FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Loop Patterns in C Programs

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Thomas Pani
Matrikelnummer 0726415

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Helmut Veith
Mitwirkung: Ass.Prof. Dipl.-Math. Dr.techn. Florian Zuleger

Wien, 01.12.2013

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Loop Patterns in C Programs

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Thomas Pani
Registration Number 0726415

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Helmut Veith
Assistance: Ass.Prof. Dipl.-Math. Dr.techn. Florian Zuleger

Vienna, 01.12.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Thomas Pani
Lerchengasse 33, 3430 Tulln

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

In a time of drastic change it is the learners who inherit the future. The learned usually find themselves equipped to live in a world that no longer exists.

— Eric Hoffer, *Reflections on the Human Condition*

Acknowledgements

First of all, I would like to thank my advisor Helmut Veith for his guidance and support during my master's studies. He suggested the topic of this thesis, pointed out relevant research, and provided direction and insight. Most importantly, he recognized and believed in my strengths and gave me opportunity and room to develop – thank you!

Next, I would like to thank my co-advisor Florian Zuleger for his advice and interest in my work. His sense for focusing on the important problems, as well as communicating them clearly in a big-picture view have greatly helped me understand my work in context and relevance to related work. He also put a tremendous amount of time into giving detailed feedback on both content and structure, without which the work at hand would not exist.

This thesis has connections to Yulia Demyanova's work on variable roles, and Moritz Sinn's work on bound computation. Thank you both for discussing your work with me, and for sharing your experience with implementing Clang- and LLVM-based tools. In general, I'm grateful to all my colleagues from the FORSYTE group and on the "third floor" for this pleasant work environment.

Pursuing an academic degree *and* various other interests, one needs a stable frame to support the picture. I would like to thank my parents and grandparents for their continued support of my plans. My sisters Carina and Christina have held the frame as well as painted the picture with me. Thank you for your love, understanding, and the possibility to discuss just about anything.

Along the way, many more people have considered it worth their time to teach me. Even though their number is finite, I cannot possibly hope to compile an exhaustive list. I am greatly indebted to all of you, for sharing your wisdom and for shaping my life.

Vienna, December 2013
Thomas Pani

Abstract

Loop statements are an indispensable construct to express repetition in imperative programming languages. At the same time, their ability to express indefinite iteration entails undecidable problems, as famously demonstrated by Turing's work on the halting problem. Due to these theoretic limitations, efficacy of a termination procedure can only be demonstrated on practical problems – by Rice's theorem the same goes for any automated program analysis. Even though we have techniques to tackle some but by far not all decidable cases, so far no established notion of difficulty with regard to handling loops exists in the scientific community. Consequently, the occurrence of such "simple" and "difficult" loops in practice has not yet been investigated.

We propose to conduct a systematic study that describes classes of loops and empirically determines difficulty of automated program analysis for each class. To arrive at this classification, we draw on a recent direction in software verification: Instead of treating programs merely as mathematical objects, we also consider them as human artifacts that carry important information directed at human readers of the program. This engineering information should be leveraged for automated program analysis, but is usually provided informally (e.g. as comments) or implicitly (e.g. as patterns followed by programmers). Thus, a formally sound framework to extract such information about loops is needed.

A direct application of this study is the empiric evaluation of automated program analysis: As discussed, such an evaluation is of central importance to establish academic merit as well as applicability to the software engineering domain. However, as we show, experimental results are currently insufficiently comparable, due to the wide range of benchmark programs used. We propose the introduction of *benchmark metrics*, such as the number of "simple" loops mentioned above, as a way to characterize benchmarks and achieve better comparability.

Motivated by the state of the art, this thesis conducts a systematic study of loops in C programs to classify loops based on the difficulty of automated program analysis. To arrive at our classification, we leverage the program's structure as imposed by the programmer: We introduce the class *syntactically terminating loops*, a common engineering pattern for which termination proofs can be obtained using minimal information, and describe how to determine class membership. Subsequently, we derive the family of *simple loops* by systematically weakening syntactically terminating loops along different dimensions, which allows us to describe a majority of loops in our benchmarks while still capturing syntactic termination criteria well. Finally, we present our tool SLOOPY and compare the identified loop classes with results experimentally obtained from LOOPUS, a tool to compute symbolic loop bounds. Thus we show that simple loops indeed capture the difficulty of automated program analysis.

Kurzfassung

Schleifen sind ein unverzichtbares Konstrukt imperativer Programmiersprachen. Wie Turings frühe Arbeit zum Halteproblem zeigt, bringt die Ausdrucksstärke solcher Anweisungen allerdings unentscheidbare Probleme mit sich. Aus diesem Grund kann die Wirksamkeit eines automatischen Terminierungsbeweises nur anhand praktischer Beispiele demonstriert werden – nach dem Satz von Rice gilt dies gleichermaßen für jede Form der automatischen Programmanalyse. Obwohl nach dem Stand der Technik einige, aber bei weitem nicht alle entscheidbaren Fälle bewältigt werden können, existiert noch keine etablierte Vorstellung von der Schwierigkeit Schleifen zu behandeln in der Scientific Community. Infolgedessen wurde auch das Auftreten "einfacher" und "schwieriger" Schleifen in der Praxis noch nicht beschrieben.

Wir empfehlen, dies in einer Studie zu untersuchen. Um Schleifen zu klassieren, bedienen wir uns einer neuen Forschungsrichtung in der Softwareverifikation: Anstatt Programme ausschließlich als mathematische Objekte zu betrachten, behandeln wir sie als von Menschen geschaffene Artefakte, die Informationen an die Leser des Programms richten. Diese sollten zum Zweck der automatischen Programmanalyse eingesetzt werden, sind aber meist nur informell (z.B. als Kommentar) oder implizit (z.B. als von Programmierern benutztes Muster) beschrieben. Daher benötigen wir eine Methode, derartige Informationen über Schleifen zu extrahieren.

Eine direkte Anwendung ist die empirische Evaluierung automatischer Programmanalyse – wie beschrieben von zentraler Bedeutung zur Feststellung akademischer Leistungen als auch der Anwendbarkeit im Software Engineering. Die derzeit übliche Praxis erlaubt jedoch aufgrund der Vielzahl an verwendeten Benchmarks nur eingeschränkte Vergleichbarkeit. *Benchmarkmetriken*, wie zum Beispiel die Anzahl der oben erwähnten "einfachen" Schleifen, können zur Beschreibung von Benchmarks herangezogen werden und derart die Vergleichbarkeit verbessern.

Motiviert vom Stand der Technik führt diese Diplomarbeit eine Studie von Schleifen in C Programmen durch, welche wir basierend auf der Schwierigkeit automatischer Programmanalyse klassifizieren: Wir führen die Klasse der *syntaktisch terminierenden Schleifen* ein, die ein von Programmierern häufig benutztes Muster beschreiben und einen Terminierungsbeweis anhand minimaler Information erlauben. Wir leiten die Familie der *simplen Schleifen* ab, indem wir die Kriterien syntaktisch terminierender Schleifen systematisch abschwächen. Dieser Ansatz deckt die Mehrzahl der Schleifen in unseren Benchmarks ab, während er gleichzeitig die Terminierung nach wie vor hinreichend gut beschreibt. Abschließend stellen wir unsere Implementierung SLOOPY vor und vergleichen die identifizierten Klassen mit experimentellen Resultaten von LOOPUS, einem Tool zur Berechnung symbolischer Bounds von Schleifen. Derart zeigen wir, dass simple Schleifen tatsächlich die Schwierigkeit automatischer Programmanalyse erfassen.

Contents

CHAPTER 1

Introduction

Turing complete languages provide constructs capable of expressing *conditional execution* and *indefinite iteration*. In imperative languages, these are usually implemented as loop statements – a typical example is the `while` statement of the C programming language. As shown by Turing's work on the *halting problem*, this ability to express indefinite iteration also entails undecidable problems for automated program analysis [52].

1.1 State of the Art in Program Analysis

Proving program termination has resurged as a research topic in the last decade inspired by the success of automated program verification [15]. Moreover, proving program termination is famously known to be undecidable. The only way to show the usefulness of a termination procedure is to demonstrate its efficacy on practical problems. The same goes for all program analysis problems due to Rice's theorem.

Despite the fact that we have techniques to tackle some, but by far not all decidable cases, so far no notion of difficulty with regard to handling loops has been established in the scientific community. Consequently, also the occurrence of "simple" versus "difficult" loops in practice, as well as properties of such loops have not been studied. As Dijkstra put it in his Turing Award Lecture "The Humble Programmer", "it is our business to design classes of computations that will display a desired behavior [21]", and we intend to introduce such classes for loops that can algorithmically be shown to terminate.

1.2 Programs as Human Artifacts

Historically, a few key insights have vastly increased the scalability of automated program analysis to tackle additional decidable cases from whole new application areas. We illustrate this on the example of model checking, a verification technique: During the thirty years since its inception [13, 46], first the introduction of symbolic model checking [10] made it possible to verify hardware and embedded systems. Subsequently, bounded model checking [12] and counterexample-guided abstraction refinement (CEGAR) [14] allowed its application to small pieces of software such as device drivers [2]. However, so far the verification of more complex systems, such as applications code has barely been tapped. Especially CEGAR can in principal scale to larger problems: The concrete, exponentially large system to analyze is projected onto a smaller abstract system, for which exploration by the model checker is feasible. The challenge lies in the automated selection of good abstraction functions, which we illustrate for integer variables:

Consider Listing 1.1: Variable `b` is a *boolean flag*, i.e. it takes only values 0 and 1. Analysis will benefit from an abstraction on the predicate $b = 0$. Variable `i` on the other hand is used as *counter variable*, and choosing the same abstraction $i = 0$ is most likely too coarse. Note that the terms we naturally used to refer to these variables are not directly reflected in the programming language: both are declared as `int` variables – their role is an intuitive notion attributed by the human reader of the program. One can however use patterns of data flow and control flow to algorithmically extract such intuitive notions from programs [19].

Listing 1.1 An example illustrating variable roles.

```
for (int i = 0; i < N; i++) {

    int b;
    if (f(i) == X) {
        b = 0;
    } else {
        b = 1;
    }

    // ...
}
```

In short, this approach follows a new direction in automated program analysis: Instead of purely treating programs as mathematical objects, they are also considered as human artifacts containing information directed at human readers of the program. Such engineering information is usually provided informally (e.g. as comments) or implicitly (e.g. as patterns followed by programmers) and can provide additional insight that is much needed for the advancement of automated program analysis. Again, Dijkstra referred to the power of such knowledge introduced by engineers in his Turing Award Lecture: “The effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer [21]”. Due to the intricate properties of loop statements outlined above, a systematic study of loops on this level can provide interesting results.

1.3 A Systematic Study of Loop Patterns

Motivated by the hard problems entailed by indefinite iteration and the current challenges in automated program analysis, we propose to conduct a systematic study of loops. We sketch the key requirements of such a study:

1. The study should consider a reasonable amount of code from different application areas to arrive at meaningful results.
2. The obtained results should be reproducible on arbitrary benchmark cases.

Due to these constraints, a manual approach seems infeasible and we need to provide an algorithmic solution. Furthermore, we need to divide loops into meaningful classes for which the difficulty of program analysis is studied. How to arrive at this classification is non-trivial: As most programming languages provide means to express arbitrary iteration by the same statement, we cannot rely on a purely syntactic pattern matching. On the other hand, it is not sensible to implement another tool trying to prove program termination. That way we would end up with two classes "handled by automated program analysis implemented in tool X" and "not handled by automated program analysis implemented in tool X" – i.e., the status quo instead of a classification applicable across existing tools that assigns classes of loops a relative difficulty.

Motivated by the work on variable roles introduced earlier, we decide to incorporate the notion of software as a human artifact in our classification: If engineers follow specific patterns to express certain classes of loops, it should be possible to extract a classification according to these patterns from the source code.

In the following, Section 1.4 first illustrates an application of the systematic study of loop patterns proposed here. Afterwards, Section 1.5 details the concrete loop patterns we study in this thesis.

1.4 An Application: Empiric Evaluation

Given the wide range of available techniques in automated program analysis, as well as the theoretic limitations of undecidability, empiric evaluation is of central importance to establish both academic merit as well as applicability to the software engineering domain. However, the selection of benchmarks to perform the evaluation on is mostly one of the authors' preference. We propose the introduction of *benchmark metrics* such as the loop classes introduced in this work to address this problem:

To illustrate the current practices of benchmarking in the automated program analysis community, we surveyed a sample of participants in the first *Competition on Software Verification* (SV-COMP). In our survey, we collected on which benchmark suites the tools had originally[1] been evaluated by their authors. The surveyed participants, discovered benchmarks, and results are shown in Table 1.1: If we found a tool to be benchmarked on a particular benchmark, the respective cell contains a check mark. If only a subset or modified version was used, the check mark is in parentheses.

[1]any publication about the tool before SV-COMP'12

Benchmark / Tool	CPAchecker-ABE [8]	FShell [34, 35]	HSF(C) [28]	Predator [23, 22]	SatAbs [56]	Wolverine [40]
`test_blocks` [8]	✓					
`ssh-simplified` [7]	✓		✓			
`ntdrivers` [7]	(✓)[2]	(✓)[3]	✓			
Numerical Recipes [45]			✓			
`matlab.c` [34, 35]		✓				
`memman.c` [35]		✓				
PicoSAT [9]		✓				
Linux VFS [26]		(✓)[4]				
Busybox coreutils [11]		(✓)[5]				
`lvm2` [57]				✓		
NSPR memory allocator [57]				✓		
`cdrom.c` [57, 22]				✓		
SLAyer driver snippets [3]				✓		
Predator case studies [23]				✓		
`nbd` [40]						✓
`machzwd` [56]					✓	✓

Table 1.1: A sample of SV-COMP participants and benchmarks originally used.

As Table 1.1 illustrates, only three of the sixteen benchmarks were used for multiple tools: `ntdrivers`, `machzwd` and `ssh-simplified`. Moreover, `ntdrivers` only recurred as a simplified or stripped-down version. `machzwd` occurred twice, but in publications sharing one author. The only independently and fully reused benchmark is `ssh-simplified`, which only two of the sampled six participants had used for evaluation. We believe this state of the art is less than ideal as comparability of experimental results is essentially lost due to non-comparable benchmark programs that the experiments were based on. Thus, we discuss two possible strategies to improve upon the state of the art:

[2]a simplified version of the sources, `ntdrivers-simplified`

[3]only `kbfiltr.c`

[4]a single example use case

[5]selected tools

Commonly Agreed-upon Benchmarks

One way of resolving the distortion of empiric results is to establish an commonly accepted benchmark suite. SV-COMP is prominently promoting this direction and sets one of its goals to "establish a set of benchmarks for software verification in the community" [4, 5]. Having a set of established and widely accepted benchmarks comes with a number of benefits: First, experimental results for tools benchmarked on this suite are easily comparable, especially if the results are established in a recurring competition under controlled settings. Second, establishing the suite itself, as well as related discussion, e.g. on the properties to check, happens within the scientific process. However, even if an extensive, established and accepted benchmark suite exists, some concerns remain. These challenges also illustrate why re-benchmarking an existing tool on a different benchmark is a questionable practice:

Domain-specific tools. Many tools are created for the purpose of demonstrating specialized methods on a well-defined problem domain. Consider systems level code, which is largely unthinkable without pointer types, whereas applications code dedicated to problem solving domains will prefer integer-based arithmetic, and scientific code may extensively rely on floating-point arithmetic. In terms of resource efficiency, authors of such tools may restrict their evaluation to show merit on that specific problem domain. Benchmarks used may largely or completely be comprised of manufactured examples that explicitly illustrate this one problem domain. Applying such tools to less specialized benchmarks would distort the value of their contribution. On the other hand, establishing commonly agreed-upon benchmarks for every single problem domain seems infeasible.

Academic prototypes. Producing industrial-grade, reliably maintained tools seldomly is the output of academic research. Rather, research prototypes provide a well-enough solution for demonstrating theoretic underpinnings. As such, research prototypes are often optimized for a specific set of benchmarks to illustrate what may be accomplished – given the proper resources – in the general case. (Re-)benchmarking these tools on a wider set of benchmarks than the authors can afford to optimize for may again lead to distortion of the actual merit.

Benchmark Metrics

While establishing a commonly agreed-upon set of benchmarks is certainly of value for general-purpose tools, this approach comes with the above-mentioned restrictions. Orthogonally, we propose to augment the process of empiric evaluation by a new family of metrics. These should evidently capture common properties of programs (benchmarks) and indicate the difficulty of automatically reasoning about certain properties of this program. Traditionally, software metrics have been used to indicate modules in which an increased number of faults is expected. As our proposal applies metrics to measure properties of benchmark programs, we call these metrics *benchmark metrics*. The classes of loops we introduce in this thesis may serve as such a benchmark metric. For example, one can report the number of *simple loops* (introduced in Chapter 4) in a benchmark program.

