Counterexample Guided Path Reduction for Static Program Analysis

Ansgar Fehnker, Ralf Huuck, and Sean Seefried

National ICT Australia Ltd. (NICTA)*
Locked Bag 6016
University of New South Wales
Sydney NSW 1466, Australia

Abstract. In this work we introduce counterexample guided path reduction based on interval constraint solving for static program analysis. The aim of this technique is to reduce the number of false positives by reducing the number of feasible paths in the abstraction iteratively. Given a counterexample, a set of observers is computed which exclude infeasible paths in the next iteration. This approach combines ideas from counterexample guided abstraction refinement for software verification with static analysis techniques that employ interval constraint solving. The advantage is that the analysis becomes less conservative than static analysis, while it benefits from the fact that interval constraint solving deals naturally with loops. We demonstrate that the proposed approach is effective in reducing the number of false positives, and compare it to other static checkers for C/C++ program analysis.

1 Introduction

Static program analysis and software model checking are two automatic analysis techniques to ensure (limited) correctness of software or at least to find as many bugs in the software as possible. In contrast to software model checking, static program analysis typically works on a more abstract level, such as the control flow graph (CFG) without any data abstraction. As such a syntactic model of a program is a very coarse abstraction, and reported error traces can be spurious, i.e. they may correspond to no actual run in the concrete program. The result of such infeasible paths are false positives in the program analysis.

This work presents counterexample guided path reduction to remove infeasible paths. To do so semantic information in the form of interval equations is added to a previously purely syntactic model. This is similar to abstract interpretation based interval analysis, e.g., used for buffer overflow detection. That approach transforms the entire programm into a set of interval equations, and characterizes its behavior by the precise least solution [1].

* National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs.

Our approach, in contrast, constructs this kind of equation system only for a given path through the program. This is not only computationally cheaper, but it has the advantage that the analysis becomes less conservative. The main reason for this improvement is that a path has fewer transitions and fewer join-operators, the main source for over-approximation errors, than the entire program. The paths themselves are determined by potential counterexamples. A counterexample path is *spurious* if the least solution of a corresponding equation system is empty. The subset of equations responsible for an empty solution is called a *conflict.* In a second step we refine our syntactic model by a finite *observer* which excludes all paths that generate the same conflict. These steps are applied iteratively, until either no new counterexample can be found, or until a counterexample is found that cannot be proven to be spurious.

This approach is obviously inspired by counterexample guided abstraction refinement (CEGAR), as used in [2, 3]. A main difference is the use of the *precise least solution* for interval equations [1] in the place of SAT-solving. This technique deals directly with loops, without the need to discover additional loop-predicates [3, 4], or successive unrolling of the transition relation [5]. An alternative to SAT-based CEGAR in the context of static analysis, using polyhedral approximations, was proposed in [6, 7]. Pre-image computations along the counterexamples are used to improve the accuracy of the polyhedral approximation. Our approach in contrast uses an *precise least solution* of an interval equation system, which is computationally faster, at the expense of precision. Our proposed approach combines nicely with the static analysis approach put forward in [8] and implemented in the tool Goanna. It defines, in contrast to semantic software model checking, syntactic properties on a syntactic abstraction of the program.

The experimental results confirm that the proposed path reduction technique situates our tool in-between software model checking and static analysis tools. False positives are effectively reduced while interval solving converges quickly, even in the presence of loops.

The next section introduces preliminaries of labeled transition systems, interval solving and model checking for static properties. Section 3 introduces Interval Automata, the model we use to capture the program semantics. Section 4 presents the details of our approach. Implementation details and a comparison with other tools are given in Section 5.

2 Preliminaries

2.1 Labeled Transition Systems

In this work we use *labeled transition systems* (LTS) to describe the semantics of our abstract programs. An LTS is defined by (S, S_0, A, R, F) where S is a set of states, $S_0 \subseteq S$ is a sub-set of initial states, A is a set of actions and $R \subseteq S \times A \times S$ is a transition relation where each transition is labeled with an action $a \in A$, and $F \subseteq S$ is a set of final states. We call an LTS *deterministic* if

for every state $s \in S$ and action $a \in A$ there is at most one successor state such that $(s, a, s') \in R$.

The finite sequence $\rho = s_0 a_0 s_1 a_1 \dots a_{n-1} s_n$ is an *execution* of an LTS $P = (S, S_0, A, R, F)$, if $s_0 \in S_o$ and $(s_i, a_i, s_{i+1}) \in R$ for all $i \geq 0$. An execution is *accepting* if $s_n \in F$. We say $w = a_0 \dots a_{n-1} \in A^*$ *is a word in* P, if there exist s_i, $i \geq 0$, such that $s_0 a_0 s_1 a_1 \dots a_{n-1} s_n$ form an execution in P. The *language* of P is defined by the set of all words for which there exists an accepting execution. We denote this language as $\mathcal{L}_P$.

The *product* of two labeled transition systems $P_1 = (S_1, S_{1_0}, A, R_1, F_1)$ and $P_2 = (S_2, S_{2_0}, A, R_2, F_2)$, denoted as $P_\times = P_1 \times P_2$, is defined as $P_\times = (S_1 \times S_2, S_{1_0} \times S_{1_0}, A, R_\times, F_1 \times F_2)$ where $((s_1, s_2), a, (s_1', s_2')) \in R_\times$ if and only if $(s_1, a, s_1') \in R_1$ and $(s_2, a, s_2') \in R_2$. The language of $P_\times$ is the *intersection* of the language defined by P_1 and P_2.

2.2 Interval Equation Systems

We define an *interval lattice* $\mathcal{I} = (I, \subseteq)$ by the set $I = \{\emptyset\} \cup \{[z_1, z_2] | z_1 \in \mathbb{Z} \cup \{-\infty\}, z_2 \in \mathbb{Z} \cup \{\infty\}, z_1 \leq z_2\}$ with the partial order implied by the *contained in* relation "$\subseteq$", where a non-empty interval $[a, b]$ is *contained in* $[c, d]$, if $a \geq c$ and $b \leq d$. The empty element is the bottom element of this lattice, and $[-\infty, +\infty]$ the top element. For dealing with interval boundaries, we assume that $\leq$, $\geq$, $+$, $*$ as well as min and max are extended in the usual way to the infinite range. Moreover, we consider the following operators on intervals: intersection $\sqcap$, union $\sqcup$, addition $(+)$ and multiplication $(\cdot)$ with the usual semantics $[\![.]\!]$for intersection and

$$
\begin{aligned}
[\![[l_1, u_1] \sqcup [l_2, u_2]]\!] &= [\min\{l_1, l_2\}, \max\{u_1, u_2\}] \\
[\![[l_1, u_1] + [l_2, u_2]]\!] &= [l_1 + l_2, u_1 + u_2] \\
[\![[l_1, u_1] \cdot [l_2, u_2]]\!] &= [\min(\mathit{product}), \max(\mathit{product})]
\end{aligned}
$$

where $\mathit{product} = \{l_1 * l_2, l_1 * u_2, l_2 * u_1, u_1 * u_2\}$.

For a given finite set of variables $X = \{x_0, \dots, x_n\}$ over $\mathcal{I}$ we define an *interval expression* ϕ as follows:

$$\phi \doteq a \mid x \mid \phi \sqcup \phi \mid \phi \sqcap \phi \mid \phi + \phi \mid \phi \cdot \phi$$

where $x \in X$, and $a \in \mathcal{I}$. The set of all expression over X is denoted as $\mathcal{C}(X)$.

