Electronic Communications of the EASST
Volume 72 (2015)

Proceedings of the
15th International Workshop on
Automated Verification of Critical Systems (AVoCS 2015)

Loop Patterns in C Programs

Thomas Pani, Helmut Veith, Florian Zuleger

15 pages

Guest Editors: Gudmund Grov, Andrew Ireland
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Loop Patterns in C Programs

Thomas Pani, Helmut Veith, Florian Zuleger[*]

Vienna University of Technology

Abstract: In this work, we conduct a systematic study of loops in C programs. We describe static analyses capable of efficiently identifying definite iteration in C code. Our experiments show that over one third of loops in our benchmarks take this form. To cover further loops, we systematically weaken our definition of definite iteration and derive a family of loop classes that are heuristics for definite iteration. We then measure the occurrence of these classes on real-world C code and investigate which statements are used to express them. Finally, we empirically show that our classification is meaningful – (a) it describes the majority of loops in our benchmarks, (b) the classes are good heuristics for termination, and (c) they can be used as software metrics to characterize benchmarks for software verification.

Keywords: Loops, loop patterns, structured programming, definite iteration, software metrics, program features.

1 Introduction

Historically, some programming languages provide restricted loop statements, such as the Fortran `do` statement or the Ada `for` statement. Such statements express *definite iteration*, i.e. structured iteration over the elements of a finite set, such as an integer sequence or the elements of a data structure [Sta95]. Other languages – like C – do not provide such constructs, and all language-provided loop statements have the full expressive power inherent to Turing-complete languages. Lately, object-oriented languages implement definite iteration via the *iterator pattern*, often supported by syntactic sugar (e.g. `foreach`-like statements in Java, C#, C++11, Python). However, we are not aware of a recent study determining if and how often this form of restricted iteration occurs in practice. This is the gap we intend to close.

In this paper, we study "typical" iteration patterns in real-world C code. Such a study is of interest because it allows us – for example – to answer the following questions: How do loops in source code usually "look like" (what are their properties)? How often do different kinds of loops appear in program source code? How difficult are these different kinds of loops to analyze for an automated procedure? In spirit of Dijkstra's famous *Go-to statement considered harmful* [Dij68]: Can practical examples of iteration be expressed using a more well-behaved, structured construct? As we target the C language, which does put any restriction on looping constructs, does it make sense to introduce restricted loop statements like in Fortran or Ada?

To our knowledge, no up-to-date systematic study of definite iteration in real-world code exists. [Sta93] does a manual, ad-hoc classification of iteration over high-level, abstract structures

[*] The authors are supported by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF) and by the Vienna Science and Technology Fund (WWTF) through grants PROSEED, ICT12-059, and VRG11-005.

such as sets, sequences or finite mappings. Our work takes an algorithmic approach, yielding a concise definition and automated classification. [RAPG14] presents a simple, powerful heuristic similar to our work for computing loop trip counts. In contrast, we base our study on a sound analysis, and present extensive experimental results.

We thus aim to study the occurrence of definite iteration in real-world C code. However, there is no dedicated, well-defined construct supporting definite iteration in the C language. We thus describe a pattern-based, lightweight static analysis to identify such loops (Section 2), which we call *FOR loops* $\mathscr{L}^{\mathrm{FOR}}$: We define three easily verifiable restrictions on loops which capture the structured nature of definite iteration. As our experiments (Section 5) show, this allows us to identify about one third of the loops in our benchmarks as FOR loops – a major portion, but not the majority of loops in our benchmarks. We conjecture that programmers enjoy a bit more flexibility than what is provided by our definition of FOR loops.

In an attempt to understand further loop patterns, we extend our analysis to describe loops similar to, but not identified as FOR loops. To do so, we derive the family of *generalized FOR loops* $\mathscr{L}^{\mathrm{FOR}(\dots)}$ from the definition of $\mathscr{L}^{\mathrm{FOR}}$ by systematically weakening the three restrictions (Section 3). This lets us describe up to 82% of the loops in our benchmarks (Section 5). Next, it is natural to look at statement usage: The C language provides four statements of equal expressive power capable of implementing iteration (`while`, `do-while`, `for`, `goto`). We thus measure their use for each of the generalized FOR loop classes (Section 5.1).

Finally, we conduct two experiments to show that the defined loop classes are meaningful: We compare the loops matching our loop classes with the state-of-the-art bound analysis tool LOOPUS (Section 6.1). LOOPUS tries to statically determine a symbolic bound on the number of times a loop is executed. As the number of iterations of a FOR loops is predetermined, loop classes should align with results from bound analysis. We also sketch how to derive software metrics from our loop classification, and that they describe properties interesting for program analysis by applying them in a machine-learning portfolio for software verification (Section 6.2).

Summarizing, our work conducts a systematic study of loops in C programs, making the following contributions:

1. We give a definition of definite iteration, *FOR loops*, for the C programming language, which does not have dedicated support for this concept (Section 2).

2. We define the family of *generalized FOR loops*, which capture some aspects of definite iteration and allow us to describe a majority of loops in our benchmarks (Section 3).

3. We study the occurrence of these loop classes on benchmarks taken from different problem domains (Section 5) by measuring the occurrence of FOR loops and generalized FOR loops, and study which C statements are used to express them (Section 5.1).

4. We give empirical evidence for the usefulness of our loop classes (Section 6):

   (a) We show that the generalized FOR loop classes capture the difficulty of automated program analysis (Section 6.1).

   (b) We sketch how to use the loop classes as software metrics, and how to apply them in a machine-learning portfolio for software verification (Section 6.2, [DPVZ15]).

2 FOR Loops

In this section, we introduce a first loop pattern, *FOR loops* $\mathscr{L}^{\mathrm{FOR}}$, expressing definite iteration. One central implication of FOR loops is that once started executing, they always terminate. We thus characterize FOR loops $\mathscr{L}^{\mathrm{FOR}}$ by giving an efficient, syntactic pattern-based termination proof for its members. This termination procedure exploits that in many cases, local reasoning is powerful enough to decide termination of loops expressing definite iteration. Such loops usually alter a limited set of variables during each iteration, which are then compared against a fixed (loop-invariant) bound to decide termination.

