Integrating Model-Checking and Theorem Proving for Automating the Generation of Abstractions

Shiva Nejati and Mehrdad Sabetzadeh

Department of Computer Science, University of Toronto
Toronto, ON M5S 3G4, Canada.
{shiva,mehrdad}@cs.toronto.edu

Abstract. We investigate how the integration of model-checking and theorem proving techniques can help in automating the process of constructing abstractions. We survey some of the existing approaches to automatic generation of abstract models. Each approach is exemplified and its important details are outlined. In addition, we show how the optimizations proposed in [NGC03] can be expressed in terms of satisfiability problems and incorporated into a 3-valued abstraction framework.

1 Introduction

Theorem provers have been successfully employed in proof mechanization of highly expressive logics like first order and higher order logics. Theorem provers are based on deductive methods making it possible to use them for reasoning about problems not expressible in decidable theories. However, a considerable amount of human guidance may be required before a theorem prover can carry out a verification task. This is due to the inherent limitations that, for many logics, it is impossible to have a complete automatization.

Model-checkers are, in contrast, based on model-theoretic methods. There are fully automated model-checking algorithms for decidable logic fragments like temporal logics; however, these algorithms are inadequate for reasoning about infinite systems, and worse still, they all suffer from the state explosion problem in realistically large (but finite) models.

Recently, there has been a move toward integrating model-checking and theorem proving [RSS95,SS99]. The basic idea is to call a model-checker from within a theorem prover as a decision procedure for verifying a decidable property. In [RSS95], an approach for such an integration has been proposed. There, a BDD-based model-checker for the propositional μ-calculus [Koz83] has been integrated with the PVS theorem prover [ORS92]. PVS is a verification toolkit consisting of a specification language, a powerful higher order theorem prover, and a set of accessory tools.

In this report, we investigate how the integration of model-checking and theorem proving techniques can help in automating the process of constructing finite-state (and hence model-checkable) abstractions and verifying their validity. We survey some of the existing approaches to automatic construction of abstract models. Most of the approaches surveyed do not provide worked-out examples for their proposed algorithms.

To address this problem, we exemplify each approach using our own examples. The examples also provide a context for showing the similarities and differences between the surveyed approaches. Each example is accompanied by a specification written in the input language of the PVS theorem prover [ORS92]. We chose PVS because of its built-in capabilities for model-checking and generation of abstractions.

The surveyed approaches are all based on *predicate abstraction* [GS97]. [GS97], [DDP99], and [SS99] discuss abstractions that are sound for universal properties (i.e. over-approximation abstraction); and [GHJ01] discusses abstractions that are sound for both universal and existential properties (i.e. mixed abstractions).

We show how the optimizations proposed in [NGC03] can be expressed in terms of satisfiability problems and seamlessly incorporated into the 3-valued abstraction framework discussed in [GHJ01]. Further, we apply these optimizations to our examples in order to obtain more conclusive abstract models.

We assume familiarity with the basic concepts of model-checking [CGP00], and 3-valued abstraction [DGG97].

The report is structured as follows: Section 1 reviews predicate abstraction and the PVS capabilities for model-checking and abstraction. Section 2 studies the automatic generation of mixed (i.e. 3-valued) abstract models and Section 3 demonstrates how the optimizations discussed in [NGC03] can be applied to sharpen mixed abstractions.

2 Abstraction via Theorem Proving

Abstraction is one of the most effective methods to deal with the state explosion problem. In short, abstraction refers to the process of building a smaller model A from a given model C in such a way that if a property holds in A, it also holds in C.

The process of verifying an infinite system by abstraction can be broken down to three distinct stages: First, a finite abstract model is generated either manually or automatically; second, the soundness of abstraction is verified by a theorem prover; and finally, the properties of interest are checked over the abstract model. In [RSS95], the first stage is done manually while in more recent works [GS97,GHJ01,DDP99], this stage is done automatically.

We illustrate the process of deriving an abstract model from an infinite state concrete model in PVS by an example. Figure 1 depicts a transition system modeling an infinite integer counter that initially starts with -4 and is incremented by one at each step.

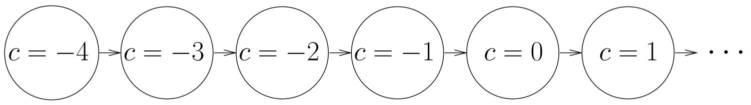

Fig. 1. Transition system for an integer counter

The PVS specification of the above transition system is given in Figure 2.

The specification begins with the declaration of a PVS theory. In the theory, a record type `state` with an integer variable `c` has been declared. The predicate `init(s)`

```
counter: THEORY
BEGIN
     state: TYPE = [# c: int #]

     s, s0, s1 : VAR state

     init(s): bool = c(s) = -4

     next(s0,s1): bool =  s1 = s0 WITH [c := c(s0) + 1]
END counter
```

Fig. 2. PVS specification of the integer counter.

(1)	$c < 0 \Rightarrow AF(c \geq 0)$	If c is negative, then on all paths c will eventually be positive
(2)	$c < 0 \Rightarrow EF(c \geq 0)$	If c is negative, then on some paths c will eventually be positive
(3)	$c \geq 0 \Rightarrow AG(c \geq 0)$	If c is positive, then on all paths c will always be positive
(4)	$c \geq 0 \Rightarrow EG(c \geq 0)$	If c is positive, then on some paths c will always be positive

Table 1. Properties of interest

declared in the theory is true if the value of c is equal to -4 in state s, that is, init(s) holds if s is the initial state. Note that in PVS, a predicate is simply a function with a boolean return value. To create a transition system, we also need to specify its *transition relation*: given a particular state, we should specify its possible successor states. In PVS, a transition relation is typically expressed as binary predicate next(s0,s1) that is true if and only if s1 is a successor of s0. In our example, s1 is the successor of s0 if and only if the value of c in s1 is one unit larger than that in s0.

The properties of interest are shown in Table 1. One may think that it would be better if we specified the above properties in LTL rather than CTL because of the fact that the model is not branching. However, as we shall see later, the abstract model is branching; therefore, it makes perfect sense to use CTL.

The PVS encoding of the above CTL properties is shown in Figure 3.

```
pos(s): bool = c(s) >= 0
neg(s): bool = c(s) < 0
prop1: theorem neg(s) implies AF(next, pos)(s)
prop2: theorem neg(s) implies EF(next, pos)(s)
prop3: theorem pos(s) implies AG(next, pos)(s)
prop4: theorem pos(s) implies EG(next, pos)(s)
```

Fig. 3. PVS encoding of the properties of interest.

In PVS, a μ-calculus property is implemented as a function that, when given a transition relation (e.g. $next(s_0, s_1)$) and an assertion over states (e.g. $pos(s)$ or $neg(s)$), computes the fixpoints. The higher-order specification language of PVS allows defining μ-calculus theories parametric in a state type and a given next-state relation over this state type. Note that all CTL operators can be defined in μ-calculus.

Obviously, the model shown in Figure 1 is not model-checkable because it is infinite. Thus, we need to construct an abstract model.

A popular abstraction technique for infinite models is *predicate abstraction* [GS97], also known as boolean abstraction. In predicate abstraction, each abstract state corresponds to an n-ary vector of truth values of predicates $\Phi = \{\varphi_1, \varphi_2, \cdots, \varphi_n\}$. Each predicate is typically a first order *quantifier-free* formula over the variables of the concrete model. By definition, a first-order formula φ is quantifier-free if every variable in φ is free, i.e. there is no quantifier in φ. In such a formula, all variables are *implicitly quantified* by a universal quantifier. The reason that we restrict the predicates to being quantifier-free is that the quantifier-free fragment of first order logic is decidable and therefore, there are fully automated decision procedures for it.

