

PREDICATE ABSTRACTION WITH UNDER-APPROXIMATION REFINEMENT

CORINA S. PĂȘĂREANU, RADEK PELÁNEK, AND WILLEM VISSER

QSS, NASA Ames Research Center, Moffett Field, CA 94035, USA
e-mail address: pcorina@email.arc.nasa.gov

Masaryk University Brno, Czech Republic
e-mail address: xpelaneck@fi.muni.cz

RIACS/USRA, NASA Ames Research Center, Moffett Field, CA 94035, USA
e-mail address: wvisser@email.arc.nasa.gov

ABSTRACT. We propose an abstraction-based model checking method which relies on refinement of an under-approximation of the feasible behaviors of the system under analysis. The method preserves errors to safety properties, since all analyzed behaviors are feasible by definition. The method does not require an abstract transition relation to be generated, but instead executes the concrete transitions while storing abstract versions of the concrete states, as specified by a set of abstraction predicates. For each explored transition the method checks, with the help of a theorem prover, whether there is any loss of precision introduced by abstraction. The results of these checks are used to decide termination or to refine the abstraction by generating new abstraction predicates. If the (possibly infinite) concrete system under analysis has a finite bisimulation quotient, then the method is guaranteed to eventually explore an equivalent finite bisimilar structure. We illustrate the application of the approach for checking concurrent programs.

1. INTRODUCTION

Over the last few years, model checking based on abstraction-refinement has become a popular technique for the analysis of software. In particular the abstraction technique of choice is a property preserving over-approximation called predicate abstraction [18] and the refinement removes spurious behavior based on automatically analyzing abstract counter-examples. This approach is often referred to as CEGAR (counter-example guided automated refinement) and forms the basis of some of the most popular software model checkers [4, 7, 22]. Furthermore, a strength of model checking is its ability to automate

2000 ACM Subject Classification: D.2.4, F.3.1.

Key words and phrases: model checking, predicate abstraction, under-approximation, automatic abstraction refinement.

This paper is an extended version of [29].

Partially supported by the Grant Agency of Czech Republic grant No. 201/07/P035 and by the Academy of Sciences of Czech Republic grant No. 1ET408050503.

the detection of subtle errors and to produce traces that exhibit those errors. However, over-approximation based abstraction techniques are not particularly well suited for this, since the detected defects may be spurious due to the over-approximation — hence the need for refinement. We propose an alternative approach based on refinement of under-approximations, which effectively preserves the defect detection ability of model checking in the presence of aggressive abstractions.

The technique uses a combination of (explicit state) model checking, predicate abstraction and automated refinement to efficiently analyze increasing portions of the feasible behavior of a system. At each step, either an error is found, we are guaranteed no error exists, or the abstraction is refined. More precisely, the proposed model checking technique traverses the concrete transitions of the system and for each explored concrete state, it stores an abstract version of the state. The abstract state, computed by predicate abstraction, is used to determine whether the model checker’s search should continue or backtrack (if the abstract state has been visited before). This effectively explores an under-approximation of the feasible behavior of the analyzed system. Hence all counter-examples to safety properties are preserved.

Refinement uses weakest precondition calculations to check, with the help of a theorem prover, whether the abstraction introduces any loss of precision with respect to each explored transition. If there is no loss of precision due to abstraction (we say that the abstraction is *exact*) the search stops and we conclude that the property holds. Otherwise, the results from the failed checks are used to refine the abstraction and the whole verification process is repeated anew. In general, the iterative refinement may not terminate. However, if a finite bisimulation quotient [24] exists for the system under analysis, then the proposed approach is guaranteed to eventually explore a finite structure that is bisimilar to the original system.

The technique can also be used in a lightweight manner, without a theorem prover, i.e. the refinement guided by the exactness checks is replaced with refinement based on syntactic substitutions [26] or heuristic refinement. The proposed technique can be used for systematic testing, as it examines increasing portions of the system under analysis. In fact, our method extends existing approaches to testing that use abstraction mappings [19, 35], by adding support for automated abstraction refinement.

Our approach can be contrasted with the work on predicate abstraction for modal transition systems [16, 31], used in the verification and refutation of branching time temporal logic properties. An abstract model for such logics distinguishes between *may* transitions, which over-approximate transitions of the concrete model, and *must* transitions, which under-approximate the concrete transitions (see also [2, 10, 11, 30]). As we show in the next section (and we discuss in more detail in Section 6), the technique presented here explores and generates a structure which is more precise (contains more feasible behaviors) than the model defined by the *must* transitions, for the same abstraction predicates. The reason is that the model checker explores transitions that correspond not only to *must* transitions, but also to *may* transitions that are feasible.

Moreover, unlike [16, 31] and over-approximation based abstraction techniques [4, 7], the under-approximation and refinement approach does not require the a priori construction of the abstract transition relation, which involves exponentially many theorem prover calls (in the number of predicates), regardless of the size of (the reachable portion of) the analyzed system. In our case, the model checker executes concrete transitions and a theorem prover is only used during refinement, to determine whether the abstraction is exact with respect to each executed transition. Every such calculation makes at most two theorem prover

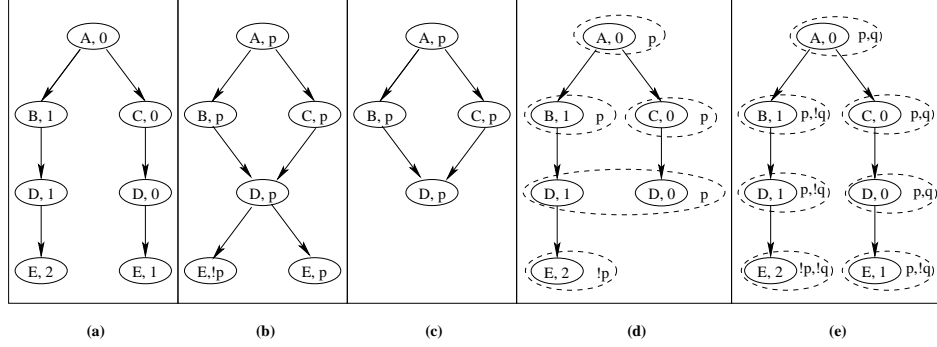


Figure 1: **(a)** Concrete system **(b)** *May* abstraction using predicate $p = x < 2$ **(c)** *Must* abstraction using p **(d)** Concrete search with abstract matching using p **(e)** Concrete search with abstract matching using predicates p and $q = x < 1$

calls, and it involves only the *reachable* state space of the system under analysis. Another difference with previous abstraction techniques is that the refinement process is not guided by the spurious counter-examples, since no spurious behavior is explored. Instead, the refinement is guided by the failed exactness checks for the explored transitions.

To the best of our knowledge, the presented approach is the first predicate abstraction based analysis which focuses on automated refinement of under-approximations with the goal of efficient error detection. We illustrate the application of the approach for checking safety properties in concurrent programs.

The rest of the paper is organized as follows. Section 2 shows an example illustrating our approach. Section 3 gives background information. Section 4 describes the main algorithm for performing concrete model checking with abstract matching and refinement. Section 5 discusses correctness and termination; Section 6 discusses other interesting properties for the algorithm. Section 7 proposes extensions to the algorithm. Section 8 illustrates applications of the approach, Section 9 discusses related work, and Section 10 concludes the paper.

2. EXAMPLE

The example in Figure 1 illustrates some of the main characteristics of our approach. Figure 1 **(a)** shows the state space of a concrete system that has only one variable x ; states are labelled with the program counter (e.g. A, B, C, \dots) and the concrete value of x . Figure 1 **(b)** shows the abstract system induced by the *may* transitions for predicate $p = x < 2$. Figure 1 **(c)** shows the abstract system induced by the *must* transitions for predicate p .

Figure 1 **(d)** shows the state space explored using our proposed approach, for an abstraction specified by predicate p . Dotted circles denote the abstract states which are stored, and used for matching, during the concrete execution of the system. The approach explores only the *feasible* behavior of the concrete system, following transitions that correspond to both *may* and *must* transitions, but it might miss behavior due to abstract matching. For example, state $(E, 1)$ is not explored, assuming a breadth-first search, since $(D, 0)$ was matched with $(D, 1)$ — both have the same program counter and both satisfy p . Notice that, with respect to reachable states, the produced structure is a better under-approximation (it “covers” more states) than the *must* abstraction. Figure 1 **(e)** illustrates

concrete execution with abstract matching, after a refinement step, which introduced a new predicate $q = x < 1$. The resulting structure is an exact abstraction of the concrete system.

