Using Loop Bound Analysis For Invariant Generation

Pavel Čadek*, Clemens Danninger*, Moritz Sinn†, Florian Zuleger*

*TU Wien
†St. Pölten UAS

***Abstract*—The problem of loop bound analysis can conceptually be seen as an instance of invariant generation. However, the methods used in loop bound analysis differ considerably from previous invariant generation techniques. Interestingly, there is almost no previous work comparing the two sets of techniques. In this paper, we show that loop bound analysis methods can efficiently produce invariants which are hard to prove for state-of-the-art invariant generation techniques (e.g., polynomial invariants or invariants relating many variables) and thus enrich the tool-set of invariant analysis.**

I. Introduction

In this paper we aim at connecting two fields of program analysis: invariant generation and loop bound analysis. Specifically, we suggest the use of loop bound analysis techniques for invariant generation. *Invariant generation* is a traditional discipline with a long history. Invariants are program properties (usually given as formulas over program variables) holding on a specific program location in each program run. A special case of interest are *loop invariants* which hold before and after each loop iteration. For example, in the program from Figure 1(a), we have the loop invariant $x + c \leq n \ \wedge \ x \geq 0$. *Loop bound analysis* is a younger field, where most of the research was done in the last decade. Its goal is to find an upper bound on the number of iterations of a given loop inside a program. *Reachability bound analysis* [9] generalizes the problem to finding an upper (or lower) bound on the number of executions of a specific part of a program (e.g., a branching inside the loop). For example, in the program from Figure 1(a), n is a bound of the only loop, as well as a reachability bound of basic block B_1. We will use the term *bound analysis* to cover both, loop and reachability bound analysis.

Invariants and loop bounds are linked: 1) Invariants can be used to infer loop bounds. A straight-forward idea is to introduce a counter variable c for the loop of interest and to compute an invariant of shape $c \leq bound$. While this idea only works for simple loops, more elaborated approaches have been proposed in the literature [8]. 2) Loop bounds can be used to infer invariants. We are not aware of any publication devoted to this point except for the brief discussions in [18] and [16]. In this paper, we address this gap and also show that invariant generation using loop bound analysis techniques can be more effective than state-of-the-art methods.

We illustrate the use of bound analysis for invariant generation on example (a) in Figure 1. The invariant $c \leq n$ holding after the loop is hard to prove for state-of-the-art invariant analysis approaches, because of their need to derive suitable loop invariants. Here, we specifically need the loop invariant $x + c \leq n \ \wedge \ x \geq 0$. Its problematic part is the relation $x + c \leq n$. Since it does not syntactically appear in the program, it is hard to discover for template-based [15] or predicate-abstraction [12] approaches, because they need to rely on heuristics for template/predicate selection. In contrast, current abstract interpretation based approaches usually fix the expressible invariants in advance: the popular octagon abstract domain [14] cannot express the loop invariant (it can relate at most two variables); the polyhedra domain [4] can express it, but needs to be carefully controlled in order to scale to larger problems.

The central idea of using bound analysis for invariant generation is that variable values after a loop are determined by their values before the loop and the number of times they are increased or decreased inside the loop. In our example, we obtain the equation

$$Post^{\uparrow}(c) = Pre^{\uparrow}(c) + Exec^{\uparrow}(B_1) \cdot 1, \tag{1}$$

where $Post^{\uparrow}(c)$ (resp. $Pre^{\uparrow}(c)$) denotes an upper bound of c after (resp. before) the loop and $Exec^{\uparrow}(B_1)$ denotes an upper bound on the number of executions of the basic block B_1 (containing the instruction `c++`). We note that equation 1 is just a different representation of the postcondition $c \leq Pre^{\uparrow}(c) + Exec^{\uparrow}(B_1) \cdot 1$. Hence, in order to prove the postcondition $c \leq n$, it suffices to compute $Pre^{\uparrow}(c) = 0$ and $Exec^{\uparrow}(B_1) = n$. $Pre^{\uparrow}(c) = 0$ is determined from the precondition $c = 0$. The computation of $Exec^{\uparrow}(B_1)$ is where loop bound analysis comes into play, because the number of executions of block B_1 is the same as the number of loop iterations. The loop bound is inferred in the following way: Variable x is greater than 0 in the beginning of every loop iteration and it is decremented by 1 in every iteration, which means that the maximal value of x in the beginning (which is n) is an upper bound on the number of iterations. In this way, we get $Exec^{\uparrow}(B_1) = n$.

In this paper we make the following contributions:

1) We introduce a benchmark of challenging invariant generation tasks, which we took from previous invariant and loop bound analysis evaluations. We argue that these tasks are difficult for state-of-the-art invariant analysis techniques.

	a	b	c	d	e	f	g
CPAchecker	true	unk	true	t/o	unk	unk	unk
Pagai	true	unk	unk	true	unk	unk	unk
Veriabs	true	t/o	true	t/o	t/o	t/o	true
Aligator	succ	succ	fail	fail	fail	fail	fail

TABLE I
Results of the evaluation on the examples from Figure 1.

2) We present the essence of the techniques underlying bound analysis by introducing a few simple concepts. We define the concepts such that they can be easily used for generating invariants and illustrate their usage on the tasks from our benchmark of challenging examples. The concepts are sufficient to solve all benchmark tasks. We believe that the concepts will enrich the tool set of invariant analysis.
3) We provide experimental evaluations on two large benchmark sets. Our first experiment is executed on part of the SV-COMP 2018 benchmark and demonstrates that the current invariant analysis techniques can be significantly improved by means of bound analysis. Our second experiment is executed on a large industrial benchmark, it shows that the class of invariants that can be verified by state-of-the-art invariant analysis tools is to a large extent different from the class of invariants that is found by bound analysis.

II. Challenges for State-of-the-Art Invariant Generation

In this section, we introduce our small benchmark of invariant generation tasks. The tasks are given in Figure 1. They model some of the invariant generation challenges, which we found in SV-COMP [22] - category ”Loops” (tasks (a), (b), (c), (e), (g)) or cBench [21] (tasks (d), (f)). They consist of a precondition, a while-loop written in a simple imperative C-like language, and the postcondition to be proven.

a) Challenges: In order to prove the postcondition, state-of-the-art invariant generation techniques typically need to infer loop invariants (properties holding before and after each iteration of a loop). We present three main challenges of our benchmark tasks [1]:

1) *Polynomial invariants:* Some part of the loop invariant is a polynomial inequation (resp. equation).
2) *Invariants with more than 2 variables:* Some part of the loop invariant is an inequation (resp. equation) relating more than two program variables.
3) *Disjunctive invariants:* The loop invariant requires a case distinction (e.g., $\mathbf{max}\{x, y\}$).

