Lightweight Reasoning About Program Correctness

Marsha Chechik Wei Ding
Department of Computer Science
University of Toronto
Toronto, ON M5S 3G4, Canada
(416)978-3820
{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.

Keywords

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 effectiveness and potential of lightweight formal methods [18]. 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* [5] – 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 [15], SMV [22], and Murϕ [10]. 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 [25] and TCAS specifications [4]; SPIN was applied to the validation of the remote object invocation in CORBA GIOP [20], checking Java programs [14], and many others. Model checking became part of the routine V&V process during the development of Lucent's new server product [16], and has been applied to reasoning about user interfaces [12] and business processes [19].

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 allow reasoning about infinite-state systems by "executing" all paths of the system up to a certain depth [13, 15]. 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 [16]. And, although it is highly-desirable that properties hold on the abstracted model if and only if they hold in the original model [6], such assurance is difficult to obtain: a different abstraction has to be built for each class of properties under analysis [9].

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 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 latter case, the analyst will apply a technique on verification

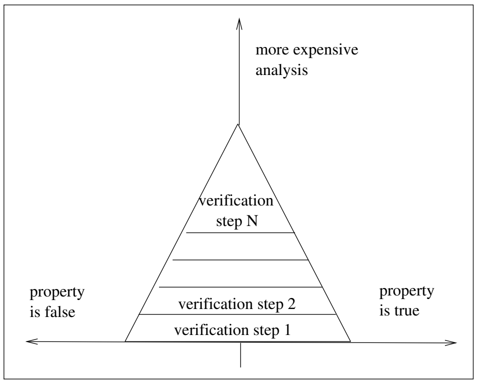

Figure 1: Framework for automatic verification.

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 will be 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.

The idea of verification with the presence of abstraction has been explored by several researchers. In particular, Jackson [17] 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. Dams [9] demonstrated how to abstract reactive systems, so that the abstracted transition systems preserves certain forms of combined safety/liveness properties. Pardo [24] built the abstract and the concrete models of the system and conservatively verified properties expressed in μ-calculus on the abstracted model. If the formula is proved false, related states are then successively refined, until the given formula is verified or computational resources are exhausted.

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. Proofs of correctness of some of these algorithms are given in the Appendix. 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 the basic definitions of abstract interpretation and model checking.

2.1 Abstract Interpretation

Abstract interpretation [8, 9] 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 domains, respectively. The *abstraction* function $\alpha : 2^{D_c} \rightarrow D_a$ maps a set of concrete values into an abstract value. α has an inverse, the *concretization* function $\gamma : D_a \rightarrow 2^{D_c}$. An abstraction is *valid* if the pair of functions (α, γ) forms a Galois connection:

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

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,

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

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

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

$\begin{pmatrix} 0 \\ - \\ + \end{pmatrix}$ 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, [8] introduced an abstract binary operator *widening*, denoted as $\overline{\nabla}$, which represents a "jump". For any abstract values i_0 and i_1, $i_0 \cup i_1 \subseteq i_0 \overline{\nabla} i_1$. [8] 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,
   - $[-1.5, 10] \overline{\nabla} [2, 44] = [-1.5, +\infty]$
   - $[22, -0.1] \overline{\nabla} [-10, -0.4] = [-\infty, -0.1]$

2.2 Model Checking

In this paper we will 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) [5]. 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 $L : S \to 2^P$. R must be a total function, i.e, $\forall s \in S,\ \exists t \in S$, 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 `mod` t_2, t_1 `exp` t_2.

`mod` 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 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) 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. Formally,

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

Note that $AF(\varphi) \equiv A[\mathit{True}\ U\ \varphi]$; and $EF(\varphi) \equiv E[\mathit{True}\ U\ \varphi]$, that is, we are using a "strong until". We also use the abbreviations $EG(\varphi)$ and $AG(\varphi)$ to represent the property that φ 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 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 line 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 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 this language. It also does not allow any user-defined (compound) data structures. A complete grammar of the language is available in [11]. Figure 2 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" [23]. We define a *state* to be the mapping between the 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 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, $w = (ln, (v_1, d_1), (v_2, d_2), \ldots, (v_n, d_n))$, 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. CTL properties about such programs can be expressed only in terms of global variables.