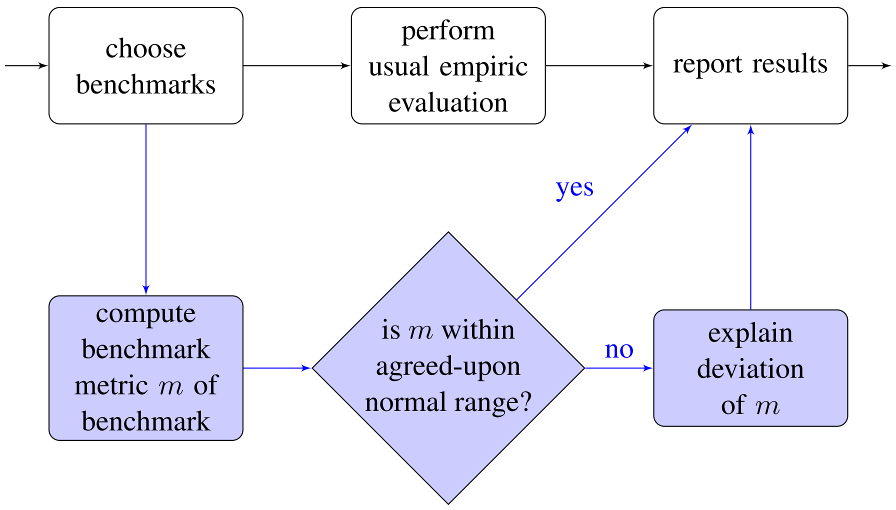

choose benchmarks
perform usual empiric evaluation
report results
yes
compute benchmark metric m of benchmark
is m within agreed-upon normal range?
no
explain deviation of m

Figure 1.1: Benchmark metrics augment (blue) the established empiric evaluation process.

Figure 1.1 illustrates the additional steps during empiric evaluation: As soon as a set of benchmarks is fixed, in parallel to performing the usual empiric evaluation, one starts establishing benchmark metrics for the chosen benchmarks. Having computed a benchmark metric m, this metric is compared to an agreed-upon normal value or range R. If m is consistent with that normal range ($m \in R$), one may simply report this fact. Otherwise, the publication should account for the deviation and provide an explanation, as we will demonstrate in Chapter 5. In each case, we conjecture this justification may be succinctly provided in a few sentences or a short table, thus not severely impacting publication length.

It is important to note that a deviation of m from the normal range R by itself is not negative in any way. It merely expresses that experimental results may not be comparable to experimental results of other tools in the specific area captured by the benchmark metric. For heavily application- or domain-dependent measures, one may even introduce a family of normal ranges $\mathcal{R}$, with one normal range $R \in \mathcal{R}$ per application area or problem domain. Mutual agreement on a set of benchmark metrics, as well as respective normal ranges, is essential to our approach. We believe that these can be reliably established as part of the current scientific process.

In terms of the concerns we raised about commonly-agreed benchmarks, the need for well-maintained mature tools is pushed from the actual prototype implementations to a set of benchmark metric tools. Given the broad application range[6] of these tools, and from our experience with implementing such a tool, we expect maintenance of high-quality benchmark metric tools to be much more practicable. If benchmark metric analysis is performed on an intermediate representation for which different language frontends exist, a single tool may be applied to benchmarks in a variety of programming languages. Likewise, the process of agreeing on a standardized set of benchmarks (which might be subject to political discussions) is replaced by agreeing on a set of benchmark metrics. If two benchmarks largely agree on this set of benchmark metrics, they are considered equivalent. If a pair of benchmarks is considered equivalent

[6]C.f. applications of this thesis discussed below and in Chapter 7 as an example for loop-related applications.

but from experience should not be, establishing this fact and providing an updated or newly introduced benchmark metric is an interesting research question by itself.

1.5 Methodological Approach

In the previous sections we gave an introduction to the state of the art and discussed the problem at hand. Furthermore, we proposed to conduct a systematic study of loop patterns and exemplarily illustrated an application of such a study. We now give a more detailed overview of the study conducted in this thesis and an introduction to our concrete approach.

Automated program analysis is usually performed on a low-level, so-called *intermediate representation* of the program obtained from its source code using a compiler. However, this intermediate representation does no longer capture the structure of the program the programmer intended, as high-level constructs of the original programming language are broken down to a small set of low-level operations and optimizations are performed by the compiler. Therefore, we base our analysis on an unaltered representation of the program's source code, the *control flow graph* (CFG), a standard data structure in compiler construction. In contrast to the second data structure capturing the the program's source code, the *abstract syntax tree* (AST), the CFG makes control flow explicit, which is essential to our further approach. To obtain the program's CFG, we may rely on existing frameworks – we choose to base our analysis on the C programming language, the lingua franca of automated program analysis tools, and implement it in CLANG, a modern C language frontend for the LLVM compiler framework.

In addition, we employ *data-flow analysis*, a standard technique used in compilers e.g. for *reaching definitions* or *live variable* analyses [1]. Due to the resource efficiency usually pursued with compilers, this technique is coarse enough to be extremely efficient. At the same time, the obtained abstraction is still detailed enough for basic analyses, as demonstrated by the aforementioned examples. Building on the program's CFG and data-flow analysis, we describe a technique to identify *loop counters*, i.e. variables whose values change in well-behaved increments during each iteration of a loop. Additionally, we describe syntactic patterns for a class of condition expressions that guard control flow exiting (and thus terminating) the loop. Expressions in this class have the property that – using the increment values obtained for loop counters before – we can determine if they will at some point during program execution direct control flow out of the loop, thus proving termination. We call the class of loops for which such a termination proof using minimal information is feasible *syntactically terminating loops*.

To evaluate our classification, we compare experimental results of the classes assigned by our implementation SLOOPY with results obtained from LOOPUS [58], a tool for automated bound computation. We found (c.f. Chapter 5) LOOPUS to fully agree on the bounded cases we describe, which make up about one fourth of loops in our benchmarks. However, LOOPUS succeeds to compute a bound on more than half – on one benchmark up to 94% – of loops found in the benchmarks. Consequently, our analysis should also identify these cases.

We systematically weaken syntactic termination criteria along three dimensions to obtain the family of *simple loops*. Intuitively, these weakenings introduce some leeway where the locality of our analysis restricts the ability to obtain a syntactic termination proof. At the same time, simple loops still capture the termination-related properties of syntactically terminating loops well. Depending on the strength of the applied weakenings, we obtain different simple loop classes within the simple loop family. The dimensions along which we perform the weakenings are structured hierarchically, thus inducing an order relation that allows us to identify weaker and stronger classes.

Finally, we present experimental results, again comparing loops classified as simple loops by our tool SLOOPY with results obtained from LOOPUS. These results show that the various simple loop classes indeed capture the difficulty of program analysis and comprise a majority of loops in our benchmarks.

1.6 Contributions

- We show a termination proof by minimal information and define the class of syntactically terminating loops.
- We systematically weaken syntactic termination criteria to obtain simple loops, a loop pattern commonly used by programmers.
- We show that simple loops indeed represent the majority of loops in our benchmarks.
- We show the various simple loop classes indeed capture the difficulty of automated program analysis.
- We sketch research directions and further applications of loop patterns.

CHAPTER 2

Structural Classification

As discussed in the previous chapter, we start our investigation by describing a partitioning of loops inspired by software metrics. We state the program's *control flow graph* (CFG) as our program model and give a graph-theoretic definition of *natural loops*, the kind of loops our analysis will handle. Further, we introduce *program slicing*, a standard technique that allows us to extract parts of the CFG that influence termination. Finally, we describe a purely structural metric that counts the number of exits from a loop and introduce three metrics on sliced CFGs.

2.1 Control Flow Graphs

A CFG is a digraph $C = (B, E)$ constructed from the program's source code. CFGs are standard data structures in compiler construction and static analysis, and their construction is discussed in detail in textbooks [1]: The nodes B are so-called *basic blocks* that describe maximal sequences of consecutive statements such that control only enters the basic block through its first statement, and control leaves the block only at its last statement. The flow graph's directed edges E describe the flow of control, i.e. $(a, b) \in E$ if and only if flow of control may enter b from a. Each edge (a, b) is labeled with the predicate $assume(a, b)$ guarding its execution, or true if a has a singleton successor. In addition to the basic blocks constructed directly from the program's source code, B contains two nodes ENTRY and EXIT: The singleton successor to ENTRY is the first executable node of the flow graph. The predecessors to EXIT are any nodes whose last statement terminates the program. We define two functions on basic blocks $b \in B$: $succs(b)$ returns the set of all immediate successor blocks of b and $preds(b)$ the set of all immediate predecessors of b. Figure 2.1 shows an example program and its CFG. As in the C programming language edge labels directly correspond to a condition expression in the preceding block, we will sometimes omit edge labels and draw solid edges for labels expressing unnegated conditions, and dashed edges for labels expressing negated conditions.

SLOOPY, the tool in which we implement our analysis and use in Chapter 5 to obtain experimental data, is built on top of CLANG, a C language frontend for the LLVM compiler infrastructure. The CLANG API allows us to parse a C program and obtain the CFG for each function

```
1  int test(unsigned a, unsigned N) {
2      int c = 0;
3      while (c < N) {
4          a = a / 2;
5          c++;
6      }
7      if (a == 0) {
8          return -1;
9      } else {
10         return a;
11     }
12 }
```

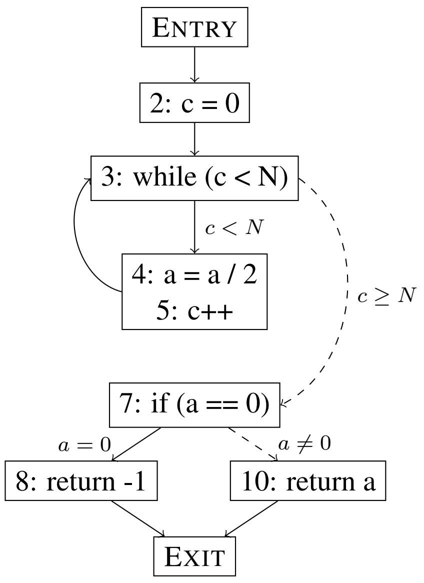


Figure 2.1: Example program and its CFG.

defined in the program. As prescribed by C semantics, ENTRY's successor is the singleton entry point of the function and any block after which control leaves the function is a predecessor to EXIT. Due to this handling of CFGs in CLANG, our analysis is intra-procedural.

2.2 Natural Loops

Having defined the CFG as the underlying data structure for analysis, we now describe loops in terms of graph-theoretic notions. While a straight-forward definition might consider any cycle in the CFG a loop, this is too lax for the purposes of static analysis. Intuitively, a loop should have a single entry point, and its edges should form at least one cycle. The graph-theoretic definition of these properties is the *natural loop*. We first define the *dominator relation* and *back edges*, which we then use to define the natural loop. Our definitions follow those presented in [1].

Definition 1 (Dominator relation)**.** A node d of a flow graph *dominates* a node n (d dom n) if and only if every path from ENTRY to n goes through d.

Definition 2 (Back edge)**.** A *back edge* is an edge $a \rightarrow b$ whose head b dominates its tail a.

Definition 3 (Natural loop)**.** Given a back edge $n \rightarrow d$, its *pre loop* consists of d plus the set of nodes that can reach n without going through d. We call d the *header* of the loop. The *natural loop* L with header d is the union of all pre loops with header d (see Figure 2.2 for a simple example). A path p through L is any path $d, b_1, b_2, \ldots, b_n, d$ where $b_i \in L$ and $b_i \neq d$ for $1 \leq i \leq n$. For the set of all such paths, we write $p \in paths(L)$.

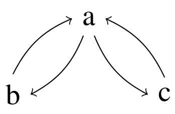
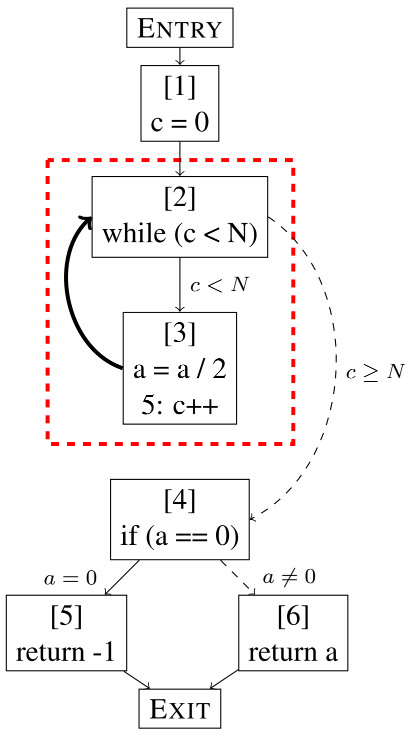
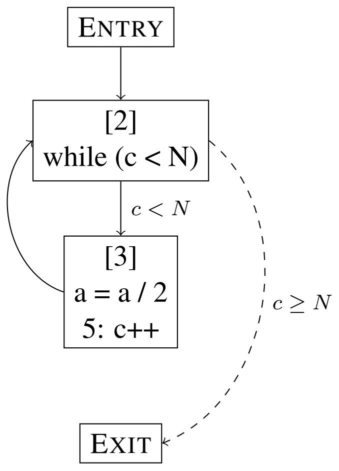


Figure 2.2: A natural loop with header a consisting of two pre loops with the back edges $b \to a$ and $c \to a$.



(a) CFG of function `test` in Figure 2.1. The backedge $[3] \to [2]$ and its natural loop are highlighted.



(b) CFG of the natural loop of backedge $[3] \to [2]$.

Figure 2.3: Control flow graph of a natural loop.

Definition 4 (Nested loop)**.** Two natural loops are either disjoint or one is entirely contained within the other. In the latter case, we call the contained loop a *nested* or *inner loop*.

As our analysis is CFG-based, it remains to define the control flow graph of a natural loop:

Definition 5 (Control flow graph of a natural loop)**.** A natural loop L's control flow graph C_L is exactly the control flow graph of the subprogram defining the natural loop. To construct C_L, we start from the subgraph of the function's CFG C_F whose vertex set equals L. To this subgraph we add fresh ENTRY and EXIT nodes, introduce an edge (ENTRY, d) where d is the header of L, and introduce an edge (a, EXIT) for each edge $(a, b), a \in L, b \notin L$ in C_F with the same edge label $assume(a, b)$.

Example. Figure 2.3a shows the CFG of function `test` from Figure 2.1 and highlights the singleton backedge as well as its natural loop. Figure 2.3b shows the natural loop's CFG constructed according to Definition 5.

Irreducible Flow Graphs

In *reducible flow graphs*, all cycles are natural loops, i.e. each cycle has a single entry point (the natural loop's header) that dominates all basic blocks in the cycle. *Irreducible flow graphs* contain one or more loops with multiple entry points. By restricting our analysis to natural loops, we ignore loops with such irreducible flow graphs. We argue that this restriction is tolerable for two reasons:

1. A recent study by Stanier and Watson [49] "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. Many static analysis tools are restricted to reducible CFGs. In particular, our main tool for comparison, LOOPUS, only supports reducible CFGs at the moment.

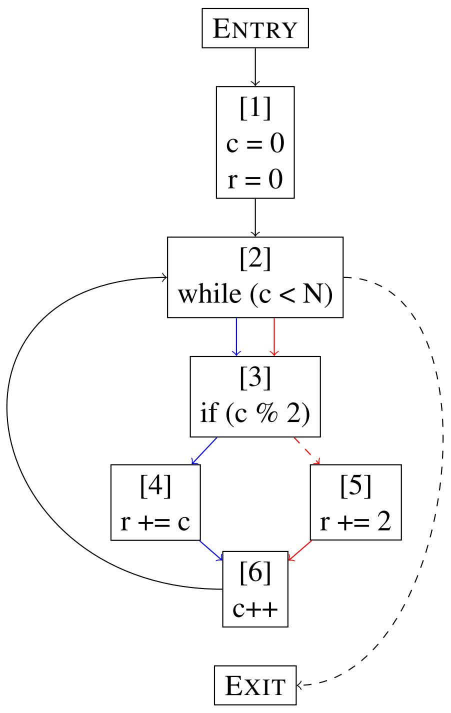
2.3 Program Slicing

Classic software metrics can be divided into *size counts* (e.g. lines of code, LOC) and *software complexity measures* (e.g. cyclometic complexity, measuring the number of linearly independent paths) [24]. However, we cannot directly compute such measures and hope to find correlation with the difficulty to prove termination: A long program may perform many auxiliary computations not influencing termination – on the other hand, a short program may perform a few intricate operations that make an automated termination proof difficult. Thus, an overly simple approach that computes measures on the original program cannot yield a proper classification.

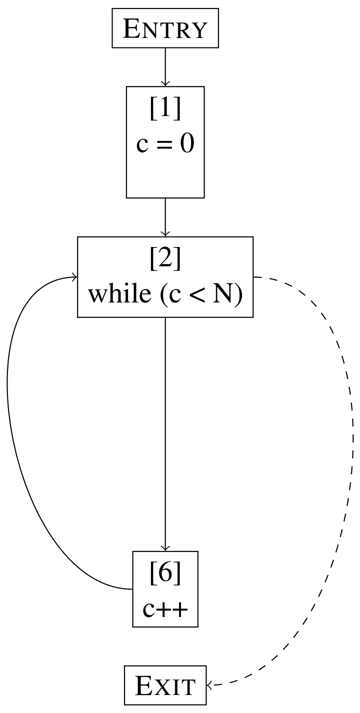
Example. Consider the flow graph in Figure 2.4a: While there are two paths, $p_1 = (2 \to 3 \to 4 \to 6)$ and $p_2 = (2 \to 3 \to 5 \to 6)$, for the loop's termination it is irrelevant which path is taken. More precisely, the loop's termination depends on the expression in block [2], and as such on the variables `c` and `N`. As neither `c` nor `N` are written in blocks [4] and [5], whether path p_1 or p_2 is executed, is irrelevant to the loop's termination. Figure 2.4b shows a program slice ignoring statements not relevant to termination.