Example. Consider the program on the right: We can show the loop to terminate in a straightforward manner: the value of `i` is changed by the loop, while the value of `N` is fixed. The loop's condition induces a predicate $P(i) : i < N$. If $P(i)$ evaluates to false, the loop terminates. We show that executing the loop, $P(i)$ eventually evaluates to false: The domain of P can be partitioned into two intervals $[-\infty, N)$ and $[N, \infty]$, for which $P(i)$ evaluates to true or false, respectively (q in Figure 2). As `i` is (in total) incremented during each iteration, we eventually have $i \in [N, \infty]$, and thus $\neg P(i)$ and the loop terminates.

```
while (i < N) {
    if (nondet())
        i+=2;
    else
        i+=3;
    i--;
}
```

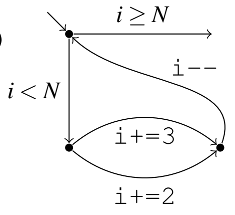


Figure 1: Motivating example, source code and labeled transition system.

2.1 Program Model

We base our analysis on the program's *labeled transition system*:

Definition 1 A *labeled transition system* (LTS) is a digraph $T = (Loc, Labels, Edges, l_0)$ constructed from the program's source code, where Loc is the finite set of program locations, $Edges \subseteq Loc \times Labels \times Loc$ is the transition relation, and $l_0 \in Loc$ is the singleton initial state. $Labels$ is the set of edge labels, consisting of the program's statements, and expressions $assume(a,b)$ corresponding to branches. If a node has a singleton successor, the edge is labeled with the corresponding program statement. Branching is modeled by two successors, where the edge labels $P : assume(a,b)$ and $\neg P : assume(a,c)$ are the predicates guarding control flow from program location a to b and c, respectively.

2.2 Determining membership in $\mathscr{L}^{\mathrm{FOR}}$

To compute the syntactic termination proof for a given loop L, we proceed in the three steps described below: First, we consider predicates $assume(a,b)$ on edges $a \in L, b \notin L$ leaving the loop. We try to show that the predicate *eventually* (during program execution) becomes true, meaning the edge becomes executable (Section 2.2.1). Second, we check that other variables $v \neq i$ occurring in the predicate are loop-invariant (Section 2.2.2). Finally, we impose a control flow constraint to make our termination proof sound (Section 2.2.3).

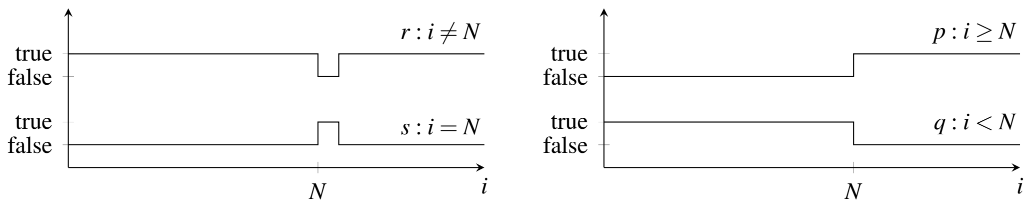


Figure 2: Monotonic (p, q) and eventually true (p, q, r, s) predicates.

Name	Informal motivation	Formal definition
Escape	i "escapes" Exp, i.e. $P(i)$ iff $i \neq Exp$. Example: $i \neq N$	f_P has exactly one non-root, i.e. there exists exactly one x s.t. $f_P(x) = 1$.
Search	i "searches" and eventually "finds" Exp, i.e. $P(i)$ iff $i = Exp$. Example: $i = N$	f_P has a single root, i.e. there exists exactly one x s.t. $f_P(x) = 0$.
Increase	i increases enough to enter interval $[Exp, \infty]$, i.e. $P(i)$ iff $i \in [Exp, \infty]$. Example $i \geq N$	f_P is monotonically decreasing and eventually 0.
Decrease	i decreases enough to enter interval $[-\infty, Exp]$, i.e. $P(i)$ iff $i \in [-\infty, Exp]$. Example $i \leq N$	f_P is monotonically increasing and eventually 0 (f_P grows from 0).

Table 1: Strategies for proving termination based on the representing function f_P of an exit predicate P. We assume expression Exp to be loop-invariant.

2.2.1 Terminating Predicates

To find predicates guarding loop termination, we consider control flow edges leaving the loop:

Definition 2 (Exit node, exit predicate.) Let $T = (Loc, Labels, Edges, l_0)$ be a loop's LTS, (a,b) such that $a \in Loc, b \notin Loc$ an edge leaving the loop, and $P : assume(a,b)$ the edge's label. We call a an *exit node* and P an *exit predicate*.

We introduce a predicate's *representing function* as a means to define necessary characteristics for our termination proof:

Definition 3 (Representing function, monotonicity, eventually true predicates.) The representing function f_P of a predicate P with the same domain takes, for each domain value, the value 0 if the predicate holds, and 1 if the predicate evaluates to false, i.e. $P(X) \Leftrightarrow f_P(X) = 0$. A predicate P is monotonically increasing (decreasing) if its representing function f_P is monotonically increasing (decreasing). A predicate P is *eventually true* if its representing function f_P is eventually 0, i.e. if there exists an x s.t. $f_P(x) = 0$. These definitions are inspired by [KW11].

Depending on the predicate's syntactic form, we describe four strategies for showing that the predicate is eventually true. These strategies were chosen to represent cases that in our experience frequently occur in practice. We illustrate the strategies in Figure 2 and describe them in Table 1. Based on the strategies from Table 1, we call predicates whose representing function takes such a form *well-formed*:

Definition 4 (Well-formed exit predicate.) Let L be a loop, (a,b) an edge leaving L, and P : *assume*(a,b) the edge's label. P is a *well-formed exit predicate* if and only if its representing function f_P matches one of the forms in Table 1.

Strategy	Condition
Escape	$0 \notin AccIncs_L(i)$
Search	$AccIncs_L(i) = \{s\} \wedge s \in \{-1,1\}$
Increase	$\forall inc \in AccIncs_L(i) \,.\, inc > 0$
Decrease	$\forall inc \in AccIncs_L(i) \,.\, inc < 0$

Table 2: Sufficient conditions for an exit predicate's terminating the loop.

