described using Presburger formulas. Presburger formulas contain affine constraints, the usual logical connectives, and the quantifiers. Relations and sets can be combined using functions such as composition, intersection, union and difference. The Omega library is a set of C++ classes for such manipulation; the Omega calculator is the text-based front-end to the Omega library. The Omega library cannot simplify all Presburger formulas efficiently (there is a 2^{2^n} nondeterministic lower bound and a $2^{2^{2^{O(n)}}}$ deterministic upper bound on the time required to verify Presburger formulas [Oppen, 1978]). However, in practice the worst case situations are not encountered, and the library is quite effective.

We use the Omega library for symbolically executing conditional expressions involving intervals; thus, our type system is limited to boolean and integer variables.

For example, suppose we are running our AI on the program fragment depicted in Figure 3. The example was chosen so that there is at most one state change per line of code. Figure 7 shows the control-flow graph for this fragment, with each state associated with the program line number. Let the input context before state 11 be $((\mathtt{xy}, \{[-20, 52]\}), (\mathtt{a}, \{[-5, 8]\}), (\mathtt{b}, \{[13, 13]\}), (\mathtt{c}, \{[2, 2]\}))$. The condition `xy == 0` evaluates to *Maybe*; therefore, we call the Omega calculator to determine that the value of `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 $\{[5, 26]\}$ 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 `MAX_INTERVAL` intervals, and further splitting cannot be done; instead, we union `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 [Ding, 2000]. The output value for `a` after state 17 is $\{[-3, -3], [-1, 8]\}$.

Loops are executed until a fixpoint on values of all variables has been achieved. In order to ensure that

Procedure EVAL-IF $(iexpr, stmt_t, stmt_f, S_i)$

Evaluate *iexpr*

IF *iexpr* is *True*

  Execute *stmt*$_t$ starting with S_i to get S_o

  Return S_o

ELSE IF *iexpr* is *False*

  Execute *stmt*$_f$ starting with S_i to get S_o

  Return S_o

ELSE IF *iexpr* is *Maybe*

  Call Omega calculator to get ${S_i}^t$, ${S_i}^f$

  Execute *stmt*$_t$ starting with ${S_i}^t$ to get ${S_o}^t$

  Execute *stmt*$_f$ starting with ${S_i}^f$ to get ${S_o}^f$

  Return ${S_o}^t \overset{\alpha}{\cup} {S_o}^f$

Figure 6: Algorithm for analyzing conditional statements.

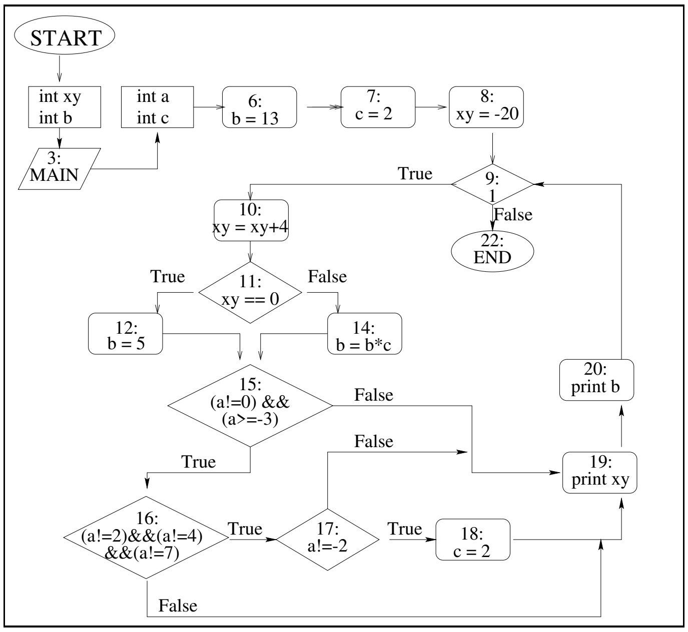

Figure 7: Control-flow graph of the program in Figure 3.

this fixpoint occurs in a finite number of steps, we change values of variables in each loop a finite number of times, referred to as MAX_LOOP and set to 20 in the current implementation of AI. Further, we keep track of whether values of variables decrease or increase between iterations. If a fixpoint was not achieved, we widen values of non-converged variables, with the increase and the decrease leading to the values of $+\infty$ and $-\infty$, respectively. Afterwards, we proceed executing the loop again to ensure that dependencies between the variables are adequately captured. Table 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 3. At the first iteration, these values are $\{[13, 13]\}$ and $\{[-20, -20]\}$, respectively. In the following 18 iterations we note that the maximum values `b` and `xy` can attain are increasing, whereas their minimum values stay the same. Thus, the widening which occurs on the 20th iteration changes only the maximum values of these variables. The 21th iteration does not bring any further changes, thus achieving a fixpoint.