Program slicing is a method first described by Weiser [54]:

Definition 6 (Program state, program trace)**.** Let P be a program and V the set of variable names occurring in P. A *program state* of P is a tuple (n, q) where n is a program location in P and q is a function mapping variable names to their values. A *program trace* is a sequence $(n_0, q_0), (n_1, q_1), \ldots$ of program states.



(a) A program with two paths p_1, p_2 that don't influence its termination. p_1 is shown in blue, p_2 in red.



(b) The sliced program containing only variables and statements relevant to termination.

Figure 2.4: Program slicing.

Definition 7 (Slicing criterion, program slice)**.** A *slicing criterion* $C = \langle l, V\rangle$, where l is a program location and V is a set of variables to observe at l, defines a projection of program states

$$\text{Proj}_{\langle l,V\rangle}(n,q) = \begin{cases} \varepsilon & \text{if } n \neq l \\ (n, q|V) & \text{otherwise} \end{cases}$$

where ε is the empty string and $q|V$ is q restricted to domain V. We extend the projection to traces:

$$\text{Proj}_{\langle l,V\rangle}(s_0, s_1, \dots) = \text{Proj}_{\langle l,V\rangle}(s_0), \text{Proj}_{\langle l,V\rangle}(s_1), \dots$$

Given a program P and an input I, let $T(P,I)$ denote the trace of running P on I. A *program slice* P' with respect to slicing criterion C is any program obtained by removing statements from P, such that $\text{Proj}_C(T(P,I)) = \text{Proj}_C(T(P',I))$.

Weiser [54] also defines a fixed point iteration algorithm to compute program slices.

Termination Slicing Criteria

Because a loop may be exited at multiple locations, a slicing criterion specifying a window for observing loop termination has to consider multiple program locations. Thus, we adapt

Weiser's definition of a slicing criterion to a *generalized slicing criterion* $C' = \{(l, V) \mid l \text{ is a program location, and } V \text{ is a set of variables}\}$, i.e. C' is a set of slicing criteria. A generalized slicing criterion observing loop termination for a natural loop's flow graph is given by

$$C_{term} = \bigcup_{P \in preds(\textsc{Exit})} \big\{(P, \{v \mid v \text{ is a variable in } assume(P, \textsc{Exit})\})\big\}$$

Intuitively, the criterion observes the value of any variable in the conditions guarding termination. From now on, when referring to a sliced program, we mean a program sliced with respect to criterion C_{term}. Weiser's original algorithm can easily be adapted to deal with a generalized slicing criterion. We give a non-trivial example for slicing with respect to C_{term}:

Example. Figure 2.5a shows a loop statement adapted from `rpe.c` in the cBench benchmark suite [18], and Figure 2.5b the CFG of the corresponding natural loop. Program locations defined in the slicing criterion, i.e. the predecessors to $\textsc{Exit}$ – blocks [2] and [6] – are included in the slice by default, as they directly control termination. We apply a red frame to statements included in the slice.

We then compute the preliminary set of relevant variables for each statement – clearly, from the definition of a slicing criterion, for each pair $(l, V) \in C_{term}$, all $v \in V$ are relevant at l. For C_{term} this means `i`, obtained from the edge label $assume(2, \textsc{Exit}) : i > 5$, is relevant at block [2], `exp` referenced in $assume(6, \textsc{Exit}) : exp > 5$ is relevant at block [6]. For any other statement n, we have to consider its immediate successors: If variable v is relevant at an immediate successor m and v is defined at n, then all variables referenced in defining v are relevant at n. For an example, see block [8]: `exp` is relevant in [9] and defined in [8], thus `exp` is no longer relevant, but `d`, which is referenced to define `exp`, is. If v is relevant at successor m but not defined at n, v is still relevant at n. For an example, refer to block [7]: no variables are defined, thus it inherits its relevant variables from both [8] and [9]. Figure 2.5c shows the relevant variables after this step – for simplicity we summarize the relevant variables for all statements of a block in parentheses.

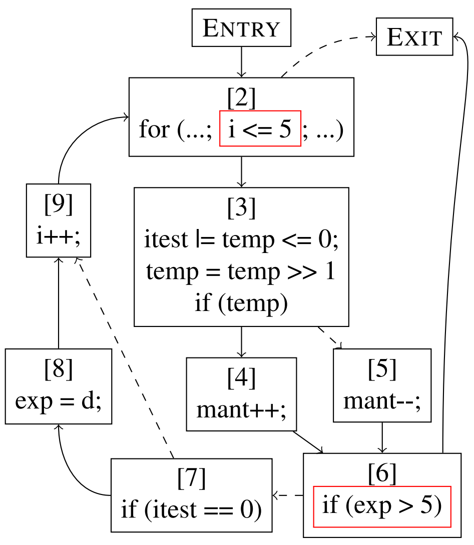
Next, we compute the preliminary set of statements to include in the slice: These are all statements n defining a variable v that relevant in an immediately succeeding statement. There are two statements of this kind: the assignment of `exp` in block [8] and the increment of `i` in block [9]. Thus, we include them in the slice, in addition to the statements from blocks [2] and [6] that were included from the beginning. Figure 2.5c highlights all statements included in the slice after this step with a red frame.

So far, we have only considered statements directly influencing the slicing criterion. Consider block [7], which decides if the statement in block [8], which we already included in the slice, is executed – we say [8] is *control-dependent* on [7]. Thus, we add those branch statements to the slice that statements already included in the slice are control-dependent on, and introduce a *branch statement criterion* $(b, REF(b))$ for each such branch statement b where $REF(b)$ is the set of variables referenced in b. To illustrate why this is necessary, consider block [7]: variable `itest` controls subsequent execution of statements in the slice, thus we need to observe its value at this location.

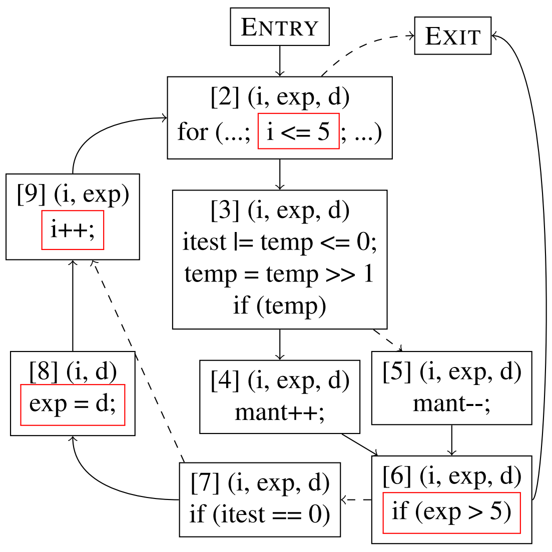
We re-iterate computation of relevant variables and statements included in the slice with respect to the already computed relevant variables, and all variables relevant to newly introduced

```
1   for (i = 0; i <= 5; i++) {
2       itest |= temp <= 0;
3       temp = temp >> 1;
4
5       if (temp) {
6           mant++;
7       } else {
8           mant--;
9       }
10
11      if (exp > 5)
12          return;
13
14      if (itest == 0)
15          exp = d;
16  }
```

(a) A loop statement adapted from `rpe.c` in the cBench benchmark.

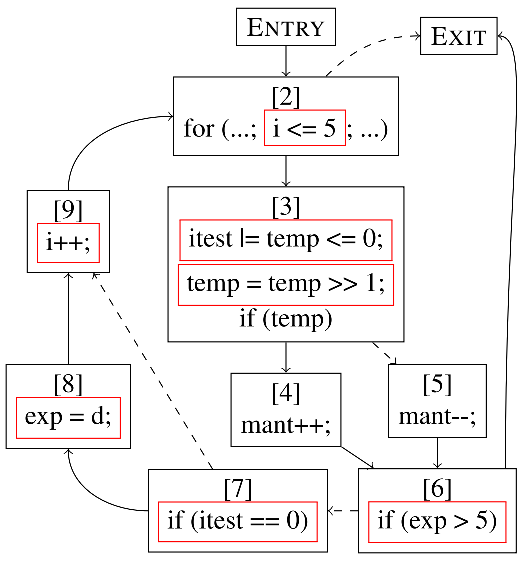


(b) The corresponding natural loop's CFG.



ENTRY
EXIT
[2] (i, exp, d) for (...; i <= 5 ; ...)
[3] (i, exp, d) itest |= temp <= 0; temp = temp >> 1 if (temp)
[4] (i, exp, d) mant++;
[5] (i, exp, d) mant--;
[6] (i, exp, d) if (exp > 5)
[7] (i, exp, d) if (itest == 0)
[8] (i, d) exp = d;
[9] (i, exp) i++;

(c) The CFG after one iteration of Weiser's algorithm.



(d) The fixed point computed by Weiser's algorithm.

Figure 2.5: An example for the termination slicing process.

branch slicing criteria until a fixed point is reached. Figure 2.5d shows the statements ultimately included in the slice: Lines 5–9 of the original program are not included, because `mant`, and thus the `if` statement on line 5 have no influence on the loop's termination.

2.4 Metric-based Classification

Basic structural classification inspired by classic software metrics can be defined on the CFG. Zuleger et al. [58] consider loops with exactly one path (in the program slice) as *trivial*. While determining the class of loops with a single path is linear in the number of basic blocks, for more than one path this notion becomes undesirable for our purposes for two reasons:

1. If we consider all paths, the number of paths may still be countably infinite, caused by the presence of a nested loop.
2. If we restrict our definition to all simple paths, counting them is hard (shown *#P-complete* in [53] for general graphs – we do not know of a result for reducible graphs).

Rather than counting the paths of a loop, we will therefore consider the number of exits from the loop, in terms of software metrics a size count:

Definition 8. Let $\mathcal{L}$ denote the set of all loops. Then $\mathcal{L}^{E=i}$ is defined as the class of loops with i exits, i.e. $\mathcal{L}^{E=i} = \{L \in \mathcal{L} \mid |preds(\textsc{Exit})| = i\}$.

One may also impose other relations than equality to describe loops with a number of exits from an interval, as demonstrated in Figure 2.6.

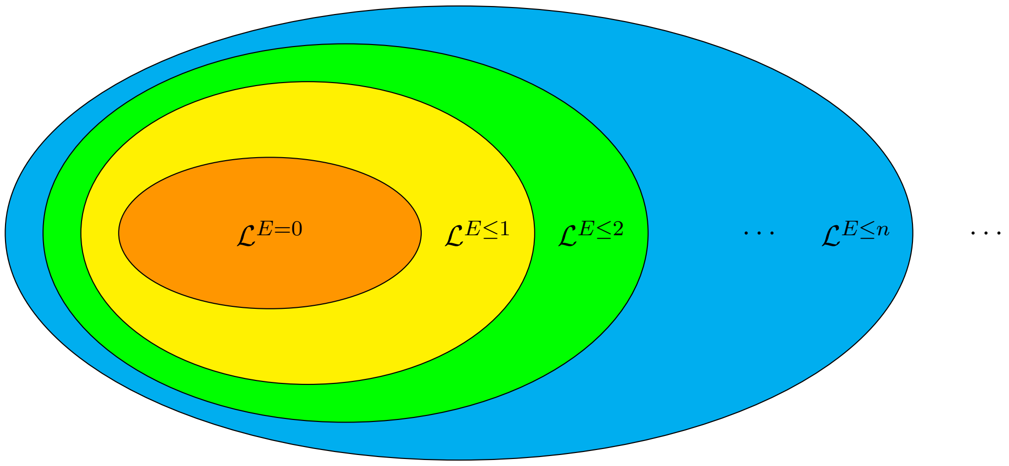


Figure 2.6: Loops with at most i exits $\mathcal{L}^{E\leq i}$ are totally ordered under inclusion.

Slice-based Metrics

Software metrics and their measurement are a long standing topic of discussion in the software engineering community, and many metrics routinely used in industry have been dismissed as inconclusive for various reasons [24]. In his original paper [54], Weiser lists a number of slicing-based metrics. As only computation influencing termination remains in a sliced program, we can consider an additional size count and two complexity metrics of the loop:

Number of basic blocks The number of basic blocks in the sliced CFG. This is merely measuring size of the slice, but because it is adjusted for termination, it is more meaningful than just referring to lines of code (LOC).

Nesting depth The depth of a depth-first spanning tree of the sliced CFG. This is a complexity software metric. Intuitively, there should be a reason for the programmer to introduce additional branches influencing termination, and we want to study if there is a correlation to the difficulty of proving termination.

Control variables The number of variables affecting what is observed through the slicing criterion C_{term}. For this metric, we compute the union of relevant variables over all statements. Another complexity metric, this gives an estimate of how intricate the computation controlling the terminating branch statements is.

CHAPTER 3

Syntactic Termination Proofs

One class of loops that an automated tool should definitely be able to handle, are those that can be shown to terminate relatively easily, employing light-weight techniques such as data-flow analysis. This class should then be regarded as the base case any tool must handle, the amount of such trivial cases in benchmarks reported, and deviations from an prior established value be acknowledged and discussed.

3.1 Motivating Example

Our syntactic termination proof is based on the observation that many loops alter a set of variables during each iteration. These variables are then compared against a fixed (loop-invariant) bound to decide termination.

Listing 3.1 Motivating example for the syntactic termination proof.

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

Consider the program shown in Listing 3.1: 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 (c.f. Figure 3.1). As `i` is incremented during each iteration, we eventually have $i \in [N, \infty]$, and thus $\neg P(i)$ and the loop terminates.

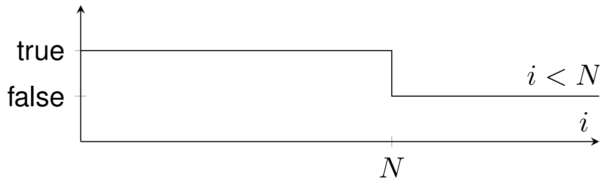


Figure 3.1: $i < N$

The following sections discuss the analysis of syntactic loop termination in detail: Section 3.2 introduces data-flow analysis, the framework our classification is based on. Section 3.3 describes how the increment of a variable is determined, Section 3.4 introduces predicates which eventually hold given this increment, and Section 3.5 defines the control flow of loops where such predicates lead to termination.

3.2 Data-Flow Analysis

We will make heavy use of data-flow analysis in this chapter, therefore we give an introduction in this section. Data-flow analysis is a well-known technique for static analysis introduced by Kildall [37], often used by compilers to obtain information used in optimizations [1].

Data-Flow Analysis Frameworks

Data-flow values describe sets of states that may be observed at a program point (e.g., the set of definitions reaching that point). Let $\textsc{In}[B]$ and $\textsc{Out}[B]$ be the data-flow values just before and after executing basic block B.

A family of *transfer functions* $F_B = \{f'_B, f''_B, \ldots, f^n_B\}$ relates $\textsc{In}$ and $\textsc{Out}$ of block B, depending on the direction of information flow: For forward data-flow problems, we have $\textsc{Out}[B] = f^i_B(\textsc{In}[B])$, while for backward data-flow problems we have $\textsc{In}[B] = f^i_B(\textsc{Out}[B])$.

The *meet operator* $\wedge$ combines data-flow values from multiple branches: In a forward data-flow problem, $\textsc{In}$ for a block B is computed from $\textsc{Out}$ of its predecessors:

$$\textsc{In}[B] = \bigwedge_{P \in preds(B)} \textsc{Out}[P]$$

while in backward data-flow problems, $\textsc{Out}$ for B is computed from $\textsc{In}$ of the successors:

$$\textsc{Out}[B] = \bigwedge_{S \in succs(B)} \textsc{In}[S]$$

The *data-flow problem* is to compute the values of $\textsc{In}$ and $\textsc{Out}$ for all basic blocks in the flow graph. Data-flow problems are commonly described as *data-flow analysis frameworks*:

Definition 9 (Semilattice, meet-semilattice)**.** A *semilattice* is an algebraic structure $\langle S, * \rangle$, where S is a set and $*$ a binary operation, such that for all $x, y, z \in S$

$$x * (y * z) = (x * y) * z \text{ (associativity)} \tag{3.1}$$

$$x * y = y * x \text{ (commutativity)} \tag{3.2}$$

$$x * x = x \text{ (idempotency)} \tag{3.3}$$

Given a partially ordered set P in which each pair of elements x, y has a greatest lower bound $glb(x, y)$, define $x \wedge y = glb(x, y)$. Then, $\langle P, \wedge \rangle$ is a semilattice – we call $\wedge$ the *meet operator* and $\langle P, \wedge \rangle$ a *meet-semilattice*.

