Relative Completeness of Abstraction Refinement for Software Model Checking

Thomas Ball[1], Andreas Podelski[2], and Sriram K. Rajamani[1]

[1] Microsoft Research
[2] Max-Planck-Institut für Informatik

Abstract. *Automated methods for an undecidable class of verification problems cannot be complete (terminate for every correct program). We therefore consider a new kind of quality measure for such methods, which is completeness relative to a (powerful but unrealistic) oracle-based method. More precisely, we ask whether an often implemented method known as "software model checking with abstraction refinement" is complete relative to fixpoint iteration with "oracle-guided" widening. We show that whenever* backward *fixpoint iteration with oracle-guided widening succeeds in proving a property φ (for some sequence of widenings determined by the oracle) then software model checking with a particular form of* backward *refinement will succeed in proving φ. Intuitively, this means that the use of fixpoint iteration over abstractions and a particular backwards refinement of the abstractions has the effect of exploring the entire state space of all possible sequences of widenings.*

1 Introduction

Automatic abstraction is a fundamental problem in model checking software. A promising approach to construct abstractions automatically, called *predicate abstraction*, is to map the concrete states of a system to abstract states according to their evaluation under a finite set of predicates. Many efforts have been made to construct predicate abstractions of systems [1, 2, 6, 8, 13, 15, 16, 24–28]. Where do the predicates for predicate abstraction come from? A popular scheme for generating predicates is to guess a set of initial predicates, and use (spurious) counterexamples from a model checker to generate more predicates as necessary [3, 6, 22, 24]. Such schemes go by the name of *abstraction refinement*.

Property checking for software is undecidable, even for properties such as invariants (assertion violations). Thus it is impossible to come up with an abstraction refinement procedure that always generates a set of predicates that is guaranteed to (in)validate a program against a property. As a result, the process of abstraction refinement is largely a black-art, and little attention has been paid to even understand what the goal of predicate generation should be. This paper makes two contributions in this regard:

– We formalize a goodness criterion for abstraction refinement, namely relative completeness with respect to a comparable "oracle-guided" widening

method. Since the termination argument of most fixpoint analyses that operate on infinite state spaces and lose precision can be explained using widening, this criterion is appropriate. Without such a criterion, any abstraction refinement procedure would seem like "just another" simple and practical heuristic.
– We give an abstraction refinement procedure which satisfies the above criterion, using the pre operator. Our definition of abstraction refinement captures the essence of the many implementation strategies based on counterexamples but avoids their technicalities.

If a set of states is represented by a formula φ (in disjunctive-normal form) then a *widening* of φ is obtained by dropping some conjuncts from some disjuncts in φ. Widenings are used to accelerate the termination of fixpoint analyses [9, 11]. For example, suppose $x \geq 0 \wedge x \leq n$ represents the set of states before an increment of variable x in a loop. The formula $x \geq 0$ obtained by a widening (dropping the conjunct $x \leq n$) represents the limit of an iterative reachability analysis. The precision of the analysis then depends on the *widening schedule*: which conjuncts are dropped and when in the fixpoint analysis they are dropped. *Oracle-guided* widening uses an oracle to guess the best possible widening schedule.

We use such an oracle-guided widening as a quality measurement for reasoning about the "relative completeness" of abstraction refinement. We design an abstraction refinement procedure using the pre operator, and show that if the oracle-guided widening terminates with success then the abstraction refinement (which does not use an oracle) will terminate with success. The basic idea of the procedure is to iteratively apply pre "syntactically" without performing a satisfiability check on the formula constructed at intermediate stages. The resulting procedure has the ability to "skip" over (potentially non-terminating) loops.

The term "relative completeness" of program verification methods has previously been used to refer to the existence of an oracle in the form of a theorem prover for an undecidable logic, e.g. integer arithmetic (the method succeeds whenever the oracle does) [7]. In contrast, our use of "relative completeness" refers to the existence of an oracle guiding the widening in another verification method (that method serves as a point of reference). Furthermore, our results hold for incomplete theorem provers—we do not assume that theorem provers are complete. Instead, we give the minimal requirements on a theorem prover (such as the provability of certain implications) in order to construct sound approximations.

Our formal setting accounts for the situation where a finite-state model checker is used. There, a Boolean variable is introduced for each predicate. The model checker no longer keeps track of the logical meaning of the predicate that a Boolean variable stands for. As a consequence, the fixpoint termination test becomes strictly weaker.

This paper is organized as follows. Section 2 provides the abstract formal setting for our work. Section 3 defines Method I, an algorithm for abstract fixpoint analysis and our abstract refinement procedure. Section 4 defines Method II, an algorithm for concrete fixpoint analysis with oracle-guided widening. Section 5 shows that a particular version of Method I (based on "backward refinement") is relatively complete with respect to Method II. Section 6 illustrates the difference between forward and backward refinement with a small example. Section 7 discusses some other technical issues and Section 8 concludes the paper.

2 The Formal Setting

In this section, everything but the "syntactic" definition of the operator pre and an "implementation-biased" (computable) definition of implication is standard.

Programs. We express programs in the standard format of 'guarded' *commands* to which other programming languages (also concurrent ones) can be easily translated. A program is a set $\mathcal{C}$ of guarded commands, which are logical formulas c of the form

$$c \;\equiv\; g(X) \wedge x_1' = e_1(X) \wedge \ldots \wedge x_m' = e_m(X) \tag{1}$$

where $x_1, x_2, \ldots, x_m$ are all the program variables (including one or several program counters, here pc); the variable x_i' stands for the value of x after executing the guarded command c. We write X for the tuple of program variables, i.e. $X = \langle x_1, x_2, \ldots, x_m \rangle$. The formula g is written $g(X)$ in order to stress that its only free variables are among $x_1, \ldots, x_m$; it is called the *guard* of c. A program state is a valuation of X. We have a transition of one state into another one if the corresponding valuation of primed and unprimed variables satisfies one of the guarded commands $c \in \mathcal{C}$. While each guarded command is deterministic, we note that program itself can be nondeterministic since multiple guards can hold at a given program state.

Symbolic representation. A 'symbolic' method uses formulas φ (also referred to as constraints or Boolean expressions) of a fixed formalism to effectively represent infinite sets of states. The exact nature of the formalism does not matter here, although we have in mind that it is some restricted class of first-order formulas over the algebraic structure on which the program computes (e.g. linear arithmetic). Reflecting existing implementations (see e.g. [17, 11, 21, 19, 14]), we assume a fixed *infinite* set of atomic formulas and represent an infinite set of states by a formula of the form

$$\varphi \;\equiv\; \bigvee_{i \in I} \bigwedge_{j \in J_i} \varphi_{ij} \tag{2}$$

where the φ_{ij}'s are atomic formulas. We define a partial order on formulas $\varphi' \leq \varphi$ as the provability of the implication $\varphi' \Rightarrow \varphi$ by a given theorem prover. Note that this ordering need not correspond to the entailment ordering; in many cases (e.g. integer arithmetic), the validity of implication is undecidable.

