

Incremental False Path Elimination for Static Software 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 a novel approach for removing false positives in static program analysis. We present an incremental algorithm that investigates paths to failure locations with respect to feasibility. The feasibility test is done by interval constraint solving over a semantic abstraction of program paths. Sets of infeasible paths can be ruled out by enriching the analysis incrementally with observers. Much like counterexample guided abstraction refinement for software verification our approach enables to start static program analysis with a coarse syntactic abstraction and use richer semantic information to rule out false positives when necessary and possible. Moreover, we present our implementation in the Goanna static analyzer and compare it to other tools for C/C++ program analysis.

1 Introduction

One technique to find bugs in large industrial software packages is static program analysis. While it has been proven to be scalable and fast, it typically suffers to one degree or another from potential false positives. The main reason is that unlike software model checking, static program analysis typically works on a rather abstract level, such as control flow graphs (CFG) without any data abstraction. Therefore, a syntactic model of a program is a coarse abstraction, and reported error paths can be spurious, i.e. they may not correspond to an actual run in the concrete program.

In this paper we present an incremental algorithm that automatically investigates error paths and checks if they are infeasible. To do so, semantic information in the form of interval equations is automatically added to a previously purely syntactic model. This semantic information is only incorporated once a potential bug has been detected, i.e., a counterexample generated. While this ensures scalability and speed for the bug-free parts of a program, it allows to drill down further once a bug has been found.

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

The feasibility of a counterexample is analyzed using constraint solving over the interval domain, i.e., the possible ranges of all variables within a given path. 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 [1, 2]. A main difference is the use of the *precise least solution* for interval equations [3] in the place of SAT-solving. This technique deals directly with loops, without the need to discover additional loop-predicates [2, 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 a *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]. 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 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 Basic Definitions and Concepts

2.1 Labeled Transition Systems

This paper uses *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. An LTS is *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 $s_0 \in S_0$ 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 $P_\times = (S_1 \times S_2, S_{1_0} \times S_{2_0}, A, R_\times, F_1 \times F_2)$ 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 $I = \{\emptyset\} \cup \{[z_1, z_2] \mid z_1 \in \mathbb{Z} \cup \{-\infty\}, z_2 \in \mathbb{Z} \cup \{\infty\}, z_1 \leq z_2\}$ 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. Moreover, we consider the following operators on intervals: addition (+), multiplication (\cdot), union \sqcup , and intersection \sqcap with the usual semantics $\llbracket \cdot \rrbracket$ defined on them.

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

An *interval equation system* is a mapping $IE : X \rightarrow \mathcal{C}(X)$ from interval variables to interval expressions. We also denote this by $x_i = \phi_i$ where $i \in 1, \dots, n$. The *solution* of such an interval equation system is a valuation satisfying all equations, i.e., $\llbracket x_i \rrbracket = \llbracket \phi_i \rrbracket_v$ for all $i \in 1, \dots, n$. As shown in [3] 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 is to map a C/C++ program to its CFG, and to label it with occurrences of syntactic constructs of interest. The CFG together with the labels are mapped to, either a Kripke structure, or to the input language of a model checker, in our case NuSMV.

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

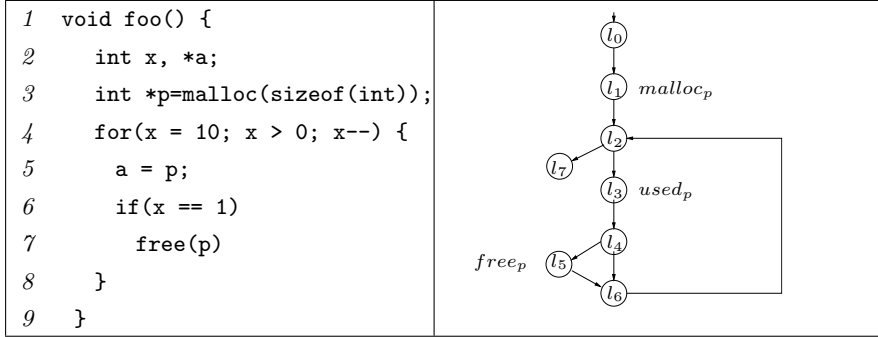


Fig. 1. Example program and labeled CFG for use-after-free check.

to `a`, and freeing the memory in the last loop iteration. 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. To check whether after freeing some allocated resource, it is not used afterwards, we can check 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 an alarm in this example, which is a false positive as p only gets freed in the last loop iteration.