Next to each example in Figure 1, we state the loop invariant needed for proving the postcondition. Note that the loop invariants are often more complex than the postconditions.

b) Experimental Results: We have evaluated several state-of-the-art invariant generation tools on our benchmark of challenging examples. Pagai (git revision 16eed0f) [11] uses abstract interpretation with linear domains (interval, octagon, polyhedra) and path focusing. CPAchecker 1.6.12 combines several analysis in different modes. We used the predicate abstraction mode which worked the best on the benchmark. Veriabs [5], the winner of subcategory ReachSafety-Loops in SV-COMP 2018, abstracts loops by static value analysis with loop acceleration and k-induction and then uses bounded model checking to prove properties. Aligator [13] (git revision eb79fef) is a representative of polynomial loop invariant generation. The technique is built on recurrence equations.

The input C-programs for the tools were generated by introducing ”assume” resp. ”assert” statements representing the pre- resp. post-condition. E.g., for example (a), we generated the statements `assume(0<=m && m<=x && x<=n && c==0)` and `assert(m<=c && c<=n)`. For Aligator, we had to manually rewrite the examples into its input format and as Aligator only generates loop invariants, we could not include the precondition and postcondition.

Table I shows the results: ”unk” stands for an unknown result, ”true” for a successful proof of the assertion, and ”t/o” for timeout - 60 seconds. A special case is Aligator. Because it only generates the loop invariants, we distinguish two cases: ”succ” for successfully generating a loop invariant which is, together with the precondition, sufficient for proving the postcondition, and ”fail” otherwise. The experiments were performed on a Linux system with an Intel dual-core 3.2 GHz processor using 1.5 GB memory. Regarding the timeouts, the tools did not finish computation even when we extended the limit to 5 minutes, so we consider such cases as failures.

The experimental results support our hypothesis that Figure 1 represents classes of problems which are difficult for state-of-the-art invariant generation techniques. In contrast, in Sections IV and V we will present simple concepts of bound analysis which suffice for analyzing the programs of Figure 1.

III. Basic Definitions

Program representation. Programs in our examples are while-loops without function calls (but with possible nesting) written in C, together with a precondition that holds before the loop. The conditions that we are not able to model or which are non-deterministic (e.g., depending on user input) are represented by the symbol $\star$.

Program Variables and States. By $\mathcal{V}$ we denote the finite set of *program variables* and by $\mathcal{C}$ its subset of *constant variables*, i.e. variables that are never altered in the program. For simplicity, we work only with integers.

A program *state* is a function $\sigma : \mathcal{V} \to \mathbb{Z}$ mapping program variables to their values. We denote the set of states by Σ.

Expressions and Conditions. *Expressions* are terms built with program variables, integers, and functions $+$, $-$, $\cdot$, $/$, $\mathbf{max}$, and $\mathbf{min}$. Division by default rounds down. We denote division with rounding up semantics by wrapping it with the brackets $\lceil . \rceil$. The set of expressions is denoted by Expr. A *constant expression* is an expression that does not contain any non-constant program variable. We denote the set of constant expressions by Expr^c.

[1]Although there is a variety of invariant generation techniques tackling with these challenges, they are either computationaly expensive or rely on heuristics. For a lack of space, we omit a detailed discussion about the techniques and their drawbacks.

	precondition	code	postcondition	loop invariant	concepts
a	$0 \leq m \leq x \leq n \wedge$ $c = 0$	`while(x>0){` B_1: `x--; c++;` `}`	$m \leq c \leq n$	$x + c \leq n \ \wedge$ $x + c \geq m \ \wedge$ $x \geq 0$	RF (IV-A) MF (IV-B) VB1 (V-A)
b	$n \geq 0 \wedge m > 0 \wedge$ $x = n \wedge c = 0$	`while(x>0){` B_1: `x=x-m; c++;` `}`	$c = \lceil \frac{n}{m} \rceil$	$x + m \cdot c = n \wedge$ $x \geq 1 - m$	RF (IV-A) MF (IV-B) VB1 (V-A)
c	$0 \leq x \leq n \wedge$ $c = 0 \wedge$ $y = 3$	`while(x>0){` B_1: `x--; c++;` `if(⋆)` B_2: `y=c;` `}`	$y \leq \mathbf{max}\{3, n\}$	$x + c \leq n \wedge$ $x \geq 0 \wedge$ $y \leq \mathbf{max}\{3, n\}$	RF (IV-A) VB2 (V-B)
d	$n \geq 0 \wedge x = n \wedge$ $c = 0 \wedge r = 0$	`while(x>0){` B_1: `x--;` `if(⋆)` B_2: `r++;` `else` `while(r>0){` B_3: `r--; c++;}` `}`	$c \leq n$	$c + r + x \leq n \wedge$ $r \geq 0 \wedge$ $x \geq 0$	LRF (IV-C) VB1 (V-A)
e	$0 \leq x \leq n \wedge$ $m \geq 0 \wedge$ $y = c = 0$	`while(x>0){` B_1: `x--; y=m;` `while(y>0){` B_2: `y--; c++;` `}` `}`	$c \leq m \cdot n$	$c \leq m \cdot (n - x) \wedge$ $x \geq 0$	LRF (IV-C) VB1 (V-A)
f	$n \geq 0 \wedge$ $r = y = n \wedge$ $c = x = 0$	`while(y>0){` B_1: `x=r;` `while(y>0 && ⋆){` B_2: `x++; y--; }` `while(x>0 && ⋆){` B_3: `x--; r--; c++;` `}` B_4: `y--;` `}`	$c \leq 2n$	$c + \mathbf{max}\{r, x\} + y \leq 2n$ $\wedge\ y \geq 0 \ \wedge \ x \geq 0$	LRF (IV-C) VB1 (V-A)
g	$x \geq 0 \ \wedge \ j = 0 \wedge$ $i = 0$	`while(i<x){` B_1: `i++; j=j+i;` `}`	$j \leq \frac{(x-1) \cdot x}{2}$	$j \leq \frac{(i-1) \cdot i}{2} \wedge$ $i \leq x$	RF (IV-A) VBRE (V-C)

Fig. 1. Our small invariant generation benchmark. Each task consists of a precondition, a while-loop written in C, and the postcondition to be proven. The symbol $\star$ is used to abstract from some conditions in the programs, it represents a non-deterministic boolean value (e.g., dependent on a user input). Each label B_i denotes the basic block (sequence of assignments) associated with the respective line. For each program, we also state the loop invariant needed for state-of-the-art invariant generation techniques to prove the postcondition. The last column states combinations of concepts from Sections IV and V which are sufficient to prove the postcondition.

For expressions $e_1, e_2 \in \mathit{Expr}$ and a variable v, $e_1[v/e_2]$ is the expression e_1 where all occurrences of v are simultaneously replaced by e_2. Further, $e[v_i/e_i \mid i \in I]$ denotes multiple simultaneous replacements.