Figure 8 shows the final Kripke structure built from the control-flow graph of Figure 7. 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 statement. For example, state 10 of Figure 8 is an aggregation of states 10 and 11 of Figure 7. Solid and dashed lines indicate *True* and *Maybe* transitions, respectively. For example, a transition between states 6 and 8 is known to correspond to a transition in the concrete system and thus is marked as *True*.

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

Table 1: Execution of the while loop of the program in Figure 3.

The resulting Kripke structure becomes input to the model checker whose algorithm is described in Section 3. For example, we can model check the structure depicted in Figure 8 against CTL properties $AG((\mathtt{xy} + \mathtt{b}) \leq 0)$, $EF(\mathtt{b} = 5)$, and $EF(\mathtt{b} = 12)$. Our model checker returns *False* for the first property because it is violated in state corresponding to line 12 of the program. The second property is determined

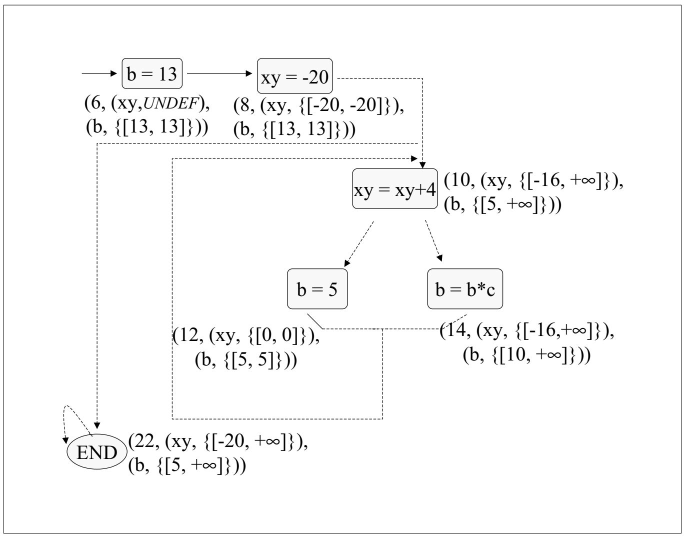

Figure 8: Kripke structure K^{α} built from the program fragment in Figure 3.

to be *True* because it is satisfied in the state corresponding to line 12. The third property is determined to be *Maybe*: it *Maybe* holds in the state corresponding to line 14 and does not definitely hold in any state.

We are now ready to analyze performance of our algorithm. 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 `MAX_LOOP` times; therefore, each statement in PG can be changed at most `MAX_LOOP` $\times$ $|V|$ times, and there are $n \times$ `MAX_LOOP` $\times$ $|V|$ 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 $O((n-1) \times |V| \times \texttt{MAX\_INTERVAL})$ steps. In addition, each change of state may be associated with a conditional statement which takes $\omega(|V|)$ – the number of steps taken by the Omega calculator for the variables in V. Therefore, the entire computation of the abstract interpretor takes $\texttt{MAX\_LOOP} \times |V| \times n \times \texttt{MAX\_INTERVAL} \times (n-1) \times |V| \times \omega(|V|)$ steps which is $O(|V|^2 \times n^2 + n \times |V|) \times \omega(|V|)$. This complexity measure seems very high (in the worst case, each call to the Omega library takes $2^{2^{2^{\cdots}}}$ steps, where the number of powers of 2 is the number of variables in the expression under analysis). However, in practice this complexity is significantly lower. First, calls to the calculator are made only for deciding conditional expressions, and these are usually composed of a fairly small number of variables. Second, Omega calculator's average-case performance is significantly faster than the theoretical measure, making it possible to incorporate this tool in a number of optimizing compilers.

To compute the performance of our model checker, we let $|P|$ be the length of a property P. Among all the CTL formulas, $A[\varphi U \psi]$ is the most complex. For this algorithm, N_i can change value at most n times before a fixpoint is reached, and it takes $n-1$ steps to compute N_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 $O(|P| \times n^2)$.