For all operation we have that $[\![\phi \circ \varphi]\!]$ is $[\![\phi]\!] \circ [\![\varphi]\!]$, where $\circ$ can be any of $\sqcup, \sqcap, +, \cdot$. A *valuation* is a mapping $v : X \to \mathcal{I}$ from an interval variable to an interval. Given an interval expression $\phi \in \mathcal{C}(X)$, and a valuation v, the $[\![\phi]\!]_v$ denoted the expression ϕ evaluated in v, i.e. it is defined to be the interval $[\![\phi[v(x_0)/x_0, \dots, v(x_n)/x_n]]\!]$, which is obtained by substituting each variable x_i with the corresponding interval $v(x_i)$.

An *interval equation system* is a mapping $IE : X \to \mathcal{C}(X)$ from interval variables to interval expressions. We also denote this by $x_i = \phi_i$ where $i \in 1, \ldots, n$. The *solution* of such an interval equation system is a valuation satisfying all equations, i.e., $[\![x_i]\!] = [\![\phi_i]\!]_v$ for all $i \in 1, \ldots, n$. As shown in [1] there always is a *precise least solution* which can be efficiently computed. By precise we mean precise with respect to the interval operators's semantics and without the use of additional widening techniques. Of course, from a program analysis point of view over-approximations are introduced, e.g., when joining two intervals $[1, 2] \sqcup [4, 5]$ results in $[1, 5]$. This, however, is due to the domain we have chosen.

2.3 Static Analysis by Model Checking

This work is based on an automata based static analysis framework as described in [8], which is related to [9–11]. The basic idea of this approach is to map a C/C++ program to its CFG, and to label this CFG with occurrences of syntactic constructs of interest. The CFG together with the labels can easily be mapped to the input language of a model checker, in our case NuSMV, or directly translated into a Kripke structure for model checking.

A simple example of this approach is shown in Fig. 1. Consider the contrived program `foo` which is allocating some memory, copying it a number of times to `a`, and freeing the memory in the last loop iteration.

One example of a property to check is whether after freeing some resource, it still might be used. In our automata based approach we syntactically identify program locations that allocate, use, and free resource p. We automatically label the program's CFG with this information as shown on the right hand side of Fig. 1. This property can then be checked by the CTL property

$$AG\ (malloc_p \Rightarrow AG\ (free_p \Rightarrow \neg EF\ used_p)),$$

which means that whenever there is `free` after `malloc` for a resource p, there is no path such that p is used later on. Obviously, neglecting any further semantic information will lead to a false alarm in this example.

3 Interval Automata

This section introduces *interval automata* (IA) which abstract programs and capture their operational semantics on the domain of intervals. We define an IA as an extended state machine where the control structure is a finite state machine, extended by a mapping from interval variables to interval expressions. We will show later how to translate a C/C++ program to an IA.

Definition 1 (Syntax). *An* interval automaton *is a tuple* $(L, \mathbf{l}_0, X, E, update)$*, with*

- *a finite set of locations* L*,*
- *an initial location* $\mathbf{l}_0$*,*

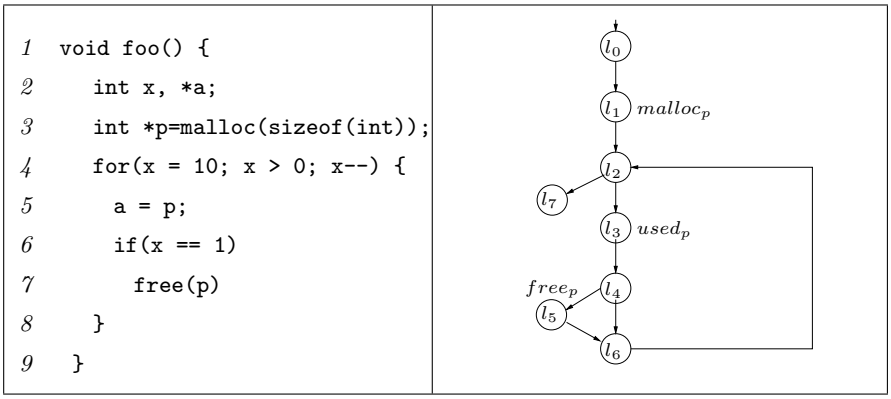

Fig. 1. Example program and labeled CFG for use-after-free check.

- *a set of interval variables* X,
- *a finite set of edges* $E \subseteq L \times L$, *and*
- *an effect function* $\text{update} : E \to (X \times \mathcal{C}(X))$.

The effect *update* assigns to each edge a pair of an interval variable and an interval expression. We will refer to the (left-hand side) variable part $update|_X$ as lhs, and to the (right-hand side) expression part $update|_{\mathcal{C}(X)}$ as $rhsexpr$. The set of all variables that appear in $rhsexpr$ will be denoted by $rhsvars$. Note, that only one variable is updated on each edge. This restriction is made for the sake of simplicity, but does does not restrict the expressivity of an IA. Fig. 2 shows an example of an IA.

Definition 2 (Semantics). *The semantics of an IA* $P = (L, \mathbf{l}_0, X, E, update)$ *is defined by a labeled transition system* $LTS(P) = (S, S_0, A, R, F)$ *where*

- S *is the set of states* (l, v) *with location* l *and an interval valuation* v.
- S_0 *is the set of initial states* $s_0 = (\mathbf{l}_0, v_0)$, *with* $v_0 \equiv [-\infty, \infty]$.
- A *is the alphabet consisting of the set of edges* E.
- $R \subseteq S \times A \times S$ *is the transition relation of triples* $((l, v), (l, l'), (l', v'))$, *i.e, transitions from state* (l, v) *to* (l', v') *labeled by* (l, l'), *if there exists a* (l, l') *in* E, *such that* $v' = v[lhs(e) \leftarrow [\![rhsexpr(e)]\!]_v]$ *and* $[\![rhsexpr(e)]\!]_{v'} \neq \emptyset$.
- $F = S$ *is the set of final states, i.e., all states are final states.*

It might seem a bit awkward that the transitions in the LTS are labeled with the edges of the IA, but this will be used later to define the synchronous composition with an observer. Since each transition is labeled with its corresponding edge we obtain a deterministic system, i.e., for a given word there exists only one possible run. We identify a word $((l_0, l_1), (l_1, l_2), \ldots, (l_{m-1}, l_m))$ in the remainder by the sequence of locations $(l_0, \ldots, l_m)$.

Given an IA P. Its language $\mathcal{L}_P$ contains all sequences $(l_0, \ldots, l_n)$ which satisfy the following:

$$
\begin{aligned}
& l_0 = \mathbf{l}_0 && (1)\\
\wedge\ & \forall i = 0, \ldots, n-1.\ (l_i, l_{i+1}) \in E && (2)\\
\wedge\ & v_0 \equiv [-\infty, +\infty] \wedge \exists v_1, \ldots, v_n.([\![rhsexpr(l_i, l_{i+1})]\!]_{v_i} \neq \emptyset && (3)\\
& \qquad \wedge\ v_{i+1} = v_i[lhs(l_i, l_{i+1}) \leftarrow [\![rhsexpr(l_i, l_{i+1})]\!]_{v_i}])
\end{aligned}
$$

This mean that a word (1) starts in the initial location, (2) respects the edge relation E, and (3) there exists a sequence of non-empty valuations that satisfies the updates associated with each edge. We use this characterization of words as a satisfiability problem to generate systems of interval equations that have a non-empty solution only if a sequence $(l_0, \ldots, l_n)$ is a word. We will define for a given IA P and sequence w a *conflict* as an interval equation system with an empty least solution, which proves that w cannot be a word of the IA P.