3. BACKGROUND

3.1. Program Model and Semantics. To make the presentation simple, we use as a specification language a guarded commands language over integer variables. Most of the results extend directly to more sophisticated programming languages. Let V be a finite set of integer variables. Expressions over V are defined using standard boolean ($=, <, >$) and binary ($+, -, \cdot, \dots$) operations.

Definition 1. A *model* is a tuple $M = (V, T)$. $T = \{t_1, \dots, t_k\}$ is a finite set of transitions, where $t_i = (g_i(\vec{x}) \mapsto \vec{x} := e_i(\vec{x}))$, $g_i(\vec{x})$ is a guard and $e_i(\vec{x})$ are assignments to the variables represented by tuple \vec{x} .

Throughout the paper, we write concurrent assignments $\vec{x} := e_i(\vec{x})$ as sequences, to improve readability. The semantics of program models uses transition systems.

Definition 2. A *transition system* over a finite set of atomic propositions AP is a tuple (S, R, s_0, L) where S is a (possibly infinite) set of states, $R = \{\xrightarrow{i}\}$ is a finite set of deterministic transition relations: $\xrightarrow{i} \subseteq S \times S$, s_0 is an initial state, and $L : S \rightarrow 2^{AP}$ is a labeling function.

State s is *reachable* if there exists a sequence of zero or more transitions from the initial state such that $s_0 \xrightarrow{i_1} s_1 \xrightarrow{i_2} s_2 \dots \xrightarrow{i_n} s_n = s$ (denoted $s_0 \xrightarrow{*} s$). The set of *reachable labelings* $RL(T)$ is $\{L(s) \mid s \in S : s_0 \xrightarrow{*} s\}$. The notation $s \not\xrightarrow{i}$ means that there is no i transition from the state s .

Definition 3. The *concrete semantics* of model M is transition system $\llbracket M \rrbracket = (S, \{\xrightarrow{i}\}, s_0, L)$ over AP , where:

- $S = 2^{V \rightarrow \mathbb{Z}}$, i.e. states are valuations of variables,
- $s \xrightarrow{i} s' \Leftrightarrow s \models g_i \wedge s' = e_i(s)$; the semantics of guards (boolean expressions) and updates is as usual; guards are functions $(V \rightarrow \mathbb{Z}) \rightarrow \{true, false\}$, written as $s \models g_i$; updates are functions $e_i : (V \rightarrow \mathbb{Z}) \rightarrow (V \rightarrow \mathbb{Z})$,
- s_0 is the zero valuation ($\forall v \in V : s_0(v) = 0$),
- $L(s) = \{p \in AP \mid s \models p\}$.

3.2. Strongest Postcondition and Weakest Precondition. Let ϕ be a predicate representing a set of states. Then the *strongest postcondition* of ϕ with respect to transition i is $sp(\phi, i) = \exists s'. (s' \xrightarrow{i} s \wedge \phi(s'))$; $sp(\phi, i)$ defines the successors by transition i of the states characterized by ϕ . The *weakest precondition* of ϕ with respect to transition i is $wp(\phi, i) = \forall s'. (s \xrightarrow{i} s' \Rightarrow \phi(s'))$; $wp(\phi, i)$ characterizes the largest set of states whose successors by transition i satisfy ϕ . For guarded commands, the weakest precondition can be expressed as $wp(\phi, i) = (g_i \Rightarrow \phi[e_i(\vec{x})/\vec{x}])$. We will use the following property [18]: $sp(\phi, i) \Rightarrow \phi'$ iff $\phi \Rightarrow wp(i, \phi')$.

3.3. Predicate Abstraction. Predicate abstraction is a special instance of the framework of abstract interpretation [9] that maps a (potentially infinite state) transition system into a finite state transition system via a set of predicates $\Phi = \{\phi_1, \dots, \phi_n\}$ over the program variables. Let \mathbb{B}_n be a set of bitvectors of length n . We define abstraction function $\alpha_\Phi : S \rightarrow \mathbb{B}_n$, such that $\alpha_\Phi(s)$ is a bitvector $b_1 b_2 \dots b_n$ such that $b_i = 1 \Leftrightarrow s \models \phi_i$. Let Φ_s be the set of all abstraction predicates that evaluate to *true* for a given state s , i.e. $\Phi_s = \{\phi \in \Phi \mid s \models \phi\}$. For succinctness we sometimes write $\alpha_\Phi(s)$ to denote $\bigwedge_{\phi \in \Phi_s} \phi \wedge \bigwedge_{\phi \notin \Phi_s} \neg \phi$.

We also give here the definitions of *may* and *must* abstract transitions. Although not necessary for formalizing our algorithm, these definitions clarify the comparison with related work. For two abstract states (bitvectors) a_1 and a_2 :

- $a_1 \xrightarrow{i}_{must} a_2$ iff for all concrete states s_1 such that $\alpha_\Phi(s_1) = a_1$, there exists concrete state s_2 such that $\alpha_\Phi(s_2) = a_2$ and $s_1 \xrightarrow{i} s_2$,
- $a_1 \xrightarrow{i}_{may} a_2$ iff there exists concrete state s_1 such that $\alpha_\Phi(s_1) = a_1$ and there exists concrete state s_2 such that $\alpha_\Phi(s_2) = a_2$, such that $s_1 \xrightarrow{i} s_2$.

Algorithms for computing abstractions using over-approximation based predicate abstraction are given in e.g. [4, 18] (they compute *may* abstract transitions automatically, with the help of a theorem prover). In the worst case, these algorithms make $2^n \times n \times 2$ calls to the theorem prover for each program transition.

3.4. Bisimulation.

Definition 4. A symmetric relation $R \subseteq S \times S$ is a *bisimulation relation* iff for all $(s, s') \in R$:

- $L(s) = L(s')$
- For every $s \xrightarrow{i} s_1$ there exists $s' \xrightarrow{i} s'_1$ such that $R(s_1, s'_1)$

The *bisimulation* is the largest bisimulation relation, denoted \sim . Two transition systems are bisimilar if their initial states are bisimilar. As \sim is an equivalence relation, it induces a *quotient* transition system whose states are equivalence classes with respect to \sim and there is a transition between two equivalence classes A and B if $\exists s_1 \in A$ and $\exists s_2 \in B$ such that $s_1 \xrightarrow{i} s_2$.

4. CONCRETE MODEL CHECKING WITH ABSTRACT MATCHING

4.1. Algorithm. Figure 2 shows the reachability procedure that performs model checking with abstract matching (α SEARCH). It is basically concrete state space exploration with matching on abstract states; the main modification with respect to classical state space search is that we store $\alpha_\Phi(s)$ instead of s . The procedure uses the following data structures:

- *States* is a set of abstract states visited so far,
- *Transitions* is a set of abstract transitions visited so far,
- *Wait* is a set of concrete states to be explored.

The procedure performs validity checking, using a theorem prover, to determine whether the abstraction is *exact* with respect to each explored transition — see discussion below. The set Φ_{new} maintains the list of abstraction predicates. The procedure returns the computed structure and a set of new predicates that are used for refinement. Note that we never abstract the program counter.

```

proc  $\alpha\text{SEARCH}(M, \Phi)$ 
   $\Phi_{\text{new}} = \Phi$ ; add  $s_0$  to Wait; add  $\alpha_\Phi(s_0)$  to States
  while Wait  $\neq \emptyset$  do
    get  $s$  from Wait
     $L(\alpha_\Phi(s)) = \{a \in AP \mid s \models a\}$ 
    foreach  $i$  from 1 to  $n$  do
      if  $s \models g_i$  then
        if  $\alpha_\Phi(s) \Rightarrow g_i$  is not valid
          then add  $g_i$  to  $\Phi_{\text{new}}$  fi
         $s' = e_i(s)$ 
        if  $\alpha_\Phi(s) \Rightarrow \alpha_\Phi(s')[e_i(\vec{x})/\vec{x}]$  is not valid
          then add predicates in  $\alpha_\Phi(s')[e_i(\vec{x})/\vec{x}]$  to  $\Phi_{\text{new}}$  fi
        if  $\alpha_\Phi(s') \notin \text{States}$  then
          add  $s'$  to Wait
          add  $\alpha_\Phi(s')$  to States
        fi
        add  $(\alpha_\Phi(s), i, \alpha_\Phi(s'))$  to Transitions
      else
        if  $\alpha_\Phi(s) \Rightarrow \neg g_i$  is not valid
          then add  $g_i$  to  $\Phi_{\text{new}}$  fi
      fi
    od
  od
   $A = (\text{States}, \text{Transitions}, \alpha_\Phi(s_0), L)$ 
  return  $(A, \Phi_{\text{new}})$ 
end

```

Figure 2: Search procedure with checking for exact abstraction

Figure 3 gives the iterative refinement algorithm for checking whether M can reach an error state described by φ (which is a boolean combination of propositions from AP). The algorithm starts with AP as the initial set of abstraction predicates. At each iteration of the loop, the algorithm invokes procedure αSEARCH to analyze an under-approximation of the system, which either violates the property, it is proved to be correct (if the abstraction is found to be exact with respect to all transitions), or it needs to be refined. Counter-examples are generated as usual (with depth-first search order using the stack, with breadth-first search order using parent pointers).