We purposefully do not require that theorem provers implement the test of the (in general, undecidable) validity of implication. As we will see, in order for our results to hold, a theorem prover only must be able prove that $\varphi \wedge \varphi' \Rightarrow \varphi$, as well as that $\varphi \Rightarrow \varphi \vee \varphi'$, for all formulas φ and φ'.

Pre and Post. For a guarded command c of the form (1), we define the application of the operator pre_c on a formula φ by the simultaneous substitution of the variables $x_1, x_2, \ldots, x_k$ in φ by the expressions $e_1, \ldots, e_k$. The operator pre for a program (a set of guarded commands) is simply the disjunction of the pre_c.

$$\mathsf{pre}_c(\varphi) \equiv g(X) \wedge \varphi[e_1(X), \ldots, e_m(X)/x_1, \ldots, x_m]$$
$$\mathsf{pre}(\varphi) \equiv \bigvee_{c \in \mathcal{C}} \mathsf{pre}_c(\varphi)$$

In deviation from more familiar definitions, we do not perform a satisfiability check in the computation of pre_c. This is crucial in the definition of the backward refinement procedure in Section 3, but not for the fixpoint procedure in Section 4. In our formulation, we use a theorem prover only to check the ordering $\varphi \leq \varphi'$; we thus do not model the standard optimization of eliminating unsatisfiable disjuncts in a formula φ.

The application of the operator post_c on a formula φ is defined as usual; its computation requires a quantifier elimination procedure.

$$\mathsf{post}_c(\varphi) \equiv (\exists X.\, \varphi \wedge g(X) \wedge x_1' = e_1(X) \wedge \ldots \wedge x_m' = e_m(X))[X/X']$$
$$\mathsf{post}(\varphi) \equiv \bigvee_{c \in \mathcal{C}} \mathsf{post}_c(\varphi)$$

Invariants. In order to specify correctness, we fix formulas init and safe denoting the set of *initial* and *safe* states, respectively, as well as formulas $\mathsf{nonInit}$ and unsafe denoting their complements. These formulas are in the form given for φ in (2). We define the given program to be *correct* if no unsafe state is reachable from an initial state.

The correctness can be proven by showing one of the two conditions below. Here, $\mathsf{lfp}(\mathsf{F}, \varphi)$ stands for the least fixpoint of the operator F above φ.

$$\mathsf{lfp}(\mathsf{post}, \mathsf{init}) \leq \mathsf{safe}$$
$$\mathsf{lfp}(\mathsf{pre}, \mathsf{unsafe}) \leq \mathsf{nonInit}$$

The least fixpoint implicitly refers to the quotient lattice of formulas wrt. the pre-order "$\leq$".[1]

A *safe invariant* is an *inductive* invariant that implies safe, i.e. a formula ψ such that

[1] The quotient lattice is defined by the partial-order that results from collapsing strongly-connected components in the pre-order $\leq$. This identifies equivalance classes of formulas that are logically equivalent (as defined by $\leq$). It is not necessary to introduce extra notation for the quotient lattice since e.g. $\varphi \leq \varphi'$ is equivalent to the fact that the equivalence class of φ is smaller than or equal to the one of φ' in the quotient lattice. We leave this implicit in order to keep the notation concise. Additionally, the quotient lattice has a top element true which is greater than every element in the lattice and a bottom element false that is less than every element in the lattice.

– $\mathsf{init} \le \psi$,
– $\mathsf{post}(\psi) \le \psi$,
– $\psi \le \mathsf{safe}$.

We will call a safe invariant a *forward invariant* in order to distinguish it from what we call a *backward invariant*, namely a formula ψ such that

– $\mathsf{unsafe} \le \psi$,
– $\mathsf{pre}(\psi) \le \psi$,
– $\psi \le \mathsf{nonInit}$.

We can establish correctness by computing either a forward invariant or a backward invariant. In order to have a generic notation that allows us to cover both cases, we introduce meta symbols F, start and bound such that $\langle \mathsf{F}, \mathsf{start}, \mathsf{bound} \rangle$ will be instantiated to $\langle \mathsf{post}, \mathsf{init}, \mathsf{safe} \rangle$ and to $\langle \mathsf{pre}, \mathsf{unsafe}, \mathsf{nonInit} \rangle$; an $\langle \mathsf{F}, \mathsf{start}, \mathsf{bound} \rangle$-invariant is then either a forward invariant or a backward invariant. Therefore we can express either of the two correctness conditions above as the existence of an $\langle \mathsf{F}, \mathsf{start}, \mathsf{bound} \rangle$-invariant, which is a formula ψ such that

– $\mathsf{start} \le \psi$,
– $\mathsf{F}(\psi) \le \psi$,
– $\psi \le \mathsf{bound}$.

The domain of formulas is closed under the application of F; the domain need not, however, contain $\mathsf{lfp}(\mathsf{F}, \mathsf{start})$. Even if it does not, it may still contain a formula denoting an $\langle \mathsf{F}, \mathsf{start}, \mathsf{bound} \rangle$-invariant. We note that we do not need the completeness of the domain for our results since we only consider fixpoints obtained by finite iteration sequences.

Using the generic notation, a possible approach to establish correctness is to find an upper abstraction F' of the operator F (i.e. where $\mathsf{F}(\varphi) \le \mathsf{F}'(\varphi)$ holds for all formulas φ) such that $\mathsf{lfp}(\mathsf{F}', \mathsf{start})$, the least fixpoint of F' above start, can be computed and is contained in bound. Then, $\mathsf{lfp}(\mathsf{F}', \mathsf{start})$ is an $\langle \mathsf{F}, \mathsf{start}, \mathsf{bound} \rangle$-invariant because of the simple fact that $\mathsf{F}'(\varphi) \le \varphi$ entails $\mathsf{F}(\varphi) \le \varphi$.

In the following two sections, we will use two methods that use predicate abstraction and widening, respectively, to find such an upper abstraction F'. The two possible instantiations of $\langle \mathsf{F}, \mathsf{start}, \mathsf{bound} \rangle$ to $\langle \mathsf{post}, \mathsf{init}, \mathsf{safe} \rangle$ and to $\langle \mathsf{pre}, \mathsf{unsafe}, \mathsf{nonInit} \rangle$ yield the two basic variations of each of the two methods.