An abstraction framework is systematically defined by a pair of functions $\langle \alpha, \gamma \rangle$ that are Galois connected [CC77]. The function γ, called the *concretization function*, associates with every abstract state the set of concrete states that it represents; and the function α, called the *abstraction function*, associates with every set of concrete states a corresponding set of abstract states. In predicate abstraction, $\langle \alpha, \gamma \rangle$ and the abstract transition relation are computed as follows:

Computing γ and α: Let $\{\varphi_1, \ldots, \varphi_n\}$ be a set of abstraction predicates, and let $b_1, \ldots, b_n$ be the corresponding abstract boolean variables, where each abstract variable b_i represents all concrete states satisfying the predicate φ_i. Notice that each abstract state can be represented as a formula $f(b_1, \ldots, b_n)$. The set of concrete states represented by an abstract state $f(b_1, \ldots, b_n)$ is computed as follows:

$$\gamma(f(b_1, \ldots, b_n)) = f(\varphi_1/b_1, \ldots, \varphi_n/b_n)$$

The formula $f(\varphi_1/b_1, \ldots, \varphi_n/b_n)$ is a boolean combination of predicates $\varphi_1, \ldots, \varphi_n$ that identifies a set of concrete states.
The abstraction function α takes a predicate ψ and returns a conjunction of abstract states corresponding to ψ:

$$\alpha(\psi) = \bigwedge\{f(b_1, \ldots, b_n) \mid \psi \Rightarrow \gamma(f(b_1, \ldots, b_n))\}$$

Computing the abstract transition relation: In [GS97], the abstract transition relation is an over-approximation of the concrete transition relation. In an over-approximation abstraction, the existence of a transition between two concrete states results in a transition between their corresponding abstract states. This ensures that there is a *simulation* relation between the abstract and the concrete models [Mil71].

In our example, we choose $\varphi_1(s) = (c(s) < 0)$ and $\varphi_2(s) = (c(s) \geq 0)$ as the abstraction predicates. The resulting over-approximation abstract model is shown in Figure 4 and its PVS specification is shown in Figure 5.

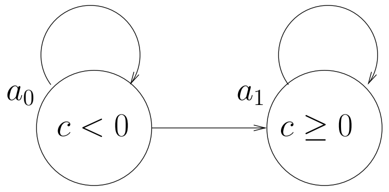

Fig. 4. Over-approximation abstraction of the integer counter.

```
a_state: TYPE = [# b1: bool,  b2: bool #]

as, as0, as1 : VAR a_state

a_init(as): bool = b1(as) AND Not b2(as)

a_may_next(as0,as1): bool =
  (
   (as1 = as0) OR
   ((b1(as0) AND Not b2(as0)) AND as1 = as0
   WITH [ b1 := false, b2 := true])
  )

apos(as): bool = Not b1(as) AND b2(as)
aneg(as): bool = b1(as) AND Not b2(as)

a_may_prop1: THEOREM aneg(as) IMPLIES AF(a_may_next, apos)(as)
a_may_prop2: THEOREM aneg(as) IMPLIES EF(a_may_next, apos)(as)
a_may_prop3: THEOREM apos(as) IMPLIES AG(a_may_next, apos)(as)
a_may_prop4: THEOREM apos(as) IMPLIES EG(a_may_next, apos)(as)
```

Fig. 5. PVS specification of the counter's over-approximation abstraction

We now explain how to verify the soundness of this abstraction using the PVS theorem prover. We define the abstraction function α. For abstraction predicates $\varphi_1(s) = c(s) < 0$ and $\varphi_2(s) = c(s) \geq 0$, the function α is as follows:

$$\alpha(\varphi_1(s)) = b_1(as) \wedge \neg b_2(as)$$
$$\alpha(\varphi_2(s)) = \neg b_1(as) \wedge b_2(as)$$

where s is a concrete state and as is an abstract state.

We prove that α preserves the initial state and the concrete transition relation. More precisely, we prove that when there is a transition between two concrete states, there should be a transition between their corresponding abstract states. These constraints on the abstraction function can be expressed by the PVS theorems shown in Figure 6.

```
%% Definition of the absraction function
abst(s): a_state =
            (
              IF (c(s) < 0) THEN (# b1 := true, b2 := false #)
              ELSE (#b1 := false, b2 := true #)
              ENDIF
            )

%% The abstraction function preserves
%% the initial state.
init_simulation: THEOREM
           init(s) IMPLIES a_init(abst(s))

%% The abstraction function preserves
%% the concrete transition relation.
may_next_simulation: THEOREM
           next(s0, s1) IMPLIES a_may_next(abst(s0), abst(s1))
```

Fig. 6. Soundness constraints on the abstraction function

It is worth pointing out that the soundness criteria (i.e. `init_simulation` and `may_next_simulation`) were proven by PVS's `(grind)` rule without need for human guidance.

The final step is model-checking the abstract model. This can be done by the PVS `(model-check)` proof rule. `(model-check)` translates a given CTL property into a propositional μ-calculus formula. This propositional μ-calculus formula will then be fed to PVS's built-in μ-calculus model-checker. At the time of this writing, PVS's μ-calculus model-checker is in early stages of development and does not return any useful feedback (e.g. counter-examples) for the properties that do not hold.

Since the abstraction is an over-approximation, it is conclusive only for the universal properties that evaluate to true. Among the properties in Table 1, only the third one is conclusive over the abstract model (see Table 2).

(1)	$AF(c \geq 0)$	Inconclusive
(2)	$EF(c \geq 0)$	Inconclusive
(3)	$c \geq 0 \Rightarrow AG(c \geq 0)$	true
(4)	$c \geq 0 \Rightarrow EG(c \geq 0)$	Inconclusive

Table 2. Results of model-checking the over-approximation abstraction.

So far, our focus has been on demonstrating how a theorem prover can be used for verifying the soundness of a manually constructed abstraction. We now turn our attention to how a theorem prover can facilitate automating the construction of abstractions.

In [SS99], a conservative abstraction scheme has been implemented as a proof rule in PVS. There, formulae over the set of concrete state variables are abstracted so as to yield formulae over the set of abstract state variables. The approach applies predicate abstraction to all assertions and the transition relation. Below, we briefly explain the approach in [SS99]; however, we will not base our work on their approach primarily because the first order formula that the approach yields for computing the under-approximation transition relation is not quantifier-free and hence, there is no guarantee that it can be automatically verified.

Assertion Abstraction

[SS99] defines both over- and under-approximation abstraction for assertions. For every assertion $p(s)$ with s being a concrete variable, the over-approximation abstraction function $\alpha_+(p(s))$ can be computed as follows:

$$\alpha_+(p(s)) = \bigwedge\{f(b_1, \ldots, b_n) \mid p(s) \Rightarrow \gamma(f(b_1, \ldots, b_n))\}$$

As an example, suppose $\varphi_1(s) = c(s) < 0$ and $\varphi_2(s) = c(s) \geq 0$ are the abstraction predicates and b_1 and b_2 are the corresponding abstract boolean variables. For an assertion $p(s) = s \leq 0$, we have: $\alpha_+(p(s)) = b_1 \vee b_2$. This is because $p(s)$ implies $\gamma(b_1 \vee b_2) = (s < 0) \vee (s \geq 0)$.

Dually, for every assertion $p(s)$, the under-approximation abstraction function $\alpha_-(p(s))$ can be computed as follows:

$$\alpha_-(p(s)) = \bigvee\{f(b_1, \ldots, b_n) \mid \gamma(f(b_1, \ldots, b_n)) \Rightarrow p(s)\}$$

Transition Relation Abstraction

In PVS, transitions are expressed as predicates over pairs of concrete states. Let $next(s_0, s_1)$ be the transition relation predicate for the concrete model. There is an over-approximation transition between two abstract states $f(b_1, \ldots, b_n)$ and $g(b_1, \ldots, b_n)$ if and only if:

$$\big(\exists s_0 \in \gamma(f(b_1, \ldots, b_n))\big) \wedge \big(\exists s_1 \in \gamma(g(b_1, \ldots, b_n))\big) \wedge next(s_0, s_1)$$

However, the above formula is not quantifier-free and hence cannot be proved automatically by `(grind)`. In order to make the over-approximation formula quantifier-free, we rewrite it as follows:

$$\forall s_0, s_1 \cdot \big(s_0 \in \gamma(f(b_1, \ldots, b_n)\big) \wedge next(s_0, s_1) \implies s_0 \in \gamma(\psi)\ (F1)$$

where ψ is the disjunction of those abstract states $g_i(b_1, \ldots, b_n)$ for which the following property holds: $\big(\exists s_0 \in \gamma(f(b_1, \ldots, b_n))\big) \wedge \big(\exists s_1 \in \gamma(g_i(b_1, \ldots, b_n))\big) \wedge next(s_0, s_1)$. Here, we do not elaborate the details of the algorithm for computing ψ. The interested reader can consult [SS99] for the details.