We can now extend the notion of a program state to whole expressions. For an expression $e \in \mathit{Expr}$ and a state $\sigma \in \Sigma$, we define $\sigma(e) = e[x/\sigma(x) \mid x \in \mathcal{V}]$. We say that $\sigma(e)$ is the *value of e in state σ*.

Conditions (except the non-deterministic condition $\star$) are formulas built from expressions and classical relational and logical operators. The set of conditions is denoted by Cond with Init being the precondition. We extend the concept of a simultaneous replacement from expressions to conditions. We say that a state σ *satisfies a condition* γ (denoted by $\sigma \models \gamma$) if $\gamma[x/\sigma(x) \mid x \in \mathcal{V}]$ is a tautology.

Basic Blocks. A program part consisting only of assignments is called a *basic block*. We denote the set of basic blocks in a program by $\mathcal{B}$. We assume a special *initial basic block* $B_b \in \mathcal{B}$ and *final basic block* $B_e \in \mathcal{B}$, which both consist of zero assignments. We also assume that each block either has a single successor or exactly two successors connected by a branching condition.

We further require that B_b does not have any predecessor and that B_e does not have any successor. In our examples, each basic block is given as one line of code. For example, in the program from Figure 1(a), the basic block B_1 consists of two assignments, `x--` and `c++`. The blocks B_b and B_e are not explicitly marked in our examples.

Program semantics. The *effect of a basic block* is a function $\mathcal{E} : \mathcal{B} \to (\mathcal{V} \to \mathit{Expr})$ such that whenever a block B is executed in a program state σ resulting in a state σ', it holds that $\sigma'(v) = \sigma(\mathcal{E}(B)(v))$.

E.g., if $\mathcal{V} = \{x, y, c\}$, $\mathcal{C} = \{c\}$, and $B =$ `x++; y+=x;`, then $\mathcal{E}(B)(x) = x{+}1$, $\mathcal{E}(B)(y) = y{+}x{+}1$, and $\mathcal{E}(B)(c) = c$.

We assume that for each constant variable $c \in \mathcal{C}$, $\mathcal{E}(B)(c) = c$, i.e., constant variables never change their value.

We extend the effect of a basic block to expressions as follows: For $e \in \mathit{Expr}$, $\mathcal{E}(B)(e) = e[x/\mathcal{E}(B)(x) \mid x \in \mathcal{V}]$

A *program run* is a (possibly infinite) sequence $(\sigma_0, B_0), (\sigma_1, B_1), \ldots,$ where each $(\sigma_i, B_i) \in \Sigma \times \mathcal{B}$, $B_0 = B_b$, $\sigma_0 \models \mathit{Init}$, each B_{i+1} is a successor of B_i, $\sigma_i \models \gamma$ for any branching condition γ between B_i and B_{i+1}, and for all $v \in \mathcal{V}$ we have $\sigma_{i+1}(v) = \sigma_i(\mathcal{E}(B_i)(v))$. Further, if the program run is finite with B_n being the last basic block, then $B_n = B_e$.

Execution bounds. Given a program run ρ, $\#(B, \rho)$ denotes the number of occurrences of the basic block B in ρ.

An *upper (resp. lower) execution bound for a basic block* B is a constant expression $b \in \mathit{Expr}^c$ such that for each program run $\rho \equiv (\sigma_0, B_0), (\sigma_1, B_1) \ldots,$ $\#(B, \rho) \leq \sigma_0(b)$ (resp. $\#(B, \rho) \geq \sigma_0(b)$).

An execution upper (resp. lower) bound mapping is a function $\mathit{Exec}^{\uparrow} : \mathcal{B} \to \mathit{Expr}^c$ (resp. $\mathit{Exec}^{\downarrow} : \mathcal{B} \to \mathit{Expr}^c$) that maps an upper (resp. lower) bound to each basic block in the program. E.g., $\mathit{Exec}^{\downarrow}(B_1) = m$ and $\mathit{Exec}^{\uparrow}(B_1) = n$ in Figure 1(a). [2]

Invariants and expression/variable bounds. Let $B \in \mathcal{B}$ be a basic block. We say a condition $\gamma \in \mathit{Cond}$ is an *invariant before* B if for each program run $(\sigma_0, B_0), (\sigma_1, B_1) \ldots$ holds that if $B_i = B$ then $\sigma_i \models \gamma$. For example, $x > 0$ is an invariant before B_1 in Figure 1(a).

A *precondition* (resp. *postcondition*) is an invariant before B_b (resp. B_e). We use the term *universal invariant* to denote the invariant holding before each $B \in \mathcal{B}$.

An *initial, resp. final, resp. universal upper bound of an expression* e is a *constant* expression $b \in \mathit{Expr}^c$ such that $b \geq e$ is a precondition, resp. postcondition, resp. universal invariant.

An *initial, resp. final, resp. universal upper bound mapping* is a partial function $\mathit{Pre}^{\uparrow}$, resp. $\mathit{Post}^{\uparrow}$, resp. $\mathit{Univ}^{\uparrow}$ that maps an initial, resp. final, resp. universal upper bound to each expression.

We define the initial, resp. final, resp. universal *lower* bound analogically with the upper bounds (only the invariant is $b \leq e$ instead of $b \geq e$) and the bound mappings are denoted as $\mathit{Pre}^{\downarrow}$, $\mathit{Post}^{\downarrow}$, and $\mathit{Univ}^{\downarrow}$.

For example, in Figure 1(a), we have $\mathit{Pre}^{\downarrow}(x) = m, \mathit{Pre}^{\uparrow}(x) = n, \mathit{Univ}^{\downarrow}(x) = 0, \mathit{Univ}^{\uparrow}(x) = n$, and $\mathit{Post}^{\downarrow}(x) = \mathit{Post}^{\uparrow}(x) = 0$.

For all the previous definitions, we use the term *variable bound* in case e is a variable.

Using Loop Bound Analysis For Invariant Generation. In this paper, we extract initial expression bounds from the precondition and use them to infer execution bounds. Based on the execution bounds and initial expression bounds, we compute universal and final variable bounds.

Note that computing final and universal *expression* bounds can usually be decomposed to several *variable* bound computations (whether we take an upper or a lower bound of each variable depends on the sign with which it appears in the expression). For example, we may compute $\mathit{Post}^{\uparrow}(2{\cdot}x{-}y{+}3)$ as $2 \cdot \mathit{Post}^{\uparrow}(x) - \mathit{Post}^{\downarrow}(y) + 3$.

We will describe concepts for execution bound computation in Section IV and concepts for variable bound generation in Section V.

IV. Computation of Execution Bounds

A. Ranking Functions

A lot of techniques for loop bound analysis use the concept of ranking functions (e.g., [3], [17], [18], [2]). Ranking functions are expressions that keep decreasing during an execution of a program and they are bounded from below, thereby proving that the program must eventually terminate.

Look at the program in Figure 1(a): x is a ranking function of block B_1, because (1) it is decreased by 1 with each execution of B_1, (2) it is never increased, and (3) it is bounded from below by 0. Note that in this way it measures the number of the remaining executions of B_1.