4 Path Reduction

The labeled CFG as defined in Section 2.3 is a coarse abstraction of the actual program. Like most static analysis techniques this approach suffers from false positives. In the context of this paper we define a property as a regular language, and satisfaction of a property as language inclusion. The program itself will be defined by an Interval Automaton P and its behavior is defined by the language of the corresponding $LTS(P)$. Since interval automata are infinite state systems, we do not check the IA itself but an abstraction $\hat{P}$. This abstraction is initially an annotated CFG as depicted in Fig. 1.

A *positive* is a word in the abstraction $\hat{P}$ that does not satisfy the property. A *false positive* is a positive that is not in the actual behavior of the program, i.e. it is not in the language of the $LTS(P)$. *Path reduction* is then defined as the iterative process that restricts the language of the abstraction, until either a true positive has been found, or until the reduced language satisfies the property.

4.1 Path Reduction Loop

Given an IA $P = (L, \mathbf{l}_0, E, X, update)$ we define its finite abstraction $\hat{P}$ as follows: $\hat{P} = (L, \mathbf{l}_0, E, E', L)$ is a labeled transition system with states L, initial state $\mathbf{l}_0$, alphabet E, transition relation $E' = \{(l, (l, l'), l') | (l, l') \in E\}$, and the entire set L as final states. The LTS $\hat{P}$ is an abstraction of the $LTS(P)$, and it represents the finite control structure of P. The language of $\hat{P}$ will be denoted by $\mathcal{L}_{\hat{P}}$. Each word of $\hat{P}$ is by construction a word of $LTS(P)$. Let $\mathcal{L}_\phi$ be the language defined by the specification.

We assume to have a procedure that checks if the language of LTS $\mathcal{L}_{\hat{P}}$ is a subset of $\mathcal{L}_\phi$, and produces a counterexample if this is not the case (cf. Section 4.5). If this procedure finds a word in $\mathcal{L}_{\hat{P}}$ that is not in $\mathcal{L}_\phi$, we have to check whether this word is in $\mathcal{L}_P$, i.e. we have to check whether it satisfies equation

(1) to (3). Every word $w = (l_0, \ldots, l_m)$ in $\mathcal{L}_{\hat{P}}$ satisfies by construction (1) and (2). A word $w = (l_0, \ldots, l_m)$ such that there exists no solution for (3) cannot be a word of $\mathcal{L}_P$. In this case we call the word *spurious*.

In Section 4.2) we introduce a procedure to check whether a word is spurious. We will use it in an iterative loop to check if the infinite LTS of IA P satisfies the property, by checking a finite product of abstraction the $\hat{P}$ with the observers instead. This loop is structured as follows:

1. Let $\hat{P}_0 := \hat{P}$, and $i = 0$.
2. Check if $w \in \mathcal{L}_{\hat{P}_i} \setminus \mathcal{L}_\phi$ exists. If such a w exists got to step 3, otherwise exit with "property satisfied".
3. Check if $w \in \mathcal{L}_P$. If $w \notin \mathcal{L}_P$ build observer Obs^w, otherwise exit with "property not satisfied". The observer satisfies the following
   (a) it accepts w, and
   (b) all accepted words $w' \notin \mathcal{L}_P$.
4. Let $\hat{P}_{i+1} := \hat{P}_i \times Obs^{w^C}$, with . $\hat{P}_i \times Obc^{w^C}$ is the synchronous composition of $\hat{P}_i$ and the complement of Obs^w. Increment i and goto step 3.

The role of the observers is to rule out spurious counterexamples from the accepted language. They serve a similar purpose as predicates in counterexample guided abstraction refinements that abstract the C program as Boolean program [3]. Since we abstract the program as an interval automaton, we use observer automata instead of additional predicates to achieve the refinement. This is is necessary since there is no useful equivalent of Boolean predicates in the interval domain. The remainder of this section explains how to check if a word is in $\mathcal{L}_P$, how to build a suitable observer, and how to combine it in a framework that uses NuSMV to model check the finite abstraction $\hat{P}_i$.

Example. The initial coarse abstraction as a CFG is shown in Fig. 1 loses the information that p cannot be used after it was freed. The shortest counterexample based on the CFG is to initialize x to 10 in line 4, enter the `for`-loop, take the `if`-branch, free `p` in line 7, decrement `x`, return to the beginning of the `for`-loop, and then to use `p` in line 5. This counterexample is obviously spurious. Firstly, because the `if`-branch with condition `x == 1` at line 7 is not reachable while $x = 10$. Secondly, because if the programm enters the if-branch, it implies `x == 1`, and it will be impossible to reenter the loop, given the decrement `x--` and the loop condition `x > 0`.

4.2 Checking for Spurious Words

Every word $w = (l_0, \ldots, l_m)$ in $\mathcal{L}_{\hat{P}}$ satisfies by construction (1) and (2). It remains to be checked if condition (3) can be satisfied. A straightforward approach is to execute the trace on $LTS(P)$. However this can only determine if that particular word is spurious. Our proposed approach builds an equation system instead, which allows us to find a set of conflicting interval equations that can in turn be used to show that an entire class of words is spurious. Another

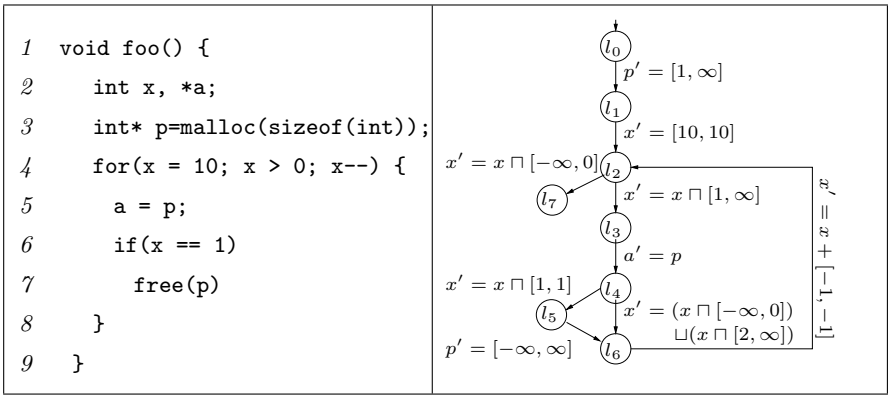

Fig. 2. Abstraction of the program as IA. Analysis of the syntactic properties of the annotated CFG in Fig.1 is combined with an analysis of the IA to the right.

straightforward approach to build such an equation system is to introduce for each variable and edge in w an interval equation, and to use an interval solver to check if a solution to this system of interval equations exists. A drawback of this approach is that it introduces $(m+1) \times n$ variables and $m \times n$ equations. In the following we present an approach to construct an equation system with at most one equation and one variable for each edge in w.

Interval Equation System for a Sequence of Transitions. We describe how to obtain an equation system for a word $w \in \mathcal{L}_{\hat{P}}$, such that it has a non-empty least solution only if $w \in \mathcal{L}_P$. This system is generated in three steps:

I. Tracking variables. For each variable X of the program P we will track its use. Let X_L be a set of fresh variables x_l, one for each variable and occurrence where it can be used. We add to X_L a special element $\top$, which will be used as default.

Given an IA P over variables X, its abstraction $\hat{P}$, and a word $w = (l_0, \ldots, l_n)$ of $\hat{P}$ we denote the location of the last update of x before the i-th transition of word w as $x^w_{(i)}$. It is recursively defined as follows:

$$x^w_{(i+1)} = \begin{cases} x^w_{l_{i+1}} & \textit{if } x = lhs(l_i, l_{i+1}) \\ x_{(i)} & \textit{otherwise} \end{cases}$$

for $i > 0$, and with $x^w_{(0)} = \top$ as base case. The function is parameterized in w, but the superscript will be omitted if it is clear from the context.