Given a predicate's representing function $f_P(i)$, we first determine the appropriate strategy in Table 1. We then consider updates on i in loop L to decide if the strategy's condition holds: For each variable i occurring in an exit predicate, we compute the set of possible updates to i in a single iteration of the loop and obtain the *accumulated increment* $AccIncs_L(i)$. We do so by folding constant increments/decrements of i along a path of the loop into a single, constant value. At nodes where branches join, we build the union of the accumulated increment along each path. In case of non-constant updates, or a nested loop that updates i, we cannot precisely determine the set and $AccIncs_L(i) = \mathbb{Z}$. We use the accumulated increment $AccIncs_L(i)$ to give sufficient conditions for an exit predicate $P(i)$'s eventual truth for a given loop L and variable i in Table 2. Intuitively, the conditions all describe scenarios under which the representing function f_P eventually takes the value 0:

- For **escape**, either $f_P(i)$ is already 0, or any non-zero increment makes it 0 in the next iteration. The condition ensures such a non-zero increment exists along all paths of the loop.
- For **search**, either $f_P(i)$ already 0, or i eventually takes all values in its type's range. The condition ensures all increments are integer successor functions in a single direction. Assuming (cf. Section 4.1) two's complement integer wrap-around on overflow, i steps through all values in its type's range.
- For **increase** (**decrease**), either $f_P(i)$ already 0, or i is incremented (decremented) on all paths. The condition ensures all updates to i along all paths are non-zero increments (decrements).

Definition 5 (Terminating node, terminating predicate.) Let P : *assume*(a,b) be an exit predicate for which the condition given by Tables 1 and 2 holds. We call a a *terminating node* and P a *terminating predicate*.

Example. For our example in Figure 1, we have $AccIncs_L(\texttt{i}) = \{2-1, 3-1\} = \{1,2\}$ and identify an exit predicate $P(i) : i \geq N$. Matching $P(i)$ against Tables 1 to 2 we can see that we need to apply strategy "Increase", i.e. check whether all elements of $AccIncs_L(\texttt{i})$ are positive. Clearly this is the case, and we proceed to check two further constraints.

2.2.2 Invariant Constraint

Our syntactic termination proof only considers predicates $P(i)$ in a single variable i. Other subexpressions are checked to be loop-invariant by verifying that none of the referenced variables are updated inside the loop.

Example. In our example (Figure 1), `N` is never updated inside the loop. The constraint is satisfied and we proceed to check the last constraint.

2.2.3 Control Flow Constraint

Finally, we need to make sure that a terminating predicate $P : assume(a,b)$ is actually evaluated (i.e. node a is reached) when it evaluates to true. Listing 1 illustrates that due to our local reasoning this is not always the case: while there is a terminating predicate $P : i > N$, it may never be evaluated (e.g. if `decide(i)` always returns false).

As determining feasibility of reaching a is in itself a hard problem, we restrict our analysis to a case in which we can ensure soundness: a has to lie on each path through the loop.

Example. In our example (Figure 1), the loop condition (and hence the exit predicate $P(i)$) is evaluated in each iteration of the loop. Thus we classify the loop as FOR loop $L \in \mathscr{L}^{\mathrm{FOR}}$:

Definition 6 (FOR loop.) Given a loop L and an exit predicate $P(i) : assume(a,b)$, we call L a *FOR loop* $L \in \mathscr{L}^{\mathrm{FOR}}$ if and only if the following conditions hold:

- **S_1: Predicate constraint.** $P(i)$ is a terminating predicate, i.e. $AccIncs_L(i)$ implies eventual truth of $P(i)$ (Section 2.2.1).
- **S_2: Invariant constraint.** We only consider predicates $P(i)$ in a single variable i. All other subexpressions are loop-invariant (Section 2.2.2).
- **S_3: Control flow constraint.** a lies on each path through the loop (Section 2.2.3).

2.3 Strengthening Syntactic Termination

So far, we have only considered an isolated notion of loop termination: If execution starts from l_0, any path of execution leaves the loop. A stronger notion considers a loop to be bounded if and only if the number of executions of the loop's paths is bounded. The example below shows where the two notions differ:

```
while (1) { for (unsigned i = 0; i < 42; i++) {} }
```

While the nested loop itself terminates whenever executed, the number of executions of the nested loop is infinite. We can strengthen the notion of syntactic termination to accommodate this property:

Definition 7 (Syntactically bounded loop.) Given a loop $L \in \mathscr{L}^{\mathrm{FOR}}$, we call L *syntactically bounded* $L \in \mathscr{L}^{\mathrm{SB}}$ if and only if L itself and all of its nesting (outer) loops are in $\mathscr{L}^{\mathrm{FOR}}$.

3 Generalized FOR Loops

In this section, we aim for a classification of the remaining loops, i.e. those loops not identified as FOR loops. We develop heuristic loop classes based on $\mathscr{L}^{\mathrm{FOR}}$, by systematically weakening $\mathscr{L}^{\mathrm{FOR}}$ criteria along three dimensions corresponding to the three FOR loop constraints S_1–S_3

(cf. Definition 6). The heuristics are designed to leave enough leeway to match a considerable amount of loops, but still capture some of the termination-related properties of FOR loops. We call this family of loop classes *generalized FOR loops* $\mathscr{L}^{\mathrm{FOR}(\ldots)}$.

3.1 Dimensions of Generalized FOR Loops

3.1.1 Predicate Constraint

Depending on the chosen abstract domain and abstract semantics, the computed set $AccIncs_L(i)$ (cf. Section 2.2.1) may be too coarse to show termination. Thus we aim to cover cases where symbolic updates take sensible values, but our local, path-insensitive analysis cannot establish them. Our heuristic decouples the termination property from predicates by not requiring the accumulated increment to imply eventual truth of the predicate: We alter the **constraint $\mathbf{S_1}$** (Definition 6), which only considers terminating predicates, to **heuristic $\mathbf{W_1}$**, which is content with well-formed predicates (Definition 4).

```
extern int decide(int);
extern int update(int);

int i=0;
while (1) {
  if (decide(i))
    if (i > N)
      break;

  if (decide(i)) N--;
  i = update(i);
  i++;
}
```

Listing 1: Almost a FOR loop: several obstacles for classifying a loop as FOR loop are addressed by generalized FOR loops.

Example. In Listing 1, we cannot determine which values `update(i)` returns. Thus we do not classify the loop under constraint S_1. For constraint W_1, we only consider the syntactic form of $P(i) : i > N$ and do not check the constraint from Table 2. In our example, pattern matching against (Table 1) succeeds, and $P(i)$ fulfills W_1. *Intuition:* We assume `update(i)` returns sensible values.

