The Synergy of Precise and Fast Abstractions for Program Verification

Natasha Sharygina
University of Lugano,
Switzerland
natasha.sharygina@unisi.ch

Stefano Tonetta
Fondazione Bruno Kessler,
Trento, Italy
tonettas@fbk.eu

Aliaksei Tsitovich
University of Lugano,
Switzerland
aliaksei.tsitovich@lu.unisi.ch

ABSTRACT

Predicate abstraction is a powerful technique to reduce the state space of a program to a finite and affordable number of states. It produces a conservative over-approximation where concrete states are grouped together according to a given set of predicates. A precise abstraction contains the minimal set of transitions with regards to the predicates, but as a result is computationally expensive. Most model checkers therefore approximate the abstraction to alleviate the computation of the abstract system by trading off precision with cost. However, approximation results in a higher number of refinement iterations, since it can produce more false counterexamples than its precise counterpart. The refinement loop can become prohibitively expensive for large programs.

This paper proposes a new abstraction refinement technique that combines slow and precise predicate abstraction techniques with fast and imprecise ones. It allows computing the abstraction quickly, but keeps it precise enough to avoid too many refinement iterations. We implemented the new algorithm in a state-of-the-art software model checker. Our tests with various real life benchmarks show that the new approach systematically outperforms both precise and imprecise techniques.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*

General Terms

Algorithms, Theory, Verification.

Keywords

Predicate Abstraction, Abstraction Refinement, CEGAR

1. INTRODUCTION

Predicate abstraction [17, 14], when combined with reachability analysis and an automated abstraction refinement mechanism (also known as *Counterexample Guided Abstraction Refinement* (CEGAR)[4, 10]), is an effective model checking strategy. The CEGAR-based verification consists of constructing and evaluating a finite-state system that is an abstract model of the original system with respect to a set of predicates.

The abstract model is a conservative over-approximation of the original program with respect to the set of given predicates. Thus, if the property holds on the abstract model, it also holds on the original program. The drawback of the conservative abstraction is that when model checking of the abstract program fails, it may produce a counterexample that does not correspond to any concrete counterexample. This is called a *spurious counterexample*. When a spurious counterexample is encountered, *refinement* is performed by adjusting the set of predicates in a way that eliminates the given counterexample. The overall efficiency of verification is highly dependent on the efficiency of the abstraction and refinement procedures.

Computing the abstract model relies on enumerating the abstract states and checking, for each pair of states, if there exists an abstract transition. This computation is expensive since it requires an exponential number of calls to a theorem prover [16, 2, 1]. In [27, 8, 26], the abstraction is computed by means of dedicated decision procedures based on BDDs, SAT or SAT modulo theories (SMT). As another direction, various techniques have been proposed to alleviate this computation by approximating the abstract transition relation (see, for example, [15, 2, 3, 1, 23]).

We distinguish between *precise* abstraction and *approximated* abstraction (as also done, for example, in [10, 15, 23]): a precise abstraction is minimal in the sense that it contains only those transitions that correspond to some transition in the concrete model; instead, an approximated abstraction is a further over-approximation of the minimal abstract model so that the transition relation is relaxed. In the paper, we will refer to the latter simply as approximation.

Approximation techniques are important because they allow a less expensive (as compared to precise abstraction) computation of the abstract transition relation. Cartesian abstraction [3], for example, loses every relationship among predicates, but has been successfully used to verify large programs, such as operating system device drivers. However, abstraction approximations add spurious behaviors in addition to the spurious counterexamples resulting from precise abstraction. In order to rule out this kind of "impurity", the approximation must be refined without changing the set of predicates and focusing only on the spurious transitions caused by the approximation [15]. This procedure on its own might become very costly and can not fit to verification of large programs.

When refining the abstract model, we distinguish between two types of spurious behavior (as also done in [12]). 1) *Spurious path*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'09 March 8-12, 2009, Honolulu, Hawaii, U.S.A.
Copyright 2009 ACM 978-1-60558-166-8/09/03 ...$5.00.

```
    void main() {
    int x=*;
    int y;
l0: y=x+1;
l1: if (x<0)
l2: if (!(x<y))
l3: assert(0);
l4:}
```

(a)

```
void main() {
int x = *;
int y;
int z;
y=x+1;
z=y-1;
if (x<0||y>0||z>0){
x++;
z++;
if (y<z && z>x)
assert(0); } }
```

(b)

Figure 1: Sample program for which the approximated abstraction causes spurious transitions.

is due to the over-approximating nature of the precise abstraction: states are merged together so that some resulting paths cannot be simulated on the concrete system. This happens when the set of predicates is not sufficient to capture the relevant behaviors of the concrete system. 2) *Spurious transitions* are abstract transitions which do not have corresponding concrete transitions. By definition, spurious transitions cannot appear in the most precise abstraction and are caused by using the approximation techniques. Clearly, the efficiency of the approximated abstraction depends on a trade-off between time spent in computing the abstraction and refining spurious transitions.

In order to illustrate the abstraction approximation and its refinement procedures, consider the example of Figure 1(a). The variable `x` is assigned non-deterministically with an unknown value "`*`". The property we verify is the reachability of line `l3`. It never can be true since the condition `!(x<y)` at line `l2` never holds (the conditional statement with guard `x<0` is necessary to avoid integer overflow). Thus if in the abstract program there is a path leading to the assertion, then it is spurious. The predicates `x<0` and `x<y` are sufficient to prove the property. However, approximate methods like Cartesian abstraction cannot prove it because they cannot infer that after the assignment `y=x+1`, the condition `!(x<0) || !(x<y)` is true. Thus, most model checkers that use such abstractions refine the transition relation by adding a constraint that removes the spurious transition.