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 for 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 ($w^\alpha = \alpha(w)$), where each $d_i \in 2^{D_i}$ is mapped onto an abstract value ${D^\alpha}_i$. The result is an abstract state space W^α, in which each $w^\alpha \in W^\alpha$ is an n+1-tuple $w^\alpha = (ln, (v_1, {D^\alpha}_1), (v_2, {D^\alpha}_2), \ldots(v_n, {D^\alpha}_n))$. 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. Moreover, there is only one state associated with each line of code. Let R^α be a transition relation over the abstract state space, $R^\alpha \subseteq W^\alpha \times W^\alpha$. R^α 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 ${L^\alpha}_T, {L^\alpha}_F : W^\alpha \to 2^P$. 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 ${L^\alpha}_T \cup {L^\alpha}_F \subseteq 2^P$. The resulting abstract finite-state program is $PG^\alpha = (W^\alpha, R^\alpha, I^\alpha, P, {L^\alpha}_T, {L^\alpha}_F)$

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 p on a state such that $p(y)$ is *True* iff $\exists x, (x, y) \in R^\alpha$, $\exists g \in G$, 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^α in which 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^α (denoted $R^{\alpha *}$). s^α is defined recursively as follows:

- if $p(w^\alpha)$ is true, then $w^\alpha \in s^\alpha$.

```
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;
        print(``xy is '', xy);
        print(``b is '', b);
    }
}
```

Figure 2: A program fragment.

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

Note that values of global variables within s^α are the same. We will refer to L_T^α and L_F^α as 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. Finally, the transitions between states in S^α, $E^\alpha \subseteq S^\alpha \times S^\alpha$, are thus defined as:

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

Our abstract Kripke structure K^α = (S^α, E^α, I^α, 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. We also define a *predecessor* function 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 [3], is given in Figure 3. For example, a property $p \wedge q$ holds in state s^α if s^α is in Yes sets of both p and q. The same property does not hold in state s^α if s^α 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^α, then EXp holds in s^α; any immediate successor in which p may hold ($S^\alpha -$ No(p)) should be excluded from No(EXp). $A[pUq]$ is computed recursively as follows: $A[pUq]$ is *True* in all states S_0 in which q holds; it is also *True* in predecessors of S_0 in which p holds and all of which successors are in S_0. $A[pUq]$ is *False* in a state s^α iff q does not hold in s^α and either p does not hold in s^α or one of its successors does not lead to q.

4 Implementation

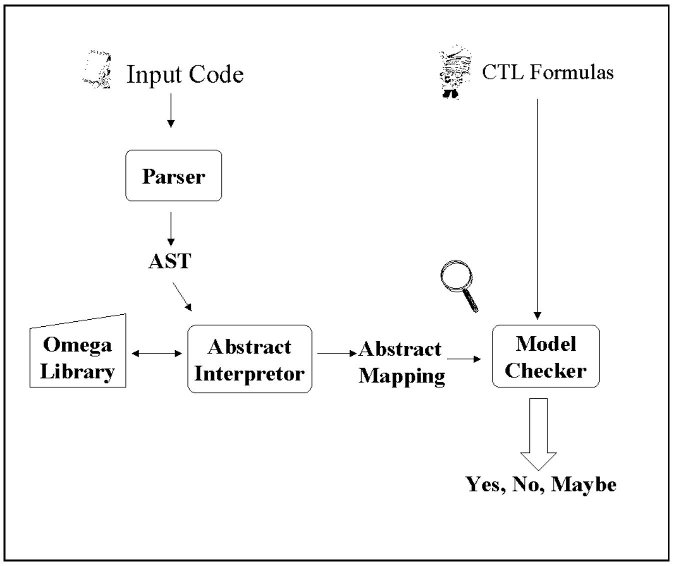

Figure 4: Architecture of the Abstract Model Checker.

