Counterexamples with Loops for Predicate Abstraction[*]

Daniel Kroening and Georg Weissenbacher[**]

Computer Systems Institute, ETH Zurich, 8092 Zurich, Switzerland
{daniel.kroening, georg.weissenbacher}@inf.ethz.ch

Abstract. Predicate abstraction is a major abstraction technique for the verification of software. Data is abstracted by means of Boolean variables, which keep track of predicates over the data. In many cases, the technique suffers from the fact that it requires at least one predicate for each iteration of a loop construct in the program. We propose to extract *looping counterexamples* from the abstract model, and to parameterize the simulation instance in the number of loop iterations.

1 Introduction

Software Model Checking [1] promises an automatic way to discover flaws in large computer programs. Despite of this promise, software model checking techniques are applied rarely, as software verification tools lack scalability due to the state-space explosion problem.

Abstraction techniques map the original, concrete set of states to a smaller set of states in a way that preserves the property of interest. Predicate abstraction is one of the most popular and widely applied methods for systematic state-space reduction of programs [2]. This technique is promoted by the success of the SLAM project [3,4]. SLAM is used to show lightweight properties of Windows device drivers, and predicate abstraction enables SLAM to scale to large instances.

In predicate abstraction, data is abstracted by keeping track of certain predicates over the data. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated. The resulting Boolean program is an over-approximation of the original program. One starts with a coarse abstraction, and if it is found that an error-trace reported by the model checker is not realistic, the error trace is used to refine the abstract program, and the process proceeds until no spurious error traces can be found [5]. The actual steps of the loop follow the *abstract-verify-refine* paradigm [6]. A second well-known implementation of this method is the software model checker BLAST [7].

In many cases, the technique suffers from the fact that it requires at least one predicate for each iteration of a loop construct in the program. This is due to the fact that the simulation and refinement phases are ignorant of program loops.

[*] This research is supported in part by an award from IBM Research.

[**] Supported by a Microsoft Research European PhD scholarship.

The existing simulation techniques exactly simulate as many loop iterations as contained in the abstract trace. Most of the existing refinement techniques correspond to performing one more unwinding of the loop.

The information about looping structures is actually contained within the abstract model $\hat{M}$. However, the model checkers for $\hat{M}$ never output error traces with loops, as they aim at counterexamples that are as short as possible.

Contribution. We propose a novel predicate abstraction algorithm that makes two contributions:

1. We extend the abstraction refinement framework with the concept of abstract counterexamples that contain (possibly nested) loops. We add the capability to compute such counterexamples to Boppo [8], a symbolic model checker for Boolean programs. The computation is done by means of a propositional SAT solver.
2. We describe a two-phase algorithm for simulating such a *looping counterexample* on the concrete model. The first phase attempts to compute a number n that corresponds to the number of loop iterations necessary to reach an error state. It is built using closed form solutions of recurrences and over-approximates the program. The second phase is a conventional simulation with n unwindings of the loop, which rules out spurious counterexamples. The predicates contained in the equation built for the first phase are used to improve the refinement in case the trace is spurious.

We report experimental results, which demonstrate that that our algorithm improves the performance significantly for benchmarks where a conventional abstraction refinement implementation has to perform repeated refinement steps to unroll the loop.

Related Work. The Newton tool is used by the Slam toolkit to decide the feasibility of counterexamples and to generate new predicates in order to refine the abstraction [9]. Newton is limited to finite counterexamples without loops. Therefore, Slam suffers from the problem described above.

Path Slicing is an approach that shortens counterexamples by dropping statements that have no impact on the reachability of the program location in question [10]. The statements and branches that can be bypassed are eliminated by backward slicing: For each program location, the set of relevant variables whose valuations at that point determine whether or not the error location is reachable is computed. The feasibility of a path slice implies the feasibility of the original counterexample, but assumes termination of the omitted code sequences.

Path slicing eliminates loops during the symbolic simulation if and *only* if they do *not* contribute to the reachability of the error location. Therefore, path slicing is orthogonal to the approach that we present, since it prevents expensive unrolling of loops that are not related to the error.

Linear programs have been proposed by Armando as an alternative, finer grained formalism for abstractions of sequential programs [11]. Due to the higher expressiveness of linear programs (in comparison to Boolean programs), this approach may yield a smaller number of spurious execution traces. However, the abstraction algorithm is restricted to a pointer-less subset of the C programming language that employs linear arithmetics and arrays [12].

Rybalchenko and Podelski present a complete method for detecting linear ranking functions of unnested program loops [13]. The inferred ranking function poses an upper bound for the iterations of the loop. This bound is not necessarily tight. Combined with abstraction-refinement, this approach enables proofs of program termination [14]. A proof of termination is insufficient to show the feasibility of counterexamples with loops, since the violation of the property usually depends on the number of iterations. Therefore, we utilize an incomplete method that provides the exact number of loop iterations necessary to reach the error state.