Definition 1: Let $\rho \equiv (\sigma_0, B_0), (\sigma_1, B_1) \ldots$ be a program run and $e \in \mathit{Expr}$ an expression. We define $\nabla(e, \rho) = |\{i \mid \sigma_i(e) > \sigma_{i+1}(e)\}|$, so $\nabla(e, \rho)$ denotes the number of times e is decreased on ρ.

Definition 2: An expression e is called a *ranking function of a basic block* $B \in \mathcal{B}$ if for each run $\rho \equiv (\sigma_0, B_0), (\sigma_1, B_1) \ldots$ the following holds:

1) $\#(B, \rho) \leq \nabla(e, \rho)$ (to each execution of B, we can assign at least one decrement of e)
2) $\forall i \geq 0.\ \sigma_i \models e \geq 0$ (e is bounded from below by 0)
3) $\forall i \geq 0.\ \sigma_i(e) \geq \sigma_{i+1}(e)$ (e is never increased)

The right ranking function can be found by simple heuristics. E.g., in [18], the expression e is a candidate if $e > 0$ appears in the looping condition.

[2]We note that the upper and lower bound mappings are not unique. We can infer different mappings depending on the used concept or ranking (resp. metering) function (see next section). However, for technical convenience and better readability we always use the same name $\mathit{Exec}^{\uparrow}$ and $\mathit{Exec}^{\downarrow}$ for the mappings.

Concept RF. Let B be a basic block and e its ranking function. Then $Exec^{\uparrow}(B) = Pre^{\uparrow}(e)$ is an upper execution bound for B.

Example 1: As we already mentioned, x is a ranking function of B_1 in Figure 1(a), so we get $Exec^{\uparrow}(B_1) = Pre^{\uparrow}(x) = n$ by Concept RF. We summarize the results for all the examples in the following table:

	ranking fnct	execution bounds
a	$B_1 :\ x$	$Exec^{\uparrow}(B_1) = Pre^{\uparrow}(x) = n$
b	$B_1 :\ \lceil \frac{x}{m} \rceil$	$Exec^{\uparrow}(B_1) = Pre^{\uparrow}(\lceil \frac{x}{m} \rceil) = \lceil \frac{n}{m} \rceil$
c	$B_1, B_2 :\ x$	$Exec^{\uparrow}(B_1) = Exec^{\uparrow}(B_2) = Pre^{\uparrow}(x) = n$
d	$B_1, B_2 :\ x$	$Exec^{\uparrow}(B_1) = Exec^{\uparrow}(B_2) = Pre^{\uparrow}(x) = n$
e	$B_1 :\ x$	$Exec^{\uparrow}(B_1) = Pre^{\uparrow}(x) = n$
f	$B_1, B_2,$	$Exec^{\uparrow}(B_1) = Exec^{\uparrow}(B_2) = Exec^{\uparrow}(B_4)$
	$B_4 :\ y$	$= Pre^{\uparrow}(y) = n$
g	$B_1 :\ x - i$	$Exec^{\uparrow}(B_1) = Pre^{\uparrow}(x - i) = x$

Note that the expression representing the ranking function of some basic block does not have to decrease by executing the block itself. E.g., in Figure 1(d), x is a ranking function for block B_2 because each execution of B_2 is preceded by an execution of B_1 which decreases x. To find the ranking function of some basic block, it usually suffices to analyse all possible paths from the block back to itself and find expressions that decrease on these paths. Also note that the ranking function does not have to be just one variable (as can be seen in examples (b) and (g)).

We also want to remark that if we delete the initial condition $0 \leq n$ in Figure 1(a) (respectively in the other examples), the analysis does not fail. We would infer the ranking function $\mathbf{max}\{x, 0\}$. The conditions on non-negativity are added to the examples only for simplicity.

B. Metering Functions

Because the research in bound analysis so far focused mainly on computing upper execution bounds, there is no widely adopted term analogical to "ranking function" for computing lower execution bounds. For our paper, we have chosen to adapt (and redefine) the term *metering function* used in [7].

Definition 3: An expression e is called a *metering function of a basic block* $B \in \mathcal{B}$ if for each run $\rho \equiv (\sigma_0, B_0), (\sigma_1, B_1) \ldots$ the following holds:

1) $\#(B, \rho) \geq \nabla(e, \rho)$ (to each decrement of e, we can assign at least one execution of B)
2) $\exists j.\ \sigma_j \models e \leq 0$ (e is eventually non-positive)
3) $\forall i \geq 0.\ \sigma_i(e) \leq \sigma_{i+1}(e) + 1$ (e is never decreased by more than 1 on one block)

Conditions (2) and (3) guarantee that the number of decrements of e is greater or equal to its lowest possible initial value. Because of condition (1), also the number of executions of B is greater or equal to e's lowest possible initial value. Note that the requirement that e is eventually less or equal to zero can be checked by a simple analysis. Usually, it follows from the negated looping condition of the loop. The candidates for metering functions can be chosen by the same heuristics as in the case of ranking functions.

Concept MF. Let B be a basic block and e its metering function. Then $Exec^{\downarrow}(B) = Pre^{\downarrow}(e)$ is a lower execution bound for B.

Example 2: As in the previous subsection, we summarise the metering functions and lower execution bounds in a table:

	metering function	execution bounds
a	$B_1 :\ x$	$Exec^{\downarrow}(B_1) = Pre^{\downarrow}(x) = m$
b	$B_1 :\ \lceil \frac{x}{m} \rceil$	$Exec^{\downarrow}(B_1) = Pre^{\downarrow}(\lceil \frac{x}{m} \rceil)$
		$= \lceil \frac{n}{m} \rceil$
c	$B_1 :\ x$	$Exec^{\downarrow}(B_1) = Pre^{\downarrow}(x) = n$
d	$B_1 :\ x$	$Exec^{\downarrow}(B_1) = Pre^{\downarrow}(x) = n$
e	$B_1 :\ x$	$Exec^{\downarrow}(B_1) = Pre^{\downarrow}(x) = 0$
f	$B_1, B_4 :\ y$	$Exec^{\downarrow}(B_1) = Exec^{\downarrow}(B_4)$
		$= Pre^{\downarrow}(y) = n$
g	$B_1 :\ x - i$	$Exec^{\downarrow}(B_1) = Pre^{\downarrow}(x - i) = 0$

Note that for block B_2 in example (c), x is a ranking function, but not a metering function. For all basic blocks that are not mentioned in the table, we may use the implicit metering function 0, which always satisfies all the conditions of a metering function and leads to the trivial lower execution bound 0.