II. Generating equations. For each edge in w we generate an interval expression over X_L. We define $expr_w : \{0, \ldots, m\} \to \mathcal{C}(X_L)$ as follows:

$$expr_w(i) \mapsto \quad rhsexpr(l_{i-1}, l_i)[x_{(i-1)}/x]_{x \in rhsvars(l_{i-1}, l_i)} \tag{4}$$

w	var	$expr_w(i)$	IE_w	Reduced conflicts
(l_0, l_1)	p_{l_1}	$[1, \infty]$	$p_{l_1} = [1, \infty]$	$x_{l_2} = [10, 10]$
(l_1, l_2)	x_{l_2}	$[10, 10]$	$x_{l_2} = [10, 10] \sqcup (x_{l_5} + [-1, -1])$	$x_{l_3} = x_{l_2} \sqcap [1, \infty]$
(l_2, l_3)	x_{l_3}	$x_{l_2} \sqcap [1, \infty]$	$x_{l_3} = (x_{l_2} \sqcap [1, \infty])$	$x_{l_5} = x_{l_3} \sqcap [1, 1]$ (conflict 1)
(l_3, l_4)	a_{l_4}	p_1	$a_{l_4} = p_{l_1} \sqcup p_{l_6}$	
(l_4, l_5)	x_{l_5}	$x_{l_3} \sqcap [1, 1]$	$x_{l_5} = x_{l_3} \sqcap [1, 1]$	$x_{l_5} = [-\infty, \infty] \sqcap [1, 1]$
(l_5, l_6)	p_{l_6}	$[-\infty, \infty]$	$p_{l_6} = [-\infty, \infty]$	$x_{l_2} = x_{l_5} + [-1, -1]$
(l_6, l_2)	x_{l_2}	$x_{l_5} + [-1, -1]$		$x_{l_3} = x_{l_2} \sqcap [1, \infty]$ (conflict 2)
(l_2, l_3)	x_{l_3}	$x_{l_2} \sqcap [1, \infty]$		
(l_3, l_4)	a_{l_4}	p_{l_6}		

Fig. 3. Equations for counterexample w of the IA depicted in Fig. 2

An expression $expr_w(i)$ is the right-hand side expression of the update on (l_{i-1}, l_i), where all occurring variables are substituted by variables in X_L. We will use $var^w_{(i)}$ to denote the function from $rhsvars(l_{i-1}, l_i)$ to X_L, that assigns $x \mapsto x_{(i)}$. One might think of this as a partial function over X, restricted to the variables that are actually used on the right-hand side. We use $\emptyset$ to denote the functions that have the empty set as domain.

III. Generating equation system. Locations may occur more than once in a word, and variables maybe updated by multiple edges. Let $writes_w \subseteq X_L$ the set $\{x_l | \exists i \text{ s.t.} x = lhs(l_{i-1}, l_i)\}$, and $indices_w$ be a mapping, that assigns to each $x_l \in writes_w$ the set $\{i | x = lhs(l_{i-1}, l_i) \wedge l_i = l\}$. The system $IE_w : X_L \to \mathcal{C}(X_L)$ is defined as follows:

$$x_l \mapsto \begin{cases} \bigsqcup_{i \in indices_w(x_l)} expr_w(i) & \text{if } x_l \in writes_w \\ [-\infty, \infty] & \text{otherwise} \end{cases} \tag{5}$$

System IE_w assigns each variable $x_l \in writes_w$ to a union of expressions; one expression for each element in $indices_w(x_l)$. The default value $\top$ is mapped to $[-\infty, \infty]$, since by construction $\top \notin writes_w$.

Example. Fig. 3 depicts for word $w = (l_0, l_1, , l_2, l_3, l_4, l_5, l_6, l_2, l_3, l_4)$ how to generate IE_w. The first column gives the transitions in w. The second column gives the variable in $writes_w$. The variable p_{l_1}, for example, refers to the update of p on the first transition (l_0, l_1) of the IA in Fig. 2. We have that $x_{(5)} = x_{l_3}$, since the last update of x before (l_4, l_5) was on edge (l_2, l_3).

The third column gives the equations $expr_w(i)$. For example, the right-hand side $rhsexpr(l_4, l_5)$ is $x' = x \sqcap [1, 1]$. Since $x_{(4)} = x_{l_3}$, we get that $expr_w(5)$ is $x_{l_3} \sqcap [1, 1]$. The fourth column shows the equation system IE_w derived from the equations. We have, for example, that x is updated on (l_1, l_2), the second edge in w, and (l_6, l_2), the 8th edge. Hence, $indices_w(x_{l_2}) = \{2, 8\}$. Equation $IE_w(x_{l_2})$ is then defined as the union of $expr_w(2)$, which is $[10, 10]$, and $expr_w(8)$, which is $x_{l_5} + [-1, -1]$. The least solution of the equation system IE_w is $p_{l_1} = [1, \infty]$,

$x_{l_2} = [10, 10]$, $x_{l_3} = [10, 10]$, $a_{l_4} = [-\infty, \infty]$, $x_{l_5} = \emptyset$, and $p_{l_6} = [-\infty, \infty]$. Since $x_{l_5} = \emptyset$, there exists no solution, and w is spurious. □

Lemma 1. *Given a word $w \in \mathcal{L}_{\hat{P}}$. Then there exist a sequence of non-empty valuations $v_1, \ldots, v_m$ such that (3) holds for w only if IE_w has a non-empty least solution.*

Proof. Given a solution to (3) we can construct a non-empty solution of IE_w, which must be included in the least solution of IE_w. □

One advantage of the interval equations framework is that it reasons naturally over loops. A third or fourth repetition does not introduce new variable in $writes_w$, and neither new expressions. This means that the equation system for a word that is a concatenation $\alpha\beta\beta\beta\gamma$ has an empty solution, if the concatenation $\alpha\beta\beta\gamma$ has. This is captured by the following lemma:

Lemma 2. *Given a word $w \in \mathcal{L}_{\hat{P}}$, with w being a concatenation $\alpha\beta\beta\gamma$ of sequences $\alpha = (l_0^\alpha, \ldots, l_{m_\alpha}^\alpha)$, $\beta = (l_1^\beta, \ldots, l_{m_\beta}^\beta)$, and $\gamma = (l_1^\gamma, \ldots, l_{m_\gamma}^\gamma)$. Let $w' = \alpha\beta\beta\beta\gamma$. Then $w' \in \mathcal{L}_{\hat{P}}$, $writes_w = writes_{w'}$ and the solution of IE_w is also the solution of $IE_{w'}$.*

Proof: (i) Given $w \in \mathcal{L}_{\hat{P}}$, $w = (l_0, \ldots, l_{m_\alpha + 2\,m_\beta + m_\gamma}$ we have that $(l_i, l_{i+1}) \in E$. This implies that also all edges of w' are in E, and hence, $w' \in \mathcal{L}_{\hat{P}}$.
(ii) Observe, that the second and third repetition of β in w' add no fresh variables x_l to $writes_w$, hence $writes_w = writes_{w'}$.
(iii) Let $x \in rhsvars(l_{i-1}, l_i)$, with (l_{i-1}, l_i) in the second iteration of β, i.e., $i \in m_\alpha + m_\beta + 1, \ldots, m_\alpha + 2\,m_\beta$. Then $x_{(i-1)}$ refers either to x_{l_j}, with $j \in 0, \ldots, m_\alpha$ or $j \in m_\alpha + 1, \ldots, 2\,m_\alpha$. In either case we can show that $x_{(i-1+m_\beta)} = x_{(i-1)}$. Henceforth, the third iteration of the loop will lead to the same expressions $rhsexpr(l_{i-1}, l_i)[x_{x_{(i-1)}}/x]$. Taking the union of more of the same expressions will not change the solution. □

4.3 Conflict Discovery

The previous subsection described how to check if a given word $w \in \mathcal{L}_{\hat{P}}$ is spurious. Interval solving, however, leads to an over-approximation, mostly due to the $\sqcup$-operation. This subsection describes how to reduce the numbers of (non-trivial) equations in a conflict and at the same time the over-approximation error, by restricting conflicts to fragments and the cone-of-influence.