5 Case Study

To determine the effectiveness of our abstract model checker, we analyzed the simplified version of a *Safety-Injection System* [Courtois and Parnas, 1993]. 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 raises above `Permit`, the system's pressure level becomes high; otherwise, this level is "within the permitted range".

We have implemented the Safety-Injection system as a 200-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, `Overridden` of type boolean, constants `Low`, `Permit`, `TooLow`, `Permitted` and `High` and `Pressure` of type integer (our system does not support enumerated types, and the last three constants are used to indicate symbolic values of `Pressure`). The implementation also includes 7 functions and 8 local variables. The C– code for the case study appears in Appendix A.

The Safety-Injection system has been verified by two other research groups. Bharadwaj and Heitmeyer [Bharadwaj and Heitmeyer, 1999] analyzed SCR specifications using the SPIN [Holzmann, 1997] model checker. Their technique only supports finite-domain variables, including integer subranges and enumerated types. The size of the concrete state space is reduced by two methods: eliminating variables which are not relevant to the property being verified (SCR ensures that dependencies between variables form a partial order), and by replacing input variables by predicates. The latter approach makes the verification conservative, with the potential for producing false negatives. In addition, the system has been analyzed by Bultan [Bultan et al., 1998] using his infinite-state model-checker. Both approaches were conclusive on two properties:

The system will not become overridden if the system is being reset when the pressure is not too high.

1. $AG((\texttt{Reset} \ \wedge \ \texttt{Pressure} \neq \texttt{High}) \Rightarrow \neg\texttt{Overridden})$

   The system will inject the coolant if the pressure is too low and the reset button is pressed.

2. $AG((\texttt{Reset} \ \wedge \ \texttt{Pressure} = \texttt{TooLow}) \Rightarrow \texttt{Injection})$

Our analysis yielded *True* for the above properties and for two additional properties:

The system becomes overridden when the block is pressed, reset is not, and the pressure is not too high.

3. $AG((\texttt{Block} \ \wedge \ \neg\texttt{Reset} \ \wedge \ \texttt{Pressure} \neq \texttt{High}) \Rightarrow \texttt{Overridden})$

   Whenever the pressure is permitted and the water pressure raises above the allowed threshold, then the system will eventually transit into a state where the pressure is high.

4. $AG((\texttt{Pressure} = \texttt{Permitted} \wedge \ \texttt{WaterPres} \leq \texttt{Permit})$

   $\qquad\qquad \Rightarrow AX(\texttt{WaterPres} \geq \texttt{Permit} \Rightarrow AF(\texttt{Pressure} = \texttt{High})))$

and was inconclusive of three other properties.

We verified the SafetyInjection system using our algorithm on Sun UltraSPARC-II with 4 400 MHz processors and 4 GB of RAM. The entire verification effort, including building the abstract Kripke structure and checking all the properties, took 3.92 seconds (user), 6.20 seconds (system). Our model-checker yielded *True* for each of the four properties. The final Kripke structure consisted of only 30 abstract states.

6 Summary and Future Work

In this paper we proposed a framework for step-wise automatic verification and described an implementation of a very cheap and not particularly precise model checker. This model checker verifies infinite-state sequential programs written in a subset of C against CTL formulas containing arithmetic operations. It applies property-independent abstract interpretation to create an abstract Kripke structure, and then uses this extremely compact structure to verify properties in low-order polynomial time. No user-created abstractions are necessary. The verification always converges and is guaranteed to be sound: if the model checker yields *True*, the property holds in the concrete system, and if it yields *False*, the property does not hold. This 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 can further improve its running time.

However, 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. We plan to integrate our model-checking tool with the Bandera toolkit [Corbett et al., 2000] that enables abstract interpretation w.r.t. multiple abstractions.

(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. We are also considering the adoption of Java's state transition semantics.

(5) Our model checker returns *Maybe* if it cannot determine whether a property holds in the system. We believe we can reduce the number of cases for which the verification is inconclusive by improving the reasoning about abstract values and/or by choosing property-specific abstractions.

In short-term future work we hope to extend our model-checker to reasoning about CTL* [Clarke et al., 1986] which combines branching-time and linear-time operations and is strictly more expressive than CTL. We would also 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.