3 Method I: Predicate Abstraction with Refinement

We first describe the abstract fixpoint iteration method parameterized by a *refinement procedure* that generates a (generally infinite) sequence of finite sets $\mathcal{P}_n$ of predicates over states (for $n = 0, 1, \ldots$). We then instantiate it with a particular refinement procedure (introduced below). We identify a predicate with the atomic formula φ defining it. Thus, each set $\mathcal{P}_n$ is a *finite* subset of the infinite set of atomic formulas.

```
φ₀ := start
n := 0
loop
  P_n := atoms(φ_n)
  construct abstract operator F_n^# defined by P_n
  ψ := lfp(F_n^#, start)
  if (ψ ≤ bound) then
     STOP with "Success"
  φ_{n+1} := φ_n ∨ F(φ_n)
  n := n+1
endloop
```

Fig. 1. Method I: abstract fixpoint iteration with iterative abstraction refinement, where $\langle \mathsf{F}, \mathsf{start}, \mathsf{bound}\rangle$ is either $\langle \mathsf{post}, \mathsf{init}, \mathsf{safe}\rangle$ ("forward") or $\langle \mathsf{pre}, \mathsf{unsafe}, \mathsf{nonInit}\rangle$ ("backward").

We write $\mathcal{L}(\mathcal{P}_n)$ for the (finite!) free distributive lattice generated by the set of predicates $\mathcal{P}_n$, with bottom element false and top element true and the operators $\wedge$ and $\vee$. The notation $\mathcal{L}(\mathcal{P}_n)^{\sqsubseteq}$ is used to stress the partial order "$\sqsubseteq$" that comes with the lattice. We note that a constant-time fixpoint check in the free lattice can be implemented using Binary Decision Diagrams (BDD's) [4]. Each lattice element can be written in its disjunctive normal form (sometimes viewed as a set of bitvectors). In the partial order "$\sqsubseteq$" of the free lattice, predicates are pairwise incomparable. Therefore, elements written in disjunctive normal form are compared as follows.

$$\bigvee_{i\in I}\bigwedge_{j\in J_i}\varphi_{ij} \sqsubseteq \bigvee_{k\in K}\bigwedge_{j\in J'_k}\varphi'_{ij} \;\text{ if }\; \forall i\in I\;\exists k\in K\;\{\varphi_{ij}\mid j\in J_i\}\supseteq\{\varphi'_{kj}\mid j\in J'_k\}$$

We will always have that $\mathcal{L}(\mathcal{P}_n)$ contains start, but generally $\mathcal{L}(\mathcal{P}_n)$ is not closed with respect to the operator F (we recall that the triple of meta symbols $\langle \mathsf{F}, \mathsf{start}, \mathsf{bound}\rangle$ stands for either $\langle \mathsf{post}, \mathsf{init}, \mathsf{safe}\rangle$ or $\langle \mathsf{pre}, \mathsf{unsafe}, \mathsf{nonInit}\rangle$).

We use the framework of abstract interpretation [9] to construct the 'best' abstraction $\mathsf{F}_n^{\#}$ of F with respect to $\mathcal{P}_n$. This operator is defined in terms of a Galois connection,

$$\mathsf{F}_n^{\#} \equiv \alpha_n \circ \mathsf{F} \circ \gamma$$

where the composition $f \circ g$ of two functions f and g is defined from right to left: $f \circ g(x) = f(g(x))$. The abstraction function α_n maps a formula φ to the smallest (wrt. "$\sqsubseteq$") formula φ' in $\mathcal{L}(\mathcal{P}_n)$ that is larger (wrt. "$\leq$") than φ, formally

$$\alpha_n(\varphi) \equiv \mu\varphi' \in \mathcal{L}(\mathcal{P}_n)^{\sqsubseteq}.\ \varphi \leq \varphi'.$$

The meaning function γ is the identity. As before, we omit the extension of the definitions to the quotient lattice.[2]

The requirement that the mappings α_n and γ form a Galois connection (which guarantees the soundness of the approximation and hence the correctness of Method I) translates to the minimal requirement for the theorem prover: it must be able to prove the validity of the implications $\varphi \Rightarrow \varphi \vee \varphi'$ and $\varphi \wedge \varphi' \Rightarrow \varphi$ for all formulas φ and φ'. This is because the requirement of the Galois connection entails that γ is monotonic (i.e. $\varphi \sqsubseteq \varphi'$ entails $\gamma(\varphi) \leq \gamma(\varphi')$). In the free lattice, we also have that $\varphi \wedge \varphi' \sqsubseteq \varphi$ and $\varphi \sqsubseteq \varphi \vee \varphi'$. Hence, by the monotonicity of γ, $\varphi \wedge \varphi' \leq \varphi$ and $\varphi \leq \varphi \vee \varphi'$, which translates to the requirement on the theorem prover.

We will have that $\mathcal{P}_0 \subset \mathcal{P}_1 \subset \dots$ and hence $\mathcal{L}(\mathcal{P}_0) \subset \mathcal{L}(\mathcal{P}_1) \subset \dots$ which means an increasing precision of the abstraction α_n for increasing n.

Method I. The parametrized method starts with $n = 0$ and repeatedly

- constructs the abstract operator $\mathsf{F}_n^\#$ defined by $\mathcal{P}_n$,
- iterates $\mathsf{F}_n^\#$ to compute $\mathsf{lfp}(\mathsf{F}_n^\#, \mathsf{start})$,
- refines the set of predicates to get predicates $\mathcal{P}_{n+1}$,
- increases n by one

until $\mathsf{lfp}(\mathsf{F}_n^\#, \mathsf{start}) \leq \mathsf{bound}$.

If Method I terminates for some n, then $\mathsf{lfp}(\mathsf{F}_n^\#, \mathsf{start})$ is a (forward or backward) invariant (depending on whether F is instantiated by post or by pre). We note that $\mathsf{lfp}(\mathsf{F}_n^\#, \mathsf{start})$ is computed over a free lattice ordered by $\sqsubseteq$, and that its computation is guaranteed to terminate.

If we take the method with the forward or backward refinement procedure defined below, we obtain the automated verification method given in Figure 1. The algorithm uses the operator atoms to map a formula φ (in disjunctive-normal form) to its (finite) set of atomic constituent formulas:

$$\mathsf{atoms}(\bigvee_{i \in I} \bigwedge_{j \in J_i} \varphi_{ij}) = \{\varphi_{ij} \mid i \in I, j \in J_i\}.$$

Refinement. Our refinement procedure is to simply apply F to the current formula φ_n and disjoin the result with φ_n, to result in φ_{n+1}. The sequence of formulas produced by the algorithm is thus:

- $\varphi_0 = \mathsf{atoms}(\mathsf{start})$
- $\varphi_{n+1} = \varphi_n \vee \mathsf{F}(\varphi_n)$

We call the procedure 'backward refinement' if $\langle \mathsf{F}, \mathsf{start} \rangle$ is $\langle \mathsf{pre}, \mathsf{unsafe} \rangle$ and 'forward refinement' if $\langle \mathsf{F}, \mathsf{start} \rangle$ is $\langle \mathsf{post}, \mathsf{init} \rangle$.