Conceptually, a conflict is an equation system IE_w that has no non-empty solution. For matter of convenience we introduce an alternative representation of the equation system; each variable x_l in X_l is mapped to a set of pairs. Each of these pairs consists of an edge (l_{i-i}, l_i) and mapping $var_{(i)}$ from $rhsvars(l_{i-i}, l_i)$ to X_L. Each pair represents an expression $expr_w(i)$ as defined in (4); it records the edge, and the relevant variable substitutions. The conflict for a word $w = (l_0, \ldots, l_m)$ is thus alternatively represented by $conf^w(x_l) = \{((l_{i-1}, l_i), var_{(i-1)}) \mid x = lhs(l_{i-1}, l_i) \wedge l = l_i\}$. We refer to this mapping as the representation $conf^w$ of the conflict.

For the equation system of the example we have $conf^w(p_{l_1}) = \{((l_0, l_1), \emptyset)\}$, $conf^w(x_{l_2}) = \{((l_1, l_2), \emptyset), \{((l_6, l_2), (x \mapsto x_{l_5})\}$, $conf^w(x_{l_3}) = \{((l_2, l_3), (x \mapsto x_{l_2}))\}\}$, $conf^w(a_{l_4}) = \{((l_3, l_4), (a \mapsto p_{l_1})), ((l_3, l_4), (a \mapsto p_{l_6}))\}$, $conf^w(x_{l_5}) = \{((l_4, l_5), (x \mapsto x_{l_3})\}$, and $conf^w(p_{l_6}) = \{((l_5, l_6), \emptyset)\}$. The empty set occurs in a pair when the right-hand side of the update has no variables.

Fragments. For CEGAR approaches for infinite-state systems it has been observed that it is sufficient and often more efficient to find a spurious fragments of a counterexample, rather than a spurious counterexample [12, 13]. The effect is similar to small or minimal predicates in SAT-based approaches. The difference between a word and a fragment in our context is that a fragment may start in any location.

Given a word $w = (l_0, \ldots, l_m)$ a fragment $w' = (l'_0, \ldots, l'_{m'})$ is a subsequence of w. A fragment of $LTS(P)$ is defined as a sequence of edges that satisfies (2) and (3). A fragment of $\hat{P}$ is a sequence of edges satisfying the edge relation E, i.e., satisfying (2). Given a fragment w' we can construct a system of interval equations $IE_{w'}$ as described for words earlier.

For subsequence w' of a word w we can show the analog of Lemma 1. If the solution of $IE_{w'}$ is empty, i.e., if the fragment w' is spurious, then the word w is spurious as well. If there exists a sequence of non-empty valuations $v_1, \ldots, v_m$ for w, then they also define a non-empty subsequence of valuations for w'.

Rather than checking all $m^2/2$ fragments, the analysis focusses on promising candidates, based on the following two observation: (1) For an *update* in (3) to result in an empty solution, there must at least exist an element in $\mathcal{I}$ that can be mapped by *update* to the empty set. An example of such updates are intersections with constants such as $x' = x \sqcap [1, \infty]$. For any $x = [a, b]$, with $b < 1$ the next state can only satisfy $x' = \emptyset$. Updates that map only the empty set to the empty set can be omitted from the tail of a fragment. (2) Initially all variables are unconstrained, i.e. we start in valuation $v \equiv [-\infty, \infty]$. Consequently updates that map valuation $v \equiv [-\infty, \infty]$ to $[-\infty, \infty]$ can be omitted from the beginning of a fragment. The full fragment can only have an empty solution if the reduced has, without updates at the end that map only empty sets to empty sets, and without updates at the beginning that map $[-\infty, \infty]$ to $[-\infty, \infty]$.

Cone-of-influence. Let IE_w be a conflict for fragment $w = (l_0, \ldots, l_m)$. We further reduce the conflict by restricting it to the cone-of-influence of x_{l_m}.

The *cone-of-influence* of x_{l_m} is defined as the least fixpoint $\mu C.\{y_l | \exists y_{l'} \in C.\ y_l \in rhsvars(IE_w(y_{l'}))\} \cup \{x_{l_m}\})$. We denote this set as $\overline{writes}_w$. We then define the reduced conflict $\overline{IE}_w$ as

$$x_l \mapsto \begin{cases} \bigsqcup_{i \in indices_w(x_l)} expr_w(i) & \text{if } x_l \in \overline{writes}_w \\ [-\infty, \infty] & \text{otherwise.} \end{cases} \tag{6}$$

This reduction ensures that $\overline{IE}_w$ has an empty least solution if IE_w has. In the remainder we will refer to $\overline{IE}_w$ as the *reduced conflict*, which is uniquely determined by the fragment w.

The cone-of-influence reduction starts with the last edge (l_{m-1}, l_m) and the variable x that is written to, and then backtracks to all variables that it depends on. All other variables are ignored, i.e. assumed to be $[-\infty, +\infty]$, and all edges that do not contain at least one variable in $\overline{writes}_w$ are omitted. The corresponding reduced representation of the reduced conflict is $\overline{conf}^w \triangleq conf^w|_{\overline{writes}_w}$.

Example. There are two conflicts among the candidate fragments in Fig. 3. Conflict 1, for fragment $(l_1, l_2, l_3, l_4, l_5)$, has as least solution $x_{l_2} = [10, 10], x_{l_3} = [10, 10], x_{l_5} = \emptyset$. Conflict 2, for fragment $(l_4, l_5, l_6, l_2, l_3)$, has as least solution $x_{l_5} = [1, 1], x_{l_2} = [0, 0], x_{l_3} = \emptyset$. The equation was p_{l_6} was not included in the second conflict, as it is not in the cone-of-influence of x_{l_3}.

Table 3 shows the conflicts as equation system. The alternative representation of the reduced conflict for fragment $w = (l_4, l_5, l_6, l_2, l_3)$ is $\overline{conf}^w(x_{l_5}) = \{((l_4, l_5), (x \mapsto \top))\}$, $\overline{conf}^w(x_{l_2}) = \{((l_6, l_2), (x \mapsto x_{l_5}))\}$, and finally $\overline{conf}^w(x_{l_5}) = \{((l_2, l_3), (x \mapsto x_{l_2}))\}$. Note, that $(x \mapsto \top)$ is in the first set, since initially $x_i = \top$ for all $x \in X$. When the corresponding equation was generated, $\top$ was replaced by $[-\infty, \infty]$ in $\overline{IE}_w(x_{l_5})$.

4.4 Conflict Observer

Given a reduced conflict $\overline{IE}_w$ for a fragment $w = (l_0, \ldots, l_m)$, we construct an observer such that if a word $w' \in \mathcal{L}_{\hat{P}}$ is accepted, then $w' \notin \mathcal{L}_P$. The observer is an LTS over the same alphabet E as $LTS(P)$ and $\hat{P}$.