Definition 10 (Data-flow analysis framework)**.** A *data-flow analysis framework* is a tuple $(S, \wedge, D, F)$, where $\langle S, \wedge \rangle$ is a meet-semilattice, $D \in \{\mathsf{forward}, \mathsf{backward}\}$ the direction of information-flow, and F a family of transfer functions $f^i : S \to S$. Two families of frameworks are particularly useful: *Monotone frameworks*, for which $\forall f \in F.\, (x \leq y \to f(x) \leq f(y))$, and *distributive frameworks*, for which $\forall f \in F.\, (f(x \wedge y) = f(x) \wedge f(y))$.

Solving Data-Flow Problems

Without loss of generality, we assume forward-flowing data-flow frameworks below. If a solution to the data-flow problem exists, it can be computed by a fixed point iteration as shown in Algorithm 1: First, OUT of each block is initialized to a special value $\top$. In the fixed point iteration, the meet operator $\wedge$ as well as transfer functions f^i_B are used to propagate information and compute updated data-flow values. If the algorithm converges, it computes a solution to the data-flow problem.

An optimized version only considers blocks that need (re-)computation in each iterative step. As these blocks are kept in a worklist, this approach is commonly referred to as *worklist algorithm*.

Algorithm 1 Iterative algorithm for forward data-flow problems [1].

```
for each basic block $B$ do
    OUT[$B$] = $\top$
end for
while any OUT changes do
    for each basic block $B \neq$ ENTRY do
        IN[$B$] = $\bigwedge_{P \in preds(B)}$ OUT[$P$]
        OUT[$B$] = $f^i_B$(IN[$B$])
    end for
end while
```

Convergence

Convergence of the fixed point computation for a framework $\langle S, \wedge, D, F \rangle$ is usually proven by showing that $\langle S, \wedge \rangle$ is of finite height and the family of transfer functions F is monotonic:

As Algorithm 1 shows, initially $\textsc{Out}[B] = \top$ for all basic blocks B. In the fixed point iteration, $\textsc{In}[B]$ meets over all $\textsc{Out}[P]$ of B's predecessors P, i.e. $\textsc{In}[B]$ is the greatest lower bound of all $\textsc{Out}[P]$. In monotone frameworks, the old data-flow value $\textsc{Out}$ must be less or equal to the updated value $\textsc{Out}'$: $\textsc{Out}[B] \leq \textsc{Out}'[B]$. Furthermore, if the semilattice is of finite height, Algorithm 1 must eventually reach a fixed point and thus terminate. By construction, the computed solution is the greatest fixed point.

Solution

Obviously, the obtained solution (greatest fixed point) can be different from the ideal solution: data-flow analysis considers a superset of possible executions, by treating any path as feasible. We discuss guarantees on the obtained solution in terms of the ideal solution:

Let $P : B_1, \ldots, B_n$ be a path from $\textsc{Entry}$ to B_n, and f_P the composition of transfer functions $f_{B_1}, \ldots f_{B_{n-1}}$. Then the ideal solution is given by:

$$\textsc{In}_{ideal}[B] = \bigwedge_{P.P \text{ is a feasible path from } \textsc{Entry} \text{ to } B} f_P(\top)$$

By considering all (feasible *and* infeasible) paths as the data-flow abstraction does, we obtain a meet-over-paths solution

$$\textsc{In}_{mop}[B] = \bigwedge_{P.P \text{ is a path from } \textsc{Entry} \text{ to } B} f_P(\top)$$

Meeting a superset of paths, the meet-over-paths solution cannot be greater than the ideal, i.e. $\textsc{In}_{mop}[B] \leq \textsc{In}_{ideal}[B]$.

Additionally, the iterative algorithm applies the meet operator $\wedge$ early. Whereas the meet-over-paths solution applies meet over all paths (which are infinite in the presence of loops), the iterative algorithm applies meet over finitely many incoming branches. By induction over the path length, one can show that for monotone frameworks $\textsc{In}[B] \leq \textsc{In}_{mop}[B] \leq \textsc{In}_{ideal}[B]$. For distributive frameworks, $\textsc{In}[B] = \textsc{In}_{mop}[B]$, because function application distributes over $\wedge$.

3.3 Accumulated Increment

For the syntactic termination proof, it is necessary to determine the possible values by which a variable `i` is updated in one iteration of a loop. We call this set of values the *accumulated increment*. We need to consider consecutive assignments to the same variable `i` along one path and fold them into a single value. Additionally, different paths of execution might lead to different values for the accumulated increment. For example, in Listing 3.1 the accumulated increment `i` is 1 if execution takes the `if` branch, and 2 if execution takes the `else` branch. We restrict our analysis to constant increments.

There are two cases in which we cannot establish a constant increment:

1. The increment is non-constant, for example if the increment is symbolic (e.g. `i += x`). In this case, we cannot fold the symbolic increment into a constant value.

2. If the loop contains a nested loop, the accumulated increment of a variable whose value changes in this loop is not uniquely defined along the non-simple paths introduced by the nested loop.

Definition 11 (Accumulated increment along a path)**.** We use unprimed (i) and primed (i') versions to refer to variable `i`'s value just before and just after one iteration of the loop. The accumulated increment of variable `i` along a path p is defined as

$$accinc_p(\texttt{i}) = \begin{cases} i' - i & \text{if } i' \text{ is uniquely defined and } i' - i \text{ is constant} \\ \mathbb{Z} & \text{otherwise} \end{cases}$$

Intuitively, if we cannot determine a constant increment, we have to assume `i` is incremented by any possible value in $\mathbb{Z}$.

Definition 12 (Accumulated increment of a single iteration)**.** The accumulated increment of an iteration of loop L then, is the accumulated increment along all possible paths, i.e.

$$AccIncs_L(\texttt{i}) = \bigcup_{p \in paths(L)} accinc_p(\texttt{i})$$

Implementation

Computation of the accumulated increment can be reduced to solving a standard data-flow problem for the framework $\langle S, \cup, \textsf{forward}, F\rangle$, where

$$S = \mathcal{P}(\mathbb{Z}) \tag{3.4}$$

$$F(B) = \begin{cases} f_{\text{HEADER}}(X) = \{0\} \\ f_{B \neq \text{HEADER}}(X) = \{x + accinc_B(i) \mid x \in X\} \end{cases} \tag{3.5}$$

Theorem 1. *If solving the data-flow equations for the given framework on a natural loop's flow graph L for variable i computes a greatest fixed point, the accumulated increment $AccIncs_L(i)$ is given as data flow value in* IN[HEADER]. IN[HEADER] *is equal to the meet-over-paths solution.*

Proof. We show $\langle S', \cup, F'\rangle$ is a distributive framework: As the boundary transfer function f'_{HEADER} is constant, it is trivially distributive. It remains to show distributivity of f'_B:

$$f'_B(X \cup Y) = f'_B(X) \cup f'_B(Y) \tag{3.6}$$

$$f'_B(\{z \mid z \in X \cup Y\}) = f'_B(\{x \mid x \in X\}) \cup f'_B(\{y \mid y \in Y\}) \tag{3.7}$$

$$\{z + a \mid z \in X \cup Y\} = \{x + a \mid x \in X\} \cup \{y + a \mid y \in Y\} \tag{3.8}$$

$$\{z + a \mid z \in X \cup Y\} = \{z + a \mid z \in X \cup Y\} \text{ for some increment } a \in \mathbb{Z} \tag{3.9}$$

□

Note that using the family of transfer functions F given in Equation (3.5), the worklist algorithm does not converge in the presence of cycles (nested loops) p that perform non-zero increments $x \in accinc_p(i), x \neq 0$ on i. More formally, the semilattice $\langle S, \cup \rangle$ is of infinite height.

We describe an optimization that computes the data-flow solution in two forward passes through the CFG and always terminates: We start by traversing the natural loop's CFG from ENTRY using a depth-first strategy, initialize each basic block B with the accumulated increment $accinc_b(i)$ of variable i along the singleton path b of B. At the same time, we keep track of the accumulated increment $accinc_p(i)$ along the currently explored path p from ENTRY to the currently explored node. If we reach an already explored block n (i.e. we encountered the back edge $t \to n$ of a nested loop), before backtracking we compare its already assigned data-flow value $accinc_n(i)$ with the new data-flow value $accinc_p(i)$. If $accinc_n(i) \neq accinc_p(i)$, we update the data-flow value of n to $\mathbb{Z}$. In a second pass, we forward propagate the now initialized values from ENTRY through the CFG by combining incoming values with the meet operator $\cup$ and computing the pointwise sum of the incoming data-flow value and the current block's initial data-flow value from the first pass as defined in Equation (3.5).

3.4 Terminating Predicates

In this section, we describe the predicates for which we can show syntactic termination, by means of the predicate's *representing function*:

Definition 13 (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 $\exists x \,.\, f_P(x) = 0$. These definitions are inspired by [40].

Definition 14 (Exit block, exit predicate)**.** Let L be a natural loop's flow graph, (B, EXIT) an edge in L, and $P : assume(B, \text{EXIT})$ the edge's label. We call B an *exit block* and P an *exit predicate*.

Depending on the predicate's form, we describe four strategies for showing that the predicate is eventually true. These strategies cover a subset of all eventually true predicates, and where chosen to represent cases that in our experience occur in practice:

Escape The predicate $P(i)$ holds iff i does not take a special value N (e.g. $i \neq N$). If i "escapes" N, P is eventually true.

Search The predicate $P(i)$ holds iff i takes a special value N (e.g. $i = N$). If i "searches" and eventually "finds" N, P is eventually true.

Increase The predicate $P(i)$ holds iff i takes values from an interval $[N, \infty]$ (e.g. $i \geq N$). If i increases enough to enter this interval, P is eventually true.

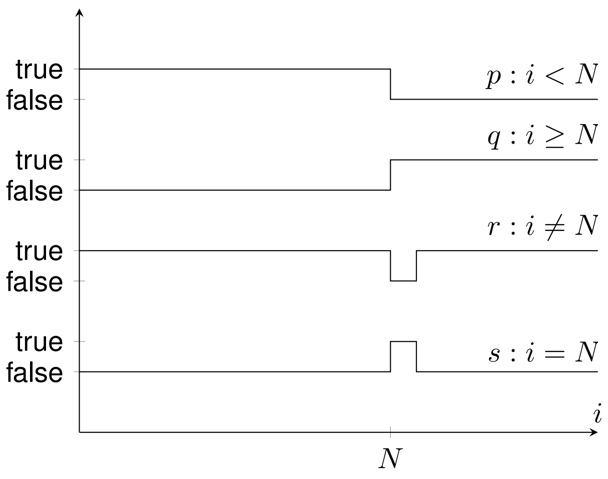


Figure 3.2: Monotonic (p, q) and eventually true (p, q, r, s) predicates.

Escape	f_P has a single non-root $\exists! x . f_P(x) = 1$.
Search	f_P has a single root $\exists! x . f_P(x) = 0$.
Increase	f_P is monotonically decreasing and eventually true.
Decrease	f_P is monotonically increasing and eventually true (f_P grows from 0).

Table 3.1: Mapping of representing function f_P's characteristics to strategies for showing an exit predicate P is eventually true.

Decrease The predicate $P(i)$ holds iff i takes values from an interval $[-\infty, N]$ (e.g. $i \leq N$). If i decreases enough to enter this interval, P is eventually true.

Figure 3.2 illustrates these strategies, and Table 3.1 contains formal definitions of the strategies just introduced in terms of the predicate's representing function. We call predicates whose representing function takes such a form *well-formed*:

Definition 15 (Well-formed predicate)**.** Let L be a natural loop's flow graph, (B, EXIT) an edge in L, and $P : assume(B, \text{EXIT})$ the edge's label. We call P a *well-formed predicate* if and only if its representing function f_P matches one of the forms in Table 3.1.

Given a predicate's representing function $f_P(i)$, we first determine the appropriate strategy as given in Table 3.1. For the obtained strategy, we need to consider updates on i in loop L to decide if the strategy's conditions hold: We use the accumulated increment $AccIncs_L(i)$ from Section 3.3 to give sufficient conditions for an exit predicate $P(i)$'s eventually evaluating to true for a given loop L:

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

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

Table 3.2: Sufficient conditions for an exit predicate's terminating the loop.

type's range. The condition ensures all increments are integer successor functions in a single direction. Assuming[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 16 (Terminating block, terminating predicate)**.** Let $P : assume(B, \textsc{Exit})$ be an exit predicate for which the condition given by Tables 3.1 and 3.2 hold. We call B a *terminating block* and P a *terminating predicate*.

Implementation

Finding the set of exit predicates $P : assume(B, \textsc{Exit})$ for a natural loop L amounts to enumerating the predecessors of EXIT. The predicate is given as C language expression on the branch statement of B. We translate this C expression to a Z3 expression [43] and use Z3's theory simplifiers to obtain a normal form. We then perform pattern matching on the obtained normal form to choose a strategy according to Table 3.1.

At the moment, our analysis only considers linear inequalities in a single variable i. Any other subexpressions must be loop-invariant. This allows us to handle condition expressions with common comparison operators of the C programming language as top-level connective, i.e. the equality operators == and != and relational operators <, >, <=, >=.

It is important to note that we do not employ Z3 as SMT solver. We merely take advantage of its (mostly syntactic) expression rewriting to obtain a normal form for pattern matching.

Assumptions

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 [36, § 6.5.6]. We assume the result of pointer arithmetic is always defined.

[1]An in-depth discussion of assumptions made is given below.

3. As another subtlety, the C standard does not define overflow on signed integer types [36, § 6.5.6]. The **search** strategy relies on covering the whole value range, thus we have to assume two's complement wrap-around behavior on overflow for **search** predicates over signed integer types.

   We may also prove termination of additional **increase/decrease** predicates under this assumption, e.g. in the loop

   ```
   for (unsigned i = 42; i < N; i--) ;
   ```

   However, if we prove **increase/decrease** predicates over *signed* integers using this assumption, we probably[2] discovered a bug [20]. We describe one such bug we encountered during experimental evaluation in Chapter 5.

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.

   Example. Consider the program shown in Listing 3.2: Depending on the values assigned to `d` and `N`, the loop may or may not terminate. Type information itself is not sufficient, as addition of `d` lifts the left hand side type to 32-bit integers, however this expression's range is still influenced and limited by `i`'s 8-bit type.

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

Listing 3.2 Loop terminating under assumptions on expression ranges.

```
int32_t d = /* ... */,
        N = /* ... */;
for (int8_t i = 0; i + d < N; i++);
```

3.5 Control Flow Constraints

Having defined the terminating predicates, we need to make sure that a terminating predicate $P : assume(B, \textsc{Exit})$ is actually evaluated (i.e. block B is executed) when it evaluates to true. Listing 3.3 illustrates that this is not always the case: while there is a terminating predicate $P : i = 42$, it is never evaluated when $i = 42$. Hence, line 5 is never executed – the loop does not terminate.

As determining feasibility of a loop's terminating predicates is in itself a hard problem, we restrict our analysis to a case in which we can ensure evaluation of the predicate:

[2]The only exception is if the compiler assumes two's complement signed integer overflow as well, which may be enabled in Clang or GCC using the -fwrapv flag.

Listing 3.3 Terminating predicate P does not terminate the loop.

```
int i = 0;
while (1) {
    if (i < 42) {
        if (i == 42) {   // terminating predicate P: i = 42
            break;       // unreachable
        }
    }
    i++;
}
```

Theorem 2. *Given a terminating predicate $P : assume(B, \text{EXIT})$ of a natural loop L, P terminates L if its associated terminating block B lies on each path through the loop.*

Definition 17 (Syntactically terminating loop)**.** Given a natural loop L, we call L *syntactically terminating* $L \in \mathcal{L}^{ST}$ if and only if there exists a terminating predicate $P : assume(B, \text{EXIT})$ of L and its associated terminating block B lies on each path through the loop.

Implementation

The condition in Theorem 2 may be decided by solving the data-flow framework $\langle \mathbb{N}, \min, \mathsf{forward}, F \rangle$, where

$$F(B) = \begin{cases} f_{\text{HEADER}}(x) = 0 \\ f_{B \neq \text{HEADER}}(x) = \begin{cases} x + 1 & |succs(B)| > 0 \\ x & \text{otherwise} \end{cases} \end{cases}$$

Intuitively, the analysis assigns to each basic block the number of open (un-joined) branches. Any basic block B of natural loop L with $\text{OUT}[B] = 0$ lies on all paths through L. For a given loop, height of the semilattice can be restricted to the maximum number of open branches M along any path, with $\top = M, \bot = 0$. $\langle \mathbb{N}, \min, F \rangle$ is a distributive framework, because $\min\{x + 1, y + 1\} = \min\{x, y\} + 1$.

3.6 Strengthening Syntactic Termination

So far, we have only considered an isolated notion of loop termination: If execution starts from ENTRY, any path of execution reaches EXIT. A stronger notion which is essential for symbolic bound computation considers a loop to terminate only if the number of executions of the loop's paths is bounded. Listing 3.4 shows an example where the two notions differ: 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:

Listing 3.4 An unbounded loop.

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

Definition 18 (Syntactically bounded loop)**.** Given a natural loop $L \in \mathcal{L}^{ST}$, we call L *syntactically bounded* $L \in \mathcal{L}^{SB}$ if and only if L itself and all of its nesting (outer) loops are in $\mathcal{L}^{ST}$.