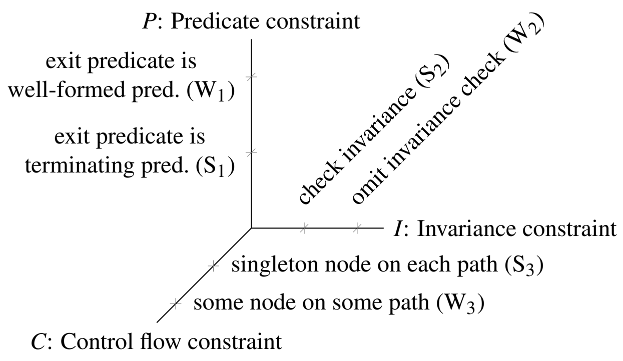
3.1.2 Invariant Constraint

Our syntactic termination proof only considers predicates $P(i)$ in a single variable i. Other subexpressions are checked to be loop-invariant (**constraint $\mathbf{S_2}$**, Definition 6). For our **heuristic $\mathbf{W_2}$**, we omit this check. Intuitively, we allow predicates in multiple variables and assume that all variables $v \neq i$ are updated in a way that does not interfere with termination.

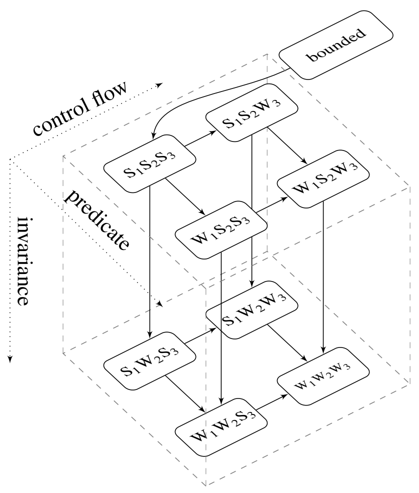
Example. In Listing 1, `N` is not loop-invariant, i.e. we have a predicate in two variables $P(i,N)$. As `N` is updated in the loop, we do not classify the loop under constraint S_2. Heuristic W_2 omits the loop-invariance check, and thus the loop fulfills heuristic W_2. *Intuition:* We assume updates to `N` take sensible values (in our example they do, as `N` is only decremented).

3.1.3 Control Flow Constraint

Instead of requiring the terminating node a associated with a terminating predicate $P : assume(a,b)$ to lie on each path through the loop (**constraint $\mathbf{S_3}$**, Definition 6), **heuristic $\mathbf{W_3}$**: only requires existence of some terminating node in the loop.


P: Predicate constraint
exit predicate is well-formed pred. (W1)
exit predicate is terminating pred. (S1)
check invariance (S2)
omit invariance check (W2)
I: Invariance constraint
singleton node on each path (S3)
some node on some path (W3)
C: Control flow constraint

(a) Dimensions of generalized FOR loops.



(b) Structure of the $\mathscr{L}^{\text{FOR}(\dots)}$ poset.

Figure 3: Dimensions of generalized FOR loops.

Example. Consider the code in Listing 1: Predicate $P(i) : i > N$ is nested inside of a conditional branch. As we cannot determine if `decide(i)` ever evaluates to true when $i > N$, we do not classify the loop under constraint S_3. For heuristic W_3 we only require that there is a terminating node somewhere in the loop. *Intuition:* We assume the node is reached when its associated predicate $P(i)$ evaluates to true.

Definition 8 (Generalized FOR loop constraint, generalized FOR loop.) A *generalized FOR loop constraint* is a tuple (P,I,C), where $P \in \{S_1,W_1\}$ is a predicate constraint, $I \in \{S_2,W_2\}$ is an invariance constraint, and $C \in \{S_3,W_3\}$ is a control flow constraint. A loop L satisfies constraint (P,I,C) if and only if all of P,I,C are satisfied. L is a *generalized FOR loop* $L \in \mathscr{L}^{\text{FOR}(PIC)}$ if and only if L satisfies (P,I,C).

For a visual interpretation, consider Figure 3a showing the three categories above as independent dimensions. Moving away from the center along dimensions P,I,C, we cover additional loops at the expense of losing soundness. Note that the strongest generalized FOR loop class $\mathscr{L}^{\text{FOR}(S_1S_2S_3)}$ is the class of FOR loops introduced in Section 3, i.e. $\mathscr{L}^{\text{FOR}(S_1S_2S_3)} = \mathscr{L}^{\text{FOR}}$. Also note that loop constraints are partially ordered, i.e. $C_1 \leq C_2$ if and only if $\mathscr{L}^{\text{FOR}(C_1)} \subseteq \mathscr{L}^{\text{FOR}(C_2)}$. The obtained partially ordered set is shown in Figure 3b.

4 Implementation

We implemented the analyses described above in our tool SLOOPY[1]. It is built on top of Clang, a C language frontend for the LLVM compiler infrastructure. The analysis proceeds as follows:

[1] Available at http://forsyte.tuwien.ac.at/~pani/sloopy/.

1. We use Clang's representation of the control flow graph to identify so-called *natural loops*. These are subgraphs whose edges form a cycle with exactly one entry point. As an implementation detail, we choose this definition of loops to simplify the definition and implementation of data-flow analysis. In general, *irreducible flow graphs* may contain loops with multiple entry points. However, a recent study [SW12] "found 5 irreducible functions in a total of 10 427, giving a total average irreducibility for this set of current programs of 0.048%". The authors conclude that irreducibility "is an extremely rare occurrence, and it will probably be even rarer in the future".

2. Next, our tool attempts to find terminating predicates (Section 2.2.1). It computes the accumulated increment using data-flow analysis and the constant propagation framework [WZ85, ASU86] for all variables occurring in exit predicates. We rewrite each exit predicate into a normal form and perform syntactic pattern matching on it. This selects a strategy according to Table 1, which we check against the accumulated increment according to Table 2. At the moment, our analysis considers linear inequalities $P(i)$ in a single variable i, which allows us to handle condition expressions with common comparison operators of the C programming language (==, != , <, >, <=, >=) as top-level connective.

3. Finally, we enforce the invariance constraint (Section 2.2.2) – by checking for statements that update variables – and the control flow constraint (Section 2.2.3) – by assigning to each basic block the number of open (un-joined) branches using data-flow analysis. Any LTS node a of natural loop L with zero open branches lies on all paths through L.

