On Interpreting Results of Model-Checking with Abstraction

Marsha Chechik and Wei Ding

Department of Computer Science, University of Toronto,

Toronto, ON M5S 3G4, Canada.

Email: {chechik,wei}@cs.toronto.edu

Abstract

Model-checking offers a potential for push-button verification. Abstraction is often used to combat the state-space explosion problem and focus the analysis on relevant properties. However, it is often difficult to interpret the results of verification on an abstract system with respect to a concrete one.

In this paper we present an abstract model-checking approach that guarantees that the *True* and *False* answers are sound with respect to the original system. The approach is applicable to a number of situations, but we emphasize reasoning about source code, using abstract interpretation to build the abstraction of the system. We describe a simple implementation of an abstract model-checker and evaluate its effectiveness on a case study.

1 Introduction

Recent years have seen an increasing interest in automated techniques for analyzing correctness of software artifacts. In particular, this interest has been stimulated by the effectiveness and potential of model-checking [Clarke et al., 1986, Vardi and Wolper, 1986] – a technique for automatically verifying critical properties of systems, such as deadlock-freedom, security, fairness, etc. The model checking problem can usually be defined as follows: given a model M (of a program, a design, or a specification) and a property P expressed in a temporal logic, check $M \models P$, i.e., check if P holds in the model M.

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. Most of model checking technology is based on search, thus limiting the verification to finite-state systems, i.e. every variable in the system should have a finite, and fairly small, domain. Programs, on the other hand, typically have extremely large state-spaces. Since model-checking for temporal logics has a worst case complexity that is at least linear to the size of the state-space of the input transition system, the analysis procedure is likely to run out of memory when the worst case is realized. Thus, we refer to techniques for analyzing very large finite state systems which do not rely on the finiteness of the state space as *infinite-state* techniques. Several model checkers allow reasoning about infinite-state systems by "executing" all paths of the system up to a certain depth [Godefroid, 1997, Holzmann, 1997]. However, such techniques cannot guarantee that the system satisfies the desired property, and thus the analysis can only be used for debugging rather than assurance. Effectively, such techniques are using an implicit abstraction – "can an error be reached by a certain depth of unfolding of the state space".

To check realistic systems, an analyst has to use explicit abstractions, computed either automatically or by hand [Holzmann, 1999]. Verification by abstraction has been applied to a number of infinite-state systems as shown in [Dingel and Filkorn, 1995, Graf, 1995, Havelund and Shankar, 1996, Visser et al., 2000, Mueller and Nipkow, 1995]. In these systems, verifiers operate on the abstracted system, with the user-specified level of abstraction. The model checking problem thus becomes the following: given a model M and a temporal logic property P, compute an abstraction M^* of the model and an abstraction P^* of the property and check whether $M^* \models P^*$ [del Mar Gallardo and Merino, 1999]. For example, we can abstract a model and a property using a technique called *predicate abstraction* [Graf and Saidi, 1997]. Suppose we are interested in checking whether x is always positive (P = *always* $x > 0$). We can define predicates $Pos(x) \equiv (x > 0)$, $Neg(x) \equiv (x < 0)$ and $Zero(x) \equiv (x = 0)$ and use them to compute M^*. For example, if $Pos(x)$ was true before a statement $x = x + 3$, it is still true after this statement. After M^* is constructed, we can check it against P^* = *always* $Pos(x)$.

When the abstraction is used to assist verification, it is highly desirable that properties hold on the abstracted model if and only if they hold in the original one [Clarke et al., 1994] ($M^* \models P^*$ iff $M \models P$). Such an abstraction is called *strong property-preserving*. However, such an assurance is difficult to obtain: a different abstraction has to be built for each class of properties under analysis [Dams et al., 1997] and often theorem-proving needs to be used to ensure that the abstraction is built correctly [Rushby, 1999]. Therefore, most of the time the users cannot trust some of the answers received from the model-checker (and are not sure which answers to trust and which not to). Since model-checking enables people without extensive training in formal methods do verification effectively, it is essential that we provide computer support to enable users to correctly interpret results of verification on abstracted systems.

In this paper we propose a "strong property-preserving"[1] approach for doing CTL model-checking on abstracted systems. The approach consists of a class of over-approximating abstractions and a model-checking algorithm. The class of abstractions is given in Section 3.3, and the computation of one such abstraction that uses interval analysis computed via abstract interpretation [Cousot and Cousot, 1976], is outlined. Unlike classical algorithms, our model-checker uses three-valued logic for its computation. It yields *True* if it can determine that the property holds in the original system, *False* if it can determine that the property does not hold, and *Maybe* in all other cases. Thus, the analysis is sound with respect to both *True* and *False* answers, which enables users to correctly interpret results of their analysis and determine when the analysis is inconclusive. Note that usually the term "strong property preservation" is associated with a particular abstraction [Dams et al., 1994, Visser et al., 2000] (it is assumed that the abstract system is analyzed using a classical model-checker), whereas our approach associates it with a combination of a class of abstractions and a model-checking algorithm.

The rest of the paper is organized as follows: in Section 2 we give an overview of abstraction techniques used in this paper and survey related work. Section 3 gives the main contribution of this paper: it describes a class of abstractions and the abstract model-checking algorithm and argues for its correctness. The proof of correctness is given in Appendix A. Section 4 gives an example of abstraction and model-checking. Sections 5 and 6 describe our experience with implementing and using an abstract model-checker. Section 7 contains concluding remarks and outlines future areas of research.

2 Abstraction for Verification

A common application of abstraction is to reduce the size of the state space of a program in order to make a verification algorithm more tractable and to enable reasoning about infinite-state systems. There are two main approaches to model-checking using abstraction [Visser et al., 2000]: an abstract state graph can be generated for model checking by executing the concrete transitions over the abstract data, or the concrete transitions can be abstracted and the resulting abstract system can be model checked. The former case is

[1] The meaning of the term here is somewhat overloaded, because our analysis is three-valued, not two-valued, as discussed below.

usually more precise because it uses dynamic information about the abstract state-space. The most widely used technique for building such abstractions is *abstract interpretation*.