[2] To be precise, the abstraction function on the quotient lattice for the pre-order "$\leq$" maps the equivalence class of φ to $\alpha_n(\varphi)$. Similarly, the meaning of an element of the free lattice is an element of the quotient lattice; i.e. the meaning function maps φ to the equivalence class of φ.

$\varphi_0',\ old,\ n := \mathsf{start},\ \mathsf{false},\ 0$
loop
  if $(\varphi_n' \leq old)$ **then**
     if $(\varphi_n' \leq \mathsf{bound})$ **then**
        STOP with "Success"
     else
        STOP with "Don't know"
  else
  $old := \varphi_n'$
  $i :=$ guess provided by oracle
  $\varphi_{n+1}' := \mathsf{widen}(i, (\varphi_n' \vee \mathsf{F}(\varphi_{n+1}')))$
  $n := n + 1$
endloop

Fig. 2. Method II: fixpoint iteration with abstraction by oracle-guided widening. Here, $\langle \mathsf{F}, \mathsf{start}, \mathsf{bound} \rangle$ is either $\langle \mathsf{post}, \mathsf{init}, \mathsf{safe} \rangle$ ("forward") or $\langle \mathsf{pre}, \mathsf{unsafe}, \mathsf{nonInit} \rangle$ ("backward").

4 Method II: Oracle-Guided Widening

Method II iteratively applies the 'concrete' operator F over formulas and afterwards calls an oracle which determines a widening operator and applies the widening operator to the result of the application of F (the chosen widening operator may be the identity function). The precise definition of the method is given in Figure 2. Again, the instantiations of $\langle \mathsf{F}, \mathsf{start}, \mathsf{bound} \rangle$ to $\langle \mathsf{post}, \mathsf{init}, \mathsf{safe} \rangle$ and to $\langle \mathsf{pre}, \mathsf{unsafe}, \mathsf{nonInit} \rangle$ yield the forward (resp. backward) variations of the method.

The only requirement that we impose on each operator widen chosen by the oracle is that the application of widen on a formula φ yields a weaker formula φ' (denoting a larger set of states) in which some conjuncts in some disjuncts have been dropped (possibly none), i.e.

$$\mathsf{widen}(\textstyle\bigvee_{i \in I} \bigwedge_{j \in J_i} \varphi_{ij}) = \bigvee_{i \in I} \bigwedge_{j \in J_i'} \varphi_{ij} \quad \text{where} \quad J_i' \subseteq J_i \quad \text{for all } i. \tag{3}$$

We suppose that we have an enumeration of widening operators $\mathsf{widen}(0)$, $\mathsf{widen}(1)$, ... and that the oracle determines a particular one, $\mathsf{widen}(i)$, by returning a natural number i at each iteration step. We write $\mathsf{widen}(i, x)$ short for $\mathsf{widen}(x)$ where $\mathsf{widen} = \mathsf{widen}(i)$. Thus, each sequence of natural numbers produced by the oracle uniquely determines a fixpoint iteration sequence.

5 Relative Completeness for Backward Refinement

For the following theorem, we consider Method I and Method II with $\mathsf{F}, \mathsf{start}$ and bound instantiated to $\mathsf{pre}, \mathsf{unsafe}$ and $\mathsf{nonInit}$, respectively. The theorem says that for every program, Method I is guaranteed to terminate with success (i.e. proving the correctness of the program) if there exists an oracle such that Method II terminates with success.

Theorem 1 (Relative Completeness of Abstract Backward Iteration with Backward Refinement). *Method I with* $\langle \mathsf{F}, \mathsf{start}, \mathsf{bound}\rangle$ *instantiated to* $\langle \mathsf{pre}, \mathsf{unsafe}, \mathsf{nonInit}\rangle$ *will terminate with success if Method II with* $\langle \mathsf{F}, \mathsf{start}, \mathsf{bound}\rangle$ *instantiated to* $\langle \mathsf{pre}, \mathsf{unsafe}, \mathsf{nonInit}\rangle$ *terminates with success.*

The theorem means that the (possibly infinite) sequence of finite abstract fixpoint iteration sequences

$$(\mathsf{start}, \mathsf{pre}_n^{\#}(\mathsf{start}), \dots, \mathsf{lfp}(\mathsf{pre}_n^{\#}, \mathsf{start}))_{n=1,2,\dots}$$

'simulates' the tree consisting of all the infinitely many, possibly infinite branches

$$(\mathsf{start}, \mathsf{widen}(i_1) \circ \mathsf{pre}(\mathsf{start}), \dots)_{(i_1, i_2, \dots) \in \mathbb{N}^{\mathbb{N}}}$$

that arise from the different choices for the operator $\mathsf{widen}(i_k)$ at each level k (corresponding to the different sequences $(i_1, i_2, \dots)$ of natural numbers that can be returned by the oracle). 'Simulates' here informally refers to the search of a backward invariant.

Proofs. The following lemma is of intrinsic interest; it relates the expressiveness of a set of predicates $\mathcal{P}$ with the precision of the abstract fixpoint operator $\mathsf{F}^{\#}$ induced by $\mathcal{P}$ as described in Section 3.[3]

Lemma 1. *If the set of predicates* $\mathcal{P}$ *can express an* $\langle \mathsf{F}, \mathsf{start}, \mathsf{bound}\rangle$*-invariant* ψ *(i.e,* $\mathcal{L}(\mathcal{P})$*, the free lattice generated by* $\mathcal{P}$*, contains a formula* ψ *such that* $\mathsf{start} \leq \psi$*,* $\mathsf{F}(\psi) \leq \psi$ *and* $\psi \leq \mathsf{bound}$*), then the least fixpoint of* $\mathsf{F}^{\#}$*, the best abstraction of* F *over* $\mathcal{L}(\mathcal{P})$*, is an* $\langle \mathsf{F}, \mathsf{start}, \mathsf{bound}\rangle$*-invariant as well.*