Definition 3. *Given an IA $P = (L, \mathbf{l}_0, E, X, update)$ and reduced conflict $\overline{IE}_w$, with representation $\overline{conf}^w$, for a fragment $w = (l_0, \ldots, l_m)$, define X^w as the set of all variables $x \in X$ such that $x_{l_i} \in \overline{writes}_w$, for some edge l_i in w. The observer Obs^w is a LTS with the*

- *set S_{Obs} of states $(current, eqn, conflict)$ with valuation $current : X^w \to (\overline{writes}_w \cup \top)$, valuation $eqn : \overline{writes}_w \to \{unsat, sat\}$, and location $conflict \in \{all, some, none\}$,*
- *initial state $(current_0, eqn_0, conflict_0)$ with $current_0 \equiv \top$, $eqn_0 \equiv unsat$, and $conflict_0 = none$,*
- *alphabet E,*
- *transition relation $T \subseteq S_{Obs} \times E \times S_{Obs}$ (see Def. 4 below), and*
- *a set final states F. A state is final if $conflict = all$.*

Before we define the transition relation formally, we give a brief overview of the role the different variables have.

- Variable *current* is used to records for each variable the location of the last update. It mimics $x_{(i)}$ in the previous section.
- Variable *eqn* represents $\overline{IE}_w(x_l)$, or alternatively $\overline{conf}^w(x_l)$, for $x_l \in writes_w$. This variable records if $\overline{IE}_w(x_l)$ is satisfied.
- Variable *conflict* has value *all*, *some*, *none*, if $eqn'(x_l) = sat$ for all, some or no $x_l \in \overline{writes}_w$, respectively. It records if all, some or none of $\overline{IE}_w(x_l)$ is currently satisfied.

The transitions can be informally characterized as follows:

- To update *current*, the observer needs to check if the observed edge (λ, λ') has an update that modifies any variable in $x \in X^w$. In this case *current* takes the value $x_{\lambda'}$.
- To update *eqn* for x_l, the observer needs to check if the update on the observed edge (λ, λ') creates an expression that appears in $\overline{IE}_w(x_l)$, i.e. it needs to check if the transition label and the state of *current* matches a pair in $\overline{conf}^w(x_l)$. If it does, then $eqn(x_l)$ becomes *sat*.
- To update *conflict*, we check if *eqn* is *sat* in the next state for all $x_l \in \overline{writes}_w$.

For each of these three variables there are a few exceptions:

- The next state of *current* will be $\top$ for all variables, if none of *eqn* is *sat*, i.e. if $conflict = none$.
- Variables $eqn(x_l)$ will be reset to their initial state $\top$, if the edge (λ, λ') writes to a variable in $x'_l \in X^w$, while neither (λ, λ') nor the *current* match any pair in $\overline{conf}^w(x'_l)$. In this case $eqn(x_l)$ will be set to its initial state *unsat*.
- Once *conflict* is in *all*, it remains there forever.

All the different variables depend on each other, but there is no circular dependency. The next state of *current* depends on the next state of *conflict*. The next state of *conflict* the depends on the next state of *eqn*. But the next state of *eqn* depends on the current state of *current*.

Before we define the transition relation, we give two Boolean predicates. Given an edge (λ, λ') and a variable $x_{\lambda'} \in \overline{writes}_w$ predicate $match(\lambda, \lambda', x_{\lambda'})$ is true if the update on (λ, λ') in the current state matches some expression in $\overline{IE}_w(x_{\lambda'})$. Recall that the observer is defined with respect to some fragment $w = (l_0, \ldots, l_m)$.

$$match(\lambda, \lambda', x_{\lambda'}) \triangleq \exists((l_{i-1}, l_i), var_{(i)}) \in \overline{conf}^w(x_{\lambda'}) \text{ s.t. } (l_{i-1}, l_i) = (\lambda, \lambda') \text{ and } \forall y \in rhsvars(\lambda, \lambda').\ var_{(i)}(y) = \top \ \vee \ var_{(i)}(y) = current(y)$$

Related to the match is the following predicate $reset(\lambda, \lambda', x_{\lambda'})$, which is true when there is no suitable match.

$$reset(\lambda, \lambda', x_{\lambda'}) \triangleq \forall((l_{i-1}, l_i), var_{(i)}) \in \overline{conf}^w(x_{\lambda'}) \text{ s.t. } (l_{i-1}, l_i) = (\lambda, \lambda') \text{ and } \exists y \in rhsvars(\lambda, \lambda').\ var_{(i)}(y) \neq \top \ \wedge \ var_{(i)}(y) \neq current(y)$$

The state of the observer will also be reset when (λ, λ') is not equal to any edge appearing in $\overline{conf}^w$, while $x_{\lambda'} \in \overline{writes}_w$ and $x = lhs(\lambda, \lambda')$. The update (λ, λ') writes to x and we conservatively assume that it matches none of the expressions in $\overline{IE}_w(x_{\lambda'})$.

The transition relation for the observer is then defined as follows:

Definition 4 (Transition relation). *Transitions from* $(current, eqn, conflict)$ *to* $(current', eqn', conflict')$ *labeled* (λ, λ') *for the observer* Obs^w *are defined as follows:*

$conflict$
  if $conflict = all$
    $conflict' = all$
  else if $\forall x_l \in \overline{writes}_w.\ eqn'(x_l) = sat$
    $conflict' = all$
  else if $\exists x_l \in \overline{writes}_w.\ eqn'(x_l) = sat$
    $conflict' = some$
  otherwise
    $conflict' = none$

$eqn(x_{\lambda'})$
  if $x = lhs(\lambda, \lambda') \wedge \forall((l_{i-1}, l_i), var_{(i)}) \in \overline{conf}^w(x_{\lambda'}).\ (l_{i-1} \neq \lambda \wedge \lambda' = l))$
    $eqn'(x_{\lambda'}) = unsat$
  else if $x = lhs(\lambda, \lambda') \wedge \exists y_l \in \overline{writes}_w.\ reset(\lambda, \lambda', y_l)$
    $eqn'(x_{\lambda'}) = unsat$
  else if $x = lhs(\lambda, \lambda') \wedge match(\lambda, \lambda', x_l)$
    $eqn'(x_{\lambda'}) = sat$
  otherwise
    $eqn'(x_{\lambda'}) = eqn(x_{\lambda'})$

$current(x)$
  if conflict'= none
    $write'(x) = \top$
  else
    if $x = lhs(\lambda, \lambda')$
      $write'(x) = \lambda'$
    otherwise
      $write'(x) = write(x)$

The interaction between *current*, *eqn*, and *conflict* is somewhat subtle. The idea is that the observer is initially in $conflict = none$. If an edge is observed, which generates an expression $expr(i)$ that appears in $\overline{IE}_w(x_l)$ (see Eq. (5)), then $conflict' = some$, and the corresponding $eqn(x_l) = sat$. It can be deduced $\overline{IE}_w(x_l)$ is satisfied, unless another expression is encountered that might enlarge the fixed point. This is the case when an expression for x_l will generated, that does not appear in $\overline{IE}_w(x_l)$. It is conservatively assumed that this expression increases the fixed point solution.

If $conflict = all$ it can be deduced that the observed edges produce an equation system $\overline{IE}_{w'}$ that has a non-empty solution only if $\overline{IE}_w$ has a non-empty solution. And from the assumption we know that $\overline{IE}_w$ has an empty-solution, and thus also $\overline{IE}_{w'}$. Which implies that the currently observed run is infeasible and cannot satisfy Eq. 3.