In this section we discuss the notion of property preservation and its relation to over- and under-approximating abstractions. We then survey some of the related work in the light of property preservation. The section is concluded with a brief description of abstract interpretation and how to use it for model-checking.

2.1 Property Preservation and Abstractions

The main concern with using abstraction to aid verification is to ensure that abstractions are *property-preserving*. As we mentioned in the introduction, abstracted systems are typically verified using classical model-checkers. Thus, in the literature, property preservation is associated with the abstraction and a class of temporal properties. We discuss two forms of property-preserving abstractions [Visser et al., 2000] below:

Weak Preservation – An abstraction is weak property-preserving if a set of properties *True* in the abstract system has corresponding properties in the concrete system that are also *True*, i.e.

$$M^* \models P^* \;\Rightarrow\; M \models P$$

Strong Preservation – An abstraction is strong preserving if a set of properties with truth values either *True* or *False* in the abstract system has corresponding properties in the concrete system with the same truth values, i.e.

$$M^* \models P^* \;\Rightarrow\; M \models P \text{ and } M^* \not\models P^* \;\Rightarrow\; M \not\models P,$$

or, if you prefer,

$$M^* \models P^* \;\Leftrightarrow\; M \models P.$$

The common wisdom is that strong preservation is fairly hard to achieve, and that only abstractions that are weak-preserving have a real chance of aggressively reducing the state space of the system.

When systems are abstracted, it is done either by under-approximating or by over-approximating. In the former case, the abstract system represents fewer behaviors than are present in the concrete one. For example, we can reduce the size of a buffer, decrease the number of processes in the system, etc. This abstraction, often referred to as "the machete approach" [Bensalem et al., 2000] preserves only existential properties such as reachability (EF(p)). However, this method is quite effective, and is often applied when model-checking is used for debugging rather than assurance. Indeed, if a bug manifests itself on a simplified system, e.g. a system with three processors instead of twenty, it is likely to persist in a larger system. Testing is the most common example of under-approximating techniques for the environment of the system [Visser et al., 2000], replacing a potentially infinite number of possible behaviors with a finite number of test cases.

Over-approximation of the behaviors of the system occurs when the abstracted system represents more behaviors than are present in the concrete one. This approach is weak property-preserving for universally-quantified properties, e.g. properties expressed in $\forall$CTL and LTL. If more behaviors are added and a universally-quantified property is *True*, then it will still be *True* on a concrete system. This method works well with safety properties (e.g., "the aircraft will never land when its chasse is up") but often leads to false negatives with reasoning about liveness properties (e.g., "every request to use the printer will eventually be serviced"). Spurious errors can be removed by constraining the over-approximation, but it is hard to do; this problem is similar to making static analysis more precise [Schmidt, 1998].

2.2 Related Work

The idea of verification with the presence of abstraction has been explored by several researchers. Most of such abstractions are *weak* property-preserving, e.g. [Jackson, 1994, Clarke et al., 1995, Kelb et al., 1995, Colon and Uribe, 1998, Saidi, 1999]. The approach is typically 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. We discuss some of the *strongly* property-preserving approaches below.

Bharadwaj and Heitmeyer [Bharadwaj and Heitmeyer, 1999, Heitmeyer et al., 1998] analyzed SCR specifications using the SPIN [Holzmann, 1997] model checker. The system is checked against state and transition invariants. The size of the concrete state space is reduced by three abstraction methods: eliminating variables which are not relevant to the property being verified, i.e. slicing (SCR ensures that dependencies between variables form a partial order); replacing raw input variables with results of intermediate computation, i.e., removing irrelevant preprocessing, and replacing input variables with predicates. The first two abstractions are strong property-preserving (although the latter one is manual), whereas the third one is weak property-preserving.

Bultan [Bultan et al., 2000, Bultan et al., 1999, Bultan et al., 1998] and his collegues 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 converges, is strong property-preserving. 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 2.3 below) that allows 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). This approach relies on *widening* (see Section 2.3 below) to achieve finite-time convergence of the fixpoint computation.

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.

2.3 Abstract Interpretation

Given a concrete 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 variable analysis [Aho et al., 1988], are examples of abstract interpretation. In particular, abstract interpretation can be used [Chechik and Ding, 2001] 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 domain, 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:

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

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

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

In this case, the vocabulary[2] of M^*, the resulted abstract system, is different from the one of M.

Another abstraction constitutes going from a set of values that a variable may have at a certain point of the program to an interval by taking the minimum (maximum) value from the set as the left (right) bound of the interval. This is known as *Galois insertion*. For example,

- α({-1}) = [-1, -1]
- α({0.5, 1.3, 23}) = [0.5, 23]

Here, the vocabulary of M^* is the same as M. Both refer to concrete variables and their values.

Techniques have been developed [Cousot and Cousot, 1977] to enable finite representations of infinite sets of values and to ensure convergence of the computation of the abstract model in a finite number of steps.

2.4 Interpreting Verification Results

Abstract interpretation, discussed in Section 2.3, typically changes the vocabulary of the system and yields approximate results that lead either to over-approximation or to under-approximation. In particular, the interval analysis is an over-approximating technique that typically does not preserve existential properties.

The approach is to use three-valued logic for determining validity of atomic propositions. Suppose we have abstracted a program M to M^* using sign analysis and thus have the information that x is $(+)$ in the current state s. This information is sufficient to conclude that $x < -2$ is *False* in s, whereas $x > -2$ is *True*. However, we cannot conclude anything about the value of the proposition $x > 4$; thus, we will assign it the value *Maybe*. Note that the vocabulary of P, the property we are interested in, and P^*, the property we end up checking, are the same.

However, for the above approach to work for verification of temporal properties, we need a model-checking algorithm combining the results of checking atomic properties on individual states. If the abstraction introduces new transitions into the system (because several states are abstracted into a single one), the resulting technique cannot be strong property-preserving – some states in M^* will have more

[2] A vocabulary of a system is its variables and their types.

successors than in M. Thus, our proposed approach can only work if the abstraction handles transitions between states correctly.