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 $(L, l_0, X, E, update)$, with

- a finite set of locations L ,
- an initial location l_0 ,
- a set of interval variables X ,
- a finite set of edges $E \subseteq L \times L$, and
- an effect function $update : E \rightarrow (X \times \mathcal{C}(X))$.

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 not restrict the expressivity of an IA.

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

- the set of states S contains all states (l, v) with location l , and an interval valuation v .
- the set of initial states S_0 contains all states $s_0 = (\mathbf{l}_0, v_0)$, with $v_0 \equiv [-\infty, \infty]$.
- the alphabet A is the set of edges E .
- the transition relation $R \subseteq S \times A \times S$ contains a triple $((l, v), (l, l'), (l', v'))$, i.e. a transition from state (l, v) to (l', v') labeled (l, l') , if there exists a (l, l') in E , such that $v' = v[lhs(e) \leftarrow \llbracket rhsexpr(e) \rrbracket_v]$ and $\llbracket rhsexpr(e) \rrbracket_{v'} \neq \emptyset$.
- the set of final states $F = S$, 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 $((l_0, l_1), (l_1, l_2), \dots, (l_{m-1}, l_m))$ in the remainder by the sequence of locations (l_0, \dots, l_m) .

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

$$l_0 = \mathbf{l}_0 \tag{1}$$

$$\wedge \forall i = 0, \dots, n-1. (l_i, l_{i+1}) \in E \tag{2}$$

$$\wedge v_0 \equiv [-\infty, +\infty] \wedge \exists v_1, \dots, v_n. (\llbracket rhsexpr(l_i, l_{i+1}) \rrbracket_{v_i} \neq \emptyset \wedge v_{i+1} = v_i[lhs(l_i, l_{i+1}) \leftarrow \llbracket rhsexpr(l_i, l_{i+1}) \rrbracket_{v_i}]) \tag{3}$$

This mean that a word (1) starts in the initial location, (2) respects the edge relation E , and (3) that there exists a sequence of non-empty valuations that satisfies the updates associated with the edges. We use this characterization of words as satisfiability problem to generate systems of interval equations that have a non-empty solution only if a sequence (l_0, \dots, 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 $E' = \{(l, (l, l'), l') \mid (l, l') \in E\}$, 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}_ϕ be the language of 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}_ϕ , and produces a counterexample if this is not the case (cf. Section 4.5). If it finds a word in $\mathcal{L}_{\hat{P}}$ that is not in \mathcal{L}_ϕ , 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 $w = (l_0, \dots, l_m)$ in $\mathcal{L}_{\hat{P}}$ satisfies by construction (1) and (2). If a word $w = (l_0, \dots, l_m)$ can be found such that there exists no solution for (3), it cannot be a word of \mathcal{L}_P . In this case we call the word *spurious*.

If we assume in addition to have a procedure to check whether a word is spurious (cf. Section 4.2), we can use these in an iterative loop to check if the infinite LTS of IA P satisfies the property, by checking a finite product of abstraction \hat{P} with observers instead 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 (1) accepts w , and (2) satisfies that all accepted words w' are not in \mathcal{L}_P .
4. Let $\hat{P}_{i+1} := \hat{P}_i \times Obs_w^C$, with $\cdot \hat{P}_i \times Obs_w^C$ is the synchronous composition of \hat{P}_i and the complement of Obs_w . Increment i and goto step 3.

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 which uses NuSMV to model check the finite abstraction \hat{P}_i .