C. Lexicographic Ranking Functions

We will now extend the concept of a ranking function. Let us look at example (d) in Figure 1: r would be a ranking function of B_3 if it was not incremented on B_2. However, we may notice that B_2 itself has the ranking function x, so B_2 is executed at most $Pre^{\uparrow}(x) = n$ times (by Concept RF) and thus r is increased altogether by at most n (by 1 with each execution of B_2). Hence B_3 cannot be executed more than $Pre^{\uparrow}(r) + n = n$ times. We will call r a local ranking function of B_3.

Definition 4: The expression e is called a *local ranking function of a basic block* $B \in \mathcal{B}$ if for each run $\rho \equiv (\sigma_0, B_0), (\sigma_1, B_1) \ldots$ the following holds:

1) $\#(B, \rho) \leq \nabla(e, \rho)$
2) $\forall i \geq 0.\ \sigma_i \models e \geq 0$

Note that the only difference to Definition 2 is that a local ranking function may increase during a program run (condition (3) is missing).

Definition 5: Let $B_1, \ldots, B_n$ be basic blocks with local ranking functions $e_1, \ldots, e_n$ such that for each $1 \leq i < j \leq n$ it holds that $\mathcal{E}(B_j)(e_i) \leq e_i$ (i.e., we can order the basic blocks such that an execution of any of them does not increase local ranking functions of the blocks that are higher in the order). Then we say that $(e_1, \ldots, e_n)$ is a *lexicographic ranking function of blocks* $(B_1, \ldots, B_n)$.

Concept LRF. Let $(e_1, \ldots, e_n)$ be a lexicographic ranking function of blocks $(B_1, \ldots, B_n)$. Let $c_{i,j} \in \mathit{Expr}^c$ denote the maximal value by which one execution of B_i can increase e_j, i.e. $\mathcal{E}(B_i)(e_j) \leq e_j + c_{i,j}$. Then we set:

$$\mathit{Exec}^{\uparrow}(B_j) = \mathit{Pre}^{\uparrow}(e_j) + \sum_{k=1}^{j-1} \mathit{Exec}^{\uparrow}(B_k) \cdot c_{k,j}$$

More details about lexicographic ranking functions can be found in [17], [3], and [2]. For the computation of lower bounds, nothing like lexicographic metering functions has been published so far. It is possible to define such functions, but we omit the definition here for lack of space.

Example 3: In example (d) from Figure 1, we have the lexicographic ranking function (x, r) of blocks (B_2, B_3). We also have $c_{1,2} = 1$ (otherwise $c_{i,j} = 0$). Hence by applying Concept LRF, we get $\mathit{Exec}^{\uparrow}(B_2) = \mathit{Pre}^{\uparrow}(x) = n$ and $\mathit{Exec}^{\uparrow}(B_3) = \mathit{Pre}^{\uparrow}(r) + 1 \cdot \mathit{Exec}^{\uparrow}(B_2) = 0 + 1 \cdot n = n$.

On this example, we can see how lexicographic ranking functions can handle amortized complexity problems. Even though there are n iterations of the outer loop and B_3 can be executed up to $n-1$ times during one iteration, it cannot be altogether executed more than n times.

For example (e), we infer the lexicographic ranking function (x, y) of blocks (B_1, B_2). We now need an auxiliary invariant $y \geq 0$ holding before B_1 (which is easily found by a simple invariant generator) to infer that the upper ranking function y of B_2 is increased by at most m with each execution of B_1. By applying Concept LRF, we get $\mathit{Exec}^{\uparrow}(B_1) = \mathit{Pre}^{\uparrow}(x) = n$ and $\mathit{Exec}^{\uparrow}(B_2) = \mathit{Pre}^{\uparrow}(y) + m \cdot \mathit{Exec}^{\uparrow}(B_1) = 0 + m \cdot n = m \cdot n$.

On example (f), we can see that the choice of the local ranking functions can have impact on the obtained bounds. When choosing x as a local ranking function of B_3, we get the lexicographic ranking function (y, y, x) of blocks (B_1, B_2, B_3) with $c_{1,3} = n$, $c_{2,3} = 1$, and $c_{1,2} = 0$. Thus $\mathit{Exec}^{\uparrow}(B_3) = \mathit{Pre}^{\uparrow}(x) + n \cdot \mathit{Exec}^{\uparrow}(B_1) + 1 \cdot \mathit{Exec}^{\uparrow}(B_2) = 0 + n \cdot n + 1 \cdot n = n^2 + n$. However, this bound is unnecessarily coarse. When choosing $\mathbf{max}\{x, r\}$ as a local ranking function of B_3[3], the reset `x=r` on B_1 does not have any effect on the local ranking function and we get $c_{1,3} = 0$ and thus $\mathit{Exec}^{\uparrow}(B_3) = \mathit{Pre}^{\uparrow}(\mathbf{max}\{x, r\}) + 0 \cdot \mathit{Exec}^{\uparrow}(B_1) + n \cdot \mathit{Exec}^{\uparrow}(B_2) = n + 0 + n = 2n$.

V. Computation of Variable Bounds

A. A Simple Variable Bound Computation

The simplest approach to variable bound computation was already suggested in the introduction and it follows from [3] and [17]. For a variable that is changed only at one basic block where it is incremented by a constant, we compute the final variable upper bound by multiplying the constant by the maximal number of executions of the block and adding the maximal initial value of the variable.

[3]Both x and r decrease on B_3 so $\mathbf{max}\{x, r\}$ is decremented by 1 with each execution of B_3 and x is always non-negative so also $\mathbf{max}\{x, r\}$ is non-negative.

We now generalise this idea for a computation of lower variable bounds and we involve variables that are not only incremented, but also decremented on possibly more than one basic block. For each variable v, we define a set of tuples (B, d) where B is a basic block and d is the amount by which B increments or decrements v.

Definition 6: Let $v \in \mathcal{V}$. We define the increments and decrements of v in the following way:

$$\begin{aligned} I(v) &= \{(B, d) \in \mathcal{B} \times \mathit{Expr}^c \mid \mathcal{E}(B)(v) = v + d \wedge \\ &\quad\ \mathit{Init} \implies d > 0\} \\ D(v) &= \{(B, d) \in \mathcal{B} \times \mathit{Expr}^c \mid \mathcal{E}(B)(v) = v - d \wedge \\ &\quad\ \mathit{Init} \implies d > 0\} \end{aligned}$$

In the following simple concept, we require that the variable, for which we are computing the bound, cannot be changed in any other way than incrementing and decrementing it by a constant. Note that if there is an assignment incrementing or decrementing the variable by a non-constant expression, we can replace the non-constant expression by its universal upper or lower bound (depending on whether we compute an upper or lower variable bound).