3 Abstract Model Checking

We start this section by briefly describing the CTL model-checking. We proceed by outlining the three-valued logic that we use for for combining model-checking with abstraction. Then we describe our verification approach. As we mentioned earlier, in this paper we are interested in abstractions which strongly-preserve arbitrary CTL properties. To achieve this, we formalize the ideas presented in Section 2 by defining a class of abstractions (Section 3.3) and doing away with the classical model-checker, replacing it with a three-valued one (Section 3.4). We conclude the section with the analysis of correctness and running time of our model-checking algorithm.

Note that the algorithm we are going to develop may yield inconclusive *Maybe* answers. Note also that an algorithm that always returns *Maybe* satisfies the above definition (since we never receive *Yes* or *No* answers) and is therefore considered correct. Philosophically, we feel that it is a good idea for a model-checker to "give up" when faced with a problem it is unable to solve. In fact, regular model-checkers do this all the time when they run out of memory, making the attempted analysis inconclusive. Our model-checker will do the same, but will not require significant time/space resources for that.

3.1 CTL Model-Checking

CTL model-checking is 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 $M, s \models P$ indicates that a formula P 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 $I \in S$, a set of atomic propositions A, and a labeling function $L : S \rightarrow 2^A$. R must be total, 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 $(s_n, s_n) \in R$. For each $s \in S$, the labeling function provides a list of atomic propositions which are *True* in this state.

CTL is defined as follows:

1. Every atomic proposition $a \in A$ 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,

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

where the remaining operators are defined as follows:

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

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.

When CTL is used to express properties of programs, it usually involves properties combining temporal operators and arithmetic on variables and their values, e.g. "a is never less than b", "immediately after

$a + b > 5$, c will be *True*". Of course, the exact language of atomic propositions depends on the choice of the abstraction.

3.2 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 $a \wedge b \equiv a \text{ if } a \sqsubseteq b \text{ ; } b \text{ otherwise}$. For example, *False* $\wedge$ *Maybe* $=$ *False*. Disjunction is defined as $a \vee b \equiv \ a \text{ if } b \sqsubseteq a \text{ ; } a \text{ otherwise}$. 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]. Finally, material implication is defined as

$$a \to b \equiv \neg a \vee b$$

For example, *Maybe* $\to$ *False* $= \neg$*Maybe* $\vee$ *False* $=$ *Maybe*.

3.3 Abstractions

Here we describe a family of over-approximating abstractions which can be used in conjunction with our abstract model-checker. Computation of one such abstraction is discussed in detail in [Chechik and Ding, 2001]. We also give an illustration of this abstraction in Section 4.

Abstraction starts with an (infinite-state) model (program, specification, design) $M = (S, I, E, A, L_T, L_F)$, where S is a (infinite) set of states, $I \in S$ is the initial state, $E \subseteq S \times S$ is the total transition relation, A is the set of atomic propositions, 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 ($L_T, L_F : S \to 2^A$). Naturally, for each state $s \in S$, $L_T(s) \cup L_F(s) = A$ and $L_T(s) \cap L_F(s) = \emptyset$.

The goal of the abstraction is to create an abstract Kripke structure $M^* = (S^*, s_0^*, E^*, A^*, L^*{}_T, L^*{}_F)$. We would like this structure to have a finite state space and use the same vocabulary of atomic propositions ($A^* = A$). In addition, the abstraction *Abs*($M \to M^*$) should be *valid*, i.e., allow correct interpretation of properties.

3.3.1 Abstraction of the State Space

Here we discuss requirements for building the abstract state space S^*. First, we want to ensure that S^* is an over-approximation of S (the assumption of the Galois connection framework of Section 2.3), that is,

$$\forall s \in S \cdot \exists s^* \in S^* \cdot s^* = \alpha(s)$$

To infer results of the abstraction correctly, the analyst needs to determine an abstraction function α and the correct interpretation of the results of abstraction in each state: functions L_T^* and L_F^*. These functions should be "sound", i.e.

$$\begin{aligned} \forall s^* \in S^* \cdot \quad & L_T^*(s^*) \cap L_F^*(s^*) = \emptyset \wedge \\ & L_T^*(s^*) \cup L_F^*(s^*) \subseteq A^* \end{aligned}$$

Recall that we want $A^* = A$. Further, we require that

$$\begin{aligned} \forall s \in S^* \cdot \forall s^* = \alpha(s) \cdot \quad & L_T^*(s^*) \subseteq L_T(s) \wedge \\ & L_F^*(s^*) \subseteq L_F(s) \end{aligned}$$

Suppose that the analyst chooses "sign analysis" (see Section 2.3) as the abstraction, and it was determined that in a given state s, x is positive and y is negative. Then, "x>-2" $\in L_T^*(s)$, but we cannot safely determine the value of "x+y>5". So, "x+y>5" $\notin (L_T^*(s) \cup L_F^*(s))$, that is, this proposition has value *Maybe*, or it is unknown whether it holds in the concrete system.

An abstraction function α should be chosen so that the abstraction can be constructed in a finite number of steps (i.e., there should not be an infinite increasing chain in the lattice of abstract values [Cousot and Cousot, 1976]). Further, a bad choice of the abstraction may result in most interesting atomic properties evaluating to *Maybe*.

3.3.2 Abstraction of the Transition Relation

As we build our abstraction, it is essential 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 split E^*, the desired abstraction of E, into a tuple (E_T^*, E_M^*). E_T^* captures *definite* transitions, i.e.

$$(s^*, t^*) \in E_T^* \quad \rightarrow \quad \forall (s \in \gamma(s^*), t \in \gamma(t^*)) \cdot (s, t) \in E$$

whereas E^*_M captures *possible* transitions, i.e.

$$(s^*, t^*) \in E^*_M \quad \rightarrow \quad \exists(s \in \gamma(s^*), t \in \gamma(t^*)) \cdot (s, t) \in E$$

Clearly, $E^*_T \subseteq E^*_M$. Further, we require that

$$\forall s, t \in S \cdot (s, t) \in E \rightarrow (\alpha(s), \alpha(t)) \in E^*_M$$

Often, the resulting system needs to be further abstracted to ensure that each transition represents a change to a globally-visible variable. We give an example of such an abstraction in Section 4.

3.4 Abstract model-checking algorithm

We now present the algorithm that receives a Kripke structure M^* constructed above and a CTL property P, and determines whether or not the property holds in the concrete system M.