4.1 Restrictions and Assumptions of Our Implementation

Our analysis makes a number of assumptions to determine eventual truth of predicates:

1. Due to the locality of our analysis, we assume the absence of aliasing and impure functions.

2. The C standard leaves pointer arithmetic undefined if operands and/or the result don't point into or just beyond the same array object [ISO, 6.5.6]. We assume the result of pointer arithmetic is always defined.

3. The C standard does not define overflow on signed integer types [ISO, 6.5.6]. The **search** strategy relies on covering the whole value range, thus we assume two's complement wrap-around behavior on overflow for **search** predicates over signed integer types.

   We may also prove termination of **increase/decrease** predicates over unsigned integers under this assumption, e.g. in the loop `for (unsigned i = 42; i < N; i--) ;` If we prove **increase/decrease** predicates over *signed* integers using this assumption, we likely discovered a bug [DLRA12]. We actually encountered one such bug, which had gone undetected for sixteen years in GPL Ghostscript, during experimental evaluation.

4. For any strategy other than **escape**, we assume that i in an exit predicate $P(i) : E(i) \circ N$, where $E(i)$ is an expression in i and $\circ \in \{<, \leq, \geq, >\}$, is not prevented by its (finite integer) type to enter the required interval to make $P(i)$ evaluate to true. As both $E(i)$ and N are expressions, we cannot determine their ranges by merely syntactic means.

Name	Description	$\lvert\mathscr{L}\rvert$
cBench [cBe]	Open-source sequential programs used in program and compiler optimization.	4157
coreutils [cor]	GNU Core Utilities, a collection of basic userland utilities.	1002
SPEC CPU2006 [Hen06]	Compute-intensive benchmarks composed from real life applications code. We only consider benchmarks written in C.	15 043
Mälardalen WCET [GBEL10]	Used in Worst-Case Execution Time (WCET) analysis.	262

Table 3: Benchmarks.

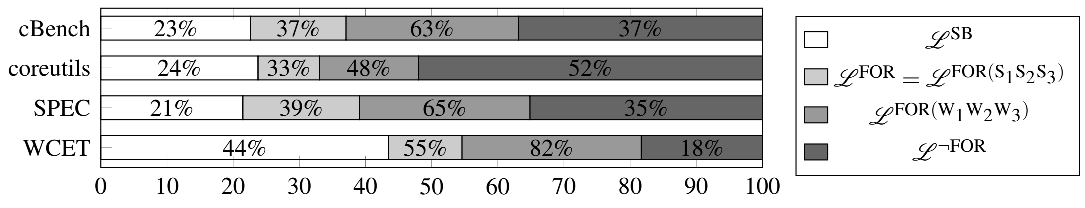


Figure 4: Percentage of loops covered by four selected loop classes.

5. For strict inequalities $P(i) : i < N$ ($i > N$), we assume N evaluates to less (more) than its type's minimum (maximum) value. Otherwise, P never evaluates to true for any i.

5 Experiments: Occurrence of Loop Patterns

In this section, we measure the occurrence of the various loop classes introduced above on four widely used benchmark suits from different domains. Table 3 summarizes these benchmarks.

Figure 4 and Table 4 show the percentage of loops in each of the listed classes for the respective benchmark: To keep the comparison working, in charts we only consider syntactically bounded loops $\mathscr{L}^{SB}$, the strongest generalized FOR loop class $\mathscr{L}^{FOR(S_1S_2S_3)} = \mathscr{L}^{FOR}$, the weakest generalized FOR loop class $\mathscr{L}^{FOR(W_1W_2W_3)}$, all loops of the benchmark $\mathscr{L}$, and all loops not in any simple loop class $\mathscr{L}^{\neg FOR} = \mathscr{L} \setminus \mathscr{L}^{FOR(W_1W_2W_3)}$. These loops are especially interesting, because those in $\mathscr{L}^{SB}$ can directly be compared with bound analysis tools, $\mathscr{L}^{FOR(S_1S_2S_3)}$ and $\mathscr{L}^{FOR(W_1W_2W_3)}$ represent the range of simple loops, and $\mathscr{L}^{\neg FOR}$ represents loops most different from our description of definite iteration.

	SB	$S_1S_2S_3$	$S_1S_2W_3$	$S_1W_2S_3$	$S_1W_2W_3$	$W_1S_2S_3$	$W_1S_2W_3$	$W_1W_2S_3$	$W_1W_2W_3$
cBench	22.6	37.0	46.6	37.2	47.0	48.0	62.7	48.3	63.1
coreutils	23.6	33.7	40.7	33.7	40.8	37.5	48.3	37.6	48.5
SPEC	21.5	39.2	45.9	39.3	46.0	56.0	64.5	56.2	64.7
WCET	43.5	54.6	66.8	55.0	67.6	67.2	80.5	67.6	81.7

Table 4: Percentage of loops covered by $\mathscr{L}^{SB}$ and the generalized FOR loop classes $\mathscr{L}^{FOR(\dots)}$.

Discussion. The percentage of loops in $\mathscr{L}^{\mathrm{SB}}$ is around 23%, except for WCET where it is at 44%. Some 9–18% more loops are only in $\mathscr{L}^{\mathrm{FOR}}$, which amounts to about 36% of all loops, except for WCET (55%). The weakest generalized FOR loop class $\mathscr{L}^{\mathrm{FOR}(\mathrm{W}_1\mathrm{W}_2\mathrm{W}_3)}$ additionally contains 26–27% more loops on all benchmarks except coreutils, comprising about 64% of all loops in cBench and SPEC, and 82% in WCET. For coreutils, the ratio of $\mathscr{L}^{\mathrm{FOR}(\mathrm{W}_1\mathrm{W}_2\mathrm{W}_3)}$ is at 48% of all loops.

The percentage of loops in $\mathscr{L}^{\mathrm{SB}}$ and $\mathscr{L}^{\mathrm{FOR}}$ is stable across all benchmarks except WCET. This can be explained by the fact that WCET stems from a narrow domain and only contains single-path programs, which naturally fulfill the strict control flow constraint of FOR loops. We have included it to showcase the difference to more general-purpose benchmarks.