Concept VB1. Let $v \in \mathcal{V}$ and for each basic block B which does not appear in $I(v)$ or $D(v)$ it holds that $\mathcal{E}(B)(v) = v$ (i.e., it does not change v). Then we set

$$\begin{aligned} \mathit{Post}^{\uparrow}(v) &= \mathit{Pre}^{\uparrow}(v) + \textstyle\sum_{(B,d) \in I(v)} \mathit{Exec}^{\uparrow}(B) \cdot d \\ &\quad - \textstyle\sum_{(B,d) \in D(v)} \mathit{Exec}^{\downarrow}(B) \cdot d \\ \mathit{Post}^{\downarrow}(v) &= \mathit{Pre}^{\downarrow}(v) - \textstyle\sum_{(B,d) \in D(v)} \mathit{Exec}^{\uparrow}(B) \cdot d \\ &\quad + \textstyle\sum_{(B,d) \in I(v)} \mathit{Exec}^{\downarrow}(B) \cdot d \\ \mathit{Univ}^{\uparrow}(v) &= \mathit{Pre}^{\uparrow}(v) + \textstyle\sum_{(B,d) \in I(v)} \mathit{Exec}^{\uparrow}(B) \cdot d \\ \mathit{Univ}^{\downarrow}(v) &= \mathit{Pre}^{\downarrow}(v) - \textstyle\sum_{(B,d) \in D(v)} \mathit{Exec}^{\uparrow}(B) \cdot d \end{aligned}$$

Example 4: For example (a) in Figure 1, we have $I(c) = \{(B_1, 1)\}$ and $D(c) = \emptyset$. Thus $\mathit{Post}^{\uparrow}(c) = \mathit{Pre}^{\uparrow}(c) + \mathit{Exec}^{\uparrow}(B_1) \cdot 1 = n$ and $\mathit{Post}^{\downarrow}(c) = \mathit{Pre}^{\downarrow}(c) + \mathit{Exec}^{\downarrow}(B_1) = m$. We may apply the same computation for examples (b), (d), (e), and (f) and use the execution bounds computed in the previous subsections.

We now demonstrate the computation of universal upper and lower bounds for variable r in example (f). We have $I(r) = \emptyset$ and $D(r) = \{(B_3, 1)\}$. We have computed $\mathit{Exec}^{\uparrow}(B_3) = 2n$ in the previous subsection and thus we get $\mathit{Univ}^{\uparrow}(r) = \mathit{Pre}^{\uparrow}(r) = n$ and $\mathit{Univ}^{\downarrow}(r) = \mathit{Pre}^{\downarrow}(r) - \mathit{Exec}^{\uparrow}(B_3) \cdot 1 = n - 2n = -n$.

B. Variable Bound Computation with Resets

We will now extend the previous simple variable bound concept such that it covers also basic blocks which *reset* the given variable, i.e. which set it to a new value independent of its previous value. Assume that we want to compute the final upper variable bound for a variable v. We do not analyse the order in which the blocks are executed, but we can safely assume that all decrements of v happen before any reset (so they would not have any effect on the final value), then v is reset to the biggest possible value and afterwards incremented the maximum number of times. We proceed analogically with the lower bound. The concept is partially inspired by [18].

As in the previous subsection, we first define the set of resets for a given variable. For simplicity, we consider only resets of the form `v=a+expr`, where a is a variable and $expr$ is a constant expression.

Definition 7: Let $v \in \mathcal{V}$. We define the resets of v in the following way:

$$R(v) \quad = \quad \{(B, a, d) \in \mathcal{B} \times \mathcal{V} \times Expr^c \mid \mathcal{E}(B)(v) = a + d\}$$

To formulate the concept, we will yet need two auxiliary sets which contain the largest (resp. smallest) values (given as constant expressions) to which a given variable may be reset. For that purpose, we consider the initial value of a variable also as a reset.

Definition 8:

$$R_c^{\uparrow}(v) = \{Pre^{\uparrow}(v)\} \cup \bigcup_{(B,a,d)\in R(v)} \{Univ^{\uparrow}(a) + d\}$$
$$R_c^{\downarrow}(v) = \{Pre^{\downarrow}(v)\} \cup \bigcup_{(B,a,d)\in R(v)} \{Univ^{\downarrow}(a) + d\}$$

Now we can formulate the concept. Note that (as discussed earlier) decrements have no effect on the upper variable bound and increments have no effect on the lower variable bound. Thus, there is also no difference between final and universal variable bounds here.

Concept VB2. Let $v \in \mathcal{V}$ and for each basic block B that does not appear in $I(v)$, $D(v)$, or $R(v)$ it holds that $\mathcal{E}(B)(v) = v$ (i.e., it does not change v). Then we set

$$Post^{\uparrow}(v) = Univ^{\uparrow}(v) = \begin{array}{l}\mathbf{max}(R_c^{\uparrow}(v)) \\ \quad + \sum_{(B,d)\in I(v)} Exec^{\uparrow}(B) \cdot d\end{array}$$
$$Post^{\downarrow}(v) = Univ^{\downarrow}(v) = \begin{array}{l}\mathbf{min}(R_c^{\downarrow}(v)) \\ \quad - \sum_{(B,d)\in D(v)} Exec^{\uparrow}(B) \cdot d\end{array}$$

Example 5: In example (c) from Figure 1, we have $R(y) = \{(B_2, c, 0)\}, D(x) = \{(B_1, 1)\}, I(c) = \{(B_1, 1)\}$ and $I(y) = D(y) = I(x) = R(x) = D(c) = R(c) = \emptyset$. By Concept VB1, $Univ^{\uparrow}(c) = Pre^{\uparrow}(c) + Exec^{\uparrow}(B_1) = 0 + n = n$, and thus we can compute $R_c^{\uparrow}(y) = \{Pre^{\uparrow}(y), Univ^{\uparrow}(c) + 0\} = \{3, n\}$, and $Post^{\uparrow}(y) = \mathbf{max}\{3, n\}$ by Concept VB2.

C. Variable Bounds by Recurrence Equations

An alternative approach to Concept VB1 and Concept VB2 for variable bound computation is based on recurrence equations. It is similar to the technique [13] used by ALIGATOR.

If we have a non-nested loop, we can express variable values as functions over the loop counter (which represents the number of finished iterations). For example, the fact that a variable v is incremented by 1 in each iteration of a loop can be represented by a recurrence equation $[v](n) = [v](n-1)+1$ where $[v](n)$ denotes the value of v after n iterations (n is here the loop counter).

In comparison with [13], our generated invariants are inequalities instead of equalities and we incorporate the execution bounds, and thus take into account conditions in the loop. The advantage of using recurrence equations over the concepts VB1 and VB2 is that we can achieve more precise bounds (as demonstrated next). The disadvantage is that they are less general - in the way they are defined here, we may apply them only on non-nested loops without branching. However, the concept can be further extended to multi-path or nested loops as in [19].

Definition 9: For $v \in \mathcal{V}$, we introduce the functions $[v]^{\uparrow} : \mathbb{N} \to Expr^c$ and $[v]^{\downarrow} : \mathbb{N} \to Expr^c$ such that $[v]^{\downarrow}(n) \leq v$ and $v \leq [v]^{\uparrow}(n)$ holds after n iterations[4] of the main program loop.