The algorithm recursively goes through the structure of P, associating each subproperty φ with a pair of sets of states (Yes(φ), No(φ)). Yes(φ) $\subseteq S^*$ is a set of states in which φ is *True*, or, more formally,

$$s^* \in \text{Yes}(\varphi) \equiv \varphi \in {L^*}_T(s^*)$$

No(φ), which represents a set of states in which φ is *False*, is defined similarly. φ is considered to have a value *Maybe* hold in a state s^* if $s^* \in (S^* - (\text{Yes}(\varphi) \cup \text{No}(\varphi)))$. We also define two *predecessor* functions: $\text{pred}_T : 2^{S^*} \rightarrow 2^{S^*}$ takes a set of states Q and returns all states that can reach some state in Q in one *True* transition:

$$s^* \in \text{pred}_T(Q) \equiv \exists t^* \in Q \wedge (s^*, t^*) \in E^*_T$$

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:

$$s^* \in \text{pred}_M(Q) \equiv \exists t^* \in Q \wedge (s^*, t^*) \in E^*_M$$

The algorithm, inspired by Bultan's symbolic model checker for infinite-state systems [Bultan et al., 1997], is given in Figure 1. For example, a property $\varphi \wedge \psi$ holds in state s^* if s^* is in Yes sets of both φ and ψ. The same property does not hold in state s^* if s^* is in the No set of either φ or ψ. When verifying $EX\varphi$, we note that if φ holds in some immediate *True* successor of state s^*, then $EX\varphi$ holds in s^*; any immediate *Maybe* successor in which φ may hold (S^*- No(φ)) should be excluded from No($EX\varphi$).

Procedure CHECK(M^*, 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)), $S^* -$ pred$_M$($S^* -$ No(φ)))
$p = AX\varphi$	:	Return ($S^* -$ pred$_M$($S^* -$ 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)
		$N_{i+1} = N_i \cap$ (($S^* -$ pred$_M$($S^* - N_i$)) $\cup$ 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$_M$($S^* - Y_i$)) $\cap$ 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 1: Model Checking Algorithm.

$A[\varphi U \psi]$ is computed recursively as follows: $A[\varphi U \psi]$ is *Yes* in all states Y in which ψ holds; it is also *Yes* in *True* predecessors of Y in which φ holds and all of whose *True* successors are in Y. $A[\varphi U \psi]$ is *No* in a state s^* iff ψ does not hold in s^* and either φ does not hold in s^*, or one of its *Maybe* successors does not lead to ψ.

Theorem 1 *Let* Abs*:* $M \to M^*$ *be a valid abstraction and* p *be an arbitrary CTL formula. Then,*

$$\begin{aligned} Check(M^*, p) = \textit{True} &\quad \to \quad (M \models p) \\ Check(M^*, p) = \textit{False} &\quad \to \quad (M \not\models p) \end{aligned}$$

Proof:

Refer to Appendix A for the proof of this theorem. □

See [Ding, 2000] for the remaining proofs of correctness and Section 4 for a concrete example.

3.5 Running Time

To determine the running time of our algorithm, we note that this time is dominated by the fixpoint computations of EU and AU. Consider computing a Yes set for a property $A[\varphi U \psi]$. Y_i can change value at most $|S^*|$ times before a fixpoint is reached. Each step of the algorithm includes two computations of predecessors (each bounded above by $|S^*|^2$), and a difference, union, and intersection operation, each taking $O(|S^*|)$. We are also assuming that L_T^* and L_F^* are constant-time functions, and thus do not affect the computation time. Computation of a No set for this property also converges in at most $|S^*|$ iterations, with each one taking $O(|S^*|^2)$ time. EU has the same running time. Since a given CTL formula p contains at most $|p|$ different subformulas, then the total running time is $O(|p| \times |S^*|^3)$. If instead of $|S^*|$ we use n to represent the size of the abstract model, which includes $|S^*| + |E^*|$, then the overall running time is $O(n^2 \times |p|)$, which is the expected result for CTL model-checking [Clarke et al., 1999]. Finally, we expect that $S^* << S$, and $E^* << E$, so the running time of the abstract model-checker is significantly smaller than that of the concrete one.

Note also that the verification process builds upon the abstract interpretor, i.e., this time complexity does not include the effort that the abstract interpretor takes to generate the abstract finite-state program.

4 Example

In this section we illustrate the concepts described earlier in the paper using a simple sequential program given in Figure 2.

To start, we need to define a concept of *program state*.

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

Figure 2: A program fragment.

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

There are a number of ways of abstracting program states. Suppose we are dealing with a sequential program M with n state changes *in its text* (e.g., one state change per line, without expanding loops). A possible state space in the abstract model M^* is the one that includes up to n states, where each variable is associated with an abstraction of the set of values it can have if the control reaches that point in the program.

We choose an abstraction that ensures that L_T^α and L_F^α are constant-time. One such abstraction is interval analysis – keeping track of an interval of values that each global variable can take at each point of the program. For example, if in a given state the value of x is $[-3, 3]$, and the value of y is $[0, 12]$, then $x < 5$ is *True*, $x + y > 5$ is *False*, and $(x + 2) * 4 = y$ is *Maybe*. Figure 3 shows the control-flow graph for the program fragment in Figure 2, with each state associated with the program line number.

The transition relation is built using the control-flow graph of the program and the abstraction of the state space. Clearly, if two nodes (s^*, t^*) in a control-flow graph do not have an edge between them, there is no transition between these states in the concrete system. Thus, a possible transition relation is the one in which each edge of the control-flow graph is in E_M^*, and $E_T^* = \emptyset$. However, the analysis can be

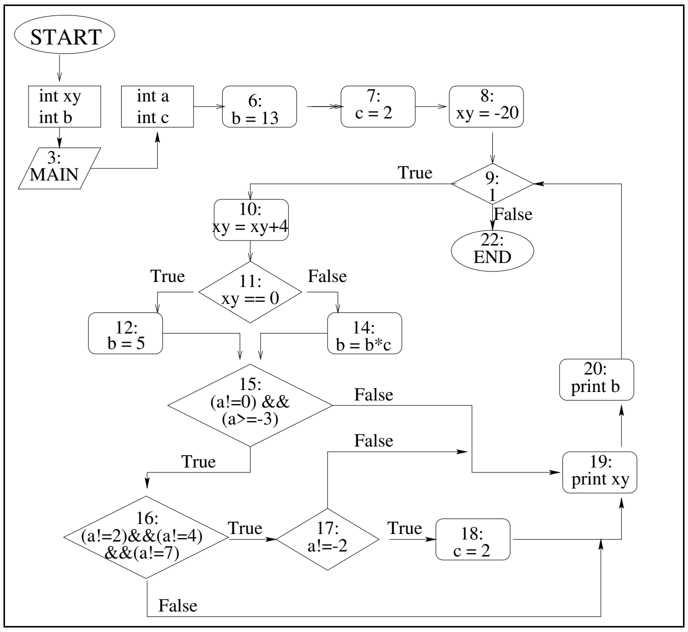

Figure 3: Control-flow graph of the program in Figure 2.

