Concrete Model Checking with Abstract Matching and Refinement

Corina S. Păsăreanu[1], Radek Pelánek*[2], and Willem Visser[3]

[1] Kestrel Technology/QSS, NASA Ames, Moffett Field, CA 94035, USA
[2] Masaryk University Brno, Czech Republic
[3] RIACS/USRA, NASA Ames, Moffett Field, CA 94035, USA

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. We also show how a lightweight variant can be used for efficient software testing.

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 [13] 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 [2, 3, 17]. Furthermore, a strength of model checking is its ability to automate 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.

* Partially supported by the Grant Agency of Czech Republic grant No. 201/03/0509 and by the Academy of Sciences of Czech Republic grant No. 1ET408050503.

The technique uses a novel 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 [19] 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 [21] 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 [14, 28], by adding support for automated abstraction refinement.

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 and for testing container implementations.

Comparison with Related Work The most closely related work to ours is that of Grumberg et al. [15] where a refinement of an under-approximation is used to improve analysis of multi-process systems. The procedure in [15] 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.

Our approach can be contrasted with the work on predicate abstraction for modal transition systems [12, 24], 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 [1, 6, 7]). The method 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 (see Section 2).

Moreover, unlike [12, 24] and over-approximation based abstraction techniques [2, 3], 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 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.

In previous work [22], we developed a technique for finding guaranteed 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 [18] advocates the use of abstraction mappings during concrete model checking in a way similar to what we present here. The CMC model checking tool [20] 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 [11]. While these techniques use abstractions in an ad-hoc manner, our work contributes the automated generation and refinement of abstractions.

Dataflow and type-based analyzes have been used to check safety properties of software (e.g. [25]). Unlike our work, these techniques analyze over-approximations of system behavior and may generate false positive results due to infeasible paths.

Layout 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 for the algorithm. Section 6 proposes extensions to the main algorithm. Section 7 illustrates applications of the approach and Section 8 concludes the paper.

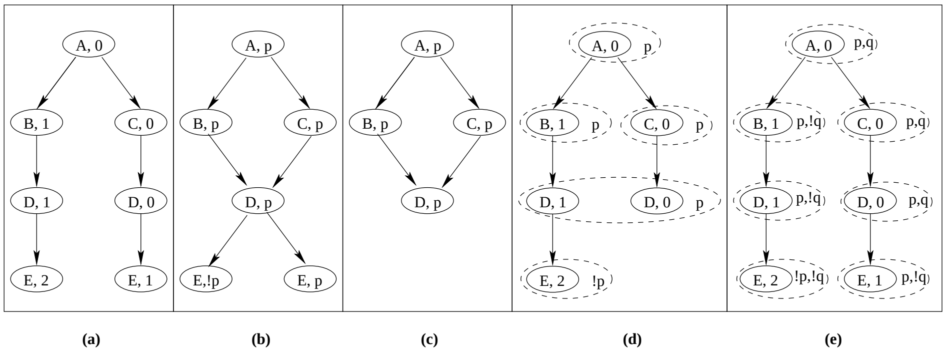

Fig. 1. **(a)** Concrete system **(b)** *May* abstraction using predicate $p = x < 2$ **(c)** *Must* abstraction using p **(d)** Concrete search with abstract matching using p **(e)** Concrete search with abstract matching using predicates p and $q = x < 1$.

2 Example

The example in Fig. 1 illustrates some of the main characteristics of our approach. Fig. 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 ...) and the concrete value of x. Fig. 1 **(b)** shows the abstract system induced by the *may* transitions for predicate $p = x < 2$. Fig. 1 **(c)** shows the abstract system induced by the *must* transitions for predicate p.