3.7 Loop Counters

We can use the previous definition of terminating predicates to define *loop counter variables*:

Definition 19 (Loop counter variable)**.** Given a terminating predicate $P(i) : assume(B, \textsc{Exit})$ of a natural loop L, we call i a *loop counter variable* of L.

Note that there may be more than one terminating block for whose terminating predicate we can obtain a syntactic termination proof. Thus, the set of loop counters $LoopCounters(L)$ contains any i for which a termination proof may be obtained. In other words, in order to build $LoopCounters(L)$, we need to find *all* proofs, not just a single one that is required to show termination.

CHAPTER 4

Simple Loops

```
I like the little-known operator
"-->", also know as "goes to."

Here's an example:

#include <stdio.h>
int main()
{
  int x = 10;
  while( x --> 0 ) // x goes to 0
  {
    printf("%d ", x);
  }
}
```

— comp.lang.c++.moderated, *Hidden Features and Dark Corners of C++/STL*

In Chapter 3 we introduced a class of loops for which termination can be shown with minimal information in a *syntactic termination proof*. These proofs were obtained by leveraging locally available information about updates to a counter variable and the syntactic structure of the loop's exit predicates. Obviously, current techniques and tools are capable of handling more complicated cases. As we shall see in Chapter 5, LOOPUS can bound 79% of the loops in the cBench benchmark suite, while only 37% of all loops are syntactically terminating loops.

Formal methods-based techniques like LOOPUS usually achieve this wider coverage by considering a more complete representation of the program: The effect of statements is chained together, usually into a formula in classical logic that represents the whole program. Search for a property is then left to an off-the-shelf SAT or SMT solver. This approach is illustrated in Listing 4.1, where the comments show an encoding of the C program in the first-order theories of equality and linear arithmetic.

Listing 4.1 A sample C program expressed as conjunction over equality and linear arithmetic.

```
int i = 0;          // i0 = 0,
i++;                // i1 = i0 + 1,
if (i == 0) {       // i1 = 0 → (
    i--;            //        i2 = i1 − 1
} else {            // ), i1 ≠ 0 → (
    i++;            //        i2 = i1 + 1
}                   // ),
assert(i==2);       // i2 = 2
```

.

It is unreasonable to reimplement such an approach for our analysis, as it would basically make our tool SLOOPY a tool that *proves termination*, instead of a tool that *efficiently determines difficulty to prove termination*. Instead, to express a superset of syntactically terminating loops we will exploit the fact that programmers follow patterns: Assume an at least slightly experienced programmer is asked to implement a subprogram that performs a task n times. He will probably come up with program P, or some slight variation of the same basic pattern, even though infinitely many subprograms conforming to the same specification exist (e.g. illustrated in program Q below).

Listing 4.2 Example program P.

```
for (int i = 0; i < n; i++) {
    // perform task
}
```

Listing 4.3 Example program Q.

```
int j = 3;
int *i = &j;
loop:
// perform task
*i = j-1;
j += 2;
if (3+n == **&i) ;
else goto loop;
```

If we can weaken syntactically terminating loops to describe such patterns in a way that leaves enough leeway to match a considerable amount of loops, but still captures the termination-related properties of syntactically terminating loops well, we have found our desired classification. Additionally, a classification with respect to these patterns can be used to guide the non-trivial decision where to apply and how to parametrize formal methods, as discussed in the introduction to this work. We describe *simple loops*, a family of such patterns:

4.1 Dimensions of Simple Loops

As proposed, we derive simple loops from syntactically terminating loops. We do so by systematically weakening the constraints used to define syntactically terminating loops along three

```
for (int i = 0; ; i++) {
    if (i < 100) {
        if (i == 10) break;
    } else {
        if (i == 110) break;
    }
}
```

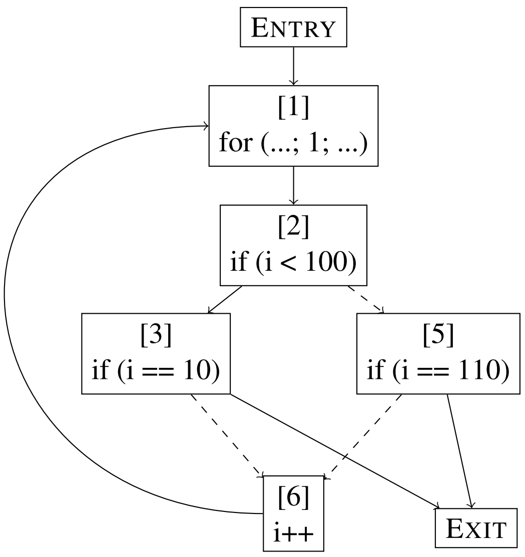


Figure 4.1: A terminating loop for which our analysis cannot compute a syntactic termination proof because of non-local control flow.

dimensions, allowing us to tweak how many features of this class are retained. We start by introducing these dimensions and then describe the family of all possible combinations along these dimensions.

Control flow constraints In Section 3.5, we put a strong restriction on the control flow of syntactically terminating loops: The terminating block b associated with a terminating predicate $P : assume$(b, EXIT) has to lie on each path through the loop. Figure 4.1 illustrates a loop that terminates, but does not have a singleton terminating block on all paths:

Example. The loop shown in Figure 4.1 counts `i` upwards from 0. When $i = 10$, the `if` statement in block [2] branches to block [3]. There, the `if` statement's condition evaluates to true, the `break` statement is executed and leaves the loop. However, because of the locality of our analysis, we cannot capture the interplay between block [2] and blocks [3] or [5], respectively. More formally, our path-insensitive analysis cannot decide that block [2] branches to block [3] when $i = 10$.

This lends itself to introduce the following heuristic instead: Given the set of all terminating blocks T, assume some terminating block in $t \in T$ associated with exit predicate P is executed at some point when P holds, thus terminating the loop. We give a formal definition:

Definition 20. Given a natural loop L, let predicate $tb : L \to \{\text{true}, \text{false}\}$ determine if its argument is a terminating block, i.e. $tb(b)$ if and only if b is a terminating block. The three control flow constraints $\mathsf{s}, \mathsf{p}, \mathsf{w}$ are defined as follows:

$$\begin{aligned}\mathsf{p} &: \exists b \in L.\, \forall p \in paths(L).\, tb(b) \wedge b \in p \\ \mathsf{s} &: \forall p \in paths(L).\, \exists b \in L.\, tb(b) \wedge b \in p \\ \mathsf{w} &: \exists b \in L.\, tb(b)\end{aligned}$$

Intuitively, p requires a singleton terminating block that lies on each path, s requires some terminating block (not necessarily the same) on each path, and w merely requires existence of some terminating block.

Well-formedness constraints As discussed in Section 3.3, we cannot fold non-constant increment values into the accumulated increment. However, often a symbolic delta will take sensible values, but again with our local, path-insensitive analysis we cannot establish those. In Listing 4.4 we illustrate such a terminating loop for which our analysis cannot establish a syntactic termination proof:

Listing 4.4 A terminating loop for which our analysis cannot compute a syntactic termination proof because of non-constant increments.

```
int i = 0,
    D = 3;
while (i < 10) {
    i += D;
    D++;
}
```

Example. The loop in Listing 4.4 increments `i` by the value of `D`. At line 3 we consecutively see the values 0, 3, 7, 12 for `i` – at the last value iteration terminates. However, due to lack of contextual information about the value of `D` at line 4, we loose all information about the accumulated increment of `i`.

As above, we introduce a heuristic for this case: We decouple the termination property of exit predicates by not requiring the accumulated increment to imply eventual truth of the predicate:

Definition 21. Given a natural loop L satisfying a control flow constraint C, let $B \subseteq L$ denote the minimal set of blocks needed to satisfy C. Let predicate wp determine if its argument is a well-formed predicate, i.e. $wp(P)$ if and only if P is a well-formed predicate (c.f. Definition 15). Let predicate tp determine if its argument is a terminating predicate, i.e. $tp(P)$ if and only if P is a terminating predicate (c.f. Definition 16). The two well-formedness constraints t, w are defined as follows:

$$\begin{aligned}\mathsf{t} &: \exists b \in B.\, tp(assume(b, \textsc{Exit})) \\ \mathsf{w} &: \exists b \in B.\, wp(assume(b, \textsc{Exit}))\end{aligned}$$

We extend the control flow constraint above to consider blocks of well-formed predicates instead of terminating blocks if the weaker constraint w is chosen.

Invariant bound constraints As mentioned in Section 3.4, we only consider linear inequalities in a single variable for our syntactic termination proof. Other subexpressions are checked to be loop-invariant. If we omit this check, we obtain a weaker constraint on the loop's behavior.

Definition 22. Let L be a natural loop, and V the set of variables that need to be checked for loop-invariance. Let predicate $inv : V \to \{\text{true}, \text{false}\}$ determine if its argument is loop-invariant, i.e. $inv(v)$ if and only if v is loop-invariant in L. The two invariance constraints i, x are defined as follows:

$$\begin{aligned} \mathsf{i} &: \forall v \in V.\, inv(v) \\ \mathsf{x} &: \text{ true} \end{aligned}$$

In addition to the previous three dimensions that derive loop classes from the constraints of syntactically terminating loops, we can restrict simple loops by intersecting them with any other loop class. We exemplarily do this for a structural loop class from Chapter 2:

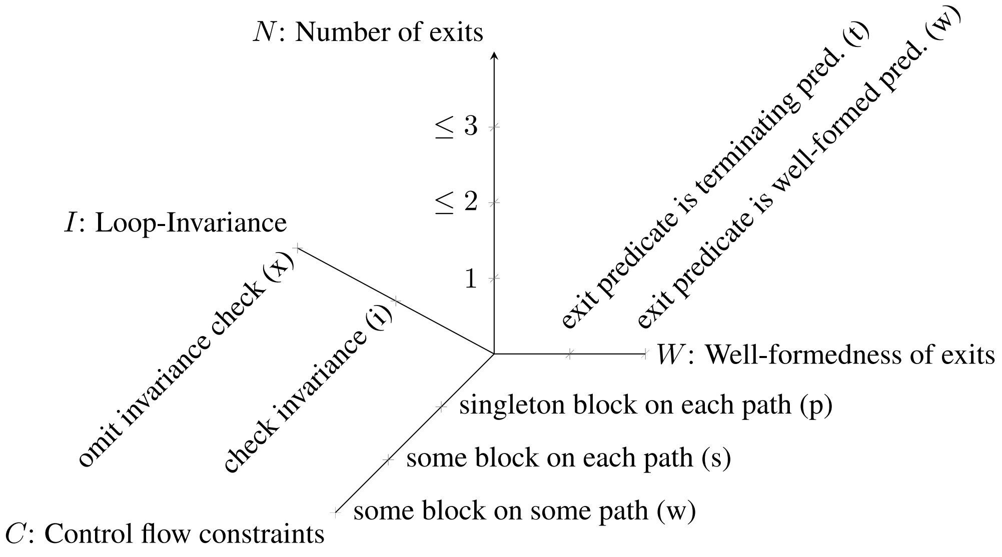
Exit constraint As discussed in Section 2.4, we can impose constraints of the number of exits. This is especially useful in conjunction with control flow constraints, e.g. we may only require a terminating block on some path, but enforce that this exit is the singleton exit from the loop. Additionally, in our experience (c.f. Section 5.2) fewer exits are a strong indicator for low difficulty to compute a termination proof (as indicated by LOOPUS). This restriction can be achieved by intersecting the set of loops in some class C and the set of loops with n exits: $\mathcal{L}^{C} \cap \mathcal{L}^{E=n}$.

Definition 23. Let L be a natural loop. The exit constraints $\mathsf{1}, ^{\star}$ are defined as $\mathsf{1} : L \in \mathcal{L}^{E=1}$ and $\star$: true.

Definition 24 (Simple loop constraint)**.** A *simple loop constraint* is a tuple (N, C, I, W), where N is an exit constraint, C is a control flow constraint, I is a loop-invariance constraint, and W is a well-formedness constraint. A natural loop L satisfies constraint (N, C, I, W) if and only if all of N, C, I, W are satisfied for L. L is a simple loop $L \in \mathcal{L}^{simple(NCIW)}$ if and only if L satisfies (N, C, I, W).

For a visual interpretation, consider Figure 4.2 showing the four categories above as independent dimensions. Moving away from the center along dimensions C, I, or W, we cover additional loops at the expense of losing soundness. Along dimension N, we do not trade soundness, but allow tweaking of the loop's simplicity via a structural measure. One may also leave the N dimension uninstantiated, thus covering loops with any number of exits.

To aid communication about simple loops, we introduce a naming scheme for the classes of the simple loop family: Loops with a single exit ($N = \mathsf{1}$) are referred to as *single-exit loops*. Depending on the control flow constraint, we have *proved-cf*, *strong-cf*, *weak-cf loops*, if they



N: Number of exits
≤ 3
≤ 2
1
I: Loop-Invariance
omit invariance check (x)
check invariance (i)
exit predicate is terminating pred. (t)
exit predicate is well-formed pred. (w)
W: Well-formedness of exits
singleton block on each path (p)
some block on each path (s)
some block on some path (w)
C: Control flow constraints

Figure 4.2: Four dimensions of simple loops.

$N = 1$	single-exit
$C = \text{p}$	proved-cf
$C = \text{s}$	strong-cf
$C = \text{w}$	weak-cf
$I = \text{i}$	invariant
$W = \text{t}$	terminating
$W = \text{w}$	well-formed

Table 4.1: Naming scheme for simple loops.

have a singleton block on each path ($C = \mathsf{p}$), some block on each path ($C = \mathsf{s}$), some block on some path ($C = \mathsf{w}$), respectively. Loops fulfilling the loop-invariance restriction ($C = \mathsf{i}$) are *invariant loops*. If the increment implies termination ($W = \mathsf{t}$) we speak of *terminating loops*. If we mitigate this constraint to a form of syntactic well-formedness ($W = \mathsf{w}$), we speak of *well-formed loops*. This naming scheme is summarized in Table 4.1.

Note that the class $\mathcal{L}^{simple(\mathsf{*pit})}$ is the class of syntactically terminating loops introduced in Chapter 3: $\mathcal{L}^{simple(\mathsf{*pit})} = \mathcal{L}^{ST}$. In addition to $\mathcal{L}^{ST}$, we obtain the eleven additional simple loop classes listed in Table 4.2 and the corresponding single-exit classes $\mathcal{L}^{simple(\mathsf{1}CIW)}$.

4.2 Structure of Simple Loops

To get a better intuition for the interrelation of different simple loop constraints, we define a partial order based on the loops covered by the constraints:

Class	Name
$\mathcal{L}^{simple(\text{*pit})}$	syntactically terminating
$\mathcal{L}^{simple(\text{*sit})}$	(any-exit) strong-cf invariant terminating
$\mathcal{L}^{simple(\text{*wit})}$	(any-exit) weak-cf invariant terminating
$\mathcal{L}^{simple(\text{*piw})}$	(any-exit) proved-cf invariant well-formed
$\mathcal{L}^{simple(\text{*siw})}$	(any-exit) strong-cf invariant well-formed
$\mathcal{L}^{simple(\text{*wiw})}$	(any-exit) weak-cf invariant well-formed
$\mathcal{L}^{simple(\text{*pxt})}$	(any-exit) proved-cf terminating
$\mathcal{L}^{simple(\text{*sxt})}$	(any-exit) strong-cf terminating
$\mathcal{L}^{simple(\text{*wxt})}$	(any-exit) weak-cf terminating
$\mathcal{L}^{simple(\text{*pxw})}$	(any-exit) proved-cf well-formed
$\mathcal{L}^{simple(\text{*sxw})}$	(any-exit) strong-cf well-formed
$\mathcal{L}^{simple(\text{*wxw})}$	(any-exit) weak-cf well-formed

Table 4.2: Simple loop classes.

Given two simple loop constraints C_1, C_2, the constraints are ordered by $C_1 \leq C_2$ if and only if $\mathcal{L}^{simple(C_1)} \subseteq \mathcal{L}^{simple(C_2)}$. The obtained partially ordered set is shown in Figure 4.3. Note that the order between any-exit and single-exit loops is only shown exemplarily for the pairs (1wxw $\rightarrow$ *wxw) and (1wxt $\rightarrow$ *wxt) to facilitate readability. Semitransparent nodes (single-exit syntactically terminating loops) refer to constraints whose instantiation is irrelevant in practice if termination is the desired property, as termination can even be shown for the less restrictive any-exit classes.