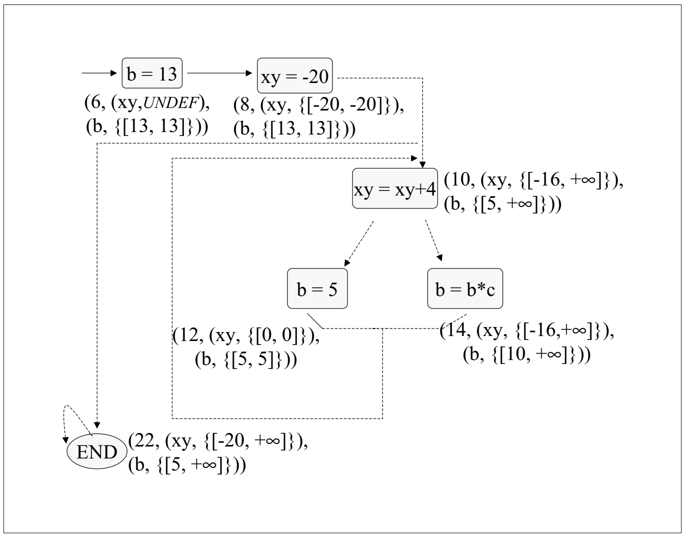

Figure 4: Kripke structure K^α built from the program fragment in Figure 2.

improved by moving an edge (s^*, t^*) into E_T^* if it can be guaranteed that $(\gamma(s^*), \gamma(t^*)) \in E$. Consider the example in Figure 3. Variables in states 6 and 7 have singleton values; thus, a transition between these states is a definite one and should be in E_T^*.

Then we construct an abstract finite-state program in which every state is a mapping of variables to the values they can attain when the program control reaches this point. We note that the above fragment has only two global variables, `b` and `xy`. Thus, we further abstract the model, resulting in a Kripke structure in Figure 4. The example in Figure 2 is chosen so that there is at most one state change per each line of code. Therefore, in Figure 4 we can associate each state 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 program point. For example, state 10 of Figure 4 is an aggregation of states 10 and 11 of Figure 3. 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*.

The resulting Kripke structure becomes input to the model checker whose algorithm is described in Section 3.4. For example, we can model check the structure depicted in Figure 4 against CTL properties $AG((\texttt{xy} + \texttt{b}) \leq 0)$, $EF(\texttt{b} = 5)$, $EF(\texttt{b} = 12)$, and $AG(\texttt{xy} = -20 \rightarrow EX(\texttt{xy} \geq -20))$. The 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 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. The fourth property is also determined to be *Maybe*. The left-hand side of the implication is *True* only in state 8. This state has two successors, 22 and 10, and $\texttt{xy} \geq -20$ is *True* in both of them. However, both states are only reachable via a *Maybe* transition, and thus the property evaluates to *Maybe*.

5 Implementation

To show the feasibility of our approach, we have built a simple model-checking system [Ding, 2000, Chechik and Ding, 2001]. The system takes a sequential program expressed in a subset of C (without pointers and recursion and with limited data structures) and allows to specify and verify CTL properties involving arithmetic ($+$, $-$, $*$, `div`, `mod`, `exp`) and logical operations ($=$, $\neq$, $>$, $<$, $\leq$, $\geq$) on the values of global variables of the program. $x > 5$ and $(x + 2) * 4 = y$ are some of the atomic propositions in this version of CTL.

As in the example in Section 4, we select an abstraction that insures that L_T^* and L_F^* are constant-time.

Our abstraction associates each variable of the program with a *set* of intervals representing values that this variable can take if the control reaches this point of the program. This abstraction is somewhat more precise than the regular interval analysis and still over-approximating. Operations on sets of intervals can be easily constructed and are described in detail elsewhere [Ding, 2000].

Expressions are evaluated using the Omega library/calculator [Kelly et al., 1996, Pugh, 1992]. The Omega library/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.
$\{[i, j] \rightarrow [j, j'] : 1 \leq i < j < j' \leq n\}$	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 Omega library for symbolically executing conditional expressions involving intervals; thus the type system is limited by boolean and integer values. For example, suppose we are running our analysis on the program fragment depicted in Figure 2, and the value of `xy` is $[-20, 52]$ before executing the conditional statement `xy == 0` in state 11. We are interested in values of this variable in the input contexts for states 12 and 14, which, using a call to the Omega library, evaluate to $\{[0, 0]\}$ and $\{[-20, -1], [1, 52]\}$, respectively.

The final result may include too many disjuncts. To ensure convergence, we also assign an upper bound of disjoint intervals we may wish to represent (same approach is used in [Bultan et al., 1999]). If the bound is reached, the intervals are unioned, and the analysis continues. The technique introduces a loss of precision, but is strictly conservative. Furthermore, this number is set to 30 in the implementation, and in the experiments we conducted so far, we seldom had to resort to this technique.

The transition relation is computed on top of the Control-Flow graph, as discussed in Section 4.

The abstraction takes $O(|V|^2 \times n^2 + n \times |V| \times \omega(|V|)$ steps, where $|V|$ is the total number of variables in the program under analysis, n is the number of state changes in its text, and $\omega(|V|)$ is the number of steps taken by the Omega library for the variables in V. Although in the worst case the performance of the Omega library is triple-exponential, we never achieve the pathological cases. First of all, calls to the calculator are made only for deciding conditional expressions (i.e., not quantification or uninterpreted functions), and these are composed of a fairly small number of variables. Furthermore, 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.