Example. The observer for the first conflict in Fig. 3 accepts a word if a fragment generates a conflict $x_{l_2} \mapsto [10, 10], x_{l_3} \mapsto x_{l_2} \sqcap [1, \infty], x_{l_5} \mapsto x_{l_3} \sqcap [1, 1]$. This is the case if it observes edge (l_1, l_2), edge (l_2, l_3) with a last write to x at l_2, and edge (l_4, l_5) with a last write to x at l_3. All other edges are irrelevant, as long as they do not write to x_2, x_3 or x_5, and change the solution. For example, this

would be the case for (l_2, l_3) if $current(x) \neq l_2$. It creates an expression different from $x_{l_2} \sqcap [1, \infty]$, and thus potentially enlarges the solution set. The NuSMV model of this observer can be found in the Appendix A.

The observer for the other conflicts is constructed similarly. The complement of these observers are obtained by labeling all states in $S \setminus F$ as final. The product of the complemented observers together with the annotated CFG in Fig.1 removes all potential counterexamples. The observer for the first conflict prunes all runs that enter the `for`-loop once, and then immediately enter the `if`-branch. The observer for the second conflict prunes all words that enter the `if`-branch and return into the loop.

Lemma 3. *Given a reduced conflict $\overline{IE}_w$ and its representation $\overline{conf}^w$ for a fragment $w = (l_0, \ldots, l_m)$, the observer Obs^w satisfies the following:*

- *If a word $w' \in \mathcal{L}_{\hat{P}}$ contains fragment w'', such that $\overline{IE}_{w''}$ has non non-empty solution, and such that $\overline{conf}^{w''} = \overline{conf}^w$, then w' is accepted by Obs^w.*
- *If a word $w' \in \mathcal{L}_{\hat{P}}$ is accepted by Obs^w, then $w \notin \mathcal{L}_P$.*

Proof. (i) Let w'' be the first occurrence of a fragment such that that $\overline{IE}_{w''}$ has non non-empty solution, and such that $\overline{conf}^{w''} = \overline{conf}^w$. If *conflict* is in *all* at the beginning of the fragment, the word is trivially accepted. Assume that *conflict* is in *some* or *none* at the beginning of the fragment. By assumption $\overline{conf}^{w''} = \overline{conf}^w$, we know that none of the edges in w'' will satisfy *reset*. It is also guaranteed that if an edge writes to a variable x in location l it satisfies *match*. By assumption we also know that at the end of the fragment *conflict* will be in *all*. (ii) If a word is accepted, it means that there exists a fragment $w'' = (l_0, \ldots, l_m)$ such that *conflict* will be in state *none* at l_0, in state *some* for $l_1 \ldots, l_{m-1}$ at the beginning, and in state *all* at l_m. This fragment will generate for each variable $x_l \in \overline{writes}_{w'}$ an equation $\overline{IE}_{w''}(x_l)$ that is at least as restrictive as $\overline{IE}_{w'}(x_l)$. Consequently, if $\overline{IE}_{w'}(x_l)$ has no non-empty solution, then neither can $\overline{IE}_{w''}(x_l)$ have. Hence w' which contain w'' as fragment, cannot be in $\mathcal{L}_P$. □

The second property ensures that each observer is only constructed once. This is needed to guarantee termination. Each observer is uniquely determined by the finite set expressions that appear in it, and since X_L and E are finite, there exists only a finite set of possible expressions that may appear in $\overline{conf}$. Consequently, there can only exist a finite set of conflicts. The second first property states that the language of $\hat{P}' = \hat{P} \times Obs^{wC}$ contains $\mathcal{L}_P$.

4.5 Path Reduction with NuSMV

The previous subsections assumed a checker for language inclusion for LTS. In practice we use however the CTL model checker NuSMV. The product of $\hat{P}$ with the complement of the observers is by construction a proper abstractions of $LTS(P)$. The results for language inclusion therefore extend to invariant checking, path inclusion, LTL and ACTL model checking. For pragmatic reasons we

do not restrict ourselves to any of these, but use full CTL and LTL[1]. Whenever NuSMV produces a counterexample path, we use interval solving as described before to determine if this path is spurious.

Note, that path reduction can also be used to check witnesses, for example for reachability properties. In this case path reduction will check if a property which is true on the level of abstraction is indeed satisfied by the program.

The abstraction $\hat{P}$ and the observers are composed synchronously in NuSMV. The observer synchronizes on the current and next location of $\hat{P}$. The property is defined as a CTL property of $\hat{P}$. The acceptance condition of the complements of observers is modeled as LTL fairness condition $G\neg(\mathit{conflict} = \mathit{all})$. The NuSMV code can be found in Appendix A.

5 Implementation and Experiments

5.1 C to Interval Equations

This section describes how to abstract a C/C++ program to a set of interval equations, and covers briefly expression statements, condition statements as well as the control structures.

Expressions statements involving simple operations such as addition and multiplication are directly mapped to interval equations. E.g., an expression statement `x=(x+y)*5` is represented as $x_{i+1} = (x_i + y_i) * [5, 5]$. Subtraction such as $x = x - y$ can be easily expressed as $x_{i+1} = x_i + ([-1, -1] * y_i)$.

Condition statements occur in constructs such as if-then-else, for-loops, while-loops etc. For every condition such as `x<5` we introduce two equations, one for the true-case and one for the false-case. Condition `x<5` has two possible outcomes $x_{tt} = x \sqcap [-\infty, 4]$ and $x_{ff} = x \sqcap [5, \infty]$. More complex conditions involving more than one variable can also be approximated and we refer the interested reader to [14].

Joins are introduced were control from two different branches can merge. For instance, let x_i be the last *lhs*-variable in the if-branch of an if-then-else, and let x_j be the last *lhs*-variables in the else-branch. The values after the if-then-else is then the union of both possible values, i.e., $x_k = x_i \sqcup x_j$.

For all other operations that we cannot accurately cover we simply over-approximate their possible effect. Function calls, for example, are handled as conservatively as possible, i.e., `x=foo()` is abstracted as $x_i = [-\infty, +\infty]$. The same holds for most of pointer arithmetic, floating point operations and division.

It should be noted that infeasible paths mostly depend on combinations of conditions that cannot be satisfied. Typically, condition expressions and the operations having an effect on them are rather simple. Therefore, it is a sufficient first approach to over-approximate most but the aforementioned constructs.

[1] NuSMV 2.x supports LTL as well.

	no violation					violation				
	P1	P2	P3	P4	P5	P1	P2	P3	P4	P5
Splint	+	+	-	-	-	-	-	+	+	+
UNO	(-)	(-)	(-)	(-)	-	(+)	(+)	+	+	+
com. tool	+	+	+	+	-	-	-	+	+	+
Goanna	+	+	+	+	-	+	+	+	+	+

Table 1. For each tool the table shows if the tool found a true positive/negative "+" or a false positive/negative "-". Entries "(-)" and "(+)" refer to warnings that there *may* be an error.

5.2 Comparison

To evaluate our path reduction approach we added it to our static checker Goanna and compare its results with three other static analysis tools. One of the static analyzers is a commercial tool. We applied these tools to five programs.[2]

- The first program is the one discussed throughout the paper. It is similar to the example discussed in [4].
- The second program is identical, except that the loop is initialized to `x=50`.
- The third program tests how different tools deal with unrelated clutter. The correctness depends on two if-conditions: the first sets a flag, and the second assigns a value to a variable only if the first is set. In between the two if-blocks is some unrelated code, in this case a simple if-then-else.
- The fourth program is similar to the third, except that a for-loop counting up to 10 was inserted in-between the two relevant if-blocks .
- The last one is similar to the first program. This program uses however two counter-variables `x` and `y`. The correctness of the programm depends on the loop-invariant $x = y$. It is similar to the examples presented in [15].