Now, (F1) contains only universal quantifiers and hence, can be written as a quantifier-free first-order formula.

As an example, suppose $\varphi_1(s) = c(s) < 0$ and $\varphi_2(s) = c(s) \geq 0$ are the abstraction predicates and b_1 and b_2 are the corresponding abstract boolean variables. Suppose the concrete transition is specified by the following PVS predicate:

```
next(s0,s1): bool =  s1 = s0 WITH [c := c(s0) + 1]
```

For the abstract state $b_1 \wedge \neg b_2$, the minimal disjunction making (F1) a valid formula is $\psi = (b_1 \wedge \neg b_2) \vee (\neg b_1 \wedge b_2)$. Therefore, the abstract state $b_1 \wedge \neg b_2$ has two successors, namely $b_1 \wedge \neg b_2$ and $b_1 \wedge \neg b_2$, in the over-approximation abstraction.

The PVS abstraction algorithm is implemented as a proof rule `(abstract-and-mc)`. This proof rule takes the abstraction predicates as parameter and calls the built-in model-checker to verify the temporal properties expressed by PVS theorems. The `(abstract-and-mc)` proof rule can be used as follows to automatically compute the abstraction in Figure 4 and verify the properties of interest:

```
(abstract-and-mc ("lambda(s):c(s) >= 0" "lambda(s):c(s) < 0"))
```

Since the resulting abstraction is an over-approximation, we obtain the same results as those shown in Table 2.

Unfortunately, [SS99] does not discuss under-approximation abstraction for transition relations. To address this shortcoming, we attempted to find a quantifier-free formula that describes an under-approximation transition relation, but we were not successful. This was due an structural difference between the formulae for over- and under-approximation: the formula corresponding to the over-approximation abstraction has two existential quantifiers whereas the formula corresponding to the under-approximation abstraction, has one universal and one existential quantifier.

3 Automatic Generation of Mixed (3-Valued) Abstractions

In this section, we review [GHJ01]'s approach to automating the construction of over- and under- approximation abstractions. As mentioned earlier, over-approximation abstraction is conclusive only for the universal properties that hold positively in the abstract model. In order to verify existential properties, we have to build an *under-approximation* abstraction. Under-approximation abstraction preserves existential properties that hold positively in the abstract model. Figure 7 shows the under-approximation abstraction of the integer counter.

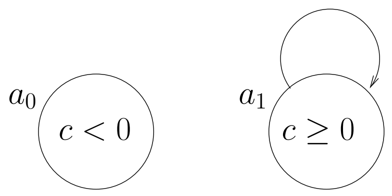

Fig. 7. The under-approximation abstraction of the integer counter.

In most cases, neither over-approximation nor under-approximation is interesting to us individually. This is because we cannot reason about properties with both universal and existential operators in either type of abstraction.

The abstract model resulting from the combination of over- and under-approximation models can be seen as a 3-valued (or a mixed) model [HJS01]: a transition between two abstract states is assigned the value true if the transition exists in both over- and under-approximations; maybe if it exists in the over-approximation but not in the under-approximation; and false otherwise. Figure 8 shows a 3-valued abstraction of the integer counter.

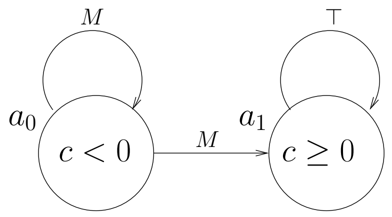

Fig. 8. 3-valued abstraction of the integer counter.

The automatic construction of abstract models as given in [GS97,DDP99] only discusses the generation of over-approximation abstraction. [GHJ01] extends this so as to take into account the generation of under-approximation abstraction and gives a framework for 3-valued abstraction.

Although [GHJ01] does not provide an implementation, its proposed algorithm is substantially similar to that given in [DDP99]. This makes it possible to use the implementation provided in [DDP99] as a basis for generating 3-valued transition relation abstractions. In addition to 3-valued transition relation abstraction, [GHJ01] discusses the generation of abstractions based on the *Cartesian abstraction* technique [BPR01]. In Cartesian abstraction, predicates can be true, false, or maybe. This is in contrast to predicate abstraction where predicates can be either true or false. Since the propositions in our examples are 2-valued, we will only talk about predicate abstraction in this report.

The key observation for describing an under-approximation transition relation by a quantifier-free formula is that a transition relation can be written as a (quantifier-free) pre-image function [GS97]: let $\{\varphi_1, \ldots, \varphi_n\}$ be a set of abstraction predicates, and let $b_1, \ldots, b_n$ be the corresponding abstract boolean variables. As before, each abstract state is represented as a formula $f(b_1, \ldots, b_n)$.

Definition 1 *For any predicate $\psi = \gamma(f(b_1, \ldots, b_n))$, `post` and `pre` functions are defined as follows:*

$$\begin{aligned}\texttt{post}(\psi) &\triangleq \{s' \in S \mid \exists s \in S \cdot R(s, s') \wedge s \models \psi\} \\ \texttt{pre}(\psi) &\triangleq \{s \in S \mid \forall s' \in S \cdot R(s, s') \Rightarrow s' \models \psi\}\end{aligned}$$

$\texttt{post}(\psi)$ is the set of successors of those states that satisfy ψ; and $\texttt{pre}(\psi)$ is the set of the states all of whose successors satisfy ψ. For a program in guarded command form, if ψ is quantifier-free then $\texttt{pre}(\psi)$ can be written in quantifier-free form, as well.

The following theorem from [GHJ01] shows how the computation of over- and under-approximation transition relations can be expressed in terms of satisfiability problems:

Theorem 1 *Let $f(b_1, \ldots, b_n)$ and $g(b_1, \ldots, b_n)$ be abstract states. And let $\psi_1 = \gamma(f(b_1, \ldots, b_n))$ and $\psi_2 = \gamma(g(b_1, \ldots, b_n))$.*

1. *There is an over-approximation transition from $f(b_1, \ldots, b_n)$ to $g(b_1, \ldots, b_n)$ iff the formula $\psi_1 \wedge \neg\texttt{pre}(\neg\psi_2)$ is **satisfiable**.*
2. *There is an under-approximation transition from $f(b_1, \ldots, b_n)$ to $g(b_1, \ldots, b_n)$ iff the formula $\psi_1 \wedge \texttt{pre}(\neg\psi_2)$ is **unsatisfiable**.*

Proof.

over-approximation transitions:

$\exists s \cdot \psi_1(s) \wedge s \in \neg\texttt{pre}(\neg\psi_2)$
$\Rightarrow$ (by Definition of pre)
$\exists s \cdot \psi_1(s) \wedge \exists s' \cdot R(s, s') \wedge s' \not\models \neg\psi_2$
$\Rightarrow$ (by Definition of negation)
$\exists s \cdot \psi_1(s) \wedge \exists s' \cdot R(s, s') \wedge \psi_2(s')$
$\Rightarrow$ (by the Assumptions that $\psi_1 = \gamma(f(b_1, \ldots, b_n))$ and $\psi_2 = \gamma(g(b_1, \ldots, b_n))$)
$\exists s, s' \cdot s \in \gamma(f(b_1, \ldots, b_n)) \wedge s' \in \gamma(g(b_1, \ldots, b_n)) \wedge R(s, s')$

under-approximation transitions:

$\forall s \cdot \psi_1(s) \Rightarrow s \notin \texttt{pre}(\neg\psi_2)$
$\Rightarrow$ (by Definition of pre)
$\forall s \cdot \psi_1(s) \Rightarrow \exists s' \cdot R(s, s') \wedge s' \not\models \neg\psi_2$
$\Rightarrow$ (by Definition of negation)
$\forall s \cdot \psi_1(s) \Rightarrow \exists s' \cdot R(s, s') \wedge \psi_2(s')$
$\Rightarrow$ (by the Assumptions that $\psi_1 = \gamma(f(b_1, \ldots, b_n))$ and $\psi_2 = \gamma(g(b_1, \ldots, b_n))$)
$\forall s \cdot s \in \gamma(f(b_1, \ldots, b_n)) \Rightarrow \exists s' \cdot s' \in \gamma(g(b_1, \ldots, b_n)) \wedge R(s, s')$

$\square$

In [DDP99], the implementation of over-approximation computation uses the CMU BDD library and SVC [DDP99] which is an efficient decision procedure for quantifier-free first order logic. Given a predicate ψ represented as a BDD, a pair of BDDs ψ_{over} and ψ_{under} is constructed. The two BDDs respectively characterize the set of over- and under-approximation successors of ψ in the abstract model. To construct ψ_{over} and ψ_{under} for a predicate ψ, there may be at most 2^n calls to the theorem prover. Intuitively, given n predicates $\varphi_1, \ldots, \varphi_n$, the number of possible boolean combinations (i.e. the number of all possible abstract states) is 2^n. To decide whether there is an over- or under-approximation transition between two abstract states, one call to the theorem prover should be made. Therefore, we may need 2^n calls to the theorem prover in order to determine the set of all successors of a single abstract state.