6 Case Study

To determine the effectiveness of our abstract model checker, we analyzed the implementation of 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".

This system has been analyzed by Bharadwaj and Heitmeyer [Bharadwaj and Heitmeyer, 1999] using SPIN and a set of abstractions built into SCRTool (see Section 2.2) and by Bultan [Bultan et al., 1998] using his infinite-state model-checker. These researchers identified the following properties of the Safety-Injection System:

1. $\texttt{Overridden}$ will become *False* if the system has been reset and the pressure is not high.

   $AG((\texttt{Reset} \ \wedge \ \texttt{Pressure} \neq \texttt{High}) \rightarrow \neg\texttt{Overridden})$

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

   $AG((\texttt{Reset} \ \wedge \ \texttt{Pressure} = \texttt{TooLow}) \rightarrow \texttt{Injection})$

We have implemented the Safety-Injection system as a 116-line C program with 8 global variables closely reflecting those of the specification `WaterPres` of type integer, `Block` and `Reset` of type boolean, `Injection` of type boolean, `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. See Appendix B for the source code of this implementation.

Using our Abstract Model-Checker we represented the Safety Injection system compactly using only 30 states, and were able to verify properties 1 and 2 successfully, i.e., both evaluated to *True*. In addition, we tried verifying three other properties:

3. $AG((\neg\texttt{Block} \ \wedge \ \texttt{Pressure} = \texttt{TooLow}) \rightarrow \texttt{Injection})$
4. $AG(\texttt{WaterPres} \geq 0)$
5. $AG((\texttt{Pressure} = \texttt{Permitted} \ \wedge \texttt{WaterPres} \leq \texttt{Permit})$

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

Property 3 was identified in [Bharadwaj and Heitmeyer, 1999] as *False*, and properties 4-5 were chosen to include an infinite-state variable `WaterPres`. Our model-checker returned *Maybe* for properties 3 and 4, indicating an inconclusive result, and *True* for property 5. For a more detailed description of the case study, please refer to [Ding, 2000].

7 Discussion and Conclusion

In this paper we presented a CTL model-checking algorithm which allows reasoning about abstracted systems. The algorithm works for an arbitrary CTL property and a family of over-approximating abstractions, and allows to correctly interpret results of the verification. When the algorithm yields *Yes* (*No*), the property is *True* (*False*) in the concrete system. In cases where the analysis is inconclusive, the algorithm yields *Maybe*.

The effectiveness of the analysis depends on the abstraction function and on the sophistication of the abstraction propagation algorithm. The simplest strong property-preserving model-checking algorithm is the one that always returns *Maybe*. And clearly, a poorly chosen abstraction may achieve nothing more than this in our approach. More effective abstraction and verification algorithms are likely to be more expensive in terms on the work required by the user and the computing resources required.

In this paper we demonstrated the effectiveness of our approach by building a fairly cheap abstract interpretor that performs interval analysis. The model-checking algorithm based on this interpretation was conclusive on several properties of the Safety Injection System, although it yielded *Maybe* on some properties.

The work presented here can be extended in a number of directions. In particular, we hope to develop an abstract model-checking algorithm for CTL* [Clarke et al., 1986]. CTL* combines branching-time and linear-time operations and is strictly more expressive than CTL. Also, we are interested in benchmarking abstraction functions and analysis tools based on them in terms of complexity and conclusive results. Our sample implementation of an abstract model-checker needs a number of improvements, e.g. using multiple abstraction functions, tightening up reasoning about abstract values, allowing a more expressive input language. We plan to integrate our model-checking algorithm with the Bandera toolkit [Corbett et al., 2000] that enables abstract interpretation w.r.t. multiple abstractions.

Acknowledgments

We would like to thank Steve Easterbrook and the anonymous referees for helping us improve the presentation of this paper. We are also grateful to Mark Pichora, Albert Lai, Ric Hehner, Radu Iosif, Daniel House, Benet Devereux and Arie Gurfinkel for many interesting discussions.

References

[Aho et al., 1988] Aho, A., Sethi, R., and Ullman, J. (1988). *Compilers: Principles, Techniques, and Tools, Chapter 10*. Addison Wesley.

[Bensalem et al., 2000] Bensalem, S., Ganesh, V., Lakhech, Y., Munox, C., Owre, S., Rueb, H., Rushby, J., Rusu, V., Saidi, H., Shankar, N., Singerman, E., and Tiwari, A. (2000). “An Overview of SAL”. In *Proceedings of LMF2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196.

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

[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*.

[Chechik and Ding, 2001] Chechik, M. and Ding, W. (2001). “Lightweight Reasoning about Program Correctness”. In *Proceedings of CASCON*. To appear.

[Chechik et al., 2001] Chechik, M., Easterbrook, S., and Petrovykh, V. (2001). “Model-Checking Over Multi-Valued Logics”. In *Proceedings of 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., 1999] Clarke, E., Grumberg, O., and Peled, D. (1999). *Model Checking*. MIT Press.

[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.

[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.

[del Mar Gallardo and Merino, 1999] del Mar Gallardo, M. and Merino, P. (1999). “A Framework for Automatic Construction of Abstract Promela Models”. In *Proceedings of the 6th International SPIN Workshop (SPIN’6)*, volume 1680 of *LNCS*, pages 184–199. Springer-Verlag. In association with FM’99: World Congress in Formal Methods.

[Dijkstra and Scholten, 1990] Dijkstra, E. and Scholten, C. (1990). *Predicate Calculus and Program Semantics*. Springer.

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

[Dingel and Filkorn, 1995] Dingel, J. and Filkorn, T. (1995). “Model Checking for Infinite State Systems using Data Abstraction, Assumption-commitment Style Reasoning and Theorem Proving”. In *Proceedings of 7th International Conference on Computer-Aided Verification (CAV’95)*, volume 939 of *LNCS*, pages 54–69. Springer-Verlag.

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

[Graf, 1995] Graf, S. (1995). "Characterization of a Sequentially Consistent Memory and Verification of a Cache Memory by Abstraction". *Distributed Computing*.