In order to experience the difference in performance between precise and approximated abstractions, let us extend the previous example in order to have more spurious behaviors. The program of Figure 1(b) has one more variable and a slightly more complex control flow graph. As before the assertion is not reachable, and all abstract counterexamples are spurious. Though, if we consider the predicates in the guards of the program, an approximated abstraction may produce many spurious behaviors. Table 1 reports the verification results obtained with the SATABS model checker [13], by running approximated and precise abstractions. The final number of predicates is in all cases 10. The approximated abstraction spends most of time in refining the transition relation (Ref). Since it runs for 12 iterations (or even 42 in case when we used the refinement procedure of [15]), also the time for the verification (MC) is not negligible. On the contrary, the precise abstraction takes only 2 iterations to terminate (the first refinement is necessary to add a sufficient set of predicates). Nevertheless, the amount of time spent in computing the abstraction is too high for such example.

A low number of refinement iterations is fundamental for the success of the CEGAR loop, especially when applied to industrial benchmarks: in fact, when the system is complex, the number of predicates required to verify the property becomes high, and the time spent in the reachability (model checking) procedure grows exponentially. For this reason, it is of paramount importance to avoid as many redundant iterations as possible: even a single saved iteration can result into a huge saving in time for large systems.

Contributions

This paper presents a CEGAR-based technique that controls the number of iterations and reduces the verification time by interleaving precise (but slow) and approximated (but fast) abstractions. The abstraction is first computed with a high level of approximation exploiting the weakest precondition of the predicates. Then, during the refinement step, our technique uses the SAT-based quantifier elimination in order to compute a precise abstraction.

The blow up that we would experience in computing the precise abstraction of the whole program is avoided by exploiting the localized abstraction: as in static analysis [30], in most model checkers (such as SLAM [2], BLAST [20], SATABS [13], F-Soft [22]) the abstract model keeps the control flow graph of the original program and has a different abstract transition relation for each location of the control-flow graph[1]. This way, during the refinement step, we add the constraints built with a precise abstraction only to relevant transition relations, affecting only those parts of the system that caused the spurious counterexample.

In order to illustrate the immediate advantages of our approach, consider the fourth line of Table 1 that is based on the implementation of our technique. Our approach is able to avoid both a high number of iterations and an expensive abstraction, resulting in an optimized verification time.

We performed a thorough evaluation comparing the new technique with the purely precise and imprecise counterparts. Our tests with various real life benchmarks show a systematic advantage of our approach over both precise and imprecise techniques reaching up to 90% improvement in time.

Overall, the new technique manages the verification complexity by using the precise abstraction on demand and locally. The advantage is that the expensive abstraction is only used on a small portion of the program, yet the higher quality of abstraction refinement is sufficient to reduce the number of refinement iterations, thus improving the overall performance.

Related work

The paper addresses the problem of refining the abstraction in the presence of spurious transitions. The solution was first given by Das and Dill [15] whose technique consists of removing one spurious transition at every refinement iteration. The approach may be very expensive because it requires a high number of iterations of the abstraction-refinement loop. In practice, the technique is not feasible for real systems.

Many works such as [1] improved the refinement by strengthening the condition added to the transition relation to remove more spurious transitions. The idea in [1] is to syntactically simplify the condition and to check if a larger set of spurious transitions is found.

In [9, 21, 22], a different technique is presented based on SAT techniques. Transitions are simulated over the concrete program by means of SAT formulas. If the transition is not concretizable the SAT solver will produce a resolution proof of the unsatisfiability. It is then possible to extract from the proof either a core set of predicates or a constraint sufficient to remove the spurious transition. Though, in principle, the technique can remove many spurious transitions at once, the efficiency strongly depends on the unsatisfiability proof. In the worst case, it may require a number of

[1] Localized abstraction is further investigated in [20, 19].

	Total	Abs	MC	Ref	Iter
Approximated abstraction [15]	5.817	0.063	2.659	2.112	42
Approximated abstraction [21]	1.469	0.046	0.501	0.617	12
Precise abstraction	3.591	3.478	0.076	0.01	2
New approach	0.467	0.039	0.161	0.189	4

Table 1: Verification results on the example of Figure 1(b). *Total, Abs, MC, Ref* refer to the time, in seconds, for total verification, abstraction, model checking and refinement respectively; *Iter* refers to the number of iterations of the abstraction-refinement loop.

abstraction refinements exponential in the number of predicates.

The technique of [23] also exploits the unsatisfiability proof but it is based on interpolation. The interpolant produced by the proof is indeed an over-approximation of the exact abstraction able to remove the spurious transition. As in the case of unsat cores, the technique depends on the heuristics to produce unsatisfiability proofs. The interpolant is not always enough strong to remove all spurious transitions.

This paper instead proposes a greedy approach where all spurious transitions between two locations are removed. The idea is that the computation can be efficient because it is localized and on-demand. The technique inherits the efficiency of the approximated abstraction which is used any time new predicates are discovered. At the same time, the precision of the minimal abstraction is exploited whenever spurious transitions are found.

Summary

The paper is organized as follows: Section 2 gives an overview of related abstraction refinement techniques; Section 3 describes our new approach; Section 4 presents the experimental evaluation; finally, Section 5 draws the conclusions.

2. BACKGROUND

2.1 Transition Systems

We consider programs as Transition Systems. TSs are defined by a set V of state variables. We use V' to denote the set of next state variables $\{v'\}_{v \in V}$, where v' represents the next value of v. The set S_V of states is given by all assignments to the variables V. Given a state s, s' denotes the corresponding assignment to the next state variables, i.e. $s' = s[V'/V]$. Transitions are represented as pairs of states. For each transition $t = (s_1, s_2)$, we use $in(t)$ and $out(t)$ to denote resp. s_1 and s_2. Given a formula ϕ, we write $\phi[V/V']$ to denote the result of substituting every free occurrence of every variable $v' \in V'$ with its corresponding v. We use $\exists V(\phi)$ to denote the existential quantification of every variable in V.