Simple Loop Constraint Transformers

To refer to a specific weakening along one simple loop dimension, we introduce *simple loop constraint transformers*: A transformer t applied to a simple loop constraint C_1 returns the next-weakest constraint $C_2 = t(C_1)$ along its associated dimension. Thus, from the dimensions identified earlier, we obtain the following transformers:

$$\begin{array}{rllllcllll}
cf_1((N=n, & C=\textsf{p}, & I=i, & W=w)) & = & (N=n, & C=\textsf{s}, & I=i, & W=w) \\
cf_2((N=n, & C=\textsf{s}, & I=i, & W=w)) & = & (N=n, & C=\textsf{w}, & I=i, & W=w) \\
inv((N=n, & C=c, & I=\textsf{i}, & W=w)) & = & (N=n, & C=c, & I=\textsf{x}, & W=w) \\
wf((N=n, & C=c, & I=i, & W=\textsf{t})) & = & (N=n, & C=c, & I=i, & W=\textsf{w}) \\
ex((N=\textsf{*}, & C=c, & I=i, & W=w)) & = & (N=\textsf{1}, & C=c, & I=i, & W=w)
\end{array}$$

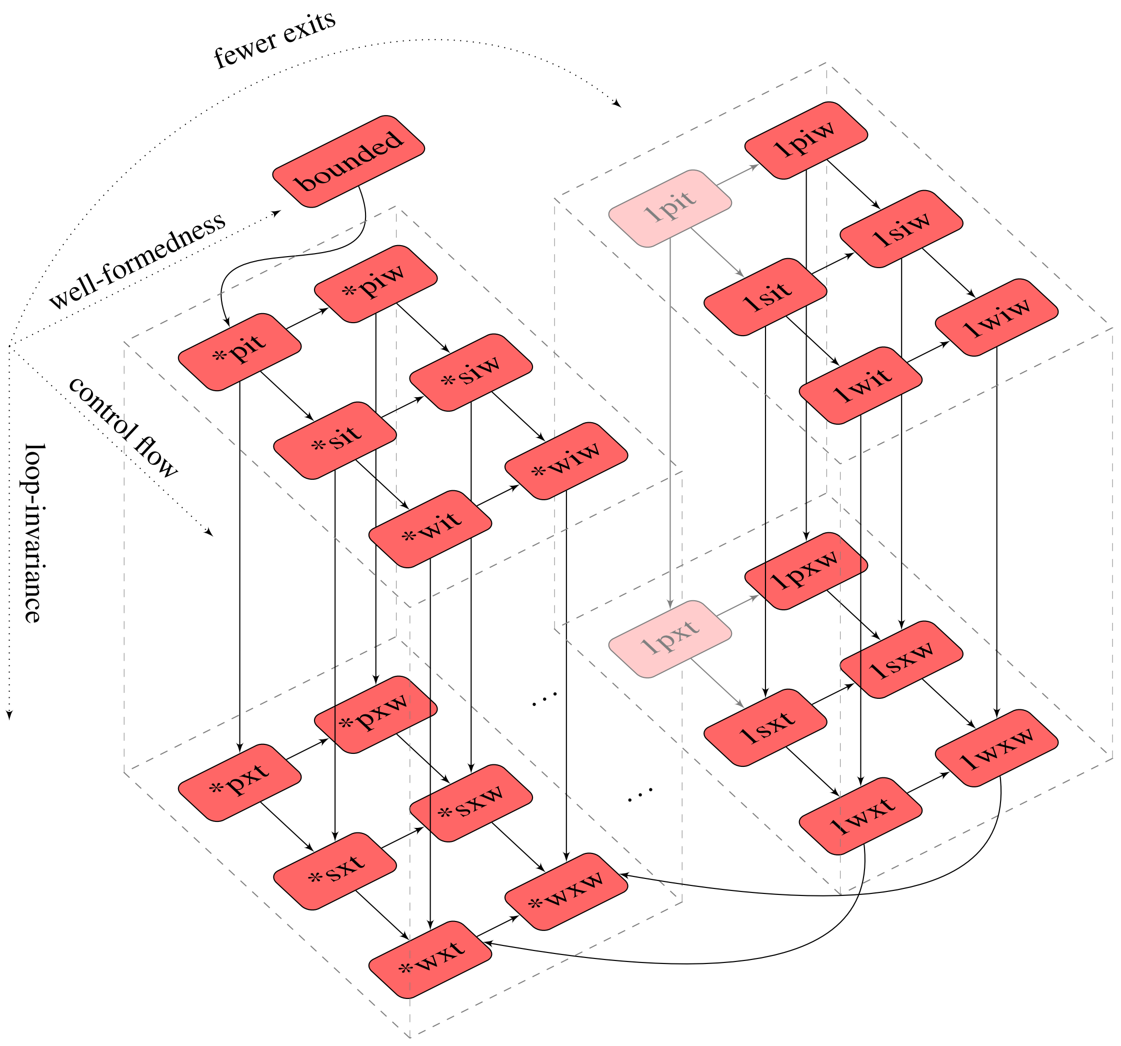


Figure 4.3: Hasse diagram of the simple loop poset.

4.3 Loop Counters

Analogously to Section 3.7, we call any variable i in an exit predicate $P(i)$ of a natural loop L that contributes to the loop's classification as simple a *loop counter*. Again, to obtain the set $LoopCounters(L)$ we need to compute all possible combinations of exit predicates that yield a classification as simple loop.

4.4 Nested Loops and Bound Computation

So far, our focus was strongly guided by termination analysis. In this section, we will extend our notion of simple loops to accommodate requirements from bound analysis and describe classes of loops with non-trivial symbolic bounds. Computing such amortized bounds on the total number of iterations of nested loops instead of merely stating the number of iterations per iteration of the immediate outer loop is a current challenge in program analysis [29]. Our motivation is to investigate if loops with non-trivial amortized bounds occur significantly often in practice and should thus be examined by research on bound computation. We will also introduce patterns based on simple loops to describe three classes of loops whose amortized bounds take a specific form. The latter provides feedback for the quality of amortized bound computation techniques, as automated evaluation of a bound's precision is itself a challenging problem [29].

Influencing and Influenced Loops

In the presence of nested loops, data dependencies between the inner and the outer loop can influence the control flow of either loop.

Example. Let us first illustrate a loop without such data dependencies: Listing 4.5 shows two nested loops, label L1 is executed $N \cdot M$ times, i.e. due to the absence of data dependencies we can trivially compute the bound as product of the respective bounds. In Listing 4.6 however, the inner loop modifies the outer loop's counter and we cannot simply compute a product-based bound.

Listing 4.5 Nested loops without data dependencies.

```
for (int i = 0; i < N; i++) {         // L_o
    for (int j = 0; j < M; j++) {   // L_i
L1:
    }
}
```

It is thus interesting to investigate the occurrence of such data dependencies between nested loops. We use program slicing (c.f. Section 2.3) to define a binary relation *influences* and two respective loop classes capturing this behavior of nested loops:

Listing 4.6 Nested loops with data dependencies influencing the loop bound.

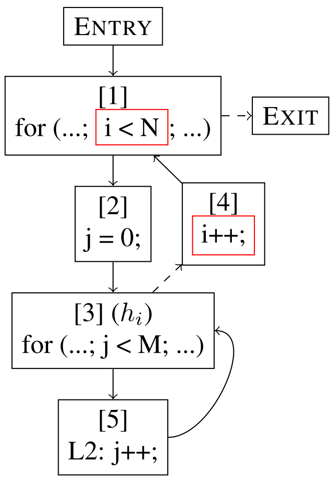
```
for (int i = 0; i < N; i++) {        // L_o
    for (int j = 0; j < M; j++) {   // L_i
        if (nondet()) {
            i++; // modifies the outer loop counter
        }
L2:
    }
}
```

Definition 25 (Influences relation, influencing loop, influenced loop)**.** Let L_i, L_o be natural loops, where L_i is entirely contained (nested) in L_o. Let L'_o be the termination slice obtained by slicing L_o with respect to slicing criterion C_{term} from Section 2.3. L_i *influences* L_o if and only if the inner loop's header h_i remains in the slice, i.e. $h_i \in L'_o$. Given a natural loop I, I is an *influencing loop* $I \in \mathcal{L}^{influencing}$ if and only if there exists an outer loop $O \supset I$ such that I influences O. Given a natural loop O, O is an *influenced loop* $O \in \mathcal{L}^{influenced}$ if and only if there exists an inner loop $I \subset O$ such that I influences O.

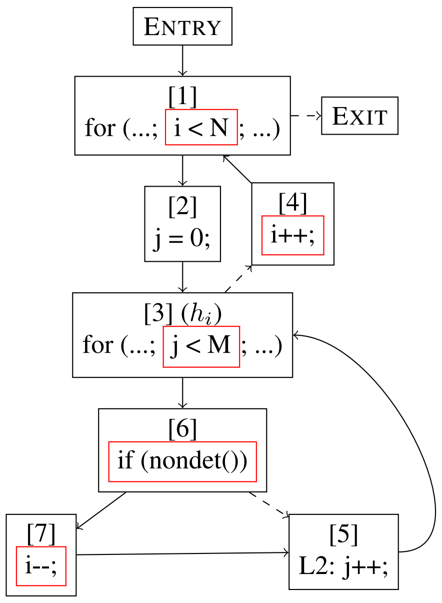
Example. Referring back to our example in Listings 4.5 to 4.6, we illustrate the classification as *influencing* and *influenced* loop: Figures 4.4a to 4.4b show the control flow graphs C_1 and C_2 of the corresponding natural loops.

The slicing of C_1 is done in a single iteration: Exit block [1] is added to the slice by default as it is specified in the slicing criterion. Because of the slicing criterion C_{term}, `i` and `N` are relevant at block [1], and – as they are never defined without being referenced as well – thus in each block of the CFG. The increment in block [4] writes to the relevant variable `i` and is also included in the slice. The newly added block [4] is only control-dependent on block [1] which is already included, thus we have reached a fixed point. We can see that the inner loop's header h_i (block [3]) is not included in the slice. Thus, this pair of inner and outer loop is not in the $influences$ relation.

We proceed by describing the slicing of C_2: Analogously to before, exit block [1] is added to the slice by default, and `i`, `N` are relevant in each block of the CFG. However, when including statements defining relevant variables, not only block [4] is included in the slice, but also block [7], which decrements `i`. Block [7] is control-dependent on block [6], which is itself control-dependent on block [3] – both are included in the slice in the following iterations. We can see that due to the increment of a relevant variable `i` inside the inner loop, this loop's header h_i (block [3]) is now in the slice. Thus, by Definition 25 we have that the inner loop (of backedge $[5] \to [3]$) *influences* the outer loop (of backedge $[4] \to [1]$). Consequently, the inner loop is an *influencing loop* and the outer loop an *influenced loop*.



(a) CFG C_1 of the loop in Listing 4.5.



(b) CFG C_2 of the loop in Listing 4.6.

Figure 4.4: Slicing for influencing and influenced loops.

Amortized Loops

While influencing and influenced loops are a useful concept by themselves, we will now describe three classes of loops whose structure induces non-trivial amortized bounds:

Non-resetting Loops (Amortized Type A)

Non-resetting loops are inner loops whose counter is never reset outside of the loop. We distinguish two different subtypes, and start by giving an example of the first:

Listing 4.7 A non-resetting nested loop (amortized type A1).

```
int i = 0,
    j = 0;
for (; i < N; i++) {                       // L_o
    for (; j < M && nondet(); j++) {   // L_i
LA1:
    }
}
```

Example. Consider the nested loops shown in Listing 4.7. As the inner loop L_i's counter `j` is never reset on the outer loop L_o, it is executed at most M times in total. An imprecise analysis may report $N \cdot M$ for the number of executions of label `LA1`. However, the M iterations of L_i

are not performed for each iteration of the outer loop L_o, but there are a total of M iterations. The challenge for automated bound computations is to come up with the much tighter bound M for the number of executions of label `LA1`.

We define a class of such loops:

Definition 26 (Amortized type A1 loops)**.** Given two properly nested loops $L_i \subset L_o$, the inner loop is an amortized loop of type A1 $L_i \in \mathcal{L}^{AmortA1}$ if and only if a variable $c \in LoopCounters(L_i)$ is never written by any statement in $L_o \setminus L_i$.

Further complicating the previous case, the inner loop's counter `j` may be written on the outer loop, but not reset, i.e. the definition of `j` also references `j`. We define these loops as type A2 and give an example:

Definition 27 (Amortized type A2 loops)**.** Given two properly nested loops $L_i \subset L_o$, the inner loop is an amortized loop of type A2 $L_i \in \mathcal{L}^{AmortA2}$ if and only if any write to a variable $c \in LoopCounters(L_i)$ occurring in a statement in $L_o \setminus L_i$ also references c.

Listing 4.8 A non-resetting nested loop (amortized type A2).

```
1  int i = 0,
2      j = 0;
3  for (; i < N; i++) {                        // L_o
4      for (; j < M && nondet(); j++) {        // L_i
5  LA2:
6      }
7      j = j - 1;
8  }
```

Example. Listing 4.8 shows a loop of type A2: contrary to the example for A1, `j` is written on the outer loop (on line 7). However, it is not reset to a constant, but merely decremented. We know that the decrement is performed N times, thus loop L_i is executed at most $N + M$ times in total.

Skipping Loops (Amortized Type B)

Another kind of amortized loop is characterized by one loop's counter skipping ahead (being incremented) by the other loop's bound. We illustrate this behavior with an example:

Example. Listing 4.9 shows an amortized loop of type B: The inner loop L_i's counter `j` iterates over all M integers in $[0, M-1]$. The outer loop L_o iterates its counter `i` from 0 to N, but importantly `i` is incremented by M in each iteration. Thus, label `LB` is executed M times for each iteration of L_o, and L_o performs $\frac{N}{M}$ iterations. For the bound we have $\frac{N}{M} \cdot M = N$, instead of the naive $N \cdot M$.

Listing 4.9 A skipping loop (amortized type B).

```
for (int i = 0; i < N; i += M) {     // L_o
    for (int j = 0; j < M; j++) {   // L_i
LB:
    }
}
```

We give a formal definition of this class of loops:

Definition 28 (Amortized type B loops)**.** Given two properly nested loops $L_i \subset L_o$, the outer loop L_o is an amortized loop of type B $L_o \in \mathcal{L}^{AmortB}$ if and only if one of the loops $L' \in \{L_i, L_o\}$ increments one of its counters $c' \in LoopCounters(L')$ by a value $\delta \in AccIncs_{L'}(c')$, such that δ occurs in an exit predicate P of the other loop $L'' \in \{L_i, L_o\} \setminus \{L'\}$, and P takes the form $E(c'') \circ \delta$ where $E(c'')$ is a linear expression in one of L'''s counters $c'' \in LoopCounters(L'')$ and $\circ \in \{<, \leq, \geq, >\}$.

Implementation

We use *simple loops* and the definition of *loop counters* previously introduced in this chapter to implement the classification of amortized loops. While any simple loop class can be used to compute counters, we use the weakest one $\left(\mathcal{L}^{simple(\text{*wxw})}\right)$, in order not to miss potential amortized loops because of the stronger constraints of other simple loop classes.

CHAPTER 5

Experimental Results

In this chapter, we introduce SLOOPY, which implements the analyses presented in the previous chapters. SLOOPY was implemented on top of CLANG, a C language frontend for the LLVM compiler infrastructure. We first introduce our benchmark setup in Section 5.1, the remaining sections contain data obtained during empiric evaluation.

5.1 Benchmark Setup

We consider four benchmarks from different areas for benchmarking:

cBench [18] Collective Benchmark (cBench) is a collection of open-source sequential programs, assembled for use in program and compiler optimization.

GNU Core Utilities [25] The GNU Core Utilities (or coreutils) are a collection of basic userland utilities for the GNU operating system.

SPEC CPU2006 [31] SPEC CPU is a suite of compute-intensive benchmarks composed from real life applications code. It is used in a wide range of industrial applications and for regression testing of LLVM. SPEC contains benchmarks in a range of programming languages – we only consider those written in C.

Mälardalen WCET [30] The Mälardalen WCET benchmarks are used in the area of Worst-Case Execution Time (WCET) analysis for experimental evaluation.

5.2 Structural Measures

Natural Loops

The following table shows the total number of natural loops $\mathcal{L}$ for each of the benchmarks. Additionally, in some benchmarks we will refer to results obtained from LOOPUS. As LOOPUS runs several LLVM optimization passes (e.g. dead code elimination) before its analysis, the reported loops may be a subset $\mathcal{L}^{\text{LOOPUS}} \subseteq \mathcal{L}$.

	cBench	coreutils	SPEC	WCET
$\mathcal{L}$	4250	1027	15 437	262
$\mathcal{L}^{\text{LOOPUS}}$	4157	1002	15 043	262

Number of Exits

We report the number of exits from loops in our benchmark array: Table 5.1 contains statistical parameters, Figures 5.1 to 5.4 illustrate the number of exits.

As proposed in Chapter 1 for benchmark metrics, we discuss the benchmark parameters:

Discussion

As we can observe in Figures 5.1 to 5.4, the number of exits follows a exponential-like distribution. The median is uniformly at $\tilde{x} = 1$ for all benchmarks, and the arithmetic mean ranges from $\bar{x} = 1.1$ (WCET) to $\bar{x} = 1.9$ (coreutils). While the standard deviation seems at an expected value for cBench and coreutils ($s \approx 1.5$), it is clearly rather small for WCET ($s = 0.2$) and rather big for SPEC ($s = 3.9$). The result for WCET is plausible, because all programs in WCET are single path programs and many exits along a single path only make sense to enhance readability or to avoid expensive computation. In fact, all loops in WCET have at most 3 exits. The standard deviation for SPEC is caused by outliers which occur in lexical analysis code in the `perlbench` and `gcc` sub-benchmarks. For this use case, a main loop with many exits returning tokens is usual, thus our results are plausible.