For $n = 0$, we set $[v]^{\uparrow}(n) = Pre^{\uparrow}(v)$ and $[v]^{\downarrow}(n) = Pre^{\downarrow}(v)$. For $n > 0$, we define $[v]^{\uparrow}(n)$ and $[v]^{\downarrow}(n)$ recursively with the use of $[v]^{\uparrow}(n-1)$ and $[v]^{\downarrow}(n-1)$ by analysing the effect of one iteration (the biggest possible increase or decrease of v). Then we can infer the closed form solution from the recursive definitions by a syntactic pattern matching to the following well known case[5]:

$$f(n) = f(n-1)+c+d{\cdot}n \;\rightsquigarrow\; f(n) = f(0)+c{\cdot}n+d{\cdot}\frac{n\cdot(n+1)}{2}$$

where $n \in \mathbb{N}$ and $c, d \in Expr^c$.

Definition 10: We say a function $f : \mathbb{N} \to Expr^c$ is *increasing (resp. decreasing)* if for each $n \in \mathbb{N}$, $f(n) \leq f(n+1)$ (resp.$f(n) \geq f(n+1)$).

Concept VBRE. Let B be a basic block located immediately after the loop header. Let v be a variable for which we know the closed form of $[v]^{\uparrow}(n)$ (resp. $[v]^{\downarrow}(n)$). Then

$$Post^{\uparrow}(v) = \begin{cases} [v]^{\uparrow}(Exec^{\uparrow}(B)) & \text{if } [v]^{\uparrow}(n) \text{ is increasing;} \\ [v]^{\uparrow}(Exec^{\downarrow}(B)) & \text{if } [v]^{\uparrow}(n) \text{ is decreasing.} \end{cases}$$

Analogically for the lower expression bound:

$$Post^{\downarrow}(v) = \begin{cases} [v]^{\downarrow}\langle Exec^{\downarrow}(B)\rangle & \text{if } [v]^{\downarrow}(n) \text{ is increasing;} \\ [v]^{\downarrow}\langle Exec^{\uparrow}(B)\rangle & \text{if } [v]^{\downarrow}(n) \text{ is decreasing.} \end{cases}$$

Example 6: In Figure 1(g), we have $[i]^{\uparrow}(n) = [i]^{\uparrow}(n-1)+1$, hence $[i]^{\uparrow}(n) = [i]^{\uparrow}(0) + n = n$. $[j]^{\uparrow}(n) = [j]^{\uparrow}(n-1) + [i]^{\uparrow}(n-1) = [j]^{\uparrow}(n-1) + n - 1 = [j]^{\uparrow}(0) + (-1) \cdot n + 1 \cdot \frac{n\cdot(n+1)}{2} = \frac{n\cdot(n-1)}{2}$. By Concept RF, we have already inferred $Exec^{\uparrow}(B_1) = x$ in Subsection IV-A. Because $\frac{n\cdot(n-1)}{2}$ is increasing (the loop counter n is non-negative), we get $Post^{\uparrow}(j) = \frac{x\cdot(x-1)}{2}$ by replacing the counter with the upper execution bound.

Note that for the computation of upper variable bound for j with Concept VB1, we would have to replace the assignment `j=j+i` (incrementing j by a non-constant expression) by the

[4]We leave the notion of a loop iteration to the reader's intuition.

[5]Closed form computation of other types of recurrences is discussed, e.g., in [13].

assignment j=j+x (incrementing j by a constant expression). Thus, we would get $I(j) = \{(B_1, x)\}$ and $Post^{\uparrow}(j) = 0 + x \cdot Exec^{\uparrow}(B_1) = x \cdot x$ which is less precise than $\frac{x \cdot (x-1)}{2}$.

VI. Experimental Evaluation on Tasks From SV-COMP

We implemented the presented concepts into the tool LOOPERMAN [19]. We set up the following experiment on base of the SV-COMP 2018 benchmark in order to evaluate the contribution that bound analysis can make to solving invariant analysis challenges: We inserted the invariants that LOOPERMAN computes based on concepts RF, MF, and VB1 in form of "assume" statements into the benchmarks from SV-COMP's ReachSafety-Loops category. Specifically the invariants are added after the loop and immediately after the loop header, where they relate the current variable values to the values before the loop. For example, the code from Figure 1(a) looks as follows after applying the described pre-processing:

```
x_0 = x;
c_0 = c;
while(x>0){
  assume(0<x && x<=x_0);
  assume(0<=c && c<x_0);
  x--;
  c++;
}
assume(x==x_0-max(x_0,0));
assume(c==c_0+max(x_0,0));
```

We excluded the false-unreach cases (those with an invalid assertion) from the benchmark as we aim at proving program properties, not at finding counter-examples, which left us with 111 files with valid assertions. We ran the tools VERIABS, CPACHECKER and PAGAI on the 111 files with valid assertions that we enriched by our invariants, as well as on the original 111 files. We did not run ALIGATOR because its inputs are restricted to a special format. For the evaluation, we used the same time limit of 900s as in SV-COMP. The files with generated invariants, LOOPERMAN, as well as a detailed table of results, is available at [1].

Table II compares the results the respective tool obtains with the help of the invariants inferred by LOOPERMAN (column 2) and without these invariants (column 1). CPACHECKER was able to validate 16 (14.4%) additional assertions with the help of the invariants. PAGAI improved its results by 9 cases (8.1%). Given that VERIABS already proves 103 of 111 assertions, it is hard to further improve its results, and in our experiment it did not not benefit from the additional "assume" statements. However, considering the results of our third experiment (see below), it seems that CPACHECKER and PAGAI are more reliable on real world code than VERIABS.

When comparing to other tools from SV-COMP 2018 on this set of programs, CPACHECKER in predicate analysis mode would move from the 5th place to the 2nd place by using our additional invariants, preceded only by VERIABS with 103 proven files, and followed by UTAIPAN [6] with 82 proven files and UAUTOMIZER [10] with 78 files.

	proven true (without invariants)	proven true (with invariants)
CPACHECKER	68 (61.26%)	84 (75.68%)
PAGAI	57 (51.35%)	66 (59.46%)
VERIABS	103 (92.79%)	103 (92.79%)

TABLE II
RESULTS OF THE EVALUATION ON 111 TRUE-UNREACH PROGRAMS OF REACHSAFETY-LOOPS CATEGORY FROM SV-COMP 2018 WITH AND WITHOUT OUR GENERATED INVARIANTS.

	fail	oom	timeout	unknown	false	true	true (%)
CPACHECKER	98	0	187	165	0	311	40.87%
PAGAI	0	8	36	342	0	375	49.2%
VERIABS	183	0	265	262	10	41	5.4%
CBMC	0	8	474	0	0	279	36.66%

TABLE III
RESULTS OF THE EVALUATION ON ALL 761 LOOPS IN CBENCH FOR WHICH LOOPUS INFERRED A BOUND OVER THE STACK VARIABLES.