Fig. 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 than the *must* abstraction. Fig. 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

Program Model 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 $(+, -, \cdot, ...)$ operations. A *model* is a tuple $M = (V, T)$. $T = \{t_1, \ldots, t_k\}$ is a finite set of transitions, where $t_i = (g_i(\boldsymbol{x}) \longmapsto \boldsymbol{x} := e_i(\boldsymbol{x}))$. $g_i(\boldsymbol{x})$ is a guard and $e_i(\boldsymbol{x})$ are assignments to the variables represented by tuple $\boldsymbol{x}$; throughout the paper, we write this as a sequence of assignments.

Semantics As a semantics of a model we use transition systems. 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 $L : S \to 2^{AP}$ is a labelling function. State s is *reachable* if it is reachable from the initial state via zero or more transitions, i.e. $s_0 \to^* s$. The set of *reachable labellings* RL is $\{L(s) \mid \exists s \in S : s_0 \to^* s\}$. The *concrete semantics* of model M is the transition system $C(M) = (S, \{\xrightarrow{i}\}, s_0, L)$ over AP, where:

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

Weakest precondition The *weakest precondition* of a set of states X with respect to transition i is $wp(X, i) = \{s \mid s \xrightarrow{i} s' \Rightarrow s' \in X\}$. If the set of states X is characterized by a predicate ϕ, then the weakest precondition with respect to transition i can be expressed as $wp(\phi, i) = (g_i \Rightarrow \phi[e_i(\boldsymbol{x})/\boldsymbol{x}])$.

Predicate abstraction Predicate abstraction is a special instance of the framework of abstract interpretation [5] that maps a (potentially infinite state) transition system into a finite state transition system via a set of predicates $\Phi = \{\phi_1, \ldots, \phi_n\}$ over the program variables. Let $\mathbb{B}_n$ be a set of bitvectors of length n. We define abstraction function $\alpha_\Phi : S \to \mathbb{B}_n$, such that $\alpha_\Phi(s)$ is a bitvector $b_1 b_2 \ldots b_n$ 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 $\alpha_\Phi(s)$ (or just $\alpha(s)$) 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:

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

Algorithms for computing abstractions using over-approximation based predicate abstraction are given in e.g. [2, 13] (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. Note that our approach does not require the computation of abstract transitions, since it executes the concrete transitions directly.

Bisimulation 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

Algorithm Fig. 2 shows the reachability procedure that performs model checking with abstract matching (αSearch). 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 $\alpha(s)$ 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.

Fig. 3 gives the iterative refinement algorithm for checking whether M can reach an error state described by φ. 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).

Checking for Exact Abstraction and Refinement We say that an abstraction function α is *exact* with respect to transition $s \xrightarrow{i} s'$ iff for all s_1 such that $\alpha(s) = \alpha(s_1)$ there exists s'_1 such that $\alpha(s'_1) = \alpha(s')$ and $s_1 \xrightarrow{i} s'_1$. In other words, α is *exact* with respect to $s \xrightarrow{i} s'$ iff $\alpha(s) \xrightarrow{i}_{must} \alpha(s')$. This definition is also related to the notion of *completeness* in abstract interpretation (see e.g. [10]), which states that no loss of precision is introduced by the abstraction.

Checking that the abstraction is *exact* with respect to concrete transition $s \xrightarrow{i} s'$ is equivalent to checking that $\alpha_\Phi(s) \Rightarrow wp(\alpha_\Phi(s'), i)$ is valid. This

```
proc αSEARCH($M, \Phi$)
  $\Phi_{new} = \Phi$; 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 $\Phi_{new}$ fi
                    $s' = u_i(s)$
                    if $\alpha_\Phi(s) \Rightarrow \alpha_\Phi(s')[e_i(\boldsymbol{x})/\boldsymbol{x}]$ is not valid
                      then add predicates in $\alpha_\Phi(s')[e_i(\boldsymbol{x})/\boldsymbol{x}]$ to $\Phi_{new}$ fi
                    if $\alpha_\Phi(s') \notin States$ 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 $\Phi_{new}$ fi
          fi
        od
  od
  $A = (States, Transitions, \alpha_\Phi(s_0), L)$
  return $(A, \Phi_{new})$
end
```