Proof. The operator $\mathsf{F}^{\#}$ is defined by (see Section 3) $\mathsf{F}^{\#} = \alpha_{\mathcal{P}} \circ \mathsf{F} \circ \gamma$ where $\alpha_{\mathcal{P}}(\varphi) = \mu\varphi' \in \mathcal{L}(\mathcal{P}).\ \varphi \leq \varphi'$. We will show that for all k,

$$\mathsf{F}^{\#^k}(\mathsf{start}) \leq \psi \tag{4}$$

from which $\mathsf{lfp}(\mathsf{F}^{\#}, \mathsf{start}) \leq \psi$ follows directly. We will show (4) by induction. The case $k = 0$ follows by the assumption that ψ is an invariant. For $k+1$, we have

$$\mathsf{F}^{\#}(\mathsf{F}^{\#^k}(\mathsf{start})) \leq \mathsf{F}^{\#}(\psi)$$

[3] Successive abstraction refinement and iteration of $\mathsf{F}^{\#}$ will succeed if (and only if) the abstraction refinement procedure is 'good enough'. As a consequence of Lemma 1, the procedure is 'good enough' if it eventually generates a set of predicates $\mathcal{P}$ that is 'expressive enough' (in the precise sense of Lemma 1). We cannot expect a realistic abstraction refinement procedure that generates such a set $\mathcal{P}$ whenever it exists. We can, however, try to find procedures with 'relative' power.

by the induction hypothesis and the monotonicity of $\mathsf{F}^{\#}$ (we also use the monotonicity of γ to go from $\ldots \sqsubseteq \ldots$ to $\ldots \leq \ldots$). We now need to show that $\mathsf{F}^{\#}(\psi) \leq \psi$. We know the following:
(1) by the definition of $\alpha_{\mathcal{P}}$, $\mathsf{F}^{\#}(\psi)$ is the least element in $\mathcal{L}(\mathcal{P})$ that is greater than or equal to $\mathsf{F}(\psi)$;
(2) by the assumption that ψ is an invariant, $\mathsf{F}(\psi) \leq \psi$;
(3) ψ is an element of $\mathcal{L}(\mathcal{P})$.
Therefore, we have $\mathsf{F}^{\#}(\psi) \leq \psi$, which completes the proof by induction. $\square$

Proof (of Theorem 1). We first observe that the operator pre_c for the program consisting of only the guarded command c distributes over conjunction and disjunction.

$$\mathsf{pre}_c(\bigvee_{i\in I}\bigwedge_{j\in J_i} \varphi_{ij}) = \bigvee_{i\in I}\bigwedge_{j\in J_i} \mathsf{pre}_c(\varphi_{ij}) \tag{5}$$

It follows from this observation and the definition of $\mathsf{pre}(\varphi)$ as $\bigvee_{c\in C} \mathsf{pre}_c(\varphi)$ that

$$\mathsf{atoms}(\mathsf{pre}(\bigvee_{i\in I}\bigwedge_{j\in J_i} \varphi_{ij})) = \bigcup_{c\in C}\{\mathsf{atoms}(\mathsf{pre}_c(\varphi_{ij})) | i \in I, j \in J_i\} \tag{6}$$

Using the above observations we prove that $\mathsf{atoms}(\varphi_n) \supseteq \mathsf{atoms}(\varphi'_n)$, where φ_n is the formula in Method I at (the beginning of) iteration n and φ'_n is the formula in Method II at (the beginning of) iteration n. The proof goes by induction over n. The base case is simple as $\varphi_0 = \mathsf{start}$ and $\varphi'_0 = \mathsf{start}$. The values φ_{n+1} and φ'_{n+1} are constructed as follows by Methods I and II (respectively):

- $\varphi_{n+1} = \varphi_n \vee \mathsf{pre}(\varphi_n)$
- $\varphi'_{n+1} = \mathsf{widen}(i, \varphi'_n \vee \mathsf{pre}(\varphi'_n))$

By the induction hypothesis, $\mathsf{atoms}(\varphi_n) \supseteq \mathsf{atoms}(\varphi'_n)$. It follows directly from (6) that $\mathsf{atoms}(\mathsf{pre}(\varphi_n)) \supseteq \mathsf{atoms}(\mathsf{pre}(\varphi'_n))$. Since the widen operator can only drop atomic formulae, we have $\mathsf{atoms}(\varphi_{n+1}) \supseteq \mathsf{atoms}(\varphi'_{n+1})$, which completes the proof by induction.

Therefore, if Method II terminates with success at the n-th iteration with the result φ'_n, then φ'_n is a backward invariant (below $\mathsf{nonInit}$) that can be expressed by $\mathcal{P}_n = \mathsf{atoms}(\varphi_n) \supseteq \mathsf{atoms}(\varphi'_n)$. Hence, by Lemma 1, the least fixpoint of $\mathsf{pre}^{\#}_n$ above unsafe is also a backward invariant (below $\mathsf{nonInit}$). Hence Method I also terminates with success. $\square$

Forward fixpoint iteration with backward refinement. Can we use abstract *forward* fixpoint iteration with backward refinement and still have relative completeness? The answer is yes if we use the dual $\widetilde{\mathsf{pre}}$ of pre for the backward refinement. The operator $\widetilde{\mathsf{pre}}$ (sometimes called the weakest liberal precondition operator) is defined by $\widetilde{\mathsf{pre}}(\varphi) = \neg\mathsf{pre}(\neg\varphi)$.

We define *dual* backward refinement as the procedure that iterates $\widetilde{\mathsf{pre}}$ starting from safe; i.e., it generates the sequence of sets of predicates $\mathcal{P}_i = \mathsf{atoms}(\varphi_i)$ $(n \geq 0)$ where

```
φ0 := safe
n := 0
loop
  P_n := atoms(φ_n)
  construct abstract operator post#_n defined by P_n
  ψ := lfp(post#_n, start)
  if (ψ ≤ safe) then
     STOP with "Success"
  φ_{n+1} := φ_n ∨ p̃re(φ_n)
  n := n+1
endloop
```

Fig. 3. Method III: forward abstract fixpoint iteration with backwards iterative abstraction refinement.

- $\varphi_0 = \mathsf{safe}$
- $\varphi_{n+1} = \varphi_n \vee \widetilde{\mathsf{pre}}(\varphi_n)$

This new method (Method III) is made precise in Figure 3. One possible interpretation of the following theorem is that the crucial item in the statement of Theorem 1 is the *backward* direction of the refinement (and not the direction of the abstract fixpoint iteration).

Theorem 2 (Relative Completeness of Abstract Forward Iteration with Dual Backward Refinement). *For every program, Method III is guaranteed to terminate with success if Method II terminates with success.*

Proof (of Theorem 2). Let Method II terminate with success at, say, the n-th iteration, with, say, the result ψ, then ψ is a backward invariant (more precisely, a $\langle \mathsf{pre}, \mathsf{unsafe}, \mathsf{nonInit}\rangle$-invariant). It is not difficult to show that $\neg\psi$ is a $\langle \mathsf{post}, \mathsf{init}, \mathsf{safe}\rangle$-invariant.