4.2. Checking for Exact Abstraction and Refinement. We say that abstraction function α_Φ is *exact* with respect to transition $s \xrightarrow{i} s'$ iff for all s_1 such that $\alpha_\Phi(s) = \alpha_\Phi(s_1)$ there exists s'_1 such that $\alpha_\Phi(s'_1) = \alpha_\Phi(s')$ and $s_1 \xrightarrow{i} s'_1$. In other words, $s \xrightarrow{i} s'$ is exact with respect to α_Φ iff $\alpha_\Phi(s) \xrightarrow{i}_{\text{must}} \alpha_\Phi(s')$.

Moreover, the abstraction function α_Φ is *exact* with respect to a state s iff the following conditions hold: (1) α_Φ is exact with respect to all transitions $s \xrightarrow{i} s'$ and (2) if $s \not\xrightarrow{i}$ then for all s_1 such that $\alpha_\Phi(s) = \alpha_\Phi(s_1)$ we have $s_1 \not\xrightarrow{i}$.

```

proc REFINEMENTSEARCH( $M, \varphi$ )
   $j = 1$ ;  $\Phi_j = AP$ 
  while true do
     $(A_i, \Phi_{j+1}) = \alpha\text{SEARCH}(M, \Phi_j)$ 
    if  $\varphi$  is reachable in  $A_j$  then return counter-example fi
    if  $\Phi_{j+1} = \Phi_j$  then return unreachable fi
     $j = j + 1$ 
  od
end

```

Figure 3: Iterative refinement algorithm

The notion of exactness is related to *completeness* in abstract interpretation (see [14]), which states that no loss of precision is introduced by the abstraction. Checking that the abstraction is exact with respect to a concrete transition $s \xrightarrow{i} s'$ amounts to checking that $sp(\alpha_\Phi(s)) \Rightarrow \alpha_\Phi(s')$, equivalent to $\alpha_\Phi(s) \Rightarrow wp(\alpha_\Phi(s'), i)$, is valid.

Note that $wp(\alpha_\Phi(s'), i) = (g_i \Rightarrow \alpha_\Phi(s')[e_i(\vec{x})/\vec{x}])$. Therefore $\alpha_\Phi(s) \Rightarrow wp(\alpha_\Phi(s'), i)$ is equivalent to $\alpha_\Phi(s) \Rightarrow (g_i \Rightarrow \alpha_\Phi(s')[e_i(\vec{x})/\vec{x}])$. The abstraction is exact with respect to state s when the following conditions hold: (1) $\alpha_\Phi(s) \Rightarrow (g_i \wedge \alpha_\Phi(s')[e_i(\vec{x})/\vec{x}])$, equivalent to $(\alpha_\Phi(s) \Rightarrow g_i) \wedge (\alpha_\Phi(s) \Rightarrow \alpha_\Phi(s')[e_i(\vec{x})/\vec{x}])$, is valid for each i such that $s \models g_i$ and (2) $\alpha_\Phi(s) \Rightarrow \neg g_i$ is valid for each i such that $s \not\models g_i$.

Checking the validity for these formulas is in general undecidable. As is customary, if the theorem prover can not decide the validity of a formula, we assume that it is not valid. This may cause some unnecessary refinement, but it keeps the correctness of the approach. If the abstraction can not be proved to be exact with respect to some transition, then the new predicates from the failed formula are added to the set of abstraction predicates. Intuitively, these predicates will be useful for proving exactness in the next iteration.

5. CORRECTNESS AND TERMINATION

In this section we discuss the main properties of the iterative refinement algorithm. We first state the main theorems, after which we give the technical lemmas and proofs (the reader may wish to skip this technical material on the first reading).

5.1. Main Results. We first show that, if the iterative algorithm terminates then the result is correct and moreover, if the error state is unreachable, the output structure is bisimilar to the system under analysis:

Theorem 1. (Correctness) If REFINEMENTSEARCH(M, φ) terminates then:

- if it returns a counter-example, then it is a real error,
- if it returns ‘unreachable’, then the error state is indeed unreachable in M and moreover the computed structure is bisimilar to $\llbracket M \rrbracket$.

In general, the proposed algorithm might not terminate (the reachability problem for our modeling language is undecidable). However, the algorithm is guaranteed to eventually find all the reachable labelings (including all the reachable errors) of the concrete program, although it might not be able to detect that (to decide termination). Moreover, if the

(reachable part of the) system under analysis has a finite bisimulation quotient, then the algorithm eventually produces a finite bisimilar structure.

Theorem 2. (Termination) Let the αSEARCH use breadth-first search order and let A_1, A_2, \dots be a sequence of transition systems generated during iterative refinement performed by $\text{REFINEMENTSEARCH}(M, \varphi)$. Then

- there exists j such that $RL(A_j) = RL(\llbracket M \rrbracket)$,
- if the reachable part of the bisimulation quotient is finite, then there exists j such that $A_j \sim \llbracket M \rrbracket$.

Note that a consequence of this theorem is that if an error is reachable it is eventually reported by our algorithm. Also note that for the second part of the theorem, we do not require that both the reachable and unreachable parts of the system have a finite bisimulation quotient, but only the reachable part needs to be finite (of course, if both the reachable and unreachable parts are finite, then it follows that the reachable part is also finite; the converse is not true).

5.2. Technical Material. Here we provide several technical lemmas and the proofs for the two main theorems. We use the following notation: a state s is *visited* during the search if it is inserted into *Wait*; a state s is *considered* during the search if it is generated as a successor of some state in the **foreach** loop; a state s_1 is *matched* to a state s_2 if the check $\alpha_\Phi(s_1) \notin \text{States}$ fails because $\alpha_\Phi(s_1) = \alpha_\Phi(s_2)$ and s_2 was visited before.

We say that transition $s \xrightarrow{i} s'$ is exact if α_Φ is exact with respect to it. Note that sometimes we let $\alpha\text{SEARCH}(M, \Phi)$ denote just the structure A computed by the algorithm and not the tuple (A, Φ_{new}) . Also note that REFINEMENTSEARCH starts with AP as the initial set of predicates. For the proofs, we need to refine the definition of bisimulation.

Definition 5. A symmetric relation $R \subset S \times S$ is a k -bisimulation relation iff:

- for all $(s, s') \in R : L(s) = L(s')$
- if $k > 0$ then there exists $(k-1)$ -bisimulation relation R' such that for all $(s, s') \in R : (\forall s \xrightarrow{i} s_1 \Rightarrow \exists s' \xrightarrow{i} s'_1 \wedge (s_1, s'_1) \in R')$

The k -bisimulation is the largest k -bisimulation relation, denoted \sim_k . Note that the bisimulation is a k -bisimulation relation for every k .

Proof of Theorem 1. We first show that the reachable labelings computed by the iterative algorithm REFINEMENTSEARCH is indeed an *under-approximation* of the reachable labelings of the program under analysis (Lemmas 1 and 2). Therefore, all the reported counter-examples correspond to real errors. We then show that when REFINEMENTSEARCH reports 'unreachable' (i.e. when the set Φ_{new} of predicates returned for the current iteration is equal to the set Φ of predicates from the previous iteration) then the computed structure A is bisimilar to $\llbracket M \rrbracket$ (Lemmas 3 and 4).

Lemma 1. If a state s is reachable in $\llbracket M \rrbracket$ via exact transitions with respect to α_Φ , then there exists s' such that s' is visited during the $\alpha\text{SEARCH}(M, \Phi)$ and $\alpha_\Phi(s) = \alpha_\Phi(s')$.

Proof: By induction with respect to the number of exact transitions from the initial state necessary for reaching the state s . Basic step ($k = 0$) is trivial. For the induction step, suppose that state s is reachable via sequence of exact transitions: $s_0 \xrightarrow{i_0} \dots \xrightarrow{i_{k-1}}$

$s_k \xrightarrow{i_k} s_{k+1} = s$. By the induction hypothesis there exists s'_k such that s'_k is visited and $\alpha_\Phi(s'_k) = \alpha_\Phi(s_k)$. Because the abstraction is exact with respect to $s_k \xrightarrow{i_k} s$, there must be s' such that $s'_k \xrightarrow{i_k} s'$ and $\alpha_\Phi(s') = \alpha_\Phi(s)$. This successor s' is considered during the visit of s'_k . There are two cases to be analyzed.