Fig. 2. Search procedure with checking for exact abstraction

formula is equivalent to $\alpha_\Phi(s) \Rightarrow \alpha_\Phi(s')[e_i(\boldsymbol{x})/\boldsymbol{x}]$ when $s \models g_i$. 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 properties of the refinement algorithm. We state only the main theorems, technical lemmas and proofs are given in [23] (due to space limitations). First, we show that the set $RL(\alpha\text{SEARCH}(M, \Phi))$ of reachable labellings computed by the algorithm REFINEMENTSEARCH is a subset of the reachable labellings of the system under analysis. 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}).

```
proc RefinementSearch(M, φ)
  i = 1; Φ_i = ∅
  while true do
        (A_i, Φ_{i+1}) = αSearch(M, Φ_i)
        if φ is reachable in A_i then return counter-example fi
        if Φ_{i+1} = Φ_i then return unreachable fi
        i = i + 1
  od
end
```

Fig. 3. Iterative refinement algorithm

Theorem 1. *Let $AP \subseteq \Phi$. Then $RL(\alpha\textsc{Search}(M, \Phi)) \subseteq RL(C(M))$.*

Moreover, it holds that $RL(\alpha\textsc{Search}(M, \Phi))$ is a superset of the reachable labellings in the *must* abstraction (see Lemma 1 in [23]), hence it is (potentially) a better approximation.

We now 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 2. *If* $\textsc{RefinementSearch}(M, \varphi)$ *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 $C(M)$.*

In general, the proposed algorithm might not terminate (because of the halting problem). However, the algorithm is guaranteed to eventually find all the reachable labellings 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 will eventually produce a finite bisimilar structure.

Theorem 3. *Let the $\alpha\textsc{Search}$ use breadth-first search order and let A_1, A_2 ... be a sequence of transition systems generated during iterative refinement performed by* $\textsc{RefinementSearch}(M, \varphi)$. *Then*

- *There exits i such that $RL(A_i) = RL(C(M))$.*
- *If the reachable part of the bisimulation quotient is finite, then there exists i such that $A_i \sim C(M)$.*

The basic idea of the proof is that any two states that are in different bisimulation classes ($s \not\sim s'$) will eventually be distinguished by the abstraction function ($\alpha_{\Phi_i}(s) \neq \alpha_{\Phi_i}(s')$). Moreover, each bisimulation class will eventually be visited by RefinementSearch and the (finite set) of reachable labellings will emerge.

Discussion The search order used in αSEARCH (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.

Also note that the refinement algorithm is non-monotone, i.e. a labelling which is reachable in one iteration may not be reachable in the next iteration. A similar problem occurs in the context of *must* abstractions: the set of *must* transitions is not generally monotonically increasing when predicates are added to refine an abstract system [12, 24]. However, we should note that the algorithm is guaranteed to converge to the correct answer.

We should also note that the proposed iterative algorithm is not guaranteed to terminate even for a finite state program. This situation is illustrated by the following example (the property we are checking is that $pc = 2$ is unreachable).

$$\begin{array}{l} pc = 0 \longmapsto x := 0, y := 0, pc := 1 \\ pc = 1 \wedge y \geq 0 \longmapsto y := y + x \\ pc = 1 \wedge y < 0 \longmapsto pc := 2 \end{array}$$

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 will not terminate on this example: it will keep adding predicates $y \geq 0, y+x \geq 0, y+2x \geq 0, \ldots$. Note that, in accordance with Theorem 3, it will eventually produce a bisimilar structure. However, the algorithm will not be able to detect termination, and it will keep 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 will occur with over-approximation based abstraction techniques that use refinement based on weakest precondition calculations [3, 21]. Those techniques will introduce the same predicates.