Finally, we also obtained results from LOOPUS, which are also shown in Figures 5.1 to 5.4: The green portion of the bar indicates loops that can be bounded by LOOPUS, while for the red portion of loops it fails to compute. Additionally, we compute the ratio of bounded loops to the total number of loops in each respective subset of the partition, which we refer to as *Loopus performance* and which is plotted in blue. Clearly, LOOPUS is more successful in computing bounds for loops with fewer exits, than for loops with many exits. At the same time, Loopus performance declines slower than the number of exits distribution, and LOOPUS still handles a non-negligible amount of loops with many exits. coreutils contains a single loop with zero exits (i.e., an infinite loop). This loop is in the `yes` program that repeatedly outputs a string until killed.

Slice-based: Number of basic blocks

Next, we discuss the slice-based metrics we described in Section 2.4: We compute a program's termination-slice and count the number of basic blocks remaining in the slice, the maximal nesting depth of basic blocks in the slice, and the number of control variables considered during slicing, i.e. variables directly or indirectly influencing termination.

Benchmark	Number of exits		
	Median $\tilde{x}$	Mean $\bar{x}$	Standard deviation s
cBench	1.0	1.7	1.6
coreutils	1.0	1.9	1.5
SPEC	1.0	1.5	3.9
WCET	1.0	1.1	0.2

Table 5.1: Statistical parameters for the number of exits.

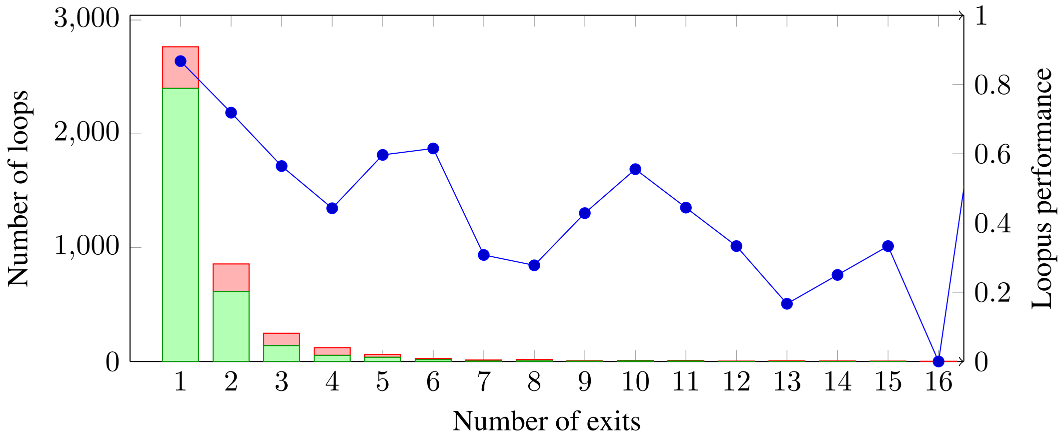


Figure 5.1: Terminating (green) and non-terminating (red) loops in cBench as determined by LOOPUS grouped by their number of exits. The blue graph shows LOOPUS performance, i.e. the ratio of loops bounded by LOOPUS to the total number of loops, in the respective class.

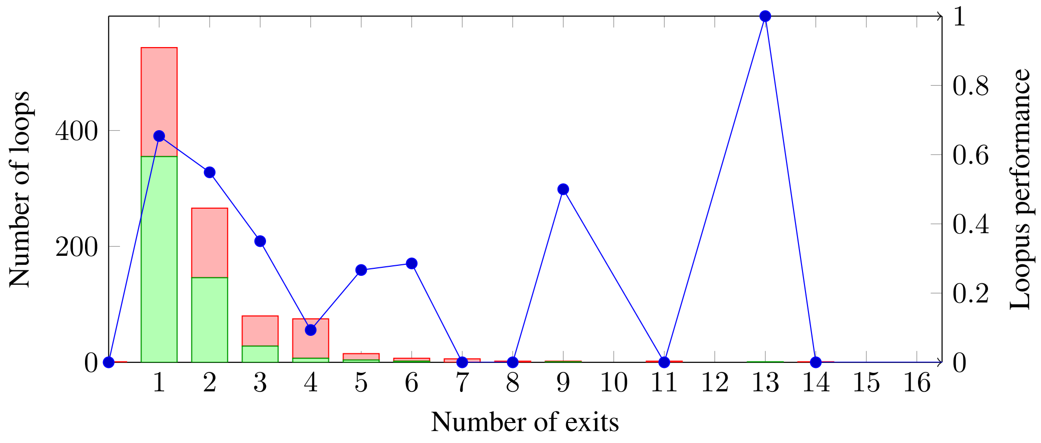


Figure 5.2: Loops in coreutils grouped by their number of exits.

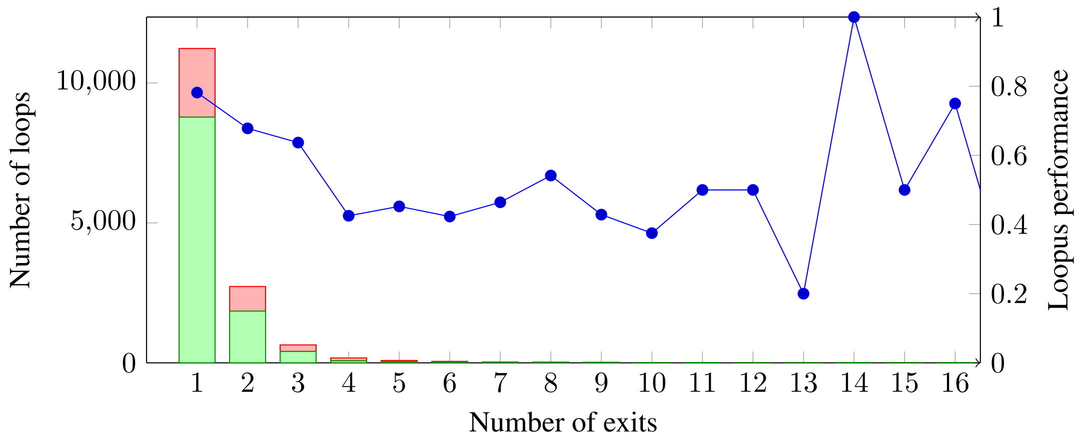


Figure 5.3: Loops in SPEC grouped by their number of exits.

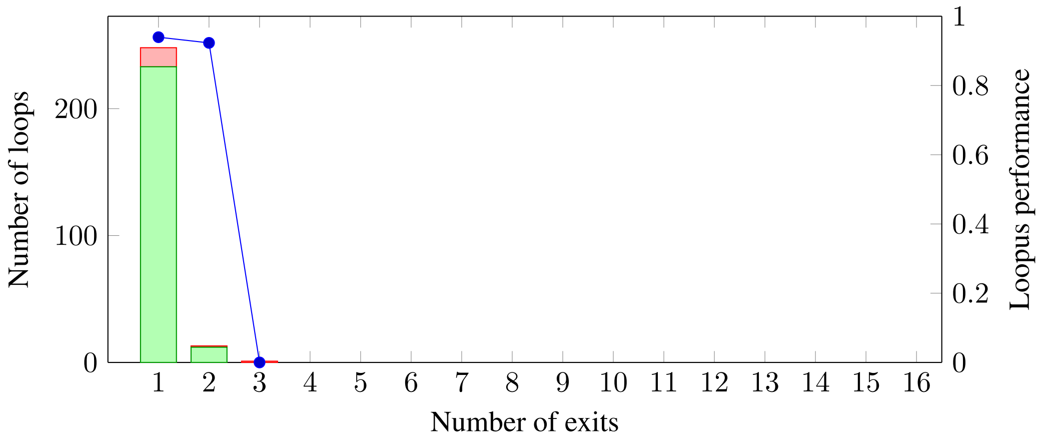


Figure 5.4: Loops in WCET grouped by their number of exits.

We first consider the number of basic blocks: The statistical parameters given in Table 5.2 show a median of $\tilde{x} = 2$ or $\tilde{x} = 3$ (for SPEC), and a mean between $\bar{x} = 3.2$ and $\bar{x} = 6.1$. The median of two basic blocks intuitively makes sense – these blocks directly correspond to the loop's header and body. The standard deviation is remarkably high for both cBench and SPEC, indicating outliers. These outliers occur across various sub-benchmarks, and we have not found an intuitive explanation.

As Figures 5.5 to 5.8 show, the number of basic blocks follows a gamma-like distribution. Loopus performance does not seem to correlate with the number of basic blocks, except for zero basic blocks. In the latter case, the termination slice degenerates into a single block with a self-loop, indicating either an infinite loop or a condition expression not containing any variables (e.g. a parameter-less function call). This is consistent with the significantly decreased Loopus performance we see for this class: either the loop really is an infinite loop, or the function call is an external call (e.g. a system call) unmodeled by LOOPUS. For WCET, due to the small

Benchmark	Number of basic blocks		
	Median $\tilde{x}$	Mean $\bar{x}$	Standard deviation s
cBench	2.0	5.7	13.0
coreutils	2.0	4.9	7.4
SPEC	3.0	6.1	19.1
WCET	2.0	3.2	3.6

Table 5.2: Statistical parameters for the number of basic blocks.

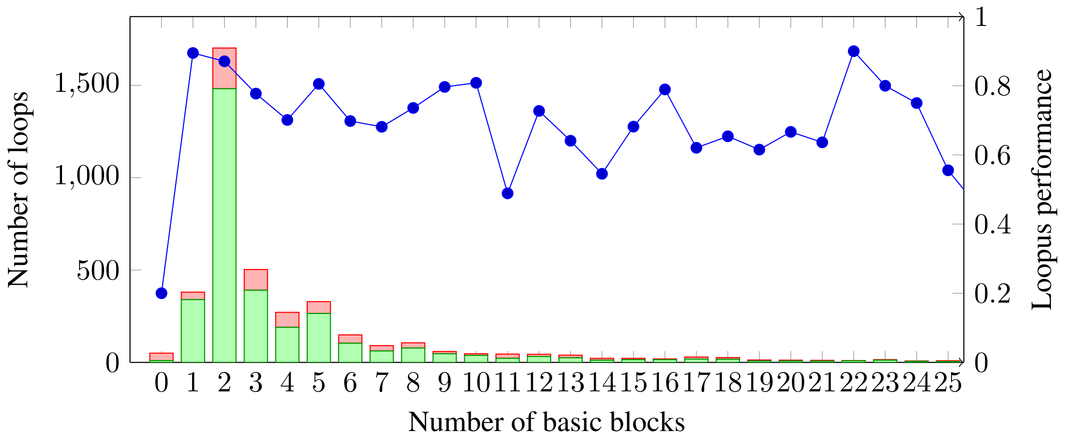


Figure 5.5: Loops in cBench grouped by their number of basic blocks.

number of loops in other classes, we must consider any number of basic blocks other than two as outliers.

Slice-based: Nesting depth

Next, we consider the maximal nesting depth of a termination-sliced program (figures in Table 5.3): The median lies at $\tilde{x} = 2$ for SPEC and $\tilde{x} = 1$ for all other benchmarks. The mean is slightly higher than this, ranging from $\bar{x} = 1.7$ for WCET to $\bar{x} = 3.1$ for SPEC. Standard deviation is about $s \approx 4$, except for WCET where $s = 1.8$. Given a low mean $\bar{x}$ and a low standard deviation s, we can assume that there are few considerably nested loops in WCET, which is empirically consistent with the fact that WCET only contains single-path programs.

Figures 5.9 to 5.12 show the distribution of the maximal nesting depth metric: Again, it follows a gamma-like distribution. Loopus performance slightly decreases with increasing nesting depth, but not as strongly as the nesting depth itself, and also less strongly than we have seen with the number of exits.

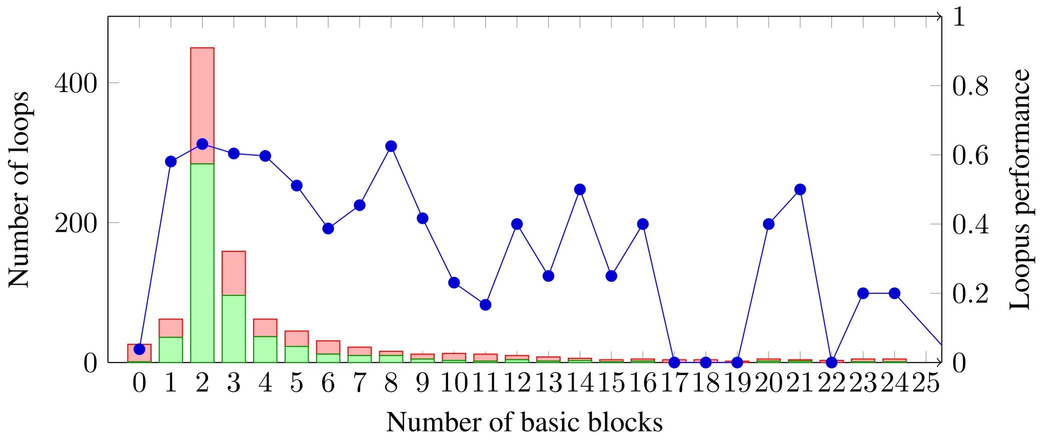


Figure 5.6: Loops in coreutils grouped by their number of basic blocks.

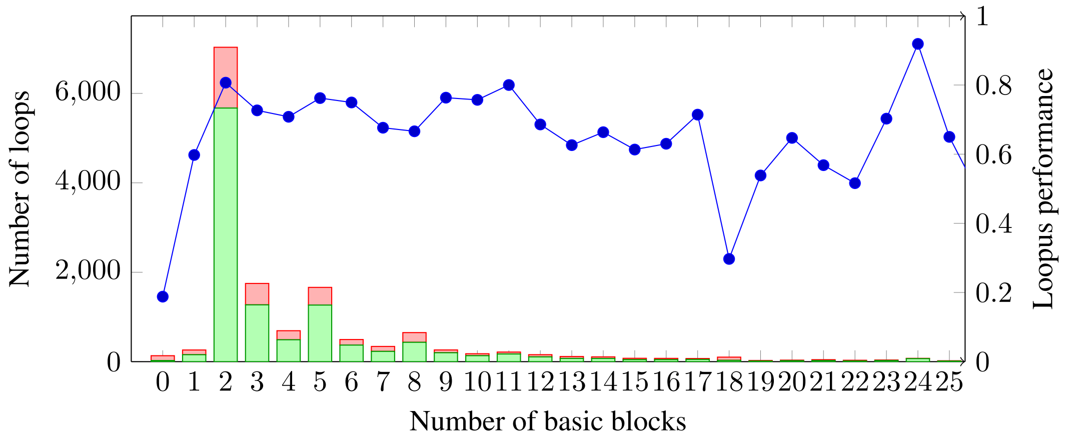


Figure 5.7: Loops in SPEC grouped by their number of basic blocks.

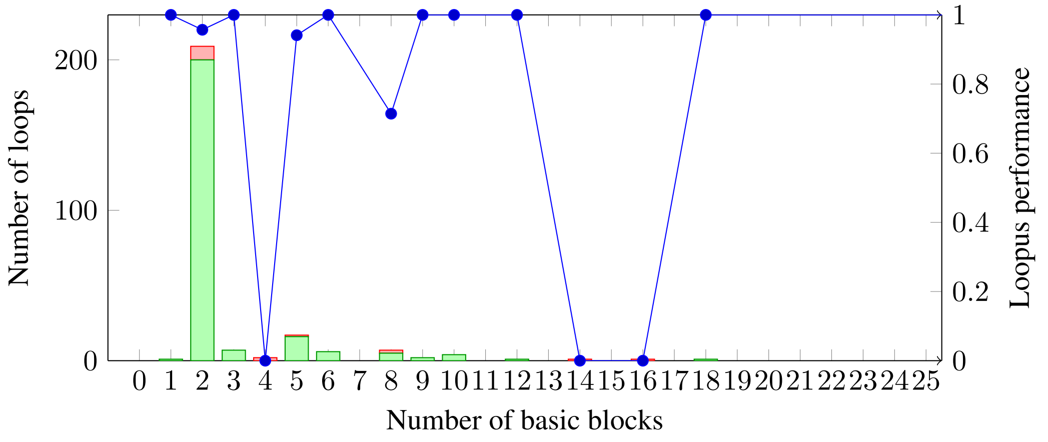


Figure 5.8: Loops in WCET grouped by their number of basic blocks.

Benchmark	Maximal nesting depth		
	Median $\tilde{x}$	Mean $\bar{x}$	Standard deviation s
cBench	1.0	2.9	4.0
coreutils	1.0	2.6	3.4
SPEC	2.0	3.1	4.1
WCET	1.0	1.7	1.8

Table 5.3: Statistical parameters for the maximal nesting depth.