Definition 1 *A Transition System (TS) is a tuple $M = \langle V, I, T \rangle$, where*

- *V is a set of variables;*
- *$I(V)$ is a formula that represents the initial states;*
- *$T(V, V')$ is a formula that represents the transitions.*

A state s is initial iff $s \models I(V)$. Given two states s_1 and s_2, there exists a transition t between s_1 and s_2 iff $s_1, s_2' \models T(V, V')$. A path of M is a finite sequence π of transitions $t_0, t_1, ..., t_n$ such that $in(t_0) \models I$, and, for every $0 \leq i < n$, $out(t_i) = in(t_{i+1})$. In general, given a transition relation T, we use $\pi \models T$ to denote that $\pi[i] \models T$ for every $0 \leq i \leq |\pi|$.

Example 1 *Consider the program of Figure 1(a). It can be represented by the TS $M = \langle V, I, T \rangle$, where*

- *$V := \{x, y, pc\}$, where pc is the program counter;*
- *$I := (pc = l_0)$;*
- $$\begin{aligned} T := \ & (pc = l_0) \rightarrow (pc' = l_1 \wedge y' = x + 1 \wedge x' = x) \wedge \\ & (pc = l_1 \wedge x < 0) \rightarrow (pc' = l_2 \wedge x' = x \wedge y' = y) \wedge \\ & (pc = l_1 \wedge !x < 0) \rightarrow (pc' = l_4 \wedge x' = x \wedge y' = y) \wedge \\ & (pc = l_2 \wedge !x < y) \rightarrow (pc' = l_3 \wedge x' = x \wedge y' = y) \wedge \\ & (pc = l_2 \wedge x < y) \rightarrow (pc' = l_4 \wedge x' = x \wedge y' = y) \wedge \\ & (pc = l_3) \rightarrow (pc' = l_4 \wedge x' = x \wedge y' = y) \end{aligned}$$

2.2 Abstraction

Definition 2 *Given two TSs $M = \langle V, I, T \rangle$ and $\hat{M} = \langle \hat{V}, \hat{I}, \hat{T} \rangle$, a relation $H(V, \hat{V})$ is an* abstraction relation *[11] iff the following conditions hold:*

- *every initial state of M corresponds to an initial state of $\hat{M}$; namely, if $s \models I(V)$, then there exists a state $\hat{s}$ of $\hat{M}$ such that $\hat{s} \models \hat{I}(\hat{V})$ and $s, \hat{s} \models H(V, \hat{V})$;*
- *every transition of M corresponds to a transition of $\hat{M}$; namely, if $s_1, \hat{s}_1 \models H(V, \hat{V})$, and $s_1, s_2' \models T(V, V')$, then there exists a state $\hat{s}_2$ of $\hat{M}$ such that $s_2, \hat{s}_2 \models H(V, \hat{V})$ and $\hat{s}_1, \hat{s}_2' \models \hat{T}(V, V')$.*

If such relation exists, we say that $\hat{M}$ is an abstraction *of M, or M* refines *$\hat{M}$ ($M \preceq \hat{M}$).*

Definition 3 *Given the abstraction relation H, we define the* abstraction function *$\alpha_H : 2^{S_V} \rightarrow 2^{S_{\hat{V}}}$ and the* concretization function *$\gamma_H : 2^{S_{\hat{V}}} \rightarrow 2^{S_V}$ as follows:*

- *$\alpha_H(Q) = \{\hat{s} \in S_{\hat{V}} \mid$ there exists $s \in Q$ s.t. $s, \hat{s} \models H(V, \hat{V})\}$, for every $Q \subseteq S_V$;*
- *$\gamma_H(\hat{Q}) = \{s \in S_V \mid$ there exists $\hat{s} \in \hat{Q}$ s.t. $s, \hat{s} \models H(V, \hat{V})\}$, for every $\hat{Q} \subseteq S_{\hat{V}}$.*

We extend γ to transitions and paths so that:

- $\gamma_H(\hat{t}) = \{t \mid in(t) \in \gamma(in(\hat{t})), out(t) \in \gamma(out(\hat{t}))\}$, for every transition $\hat{t}$ of $\hat{M}$.
- $\gamma_H(\hat{\pi}) = \{\pi \mid \pi[i] \in \gamma(\hat{\pi}[i])$ for every $0 \leq i \leq |\hat{\pi}|\}$, for every path $\hat{\pi}$ of $\hat{M}$.

If a property ϕ is universal a system M satisfies the property ($M \models \phi$) if and only if the property is satisfied by all paths of M. The abstraction relation we defined preserves universal properties, so that if $M \preceq \hat{M}$, ϕ is a universal property, and $\hat{M} \models \phi$, then $M \models \phi$ (though, in general, the reverse does not hold). Given a TS $M = \langle V, I, T \rangle$, an abstraction $\hat{M} = \langle \hat{V}, \hat{I}, \hat{T} \rangle$ of M is said to be *precise* when every abstract initial state and transition of $\hat{M}$ corresponds respectively to a concrete initial state and transition of M. Given the abstraction relation H, $\hat{M}$ can be obtained as:

- $\hat{I}_H(\hat{V}) = \exists V(I(V) \wedge H(V, \hat{V}))$,
- $\hat{T}_H(\hat{V}, \hat{V}') = \exists V \exists V'(T(V, V') \wedge H(V, \hat{V}) \wedge H(V', \hat{V}'))$

The precise abstraction is also called *minimal* or *existential* or *exact* or *eager* abstraction [11].

Given a TS $M = \langle V, I, T \rangle$, let P be a set of predicates and $\hat{v}_p$ an abstract variable for every predicate $p \in P$. The set of abstract variables is the set $\hat{V}_P = \{\hat{v}_p\}_{p \in P}$. The abstraction relation for predicate abstraction is defined as follows:

$$H_P(V, \hat{V}_P) = \bigwedge_{p \in P} \hat{v}_p \leftrightarrow p(V)$$

The minimal predicate abstraction is the TS $\hat{M} = \langle \hat{V}_P, \hat{I}_P, \hat{T}_P \rangle$, where:

- $\hat{I}_P(\hat{V}_P) = \exists V(I(V) \wedge \bigwedge_{p \in P} \hat{v}_p \leftrightarrow p(V))$
- $\hat{T}_P(\hat{V}_P, \hat{V}'_P) = \exists V \exists V'(T(V, V') \wedge \bigwedge_{p \in P}(\hat{v}_p \leftrightarrow p(V) \wedge \hat{v}'_p \leftrightarrow p(V')))$.