To solve this problem, 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 our example we would add predicates $x = 0$; $y = 0$. The abstraction will eventually become exact with respect to each transition. And since the number of reachable transitions is finite, the algorithm must terminate.

Corollary 1. *If $C(M)$ is finite state then the modified algorithm terminates.*

6 Extensions

Lightweight 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 $\alpha_\Phi(s')[e_i(\boldsymbol{x})/\boldsymbol{x}]$, 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 [21] for automated 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 [9], or predicates from known invariants of the system (generated using static analysis techniques).

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. Section 7 shows an example use of such abstractions for the analysis of Java programs.

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 [4, 16], in the context of over-approximation based predicate abstraction.

7 Applications

We have implemented our approach for the guarded command language. Our implementation is done in the language Ocaml and it uses the Simplify theorem prover [8]. The implementation uses several optimizations for checking only necessary queries. When updating Φ_{new} for refinement, we add only those conjuncts of $\alpha_{\Phi}(s')[e_i(\boldsymbol{x})/\boldsymbol{x}]$ for which we cannot prove validity. Moreover, we cache queries to ensure that the theorem prover is not called twice for the same query.

We discuss the application of our implementation for two concurrent programs: property verification for the Bakery mutual exclusion protocol and error detection in RAX (Remote Agent Experiment), a component extracted from an embedded spacecraft-control application.

These preliminary experiments show the merits of our approach. 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. We are currently doing an implementation in the Java PathFinder (JPF) model checking framework [26] for the analysis of Java programs. We briefly discuss at the end of this section the use of our approach for test-case generation for Java container implementations.

(Process 1)
$pc_1 = 0 \longmapsto x := y, pc_1 := 1$
$pc_1 = 1 \longmapsto x := x + 1, pc_1 := 2$
$pc_1 = 2 \wedge x \leq y \longmapsto pc_1 := 3$
$pc_1 = 3 \longmapsto pc_1 := 0$

(Process 2)
$pc_2 = 0 \longmapsto y := x, pc_2 := 1$
$pc_2 = 1 \longmapsto y := y + 1, pc_2 := 2$
$pc_2 = 2 \wedge y < x \longmapsto pc_2 := 3$
$pc_2 = 3 \longmapsto pc_2 := 0$

Fig. 4. Bakery example

Iteration	Concrete states	Abstract states	New predicates
1	17	11	$x \leq y$
2	18	12	$x + 1 \leq y, x \leq y + 1, y \geq 0$
3	26	19	$x + 2 \leq y, y \geq 1, x \leq 1$
4	44	32	$y \leq 1, x \leq 0, y \geq 2$
5	48	36	-

Fig. 5. Bakery example: intermediate results of the refinement algorithm

The Bakery Mutual Exclusion Protocol We have analyzed several versions of the Bakery mutual exclusion protocol (for two and more processes). These versions are infinite state but they have a finite bisimulation quotient. The guarded command representation for a simplified version of the protocol is given in Fig. 4.

The mutual exclusion property is encoded as "$pc_1 = 3 \wedge pc_2 = 3$ is unreachable". We used our tool to successfully prove that the property holds. Fig. 5 gives the intermediate results of the analysis. For each iteration, we report the number of generated concrete states, the number of stored abstract states and the newly generated predicates. Note that we never abstract the program counter. The reported results are for the breadth-first search order. For the depth-first search order the algorithm requires only 4 iterations (see the discussion in Section 5). The algorithm proceeds in similar way for the full version of the protocol.

RAX The RAX example (illustrated in Fig. 6) is derived from the software used within the NASA Deep Space 1 Remote Agent experiment, which deadlocked during flight [27]. 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, for breadth-first search order; the reported counter-example has 8 steps. For depth-first search order, the algorithm needs one more iteration to find the error, using the predicates that appear in the guards $c_1 = e_1$ and $c_2 = e_2$.