The formula ψ can be expressed by the set of predicates obtained by backward refinement in the sense of Section 3, i.e. starting with unsafe and iterating pre. If we call this set of predicates $\widetilde{\mathcal{P}_n}$, then $\neg\psi$ can be expressed by $\{\neg p \mid p \in \widetilde{\mathcal{P}_n}\}$, a set that is exactly the set $\mathcal{P}_n$ defined by 'dual' backward iteration, as used in Theorem 2.

Thus, we have a $\langle \mathsf{post}, \mathsf{init}, \mathsf{safe}\rangle$-invariant that can be expressed by the set of predicates $\mathcal{P}_n$. Using Lemma 1 with $\mathsf{post}^{\#}_n$ instantiated for $\mathsf{F}^{\#}_n$, we obtain that the forward abstract fixpoint iteration with backward abstraction refinement, terminates with success. □

6 Example: Forward vs. Backward Refinement

The example program in Figure 4 shows that the completeness of Method I relative to Method II does not hold for the forward case, i.e. when $\mathsf{F}, \mathsf{start}$ and

init $\equiv$ $pc = \ell_1$
unsafe $\equiv$ $pc = error$
variables $X = \{x, y, z\}$

guarded commands:

$$
\begin{array}{ll}
c_1: & pc = \ell_1 \rightarrow pc := \ell_2,\ x := 0 \\
c_2: & pc = \ell_2 \wedge x \geq 0 \rightarrow x := x + 1 \\
c_3: & pc = \ell_2 \wedge x < 0 \rightarrow pc := \ell_3 \\
c_4: & pc = \ell_3 \wedge y = 25 \rightarrow pc := \ell_4 \\
c_5: & pc = \ell_4 \wedge y \neq 25 \rightarrow pc := \ell_5 \\
c_6: & pc = \ell_5 \rightarrow pc := \ell_6; z := -1 \\
c_7: & pc = \ell_6 \wedge z \neq 0 \rightarrow z := z - 1 \\
c_8: & pc = \ell_6 \wedge z = 0 \rightarrow pc := error
\end{array}
$$

```
L1:  x = 0;
L2:  while (x >= 0) {
       x = x + 1;
     }
L3:  if (y == 25) {
L4:    if (y != 25) {
L5:      z = -1;
L6:      while (z != 0) {
           z = z - 1;
         }
         error:;
       }
     }
```

Fig. 4. Example program: Method I, forward abstract fixpoint iteration with forward refinement, does not terminate; Method II (iterative application of post and oracle-guided widening) terminates with success. We here use 'syntactic sugar' for guarded commands and list only the 'true' updates; for example, c_2 stands for the formula $pc = \ell_2 \wedge x \geq 0 \wedge x' = x + 1 \wedge pc' = pc \wedge y' = y \wedge z' = z$. The right hand side shows the program in C-like notation

bound are instantiated to post, init and safe, respectively. The values ℓ_1 through ℓ_6 for pc in the left hand side of Figure 4 correspond to labels `L1` through `L6` in the right hand side. In this example, for Method I to terminate, it is crucial to find the (contradictory) predicates $x = 25$ and $x \neq 25$. What is difficult is that the code path through these predicates is "bracketed" above and below by non-terminating **while** loops.

We observe the following facts about this example:

- Method II forward (iterative application of post and oracle-guided widening) terminates with success (the widening operator just drops all conjuncts containing the variable x).
- Method I with forward abstraction refinement does not terminate. Forward refinement will get "stuck" at the first **while** loop, generating an infinite sequence of predicates about x, namely $x = 0$, $x = 1$, $x = 2$, ...

This means that the analog of Theorem 1 does not hold for the forward case. Continuing the example, we also have that

- Method II (iterative application of pre and oracle-guided widening) terminates with success (the widening operator just drops all conjuncts containing the variable z).
- Method I backward terminates with success, which will follow by Theorem 1, but can also be checked directly by executing the method which terminates in four iterations. The first three iterations of pre are shown below. For

readability, conjuncts of the form $(c = c)$ for some constant c have been dropped.

$$\begin{array}{ll}
\mathsf{unsafe} = & (pc = \mathit{error}) \\
\mathsf{pre}(\mathsf{unsafe}) = & (pc = \ell_6 \wedge z = 0) \\
\mathsf{pre}^2(\mathsf{unsafe}) = & (pc = \ell_6 \wedge z \neq 0 \wedge z = 1) \vee \\
& (pc = \ell_5 \wedge -1 = 0) \\
\mathsf{pre}^3(\mathsf{unsafe}) = & (pc = \ell_6 \wedge z \neq 0 \wedge z = 2) \vee \\
& (pc = \ell_5 \wedge -1 \neq 0 \wedge -2 = 0) \vee \\
& (pc = \ell_4 \wedge y \neq 25 \wedge -1 = 0)
\end{array}$$

Note that it is crucial that we do not do a satisfiability test during the computation of pre; therefore, the backward refinement procedure retains disjuncts that have unsatisfiable conjuncts such as $-1 = 0$. Thus, the predicates $y = 25$, $y \neq 25$, $pc = \mathit{error}$, $pc = \ell_6$, $pc = \ell_5$, $pc = \ell_4$ are present in $\mathcal{P}_4$. These predicates are sufficient to ensure that $\mathsf{lfp}(\mathsf{pre}_4^{\#}, \mathsf{unsafe}) \leq \mathsf{nonInit}$.

As a secondary point, neither the iteration of the concrete post operator post nor the iteration of the concrete predecessor operator pre terminates (without using widening). We leave open the problem of designing a forward refinement procedure with relative completeness.

7 Discussion

BDD's Implement the Free Lattice. One appealing feature of predicate abstraction is that a finite-state model checker (based e.g. on BDD's) can be used to implemented the abstract fixpoint iteration. There, a Boolean variable is introduced for each predicate. This means that the logical meaning of the predicate is not used in the fixpoint termination test. Therefore, for example, the Boolean expression $[x < 2]$ (where $[x < 2]$ is the Boolean variable introduced for the predicate $x < 2$) is not subsumed by $[x < 3]$; the BDD's for $[x < 3]$ and $[x < 2] \vee [x < 3]$ are different.