2.2.1 Quantifier elimination

In order to model check the abstract TS, it is necessary to compute the set of successors of abstract states. This requires the removal of the quantifiers from the definition of the abstract transition relation. In general, given a transition relation T and a set of predicates P, *to compute* $\hat{T}_P$ means to find a quantifier-free formula that is equivalent to $\hat{T}_P$.

Example 2 *Consider the TS described in the Example 1 and the predicates $P_1 := (x < 0)$ and $P_2 := (x < y)$. Let the abstract variables $\hat{v}_1$ and $\hat{v}_2$ correspond respectively to P_1 and P_2. We do not abstract the program counter. The abstract transition relation results to be equivalent to*

- $$\begin{aligned} \hat{T}_P \equiv\ & (pc = l_0 \wedge \hat{v}_1) \rightarrow (pc' = l_1 \wedge ! \hat{v}'_2) \wedge \\ & (pc = l_0 \wedge ! \hat{v}_1) \rightarrow (pc' = l_1) \wedge \\ & (pc = l_1 \wedge \hat{v}_1) \rightarrow (pc' = l_2 \wedge \hat{v}'_1 = \hat{v}_1 \wedge \hat{v}'_2 = \hat{v}_2) \wedge \\ & (pc = l_1 \wedge ! \hat{v}_1) \rightarrow (pc' = l_4 \wedge \hat{v}'_1 = \hat{v}_1 \wedge \hat{v}'_2 = \hat{v}_2) \wedge \\ & (pc = l_2 \wedge ! \hat{v}_2) \rightarrow (pc' = l_3 \wedge \hat{v}'_1 = \hat{v}_1 \wedge \hat{v}'_2 = \hat{v}_2) \wedge \\ & (pc = l_2 \wedge \hat{v}_2) \rightarrow (pc' = l_4 \wedge \hat{v}'_1 = \hat{v}_1 \wedge \hat{v}'_2 = \hat{v}_2) \wedge \\ & (pc = l_3) \rightarrow (pc' = l_4 \wedge \hat{v}'_1 = \hat{v}_1 \wedge \hat{v}'_2 = \hat{v}_2) \end{aligned}$$

In hardware and software verification, different techniques have been conceived to compute $\hat{T}_P$. In symbolic model checking [7] of finite state machines, the existential quantification can be removed either by a Shannon expansion technique when using BDDs [6] or by SAT techniques when using CNF [29]. In software model checking, the problem is exacerbated by the fact that the concrete transition relation may contain first-order terms. The abstract transition relation can be obtained by enumerating the abstract states, and checking if, for each pair of states, there exists an abstract transition. As it is done by most software model checkers, this requires an exponential number of calls to a theorem prover [16, 2]. In [13], a SAT solver is exploited to find all possible solutions. We refer to this technique as $SATQE$.

2.3 Abstraction approximation

Precise abstractions are very expensive to compute because of the existential quantification operations. Thus, in practice, model checkers use approximations to trade-off precision with complexity.

Definition 4 *Formally, given $M_H = \langle V, I_H, T_H \rangle$ and $\tilde{M} = \langle V, \tilde{I}, \tilde{T} \rangle$, we say that $\tilde{M}$ is an* approximation *of M_H ($M_H \precsim \tilde{M}$) iff the following formulas are valid:*

- *$I_H \rightarrow \tilde{I}$, i.e., every initial state of the minimal abstraction is an initial state in the approximation;*
- *$T_H \rightarrow \tilde{T}$, i.e., every transition of the minimal abstraction is a transition in the approximation.*

Intuitively, $\tilde{M}$ has more initial states and transitions than M_H. Note that an approximation is also an abstraction namely, if $M_H \precsim \tilde{M}$, then $M_H \preceq \tilde{M}$. However, the set of predicates is not affected, in the sense that $\tilde{M}$ and M_H have the same abstract variables.

2.3.1 Approximation techniques

Many approximation techniques have been developed both in hardware and software verification. Their aim is to alleviate the computation of $\hat{T}_P$. The easiest way is to reduce the scope of quantifiers. This can be done with *early quantification* [11], by pushing quantifiers in front of predicates. *Predicate partitioning* [21] approximates $\hat{T}_P$ by taking the conjunction of its projections over subsets of predicates. This technique is pushed to its limit by Cartesian abstraction [3] that, given a set of states Q, approximates transition relation with the product of the projections on each variable. This way, the approximated abstraction ignores every relation among predicates.

2.4 Spurious behaviors

The overapproximation nature of the abstraction as we define may generate spurious paths even in the case of precise abstraction. Spurious paths are sequences of transitions that satisfy the abstract transition relation, but not the concrete one.

Definition 5 (Spurious path) *Given a TS $M = \langle V, I, T \rangle$, an abstraction $\hat{M} = \langle \hat{V}, \hat{I}, \hat{T} \rangle$, and a sequence $\hat{\pi}$ of transitions of $\hat{M}$, we say that $\hat{\pi}$ is a spurious path iff $\hat{\pi} \models \hat{T}$ and $\pi \not\models T$ for every $\pi \in \gamma(\hat{\pi})$.*

In order to refine the abstraction and remove a spurious path, refinement procedures need to add more predicates to the abstraction. There are different techniques to discover the new set of predicates, either based on weakest precondition [5], interpolation [19], or UNSAT core [18].

Besides spurious path, approximated abstraction generates another kind of spurious behavior, called spurious transitions. Spurious transitions are transitions that satisfy the abstract transition relation, but not the concrete one.