We have implemented our Abstract Model Checker (AMC) as a 26,000-line C program. Figure 4 shows the architecture

Procedure CHECK(φ)
CASE

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

Figure 3: Model checking algorithm.

of our software. 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 $K^\alpha = (S^\alpha, E^\alpha, I^\alpha, P, L^\alpha{}_T, L^\alpha{}_F)$ 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]. However, values of infinite-domain 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 will associate each infinite-domain variable with a (finite) set of intervals, with the following interpretation

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

In the current implementation of AI, each set can consist of up to 5 intervals. 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 $m \leq n$:

$$\{a\} \overset{\alpha}{\cup} \emptyset = \{a\}$$
$$\{a\} \overset{\alpha}{\cup} \{b_1, ..., b_n\} = \{a \cup b_1, ..., a \cup b_n\}$$
$$\{a_1, ..., a_m\} \overset{\alpha}{\cup} \{b_1, ..., b_n\} = \{a_1 \cup ... \cup a_m\} \overset{\alpha}{\cup} \{b_1, ..., b_n\}$$
$$\{a\} \overset{\alpha}{\cup} \{b\} = \{b\} \overset{\alpha}{\cup} \{a\}$$

When we encounter two sets, each containing more than one interval, we first union elements of the set with less intervals (in this case, $\{a_1, ..., a_m\}$) 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 [11].

The algorithm used in our AI for analyzing conditional statements is depicted in Figure 5. Given an input abstract context S_i, a conditional expression *iexpr* and statements to

[1] For brevity, we do not discuss the treatment of *UNDEF* here. For details, please refer to [11]

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 is a set of C++ classes [21] for manipulating integer tuple relations and sets described by Presburger formulas. We use it for symbolically executing conditional expressions involving intervals.

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 5: Algorithm for analyzing conditional statements.

For example, suppose we are running our AI on the program fragment depicted in Figure 2 (Figure 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 $((\texttt{xy}, \{[-20, 52]\}), (\texttt{a}, \{[-5, 8]\}), (\texttt{b}, \{[13, 13]\}), (\texttt{c}, \{[2, 2]\}))$. 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 $\{[5, 25]\}$ 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 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 [11]. 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 this fixpoint occurs in a finite number of steps, we change values of variables in each loop at most 20 times, keeping track of whether they 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 2. At the first iteration, these values are $\{[13, 13]\}$ and $\{[-20, -20]\}$, respectively. In the following 19 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 7 shows the final Kripke structure built from the control-flow graph of Figure 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 7 is an aggregation of states 10 and 11 of Figure 6.

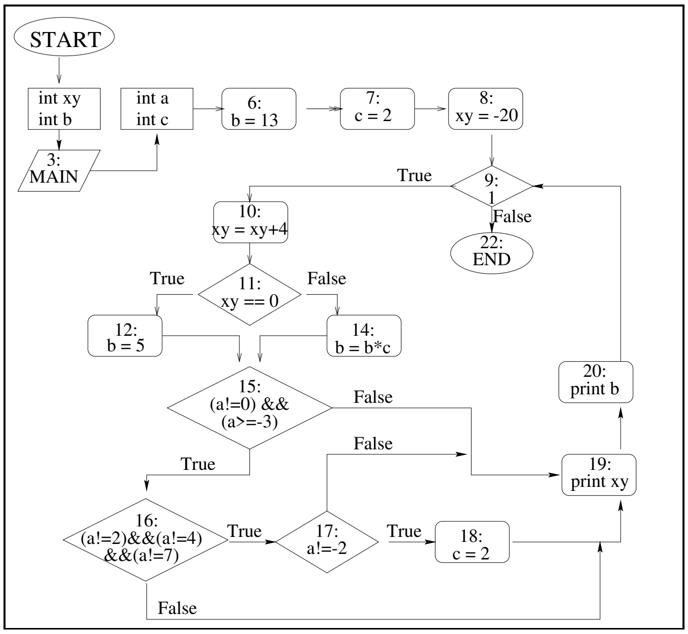

Figure 6: Control-flow graph of the program in Figure 2.

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

Table 1: Execution of the while loop of the program in Figure 2.

The resulting Kripke structure becomes input to the model checker whose algorithm is described in Section 3. For ex-