Based on Theorem 1, it is easy to see that the questions to be asked from the theorem prover follow certain patterns: for the over-approximation, the pattern is $\psi_1 \wedge \neg\mathtt{pre}(\neg\psi_2)$, and for the under-approximation, the pattern is $\psi_1 \wedge \neg\mathtt{pre}(\neg\psi_2)$. Obviously, these questions can be automatically produced in a straight-forward fashion using a tool. In this report, however, we will not be using any tools for producing these questions.

```
mixedcounter:THEORY
BEGIN

     state: TYPE = [# c: int #]

     s, s0, s1 : VAR state

     init(s): bool = c(s) = -4

     next(s0,s1): bool =  s1 = s0 WITH [c := c(s0) + 1]

     b1(s): bool = c(s) < 0
     b2(s): bool = c(s) >= 0

     preb1(s): bool = c(s) + 1 < 0
     preb2(s): bool = c(s) + 1 >= 0

     %% over-approximation transitions
     %% (1) a0 = b1 /\ ~b2 --> a0 = b1 /\ ~b2
     a0a0may: THEOREM Not ((b1(s) AND Not b2(s)) AND Not (preb2(s)))  %SAT
     %% (2) a0 = b1 /\ ~b2 --> a1 = ~b1 /\ b2
     a0a1may: THEOREM Not ((b1(s) AND Not b2(s)) AND Not (preb1(s)))  %SAT
     %% (3) a1 = ~b1 /\ b2 --> a1 = ~b1 /\ b2
     a1a1may: THEOREM NOT ((Not b1(s) AND b2(s)) AND Not (preb1(s)))  %SAT
     %% (4) a1 = ~b1 /\ b2 --> a0 = b1 /\ ~b2
     a1a0may: THEOREM NOT ((Not b1(s) AND b2(s)) AND Not (preb2(s)))  %VALID

     %% under-approximation transitions
     %% (5) a0 = b1 /\ ~b2 --> a0 = b1 /\ ~b2
     a0a0must: THEOREM Not ((b1(s) AND Not b2(s)) AND (preb2(s)))  %SAT
     %% (6) a0 = b1 /\ ~b2 --> a1 = ~b1 /\ b2
     a0a1must: THEOREM Not ((b1(s) AND Not b2(s)) AND (preb1(s)))  %SAT
     %% (7) a1 = ~b1 /\ b2 --> a1 = ~b1 /\ b2
     a1a1must: THEOREM Not ((Not b1(s) AND b2(s)) AND (preb1(s)))  %VALID
     %% (8) a1 = ~b1 /\ b2 --> a0 = b1 /\ ~b2
     a1a0must: THEOREM Not ((Not b1(s) AND b2(s)) AND (preb2(s)))  %SAT

END mixedcounter
```

Fig. 9. Constraints for over- and under-approximation abstractions

Figure 9 shows a PVS program that computes the over- and under-approximation abstractions of the integer counter. In the figure, the abstraction predicates are b_1 and b_2. The corresponding pre-image functions $preb_1$ and $preb_2$ respectively give the set of states all of whose successors satisfy b_1 and b_2. Since b_1 and b_2 are mutually exclusive, we have only two abstract states: $a_0 = b_1 \wedge \neg b_2$ and $a_1 = \neg b_1 \wedge b_2$. The abstract states corresponding to $b_1 \wedge b_2$ and $\neg b_1 \wedge \neg b_2$ cannot exist. The admissibility of each abstract state can be checked by a call to the theorem prover. This may require a total of 2^n calls to the theorem prover for all states.

We ask the questions `a`$_i$`a`$_j$`must` $(i, j = 0, 1)$ for the under-approximation transition relation and the questions `a`$_i$`a`$_j$`may` $(i, j = 0, 1)$ for the over-approximation transition relation. There are four questions in each group, so computing a mixed abstraction transition relation for our integer counter requires a total of eight calls to the theorem prover.

Given a predicate $p(x_1, \ldots, x_n)$, checking the unsatisfiability of $p(x_1, \ldots, x_n)$ (i.e. proving $\neg\exists x_1, \ldots, x_n \cdot p(x_1, \ldots, x_n)$) is equivalent to checking the validity of the negation of $p(x_1, \ldots, x_n)$, (i.e. proving $\forall x_1, \ldots, x_n \cdot \neg p(x_1, \ldots, x_n)$). Since the negated formula is in quantifier-free form, `(grind)` can automatically prove it if it is indeed valid. For this reason, in Figure 9, we check the negation of the questions in Theorem 1:

1. There is an over-approximation transition from ψ_1 to ψ_2 iff the formula $\neg(\psi_1 \wedge \neg\mathtt{pre}(\neg\psi_2))$ is *not valid*.
2. There is an under-approximation transition from ψ_1 to ψ_2 iff the formula $\neg(\psi_1 \wedge \mathtt{pre}(\neg\psi_2))$ is *valid*.

The PVS code for the mixed abstraction of the integer counter is shown in Figure 10. Since the properties of interest (Table 1) are negation-free, if a universal property is true (resp. false) in the over- (reps. under-) approximation abstraction, it will be true (resp. false) in the original model. Dually, if an existential property is true (reps. false) in the under- (reps. over-) approximation abstraction, it will be true (resp. false) in the original model [HJS01]. Table 3 shows the results of model-checking the mixed abstraction. As expected, the result is more conclusive compared to the case where only the over-approximation abstraction was considered.

(1)	$c < 0 \Rightarrow AF(c \geq 0)$	Inconclusive
(2)	$c < 0 \Rightarrow EF(c \geq 0)$	Inconclusive
(3)	$c \geq 0 \Rightarrow AG(c \geq 0)$	true
(4)	$c \geq 0 \Rightarrow EG(c \geq 0)$	true

Table 3. Results of model-checking the mixed abstraction.

```
a_state: TYPE = [# b1: bool,  b2: bool #]

as, as0, as1 : VAR a_state

a_init(as): bool = b1(as) AND Not b2(as)

a_may_next(as0,as1): bool =
  (
   (as1 = as0) OR
   ((b1(as0) AND Not b2(as0)) AND as1 = as0
                 WITH [ b1 := false, b2 := true])
  )

a_must_next(as0, as1): bool =
  (
   ((Not b1(as0) AND b2(as0)) AND as1 = as0
                  WITH [ b1 := false, b2 := true])
  )

apos(as): bool = Not b1(as) AND b2(as)
aneg(as): bool = b1(as) AND Not b2(as)

%over-approximation transitions
a_may_prop1: THEOREM aneg(as) IMPLIES AF(a_may_next, apos)(as) % false
a_may_prop2: THEOREM aneg(as) IMPLIES EF(a_may_next, apos)(as) % true
a_may_prop3: THEOREM apos(as) IMPLIES AG(a_may_next, apos)(as) % true
a_may_prop4: THEOREM apos(as) IMPLIES EG(a_may_next, apos)(as) % true

%under approximation transitions
a_must_prop1: THEOREM aneg(as) IMPLIES AF(a_must_next, apos)(as) % true
a_must_prop2: THEOREM aneg(as) IMPLIES EF(a_must_next, apos)(as) % false
a_must_prop3: THEOREM apos(as) IMPLIES AG(a_must_next, apos)(as) % true
a_must_prop4: THEOREM apos(as) IMPLIES EG(a_must_next, apos)(as) % true
```

Fig. 10. PVS specification for mixed abstraction of our integer counter.