Definition 6 (Spurious transition) *Given a TS $M = \langle V, I, T \rangle$, an abstraction $\hat{M} = \langle \hat{V}, \hat{I}, \hat{T} \rangle$, and a transition $\hat{t}$ of $\hat{M}$, we say that $\hat{t}$ is a spurious transition iff $\hat{t} \models \hat{T}$ and $t \not\models T$ for every $t \in \gamma(\hat{t})$.*

In order to refine an approximation that contains a spurious transition, a new transition relation is obtained by adding a constraint in conjunction to the old abstract transition relation. As a result, the spurious counterexample is ruled out. Different techniques uses as such constraint either the exact encoding of the spurious transition [15], or the UNSAT core produced by the SAT solver when checking if the transition is spurious [21], or an interpolant between the exact abstraction and the current approximated abstraction [23].

3. THE SYNERGY ALGORITHM

This section proposes a new refinement algorithm. It uses both the fast and precise types of abstraction to gain verification efficiency. It is independent of any particular technique used to define either procedure.

```
1  MixCegarLoop(TransitionSystem M, Property F)
2  begin
3      $\Pi$ = InitialPredicates($F$,$T$);
4      $\alpha$ = FastAbstraction($T$,$\Pi$);
5      while not TIMEOUT do
6          $\pi$ = ModelCheck($\alpha$,$F$);
7          if $\pi = \emptyset$ then return CORRECT;
8          else
9              $\sigma_{ST}$ = SpuriousTransition($\pi$);
10             if $\sigma_{ST} \neq \emptyset$ then
11                 foreach $t \in \pi$ do
12                     $C$ = PreciseAbstraction($T$,$\sigma_{ST}(t)$);
13                     $\alpha = \alpha \wedge C$;
14             else
15                 $\sigma_{SP}$ = SpuriousPath($\pi$);
16                 if $\sigma_{SP} \neq \emptyset$ then return INCORRECT;
17                 else
18                     foreach $t \in \pi$ do
19                         $\Pi = \Pi \cup \sigma_{SP}(t)$;
20                         $C$ = PreciseAbstraction($T$,$\sigma_{SP}(t)$);
21                         $\alpha = \alpha \wedge C$;
22 end
```

Algorithm 1: A new abstraction-refinement algorithm combining fast and precise abstractions.

The algorithm implements the standard CEGAR loop. Each iteration of the CEGAR loop is composed of an abstraction step, a model checking step, a simulation step and finally a refinement step.

We first present the high-level overview of the combined algorithm and then describe the specifics of the new refinement procedures. For simplicity, we first present the algorithm with regard to a monolithic transition relation. In Section 3.3 we extend it to the case where a transition relation is defined for every location of the program.

The algorithm is parameterized by a number of subroutines that take care of the abstraction and refinement. In particular, the algorithm contains the following procedures:

- `FastAbstraction`: given a set of predicates Π and a concrete transition relation T, it computes an over-approximation of $\hat{T}_{\Pi}$.
- `PreciseAbstraction`: given a set of predicates Π and a concrete transition relation T, it computes the minimal abstraction $\hat{T}_{\Pi}$.
- `SpuriousTransition`: given a path π, it returns a function σ_{ST} that maps every transition t in π to a set of predicates P, s.t., $P \subseteq \Pi$ and $t \not\models \hat{T}_P$.
- `SpuriousPath`: given a path π, it returns a function σ_{SP} that maps every transition t in π to a set of predicates P, s.t. $\pi \not\models \hat{T}_{\sigma_{SP}(t)}$. Note that P may contain new and old predicates.

Algorithm 1 shows how the `FastAbstraction` and `PreciseAbstraction` are combined. It first computes the approximated abstraction (line 4). When a spurious counterexample is encountered as a result of the model checking (line 6), the spurious transitions are removed by using the precise abstraction technique (line 12) with the predicates returned by `SpuriousTransition` (line 9). If no spurious transitions are found, the spurious path is removed by using the precise abstraction technique (line 20) with the predicates returned by `SpuriousPath` (line 15).

3.1 Refining spurious transitions (lines 9-13)

Suppose some transitions $t_1, ..., t_n$ of the counterexample π found by `ModelCheck` are spurious. This means that the function σ_{ST} returned by `SpuriousTransition` maps those transitions to some non-empty set of predicates. Let us define the clustering of predicates Γ as $\{\sigma_{ST}(t_i)\}_{1 \leq i \leq n}$ (i.e., Γ contains the set of predicates $\sigma_{ST}(t_i)$ for every transition in the spurious counterexample). The spurious transition refinement procedure proceeds as follows. For each cluster, $P \in \Gamma$, the refinement algorithm computes $\hat{T}_P$, which is a precise computation of the abstract transition relation projected on the predicates of the cluster. In order to rule out every spurious transition among $t_1, ..., t_n$, the refinement algorithm updates the abstract transition relation as follows:

$$\alpha' := \alpha \wedge \bigwedge_{P \in \Gamma} \hat{T}_P$$

Note that, in general, every cluster, P, is a subset of the global set of predicates, Π. This means that each constraint $\hat{T}_P$ is an over-approximation of the precise abstraction computed over Π. Nevertheless $\hat{T}_P$ is precise with regards to the predicates P, in the sense, that it removes all the unrealistic abstract transitions that can be defined by the those predicates.

The following theorem states the soundness of this refinement step.

Theorem 1 *For every spurious transition* t_i, $1 \leq i \leq n$, $t_i \not\models \alpha'$.

Proof Sketch. The proof comes directly from the definition of σ_{ST} (it relies therefore on the soundness of a particular `SpuriousTransition` technique): for $1 \leq i \leq n$, since $t_i \not\models \hat{T}_{\sigma_{ST}(t_i)}$, $t_i \not\models \alpha'$.