The small difference between $\mathscr{L}^{\mathrm{FOR}}$ and $\mathscr{L}^{\mathrm{FOR}(\mathrm{W}_1\mathrm{W}_2\mathrm{W}_3)}$ in coreutils can be explained by a significant amount of loops containing system calls in the loop conditions. This is consistent with the low-level nature of the benchmark, but such system calls are not captured by our definition of generalized FOR loops.

5.1 Statement Usage

An important aspect of programming language design is which constructs and patterns are actually used by programmers to formulate algorithms. In our case, we analyze which statements are used to express loops from various classes. This is especially interesting as all C statements capable of implementing iteration have the same expressive power.

Discussion. Figure 5 shows the occurrence of statements in our selection of loop classes: Overall (bar (4) in Figure 5), the `for` statement is used a lot more than other statements, even though the number varies greatly between more than 80% of all loops for SPEC and WCET and less than 50% for coreutils. The next-most used statement is `while`, with an equally wide range from 10% on WCET to 47% on coreutils, where `for` and `while` statements occur about equally often. `do` and `goto` statements make up a minor share of less than 10% and 2%, respectively.

When we compare the ratio of statements used between different loop classes of the same benchmark, an interesting observation can be made: Regardless of the overall occurrence of `for` in $\mathscr{L}$, the stricter the loop class, the higher the percentage of loops expressed using a `for` statement. At the same time, the less restrictive a loop class is, the higher the percentage of `while`, `do`, `goto` statements. This correlation is especially strong for `do` and `goto` statements, where virtually no such loops are in $\mathscr{L}^{\mathrm{FOR}}$.

6 Experiments: Usefulness of Our Definitions

6.1 Comparison with Loopus

In this section, we show evidence for a relation between our loop classes and the difficulty of automated program analysis. To this end, we describe how well Loopus [ZGSV11] – a state-of-the-art termination and bound analysis tool – performs on our selection of loop classes: Figures 6a to 9a show the percentage of loops contained in each of these classes. Bars (1)–(4) are classes with increasingly less restrictive constraints, i.e. classes we conjecture to grow harder to

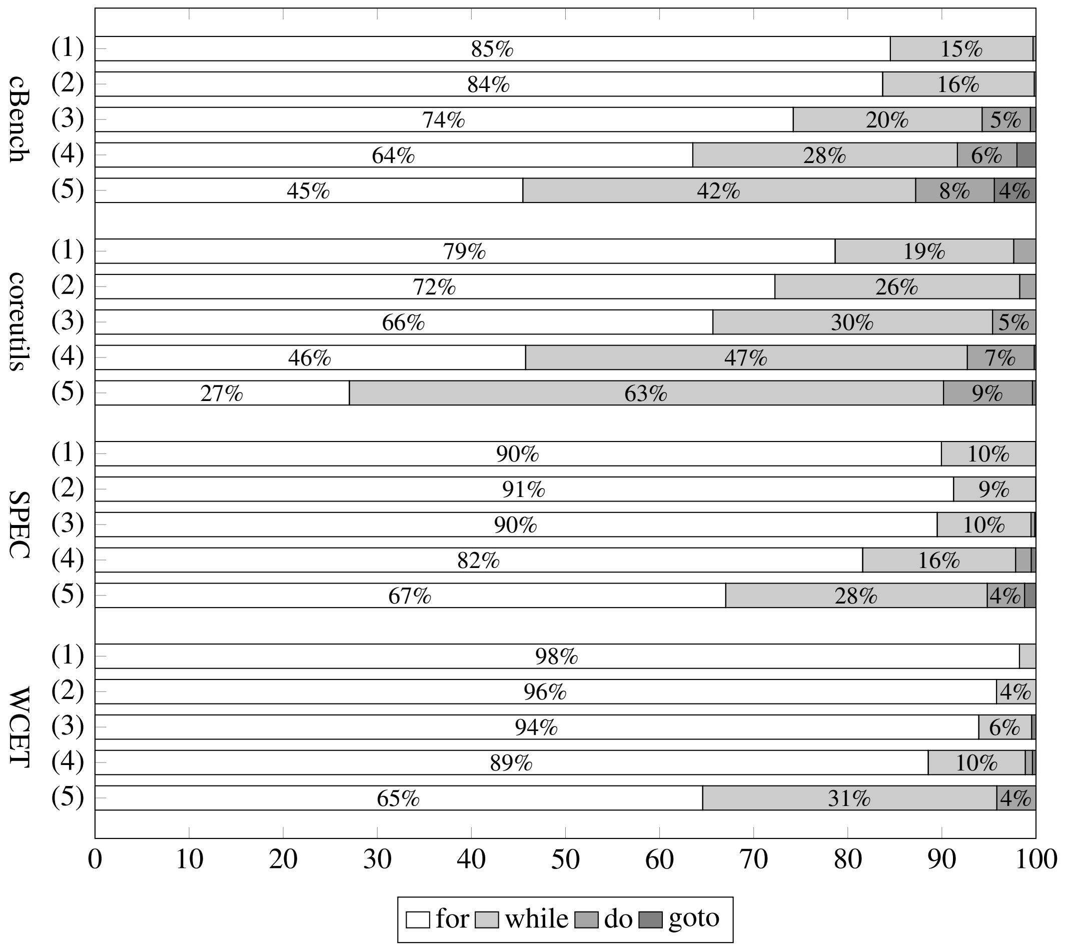
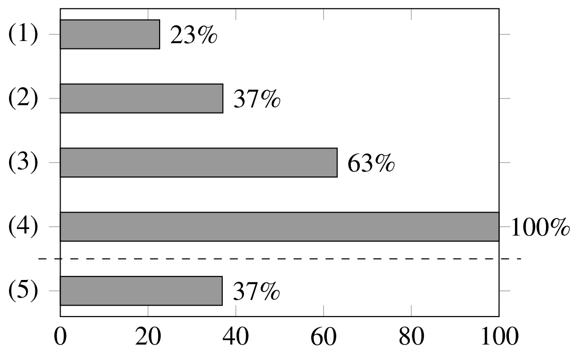
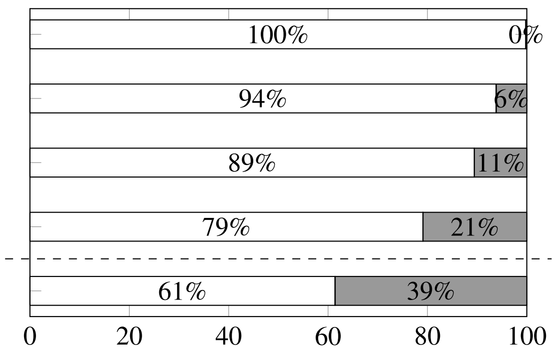


Figure 5: C statements used to express loops from loop classes (1) $\mathscr{L}^{\text{SB}}$, (2) $\mathscr{L}^{\text{FOR}} = \mathscr{L}^{\text{FOR}(\text{S}_1\text{S}_2\text{S}_3)}$, (3) $\mathscr{L}^{\text{FOR}(\text{W}_1\text{W}_2\text{W}_3)}$, (4) $\mathscr{L}$, and (5) $\mathscr{L}^{\neg\text{FOR}}$.