Example. The initial coarse abstraction as CFG in Fig. 1 loses the information that p cannot be used after it was freed. The shortest counterexample in Fig. 1 initializes x to 10 in line 4, enters the `for`-loop, takes the `if`-branch, frees p in

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

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

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

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 .

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 Ω_w be a mapping each $x_l \in writes_w$ the set $\{i | x = lhs(l_{i-1}, l_i) \wedge l_i = l\}$. The system $IE_w : X_L \rightarrow \mathcal{C}(X_L)$ is defined as follows:

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

System IE_w assigns each variable $x_l \in writes_w$ to a union of expressions; one expression for each element in $\Omega_w(x, l)$.

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, $\Omega_w(x_{l_2}) = \{2, 8\}$. 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. \square

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

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

An advantage of creating interval equations as described is that reason 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, which is a concatenation $\alpha\beta\beta\beta\gamma$, has no non-empty solution, if the concatenation $\alpha\beta\beta\gamma$ has. The least solution will be the same.

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 cone-of influence.

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 $w = (l_0, \dots, l_m)$ a fragment $w' = (l'_0, \dots, l'_{m'})$ is a subsequence of w . For fragment w' of a word w we can show the analog of Lemma 1. If the least solution of $IE_{w'}$ is empty, i.e. if the fragment w' is spurious, then the word w is spurious. If there exists a sequence of non-empty valuations v_1, \dots, v_m for w , then they also define a non-empty subsequence of valuations for w' , and these have to be contained in the least solution of $IE_{w'}$.

A word of length $m - 1$ has $m^2/2$ fragments. Rather than checking all of these we can rule out most of these based on two observations. To be useful at least the last edge of a fragment has to result in an empty solution. An *update* in (3) can only result in an empty solution, if there exist an element in \mathcal{I} that is mapped to the empty set by *update*. We call such updates *restricting*. An example of such updates are intersections with constants such as $x' = x \sqcap [1, \infty]$. We can omit all (l_{i-1}, l_i) after the last last restricting update in w' . Similarly, we find that edges with updates that map valuation $v \equiv [-\infty, \infty]$ to $[-\infty, \infty]$, can be omitted from the beginning of a fragment.

Given thus reduced fragment w' , we can reduce $IE_{w'}$ further, by omitting all equations that are not in the cone-of-influence of the last restricting update. We refer to the resulting system of equations as $\overline{IE}_{w'}$. In the remainder we will refer to $\overline{IE}_{w'}$ as the *reduced conflict*, which is uniquely determined by the fragment w' . The reduction to the cone-of-influence, guarantees that $IE_{w'}$ has an empty least solution if $\overline{IE}_{w'}$ has. The converse is not true. However, since we consider

all possible fragments, it is guaranteed that if a set of removed equations result in an conflict, these will appear as reduced conflict of another fragment.

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} . Note, that variable x in equation $\overline{IE}(x_{l_5})$ was substituted by $[-\infty, \infty]$. This expression was generated by the first edge (l_4, l_5) of the fragment $(l_4, l_5, l_6, l_2, l_3)$. The last update before this edge is outside of the scope of the fragment, and hence x was assumed to be $[-\infty, \infty]$, which is a conservative over-approximation.

4.4 Conflict Observer

Given a reduced conflict \overline{IE}_w for a fragment $w = (l_0, \dots, l_m)$, 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 a LTS over the same alphabet E as $LTS(P)$ and \hat{P} .

Definition 3. Given IA $P = (L, \mathbf{l}_0, E, X, update)$ and reduced conflict \overline{IE}_w for a fragment $w = (l_0, \dots, l_m)$, define X_w as the set of all variables $x \in X$ such that $x = lhs(l_{i-1}, l_i)$, for some edge (l_{i-1}, l_i) in w . Observer Obs_w is a LTS with

- set S_{Obs} of states (current, eqn, conflict) with valuation current : $X_w \rightarrow L$, valuation eqn : $writes_w \rightarrow \{unsat, sat\}$, and location conflict $\in \{all, some, none\}$.
- initial state (write₀, eqn₀, loc₀) with current₀ $\equiv \mathbf{l}_0$, eqn₀ $\equiv unsat$, and conflict₀ = none.
- alphabet E
- transition relation $S_{Obs} \times E \times S_{Obs}$.
- a set final states F . A state is final if conflict = all.