Linear algebra can be used for an inter-procedural program analysis that computes all affine relations which are valid at a program point [15]. The analysis presented by Müller-Olm interprets all assignment statements with affine expressions on the right hand side, while all other assignments are considered to be non-deterministic. It infers all linear and polynomial relations (up to a given degree). The approach is control-flow insensitive and cannot be used to decide reachability. The relations over the induction variables of a loop could aid the computation of the number of loop iterations that makes a counterexample feasible.

Zhang provides a sufficient condition for infinite looping and uses constraint solving techniques to detect infinite loops [16]. The method is sound, but not complete, since it is based on deciding theorems that involve non-linear integer arithmetic. The only goal of this approach is the detection of infinite loops. Feasibility of terminating loops is not discussed. Furthermore, nested loops are not considered.

Van Engelen presents an analysis method for dependence testing in the presence of nonlinear and non-closed array index expressions and pointer references [17]. His work is discussed in more detail in context with our loop simulation algorithm in Section 4. Van Engelen's approach targets compiler optimization, while our approach aims at feasibility checking and refinement.

Outline. The paper is organized as follows. Section 2 provides background on predicate abstraction refinement for software programs. The contribution of this paper is in Sections 3 to 4. Section 3 describes the syntax and semantics of looping abstract counterexamples. The simulation of such counterexamples on the concrete program is illustrated in Section 4. Experimental results are provided in Section 5.

2 Background

2.1 Predicate Abstraction and Refinement

Figure 1 shows an overview of counterexample-guided abstraction refinement. We provide background on each of the four steps of the loop.

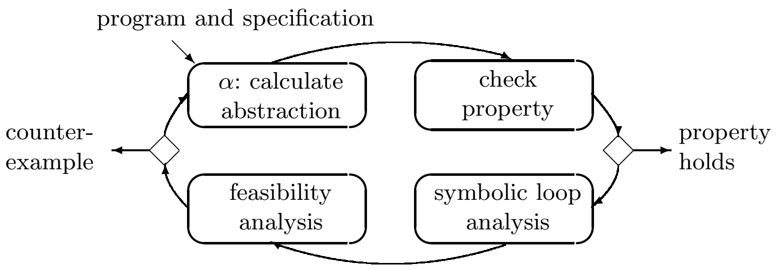


Fig. 1. Counterexample-guided abstraction refinement with two-phase simulation

Abstraction. The concrete model M is mapped to an abstract model $\hat{M}$ by means of an abstraction function α. The abstraction function α maps concrete states $s \in S$ to abstract states $\hat{s} \in \hat{S}$. We use γ to denote α^{-1}, which maps an abstract state back to a set of corresponding concrete states. Existential abstraction [18] is a reachability preserving transformation that guarantees that the abstract transition relation $\xrightarrow{a}$ is an over-approximation of $\xrightarrow{c}$, the transition relation of the original program. For reasons of efficiency, most implementations also over-approximate $\xrightarrow{a}$.

Given a set of predicates P, a *predicate abstraction* $\alpha_P(\varphi)$ is the strongest Boolean combination $\hat{\varphi}$ of these predicates such that φ implies $\hat{\varphi}$. The variables of the abstract state $\hat{s} \in \hat{S}$ correspond to the predicates in P, and their valuation is determined by $\hat{\varphi}$.

Verifying $\hat{M}$. The model checker for $\hat{M}$ searches the state space of $\hat{M}$ for states that violate a given specification. If no such state exists, the property holds on M, and the algorithm terminates. If an error state $\hat{s}_n$ exists, the model checker reports a counterexample that is a sequence of states $\hat{s}_1, \ldots, \hat{s}_n$ s.t. $\hat{s}_1$ is an initial state, $\hat{s}_i \xrightarrow{a} \hat{s}_{i+1}$ for each $i, 1 \leq i < n$, and $\hat{s}_n$ is an error state.

Bebop is a symbolic model checker for Boolean programs that is used in Slam to check the abstract model [19]. Boolean programs provide the same control flow constructs (including function calls) as C programs. Bebop uses BDDs as internal representation for states and features function summarization.

Moped is a BDD-based model checker for pushdown systems [20], which are as expressive as Boolean programs. Zing [21], an explicit-state model checker for concurrent programs, is used in an experimental version of Slam that provides support for the verification of concurrent programs [8].

Bebop, Moped, and Zing produce counterexamples $\hat{s}_1, \ldots, \hat{s}_n$ with the property $\hat{s}_i \neq \hat{s}_j$ for all $i \neq j$, since they aim at providing the shortest counterexample possible.

Simulation. An abstract counterexample $\hat{s}_1, \hat{s}_2, \ldots, \hat{s}_n$ is feasible in M iff there exists a corresponding sequence of concrete states $s_1, s_2, \ldots, s_n$ such that $s_i \in \gamma(\hat{s}_i)$ for $1 \leq i \leq n$ and there is a concrete transition $s_i \xrightarrow{c} s_{i+1}$ for $1 \leq i < n$.