4 Optimizations to Mixed Abstraction

[NGC03] proposes some optimization techniques for sharpening 3-valued abstractions by weakening the relation between concrete and abstract models. The most useful optimization proposed there is perhaps the employment of a distance-bounded reachability relation, rather than an immediate-successor relation, for computing under-approximation transition relations. Intuitively speaking, using distance-bounded reachability yields more realistic transitions in the under-approximation abstraction by reducing the gap between the over- and under-approximation abstractions. This makes the result of abstraction more conclusive.

It is well-known that reachability is not expressible in first order logic (cf. e.g. [Lib03]). We can, of course, use first order logic extended with fixpoint operators to describe reachability but this will not be of much help in the case of our integer counter because the transition system corresponding to the counter is infinite. Furthermore, if we were able to check the fixpoint operators on the concrete model, then there would be no need to generate the abstract model.

Instead of computing reachability, we compute distance-bounded reachability. That is, given a state s and an integer k, we find those states that are reachable from s by taking at most k transitions. By convention, a path of length zero is an acceptable path; therefore, every state will have a self-loop in the abstract model. Some of these self-loops may be spurious and can affect the preservation of EG and AF properties from the abstract to the concrete model [NGC03]. Later in this section, we will provide some heuristics to reduce the number of spurious paths in the abstract model.

The following theorem shows how the computation of a k-bounded under-approximation transition relation can be expressed in terms of satisfiability:

Theorem 2 *Let k be a fixed number, let $f(b_1, \ldots, b_n)$ and $g(b_1, \ldots, b_n)$ be abstract states, and let $\psi_1 = \gamma(f(b_1, \ldots, b_n))$ and $\psi_2 = \gamma(g(b_1, \ldots, b_n))$.*

– *There is an under-approximation transition from $f(b_1, \ldots, b_n)$ to $g(b_1, \ldots, b_n)$ iff the following formula is **unsatisfiable***

$$\bigwedge_{0 \le i \le k} \psi_1 \wedge \Phi_i(\psi_1, \psi_2) \qquad (F2)$$

where

$$\begin{aligned}\Phi_0(\psi_1, \psi_2) &= \neg\psi_2\\ \Phi_1(\psi_1, \psi_2) &= \mathtt{pre}(\neg\psi_2)\\ \Phi_i(\psi_1, \psi_2) &= \mathtt{pre}(\neg\psi_1 \vee \Phi_{i-1}(\psi_1, \psi_2)) \text{ for } 1 < i \le k\end{aligned}$$

Proof. We prove that the formula (F2) is unsatisfiable if and only if for every state s_0 satisfying ψ_1, there exists some path $\pi = s_0 \ldots s_i$ with length $i \le k$ such that $s_j \models \psi_1$ and $s_i \models \psi_2$ for all $1 \le j < i$. The argument can be proved by induction on the length of the path. Clearly, the formula (F2) is unsatisfiable iff every state s_0 satisfying ψ_1 fails to satisfy $\Phi_i(\psi_1, \psi_2)$ for some $0 \le i \le k$. We show that for any state s_0 that does not satisfy $\Phi_i(\psi_1, \psi_2)$, there exists a path $\pi = s_0 \ldots s_i$ such that $s_j \models \psi_1$ and $s_i \models \psi_2$ for all $1 \le j < i$.

$\forall s_0 \cdot s_0 \models \psi_1 \Rightarrow s_0 \models \neg\Phi_{i+1}(\psi_1,\psi_2)$
$\Leftrightarrow$ (by the Definition of $\Phi_{i+1}(\psi_1,\psi_2)$)
$\forall s_0 \cdot s_0 \models \psi_1 \Rightarrow \exists s_1 \cdot R(s_0,s_1) \wedge s_1 \models \psi_1 \wedge s_1 \models \neg\Phi_i(\psi_1,\psi_2)$
$\Leftrightarrow$ (by the Inductive Hypothesis)
$\forall s_0 \cdot s_0 \models \psi_1 \Rightarrow \exists s_1, s_2, \ldots, s_{i+1} \cdot R(s_{j-1},s_j) \wedge s_{j-1} \models \psi_1 \wedge s_{i+1} \models \psi_2$ for $1 \leq j \leq i+1$
$\Leftrightarrow \forall s_0 \cdot s_0 \models \psi_1 \Rightarrow$ there exists a path $\pi = s_0 \ldots s_{i+1}$ such that $s_j \models \psi_1 \wedge s_{i+1} \models \psi_2$ for $1 \leq j \leq i$

□

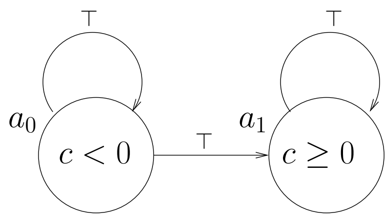

Fig. 11. The optimized abstraction of the integer counter.

Figure 11 shows the abstraction of the integer counter generated according to Theorem 2 with $k = 4$. As shown in this figure, the transition from a_0 to a_1 is an under-approximation transition because it is possible to reach to a state `s'` with `c(s') >= 0` from every state `s` with `c(s) < 0` by taking four transitions (see Figure 1). Furthermore, there is a self-loop transition for each a_0 and a_1 in the under-approximation.

Figure 12 shows the PVS code snippet for computing the under-approximation transition relation of the abstract model in Figure 11.

As in the previous section, we check the validity of the negation of the formulae in Theorem 2. Since $\mathsf{a}_0\mathsf{a}_0$`must_opt`, $\mathsf{a}_0\mathsf{a}_1$`must_opt`, and $\mathsf{a}_1\mathsf{a}_1$`must_opt` are valid, then in the under-approximation abstraction there are transitions from a_0 to a_0, a_0 to a_1 and a_1 to a_1.

The abstract models computed based on Theorem 2 are sound and complete only for existential CTL$_{-X}$ properties[1] with finite witnesses and universal CTL$_{-X}$ properties with infinite counter-examples [NGC03]. More precisely, if an EG property is true, or if an AU or AF property is false in the concrete model, no conclusions can be made as to what the value of that property is in the concrete model. However, if an EG property evaluates to false, or if an AU or AF property evaluates to true in the concrete model, the property has the same value when evaluated in the concrete model.

Table 4 shows the results of checking the properties of interest over the optimized abstraction (Figure 11). The second property, i.e. $neg \Rightarrow EFpos$, evaluates to true in the abstraction. Since the property has a finite witness, the abstraction is conclusive for this property regardless of whether it evaluates to true or false [NGC03]. However, the first property, i.e. $neg \Rightarrow AFpos$, evaluates to false in the abstraction, making it inconclusive, because AF properties have infinite counter-examples.