We formally account for this situation by ordering formulas with "$\sqsubseteq$", the ordering of the free distributive lattice. This ordering is strictly stronger than the ordering "$\leq$" based on the provability of implication by a given theorem prover; i.e., if $\varphi \sqsubseteq \varphi'$ then $\varphi \leq \varphi'$ (by the monotonicity of the meaning function γ), but the converse does not hold in general; for example, $x < 2 \not\sqsubseteq x < 3$.

Relative completeness states that Method I, a fixpoint iteration over formulas with the ordering "$\sqsubseteq$", terminates if Method II, a fixpoint iteration over formulas with the ordering "$\leq$", terminates, although the fixpoint test of Method I is strictly weaker than one of Method II. How is this possible? — The following explanation is based on a technical intricacy (of predicate abstraction) related to Lemma 1.

We observe that the operator iterated in Method I is $\mathsf{F}_n^{\#}$, defined by $\alpha_n \circ \mathsf{F} \circ \gamma$ for some n. Let Method II terminate with, say, the formula φ, i.e., $\mathsf{F}(\varphi) \leq \varphi$; we are asking why then $\mathsf{F}_n^{\#}(\alpha_n(\varphi)) \sqsubseteq \alpha_n(\varphi)$ must hold. We have that $\mathsf{atoms}(\varphi) \subseteq \mathcal{P}_n$

and hence $\gamma \circ \alpha_n(\varphi) \leq \varphi$ (and not only $\varphi \leq \gamma \circ \alpha_n(\varphi)$). Therefore, and since α_n is monotonic, we have $\alpha_n \circ \mathsf{F} \circ \gamma(\alpha_n(\varphi)) \sqsubseteq \alpha_n(\varphi)$, which we wanted to show.

Boolean expressions. Our setting of the lattice $\mathcal{L}(\mathcal{P})$ generalizes the setting of Boolean expressions that has been used so far in work on abstract model checking [1, 2, 6, 8, 13, 15, 16, 24–28]. Our more general setting allows us to determine a sense in which the negated versions of predicates generated by the abstraction refinement procedure are useless. This is important because the time for constructing the abstract fixpoint operator is exponential in the number of predicates.

We obtain the setting of Boolean expressions as an instance of ours simply by adding the negated version of each predicate. Namely, the lattice of Boolean expressions over the set of predicates $\mathcal{P}$ is $\mathcal{L}(\mathcal{P} \cup \{\neg\varphi,\ \varphi \in \mathcal{P}\})$, the lattice generated by the positive and negated versions of predicates (i.e. atomic formulas).

In the setting of Boolean expressions, Theorem 2 holds with the same backward refinement procedure as in Theorem 1. In the present formulation, the backward refinement procedure considered in Theorem 2 is equivalent to iterating pre (starting with unsafe) and adding the negation of the predicates obtained. However, with Boolean expressions, generating the positive or the negated version of a predicate amounts to the same.

Refinement Based on Error Traces. The definition of the abstraction refinement procedure in Section 3 is modeled after the standard refinement procedure as implemented e.g. by Clarke et al. [5], Ball and Rajamani [3] (who took forward refinement) and Lakhnech et al. [24], Henzinger et al. [20], and Das et al. [12] (who took backward refinement). The definition abstracts away the technicalities of the particular implementation strategy where a 'spurious'-error execution trace is used to selectively add predicates that can express a specific set of reachable states (with the effect of eliminating that error trace); the definition amounts to consider all traces of the same length as the 'spurious' execution trace. Theorems 1 and 2 also hold if we take that implementation strategy (which still generates all 'necessary' predicates under the assumption of the theorems).

More Powerful Refinement. The backward refinement procedure enhances the standard one in that it adds also predicates that occur in unsatisfiable conjuncts. For example, if c is the guarded command $pc = \ell_5 \wedge z' = -1 \wedge pc' = \ell_6$, then $\mathsf{atoms}(\mathsf{pre}_c(pc = \ell_6 \wedge z = 0))$ is $\mathsf{atoms}(pc = \ell_5 \wedge -1 = 0)$, which consists of the two predicates $pc = \ell_5$ and $-1 = 0$ (see Section 6); the predicate $-1 = 0$ will not appear in $\alpha_n(\varphi)$ for any φ. In terms of a practical, error trace-based strategy, this means that one adds predicates to eliminate more spurious transitions of the error trace than just the first one.

Forward vs. Backward Refinement. It is perhaps intriguing as to why Method I is as powerful as Method II with backward refinement, but not with forward refinement. We first try to give some intuition for the difference between the two cases and then give a more technical explanation.

In the forward direction, the 'concrete' execution of each guarded command $c \in \mathcal{C}$ is deterministic (even though the entire system defined by a set $\mathcal{C}$ of guarded commands may be non-deterministic). An 'abstract' execution (where abstraction is induced e.g. by widening) is in general non-deterministic and can reach more program points (and other program expressions) than the concrete execution. Note that abstraction refinement must be based on the concrete execution (otherwise, the spuriousness of an abstract error trace can not be detected). The deterministic execution follows only one branch and hence it may get "stuck in a loop" (for example the loop in line L2 of Figure 4).

In the backward direction, the concrete execution already is (in general) non-deterministic and can follow several branches; hence it does not get stuck in a loop (for example the loop in line L6 of Figure 4) and can reach as many program points (expressions) as an abstract execution; in order to make this always true, pre must produce also disjuncts with unsatisfiable conjuncts; we added Line L5 in the program in Figure 4 to demonstrate this point.

A more technical explanation for the difference between the forward and the backward case lies in (5). It is well-known that the predecessor operator for deterministic programs (hence in particular for one guarded command c) distributes over intersection whereas the successor operator does not. Computing pre works by simple syntactic substitution. In contrast, computing post requires existential-quantifier elimination. Therefore, it seems difficult to find a realistic, powerful forward abstraction refinement procedure.

Widening. We use the notion of a widening operator essentially in the sense of [9, 11]. In the standard setting, a widening operator is a binary operator that assigns two elements x and x' another element $x \nabla x'$ that is larger than both. In this paper, each widening operator widen is unary. This makes a difference in the context of a fixed widening operator (the second argument is used to determine the 'direction' of the extrapolation of the first by $x \nabla x'$); it does not restrict the power of the extrapolation in our setting (for each application $x \nabla x'$ of the binary operator the oracle can guess a unary one which, applied to x, yields the same result).

The restriction on the form of $\mathsf{widen}(x)$ is motivated by the goal to model widening operators such that each application can realistically be implemented (although, of course, the oracle can not). The intuition is that boundaries that need to be moved in each fixpoint iteration are getting weaker and weaker and will be dropped in the limit. Many widening operators that have been implemented by Cousot, Halbwachs, Jeannet and others (see e.g. [11, 14, 17, 21]) seem to follow that intuition.