3.2 Refining spurious paths (lines 15-21)

We adopt the cluster-based approach described above to the removal of the spurious path. Our technique uses `SpuriousPath` to produce the set of predicates that are sufficient to rule out the spurious counterexample. The set of predicates generated by the standard predicate-discovery techniques (described in Section 2) includes both current predicates and new predicates, that together rule out the spurious counterexample. Our technique considers this set of old and new predicates as a new cluster.

Suppose the path $t_1, ..., t_n$ to be spurious. This means that the function σ_{SP} returned by `SpuriousPath` maps each t_i to some non-empty set of predicates. Let us define the clustering of predicates Γ as $\{\sigma_{SP}(t_i)\}_{1 \leq i \leq n}$ (i.e., Γ contains the set of predicates $\sigma_{SP}(t_i)$ for every transition in the spurious counterexample). The computation of the updated abstract transition relation is identical to spurious transition case, i.e.

$$\alpha' := \alpha \wedge \bigwedge_{P \in \Gamma} \hat{T}_P$$

Note that this time, unlike the case of spurious transitions, the clusters involve new predicates.

By definition, the set of predicates produced by `SpuriousPath` is sufficient to remove the spurious counterexample only if the precise abstraction is used. In fact, spurious transitions over such predicates (possibly created by the approximation abstraction) might create the same spurious counterexample. Our technique guarantees that this does not happen. This is achieved by using the precise component $\hat{T}_P$.

The following theorem states the soundness of this refinement step.

```
1  MixCegarLoop(TransitionSystem M, Property F)
2  begin
3      foreach T in M do $\Pi(T)$ = InitialPredicates(F,T);
4      foreach T in M do $\alpha(T)$ = FastAbstraction(T,$\Pi$);
5      while not TIMEOUT do
6          $\pi$ = ModelCheck($\alpha$,F);
7          if $\pi = \emptyset$ then return CORRECT;
8          else
9              $\sigma$ = SpuriousTransition($\pi$);
10             if $\sigma \neq \emptyset$ then
11                 foreach $t \in \pi$ do
12                     $T = \tau(t)$;
13                     $C$ = PreciseAbstraction(T,$\sigma(t)$);
14                     $\alpha(T) = \alpha(T) \wedge C$;
15             else
16                 $\sigma_{SP}$ = SpuriousPath($\pi$);
17                 if $\sigma_{SP} \neq \emptyset$ then return INCORRECT;
18                 else
19                     foreach $t \in \pi$ do
20                         $T = \tau(t)$;
21                         $\Pi(T) = \Pi(T) \cup \sigma_{SP}(t)$;
22                         $C$ =
                           PreciseAbstraction(T,$\sigma_{SP}(t)$);
23                         $\alpha(T) = \alpha(T) \wedge C$;
24 end
```

Algorithm 2: CEGAR loop with localized abstraction.

Theorem 2 *For every spurious path π, $\pi \not\models \alpha'$.*

Proof Sketch. The proof comes directly from the definition of σ_{SP} (it relies therefore on the soundness of a particular SpuriousPath technique).

3.3 Localized abstraction

The algorithm shown in Algorithm 1 was defined for a monolithic transition relation. When the set of predicates returned by the SpuriousTransition or SpuriousPath procedures covers the whole set Π of current predicates, the constraint that MixCegarLoop adds to the abstract transition corresponds exactly to the precise abstraction. This way, the abstraction refinement becomes as expensive as PreciseAbstraction. We limit this disadvantage by localizing the abstraction to some parts of the program. Some software model checkers (e.g., BLAST [20] and SATABS [13]) use the control flow graph as a partitioning of the transition relation to implement such localization. During the abstraction refinement, they keep a set of predicates and an abstract transition relation for each program location, and perform the abstraction for each local transition relation separately.

Our algorithm implements the localized procedure as part of the CEGAR algorithm as shown in Algorithm 2. The algorithm treats the system M as a set of concrete transition relations, one for every location of the control-flow graph. For each transition relation T, it computes an abstract transition relation $\alpha(T)$ (line 4); when a spurious counterexample is encountered as a result of the model checking (line 6), spurious transitions and path are removed by using the precise abstraction technique (line 13 and 22). The difference from the monolithic case (presented earlier in this section) is that in the localized version, every transition t of the spurious counterexample π is associated with a particular abstract transition relation, denoted $\tau(t)$. Thus, when the refinement step of the algorithm has to add a new constraint, it changes only the transition relation corresponding to either the spurious transition (as part of the spurious transition refinement step, lines 9-14) or to each transition of the spurious path (as part of the spurious path refinement step, lines 16-23).

By exploiting the localized-abstraction framework, the algorithm reduces the abstraction computation to the parts of the system that are relevant to the property and keeps the approximated abstraction in all parts of the program that are irrelevant to prove the property.

4. EXPERIMENTAL RESULTS

We implemented the proposed algorithm in the framework of software model checking. We used the SATABS [13] model checker as a platform for our experiments. As described in Section 3, the new CEGAR loop uses four subroutines. We experimented with the following techniques implemented in SATABS:

- for FastAbstraction, we used a fast abstraction technique based on the computation of the weakest precondition; it assigns to the next predicate its weakest precondition if this is a current predicate; it does not allow a general Boolean combination of predicate variables;
- for PreciseAbstraction, we used a precise abstraction based on the enumeration of possible transitions by means of a SAT solver: we force the SAT solver to find all the solutions of the quantifier-elimination problem by iteratively adding the negation of previous assignments as clauses [13];
- for SpuriousTransition, we used the SAT-based technique of [21][2]; this calls a SAT solver to check if a transition is spurious; if the transition is not realistic, it inspects the UNSAT proof to find the relevant predicates;
- for SpuriousPath, we used a technique based on weakest precondition; it computes the weakest preconditions of the current predicates along the transitions of the spurious path; it uses these expressions to produce a set of current and new predicates that are sufficient to rule out the spurious path.