For each of the programs we constructed one instance with a bug, and one without. For the first program, e.g. the loop condition was changed to `x>=0`. Since not all tools check for the same, we introduced different bugs for the different tools, which however all depended on the same paths.

The results in Table 1 show that static analysis tools often fail to produce correct warnings. Splint, for example, produces for the instances with and without a bug the same warnings, which is only correct in half of the cases. Our proposed path reduction produces one false positive for the last program. The least solution of the interval equation shows that both variables take values in the same interval, but it cannot infer the stronger invariant $x = y$.

The experiments were performed on a DELL PowerEdge SC1425 server, with an Intel Xeon processor running at 3.4 GHz, 2 MiB L2 cache and 1.5 GiB DDR-2 400 MHz ECC memory. The following table gives the maximal, minimal, mean and median run-times in seconds:

[2] These programs be found at `http://www.cse.unsw.edu.au/~ansgar/fpe/`

	max	min	mean	median
Splint	0.012	0.011	0.012	0.012
UNO	0.032	0.025	0.028	0.025
com. tool	0.003	0.002	0.003	0.003
Goanna	0.272	0.143	0.217	0.226

All static analysis tools are overall fast, and their run times are almost independent of the example program. For test programs as small at these the run-time reflect mostly the time for overhead and setup. While being more precise the Goanna run-time is still negligibly short for these examples and path reduction is independent of the loop counter value, as expected. Goanna benefits from the fact that interval solving deals efficiently with loops without unrolling [1].

6 Conclusions

In this work we presented an approach to enhance static program analysis with counterexample guided path reduction to eliminate false positives. While by default we investigate programs on a purely syntactic level, once we find a potential counterexample, it is mapped to an interval equation system. In case that the least solution of this system is empty, we know that the counterexample is spurious and identify a subset of equations which caused the conflict. We create an observer for the conflict, augment our syntactic model with this observer, re-run the analysis with the new model and keep repeating this iterative process until no more counterexamples occur or none can be ruled out anymore.

One of the advantages of our approach is that we do not require to unroll loops or to detect loop predicates as done in some CEGAR approaches. In fact, path-based interval solving is insensitive to loop bounds, and handles loops just like any other construct. However, path-based interval solving adds precision to standard abstract interpretation interval analysis. Moreover, we only use one data abstraction, namely a simple interval semantics for C/C++ programs.

We implemented our approach in our static analysis tool Goanna and compared it for a set of examples to existing static program analyzers. This demonstrated that Goanna is typically more precise than standard program analyzers.

Future work is to evaluate our approach further on real life software to identify a typical false alarm reduction ratio. Given that static analysis typically turns up a few bugs per 1000 lines of code, this will require some extensive testing. Moreover, we like to explore if slightly richer domains can be used to get additional precision without a signification increase in computation time and, most importantly, if this makes any significant difference for real life software. And finally, we plan to compare our static checker to existing software model checkers that by design should be more precise than Goanna, but also require typically much longer run-times.

References

1. Gawlitza, T., Seidl, H.: Precise fixpoint computation through strategy iteration. In: ESOP. LNCS 4421, Springer Verlag (2007) 300–315
2. Henzinger, T., Jhala, R., Majumdar, R., SUTRE, G.: Software verification with BLAST. In: Proc. SPIN2003. LNCS 2648 (2003) 235–239
3. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Proc. TACAS 2005. LNCS 3440 (2005) 570–574
4. Kroening, D., Weissenbacher, G.: Counterexamples with loops for predicate abstraction. In: CAV. LNCS 4144 (2006)
5. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. TACAS 2004. LNCS 2988 (2004) 168–176
6. Gulavani, B., Rajamani, S.: Counterexample driven refinement for abstract interpretation. In: TACAS 2006. LNCS 3920 (2006)
7. Wang, C., Yang, Z., Gupta, A., Ivancic, F.: Using counterexamples for improving the precision of reachability computation with polyhedra. In: CAV 207. LNCS 4590 (2007)
8. Fehnker, A., Huuck, R., Jayet, P., Lussenburg, M., Rauch, F.: Model checking software at compile time. In: Proc. TASE 2007, IEEE Computer Society (2007)
9. Holzmann, G.: Static source code checking for user-defined properties. In: Proc. IDPT 2002, Pasadena, CA, USA (June 2002)
10. Dams, D.R., Namjoshi, K.S.: Orion: High-precision methods for static error analysis of C and C++ programs. In: Formal Methods for Components and Objects. Volume 4111 of LNCS., Berlin Heidelberg, Springer (2006) 138–160
11. Schmidt, D.A., Steffen, B.: Program analysis as model checking of abstract interpretations. In: Proc. SAS '98, Springer-Verlag (1998) 351–380
12. Fehnker, A., Clarke, E., Jha, S., Krogh, B.: Refining abstractions of hybrid systems using counterexample fragments. In: Proc. HSCC 2005. LNCS 3414 (2005) 242–257
13. Jha, S.K., Krogh, B., Clarke, E., Weimer, J., Palkar, A.: Iterative relaxation abstraction for linear hybrid automata. In: Proc. HSCC 2007. LNCS (2007)
14. Ermedahl, A., Sjödin, M.: Interval analysis of C-variables using abstract interpretation. Technical report, Uppsala University (December 1996)
15. Jhala, R., McMillan, K.L.: A practical and complete approach to predicate refinement. In: Proc TACAS. LNCS 3920 (2006) 459–473

A Appendix

```
MODULE conflict1(pc)
VAR
 current_x: {l0,l1,l2,l3,l4,l5,l6};
 eqn_x_l2, eqn_x_l3, eqn_x_l5: boolean;
 conflict:{some, none,all};

ASSIGN
 init(current_x):=l0;
 init(eqn_x_l2):=0;
 init(eqn_x_l3):=0;
 init(eqn_x_l5):=0;
 init(loc):=none;


  next(conflict):=
    case
     loc=all:all;
     next(eqn_x_l2) & next(eqn_x_l3) & next(eqn_x_l5): all;
     next(eqn_x_l2) | next(eqn_x_l3) | next(eqn_x_l5): some;
     1:none;
    esac;

 next(eqn_x_l2) :=
   case
     ((pc=l6 & next(pc) = l2) | (pc = l2 & next(pc) = l3 & current_x != l2)
     | (pc = l4 & next(pc) = l5 & current_x != l3)): 0;
     pc = l1 & next(pc) = l2: 1;
     1: eqn_x_l2;
   esac;

  next(eqn_x_l3) :=
    case
     ((pc=l6 & next(pc) = l2) | (pc = l2 & next(pc) = l3 & current_x != l2)
     | (pc = l4 & next(pc) = l5 & current_x != l3)): 0;
     pc = l2 & next(pc) = l3 & current_x=l2:1;
     1: eqn_x_l3;
   esac;

  next(eqn_x_l5) :=
    case
     ((pc=l6 & next(pc) = l2) | (pc = l2 & next(pc) = l3 & current_x != l2)
     | (pc = l4 & next(pc) = l5 & current_x != l3)): 0;
     pc = l4& next(pc) = l5& current_x=l3:1;
     1: eqn_x_l5;
   esac;

 next(current_x):=
  case
    next(conflict)=none:l0;
    pc = l1 & next(pc) = l2:l2;
    pc = l2 & next(pc) = l3:l3;
    pc = l4 & next(pc) = l5:l5;
    pc = l4 & next(pc) = l6:l4;
    pc = l6 & next(pc) = l2:l2;
    1:current_x;
  esac;

FAIRNESS ! (conflict=all)
```

Table 2. NuSMV model of the observer for conflict 1. Input `pc` is the location of abstraction $\hat{P}$.