Since *any* feasible concrete path serves our purpose, it is sufficient to demand that only the locations of corresponding states match. We give a formal definition of feasibility of counterexamples in terms of their *strongest postcondition* [22].

Definition 1 (Strongest Postcondition). *The strongest postcondition* SP *of a statement is defined as*

$$SP(x := e) = \lambda f.\exists x'.f[x'/x] \wedge (x = e[x'/x])$$
$$SP(e) = \lambda f.f \wedge e$$

where $e[x'/x]$ denotes the substitution of all free occurences of x in e by x'.

Let $\ell(\hat{s}_i)$ denote the program location that is part of the abstract state $\hat{s}_i$, and let σ_i denote the concrete statement corresponding to $\ell(\hat{s}_i)$. The strongest postcondition for the sequence of statements $\sigma_1, \ldots, \sigma_n$ is $SP(\sigma_1, \ldots, \sigma_n) := SP(\sigma_n) \circ SP(\sigma_{n-1}) \circ \ldots \circ SP(\sigma_1)$. The resulting quantifiers can be eliminated by means of skolemization. Intuitively, this corresponds to a transformation of the path into *single static assignment* form (SSA) [23]. The formula $SP(\sigma_1, \ldots, \sigma_n)(\texttt{true})$ represents all states that are reachable by executing the statements on the path $s_1, \ldots, s_n$.

Definition 2 (Feasibility of Counterexamples). *A counterexample is feasible iff $SP(\bar{\sigma})(\texttt{true})$ is satisfiable for the corresponding sequence $\bar{\sigma}$ of concrete statements. A counterexample is* spurious *if it has an infeasible prefix.*

Newton uses a general purpose Nelson-Oppen style theorem prover to determine the feasiblity of counterexamples. Our model checker SatAbs [24] translates the strongest postconditions into Boolean formulas and uses an incremental SAT solver to decide the SAT instances that result from unwinding the path.

Refinement. If the simulation yields a spurious counterexample p, $\hat{M}$ is refined such that p is removed from $\hat{M}$. This is done by adding an appropriate set of predicates. Newton uses heuristics to extract such predicates from $SP(p)$. McMillan observed that for each cut point of the path there exists a formula ψ (called the *Craig interpolant*) that represents precisely the facts that need to be known between σ_i to σ_{i+1} to prove infeasibility [25]. This approach is implemented in Blast. A preliminary analysis identifies a number of promising cut points. The resulting interpolants are then used as new predicates. Both Newton and Blast are unaware of loops and handle unrolled loops the same way as counterexamples that do not contain iterations.

2.2 Abstracting Programs with Loops

The traditional abstraction-refinement scheme with predicate abstraction performs poorly on programs that contain loops as shown in Figure 2. Slam, Blast, and previous versions of SatAbs need at least 1000 refinement steps that successively add predicates over the loop counter (as indicated in Figure 3) to produce a feasible counterexample. We present a detection algorithm for loops contained in the abstract model in Section 3 and a novel two-phased simulation approach in Section 4.

```
int i, s = 0;
int a[1000];
for (i = 0; i ≤ 1000; i = i + 1) {
    assert(i < 1000);
    s = s + a[i]; }
```

Fig. 2. A simple program with a buffer overflow

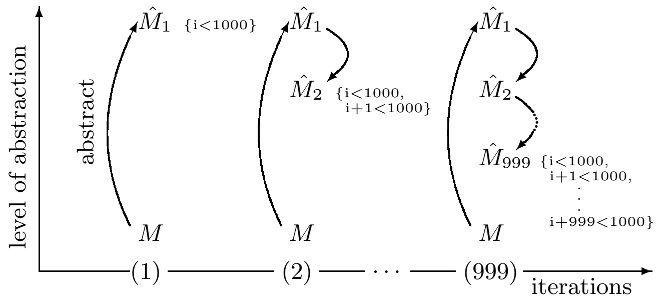


Fig. 3. Iterative abstraction refinement for the program in Figure 2

3 Abstract Counterexamples with Loops

Counterexamples with Loops. Consider the Boolean Program in Figure 4(a): It is the abstraction of the program in Figure 2 with respect to the assertion predicate ($i < 1000$) and the loop condition ($i \leq 1000$). For this program, all model checkers listed in Section 2 report the spurious counterexample 4(b). An inspection of the abstract model reveals that $\hat{M}$ contains a path with a potential iteration that traverses the same program locations as the spurious counterexample. Figure 4(c) shows a variant of the counterexample. The *repetition signs* ‖: and :‖ indicate that the sequence of enclosed states can be iterated arbitrarily often. The sequence of states to the right of the loop denotes the path that can be taken to reach the error state.