The SAT solver used by PreciseAbstraction and SpuriousTransition was MiniSAT[3]. We implemented the new algorithm and enhanced SATABS with two new procedures: the first (we will refer to it as NewST) affects how the abstraction is refined in the case of spurious transitions, as described in Section 3.1; the second (NewSP) refines the abstraction in the case of spurious paths, as described in Section 3.2.

We ran the experiments on a AMD Dual-Core Opteron(TM) 2212 machine with 2GHz CPU and Ubuntu 7.04 We compared the pure fast abstraction and the pure precise abstraction (as implemented in SATABS) with the new algorithm where we used either NewSP or NewST or both together. We evaluated the techniques on different examples with different assertions. For every experiment, we verified one property at a time.[4]

We first compared the different techniques on a C implementation of a multi server/client shopping agent system as reported in Fig. 2. This example is particularly interesting because the fast abstraction produces a number of spurious transitions exponential in the number of predicates.

As seen in Fig. 2, the performance of the weakest-precondition-based (WP) and the SAT-based abstractions (SATQE) is comparable. Notably, NewST separately and in combination with NewSP

[2] We also experimented with an implementation of technique [15], but it reached 200 CEGAR iterations even on the small examples.

[3] http://minisat.se/

[4] We observed that verifying several assertions at the same time makes the results unreliable, since the counterexample produced by the model checker may vary according to different abstract models. This way, at the same iteration we might obtain different predicates which might close the CEGAR loop in a different number of iterations.

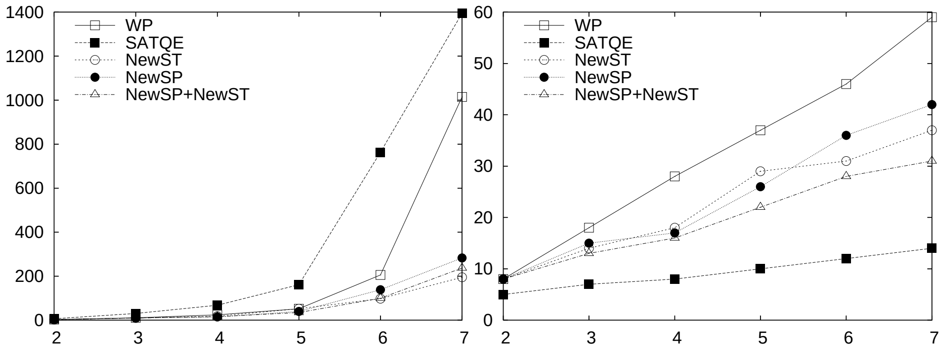

Figure 2: Total running time in seconds (left) and number of iterations (right) plotted against the number of clients in the server/client example.

is much more efficient than either WP or SATQE. WP and NewSP are sensitive to a number of spurious transitions and, due to the nature of the example, grow exponentially with the growth of the model. NewST efficiently removes spurious transitions and significantly reduces the number of iterations. In Fig. 2 (right) we note that the new technique as expected has a balanced number of iterations between WP and SATQE. This produces an evident saving in time (as shown in Fig. 2 left) comparing to either WP (up to factor of 5) and to SATQE (up to factor of 7!).

Secondly, we evaluated the techniques on the benchmark set proposed in [25]. For the benchmark set the authors collected a large number of large-scale C programs with known buffer-overflow bugs and their fixed versions. The test suite includes applications such as Sendmail, Apache HTTP server, Samba etc.; though, the original programs were stripped down by substituting libraries with stubs. The benchmark set contains 568 test cases, of which 261 are fixed versions of the programs. SATABS needs on average 106 predicates to check these programs, with a maximum of 239 predicates.

As expected, SATQE does not perform efficiently on large real programs because of the large number of predicates involved. NewST outperforms uniformly NewSP. The most interesting result remains the comparison between WP and NewST which gives a deeper understanding of the improvement of our techniques. Fig. 3 reports the scatter plots of the comparison.[5]

We pruned all claims that reached a timeout of 3 hours or 200 iterations of CEGAR and those verified in less than 100 seconds, since the performance difference was not relevant. In fact, 40% of the test cases were completed in less then 2 seconds and not more than 5 iterations.

The results show that NewST systematically outperforms WP. In 98% of the test cases it wins in number of iterations required to verify the property. The advantage in iterations leads to a total verification time win in 71% of the tests. On average, it decreased total time for 34%, reaching up to 90% improvement in a number of cases.

In the remaining 29% of the test cases, where NewST was not better than WP, the difference in verification time usually was not bigger than 15%. As an exception, we found only one test case, in which advantage in the number of iterations was not able to compensate the additional time spent for refinement. This was due to a very large number of predicates required for one particular program location. In future we want to investigate these extreme cases in order to develop a heuristic which would help to cope with them.

5. CONCLUSIONS AND FUTURE WORK

We presented a new approach to the abstraction refinement that combines precise and approximated techniques. On one hand, the proposed algorithm benefits from the precise component, because it avoids too many iterations due to spurious transitions of the abstract model. On the other hand, it uses the fast component to discover the spurious counterexample. Moreover, by exploiting the localized-abstraction framework, it reduces the abstraction computation to the parts of the system that are relevant to the property and keeps the approximated abstraction in all parts of the program that are irrelevant to prove the property. Our technique is independent of any particular abstraction or refinement procedure and can be used for any combination of the existing abstraction and refinement techniques.

We performed an extensive evaluation on large scale programs comparing the new technique with the classical precise and imprecise algorithms. Our tests with various benchmarks show that the new approach systematically outperforms both precise and imprecise techniques. Altogether it confirms that our new technique achieves the goal of reducing the number of iterations of the CEGAR loop.

In this paper, the goal of the experimental evaluation was to validate the new technique on spurious transition refinement. Thus, we maintained the same tool framework and we did not change orthogonal techniques such as predicate discovery. As a future work, we are interested in implementing the same approach in other tools such as BLAST [20] and in integrating it with interpolation-based approaches to predicate discovery [19, 24]. Another interesting direction is to investigate the same trade-off between precise and approximated approaches in the context of purely interpolation-based model checking [28] which does not need predicate abstraction.