[1] CTL$_{-X}$ is the set of CTL properties that do not contain next operators.

```
b1(s): bool = c(s) < 0
b2(s): bool = c(s) >= 0
b_init(s): bool = c(s) >= -4
f(s): state = (# c := c(s) + 1 #)

%(1) a0 = b1 --> a0 = b1    VALID
a0a0must_opt: THEOREM b_init(s) IMPLIES
      Not (
            (b1(s) AND Not b1(s))                  AND
            (b1(s) AND b2(f(s)))                   AND
            (b1(s) AND b2(f(s)) OR b2(f(f(s)))))   AND
            (b1(s) AND b2(f(s)) OR b2(f(f(s))) OR b2(f(f(s)))))
           )
%(2) a0 = b1 --> a1 = b2    VALID
a0a1must_opt: THEOREM b_init(s) IMPLIES
     Not (
             (b1(s) AND Not b2(s))                       AND
             (b1(s) AND b1(f(s)))                        AND
             (b1(s) AND (b2(f(s)) OR b1(f(f(s)))))       AND
             (b1(s) AND (b2(f(s)) OR b2(f(f(s))) OR
                                 b1(f(f(f(s))))))  AND
             (b1(s) AND (b2(f(s)) OR b2(f(f(s))) OR
                          b2(f(f(f(s)))) OR b1(f(f(f(f(s)))))))
          )
%(3) a1 = b2  --> a1 = b2    VALID
a1a1must: THEOREM b_init(s) IMPLIES
     Not (
              (b2(s) AND Not b2(s))                     AND
              (b2(s) AND b1(f(s)))                      AND
              (b2(s) AND (b1(f(s)) OR b1(f(f(s)))))     AND
              (b2(s) AND (b1(f(s)) OR b1(f(f(s))) OR
                                 b1(f(f(f(s)))))) AND
              (b2(s) AND (b1(f(s)) OR b1(f(f(s))) OR
                          b1(f(f(f(s)))) OR b1(f(f(f(f(s)))))))
          )
%(4) a1 = b2  --> a0 = b1   SAT
a1a0must: THEOREM  b_init(s) IMPLIES
     Not (
              (b2(s) AND Not (Not b2(s) AND b1(s)) AND
              (b2(s) AND b2(f(s)))           AND
              (b2(s) AND (b1(f(s)) OR b2(f(f(s))))) AND
              (b2(s) AND (b1(f(s)) OR b1(f(f(s))) OR
                                 b2(f(f(f(s)))))) AND
              (b2(s) AND (b1(f(s)) OR b1(f(f(s))) OR
                          b1(f(f(f(s)))) OR b2(f(f(f(f(s)))))))
                       )
```

Fig. 12. Constraints on the optimized abstraction

(1)	$c < 0 \Rightarrow AF(c \geq 0)$	Inconclusive
(2)	$c < 0 \Rightarrow EF(c \geq 0)$	true
(3)	$c \geq 0 \Rightarrow AG(c \geq 0)$	true
(4)	$c \geq 0 \Rightarrow EG(c \geq 0)$	true

Table 4. Results of model-checking the optimized abstraction.

The reason why the result of abstraction is not conclusive for EG properties holding in the abstraction is that there may be infinite paths made entirely of under-approximation transitions in the abstract model. Such paths do not necessarily have corresponding infinite paths in the concrete model. In other words, weakening the condition for generating under-approximation transitions may cause spurious infinite paths. Below, we propose two ways to improve the result of abstraction by removing some of the spurious paths:

– Revising Condition 2 of Theorem 2 as follows:

  "There is an under-approximation transition from $f(b_1, \ldots, b_n)$ to $g(b_1, \ldots, b_n)$ iff there is an over-approximation transition (Theorem 1 Part (1)) from $f(b_1, \ldots, b_n)$ to $g(b_1, \ldots, b_n)$ and $\bigwedge_{0 \leq i \leq k} \psi_1 \wedge \Phi_i(\psi_1, \psi_2)$ is *unsatisfiable*."

  Notice that, by Theorem 2, any state in an optimized under-approximation abstraction has a self-loop. The revised version of Condition 2 excludes some of the spurious self-loops from the under-approximation transition relation by requiring that a transition not present in the over-approximation should not be present in the under-approximation, either. Clearly, any such self-loop is spurious.
– Adding fairness constraints:
  It is known that fairness constraints can be employed to exclude spurious behaviors. Here, we describe how appropriate fairness constraints can sharpen the results of abstraction.
  Whenever an EG property holds in the abstract model, the model-checker returns an infinite under-approximation path $\pi_\alpha = a_0 \ldots a_\ell \langle a_{\ell+1} \ldots a_n \rangle^\omega$ as a witness for the property. Each a_i ($0 \leq i \leq n$) is an abstract state and hence corresponds to a boolean function $f_i(b_1, \ldots, b_n)$. Let $\psi_i = \gamma(f_i(b_1, \ldots, b_n))$ for $0 \leq i \leq n$. The concrete path that corresponds to π_α is $\pi_c = \psi_1, \ldots, \psi_\ell, \langle \psi_{\ell+1}, \ldots, \psi_n \rangle^\omega$. Since the abstraction introduced in [NGC03] preserves the finite paths of the abstract model in the concrete one, we know that the finite path $\psi_1, \ldots, \psi_\ell, \psi_{\ell+1}, \ldots, \psi_n$ is not spurious. However, we do not know if $\psi_{\ell+1}, \ldots, \psi_n$ indeed occurs an infinite number of times in the concrete model. If $\psi_{\ell+1}, \ldots, \psi_n$ occurs only a finite number of times in the concrete model, then π_α is spurious. In this case, the fairness constraint c that excludes π_α is $c = \neg(a_{\ell+1} \wedge \ldots \wedge a_n)$.
  We cannot use first order PVS theorems for showing that $\psi_{\ell+1}, \ldots, \psi_n$ occurs a finite number of times because doing so would require infinite conjunctions and this is not expressible in first order logic. What we can do is to fix an infinity threshold w and show that $\psi_{\ell+1}, \ldots, \psi_n$ occurs no more than w times. Obviously,

this cannot distinguish between the case where $\psi_{\ell+1}, \ldots, \psi_n$ occurs an infinite number of times and the case where $\psi_{\ell+1}, \ldots, \psi_n$ occurs only a finite number of times but more than w times.

Theorem 3 *Let $\tau = \psi_{\ell+1}, \ldots, \psi_n$ be the repeating suffix of a path $\pi_c = \psi_1 \ldots \psi_l \langle \psi_{l+1} \ldots \psi_n \rangle^\omega$, and let w be a fixed number. The suffix does not occur more than w times in the concrete model iff the following formula is **unsatisfiable**:*

$$\psi_\ell \wedge \neg \Psi_w(\psi_{\ell+1}, \ldots, \psi_n)$$

where

$$\begin{aligned}\Psi_0(\psi_{\ell+1}, \ldots, \psi_n) &= \texttt{pre}(\neg\psi_{\ell+1} \vee \texttt{pre}(\neg\psi_{\ell+2} \vee \ldots \vee \texttt{pre}(\neg\psi_n) \cdots)\\ \Psi_w(\psi_{\ell+1}, \ldots, \psi_n) &= \texttt{pre}(\neg\psi_{\ell+1} \vee \texttt{pre}(\neg\psi_{\ell+2} \vee \ldots \vee \Psi_{w-1}(\psi_{\ell+1}, \ldots, \psi_n) \cdots)\end{aligned}$$

Proof. By induction on the number of iterations w, we can prove that the formula $\psi_1 \wedge \neg \Psi_w(\psi_l, \ldots, \psi_n)$ is unsatisfiable if and only if there is no path $s_0 \langle s_1 \ldots s_{n-\ell} \rangle^w$ such that $s_i \models \psi_i$ for $0 \leq i \leq n - \ell$. □

Consider the abstract model in Figure 11. Earlier we saw that the property $neg \Rightarrow AFpos$ is inconclusive in this abstract model. The abstract path that violates this property is $\langle a_0 \rangle^\omega$. Using the following PVS theorem, we check to see if the concrete transition corresponding to the self-loop at state a_0 can be repeated more than $w = 4$ times.

```
% a0 = b1 -> a0 -> a0 -> a0
fairness-check: THEOREM b_init(s) IMPLIES
                                  b2(f(s)) OR
                                  b2(f(f(s))) OR
                                  b2(f(f(f(s)))) OR
                                  b2(f(f(f(f(s)))))
% VALID
```

As before, we check the negation of the formula in Theorem 3. Since `fairness-check` is valid, the abstraction path $\langle a_0 \rangle^\omega$ is spurious. We check the property $neg \Rightarrow AFpos$ again. This time we add a fairness constraint to that property:

```
% fairness: ~(a0)
fair(as): bool = Not (b1(as) AND Not b2(as))
a_fair_prop1: THEOREM aneg(as) IMPLIES
                      fairAF(a_may_next, apos)(fair)(as) % true
```

Table 5 shows the results of checking the properties of interest. As the table shows, all the four properties are conclusive in the abstract model.