Due to the limited space we give an informal definition of the transition relation. The detailed formal definition can be found on [14].

- Variable *conflict* changes its state based on the next state of *eqn*. We have that $conflict'(x_l)$ is *all*, *some*, *none*, if $eqn'(x_l) = sat$ for all, some or no $x_l \in writes_w$, respectively. Once *conflict* is in *all* it remains there forever.
- Variable *eqn* represents $\overline{IE}_w(x_l)$ for $x_l \in writes_w$. The successor $eqn'(x_l)$ will change if $x = lhs(\lambda, \lambda')$ and $\lambda' = l$. If substituting variables x by $x_{current(x)}$ in $rhserpr(\lambda, \lambda')$, results in an expression that does not appear in $\overline{IE}_w(x_l)$, for any x_l , then $eqn'(x_l) = unsat$ for all x_l . Otherwise, the successor $eqn'(x_l)$ is *sat* for matching variables x_l , and remains unchanged for all others. This is a simple syntactic match, achieved by comparing the indices and variables that appear in $\overline{IE}_w(x_l)$ with the values of $current(x)$.
- Variable *current* is used to record for each variable the location of the last update. If *conflict'* is *none*, then $current'(x) = \mathbf{l}_0$. Otherwise, if (λ, λ') is in w , and $x = lhs(\lambda, \lambda')$, then $current'(x) = \lambda'$. Otherwise, it remains unchanged.

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 such which generates an expression $expr(i)$ that appears in $\overline{IE}_w(x_l)$, with $i \in \Omega_w(x_l)$ (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 fix-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 fixpoint 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.

Given a reduced conflict \overline{IE}_w for fragment w we have two properties.

- If a word $w' \in \mathcal{L}_{\hat{P}}$ is accepted by Obs_w , then $w \notin \mathcal{L}_P$.
- Given a fragment w' such that $\overline{IE}_{w'} = \overline{IE}_w$, and a word w'' such that w' is a subsequence, then w'' is accepted by Obs_w .

The first property states that the language of $\hat{P}' = \hat{P} \times Obs_w^C$ contains \mathcal{L}_P . The second ensures that each observer is only constructed once. This is needed to ensure 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 (4). Consequently, there can only exist a finite set of conflicts as defined by (5).

Example. The observer for the first conflict in Fig. 3 accepts a word if a fragment generates 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. This would be, for example, the case for (l_2, l_3) if $current(x) \neq l_2$. This would create an expression different from $x_{l_2} \sqcap [1, \infty]$, and thus potentially enlarge the solution set.

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 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.

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 proper a abstractions

of $LTS(P)$. The results language inclusion would therefore extend also 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 LTL¹. 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 $G\neg(\text{conflict} = \text{all})$.

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 $\mathbf{x}=(\mathbf{x}+\mathbf{y})*5$ 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 $\mathbf{x}<5$ we introduce two equations, one for the true-case and one for the false-case. Condition $\mathbf{x}<5$ 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 [15].

Joins are introduced where 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 which we cannot accurately cover we simply over-approximate their possible effect. Function calls, for example, are handled as conservatively as possible, i.e., $\mathbf{x}=\mathbf{foo}()$ 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 which 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	+	+	+	+	-	+	+	+	+	+
Blast	+	+	+	+	+	+	+	+	+	+
Satabs	+	+	+	+	+	+	+	+	+	+
CBMC	+	+	+	+	+	+	+	+	+	+

	no violation					violation				
	P1	P2	P3	P4	P5	P1	P2	P3	P4	P5
Goanna	2	2	3	2	1	2	2	4	3	2
Blast	3	3	4	12	12	11	101	5	12	10
Satabs	4	4	1	1	21	10	100	3	1	19
CBMC	10	100	1	10	10	11	101	1	10	11

Table 1. The table on the left-hand side shows for each tool if it found a true positive/negative “+” or a false positive/negative “-”. Entries “(-)” and “(+)” refer to warnings that there *may* be an error. The table on the right-hand-side compares the number of iterations of the three software model checkers with Goanna.