Acknowledgments

We would like to thank Ric Hehner and Radu Iosif for reading earlier versions of this paper, and Mark Pichora, Albert Lai and Daniel House for many interesting discussions. We acknowledge the financial support of NSERC Postgraduate Scholarship.

References

[Bharadwaj and Heitmeyer, 1999] Bharadwaj, R. and Heitmeyer, C. (1999). “Model Checking Complete Requirements Specifications Using Abstraction”. *Journal of Automated Software Engineering*, 6(1).

[Bruns and Godefroid, 1999] Bruns, G. and Godefroid, P. (1999). “Model Checking Partial State Spaces with 3-Valued Temporal Logics”. In *Proceedings of CAV’99*, volume 1633 of *LNCS*, pages 274–287.

[Bruns and Godefroid, 2000] Bruns, G. and Godefroid, P. (2000). “Generalized Model Checking: Reasoning about Partial State Spaces”. In *Proceedings of CONCUR’00*, volume 877 of *LNCS*, pages 168–182.

[Bultan et al., 1998] Bultan, T., Gerber, R., and League, C. (1998). “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.

[Bultan et al., 2000] Bultan, T., Gerber, R., and League, C. (2000). “Composite Model Checking: Verification with Type-Specific Symbolic Representations”. *ACM Transactions on Software Engineering and Methodology*, 9(1):3–50.

[Bultan et al., 1999] Bultan, T., Gerber, R., and Pugh, W. (1999). “Model Checking Concurrent Systems with Unbounded Integer Variables: Symbolic Representations, Approximations and Experimental Results.”. *ACM Transactions on Programming Languages and Systems*.

[Chan et al., 1999] Chan, W., Anderson, R. J., Beame, P., Jones, D. H., Notkin, D., and Warner, W. E. (1999). “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.

[Chechik et al., 2001] Chechik, M., Easterbrook, S., and Petrovykh, V. (2001). "Model-Checking Over Multi-Valued Logics". In *Proceedings of Formal Methods Europe (FME'01)*, volume 2021 of *LNCS*, pages 72–98. Springer.

[Clarke et al., 1986] Clarke, E., Emerson, E., and Sistla, A. (1986). "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". *ACM Transactions on Programming Languages and Systems*, 8(2):244–263.

[Clarke et al., 1995] Clarke, E., Grumberg, O., Hiraishi, H., Jha, S., Long, D., McMillan, K., and Ness, L. (1995). "Verification of the Futurebus+ Cache Coherence Protocol". In *Formal Methods in System Design*, volume 6, pages 217–232.

[Clarke et al., 1994] Clarke, E. M., Grumberg, O., and Long, D. E. (1994). "Model Checking and Abstraction". *IEEE Transactions on Programming Languages and Systems*, 19(2).

[Colon and Uribe, 1998] Colon, M. and Uribe, T. (1998). "Generating Finite-State Abstractions of Reactive Systems using Decision Procedures". In *Proceedings of the 10th Conference on Computer-Aided Verification*, volume 1427 of *LNCS*. Springer-Verlag.

[Corbett et al., 2000] Corbett, J., Dwyer, M., Hatcliff, J., Laubach, S., Pasareanu, C., Robby, and Zheng, H. (2000). "Bandera: Extracting Finite-state Models from Java Source Code". In *Proceedings of 22st International Conference on Software Engineering*.

[Courtois and Parnas, 1993] Courtois, P.-J. and Parnas, D. L. (1993). "Documentation for Safety Critical Software". In *Proceedings of the 15th International Conference on Software Engineering*, pages 315–323.

[Cousot and Cousot, 1976] Cousot, P. and Cousot, R. (1976). "Static Determination of Dynamic Properties of Programs". In *Proceedings of the "Colloque sur la Programmation"*.

[Cousot and Cousot, 1977] Cousot, P. and Cousot, R. (1977). "Abstract Interpretation: A Unified Lattice Model For Static Analysis of Programs by Construction or Approximation of Fixpoints". In *Proceedings of the 4th POPL*, pages 238–252, Los Angeles, California.

[Cousot and Cousot, 1999] Cousot, P. and Cousot, R. (1999). "Refining Model Checking by Abstract Interpretation". *Authomated Software Engineering, special issue on Automated Software Analysis*, 6:69–95.

[Dams et al., 1997] Dams, D., Gerth, R., and Grumberg, O. (1997). "Abstract Interpretation of Reactive Systems". *ACM Transactions on Programming Languages and Systems*, 2(19):253–291.