Figure 5 shows the structure of the counterexample 4(c). Each iteration of the loop visits the same program locations. Due to the non-deterministic assignment at location `L5`, the final iteration traverses a different sequence of states than the previous iterations. The counterexample in Figure 5 represents an *infinite set* of conventional counterexamples, one of which corresponds to the feasible path that violates the assertion in Figure 2 after 1000 iterations.

We define the semantics of a counterexample with loops in terms of the infinite set of conventional counterexamples it represents (Figure 6). We use the following notation: The double square brackets $[\![path]\!]$ denote the *expansion* of a path. The state indicated by $path[i]$ is the i^{th} element of *path*. The function length(*path*) returns the number of states in a *path* without loops. The expression $(path_r)^* path$ denotes *all* paths that contain an arbitrary number of repetitions of $path_r$ followed by the postfix *path*. The concatenation operation $A^\frown B$ denotes all concatenations of each path p_a in set A with each path p_b in set B for which

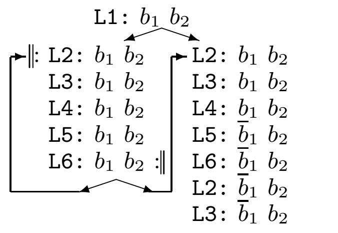
(a) Boolean program	(b) Counterexample	(c) Counterexample with loop
`    bool` b_1`; /*` $i < 1000$ `*/`	`L1:` $b_1\ b_2$	 L1: $b_1\ b_2$ ‖: L2: $b_1\ b_2$ L3: $b_1\ b_2$ L4: $b_1\ b_2$ L5: $b_1\ b_2$ L6: $b_1\ b_2$:‖ L2: $b_1\ b_2$ L3: $b_1\ b_2$ L4: $b_1\ b_2$ L5: $\overline{b_1}\ b_2$ L6: $\overline{b_1}\ b_2$ L2: $\overline{b_1}\ b_2$ L3: $\overline{b_1}\ b_2$
`    bool` b_2`; /*` $i \leq 1000$ `*/`	`L2:` $b_1\ b_2$	
`L1:` b_1, b_2`:=1,1;`	`L3:` $b_1\ b_2$	
`L2: if (!`b_2`) goto L7;`	`L4:` $b_1\ b_2$	
`L3: assert (`b_1`);`	`L5:` $\overline{b_1}\ b_2$	
`L4: skip;`	`L6:` $\overline{b_1}\ b_2$	
`L5:` b_1, b_2`:=*,*;`	`L2:` $\overline{b_1}\ b_2$	
`L6: goto L2;`	`L3:` $\overline{b_1}\ b_2$	
`L7: skip;`		

Fig. 4. Enriching counterexamples with information about loops

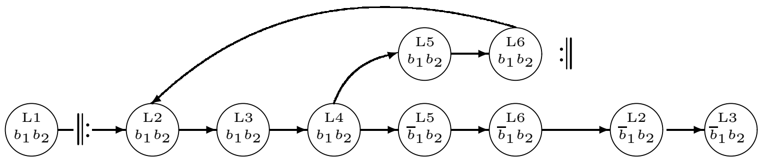


Fig. 5. Counterexample with loop for Figure 4(a)

there is an abstract transition from the last state of p_a to the first state of p_b. Note that the recursive syntax definition enables nested loops.

Definition 3 (Feasibility of counterexamples with loops). *An abstract counterexample p with loops is feasible iff $[\![p]\!]$ contains a path that is feasible according to Definition 2.*

Detection of Loops. A counterexample with loops can be constructed from a conventional counterexample $p = \hat{s}_1, \ldots, \hat{s}_n$ by performing a symbolic simulation of the abstract model along the locations $\ell(\hat{s}_1), \ldots, \ell(\hat{s}_n)$. At each location $\ell(\hat{s}_i)$ in p we search for a state $\hat{s}_j, j < i$ that allows us to fork a path that traverses the locations $\ell(\hat{s}_j), \ldots, \ell(\hat{s}_i)$ and then returns to $\hat{s}_j$. Figure 7 shows the pseudo code for this algorithm. The number of decision problems generated by this algorithm is quadratic in the length of the original path.

This loop detection algorithm obviously fails to compute all loops along p that are contained in $\hat{M}$. It misses loops that do not repeatedly visit the same state at the head of the loop. Furthermore, it (intentionally) does not detect loops that traverse different locations (e.g., branches of a conditional statement) in each iteration. Note that the latter kind of loop does not conform to the semantics given in Figure 6.

In both cases, the abstraction-refinement scheme is still sound. Any feasible counterexample that our loop detection misses is eventually found in a later iteration. Refinement boils down to successive unrolling of loops that are not detected. Thus, we either obtain a conventional counterexample, or the repetitive concatenation of the loop body results in an abstract loop that matches the criteria of the loop detection algorithm.