[Graf and Saidi, 1997] Graf, S. and Saidi, H. (1997). "Construction of Abstract State Graphs with PVS". In *Proceedings of the 9th International Conference on Computer-Aided Verification*, volume 1254 of *LNCS*. Springer-Verlag.

[Havelund and Shankar, 1996] Havelund, K. and Shankar, N. (1996). "Experiments in Theorem Proving and Model Checking for Protocol Verification". In *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051, page P. 662. Springer-Verlag.

[Heitmeyer et al., 1998] Heitmeyer, C., Kirby, J., Labaw, B., Archer, M., and Bharadwaj, R. (1998). "Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications". *IEEE Transactions on Software Engineering*, 24(11).

[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.

[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 Model Process Logic”. In *Third Annual Symposium on Logic in Computer Sciences*, pages 203–210. IEEE Computer Society Press.

[Mueller and Nipkow, 1995] Mueller, O. and Nipkow, T. (1995). “Combining Model Checking and Deduction for I/O-Automata”. In *Tools and Algorithms for the Analysis and Construction of Systems (TACAS'95)*, volume 1019 of *LNCS*, page 1. Springer-Verlag.

[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*.

[Rushby, 1999] Rushby, J. (1999). “Integrated Formal Verification: Using Model Checking with Automated Abstraction, Invariant Generation, and Theorem-Proving”. In *Proceedings of the 5th International SPIN Workshop (SPIN'5)*, volume 1680 of *LNCS*, pages 1–11. Springer-Verlag.

[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.

[Schmidt, 1998] Schmidt, D. (1998). “Data Flow Analysis is Model Checking of Abstract Interpretations”. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL'98)*.

[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.

[Vardi and Wolper, 1986] Vardi, M. Y. and Wolper, P. (1986). “An Automata-Theoretic Approach to Automatic Program Verification”. In *Proceedings of 1st Symposium on Logic in Computer Science*, pages 322–331, Cambridge MA.

[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 Proofs

In this appendix we give a proof of Theorem 1. The proof mostly follows the calculational style of [Dijkstra and Scholten, 1990].

Let $M^*, s^* \models \phi$ ($M^*, s^* \not\models \phi$) indicate that our model checker returns *True* (*False*) when checking a formula ϕ in state s^* of the abstract model M^*. We prove that for each CTL property p,

$$\begin{array}{ll} (1) & (M^*, s^* \models p) \Rightarrow (M, s \models p) \\ (2) & (M^*, s^* \not\models p) \Rightarrow (M, s \not\models p) \end{array}$$

The proof is by induction on the structure of p. We use (abs. algorithm) to indicate the abstract model-checking algorithm in Figure 1, and (mc. algorithm) to indicate the classical model-checking algorithm, e.g., the one that appeared in [Bultan et al., 1997]. We use the classical function pred: $2^S \rightarrow 2^S$ that takes a set of states Q and returns all states that can reach some state in Q in one transition:

$$s \in \mathrm{pred}(Q) \equiv \exists t \in Q \wedge (s, t) \in E$$

Finally, we use the following lemma, which is easily proven by definition of pred, pred_T, pred_M, E, E^*_T and E^*_F:

Lemma 1 *The following properties hold:*

$$\begin{array}{lcl} \forall s^*, t^* \in S^*,\ s^* \in pred_T(t^*) & \rightarrow & \forall s \in \gamma(s^*), t \in \gamma(t^*),\ s \in pred(t) \\ \forall s^*, t^* \in S^*,\ s^* \notin pred_M(t^*) & \rightarrow & \forall s \in \gamma(s^*), t \in \gamma(t^*),\ s \notin pred(t) \end{array}$$

Theorem 1 *Let* Abs*:* $M \rightarrow M^*$ *be a valid abstraction and* p *be an arbitrary CTL formula. Then,*

$$\begin{array}{lcl} Check(M^*, p) = \mathit{True} & \rightarrow & (M \models p) \\ Check(M^*, p) = \mathit{False} & \rightarrow & (M \not\models p) \end{array}$$

Proof:

Base: $p \in A$ (1) $M^*, s^* \models p$

$\Rightarrow$ (abs. algorithm)

$p \in L_T^*(s^*)$

$\Rightarrow$ (construction of L_T^*)

$p \in L_T(s)$

$\Rightarrow$ (mc. algorithm)

$M, s \models p$

(2) $M^*, s^* \not\models p$

$\Rightarrow$ (abs. algorithm)

$p \in L_F^*(s^*)$

$\Rightarrow$ (construction of L_F^*)

$p \in L_F(s)$

$\Rightarrow$ (mc. algorithm)

$M, s \not\models p$

IH: Assume (1) and (2) hold for formulas of length $\leq n$.

IC: Prove (1) and (2) for formulas of length $\leq n + 1$.

Proof: $p = \neg\varphi$ (1) $M^*, s^* \models \neg\varphi$

$\Rightarrow$ (abs. algorithm)

$M^*, s^* \not\models \varphi$

$\Rightarrow$ (IH)

$M, s \not\models \varphi$

$\Rightarrow$ (mc. algorithm)

$M, s \models \neg\varphi$

(2) The proof is similar to the one above.

$p = \varphi \wedge \psi$ (1) $M^*, s^* \models \varphi \wedge \psi$

$\Rightarrow$ (abs. algorithm)

$(M^*, s^* \models \varphi) \ \wedge \ (M^*, s^* \models \psi)$

$\Rightarrow$ (IH)

$(M, s \models \varphi) \ \wedge \ (M, s \models \psi)$

$\Rightarrow$ (mc. algorithm)

$M, s \models \varphi \wedge \psi$

(2) $M^*, s^* \not\models \varphi \wedge \psi$

$\Rightarrow$ (abs. algorithm)

$(M^*, s^* \not\models \varphi) \vee (M^*, s^* \not\models \psi)$

$\Rightarrow$ (IH)

$(M, s \not\models \varphi) \vee (M, s \not\models \psi)$

$\Rightarrow$ (mc. algorithm)

$M, s \not\models \varphi \wedge \psi$

$p = \varphi \vee \psi$ The proof is similar to the one for $p = \varphi \wedge \psi$.