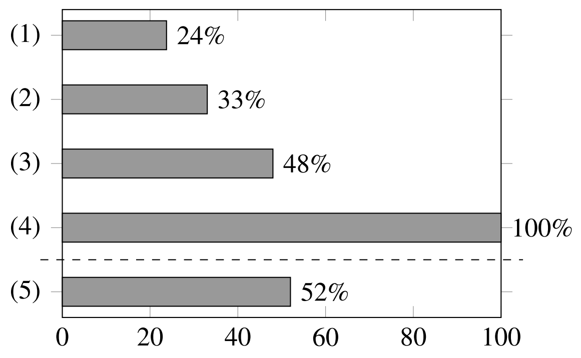


(a) Percentage of loop classes (1) $\mathscr{L}^{\text{SB}}$, (2) $\mathscr{L}^{\text{FOR}}$, (3) $\mathscr{L}^{\text{FOR}(\text{W}_1\text{W}_2\text{W}_3)}$, (4) $\mathscr{L}$, and (5) $\mathscr{L}^{\neg\text{FOR}}$.

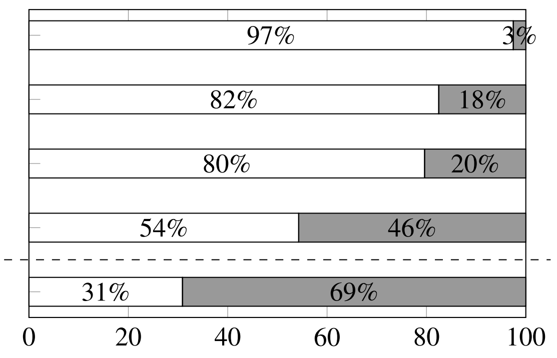


(b) Percentage of loops bounded (white) and not bounded (dark) by LOOPUS.

Figure 6: Comparison of various loop classes with LOOPUS results on cBench.

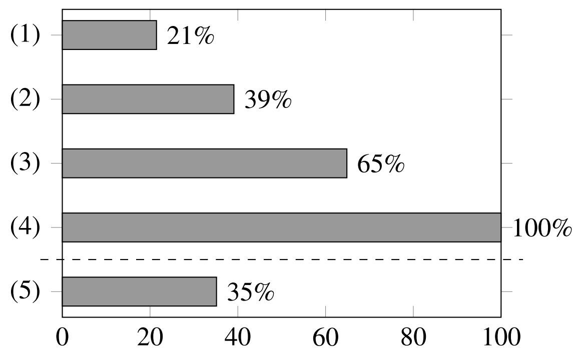


(a) Percentage of loop classes (1) $\mathscr{L}^{\text{SB}}$, (2) $\mathscr{L}^{\text{FOR}}$, (3) $\mathscr{L}^{\text{FOR}(\text{W}_1\text{W}_2\text{W}_3)}$, (4) $\mathscr{L}$, and (5) $\mathscr{L}^{\neg\text{FOR}}$.

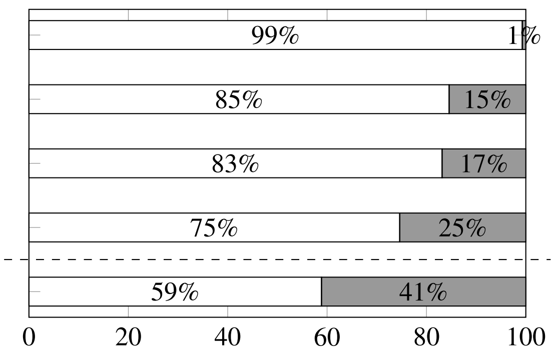


(b) Percentage of loops bounded (white) and not bounded (dark) by LOOPUS.

Figure 7: Comparison of various loop classes with LOOPUS results on coreutils.

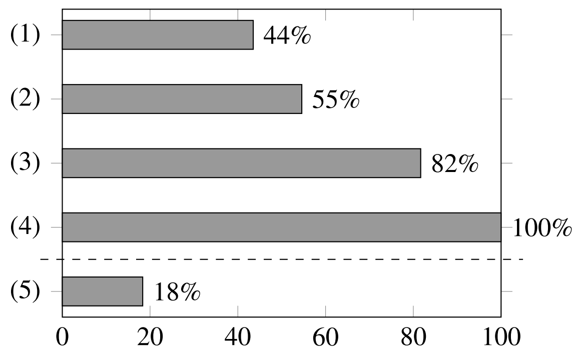


(a) Percentage of loop classes (1) $\mathscr{L}^{\text{SB}}$, (2) $\mathscr{L}^{\text{FOR}}$, (3) $\mathscr{L}^{\text{FOR}(\text{W}_1\text{W}_2\text{W}_3)}$, (4) $\mathscr{L}$, and (5) $\mathscr{L}^{\neg\text{FOR}}$.

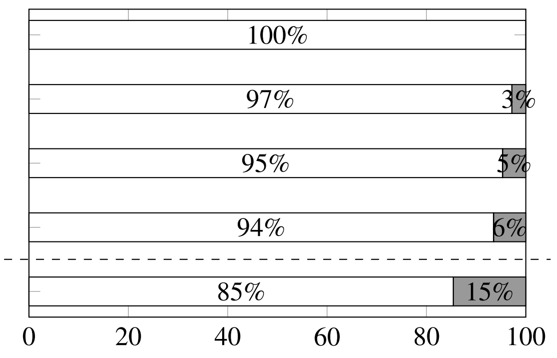


(b) Percentage of loops bounded (white) and not bounded (dark) by LOOPUS.