[Dams et al., 1994] Dams, D., Grumberg, O., and Gerth, R. (1994). *"Abstract Interpretation of Reactive System: Abstraction-preserving $\forall CTL^*$, $\exists CTL^*$ and CTL^*"*, pages 573–592. North-Holland.

[Demartini et al., 1999] Demartini, C., Iosif, R., and Sisto, R. (1999). "dSPIN: A Dynamic Extension of SPIN". In *Proceedings of the 6th SPIN Workshop on Practical Aspects of Model-Checking*.

[Dill, 1996] Dill, D. (1996). "The Murϕ Verification System". In Alur, R. and Henzinger, T., editors, *Computer-Aided Verification Computer*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393, New York, N.Y. Springer-Verlag.

[Ding, 2000] Ding, W. (2000). Analyzing infinite-state programs with abstract interpretation. Master's thesis, University of Toronto, Department of Computer Science.

[Dwyer et al., 1997] Dwyer, M., Carr, V., and Hines, L. (1997). "Model Checking Graphical User Interfaces Using Abstractions". In *Proceedings of Foundations of Software Engineering*, Zurich, Switzerland.

[Godefroid, 1997] Godefroid, P. (1997). "VeriSoft: A Tool for the Automatic Analysis of Concurrent Reactive Software". In *Proceedings of CAV'97*, pages 476–479.

[Havelund and Pressburger, 1999] Havelund, K. and Pressburger, T. (1999). "Model Checking Java Programs Using Java Pathfinder". *International Journal on Software Tools for Technology Transfer*.

[Holzmann, 1997] Holzmann, G. (1997). "The Model Checker SPIN". *IEEE Transactions on Software Engineering*, 23(5):279–295.

[Holzmann, 1999] Holzmann, G. (1999). "A Practical Method for Verifying Event-Driven Software". In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 597–607.

[Huth et al., 2001] Huth, M., Jagadeesan, R., and Schmidt, D. A. (2001). "Modal Transition Systems: A Foundation for Three-Valued Program Analysis". In *Proceedings of 10th European Symposium on Programming (ESOP)*, volume 2028 of *LNCS*, pages 155–169. Springer.

[Jackson, 1994] Jackson, D. (1994). "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.

[Jackson and Wing, 1996] Jackson, D. and Wing, J. (1996). "Lightweight Formal Methods". *IEEE Computer*.

[Janssen et al., 1999] Janssen, W., Mateescu, R., Mauw, S., Fennema, P., and van der Stappen, P. (1999). "Model Checking for Managers". In *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *LNCS*, pages 92–107. Springer-Verlag.

[Kamel and Leue, 1998] Kamel, M. and Leue, S. (1998). "Validation of Remote Object Invocation and Object Migration in CORBA GIOP using Promela/Spin". In *Proceedings of the 4th International SPIN Workshop (SPIN'4)*, Paris, France.

[Kelb et al., 1995] Kelb, P., Dams, D., and Gerth, R. (1995). "Practical Symbolic Model Checking of the Full μ-calculus using Compositional Abstractions". Technical Report 95-31, Department of Computer Science, Eindhoven University of Technology.

[Kelly et al., 1996] Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., and Wonnacott, D. (1996). "The Omega Calculator and Library, version 1.1.0". Technical report, University of Maryland.

[Kleene, 1952] Kleene, S. C. (1952). *Introduction to Metamathematics*. New York: Van Nostrand.

[Kozen, 1983] Kozen, D. (1983). "Results on the Propositional μ-calculus". *Theoretical Computer Science*, 27:334–354.

[Larsen and Thomsen, 1988] Larsen, K. and Thomsen, B. (1988). "A Modal Process Logic". In *Third Annual Symposium on Logic in Computer Sciences*, pages 203–210. IEEE Computer Society Press.

[McMillan, 1993] McMillan, K. (1993). *Symbolic Model Checking*. Kluwer Academic.

[Norrish, 1997] Norrish, M. (1997). "An Abstract Dynamic Semantics for C". Technical Report TR421-mn200, University of Cambridge Computer Laboratory.

[Oppen, 1978] Oppen, D. (1978). "A $2^{2^{2^{pn}}}$ Upper Bound on the Complexity of Presburger Arithmetic". *Journal of Computer and System Sciences*, 16(3):323–332.