$p = EX\varphi$ (1) $M^*, s^* \models EX\varphi$

$\Rightarrow$ (abs. algorithm)

$\exists t^*,\ \text{s.t.}\ (s^* \in \text{pred}_T(t^*)) \wedge (M^*, t^* \models \varphi)$

$\Rightarrow$ (IH), (Lemma 1)

$\exists t,\ \text{s.t.}\ (s \in \text{pred}(t)) \wedge (M, t \models \varphi)$

$\Rightarrow$ (mc. algorithm)

$M, s \models EX\varphi$

(2) $M^*, s^* \not\models EX\varphi$

$\Rightarrow$ (abs. algorithm)

$(s^* \in S^*)\ \wedge\ (s^* \notin \text{pred}_M(S^* - T^*))$,

where T^* is the set of states such that $\forall t^* \in T^*,\ M^*, t^* \not\models \varphi$

$\Rightarrow$ (Lemma 1), (IH)

$\forall t,\ (s \in \text{pred}(t) \Rightarrow M, t \not\models \varphi)$

$\Rightarrow$ (mc. algorithm)

$M, s \not\models EX\varphi$

$p = AX\varphi$ The proof is similar to the one for $p = EX\varphi$.

$p = E[\varphi U \psi]$ (1) The proof is by induction on the length of the path from s^* to a state in which φ holds.

Base: length = 0

$M^*, s^* \models E[\varphi U \psi]$

$\Rightarrow$ (length = 0), (abs. algorithm)

$M^*, s^* \models \psi$

$\Rightarrow$ (IH)

$M, s \models \psi$

$\Rightarrow$ (mc. algorithm)

$M, s \models E[\varphi U \psi]$

IH1: Let T^* be the set of states from which φ can be reached in $< n$ steps via *True* transitions.

Assume $\forall t^* \in T^*,\ (M^*, t^* \models E[\varphi U \psi] \Rightarrow M, t \models E[\varphi U \psi])$

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

Proof: $M^*, s^* \models E[\varphi U \psi]$

$\Rightarrow$ (abs. algorithm)

$(M^*, s^* \models \psi) \vee ((M^*, s^* \models \varphi) \wedge s^* \in \mathrm{pred}_T(T^*))$

$\Rightarrow$ (Base case)

$(M, s \models E[\varphi U \psi]) \vee ((M^*, s^* \models \varphi) \wedge s^* \in \mathrm{pred}_T(T^*))$

$\Rightarrow$ (IH), (IH1), (Lemma 1)

$(M, s \models E[\varphi U \psi]) \vee ((M, s \models \varphi)\ \wedge\ (s \in \mathrm{pred}(t)\ \wedge\ M, t \models E[\varphi U \psi]))$

$\Rightarrow$ (mc. algorithm)

$M, s \models E[\varphi U \psi]$

$p = E[\varphi U \psi]$ (2) The proof is given below:

$M^*, s^* \not\models E[\varphi U \psi]$ means that on every path emanating from s^* either (a) φ does not occur or (b) before the first occurrence of ψ, there is a state in which φ does not hold. The algorithm can be expanded as follows:

N_0 = No(ψ)

Case (a): N_{i+1} = $N_i \cap (S^* -$ pred$_M(S^* - N_i))$

Until $N_n = N_{n+1}$

Case (b): No($E[\varphi U \psi]$) = $N_n \cup (N_0 \cap$ No(φ))

Case (a) starts from the set of states where $\neg\psi$ is true and recursively intersects it with those states that do not have *Maybe* successors in which ψ holds. The result is the set of states that can never reach ψ, i.e.

$M, s \models AG(\neg\psi) \Rightarrow M, s \not\models E[\varphi U \psi]$.

Case (b) results in the set of states in which neither φ nor ψ hold, i.e.

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

$p = A[\varphi U \psi]$ (1) The proof is by induction on the length of the path from s^* to a state in which φ holds.

Base: length = 0

$M^*, s^* \models A[\varphi U \psi]$

$\Rightarrow$ (length = 0), (abs. algorithm)

$M^*, s^* \models \psi$

$\Rightarrow$ (IH)

$M, s \models \psi$

$\Rightarrow$ (mc. algorithm)

$M, s \models A[\varphi U \psi]$

IH1: Let T^* be the set of states from which ψ can be reached in $< n$ steps via *True* transitions.

Assume $M^*, t^* \models A[\varphi U \psi] \Rightarrow M, t \models A[\varphi U \psi],\ \forall t^* \in T^*$.

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

Proof: $M^*, s^* \models A[\varphi U \psi]$

$\Rightarrow$ (abs. algorithm)

$M^*, s^* \models \psi \vee (M^*, s^* \models \varphi \ \wedge \ s^* \in \mathrm{pred}_T(T^*) \ \wedge \ s^* \notin \mathrm{pred}_M(S^* - T^*))$

$\Rightarrow$ (Base case)

$M, s \models A[\varphi U \psi] \vee (M^*, s^* \models \varphi \ \wedge \ (\forall t^*,\ s^* \in \mathrm{pred}_T(t^*) \Rightarrow (t^* \in T^*)))$

$\Rightarrow$ (IH), (IH1), (Lemma 1)

$M, s \models A[\varphi U \psi] \vee (M, s \models \varphi \ \wedge \ (\forall t,\ s \in \mathrm{pred}(t) \Rightarrow M, t \models A[\varphi U \psi]))$

$\Rightarrow$ (mc. algorithm)

$M, s \models A[\varphi U \psi]$

$p = A[\varphi U \psi]$ (2) The proof is given below:

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

N_0 = No(ψ)

Case (a): N_{i+1} = $N_i \cap$ pred$_T(N_i)$

Until $N_n = N_{n+1}$

Case (b): No($A[\varphi U \psi]$) = $N_n \cup (N_0 \cap$ No(φ))

Case (a) starts from the set of states where $\neg\psi$ is true and recursively intersects it with those states that have *True* successors in which $\neg\psi$ holds. The result is the set of states that are at the beginning of paths which can never reach ψ, i.e.

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

Case (b) results in the set of states in which neither φ nor ψ hold, i.e.

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

□

B 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();
        }
}
```