- (1) s' is added to *Wait* and later visited,
- (2) s' is matched to a previously visited state s'' such that $\alpha_\Phi(s') = \alpha_\Phi(s'')$.

In both cases we get that some state with the same abstract counterpart as s is visited during the search. \square

Lemma 2. $RL(\alpha\text{SEARCH}(M, \Phi)) \subseteq RL(\llbracket M \rrbracket)$.

Proof: It is easy to verify that the following is an invariant of the search: '*Wait*' is a subset of reachable states in $\llbracket M \rrbracket$. The lemma follows. \square

Lemma 3. Let $AP \subseteq \Phi$. If for all reachable states s_1, s_2 it holds that $\alpha_\Phi(s_1) = \alpha_\Phi(s_2) \Rightarrow s_1 \sim s_2$, then $\alpha\text{SEARCH}(M, \Phi) \sim \llbracket M \rrbracket$.

Proof: Consider relation R defined as: $s_1 R s_2$ iff $s_1 = s_2$ or s_1 is matched to s_2 . Then R is a bisimulation relation between $\alpha\text{SEARCH}(M, \Phi)$ and $\llbracket M \rrbracket$. \square

Lemma 4. Let $(A, \Phi_{\text{new}}) = \alpha\text{SEARCH}(M, \Phi)$. If $\Phi_{\text{new}} = \Phi$, then $A \sim \llbracket M \rrbracket$.

Proof: Due to Lemma 3 it is sufficient to show that if $\Phi_{\text{new}} = \Phi$ then α_Φ induces a bisimulation relation on the reachable part of the transition system $\llbracket M \rrbracket$. We first show that every reachable state in $\llbracket M \rrbracket$ is reached by exact transitions. We proceed on induction by the number of transitions from the initial state to s . Basic step ($k = 0$) is trivial. For the induction step, suppose that state s is reachable via a sequence of exact transitions of length k . By Lemma 1 some state s' such that $\alpha_\Phi(s) = \alpha_\Phi(s')$ is visited during the search. During the visit of the state s' we check exactness of the abstraction (see Section 4.2). Since $\Phi_{\text{new}} = \Phi$ it follows that the abstraction is exact for s' , i.e., $s' \not\xrightarrow{i}$ iff $s \not\xrightarrow{i}$ and for every outgoing transition $s' \xrightarrow{i} s'_1$ and $\alpha(s) = \alpha(s')$ there exists s_1 such that $s \xrightarrow{i} s_1$ and $\alpha(s_1) = \alpha(s'_1)$. Since i is deterministic, it follows that s_1 is the only successor of s by transition i and transition $s \xrightarrow{i} s_1$ is also exact. Moreover, it satisfies the same criterion for bisimulation, i.e. for all s'' such that $\alpha_\Phi(s) = \alpha_\Phi(s'')$ there exists s''_1 such that $\alpha_\Phi(s_1) = \alpha_\Phi(s''_1)$ and $s'' \xrightarrow{i} s''_1$. \square

Proof: [of Theorem 1] The first claim follows from the fact that αSEARCH produces an under-approximation (Lemma 2). The second claim follows from Lemma 4. \square

Proof of Theorem 2. In order to prove Theorem 2, we study sequences $\{A_j\}_{j=0}^\infty$ of transition systems generated during REFINEMENTSEARCH. We assume that αSEARCH uses breadth-first search order. The basic idea of the proof is that any two states that are in different bisimulation classes ($s \not\sim s'$) are eventually distinguished by the abstraction function, i.e. $\exists j$ such that $\alpha_{\Phi_j}(s) \neq \alpha_{\Phi_j}(s')$ (Lemma 5). Moreover, each bisimulation class of $\llbracket M \rrbracket$ is eventually visited by REFINEMENTSEARCH (Lemma 6) and the finite set of reachable labelings emerges (Lemmas 7 and 8).

Lemma 5. Let $\{A_j\}_{j=0}^\infty$ be a sequence of transition systems generated during an infinite run of REFINEMENTSEARCH and $\text{Inf}_M = \{s \mid \text{there exists infinitely many } j \text{ such that } s \in A_j\}$. If $s \not\sim s'$ and $s \in \text{Inf}_M$ then there exists j such that $\alpha_{\Phi_k}(s) \neq \alpha_{\Phi_k}(s')$ for all $k \geq j$.

Proof: By induction with respect to k where k is the smallest number such that $s \not\sim_k s'$. Basic step: for $k = 0$ it means that $L(s) \neq L(s')$ and therefore $\alpha_{\Phi_1}(s) \neq \alpha_{\Phi_1}(s')$. Induction step ($k + 1$): Let s_1, s'_1 be such that $s \xrightarrow{i} s_1, s' \xrightarrow{i} s'_1$ and $s_1 \not\sim_k s'_1$. Since s is visited in infinitely many iterations of αSEARCH , s_1 is considered in infinitely many iteration of αSEARCH and therefore one of the following must hold:

- (1) State $s_1 \in \text{Inf}_M$. Then we can apply induction hypothesis, i.e. there exists j such that $\alpha_{\Phi_k}(s_1) \neq \alpha_{\Phi_k}(s'_1)$ for all $k \geq j$.
- (2) State s_1 is matched to some state in infinitely many runs of αSEARCH . Since we use breadth-first order, there are only finitely many states to which it can be matched (with breadth-first search order the state can be matched only to states with lower or equal distance from the initial state). Therefore, there exists a state s_2 such that s_1 is matched to s_2 in infinitely many runs of αSEARCH , this means that $\alpha_{\Phi_j}(s_1) \neq \alpha_{\Phi_j}(s_2)$ for all j . From the induction hypothesis we get that $s_1 \sim_k s_2$ and hence $s_2 \not\sim_k s'_1$. Moreover, from the induction hypothesis we get that there exists m such that $\alpha_{\Phi_k}(s_2) \neq \alpha_{\Phi_k}(s'_1)$ for all $k \geq m$. Therefore $\alpha_{\Phi_k}(s_1) \neq \alpha_{\Phi_k}(s'_1)$ for all $k \geq m$.

In both cases we get that there exists j such that α_{Φ_j} is not exact with respect to $s \xrightarrow{i} s_1$, therefore $wp(\alpha_{\Phi_j}(s_1), t_i)$ will be included in Φ_{j+1} and therefore $\alpha_{\Phi_{j+1}}(s_1) \neq \alpha_{\Phi_{j+1}}(s'_1)$. \square

Lemma 6. For each reachable bisimulation class B of $\llbracket M \rrbracket$ there exists a state $s \in B$ such that s is visited by $\text{REFINEMENTSEARCH}(M, \varphi)$ infinitely often.

Proof: By induction with respect to the length of the shortest path by which some state from B is reachable. Basic step is obvious. Induction step: let state from B be reachable via path $s_0 \xrightarrow{i_0} \dots \xrightarrow{i_{k-1}} s_k \xrightarrow{i_k} s_{k+1}$. By induction hypothesis some state $s' \sim s_k$ is reached during the refinement search infinitely often. Consider state s'' such that $s' \xrightarrow{i_k} s''$. It holds that $s'' \sim s_{k+1}$ and from Lemma 5 we get that s'' is visited infinitely often. \square

Lemma 7. Let $\{A_j\}_{j=0}^\infty$ be a sequence of transition systems generated during an infinite run of $\text{REFINEMENTSEARCH}(M, \varphi)$. There exists j such that $RL(A_j) = RL(\llbracket M \rrbracket)$.

Proof: For each $l \in RL(\llbracket M \rrbracket)$ we choose some bisimulation class B such that $s \in B \Rightarrow L(s) = l$. In this way we obtain a finite set of bisimulation classes $\{B_1, \dots, B_k\}$ which covers all labels in $RL(\llbracket M \rrbracket)$. Note that $RL(\llbracket M \rrbracket)$ is finite because AP is finite. Now we show that there exists an iteration in which at least one state from each of these classes is visited. This is done similarly to the proof of Lemma 6. \square

Lemma 8. Let $\{A_j\}_{j=0}^\infty$ be a sequence of transition systems generated during an infinite run of REFINEMENTSEARCH . If the reachable part of the bisimulation quotient is finite, then there exists j such that $A_j \sim \llbracket M \rrbracket$.