[Pardo and Hachtel, 1997] Pardo, A. and Hachtel, G. D. (1997). "Automatic Abstraction techniques for Propositional μ-calculus Model Checking". In *Proceedings of 9th International Conference on Computer-Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 12–23. Springer-Verlag.

[Pugh, 1992] Pugh, W. (1992). "The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis". *Comm. of the ACM*.

[Sagiv et al., 1999] Sagiv, M., Reps, T., and Wilhelm, R. (1999). "Parametric Shape Analysis via 3-Valued Logic". In *Proceedings of 26th Annual ACM Symposium on Principles of Programming Languages*.

[Saidi, 1999] Saidi, H. (1999). "Modular and Incremental Analysis of Concurrent Software Systems". In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, pages 92–101.

[Somenzi, 1999] Somenzi, F. (1999). "Binary Decision Diagrams". In Broy, M. and Steinbrüggen, R., editors, *Calculational System Design*, volume 173 of *NATO Science Series F: Computer and Systems Sciences*, pages 303–366. IOS Press.

[Sreemani and Atlee, 1996] Sreemani, T. and Atlee, J. M. (1996). "Feasibility of Model Checking Software Requirements: A Case Study". In *Proceedings of COMPASS'96*, Gaithersburg, Maryland.

[Visser et al., 2000] Visser, W., Park, S., and Penix, J. (2000). "Applying Predicate Abstraction to Model Check Object-Oriented Programs". In *Proceedings of 4th International Workshop on Formal Methods in Software Practice*.

A Case Study

The following is the implementation of the Safety Injection System.

```
boolean Block;
boolean Reset;
boolean Exit;
int WaterPres; /* MONITORED VARIABLES */
boolean Injection; /* CONTROLLED VARIABLES */
boolean Overriden; /* TERMS  */
boolean buttonBPressed;
boolean buttonRPressed;
boolean buttonEPressed;
int next;
int Pressure; /* MODE CLASS */
int Initialize ()  {
        WaterPres = 4;
        Pressure = 0;
        Overriden = 0;
        Injection = 0;
        Block = 0;
        Reset = 0;
        buttonBPressed = 0;
        buttonRPressed = 0;
        buttonEPressed = 0;
        next = 2;
        return 1;
}
int Get_Event (int sem) {
        int temp;
        temp = 1;
        if (sem == 1) {
                if (buttonBPressed == 0) {
                        Block = 0;
                        buttonBPressed = 1;
                    }
                else {
                        Block = 1;
                        buttonBPressed = 0;
                    }
        }
        if (sem == 2) {
                if (buttonRPressed == 0) {
                        Reset = 0;
                        buttonRPressed = 1;
                    }
                else
                        {
                        Reset = 1;
                        buttonRPressed = 0;
                    }
        }
        if (sem == 3) {
                if (buttonEPressed == 0) {
                        Exit = 0;
                        buttonEPressed = 1;
                    }
                else {
                        Exit = 1;
                        buttonEPressed = 0;
                    }
        }
        return 1;
}
int Get_Mode () {
        int temp;
        temp = 1;
```

```
	if (Pressure == 0) {
			if (WaterPres >= 5)
				Pressure = 1;
		}
	if (Pressure == 1) {
			if (WaterPres >= 15)
				Pressure = 2;
			if (WaterPres < 5)
				Pressure = 0;
		   }
	if (Pressure == 2) {
			if (WaterPres < 15)
				Pressure = 1;
		}
	return 1;
}
int Get_Term() {
	if ((Reset == 0) && (Pressure == 0))
		if (Block == 1)
			Overriden = 1;
	if ((Pressure == 1) && (Reset == 0))
		if (Block == 1)
			Overriden = 1;
	if (Pressure == 2)
		Overriden = 0;
	return 1;
}
int Get_Control () {
	if (Overriden == 0)
		Injection = 1;
	if (Pressure == 2)
		Injection = 0;
	if (Pressure == 1)
		Injection = 0;
	if ((Pressure == 0) && (Overriden == 1))
		Injection = 0;
	return 1;
}
main() {
	int flag;
	int semo;
	int flag1;
	fopen("safeinput", "r");
	flag = Initialize();
	while (1) {
		scanf(WaterPres);
		fscanf("safeinput", semo);
		flag1 = Get_Event(semo);
		flag = Get_Mode();
		flag1 = Get_Term();
		flag1 = Get_Control();
	}
}
```