5 Conclusion

We surveyed some of the approaches to automated generation of abstract models and illustrated the important details of each approach through examples. In addition, we

(1)	$c < 0 \Rightarrow AF(c \geq 0)$	true
(2)	$c < 0 \Rightarrow EF(c \geq 0)$	true
(3)	$c \geq 0 \Rightarrow AG(c \geq 0)$	true
(4)	$c \geq 0 \Rightarrow EG(c \geq 0)$	true

Table 5. Results of model-checking the optimized abstraction with fairness.

demonstrated how the optimizations proposed in [NGC03] can be integrated with the 3-valued abstraction framework in [GHJ01] to generate more conclusive abstractions.

We gained hands-on experience with the PVS toolkit and observed the limitations of its current abstraction framework. One of the biggest limitations, in our view, is that the concrete transition relation in the PVS abstraction framework is described as a binary predicate. While computing the over-approximation abstraction transitions is straightforward using this encoding, automatic computation of the under-approximation transition relation seems to be impossible. To remedy this problem, we followed the approach taken in [GS97,GHJ01] where transition systems are described using a guarded command specification language.

References

[BPR01] T. Ball, A. Podelski, and S. Rajamani. “Boolean and Cartesian Abstraction for Model Checking C Programs”. In *Proceedings of 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’01)*, volume 2031 of *LNCS*, pages 268–283, April 2001.

[CC77] P. Cousot and R. Cousot. “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints”. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CGP00] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[DDP99] S. Das, D. Dill, and S. Park. “Experience with Predicate Abstraction”. In *Computer Aided Verification*, pages 160–171, 1999.

[DGG97] D. Dams, R. Gerth, and O. Grumberg. “Abstract Interpretation of Reactive Systems”. *ACM Transactions on Programming Languages and Systems*, 2(19):253–291, 1997.

[GHJ01] P. Godefroid, M. Huth, and R. Jagadeesan. “Abstraction-based Model Checking using Modal Transition Systems”. In K.G. Larsen and M. Nielsen, editors, *Proceedings of 12th International Conference on Concurrency Theory (CONCUR’01)*, volume 2154 of *LNCS*, pages 426–440, Aalborg, Denmark, 2001. Springer.

[GS97] S. Graf and H. Saidi. “Construction of Abstract State Graphs with PVS”. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV’97)*, volume 1254 of *LNCS*, Haifa, Israel, 1997. Springer.

[HJS01] M. Huth, R. Jagadeesan, and D. A. Schmidt. “Modal Transition Systems: A Foundation for Three-Valued Program Analysis”. In *Proceedings of 10th European Symposium on Programming (ESOP’01)*, volume 2028 of *LNCS*, pages 155–169, 2001.

[Koz83] D Kozen. “Results on the Propositional μ-calculus”. *Theoretical Computer Science*, 27:334–354, 1983.

[Lib03] L. Libkin. *Elements of Finite Model Theory*. Springer-Verlag, 2003. to appear.
[Mil71] R. Milner. ”An Algebraic Definition of Simulation between Programs”. In *Proceedings of 2nd International Joint Conference on Artificial Intelligence*, pages 481–489, 1971.
[NGC03] S. Nejati, A. Gurfinkel, and M. Chechik. “Weaker Relations, Better Abstractions”. manuscript, 2003.
[ORS92] S. Owre, J. Rushby, and N. Shankar. “PVS: A prototype verification system”. In *Proceedings of the 11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.
[RSS95] S. Rajan, N. Shankar, and M. K. Srivas. “An Integration of Model Checking with Automated Proof Checking”. In P. Wolper, editor, *Proceedings of the 7th International Conference On Computer Aided Verification*, volume 939, pages 84–97, Liege, Belgium, 1995. Springer Verlag.
[SS99] H. Saidi and N. Shankar. “Abstract and Model Check While you Prove”. In *Proc. of the 11th Int. Conf. on Computer Aided Verification*, pages 443–454, 1999.

A Appendix

A.1 Complete Code for Figure 1

```
counter:THEORY
BEGIN

     state: TYPE = [# c: int #]

     s, s0, s1 : VAR state

     init(s): bool = c(s) = -5

     next(s0,s1): bool =  s1 = s0 WITH [c := c(s0) + 1]

     pos(s): bool = c(s) >= 0
     neg(s): bool = c(s) < 0

     prop1: THEOREM neg(s) IMPLIES AF(next, pos)(s)
     prop2: THEOREM neg(s) IMPLIES EF(next, pos)(s)
     prop3: THEOREM pos(s) IMPLIES AG(next, pos)(s)
     prop4: THEOREM pos(s) IMPLIES EG(next, pos)(s)

END counter
```

A.2 Complete Code for Figures 5 and 6

```
acounter:THEORY
BEGIN

     state: TYPE = [# c: int #]

     s, s0, s1 : VAR state

     init(s): bool = c(s) = -4

     next(s0,s1): bool =  s1 = s0 WITH [c := c(s0) + 1]

     pos(s): bool = c(s) >= 0
     neg(s): bool = c(s) < 0

     prop1: THEOREM neg(s) IMPLIES AF(next, pos)(s)
     prop2: THEOREM neg(s) IMPLIES EF(next, pos)(s)
     prop3: THEOREM pos(s) IMPLIES AG(next, pos)(s)
     prop4: THEOREM pos(s) IMPLIES EG(next, pos)(s)


     a_state: TYPE = [# b1: bool,  b2: bool #]

     as, as0, as1 : VAR a_state

     a_init(as): bool = b1(as) AND Not b2(as)

     a_may_next(as0,as1): bool =
       (
        (as1 = as0) OR
        ((b1(as0) AND Not b2(as0)) AND as1 = as0 WITH [ b1 := false, b2 := true])
       )
     a_must_next(as0, as1): bool =
       (
        ((Not b1(as0) AND b2(as0)) AND as1 = as0 WITH [ b1 := false, b2 := true])
       )

     apos(as): bool = Not b1(as) AND b2(as)
     aneg(as): bool = b1(as) AND Not b2(as)


     abst(s): a_state =
            (
              IF (c(s) < 0) THEN (# b1 := true, b2 := false #)
              ELSE (#b1 := false, b2 := true #)
              ENDIF
            )

     conc(as): state =
            (
              IF(b1(as) AND NOT b2(as)) THEN (# c := -1 #)
              ELSE (# c := 0 #)
              ENDIF
            )

     init_simulation: THEOREM
           init(s) IMPLIES a_init(abst(s))

     may_next_simulation: THEOREM
           next(s0, s1) IMPLIES a_may_next(abst(s0), abst(s1))

     must_next_simulation: THEOREM
           Not next(s0, s1) IMPLIES Not a_must_next(abst(s0), abst(s1))


     %EE
     a_may_prop1: THEOREM aneg(as) IMPLIES AF(a_may_next, apos)(as) % wrong
     a_may_prop2: THEOREM aneg(as) IMPLIES EF(a_may_next, apos)(as) % correct
     a_may_prop3: THEOREM apos(as) IMPLIES AG(a_may_next, apos)(as) % correct
     a_may_prop4: THEOREM apos(as) IMPLIES EG(a_may_next, apos)(as) % correct

     %AE
     a_must_prop1: THEOREM aneg(as) IMPLIES AF(a_must_next, apos)(as) % correct
     a_must_prop2: THEOREM aneg(as) IMPLIES EF(a_must_next, apos)(as) % wrong
     a_must_prop3: THEOREM apos(as) IMPLIES AG(a_must_next, apos)(as) % correct
     a_must_prop4: THEOREM apos(as) IMPLIES EG(a_must_next, apos)(as) % correct

     % fairness: ~(a0)
     fair(as): bool = Not (b1(as) AND Not b2(as))
     a_fair_prop1: THEOREM aneg(as) IMPLIES fairAF(a_may_next, apos)(fair)(as) % correct

END acounter
```

```
abscounter:THEORY
BEGIN

     state: TYPE = [# c: int #]

     s, s0, s1 : VAR state

     init(s): bool = c(s) = -5

     next(s0,s1): bool =  s1 = s0 WITH [c := c(s0) + 1]

     pos(s): bool = c(s) >= 0
     neg(s): bool = c(s) < 0

     prop1: THEOREM neg(s) IMPLIES AF(next, pos)(s) %false
     prop2: THEOREM neg(s) IMPLIES EF(next, pos)(s) %false
     prop3: THEOREM pos(s) IMPLIES AG(next, pos)(s) %true
     prop4: THEOREM pos(s) IMPLIES EG(next, pos)(s) %false

END abscounter

%(abstract-and-mc ("lambda(s):c(s) >= 0" "lambda(s):c(s) < 0") (grind))
%(abstract ("lambda(s):c(s) >= 0" "lambda(s):c(s) < 0"))
```