Figure 8: Comparison of various loop classes with LOOPUS results on SPEC.



(a) Percentage of loop classes (1) $\mathscr{L}^{\text{SB}}$, (2) $\mathscr{L}^{\text{FOR}}$, (3) $\mathscr{L}^{\text{FOR}(\text{W}_1\text{W}_2\text{W}_3)}$, (4) $\mathscr{L}$, and (5) $\mathscr{L}^{\neg\text{FOR}}$.



(b) Percentage of loops bounded (white) and not bounded (dark) by LOOPUS.

Figure 9: Comparison of various loop classes with LOOPUS results on WCET.

analyze in that order. Bars (5) below the dashed line refer to loops in $\mathscr{L}^{\neg\mathrm{FOR}}$, i.e. the complement of loops shown under (3). Figures 6b to 9b show the percentage of loops in the respective class for which LOOPUS succeeds and fails to compute a symbolic bound.

Discussion. We see that the more restrictive the investigated loop class, the better LOOPUS performs. This supports our conjecture that given two loop constraints $C_1 \leq C_2$, the stronger constraint C_1 not only describes a subset of loops, but also that this subset is less challenging for automated analysis. Additionally, as expected, for any termination proof we obtain in $\mathscr{L}^{\mathrm{SB}}$ on benchmarks that LOOPUS was optimized for (cBench and WCET), it also bounds the loop, i.e. LOOPUS agrees on each member of $\mathscr{L}^{\mathrm{SB}}$ identified by SLOOPY. Manual examination of loops in $\mathscr{L}^{\mathrm{SB}}$ unbounded by LOOPUS on coreutils and SPEC suggests that this is due to unmodeled system calls in LOOPUS. As a positive result for LOOPUS, while it performs worst on non-FOR loops $\mathscr{L}^{\neg\mathrm{FOR}}$, it is still successful on about two thirds of these loops, except for coreutils which seems a much harder benchmark.

6.2 A Portfolio Solver for Software Verification

In [DPVZ15] we present our tool VERIFOLIO, a machine learning based portfolio solver for software verification. Here, a portfolio solver is a software verification tool which uses heuristic preprocessing to select one of several existing tools [GS01].

To represent the source code of the unit under verification to the machine learning procedure, we introduce novel program metrics based on the loop classes presented in this work, and variable roles introduced in [DVZ13]. In particular, we compute class membership using SLOOPY and measure the relative occurrence m_C of four loop classes:

$$m_C = \frac{|\mathscr{L}^C|}{|Loops|} \qquad C \in \{\mathrm{SB}, \mathrm{FOR}, \mathrm{FOR}(\mathrm{W}_1\mathrm{W}_2\mathrm{W}_3), \neg\mathrm{FOR}\} \tag{1}$$

We tested our portfolio on benchmarks from the annual *International Competition on Software Verification* (SV-COMP) in its 2014 and 2015 editions [Bey14, Bey15]. In both cases, our tool VERIFOLIO is the overall winner and outperforms all other tools.

Additional experiments show that removing loop classes from the feature set yields worse results. We thus infer that they are indeed contributing to the overall performance of the portfolio. As the portfolio incurs an overhead for feature extraction compared to standalone tools, it is indispensable to base it on efficiently extractable program features. Our experiments show that the median time for computing loop class memberships per verification task in SV-COMP'14 and '15 is $\tilde{x} \approx 0.2$ seconds (obtained on a single core of a 2 GHz Intel Core i7 processor with 2 GB 1333 MHz DDR3 RAM).

Bibliography

[ASU86] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers: Princiles, Techniques, and Tools*. Addison-Wesley, 1986.

[Bey14] D. Beyer. Status Report on Software Verification - (Competition Summary SV-COMP 2014). In *TACAS*. Pp. 373–388. 2014.

[Bey15] D. Beyer. Software Verification and Verifiable Witnesses - (Report on SV-COMP 2015). In *TACAS*. Pp. 401–416. 2015.

[cBe] Collective Benchmark (cBench). http://cTuning.org/cbench. Online, accessed Feb 6, 2015.

[cor] Coreutils - GNU core utilities. http://www.gnu.org/software/coreutils/. Online, accessed Feb 6, 2015.

[Dij68] E. W. Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM* 11(3):147–148, 1968.

[DLRA12] W. Dietz, P. Li, J. Regehr, V. S. Adve. Understanding integer overflow in C/C++. In *ICSE*. Pp. 760–770. 2012.

[DPVZ15] Y. Demyanova, T. Pani, H. Veith, F. Zuleger. Empirical Software Metrics for Benchmarking of Verification Tools. In *CAV*. Pp. 561–579. 2015.

[DVZ13] Y. Demyanova, H. Veith, F. Zuleger. On the concept of variable roles and its use in software analysis. In *FMCAD 2013*. Pp. 226–230. 2013.

[GBEL10] J. Gustafsson, A. Betts, A. Ermedahl, B. Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In *WCET*. Pp. 136–146. 2010.

[GS01] C. P. Gomes, B. Selman. Algorithm portfolios. *Artif. Intell.* 126(1-2):43–62, 2001.

[Hen06] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News* 34(4):1–17, 2006.

[ISO] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C.*

[KW11] D. Kroening, G. Weissenbacher. Interpolation-Based Software Verification with Wolverine. In *CAV*. Pp. 573–578. 2011.

[RAPG14] R. E. Rodrigues, P. Alves, F. Pereira, L. Gonnord. Real-World Loops are Easy to Predict: A Case Study. In *Workshop on Software Termination (WST'14)*. 2014.

[Sta93] A. M. Stavely. An empirical study of iteration in applications software. *Journal of Systems and Software* 22(3):167–177, 1993.

[Sta95] A. M. Stavely. Verifying Definite Iteration Over Data Structures. *IEEE Trans. Software Eng.* 21(6):506–514, 1995.

[SW12] J. Stanier, D. Watson. A study of irreducibility in C programs. *Softw., Pract. Exper.* 42(1):117–130, 2012.

[WZ85] M. N. Wegman, F. K. Zadeck. Constant Propagation with Conditional Branches. In *POPL*. Pp. 291–299. 1985.

[ZGSV11] F. Zuleger, S. Gulwani, M. Sinn, H. Veith. Bound Analysis of Imperative Programs with the Size-Change Abstraction. In *SAS*. Pp. 280–297. 2011.