Proof: By contradiction. Suppose that $\forall j : A_j \not\sim \llbracket M \rrbracket$. From Lemma 3 we get that there exists reachable s, s' such that $\forall j : \alpha_{\Phi_j}(s) = \alpha_{\Phi_j}(s')$ and $s \not\sim s'$. We show (similarly to the proof of Lemma 1) that there exists such s which is visited infinitely often. From Lemma 5 we get that eventually $\alpha_{\Phi_j}(s) \neq \alpha_{\Phi_j}(s')$ which is the contradiction. \square

Proof: [of Theorem 2] This theorem is a direct consequence of Lemmas 7 and 8. \square

$$\begin{array}{ll} pc = 0 \wedge y \geq 0 & \longmapsto y := y + x \\ pc = 0 \wedge y < 0 & \longmapsto pc := 1 \end{array}$$

Figure 4: Example illustrating non-terminating refinement for finite state systems

6. PROPERTIES

Having discussed correctness and termination, we now turn to other interesting properties of the algorithm.

6.1. Non-termination for Finite State System. We should note that the proposed iterative algorithm is not guaranteed to terminate even for a finite state program. This situation is illustrated by the example from Figure 4; x and y are initialized to zero. The property that we check is that " $pc=1$ " is unreachable. Although the program is finite state (and therefore the problem can be easily solved with classical explicit model checking), it is quite difficult to solve using abstraction refinement techniques. The iterative algorithm does not terminate on this example: it keeps adding predicates $y \geq 0, y + x \geq 0, y + 2x \geq 0, \dots$. Note that, in accordance with Theorem 2, it eventually produces a bisimilar structure. However, the algorithm is not able to detect termination, and it keeps refining indefinitely. The reason is that the algorithm keeps adding predicates that refine the unreachable part of the system under analysis.

Also note that the same problem occurs with over-approximation based abstraction techniques that use refinement based on weakest precondition calculations [7, 26]. Those techniques introduce the same predicates. Moreover, unlike our technique, they will keep generating *spurious* counter-examples. For this example no may/must abstraction based on predicates and refinement with weakest precondition calculations can produce a structure that is bisimilar to the concrete system (the concrete system is rather trivial — it has only one state).

This example also illustrates another difference between the method presented here and over-approximation based predicate abstraction with refinement, in particular [26]. If the analyzed system has a *reachable* finite bisimulation quotient then our algorithm is guaranteed to find it (see Theorem 2 and Lemma 8). In contrast, the method in [26] will fail to compute a finite state abstraction for the example; this result seems to contradict the bisimulation completeness claim (Theorem 3) from [26]. We conjecture that the method in [26] is not guaranteed to compute a finite state abstraction unless both the *reachable and unreachable* quotient is finite.

To solve the problem of non-termination for finite state systems, we propose to use the following heuristic. If there is a transition for which we cannot prove that the abstraction is exact in several subsequent iterations of the algorithm, then we add predicates describing the concrete state; i.e. in the example from Figure 4 we would add predicates $x = 0$ and $y = 0$. The abstraction eventually becomes exact with respect to each transition. And since the number of reachable transitions is finite, the algorithm must terminate.

Corollary 1. If the reachable part of $\llbracket M \rrbracket$ is finite state then the modified algorithm terminates.

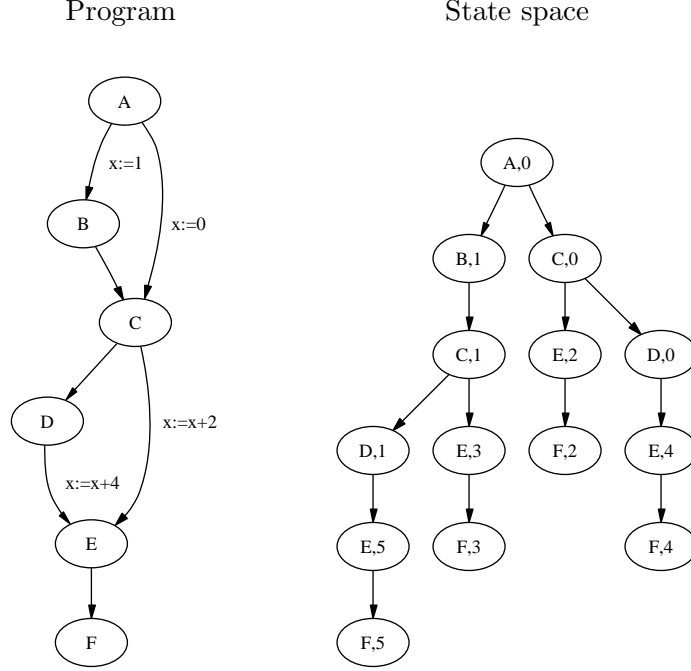


Figure 5: Example illustrating non-monotonic refinement

6.2. Search Order and Non-Monotonicity. The search order used in α SEARCH (depth-first or breadth-first) influences the size of the generated structure, the newly computed predicates, and even the number of iterations of the main algorithm. If there are two states s_1 and s_2 such that $\alpha_\Phi(s_1) = \alpha_\Phi(s_2)$ but $s_1 \not\sim s_2$ then, depending on whether s_1 or s_2 is visited first, different parts of the transition system will be explored. For our implementation, we use breadth-first search order.

Also note that the refinement algorithm is non-monotone, i.e. a labeling which is reachable in one iteration may not be reachable in the next iteration. However, the algorithm is guaranteed to converge to the correct answer. The example in Figure 5 illustrates this non-monotonic behavior. Figure 5 (left) shows the transitions of the example program (for clarity of presentation, we depict the program in a graphical notation); the program has only one variable x ; the program counter ranges over $A, B, C \dots$. Figure 5 (right) shows the whole concrete state space of the program. As usual, states are labeled by the program counter and the concrete value of program variable x . Let us consider the first iteration of the algorithm, with abstraction predicate $x \geq 3$ and with breadth-first search order – the following states are visited: $(A, 0), (B, 1), (C, 0), (E, 2), (D, 0), (F, 2), (E, 4), (F, 4)$. Assume now that the refinement step adds a new predicate $x = 1$; then, in the second iteration, the following states are visited: $(A, 0), (B, 1), (C, 0), (C, 1), (E, 2), (D, 0), (D, 1), (E, 3), (F, 2), (F, 3)$. States $(E, 4)$ and $(F, 4)$ are visited during the first iteration and they are not visited during the second one.

6.3. Relation to Other Abstractions. We discuss now the relationship between our abstraction based iterative algorithm and other (under-approximating) abstractions, in particular with the *must* abstractions from [30, 31] and with the abstractions induced by the

refined definition of *must* transitions presented in [3]. We first remark that the abstract state space explored by our approach is (potentially) a better approximation than the *must* abstraction. This is formulated by the following lemma.

Lemma 9. Let $AP \subseteq \Phi$. Then $RL(\alpha\text{SEARCH}(M, \Phi))$ is a superset of the reachable labelings in the *must* abstraction induced by Φ .

Proof: The lemma is a direct consequence of Lemma 1. \square

As mentioned, the iterative refinement in our algorithm is non-monotonic. A similar problem occurs in the context of *must* abstractions: the set of *must* transitions is not generally monotonically non-decreasing when predicates are added to refine an abstract system [16, 31]. This problem is addressed in [30, 31], by creating *hyper must* transitions (representing sets of *must* transitions). Note that the approaches presented in [3, 30, 31] require the a-priori construction of abstract *must* (and *hyper must*) transitions and therefore make an exponential number of theorem prover calls. In contrast our approach does not require the computation of abstract transitions, since it executes directly the concrete transitions (and it only makes theorem prover calls during refinement).

Recently, Ball et al. [3] defined an extension of the *must* abstraction based on so called *must*⁻ transitions: $a_1 \xrightarrow{i}_{\text{must}^-} a_2$ iff for all concrete states s_2 such that $\alpha_\Phi(s_2) = a_2$, there exists concrete state s_1 such that $\alpha_\Phi(s_1) = a_1$ and $s_1 \xrightarrow{i} s_2$ (for some Φ).

They call the classical *must* transitions *must*⁺ transitions and they describe a reachability analysis that uses both *must*⁻ and *must*⁺ transitions; the set of reachable labelings is defined as $\{L(s) \mid s \in S : s_0 \xrightarrow{*}_{\text{must}^-} s_i \xrightarrow{*}_{\text{must}^+} s\}$. This results in an under-approximation of the set $RL(\llbracket M \rrbracket)$ and at the same time it is a better under-approximation than the one obtained by classical *must* transitions.

Here we show that under-approximations based on *must*⁺/*must*⁻ transitions and our algorithm based on αSEARCH are incomparable. The (trivial) example in Figure 6 (a) illustrates that αSEARCH can be more precise than the analysis based on *must*⁺/*must*⁻ transitions. If we consider the abstraction with respect to a single predicate $x \geq 0$ we see that the program transition is neither *must*⁺ nor *must*⁻ (hence the set of reachable labelings produced by the analysis from [3] contains only a labeling $x \geq 0$) whereas αSEARCH executes the transition and finds a labeling $x < 0$.