Syntax	Semantics
$path \to$ **state**	$[\![\Vert\!:path:\!\Vert]\!] \quad = [\![(path_r)^* path_p]\!]$,
$\mid$ '$\Vert\!:$' $path$ '$:\!\Vert$'	foreach $path_p \in [\![path]\!]$
$\mid$ $path\ path$	with $path_r$ such that
	length$(path_r)$=length$(path_p)\wedge$
	$\forall i \in \{1, \ldots, \text{length}(path_r)\}.$
	$\ell(path_r[i]) = \ell(path_p[i])$
	$[\![path_1 path_2]\!] = [\![path_1]\!] ^\frown [\![path_2]\!]$,
	where $[\![A]\!]^\frown [\![B]\!]$ denotes
	$\{p_a p_b \mid\, p_a \in [\![A]\!] \wedge p_b \in [\![B]\!] \wedge$
	$p_a[\text{length}(p_a)] \xrightarrow{a} p_b[1]\}$

Fig. 6. Syntax and semantics of abstract counterexamples with loops

```
FindLoops($\hat{s}_1, \ldots, \hat{s}_n$)
foreach $i \in \{1, \ldots, n\}$, $j < i$:
    if $\exists \hat{s}'_j, \ldots, \hat{s}'_i.\ \forall k \in \{j, \ldots, i\}.\ell(\hat{s}'_k) = \ell(\hat{s}_k) \wedge$
        $\forall k \in \{j, \ldots, i-1\}.\hat{s}'_k \xrightarrow{a} \hat{s}'_{k+1} \wedge\ \hat{s}'_j = \hat{s}_j \wedge \hat{s}'_i \xrightarrow{a} \hat{s}'_j$
    then insert $\Vert\!:\ \hat{s}'_j, \ldots, \hat{s}'_i\ :\!\Vert$
return counterexample $\hat{s}_1, \ldots, \hat{s}_n$ with loops
```

Fig. 7. Pseudo code for loop detection

Our approach does not necessarily benefit from a more agressive loop detection algorithm. Our experiments indicate that it is advantageous to keep the number of loops in a counterexample small, since the simulation of concrete loops is expensive.

We have implemented the algorithm of Figure 7 in Boppo. Boppo is a symbolic model checker for asynchronous Boolean programs. The Boolean program is translated to a propositional formula (function calls are inlined) and a SAT solver is used to perform reachability checking. Each decision problem of the loop detection algorithm corresponds to a SAT instance. The average overhead of the loop detection compared to the model checking run itself is below one percent[1].

4 Simulation and Refinement with Loops

The strongest postcondition presented in Definition 2 gives us only a semi-decision procedure for the feasibility of counterexamples with loops (namely, successive enumeration of all corresponding conventional counterexamples). We propose a new two-phase simulation semi-decision procedure for feasibility (see

[1] This number is based on benchmarking 489 typical Boolean programs between 26 and 656 lines of code that were generated by Slam.

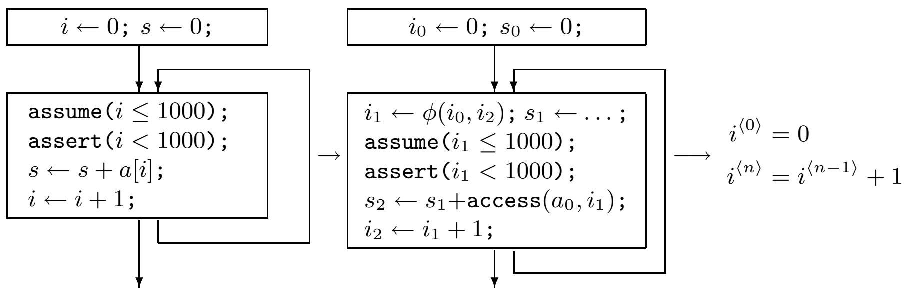

$i \leftarrow 0;\ s \leftarrow 0;$
assume($i \leq 1000$);
assert($i < 1000$);
$s \leftarrow s + a[i]$;
$i \leftarrow i + 1$;
$i_0 \leftarrow 0;\ s_0 \leftarrow 0;$
$i_1 \leftarrow \phi(i_0, i_2);\ s_1 \leftarrow \ldots;$
assume($i_1 \leq 1000$);
assert($i_1 < 1000$);
$s_2 \leftarrow s_1$+access(a_0, i_1);
$i_2 \leftarrow i_1 + 1$;
$i^{\langle 0 \rangle} = 0$
$i^{\langle n \rangle} = i^{\langle n-1 \rangle} + 1$

Fig. 8. Transforming a simple loop into a recurrent equation via SSA

Figure 1) of a counterexample p with loops. In the first phase, a heuristic is applied to pick a promising conventional counterexample p_c out of $[\![p]\!]$. In the second phase we check the feasibility of p_c using the traditional approach.