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 bisimulation quotient. The results are shown in Fig. 7 (breadth-first search order). However, if we assume that the counters in the program are non-negative, i.e. we introduce two new predicates, $e1 \geq 0$, $e2 \geq 0$, then the algorithm terminates after three iterations.

(Process 1)	(Process 2)
$pc_1 = 1 \longmapsto c_1 := 0, pc_1 := 2$	$pc_2 = 1 \longmapsto c_2 := 0, pc_2 := 2$
$pc_1 = 2 \wedge c_1 = e_1 \longmapsto pc_1 := 3$	$pc_2 = 2 \longmapsto e_1 := e_1 + 1, w_1 := 0, pc_2 := 3$
$pc_1 = 3 \longmapsto w_1 := 1, pc_1 := 4$	$pc_2 = 3 \wedge c2 = e2 \longmapsto pc_2 := 4$
$pc_1 = 4 \wedge w_1 = 0 \longmapsto pc_1 := 5$	$pc_2 = 4 \longmapsto w_2 := 1, pc_2 := 5$
$pc_1 = 2 \wedge c_1 \neq e_1 \longmapsto pc_1 := 5$	$pc_2 = 5 \wedge w_2 = 0 \longmapsto pc_2 := 6$
$pc_1 = 5 \longmapsto c_1 := e_1, pc_1 := 6$	$pc_2 = 3 \wedge c_2 \neq e_2 \longmapsto pc_2 := 6$
$pc_1 = 6 \longmapsto e_2 := e_2 + 1, w_2 := 0, pc_1 := 2$	$pc_2 = 6 \longmapsto c_2 := e_2, pc_2 := 2$

Fig. 6. RAX example

Iteration	Concrete states	Abstract states	New predicates
1	56	35	$c_1 = e_1, c_2 = e_2$
2	68	44	$e_1 = 0, e_2 = 0$
3	100	65	$e_1 = -1, e_2 = -1$
4	100	65	$e_1 = -2, e_2 = -2$
5	100	65	...

Fig. 7. RAX example: intermediate results of the refinement algorithm

The application of over-approximation based predicate abstraction to a Java version of RAX is described in detail in [27]. 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.

In general, we believe that the technique presented here is complementary to over-approximation abstraction methods and it can be used in conjunction with such methods, as an efficient way of discovering feasible counter-examples. We view the integration of the two approaches as an interesting topic for future research. Our technique explores transitions that are guaranteed to be feasible in the state space bounded by the abstraction predicates. In contrast, the over-approximation based methods may also explore transitions that are spurious and therefore could require additional refinement before reporting a real counter-example. Hence, our technique can potentially finish in fewer iterations and it can use fewer predicates (which enable more state space reduction), while retaining the model checker's capability of finding real bugs.

Testing We have used our preliminary implementation in the JPF model checker to perform test case generation to achieve code coverage for Java container classes (tree-map, linked-list, fibonacci-heap). Test cases are sequences of API calls, i.e. method calls that *add* and *remove* elements in a container, to obtain for example, branch coverage. The model checker analyzes all sequences of API calls up to a predefined sequence size and generates paths that are witnesses to testing coverage criteria encoded as reachability properties. Abstraction is used to match states between API calls and to avoid the generation of redundant tests.

We used an abstraction recording the (concrete) shape of the containers augmented with different predicate abstractions on the data fields from each container element — two states are matched if they represent containers that have the same shape and valuation for the abstraction predicates. The behavioral coverage obtained in this fashion is highly dependent on the different abstractions that are used. Therefore we believe that the capability of generating and refining the abstractions automatically is crucial for achieving good coverage. Although the work presented here is only a first step towards this goal (the JPF implementation does not yet allow automated refinement), we obtained better behavioral coverage than with exhaustive model checking. In fact, for some of the examples, exhaustive analysis runs out of memory even before generating tests that cover all the reachable branches in the code.