6. REFERENCES

[1] T. Ball, B. Cook, S. Das, and S.K. Rajamani. Refining Approximations in Software Predicate Abstraction. In *TACAS*, pages 388–403, 2004.

[2] T. Ball, R. Majumdar, T.D. Millstein, and S.K. Rajamani. Automatic Predicate Abstraction of C Programs. In *PLDI*, pages 203–213, 2001.

[3] T. Ball, A. Podelski, and S.K. Rajamani. Boolean and Cartesian Abstraction for Model Checking C Programs. *STTT*, 5(1):49–58, 2003.

[5] Complete version of results as well as tools and examples are available at http://www.verify.inf.unisi.ch/projects/synergy.

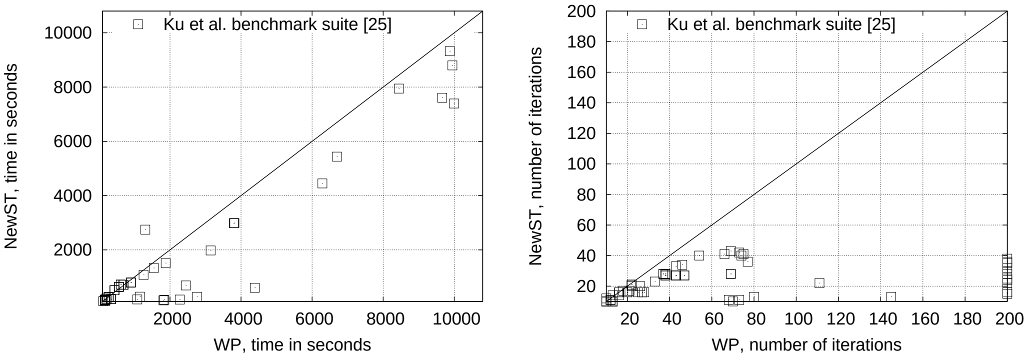

Figure 3: Comparison of time in seconds (right) and number of iterations (left) used by WP and NewST to verify benchmark suite [25]

[4] T. Ball and S.K. Rajamani. Boolean Programs: A Model and Process for Software Analysis. Technical Report 2000-14, Microsoft Research, February 2000.

[5] T. Ball and S.K. Rajamani. Generating Abstract Explanations of Spurious Counterexamples in C Programs. Technical Report 2002-09, Microsoft Research, September 2002.

[6] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[7] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.

[8] R. Cavada, A. Cimatti, A. Franzén, K. Kalyanasundaram, M. Roveri, and R. K. Shyamasundar. Computing Predicate Abstractions by Integrating BDDs and SMT solvers. In *FMCAD*, pages 69–76. IEEE, 2007.

[9] E. Clarke, M. Talupur, H. Veith, and D. Wang. SAT Based Predicate Abstraction for Hardware Verification. In *SAT*, 2003.

[10] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *CAV*, pages 154–169, 2000.

[11] E.M. Clarke, O. Grumberg, and D.E. Long. Model Checking and Abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[12] E.M. Clarke, A. Gupta, J.H. Kukula, and O. Strichman. SAT Based Abstraction-Refinement Using ILP and Machine Learning Techniques. In *CAV*, pages 265–279, 2002.

[13] E.M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate Abstraction of ANSI-C Programs Using SAT. *Formal Methods in System Design*, 25(2-3):105–127, 2004.

[14] M. Colón and T.E. Uribe. Generating Finite-State Abstractions of Reactive Systems Using Decision Procedures. In *CAV*, pages 293–304, 1998.

[15] S. Das and D.L. Dill. Successive Approximation of Abstract Transition Relations. In *LICS*, pages 51–60, 2001.

[16] S. Das, D.L. Dill, and S. Park. Experience with Predicate Abstraction. In *CAV*, 1999.

[17] S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *CAV*, pages 72–83, 1997.

[18] A. Gupta and O. Strichman. Abstraction Refinement for Bounded Model Checking. In *CAV*, pages 112–124, 2005.

[19] T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from Proofs. In *POPL*, pages 232–244, 2004.

[20] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *POPL*, pages 58–70, 2002.

[21] H. Jain, D. Kroening, N. Sharygina, and E.M. Clarke. Word Level Predicate Abstraction and Refinement for Verifying RTL Verilog. In *DAC*, pages 445–450, 2005.

[22] Himanshu Jain, Franjo Ivancic, Aarti Gupta, and Malay K. Ganai. Localization and Register Sharing for Predicate Abstraction. In *TACAS*, pages 397–412, 2005.

[23] R. Jhala and K.L. McMillan. Interpolant-Based Transition Relation Approximation. In *CAV*, pages 39–51, 2005.

[24] R. Jhala and K.L. McMillan. A Practical and Complete Approach to Predicate Refinement. In *TACAS*, pages 459–473, 2006.

[25] Kelvin Ku, Thomas E. Hart, Marsha Chechik, and David Lie. A Buffer Overflow Benchmark for Software Model Checkers. In *ASE '07*, pages 389–392. ACM Press, 2007.

[26] S.K. Lahiri, T. Ball, and B. Cook. Predicate Abstraction via Symbolic Decision Procedures. *Logical Methods in Computer Science*, 3(2), 2007.

[27] S.K. Lahiri, R. Nieuwenhuis, and A. Oliveras. SMT Techniques for Fast Predicate Abstraction. In *CAV*, LNCS, pages 424–437. Springer, 2006.

[28] Kenneth L. McMillan. Lazy abstraction with interpolants. In *CAV*, pages 123–136, 2006.

[29] K.L. McMillan. Applying SAT Methods in Unbounded Symbolic Model Checking. In *CAV*, pages 250–264, 2002.

[30] Flemming Nielson, Hanne Riis Nielson, and Chris L. Hankin. *Principles of Program Analysis*. Springer, 1999.