Widening vs. Predicate Abstraction. Our intent is not a comparison between the respective power of model checking based on predicate abstraction with refinement and of widening-based model checking. Such a comparison would be futile since the latter lacks the outer loop that performs a refinement of the abstraction. (What would such a loop look like in order to obtain relative completeness?)

It is illuminating, however, to see that predicate abstraction and widening can be formally related with each other as two abstraction methods for verification. Previously, this was thought to be impossible [23]. For static program analysis, widening was shown to be superior to predicate abstraction or any other 'static' abstraction [10]. As a consequence of our result, predicate abstraction with refinement can be understood as widening with 'optimal' guidance.

The converse of the theorems, i.e. the relative completeness of Method II wrt. Method I, does not hold.[4] Intuitively, this is because the widening in Method II (dropping a conjunct) is generally less precise than the extrapolation in Method II (which amounts to replacing a conjunct with a formula over already generated predicates). The converse would hold if we extended the widening accordingly. However, we consider that direction of relative completeness not interesting as long as we do not know of a realistic way to mimic the oracle for guessing a widening.

The power of either, Method I or II, depends on the given formalism which fixes the set of atomic formulas. For example, the methods are more powerful if equalities $x = y + c$ must be expressed by the conjunction of inequalities (e.g. if $\mathsf{atoms}(\{x = 0\})$ is not $\{x = 0\}$ but $\{x \leq 0,\ x \geq 0\}$, then Method I will succeed on the example in Section 6 also with forward refinement; similarly, Method I with backward refinement will succeed on the example program in [24]).

Termination for Incorrect Programs. Each verification method that we consider here can be modified in a straightforward way so that it will always detect (and will always terminate) in the case where the program is incorrect. The termination of a verification method is an issue only in the case of correct programs. Therefore we concentrate on that case, and gloss over the case of incorrect programs.

Finite quotients. If we assume that the program has a finite simulation or bisimulation quotient, termination of fixpoint computations can be guaranteed (both forward and backward) [25, 18]. Our work does not make any such assumptions. We focus on the uniform evaluation of a method on *all* instances of the undecidable verification problem (and not on the instances of a decidable subproblem).

Optimization. Generating a small set of predicates is always a desirable feature in designing a refinement procedure. This was not our focus in this paper. Instead, we defined what the goal of the refinement procedure should be, and designed a refinement procedure to meet this goal. Once this goal is established,

[4] To obtain a counterexample, consider a program with two independent branches, one that causes the generation of the predicates $x = y$ and $x = y + 1$, and another corresponding to the program fragment `x=0; y=0; while(*){x++; y++}; while(x!=0){y--; x--}; if(y!=0){error:}`.

and only after such a goal is formulated as an algorithmic problem, it is possible to propose and evaluate optimizations. Our work enables this to happen.

8 Conclusion

Automated refinement is presently the least understood part of automated program verification methods known under the term 'software model checking'. Up to now, different refinement procedures could be evaluated only practically, by comparing their implementations in various existing tools. The work presented here is the first that tries to evaluate them on a principled basis. We think that this is a starting point to arrive at a systematic way to design and analyze refinement procedures.

References

1. P. A. Abdulla, A. Annichini, S. Bensalem, A. Bouajjani, P. Habermehl, and Y. Lakhnech. Verification of infinite-state systems by combining abstraction and reachability analysis. In *CAV 99: Computer-aided Verification*, LNCS 1633, pages 146–159. Springer-Verlag, 1999.
2. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 01: Programming Language Design and Implementation*, pages 203–213. ACM, 2001.
3. T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 01: SPIN Workshop*, LNCS 2057, pages 103–122. Springer-Verlag, 2001.
4. R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
5. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Computer Aided Verification*, LNCS 1855, pages 154–169. Springer-Verlag, 2000.
6. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Computer-Aided Verification*, LNCS 1855, pages 154–169. Springer-Verlag, 2000.
7. S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal of Computing*, 7(1):70–91, February 1978.
8. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE 2000: International Conference on Software Engineering*, pages 439–448. ACM, 2000.
9. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL 79: Principles of Programming Languages*, pages 269–282. ACM, 1979.
10. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *Proceedings of PLILP 92: Programming Language Implementation and Logic Programming*, LNCS 631, pages 269–295. Springer-Verlag, 1992.
11. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL 78: Principles of Programming Languages*, pages 84–96. ACM, 1978.

12. S. Das and D. L. Dill. Successive approximation of abstract transition relations. In *LICS 01: Symposium on Logic in Computer Science*, 2001.
13. S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *CAV 00: Computer-Aided Verification*, LNCS 1633, pages 160–171. Springer-Verlag, 1999.
14. G. Delzanno and A. Podelski. Model checking in CLP. In *TACAS 99: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 1579, pages 223–239. Springer-Verlag, 1999.
15. R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples and refinements in abstract model checking. In *SAS 01: Static Analysis*, LNCS 2126, pages 356–373. Springer-Verlag, 2001.
16. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV 97: Computer-aided Verification*, LNCS 1254, pages 72–83. Springer-Verlag, 1997.
17. N. Halbwachs, Y.-E. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In *SAS 94: Static Analysis*, LNCS 864, pages 223–237. Springer-Verlag, 1994.
18. T. Henzinger and R. Majumdar. A classification of symbolic transition systems. In *STACS 00: Theoretical Aspects of Computer Science*, LNCS 1770, pages 13–34. Springer-Verlag, 2000.
19. T. A. Henzinger, P. Ho, and H. Wong-Toi. Hytech: a model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.
20. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. personal communication, May 2001.
21. B. Jeannet. *Dynamic partitionning in linear relation analysis and application to the verification of synchronous programs.* PhD thesis, Institut National Polytechnique de Grenoble, September 2000.
22. R. Kurshan. *Computer-aided Verification of Coordinating Processes.* Princeton University Press, 1994.
23. Y. Lakhnech. Personal communication, April 2001.
24. Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *TACAS 01: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 2031, pages 98–112. Springer-Verlag, 2001.
25. K. S. Namjoshi and R. P. Kurshan. Syntactic program transformations for automatic abstraction. In *CAV 00: Computer-Aided Verification*, LNCS 1855, pages 435–449. Springer-Verlag, 2000.
26. V. Rusu and E. Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. In *TACAS 99: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 1579, pages 178–192. Springer-Verlag, 1999.
27. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL 99: Principles of Programming Languages*, pages 105–118. ACM, 1999.
28. H. Saïdi and N. Shankar. Abstract and model check while you prove. In *CAV 99: Computer-aided Verification*, LNCS 1633, pages 443–454. Springer-Verlag, 1999.