On the other hand, consider the example in Figure 6 (b) and an abstraction with respect to a single predicate $x \geq 3$. Due to state matching on the states represented by $(pc = 1, x = 1)$ and $(pc = 1, x = 2)$, αSEARCH computes a different set of labelings, depending on which of the first two transitions is traversed first from the initial state. Therefore, the resulting set of reachable labelings contains only one of $(pc = 3, x < 3)$, $(pc = 3, x \geq 3)$. Under-approximation based on *must*⁺/*must*⁻ transitions contains both of these labelings.

7. EXTENSIONS

In this section we propose several extensions of the main algorithm.

7.1. Open Systems. Until now, we have discussed our approach in the context of “closed” systems. However, the approach can be extended to handling “open” systems (i.e. programs with inputs). In order to model open systems, we extend the guarded commands language

$$\begin{array}{ll}
\text{(a)} & x \geq 0 \longmapsto x := x - 1 \\
\\
& pc = 0 \longmapsto pc := 1, x := 1 \\
& pc = 0 \longmapsto pc := 1, x := 2 \\
\text{(b)} & pc = 1 \longmapsto pc := 2, x := x + 1 \\
& pc = 2 \wedge x \geq 3 \longmapsto pc := 3 \\
& pc = 2 \wedge x < 3 \longmapsto pc := 3
\end{array}$$

Figure 6: Examples showing that under-approximations based on αSEARCH and $must^+/must^-$ transitions are incomparable

by allowing assignments of the form $x := \text{input}$, which assigns to program variable x an arbitrary value from the input domain (in our case the set of integers). We can also allow the initial values of the program variables to be unspecified, in which case the transition system representing the open program has several (possibly unbounded) initial states.

In order to apply our approach, we need to compute, for each input variable, explicit concrete values that drive the concrete execution of the program. What we really want here is to pick one input value for each satisfiable valuation of the abstraction predicates. We can directly use the original algorithm — it will simply try all the possible values and continue the program execution only from values that satisfy the predicate combinations (most of the states that contain such input values will be matched if they lead to the same valuation of abstraction predicates). This “brute force” approach requires enumerating eventually the whole input domain, which is impossible for infinite input domains. Note however that the approach might still be very useful at detecting errors.

Alternatively, we can use a constraint solver for computing the input values that are solutions of the satisfiable combinations of abstraction predicates (provided that satisfiability is decidable for the abstraction predicates). The decision whether to use the “brute force” approach or the satisfiability approach depends on the number of abstraction predicates and the size of the input domain. With the brute force approach, the the whole input domain needs to be enumerated eventually. With the satisfiability approach, there are at most 2^k satisfiability queries (where k is number of predicates which depend on the input variable).

7.2. Transition Dependent Predicates. The predicates that are generated after the validity check for one transition are used ‘globally’ at the next iteration. This may cause unnecessary refinement — the new predicates may distinguish states which do not need to be distinguished. To avoid this, we could use ‘transition dependent’ predicates. The idea is to associate the abstraction predicates with the program counter corresponding to the transition that generated them. New predicates are then added only to the set of the respective program counter. However, with this approach, it may take longer before predicates are ‘propagated’ to all the locations where they are needed, i.e. more iterations are needed before an error is detected or an exact abstraction is found. We need to further investigate these issues. Similar ideas are presented in [8, 21], in the context of over-approximation based predicate abstraction.

7.3. Light-weight Approach. As mentioned, the under-approximation and refinement approach can be used in a lightweight but systematic manner, without using a theorem prover for validity checking. Specifically, for each explored transition t_i refinement adds the new predicates from $\alpha_\Phi(s')[e_i(\vec{x})/\vec{x}]$, regardless of the fact that the abstraction is exact with respect to transition t_i . This approach may result in unnecessary refinement. A similar refinement procedure was used in [26] for over-approximation predicate abstraction.

We are also considering several heuristics for generating new abstraction predicates. For example, it is customary to add the predicates that appear in the guards and in the property to be checked. One could also add predicates generated dynamically, using tools like Daikon [13], or predicates from known invariants of the system, generated using static analysis techniques. Section 8 shows an example where a statically computed invariant helped with the termination of the presented iterative algorithm.

In order to extend the applicability of the proposed technique to the analysis of full-fledged programming languages, we are investigating abstractions that record information about the shape of the program heap, to be used in conjunction with the abstraction predicates. We have reported about these experiments in [34].

8. IMPLEMENTATION AND APPLICATIONS

We have implemented our approach for the guarded command language. Our implementation is done in the language Ocaml¹ and it uses the Simplify theorem prover [12]. The implementation has just 590 lines of code (parsing + definition of semantics: 390 lines, α SEARCH algorithm: 170 lines, REFINEMENTSEARCH algorithm: 30 lines). The implementation uses several optimizations for reducing the number of theorem prover calls:

- When updating Φ_{new} for refinement, we add only those conjuncts of $\alpha_\Phi(s')[e_i(\vec{x})/\vec{x}]$ for which we cannot prove validity.
- We cache queries to ensure that Simplify is not called twice for the same query.
- All queries have the form of implication. Before calling the theorem prover for the implication, we check whether the right hand side is a tautology (in such case the implication is clearly satisfied). The results of these checks are also cached.

8.1. Experiments. We discuss the application of our implementation for error detection and property verification in several multi process programs. The examples are: the ticket mutual exclusion protocol, RAX (Remote Agent Experiment), a component extracted from an embedded spacecraft-control application, and the bakery mutual exclusion protocol. We also analyzed a single process device driver taken from [5], which is a “classic” example analyzed with predicate abstraction techniques. We analyzed defective and correct versions of each example program. The RAX and device driver had known errors that we checked for. For the other examples, we seeded faults to obtain the defective versions.

Note that in the described experiments, we always start the first iteration of the refinement algorithm with the program predicates which occur in guards. All the reported results are for the breadth-first search order.

Table 1 summarizes the results for each of the runs of our algorithm. The first part of the table reports the analysis results for the defective examples (denoted with the **-err** suffix), while the second part of the table reports the results for the correct examples. For

¹<http://caml.inria.fr/>

Example	Iterations	Concrete States (per iteration)	Abstract States (per iteration)	New Predicates	Queries
ticket2-err	2	15, 31	9, 17	5	38
ticket3-err	1	102	44	4	14
RAX-err	1	69	44	0	10
bakery-err	1	356	191	14	89
driver-err	1	10	10	0	2
ticket2	4	15, 15, 15, 15	9, 9, 9, 9	6	124
ticket3	5	52, 58, 58, 58, 58	25, 31, 31, 31, 31	11	603
RAX	–	–	–	–	–
bakery	3	278, 410, 537	152, 221, 292	24	1598
driver	2	10	9	0	7

Table 1: Experimental results

each example we report numbers for: refinement iterations, generated concrete states and stored abstract states, generated predicates, and queries to the theorem prover. A “–” for RAX denotes that our analysis did not finish for this example (see discussion below). Note that for the concrete and abstract states, we report *separate* numbers *for each iteration*. For example, running our tool on the error version of the ticket protocol with two processes (**ticket2-err**) discovered the error after 2 iterations; in the first iteration, the tool generated 15 concrete states and it stored 9 abstract states, while in the second iteration, it generated 31 concrete states and it stored 17 abstract states. We discuss the experiments in more detail below (full details are available at [27]).

8.2. Ticket Protocol. This is a protocol for mutual exclusion [1]; we use the formalization of the algorithm from [6]. The algorithm is based on a simple “ticket” procedure: a process which wants to enter the critical section draws a ticket number that is one larger than the number held by any other process. The process then waits until all processes with smaller numbers are served: this is checked by a “display” variable which shows the value of the ticket number which is currently the smallest. The model of the protocol is given in Figure 7. The property of interest is mutual exclusion in critical section ($\neg(pc_1 = 2 \wedge pc_2 = 2 \vee pc_2 = 2 \wedge pc_3 = 2 \vee pc_1 = 2 \wedge pc_3 = 2)$). The state space is infinite (the ticket numbers increase without any bound), but it has a finite bisimulation quotient.

We used our tool to prove successfully that the property holds. We analyzed several versions of the protocol. The intermediate analysis results for the protocol with three processes are given in Table 2. We report the following results for each iteration of the refinement algorithm: the number of generated concrete states, the number of stored abstract states, the number of queries to the theorem prover, the number of hits to a queries cache, and the newly generated predicates.