Simulation. The *symbolic loop analysis* phase provides a candidate n for the number of feasible iterations for each loop in the counterexample. The path is infeasible if no such n exists. The converse does not hold. Starting with the innermost loop, we parameterize each loop body with a fresh variable n using following algorithm:

1. Transform the loop into SSA form.
2. Generate a recurrence equation for each variable that is updated by a ϕ function.
3. Calculate the closed form of the recurrence equation (if possible). Substitute its right-hand-side for the corresponding occurrences of the variable (this step is known as *induction variable substitution* [17]). If unable to compute the closed form, assign the variable non-deterministically.
4. Generate the strongest postcondition of the loop body and existentially quantify n in the resulting formula.

Example 1. Consider once more the program in Figure 2. The loop in Figure 8 represents the set of concrete paths that corresponds to the looping counterexample in Figure 4. We transform the loop into SSA and obtain the recurrent equation in Figure 8. The closed form[2] of this recurrence is $i^{\langle n \rangle} = i^{\langle 0 \rangle} + 1 \cdot n$.

Therefore, SatAbs replaces every occurrence of i_1 with $i_0 + 1 \cdot n$. By applying SP and quantifying n we obtain

$$\begin{aligned} SP(\mathit{loop}) = \lambda f.\exists n.\exists s_0'\ i_2'.f[s_0'/s_0][i_2'/i_2] \\ \wedge ((i_0 + 1 \cdot n) \leq 1000) \wedge \neg((i_0 + 1 \cdot n) < 1000) \\ \wedge (s_0 = a_0[(i_0 + 1 \cdot n)]) \wedge (i_2 = (i_0 + 1 \cdot n) + 1) \end{aligned}$$

[2] The closed form for a recurrent equation $i^{\langle 0 \rangle} = \alpha, i^{\langle n \rangle} = i^{\langle n-1 \rangle} + \beta + \gamma n$, $n > 0$ (where α, β, γ are numeric constants or loop invariant symbolic expressions) is $i^{\langle n \rangle} = \alpha + \beta n + \gamma \frac{n \cdot (n+1)}{2}$.

Solving the SAT instance that corresponds to $(SP(loop) \circ SP(i_0 = 0))(\texttt{true})$ yields $n = 1000$. Note that there is only one valid solution for n, since $(i \leq 1000)$ is a sufficiently strong loop invariant. The weakest loop condition that does not change the program semantics is $(i \neq 1001)$ and gives us the choice $n \in \{1000, 1002, 1003, \ldots\}$. In our current implementation we have no influence on the n that the SAT solver reports in such a case. We consider to use an optimizing solver like PBS [26] in future versions of our tool to obtain the minimal values of n.

Our approach is not restricted to simple loop counters. Van Engelen provides a framework for handling affine, polynomial, and geometric index expressions composed over linear and non-linear induction variables [17]. These analysis methods and our simulation algorithm also cope with pointer arithmetic and arrays. However, our current implementation supports only a fixed simple recurrence scheme (namely the one presented in Example 1). We treat recurrences that have no closed form equivalent (e.g., $k^{\langle n\rangle} = i \cdot k^{\langle n-1\rangle} + 1$, where i is a linear induction variable) conservatively by introducing non-determinism (as explained in step 3 of our algorithm). The subsequent traditional simulation of the potentially spurious counterexample (see below) preserves soundness.

Example 2. Consider a function (e.g., as part of a library of combinatorial functions) that calculates the factorial m of a variable k by iterating over $i = \{0, \ldots, k\}$, $m = m \cdot (i + 1)$. Assume that the program contains a user-supplied assertion that the computation does not overflow. By substituting the right hand side of the closed form $i^{\langle n\rangle} = i^{\langle 0\rangle} + n$ for i one obtains $m^{\langle n\rangle} = m^{\langle n-1\rangle} \cdot (i^{\langle 0\rangle} + n + 1)$. The resulting recurrence is $m^{\langle n\rangle} = m^{\langle 0\rangle} \cdot \frac{i^{\langle 0\rangle}+n!}{i^{\langle 0\rangle}!}$.

On a 32 bit architecture, the overflow occurs at $k = 13$. This number is sufficiently small to use a bounded model checker (like CBMC [27]) to simulate the counterexample. For this reason, our current implementation ignores recurrence equations with a closed form that is a fast-growing monotonic function of n (e.g., $n!$ as in our example, or exponentiation with positive integer exponent or base). In this case, SatAbs uses the standard abstraction-refinement algorithm instead of computing a solution for n. The bit-level accurate simulation algorithm of SatAbs guarantees that an eventual overflow will be detected.

Generating Concrete Counterexamples. The symbolic loop analysis is followed by a traditional feasibility analysis (see Figure 1). Each loop of the counterexample is unrolled according to the results of the previous step. As usual, feasible counterexamples are reported to the user. The fact that they are annotated with information about loops makes them more readable. Spurious counterexamples are subject to refinement.