8 Conclusions and Future Work

We presented a novel 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 and for testing container implementations. In the future, we plan to extend the algorithm to checking liveness properties. We also plan to do an extensive evaluation of our approach on real systems.

References

1. T. Ball. A theory of predicate-complete test coverage and generation. *Technical Report MSR-TR-2004-28, Microsoft Research*, 2004.
2. T. Ball, A. Podelski, and S. Rajamani. Boolean and cartesian abstractions for model checking C programs. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *LNCS*, 2001.
3. 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.
4. S. Chaki, E. Clarke, A. Groce, and O. Strichman. Predicate abstraction with minimum predicates. In *Proc. 12th CHARME*, volume 2860 of *LNCS*, 2003.
5. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 4(2):511–547, August 1992.
6. D. Dams and K. S. Namjoshi. The existence of finite abstractions for branching time model checking. In *Proc. 19th Symposium on Logic in Computer Science (LICS'04)*, 2004.
7. L. de Alfaro, P. Godefroid, and R. Jagadeesan. Three-valued abstractions of games: Uncertainty, but with precision. In *Proc. 19th Symposium on Logic in Computer Science (LICS'04)*, 2004.
8. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. *Research Report 159, Compaq Systems Research Center*, 1998.
9. M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proc. 22nd International Conference on Software Engineering (ICSE'00)*, 2000.

10. R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples and refinements in abstract model checking. In *Proc. 8th Static Analysis Symposium (SAS'01)*, volume 2126 of *LNCS*, 2001.
11. P. Godefroid. Software Model Checking: the Verisoft Approach. *Formal Methods in Systems Design (to appear).*
12. P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based model checking using modal transition systems. In *Proc. CONCUR 2001 - Concurrency Theory*, volume 2154 of *LNCS*, 2001.
13. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proc. Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, 1997.
14. W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *Proc. International Symposium on Software Testing and Analysis (ISSTA'04)*, July 2002.
15. O. Grumberg, F. Lerda, O. Strichman, and M. Theobald. Proof-guided underapproximation-widening for multi-process systems. In *Proc. 32nd Symposium on Principles of Programming Languages (POPL'05)*, 2005.
16. T. A. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstractions from proofs. In *Proc. 31st Symposium on Principles of Programming Languages (POPL'04)*, 2004.
17. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. 29th Symposium on Principles of Programming Languages (POPL'02)*, 2002.
18. G. J. Holzmann and R. Joshi. Model-driven software verification. In *Proc. 11th SPIN Workshop*, volume 2989 of *LNCS*, Barcelona, Spain, 2004.
19. D. Lee and M. Yannakakis. Online minimization of transition systems. In *Proc. 24th ACM Symposium on Theory of Computing*, 1992.
20. 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 *Proc. 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, 2002.
21. K. S. Namjoshi and R. P. Kurshan. Syntactic program transformations for automatic abstraction. In *Proc. Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, 2000.
22. C. S. Păsăreanu, M. B. Dwyer, and W. Visser. Finding feasible abstract counterexamples. *STTT*, 5(1):34–48, November 2003.
23. C. S. Păsăreanu, R. Pelánek, and W. Visser. Concrete model checking with abstract matching and refinement (extended version). *RIACS Technical Report*, 05.04, 2005.
24. S. Shoham and O. Grumberg. Monotonic abstraction-refinement for CTL. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, Barcelona, Spain, 2004.
25. A. Venet and G. Brat. Precise and efficient static array bound checking for large embedded C programs. In *Proc. Programming Language Design and Implementation (PLDI'04)*, 2004.
26. W. Visser, K. Havelund, G. Brat, S. J. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2), April 2003.
27. W. Visser, S. Park, and J. Penix. Applying predicate abstraction to model check object-oriented programs. In *3rd ACM SIGSOFT Workshop on Formal Methods in Software Practice*, 2000.
28. T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. 19th Automated Software Engineering*, 2004.