A.3 Complete Code for Figure 9

```
mixedcounter:THEORY
BEGIN

     state: TYPE = [# c: int #]

     s, s0, s1 : VAR state

     init(s): bool = c(s) = -5

     next(s0,s1): bool =  s1 = s0 WITH [c := c(s0) + 1]

     pos(s): bool = c(s) >= 0
     neg(s): bool = c(s) < 0



     prop1: THEOREM neg(s) IMPLIES AF(next, pos)(s)
     prop2: THEOREM neg(s) IMPLIES EF(next, pos)(s)
     prop3: THEOREM pos(s) IMPLIES AG(next, pos)(s)
     prop4: THEOREM pos(s) IMPLIES EG(next, pos)(s)

     b1(s): bool = c(s) >= 0
     b2(s): bool = c(s) < 0

     preb1(s): bool = c(s) + 1 >= 0
     preb2(s): bool = c(s) + 1 < 0

     %a0 = ~b1 /\ b2 --> a0 = ~b1 /\ b2
     a0a0may: THEOREM ((Not b1(s) AND b2(s)) AND Not (preb1(s)))  %SAT
     %a0 = ~b1 /\ b2 --> a1 = b1 /\ ~b2
     a0a1may: THEOREM Not ((Not b1(s) AND b2(s)) AND Not (preb2(s)))  %SAT
     %a1 = b1 /\ ~b2 --> a1 = b1 /\ ~b2
     a1a1may: THEOREM NOT ((b1(s) AND Not b2(s)) AND Not (preb2(s))) %SAT
     %a1 = b1 /\ ~b2 --> a0 = ~b1 /\ b2
     a1a0may: THEOREM NOT ((b1(s) AND Not b2(s)) AND Not (preb1(s))) %VALID

     %a0 = ~b1 /\ b2 --> a0 = ~b1 /\ b2
     a0a0must: THEOREM Not ((Not b1(s) AND b2(s)) AND (preb1(s)))  %SAT
     %a0 = ~b1 /\ b2 --> a1 = b1 /\ ~b2
     a0a1must: THEOREM Not ((Not b1(s) AND b2(s)) AND (preb2(s)))  %SAT
     %a1 = b1 /\ ˜b2 --> a1 = b1 /\ ˜b2
     a1a1must: THEOREM Not ((b1(s) AND Not b2(s)) AND (preb2(s))) %VALID
     %a1 = b1 /\ ˜b2 --> a0 = ˜b1 /\ b2
     a1a0must: THEOREM Not ((b1(s) AND Not b2(s)) AND (preb1(s))) %SAT


END mixedcounter
```

A.4 Complete Code for Figure 12

```
stutcounter:THEORY
BEGIN

     state: TYPE = [# c: int #]

     s, s0, s1 : VAR state

     init(s): bool = c(s) = -4

     next(s0,s1): bool =  s1 = s0 WITH [c := c(s0) + 1]

     pos(s): bool = c(s) >= 0
     neg(s): bool = c(s) < 0



     prop1: THEOREM neg(s) IMPLIES AF(next, pos)(s)
     prop2: THEOREM neg(s) IMPLIES EF(next, pos)(s)
     prop3: THEOREM pos(s) IMPLIES AG(next, pos)(s)
     prop4: THEOREM pos(s) IMPLIES EG(next, pos)(s)

     b1(s): bool = c(s) >= 0
     b2(s): bool = c(s) < 0
     b_init(s): bool = c(s) > -5

     f(s): state = (# c := c(s) + 1 #)


     %a0 = ˜b1 /\ b2 --> a0 = ˜b1 /\ b2
     a0a0may: THEOREM ((Not b1(s) AND b2(s)) AND Not (b1(f(s))))  %SAT
     %a0 = ˜b1 /\ b2 --> a1 = b1 /\ ˜b2
     a0a1may: THEOREM Not ((Not b1(s) AND b2(s)) AND Not (b2(f(s))))  %SAT
     %a1 = b1 /\ ˜b2 --> a1 = b1 /\ ˜b2
     a1a1may: THEOREM NOT ((b1(s) AND Not b2(s)) AND Not (b2(f(s)))) %SAT
     %a1 = b1 /\ ˜b2 --> a0 = ˜b1 /\ b2
     a1a0may: THEOREM NOT ((b1(s) AND Not b2(s)) AND Not (b1(f(s)))) %VALID


     %a0 = ˜b1 /\ b2 --> a0 = ˜b1 /\ b2    VALID
     a0a0must: THEOREM b_init(s) IMPLIES
                       Not ( ((Not b1(s) AND b2(s)) AND Not (Not b1(s)
  AND b2(s))) AND % length zero
                             ((Not b1(s) AND b2(s)) AND b1(f(s))) AND
                             ((Not b1(s) AND b2(s)) AND (b1(f(s)) OR b1(f(f(s))))) AND
                             ((Not b1(s) AND b2(s)) AND (b1(f(s)) OR b1(f(f(s))) OR
                                                         b1(f(f(s)))))
                           )

     %a0 = ˜b1 /\ b2 --> a1 = b1 /\ ˜b2    VALID
     a0a1must: THEOREM b_init(s) IMPLIES
                       Not ( ((Not b1(s) AND b2(s)) AND Not (b1(s) AND  Not b2(s)) AND
                             ((Not b1(s) AND b2(s)) AND b2(f(s))) AND
                             ((Not b1(s) AND b2(s)) AND (b1(f(s)) OR b2(f(f(s))))) AND
                             ((Not b1(s) AND b2(s)) AND (b1(f(s)) OR b1(f(f(s))) OR
                                                         b2(f(f(f(s))))))) AND
                             ((Not b1(s) AND b2(s)) AND (b1(f(s)) OR b1(f(f(s))) OR
                                             b1(f(f(f(s)))) OR b2(f(f(f(f(s)))))))
                             )

     %a1 = b1 /\ ˜b2 --> a1 = b1 /\ ˜b2   VALID
     a1a1must: THEOREM b_init(s) IMPLIES
                       Not ( ((b1(s) AND Not b2(s)) AND Not (b1(s) AND Not b2(s)) AND
                             ((b1(s) AND Not b2(s)) AND b2(f(s))) AND
                             ((b1(s) AND Not b2(s)) AND (b2(f(s)) OR b2(f(f(s))))) AND
                             ((b1(s) AND Not b2(s)) AND (b2(f(s)) OR b2(f(f(s))) OR
                                                                b2(f(f(f(s))))))) AND
                             ((b1(s) AND Not b2(s)) AND (b2(f(s)) OR b2(f(f(s))) OR
                                                    b2(f(f(f(s)))) OR b2(f(f(f(f(s)))))))
                           )

     %a1 = b1 /\ ˜b2 --> a0 = ˜b1 /\ b2   SAT
     a1a0must: THEOREM  b_init(s) IMPLIES
                       Not ( ((b1(s) AND Not b2(s)) AND Not (Not b1(s) AND b2(s)) AND
                             ((b1(s) AND Not b2(s)) AND b1(f(s))) AND
                             ((b1(s) AND Not b2(s)) AND (b2(f(s)) OR b1(f(f(s))))) AND
                             ((b1(s) AND Not b2(s)) AND (b2(f(s)) OR b2(f(f(s))) OR
                                                                b1(f(f(f(s))))))) AND
                             ((b1(s) AND Not b2(s)) AND (b2(f(s)) OR b2(f(f(s))) OR
                                                    b2(f(f(f(s)))) OR b1(f(f(f(f(s)))))))
                           )

     % a0 -> a0
     fairnesscheck: THEOREM b_init(s) IMPLIES  b1(f(s)) OR b1(f(f(s))) OR b1(f(f(f(s)))) OR
                                               b1(f(f(f(f(s)))))

     faircheck: THEOREM b2(s) IMPLIES Not b1(f(s))

END stutcounter
```