As discussed, we also seeded an error in the protocol and used our tool for error detection. The error was seeded by changing the assignment $s := s + 1$ into $s := s + 2$. For an instance with two processes the error is found after two iterations. For an instance with three processes the error state can be reached by suitable interleaving in the first round of the protocol and the tool finds the error in the first iteration.

$$\begin{array}{ll}
pc_1 = 0 & \longmapsto pc_1 := 1, a_1 := t, t := t + 1 \\
pc_1 = 1 \wedge a_1 \leq s & \longmapsto pc_1 := 2 \\
pc_1 = 2 & \longmapsto pc_1 := 0, s := s + 1 \\
\\
pc_2 = 0 & \longmapsto pc_2 := 1, a_2 := t, t := t + 1 \\
pc_2 = 1 \wedge a_2 \leq s & \longmapsto pc_2 := 2 \\
pc_2 = 2 & \longmapsto pc_2 := 0, s := s + 1 \\
\\
pc_3 = 0 & \longmapsto pc_3 := 1, a_3 := t, t := t + 1 \\
pc_3 = 1 \wedge a_3 \leq s & \longmapsto pc_3 := 2 \\
pc_3 = 2 & \longmapsto pc_3 := 0, s := s + 1
\end{array}$$

Figure 7: Ticket protocol (instance for three processes)

Iteration	Concrete states	Abstract states	Num. queries	Cache hits	New predicates
1	52	25	14	18	$a_1 \leq s + 1, a_2 \leq s + 1, a_3 \leq s + 1,$ $t \leq s$
2	58	31	70	152	$a_1 \leq s + 2, a_2 \leq s + 2, a_3 \leq s + 2,$ $t \leq s + 1, t + 1 \leq s$
3	58	31	151	475	$t \leq s + 2$
4	58	31	173	585	$t \leq s + 3$
5	58	31	195	657	-

Table 2: Ticket protocol for three processes: intermediate results

8.3. RAX. The RAX example (illustrated in Figure 8) is derived from the software used in the NASA Deep Space 1 Remote Agent experiment, which deadlocked during flight [33]. We encoded the deadlock check as “ $pc_1 = 4 \wedge pc_2 = 5 \wedge w_1 = 1 \wedge w_2 = 1$ is unreachable”. The error is found after one iteration; the reported counter-example has 8 steps.

Note that the state space of the program is unbounded, as the program keeps incrementing the counters e_1 and e_2 , when $pc_2 = 2$ and $pc_1 = 6$, respectively. We also ran our algorithm to see if it converges to a finite bisimulation quotient. Interestingly, the algorithm does not terminate for the RAX example, although it has a finite reachable bisimulation quotient. The results are shown in Table 3. However, if we assume that the counters in the program are non-negative, i.e. we introduce two new predicates, $e_1 \geq 0$, $e_2 \geq 0$ (which can be easily discovered using static analysis), then the algorithm terminates after two iterations. The tool reports the following results : 69 concrete and 44 abstract states explored in the first iteration, 101 concrete and 65 abstract states in the second iteration, two new predicates and 40 queries.

8.4. Bakery Protocol. This is another well-known protocol for mutual exclusion. The protocol is similar to the ticket protocol (the ticket protocol requires special hardware instruction like Fetch-and-Add, whereas the bakery protocol is applicable without any special instructions). The model has 10 variables. The property of interest is again mutual exclusion. The state space is infinite with a finite bisimulation quotient. The property can be proved by the algorithm in three iterations, using 31 predicates. For this example, we

$pc_1 = 1$	\mapsto	$c_1 := 0, pc_1 := 2$
$pc_1 = 2 \wedge c_1 = e_1$	\mapsto	$pc_1 := 3$
$pc_1 = 3$	\mapsto	$w_1 := 1, pc_1 := 4$
$pc_1 = 4 \wedge w_1 = 0$	\mapsto	$pc_1 := 5$
$pc_1 = 2 \wedge c_1 \neq e_1$	\mapsto	$pc_1 := 5$
$pc_1 = 5$	\mapsto	$c_1 := e_1, pc_1 := 6$
$pc_1 = 6$	\mapsto	$e_2 := e_2 + 1, w_2 := 0, pc_1 := 2$
$pc_2 = 1$	\mapsto	$c_2 := 0, pc_2 := 2$
$pc_2 = 2$	\mapsto	$e_1 := e_1 + 1, w_1 := 0, pc_2 := 3$
$pc_2 = 3 \wedge c_2 = e_2$	\mapsto	$pc_2 := 4$
$pc_2 = 4$	\mapsto	$w_2 := 1, pc_2 := 5$
$pc_2 = 5 \wedge w_2 = 0$	\mapsto	$pc_2 := 6$
$pc_2 = 3 \wedge c_2 \neq e_2$	\mapsto	$pc_2 := 6$
$pc_2 = 6$	\mapsto	$c_2 := e_2, pc_2 := 2$

Figure 8: RAX example

Iteration	Concrete states	Abstract states	Num. queries	Cache hits	New predicates
1	69	44	10	10	$e_1 = 0, e_2 = 0$
2	101	65	20	44	$e_1 = -1, e_2 = -1$
3	101	65	26	64	$e_1 = -2, e_2 = -2$
4	101	65	32	84	...

Table 3: RAX example: intermediate results

seeded an error by changing a guard $num_1 < num_0$ into $num_1 > num_0$ which creates a nontrivial error in the protocol. The tool can find the error in the first iteration.

8.5. Device Driver. This is a “classic” example analyzed using predicate abstraction [5]. The property of interest is the correct use of a lock. Our tool can prove that the property holds after one iteration (using just the predicates from guards): the algorithm explores 10 concrete states, 9 abstract states and casts 3 queries to the theorem prover. For an erroneous version of the driver, the tool finds an error in the first iteration as well.

8.6. Discussion. These preliminary experiments show the merits of our approach. The approach proves to be effective in computing finite bisimilar structures of non-trivial infinite state systems and in finding errors using under-approximation based predicate abstraction. Of course, much more experimentation is necessary to really assess the practical benefits of the proposed technique and a lot more engineering is required to apply it to real programming languages. Extensions for handling complex features such as pointers, arrays and procedures, are tedious but conceptually not very hard.

We also note that in some cases (e.g. `ticket2`, `ticket3` and `RAX`) the number of explored concrete and abstract states stays the same after the first iteration; however our

algorithm needs more than two iterations to discover all the necessary abstraction predicates, according to the exactness criteria that we defined. The results suggest that it is possible to relax these criteria and still provide a guarantee that the relevant state space of the analyzed program has been explored. We leave this topic for future work.

8.7. Comparison and Combination with Over-approximation Based Approaches.

We should mention that the application of over-approximation based predicate abstraction to a Java version of RAX is described in detail in [33]. In that work, four different predicates were used to produce an abstract model that is bisimilar to the original program. In contrast, the work presented here allowed more aggressive abstraction to recover feasible counter-examples. Our technique explores transitions that are guaranteed to be feasible. In contrast, the over-approximation based techniques such as the ones from [4, 7, 22] may also explore transitions that are spurious and therefore could require additional refinement before reporting a real counter-example.

As mentioned, over-approximation based abstraction techniques involve exponentially many theorem prover queries (in the number of predicates), at each iteration. This computation is performed regardless of the size of (the reachable portion of) the analyzed system. In our case, theorem prover queries are only performed during refinement and they involve only the *reachable* state space of the system under analysis. On the other hand, over-approximation based techniques are good at proving properties (as they compute abstractions that are coarser than the bisimulation quotient but sufficient to prove safety properties). We believe however that the technique presented here is *complementary* to over-approximation abstractions and it should *combined* (rather than compared) with such techniques. Our technique could be used for discovering efficiently feasible counter-examples in the space bounded by the abstraction predicates (that are used in the over-approximation analysis). In the future, we plan to study more the strengths and weaknesses of each approach and to investigate their *integration*.

9. RELATED WORK

Throughout the paper, we have already discussed the relationship between our work and predicate abstraction (see the previous section and also Section 6, where we compared our work with over-approximation approaches, in particular the work of Namjoshi and Kurshan [26], and with under-approximation approaches using *must* transitions [3, 30, 31]). We discuss here other approaches that are closely related to ours.

The work of Grumberg et al. [20] uses a refinement of an under-approximation to improve analysis of multi-process systems. The procedure in [20] checks models with an increasing set of allowed interleavings of the given processes, starting from a single interleaving. It uses SAT-based bounded model checking for analysis and refinement, whereas here we focus on explicit model checking and predicate abstraction, and we use weakest precondition calculations for abstraction refinement.