Refinement. We distinguish two causes of infeasibility of the spurious counterexample p:

- There is no such n that satisfies the recurrence, i.e., phase I reports the corresponding SAT instance to be unsatisfiable. Then we can refine $\hat{M}$ using a set of predicates that remove all paths $[\![p]\!]$ from $\hat{M}$. The unsatisfiability of formula $\varphi_1 \wedge \varphi_2^{\langle n \rangle} \wedge \varphi_3$ (where φ_1 corresponds to the prefix, $\varphi_2^{\langle n \rangle}$ to the parameterized loop body, and φ_3 to the tail of p) is an explanation for the infeasibility of p. Since no n satisfies the formula, setting n to 0 yields an infeasible counterexample from which we can extract a set of refinement predicates using the traditional methods presented in Section 2.

– The traditional feasibility analysis (phase II) refutes $\varphi_1 \wedge \varphi_2^{\langle c \rangle} \wedge \varphi_3$ for the particular constant $n = c$ obtained from phase I. That means that the recurrences $\varphi_2^{\langle n \rangle}$ are not sufficiently strong to show the infeasibility of all paths $[\![p]\!]$. Therefore, we compute a set of refinement predicates from the unrolled path that corresponds to $\varphi_1 \wedge \varphi_2^{\langle c \rangle} \wedge \varphi_3$. This guarantees that the execution of c iterations of the loop is infeasible in $\hat{M}$ and that the same loop is not detected again. We expect that the recurrences are loop invariants that make spurious counterexamples other than p abstractly infeasible, too. Therefore, we consider adding the corresponding predicates even if they have no effect on the feasibility of p.

5 Experimental Results

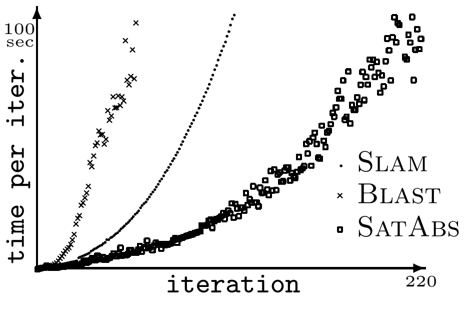
As expected, our implementation detects the buffer overflow in Figure 2 after only one iteration. The attempt to run BLAST and SATABS without loop detection on the same problem did not yield any results in reasonable time, but exposed an exponential increase of the runtime in every refinement step.

Figure 9 shows a buffer overflow in the Linux mail transfer agent AEON 0.02a. This bug allows local users to gain administrator privileges by executing malicious byte code with help of an overly long `HOME` environment variable (US-CERT CVE-2005-1019). The function `getConfig` is called immediately after the program is started and copies the string returned by `getenv` to a buffer of (fixed)

```
/* reading rc file, handling missing options */
int getConfig(char settings[MAX_SETTINGS][MAX_LEN]) {
      char home[MAX_LEN];
      FILE *fp;                          /* .rc file handler */
      int numSet = 0;                    /* number of settings */
      strcpy(home, getenv("HOME"));      /* get home path */
      strcat(home, "/.aeonrc");          /* full path to rc file */

char* strcpy (char *t, const char* s) {
      for (i = 0 ;; i++) { assert (!(t == &home)||!(i>=MAX_LEN));
           t[i] = s[i]; if (s[i] == '\0') break; }}
```

Fig. 9. Buffer overflow in AEON 0.2a



Runtime per iteration (Aeon)

MAX_LEN	Blast	Slam	SatAbs	+loops
25	161.1	44.0	57.7	25.0
50	1477.4	294.9	182.9	28.0
75	-	993.6	402.9	32.8
100	-	2446.0	765.0	34.1
150	-	9130.2	2241.9	50.3
200	-	23803.5	5402.9	55.8
300	-	-	18702.4	97.6
512	-	-	-	254.5

Total runtime (in sec) for given `MAX_LEN`

Fig. 10. Runtime of Blast, Slam, SatAbs and SatAbs with loop detection (Aeon)

size `MAX_LEN` without checking its bound (see line 5). This error is representative for many buffer overflows and is detected by SatAbs with loop detection in one iteration.