We conclude that bound analysis techniques can considerably improve state-of-the-art approaches to invariant analysis.

VII. Experimental Evaluation on an Industrial Benchmark

By our third experiment on an industrial benchmark we evaluate to which extent invariant analysis tools can solve bound analysis problems. For this purpose we ran our bound analysis tool LOOPUS on the program and compiler optimisation benchmark *Collective Benchmark* [21] (cBench, 1027 different C files with 211.892 lines of code) and annotated the inferred bounds as assertions into the code: We instrumented a counter c for each loop and added the assertion $c \leq bound$, where $bound$ is the loop bound computed by LOOPUS. We then asked CPACHECKER, PAGAI and VERIABS to prove these assertions. Since it is to be expected that loop bounds formulated over heap values are difficult to verify, we only considered those bounds which are purely formulated over the stack variables. In this way, we generated 761 assertion tasks. We also ran the bounded model checker CBMC 5.3 [20] on our benchmark in order to check the correctness of the generated assertions (by loop unrolling CBMC can disprove wrong loop bounds in many cases). We chose a timeout of 60s for LOOPUS as well as for the verification tools because increasing the timeout did not improve results significantly, neither for LOOPUS nor the verifiers. The generated files with assertions, the version of LOOPUS which we used, as well as a detailed table of results, is available at [1].

Table III shows the overall results. The column "fail" states the number of loops (assertions about the loop bounds) for which the respective tool crashed, "oom" refers to the cases for which the tool ran out of memory, "timeout" are the cases where the computation exceeded 60 seconds, and "unknown", "false", and "true" are the cases where the tool terminated with results "unknown", "false", or "true" respectively. The result "false" indicates that the tool disproved the bound inferred by LOOPUS. We checked the 10 loops for which VERIABS refuted the asserted loop bound and it turned out that the bound is actually sound.

Interestingly, even though the assertions are usually of a simple form (we used an industrial, not an academic bench-

	fail	oom	timeout	unknown	false	true	true (%)
CPACHECKER	61	0	186	160	0	309	43.16%
PAGAI	0	8	33	300	0	375	52.37%
VERIABS	179	0	247	239	10	41	5.73%
CBMC	0	8	429	0	0	279	38.97%

TABLE IV
RESULTS OF THE EVALUATION ON 761 LOOPS IN CBENCH FOR WHICH LOOPUS INFERRED A LINEAR OR CONSTANT BOUND OVER THE STACK VARIABLES.

mark), the best tool in this comparison, PAGAI, succeeded to prove only 49% of the assertions. The second CPACHECKER proved only 41%, CBMC 37%, and VERIABS 5.4%.

The low success rate is partially caused by the fact that some of the bounds (assertions) are polynomial, which is problematic for the provers, as discussed in Section II. Therefore we state the results restricted to the case of linear or constant bounds in Table IV. Nevertheless, the provers were not much more successful with 52% (PAGAI), 43% (CPACHECKER), 39% (CBMC), and 5.7% (VERIABS) proven assertions.

In conclusion, our experiment demonstrates that in many cases invariants computed by bound analysis cannot be computed by state-of-the-art invariant analysis techniques.

VIII. CONCLUSION

We have formulated simple bound analysis concepts for computing invariants which are challenging for state-of-the-art invariant generation techniques. On a set of tasks from the SV-COMP 2018 benchmark, we have demonstrated that current invariant analysis techniques can be significantly improved by means of bound analysis. Additionally, we have shown by an experimental evaluation on an industrial benchmark that the class of invariants which can be verified by state-of-the-art invariant analysis tools is to a large extent different from the class of invariants that is found by bound analysis. Our results show that using bound analysis techniques for invariant generation is very promising and we hope that they motivate further research in this area.

Acknowledgement: The first and the fourth author were supported by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF). We would also like to thank Marek Chalupa for his help with the experiments.

REFERENCES

[1] `Experimental Evaluation`. http://forsyte.at/static/people/sinn/fmcad2018/.

[2] C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *SAS*, volume 6337 of *LNCS*, pages 117–133. Springer, 2010.

[3] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating runtime and size complexity analysis of integer programs. In *TACAS*, volume 8413 of *LNCS*, pages 140–155. Springer, 2014.

[4] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96. ACM Press, 1978.

[5] P. Darke, S. Prabhu, B. Chimdyalwar, A. Chauhan, S. Kumar, A. Basakchowdhury, R. Venkatesh, A. Datar, and R. K. Medicherla. Veriabs: Verification by abstraction and test generation - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*, pages 457–462, 2018.

[6] D. Dietsch, M. Greitschus, M. Heizmann, J. Hoenicke, A. Nutz, A. Podelski, C. Schilling, and T. Schindler. Ultimate taipan with dynamic block encoding - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*, pages 452–456, 2018.

[7] F. Frohn, M. Naaf, J. Hensel, M. Brockschmidt, and J. Giesl. Lower runtime bounds for integer programs. In *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, pages 550–567, 2016.

[8] S. Gulwani, K. K. Mehra, and T. Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139. ACM, 2009.

[9] S. Gulwani and F. Zuleger. The reachability-bound problem. In *PLDI*, pages 292–304. ACM, 2010.

[10] M. Heizmann, Y. Chen, D. Dietsch, M. Greitschus, J. Hoenicke, Y. Li, A. Nutz, B. Musa, C. Schilling, T. Schindler, and A. Podelski. Ultimate automizer and the search for perfect interpolants - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*, pages 447–451, 2018.

[11] J. Henry, D. Monniaux, and M. Moy. PAGAI: A path sensitive static analyser. *Electr. Notes Theor. Comput. Sci.*, 289:15–25, 2012.

[12] R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4), 2009.

[13] L. Kovács. Reasoning algebraically about p-solvable loops. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 249–264, 2008.

[14] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[15] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *VMCAI*, volume 3385 of *LNCS*, pages 25–41. Springer, 2005.

[16] M. Sinn, F. Zuleger, and H. Veith. Complexity and resource bound analysis of imperative programs using difference constraints. *Journal of Automated Reasoning*, 59(1):3–45.

[17] M. Sinn, F. Zuleger, and H. Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In *CAV*, volume 8559 of *LNCS*, pages 745–761. Springer, 2014.

[18] M. Sinn, F. Zuleger, and H. Veith. Difference constraints: An adequate abstraction for complexity analysis of imperative programs. In *FMCAD*, pages 144–151. IEEE, 2015.

[19] P. Čadek, J. Strejček, and M. Trtík. Tighter loop bound analysis. In *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*, pages 512–527, 2016.

[20] `CBMC`. http://www.cprover.org/cbmc/.

[21] `CBench`. http://ctuning.org/wiki/index.php/CTools:CBench/.

[22] `SV-COMP`. https://sv-comp.sosy-lab.org/.