Another closely related work is that of Lee and Yannakakis [24], which proposes an on-the-fly algorithm for computing the bisimulation quotient of an (infinite state) transition system. Similar to our approach, the algorithm from [24] traverses concrete transitions while computing *blocks* of equivalent states; if some transition is found to be *unstable* the block is *split* into sub-blocks. Note however that unlike [24] our algorithm is geared towards error detection and it is formulated in terms of predicate abstraction with a clear

separation between state exploration and refinement. There are other important differences between our approach and the work presented in [24]. We use refinement globally while the block splitting in [24] is local. This makes the approach in [24] more efficient in the number of visited states. On the other hand, the global refinement has the advantage of faster propagating the new predicates across the system but it may lead to unnecessary refinement. As a consequence of this global refinement, our algorithm may not compute *the* bisimulation quotient (as in [24]) but rather just *a* bisimilar structure (due to extra refinement). We view the experimental comparison of the two approaches as an interesting topic for future work.

In previous work [28], we developed a technique for finding feasible counter-examples in abstracted programs. The technique essentially explores an under-approximation defined by the *must* abstract transitions (although the presentation is not formalized in these terms). The work presented here explores an under-approximation which is more precise than the abstract system defined by the *must* transitions. Hence it has a better chance of finding bugs while enabling more aggressive abstraction and therefore more state space reduction.

Model-driven software verification [23] advocates the use of abstraction mappings during concrete model checking in a way similar to what we present here. In their approach, the abstraction function needs to be provided by the user. The CMC model checking tool [25] also attempts to store state information in memory using aggressive compressing techniques (which can be seen as a form of abstraction), while the detailed state information is kept on the stack. These techniques allow the detection of subtle bugs which can not be discovered by classical model checking, using e.g. breadth first search or by state-less model checking [15]. While these techniques use abstractions in an ad-hoc manner, our work contributes the automated generation and refinement of abstractions.

Directed automated random testing (DART) [17] performs a concrete execution on random inputs and it collects the *path constraints* along the executed paths. These path constraints are then used to compute new inputs that drive the program along alternative paths. The approach in [17] is similar to ours as it combines concrete program execution with a symbolic analysis. However, DART applies only to sequential programs, not to concurrent programs as we do here. Moreover, DART attempts to cover all the feasible paths through the program, not the reachable (abstract) states as we do in our approach. DART does not perform any state matching, and therefore it can not detect if an (abstract) state has been visited before. As a result, DART can potentially explore redundant states, e.g. for looping, reactive, programs. Another (methodological) difference is that DART uses symbolic evaluation while our method uses predicate abstraction with refinement.

Dataflow and type-based analyzers have been used to check safety properties of software (e.g. [32]). Unlike our work, these techniques analyze over-approximations of system behavior and may generate false reports due to infeasible paths.

10. CONCLUSIONS AND FUTURE WORK

We presented a model checking algorithm based on refinement of under-approximations, which effectively preserves the defect detection ability of model checking in the presence of powerful abstractions. The under-approximation is obtained by traversing the concrete transition system and performing the state matching on abstract states computed by predicate abstraction. The refinement is done by checking exactness of abstractions with the use of a theorem prover. We illustrated the application of the algorithm for checking safety

properties of concurrent programs. In the future, we plan to investigate whether we can extend the algorithm with property driven refinement and with checking liveness properties. We also plan to investigate the integration of our approach with over-approximation based abstraction refinement and to do an extensive evaluation on large systems.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their detailed comments that helped us to improve this article significantly.

REFERENCES

- [1] G. R. Andrews. *Concurrent Programming, Principles and Practice*. Addison-Wesley Publishing Company, 1991.
- [2] T. Ball. A theory of predicate-complete test coverage and generation. In *Proceedings of Formal Methods for Components and Objects (FMCO 2004)*, volume 3188 of *LNCS*. Springer, 2004.
- [3] T. Ball, O. Kupferman, and G. Yorsh. Abstraction for falsification. In *Proceedings of Computer Aided Verification (CAV 2005)*, volume 3576 of *LNCS*, pages 52–66. Springer, 2005.
- [4] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstraction for model checking c programs. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*, pages 268–283. Springer, 2001.
- [5] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of SPIN workshop*, volume 2057 of *LNCS*, pages 103–122. Springer, 2001.
- [6] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using presburger arithmetic. In *Proceedings of Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 400–411. Springer, 1997.
- [7] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *ACM Trans. Computer Systems*, 30(6):388–402, 2004.
- [8] S. Chaki, E. Clarke, A. Groce, and O. Strichman. Predicate abstraction with minimum predicates. In *Proceedings of the 12th CHARME*, volume 2860 of *LNCS*, 2003.
- [9] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 4(2):511–547, August 1992.
- [10] D. Dams and K. S. Namjoshi. The existence of finite abstractions for branching time model checking. In *Proceedings of Logic in Computer Science (LICS 2004)*, pages 335–344. IEEE Computer Society, 2004.
- [11] L. de Alfaro, P. Godefroid, and R. Jagadeesan. Three-valued abstractions of games: Uncertainty, but with precision. In *Proceedings of Logic in Computer Science (LICS 2004)*, pages 170–179. IEEE Computer Society, 2004.
- [12] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. *Research Report 159, Compaq Systems Research Center*, 1998.
- [13] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, 2000.
- [14] R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples and refinements in abstract model checking. In *Proceedings of the 8th Static Analysis Symposium (SAS'01)*, volume 2126 of *LNCS*, 2001.
- [15] P. Godefroid. Software Model Checking: the Verisoft Approach. *Formal Methods in Systems Design*, 26(2):77–101, 2005.
- [16] P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based model checking using modal transition systems. In *Proceedings of Conference on Concurrency Theory (CONCUR '01)*, pages 426–440. Springer-Verlag, 2001.
- [17] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223. ACM, 2005.
- [18] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proceedings of Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, 1997.

- [19] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA'04)*, July 2002.
- [20] O. Grumberg, F. Lerda, O. Strichman, and M. Theobald. Proof-guided underapproximation-widening for multi-process systems. In *Proceedings of the 32nd Symposium on Principles of Programming Languages (POPL'05)*, 2005.
- [21] T. A. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstractions from proofs. In *Proceedings of the 31st Symposium on Principles of Programming Languages (POPL'04)*, 2004.
- [22] T. A. Henzinger, R. Jhala, R. Majumdar, and Gregoire Sutre. Lazy abstraction. In *Proceedings of the 29th Symposium on Principles of Programming Languages*, 2002.
- [23] G. J. Holzmann and R. Joshi. Model-driven software verification. In *Proceedings of the 11th SPIN Workshop*, volume 2989 of *LNCS*, Barcelona, Spain, 2004.
- [24] D. Lee and M. Yannakakis. Online minimization of transition systems. In *Proceedings of the 24th ACM Symposium on Theory of Computing*, 1992.
- [25] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, 2002.
- [26] K. S. Namjoshi and R. P. Kurshan. Syntactic program transformations for automatic abstraction. In *Proceedings of Computer Aided Verification (CAV 2000)*, pages 435–449. Springer-Verlag, 2000.
- [27] R. Pelánek. *Reduction and Abstraction Techniques for Model Checking*. PhD thesis, Masaryk University Brno, 2006.
- [28] C. S. Păsăreanu, M. B. Dwyer, and W. Visser. Finding feasible abstract counter-examples. *STTT*, 5(1):34–48, 2003.
- [29] C. S. Păsăreanu, R. Pelánek, and W. Visser. Concrete search with abstract matching and refinement. In *Proceedings of Computer Aided Verification (CAV 2005)*, volume 3576 of *LNCS*, pages 52–66. Springer, 2005.
- [30] S. Shoham and O. Grumberg. A game-based framework for ctl counterexamples and 3-valued abstraction-refinement. In *Proceedings of Computer Aided Verification (CAV 2003)*, volume 2725 of *LNCS*, pages 275–287. Springer, 2003.
- [31] S. Shoham and O. Grumberg. Monotonic abstraction-refinement for CTL. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *LNCS*, pages 546–560. Springer, 2004.
- [32] A. Venet and G. Brat. Precise and efficient static array bound checking for large embedded C programs. In *Proceedings of Programming Language Design and Implementation (PLDI'04)*, 2004.
- [33] W. Visser, S. Park, and J. Penix. Applying predicate abstraction to model check object-oriented programs. In *Proceedings of the 3rd ACM SIGSOFT Workshop on Formal Methods in Software Practice*, 2000.
- [34] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test input generation for red black trees using abstraction. In *Proceedings of Automated Software Engineering (ASE'05)*, pages 414–417. ACM, 2005.
- [35] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proceedings of the 19th Automated Software Engineering*, 2004.