The automatic verification condition generator of SatAbs adds the assertion `!(t == &home)||!(i>=MAX_LEN)` to the loop body of `strcpy` (see line 2 in Figure 9). Note that SatAbs does not specifically target buffer overflows, but aims at verifying arbitrary assertions in C programs. We manually added a corresponding assertion to the Aeon sources to make a comparison with Blast and Slam possible. Our attempts to detect the bug with Blast, Slam and SatAbs without loop detection failed despite a generous timeout of 25000 seconds. Therefore, we reduced the value of `MAX_LEN` (which is 512 in the original program) and compared the performance of Blast, Slam, SatAbs without loop detection, and SatAbs with loop detection. The results of this benchmark[3] are given in Figure 10. The table gives the runtime of all four tools for various values of `MAX_LEN`. As expected, the runtime of Slam grows exponentially with the size of the buffer. Blast crashes for `MAX_LEN`= 75. We did not further investigate this problem. SatAbs performs slightly better than Slam[4], but the runtime still increases exponentially with the number of iterations. The diagram in Figure 10 illustrates the exponential increase of the runtime in each abstraction-refinement iteration. We compared the runtime of all iterations that took less than 100 seconds.

SatAbs *with* loop detection spends most of the time in the simulation of the unrolled counterexample. This is because SatAbs performs SAT-based bit-level accurate simulation (unlike Slam and Blast, which model integer variables as unbounded integers). We listed the results for all four tools in the table in Figure 10.

[3] All our experiments were done on an Intel Pentium 4 with 3 GHz and 2 GB RAM.
[4] We adapted the refinement strategy of SatAbs (with respect to spurious paths and spurious transitions [28]) to match the behaviour of Slam and Blast.

We refrain from presenting other benchmarks in favor of the in depth description of the Aeon example. The SatAbs executable and more examples can be downloaded from `http://www.inf.ethz.ch/personal/daniekro/satabs/`.

6 Conclusion

This paper presents a novel approach that enables predicate abstraction to find bugs that emerge as a result of a high number of iterations of loops. We propose an algorithm to detect loops in abstract models and explain how the traditional simulation and refinement algorithms can be extended to cope with loops. Our implementation outperforms the abstraction-refinement based verification tools Blast and Slam on typical buffer overflow examples.

Currently, our implementation recognizes only basic recurrences that are sufficient to find the most common bugs. An integration of the recurrence solving algorithms of van Engelen [17] can lift this limitation.

References

1. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
2. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: CAV, Springer (1997) 72–83
3. Ball, T., Rajamani, S.: Boolean programs: A model and process for software analysis. Technical Report 2000-14, Microsoft Research (2000)
4. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In: IFM, Springer (2004)
5. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV, Springer (2000) 154–169
6. Kurshan, R.: Computer-Aided Verification of Coordinating Processes. Princeton University Press (1995)
7. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL, ACM Press (2002) 58–70
8. Cook, B., Kroening, D., Sharygina, N.: Symbolic model checking for asynchronous Boolean programs. In: SPIN, Springer (2005) 75–90
9. Ball, T., Rajamani, S.: Generating Abstract Explanations of Spurious Counterexamples in C Programs. Technical Report MSR-TR-2002-09, Microsoft Research, Redmond (2002)
10. Jhala, R., Majumdar, R.: Path slicing. In: PLDI, ACM Press (2005) 38–47
11. Armando, A., Castellini, C., Mantovani, J.: Software model checking using linear constraints. In: IFCEM 2004: 6th Int. Conference on Formal Engineering Methods. (2004) 209–223
12. Armando, A., Benerecetti, M., Mantovani, J.: Model checking linear programs with arrays. In: SoftMC, Elsevier (2006) 79–94
13. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: VMCAI, Springer (2004) 239–25
14. Cook, B., Podelski, A., Rybalchenko, A.: Abstraction-refinement for termination. In: SAS, Springer (2005) 87–101
15. Müller-Olm, M., Seidl, H.: Precise interprocedural analysis through linear algebra. In: POPL, ACM Press (2004) 330–341

16. Zhang, J.: A path-based approach to the detection of infinite looping. In: APAQS: Asia-pacific conference on quality software, IEEE Computer Society (2001) 88–96
17. van Engelen, R.A., Birch, J., Shou, Y., Walsh, B., Gallivan, K.A.: A unified framework for nonlinear dependence testing and symbolic analysis. In: ICS: International conference on Supercomputing, ACM Press (2004) 106–115
18. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. In: POPL, ACM Press (1992) 343–354
19. Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for Boolean programs. In: SPIN, Springer (2000) 113–130
20. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: CAV, Springer (2000) 232–247
21. Andrews, T., Qadeer, S., Rajamani, S.K., Xie, Y.: Zing: Exploiting program structure for model checking concurrent software. In: CONCUR, Springer (2004) 1–15
22. Gries, D.: The Science of Programming. Springer (1987)
23. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems **13** (1991) 451–490
24. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI–C programs using SAT. FMSD **25** (2004) 105–127
25. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL, ACM Press (2004) 232–244
26. Aloul, F., Ramani, A., Markov, I., Sakallah, K.: PBS: A backtrack search pseudo-Boolean solver. In: Theory and Appl. of Satisfiability Testing. (2002) 346–353
27. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS, Springer (2004) 168–176
28. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and cartesian abstraction for model checking C programs. In: TACAS, London, UK, Springer-Verlag (2001) 268–283