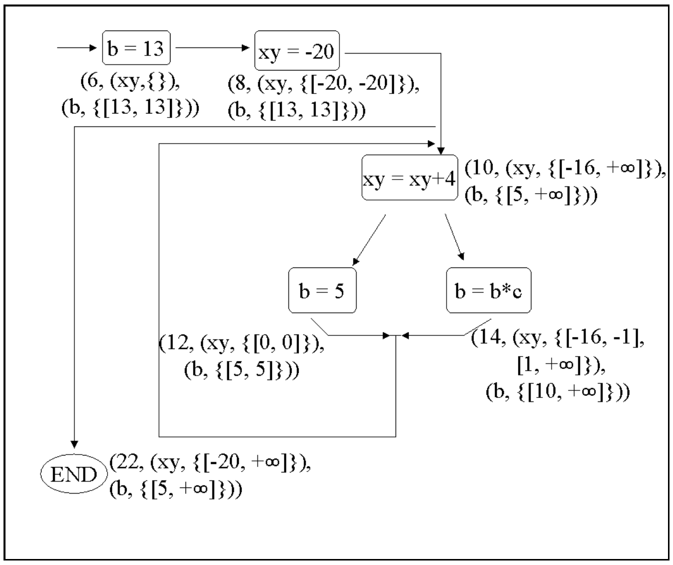

Figure 7: Kripke structure K^α built from the program fragment in Figure 2.

ample, we can model check the structure depicted in Figure 7 against CTL properties $AG((\texttt{xy} + \texttt{b}) \leq 0)$, $EF(\texttt{b} = 5)$, and $EF(\texttt{b} = 12)$. AMC returns *False* for the first property because it is violated in state corresponding to line 12 of the program. The second property is determined 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.

4.1 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 interpretor takes $20 \times |V| \times n \times m \times (n-1) \times |V|$ steps, which is $O(|V|^2 \times n^2)$.

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, 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 $O(|P| \times n^2)$.

5 Case Study

To determine the effectiveness of our abstract model checker, we analyzed the simplified version of a *Safety-Injection System* [7]. 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 allow 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 specification language of our system is expressive enough to capture complex properties of the Safety-Injection system, such as

1. $AG((\texttt{Reset} \wedge \texttt{Pressure} \neq \texttt{High}) \rightarrow \neg\texttt{Overridden})$
2. $AG((\texttt{Reset} \wedge \texttt{Pressure} = \texttt{TooLow}) \rightarrow \texttt{Injection})$
3. $AG((\texttt{Block} \wedge \neg\texttt{Reset} \wedge \texttt{Pressure} \neq \texttt{High})$
   $\rightarrow \texttt{Overridden})$
4. $AG((\texttt{Pressure} = \texttt{Permitted} \wedge \texttt{WaterPres} \leq \texttt{Permit})$
   $\rightarrow AX(\texttt{WaterPres} \geq \texttt{Permit}$
   $\rightarrow AF(\texttt{Pressure} = \texttt{High})))$

For example, Property 2 states that the system will inject the coolant if the pressure is too low and the reset button is pressed, whereas Property 4 states that 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.

We verified the above properties 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.

Safety-Injection has been verified by two other research groups. Bultan [2] built an infinite-state symbolic model checker that uses binary decision diagrams (BDDs) and a system of linear constraints to reason about models with Presburger arithmetic. This model checker also uses the Omega library together with abstract interpretation to achieve finite-time convergence of the analysis, but the properties are verified optimistically, allowing false positives. In addition, this procedure is partial, with the convergence dependent on the structure of the program and the formula to be verified. This approach does not utilize abstraction for state-space reduction. Bultan verified Properties 1 and 2, and we were not able to determine the exact size of his models. Bharadwaj and Heitmeyer [1] analyzed SCR specifications using the SPIN [15] model checker. Their technique only allows 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, allowing false negatives. The unabstracted system for Properties 1 and 2 (SPIN performs on-the-fly verification, without building a complete state space) consists of over 1.7 million states, whereas the combination of the above abstractions brings the state space down to 650 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 will 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.

(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. 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* [5] 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 Yu and Daniel House for many interesting discussions. We acknowledge the financial support of NSERC Postgraduate Scholarship.

Appendix

In this section, we give proofs of correctness of algorithms for checking $A[\varphi U \psi]$. See [11] for proofs of correctness of other algorithms. Let $(AM)_T(\phi, M^\alpha, s^\alpha)$ ($(AM)_F(\phi, M^\alpha, s^\alpha)$) indicate that our model checker returns *True* (*False*) when checking a formula ϕ in state s^α of the abstract model M^α. Assume:

$$(AM)_T(\varphi, M^\alpha, s^\alpha) \quad \Rightarrow \quad M, s \models \varphi \qquad (1)$$
$$(AM)_F(\varphi, M^\alpha, s^\alpha) \quad \Rightarrow \quad M, s \not\models \varphi \qquad (2)$$
$$(AM)_T(\psi, M^\alpha, s^\alpha) \quad \Rightarrow \quad M, s \models \psi \qquad (3)$$
$$(AM)_F(\psi, M^\alpha, s^\alpha) \quad \Rightarrow \quad M, s \not\models \psi \qquad (4)$$