5.2 Comparison

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

- P1 The first program is the one discussed throughout the paper. It is similar to the example discussed in [4].
- P2 The second program is identical, except that the loop is initialized to $x=100$.
- P3 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.
- P4 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 .
- P5 The last one is similar to the first program. This program uses however two counter-variables x and y . The correctness of the program depends on the loop-invariant $x = y$. It is similar to the examples presented in [16].

For each of the programs we constructed one instance with a bug, and one without. For the first program P1, e.g. the loop condition was changed to $x >= 0$. Since not all tools check for the same properties, we introduced slightly different bugs for the different tools, which however all depended on the same paths.

The results in Table 1 show the 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. The software model checking tools in contrast always give the correct result. Our proposed path reduction produces one false positive, for program P5. The least solution of the interval equation shows that both variables take values in the same interval, but the solver is unable to 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

² These can be found on <http://www.cse.unsw.edu.au/~ansgar/fpe/>

400 MHz ECC memory. The following table gives the maximal, minimal, mean and median run-times in seconds:

	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
Blast	58.770	0.126	6.371	0.529
Satabs	5374.037	0.076	539.694	0.336
CBMC	0.221	0.079	0.137	0.121

The static analysis tools are overall fast, and their run times are almost independent of the example program. For test programs as small as these the run-time reflect mostly the time for overhead and setup. The run-times for the model checkers vary more, the maximum is attained for the instance of program P2 ($x=100$) that contains a bug. Satabs in particular showed exponential run times. The default for Satabs is to abort after 50 iterations. It takes about 460 seconds to reach this limit. After doubling the maximal number of iterations to 100, the runtime increases more than 10 times. Memory is not a problem, as Satabs never exceeds 5% of the available memory. Blast shows also exponential run-times, but less pronounced. The run-times for CBMC, in contrast, grow about linearly with an increasing number of iterations. When the loop is initialized to $x=1000$ it takes, for example, about 2.4 sec to complete all 1000 iterations. With Goanna the run time for program P1 and P2 are independent of the number of loops, as expected. Goanna benefits from the fact that interval solving deals efficiently with loops without unrolling [3].

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 software model checkers and static

program analyzers. This demonstrated that Goanna is typically more precise than standard program analyzers and almost as precise as static model checkers. Of course, the design and strength of software model checkers is to check for a much richer class of properties.

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.

References

1. Henzinger, T., Jhala, R., Majumdar, R., SUTRE, G.: Software verification with BLAST. In: Proc. SPIN2003. LNCS 2648 (2003)
2. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Proc. TACAS 2005. LNCS 3440 (2005)
3. Gawlitza, T., Seidl, H.: Precise fixpoint computation through strategy iteration. In: ESOP. LNCS 4421, Springer Verlag (2007)
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)
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 2007. 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: FMCO 2006. LNCS 4111 (2006)
11. Schmidt, D.A., Steffen, B.: Program analysis as model checking of abstract interpretations. In: Proc. SAS '98, Springer-Verlag (1998)
12. Fehnker, A., Clarke, E., Jha, S., Krogh, B.: Refining abstractions of hybrid systems using counterexample fragments. In: Proc. HSCC 2005. LNCS 3414 (2005)
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. Fehnker, A., Huuck, R., Rauch, F., Seefried, S.: Counterexample guided path reduction. Technical Report (number to be assigned), NICTA (January 2008)
15. Ermedahl, A., Sjödin, M.: Interval analysis of C-variables using abstract interpretation. Technical report, Uppsala University (December 1996)
16. Jhala, R., McMillan, K.L.: A practical and complete approach to predicate refinement. In: Proc TACAS. LNCS 3920 (2006)