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 φ and ψ. Prove:

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

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

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

The proof is by induction on the length of the path from s^α to a state in which ψ holds.
BASE CASE: length = 0.
$(AM)_T(A[\varphi U \psi], M^\alpha, s^\alpha)$
$\Rightarrow (AM)_T(\psi, M^\alpha, s^\alpha)$ **; (3)**
$\Rightarrow M, s \models \psi$
$\Rightarrow M, s \models A[\varphi U \psi]$
IH: Let T be the set of states from which ψ can be reached in $< n$ steps. Assume $\forall t_0^\alpha \in T, AM_T(A[\varphi U \psi], M^\alpha, t_0^\alpha)$ $\Rightarrow M, t_0 \models A[\varphi U \psi]$
PROVE: The formula holds for paths of length $\leq n$.
We will not consider the case where $AM_T(\psi, M^\alpha, s^\alpha)$ because it is covered by the base case.
$(AM)_T(A[\varphi U \psi], M^\alpha, s^\alpha)$ **; algorithm**
$\Rightarrow AM_T(\varphi, M^\alpha, s^\alpha) \ \wedge \ s^\alpha \in \mathrm{pred}(T) \wedge$
$s^\alpha \notin \mathrm{pred}(S^\alpha - T)$
$\Rightarrow AM_T(\varphi, M^\alpha, s^\alpha) \wedge$
$(\forall t^\alpha, \ s^\alpha \in \mathrm{pred}(t^\alpha) \Rightarrow (t^\alpha \in T))$ **; (IH), (1)**
$\Rightarrow M, s \models \varphi \wedge$
$(\forall t, \ s \in \mathrm{pred}(t) \Rightarrow M, t \models A[\varphi U \psi])$
$\Rightarrow M, s \models A[\varphi U \psi]$

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

$A[\varphi U \psi]$ does not hold in state s^α if either (1) ψ does not occur on some path emanating from s^α, or (2) on some path emanating from s^α, before the first occurrence of ψ, there is a state in which φ does not hold.
The algorithm can be expanded as follows:

T_0 = No(ψ)
Case (1): T_{i+1} = $T_i \cap$ pred(T_i)
Until $T_n = T_{n+1}$
Case (2): Q = $T_0 \cap$ No(φ)
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 it with those states that have successors in which $\neg\psi$ holds. The result is the set of states that can never reach ψ, 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 φ nor ψ hold, i.e. $M, s \models (\neg\varphi \wedge \neg\psi) \Rightarrow M, s \not\models A[\varphi U \psi]$.

References

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

[2] T. Bultan, R. Gerber, and C. 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.

[3] T. Bultan, R. Gerber, and W. Pugh. "Symbolic Model Checking of Infinite State Programs Using Presburger Arithmetic". In *Proceedings of International Conference on Computer-Aided Verification*, Haifa, Israel, 1997.

[4] W. Chan, R. J. Anderson, P. Beame, D. H. Jones, D. Notkin, and W. 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.

[5] E. Clarke, E. Emerson, and A. 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.

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

[7] 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.

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

[9] 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.

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

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

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

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

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

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

[16] G. 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.

[17] D. 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.

[18] D. Jackson and J. Wing. “Lightweight Formal Methods”. *IEEE Computer*, April 1996.

[19] W. Janssen, R. Mateescu, S. Mauw, P. Fennema, and P. van der Stappen. “Model Checking for Managers”. In *Theoretical and Practical Aspects of SPIN Model Checking, LNCS 1680*, pages 92–107, September 1999.

[20] M. Kamel and S. Leue. “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, November 1998.

[21] W. Kelly and W. Pugh. “The Omega Calculator and Library”. Technical report, University of Maryland, November 1996.

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

[23] M. Norrish. “An Abstract Dynamic Semantics for C”. Technical Report TR421-mn200, University of Cambridge Computer Laboratory, May 1997.

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

[25] T. Sreemani and J. M. Atlee. “Feasibility of Model Checking Software Requirements: A Case Study”. In *Proceedings of COMPASS’96*, Gaithersburg, Maryland, June 1996.