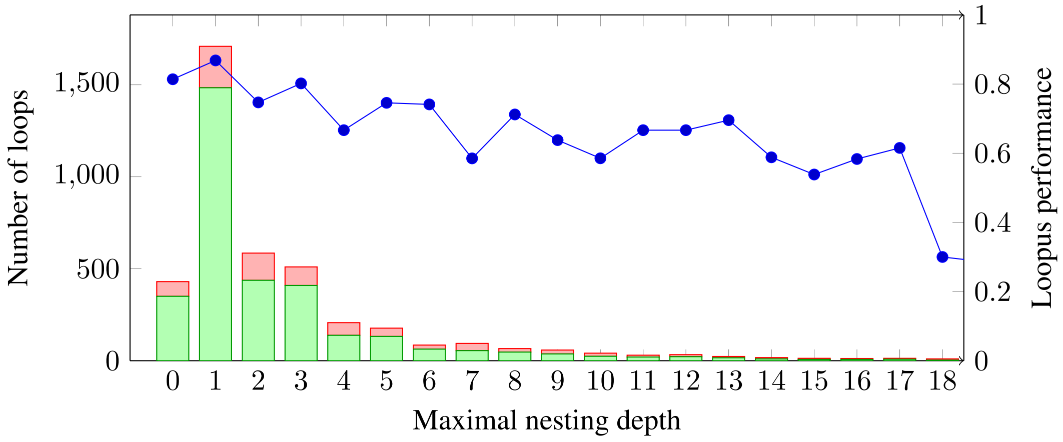


Figure 5.9: Loops in cBench grouped by their maximal nesting depth.

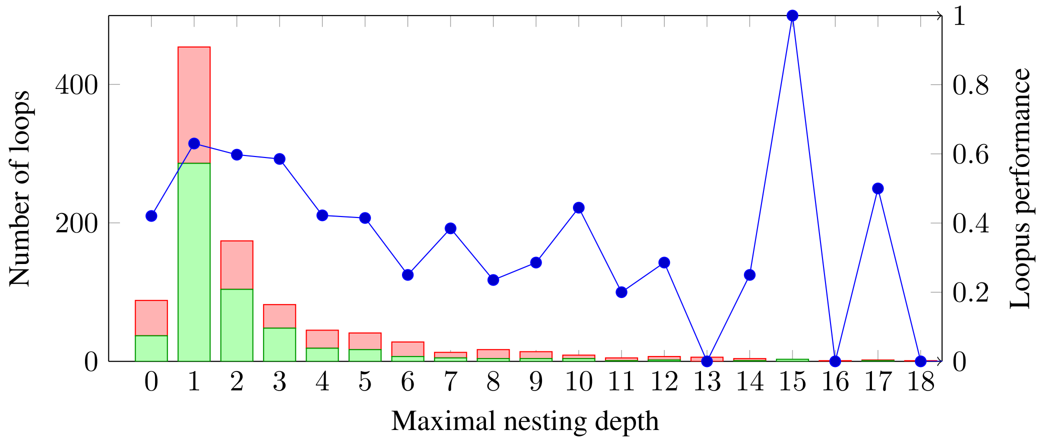


Figure 5.10: Loops in coreutils grouped by their maximal nesting depth.

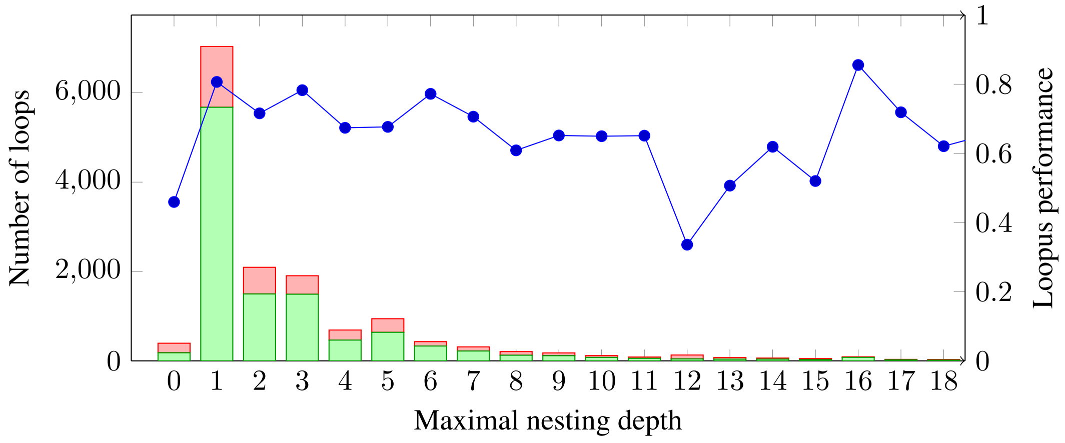


Figure 5.11: Loops in SPEC grouped by their maximal nesting depth.

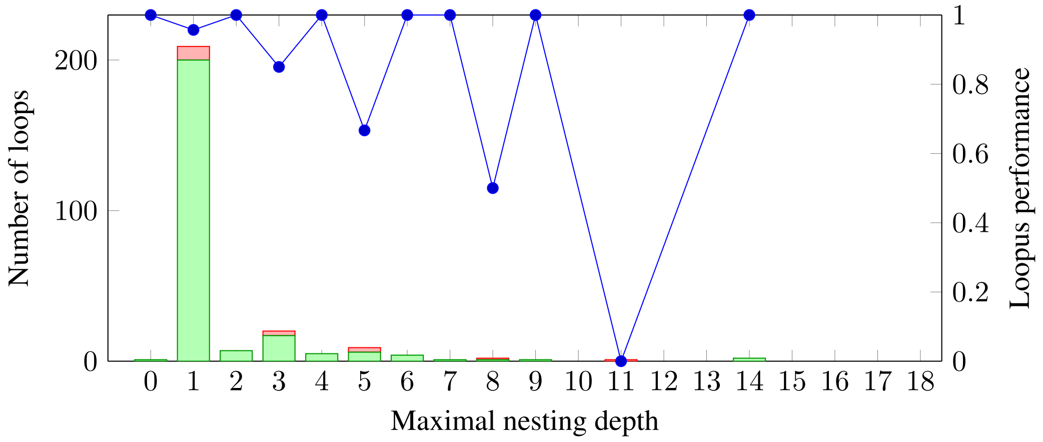


Figure 5.12: Loops in WCET grouped by their maximal nesting depth.

Slice-based: Number of control variables

Finally, we consider the number of control variables, i.e. the variables directly or indirectly influencing the loop's termination, as determined by termination slicing.

The statistical parameters summarized in Table 5.4 show a median $\tilde{x} = 2$ across all benchmarks. Again, this makes sense for condition expressions that contain two variables compared by a relational or equality operator. The arithmetic mean is between $\bar{x} = 3.3$ and $\bar{x} = 3.7$ for cBench, coreutils, and SPEC, and slightly lower at $\bar{x} = 2.1$ for WCET. Similarly, the standard deviation ranges between $s = 4.1$ and $s = 5.3$, except for WCET, where it is at $s = 1.9$.

Figures 5.13 to 5.16 again show a gamma-like distribution for this metric, sometimes degraded to an exponential-like distribution. In the comparison with LOOPUS, we see slightly higher Loopus performance on loops with less than three control variables, afterwards remaining stable at about 60% for cBench and SPEC, and rapidly dropping for coreutils and WCET.

Benchmark	Number of control variables		
	Median $\tilde{x}$	Mean $\bar{x}$	Standard deviation s
cBench	2.0	3.7	5.3
coreutils	2.0	3.6	4.1
SPEC	2.0	3.3	4.1
WCET	2.0	2.1	1.9

Table 5.4: Statistical parameters for the number of control variables.

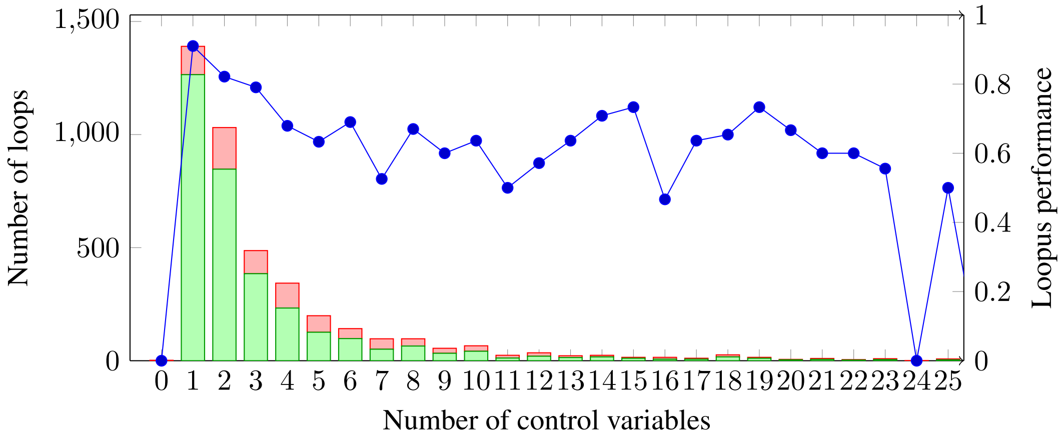


Figure 5.13: Loops in cBench grouped by their number of control variables.

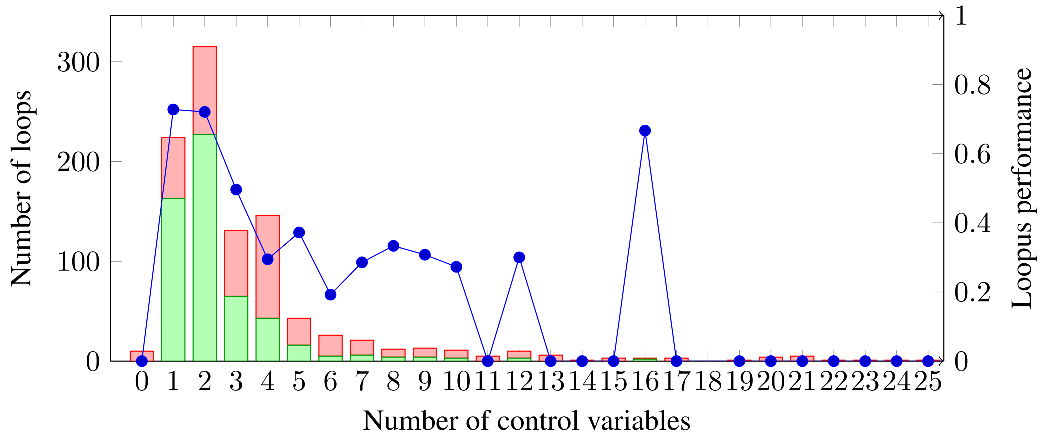


Figure 5.14: Loops in coreutils grouped by their number of control variables.

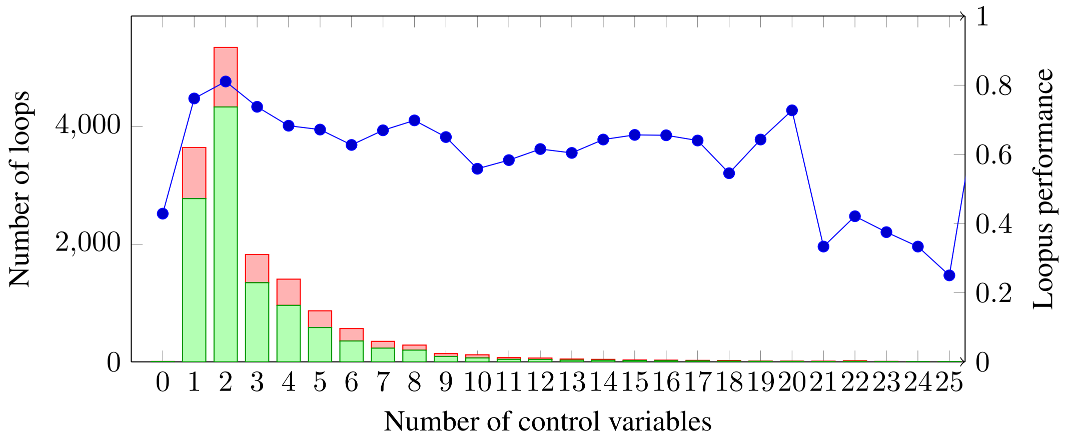


Figure 5.15: Loops in SPEC grouped by their number of control variables.

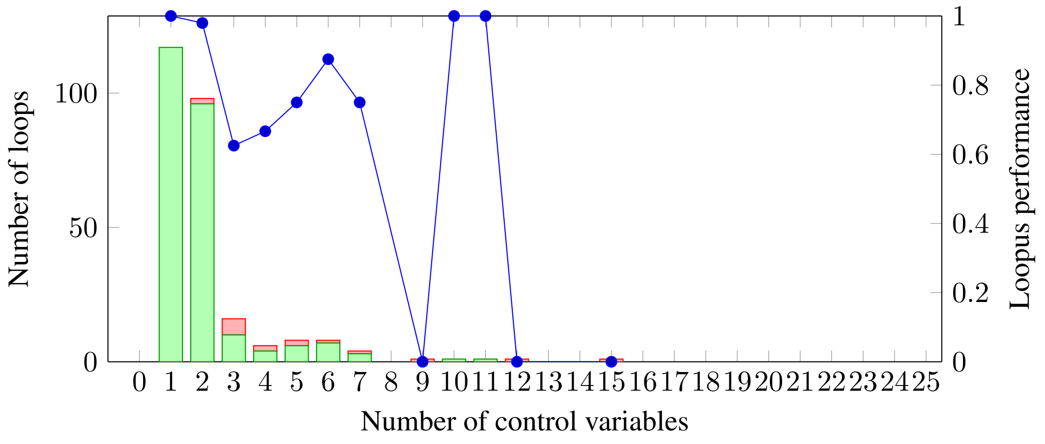


Figure 5.16: Loops in WCET grouped by their number of control variables.

5.3 Syntactically Terminating Loops & Simple Loops

After discussing structural CFG- and slice-based measures in the previous section, we will now have a look at the syntactically terminating loops introduced in Chapter 3 and the simple loops from Chapter 4. To keep the comparison working, in this section we only consider syntactically bounded loops $\mathcal{L}^{SB}$, the strongest simple loop class $\mathcal{L}^{ST} = \mathcal{L}^{simple(\text{*pit})}$, the weakest simple loop class $\mathcal{L}^{simple(\text{*wxw})}$, and all loops not in any simple loop class $\mathcal{L} \setminus \mathcal{L}^{simple(\text{*wxw})}$. These loops are especially interesting, because those in $\mathcal{L}^{SB}$ can directly be compared with LOOPUS (it should be able to bound all loops in this class), $\mathcal{L}^{simple(\text{*pit})}$ and $\mathcal{L}^{simple(\text{*wxw})}$ represent the range of simple loops, and $\mathcal{L} \setminus \mathcal{L}^{simple(\text{*wxw})}$ shows that non-simple loops are significantly more difficult than the average. A comparison of all simple loop classes based on simple loop constraint transformers is given below in Section 5.4.

Figure 5.17 shows the percentage of loops in each of the listed classes for the respective

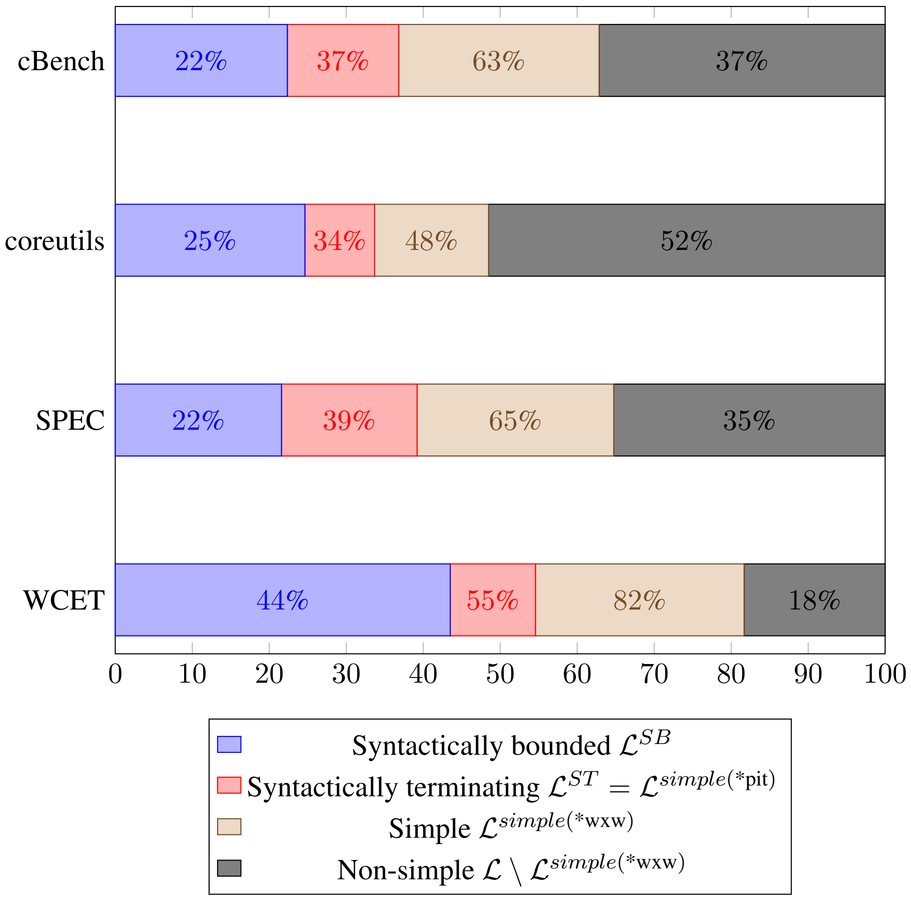


Figure 5.17: Percentage of loops covered by various loop classes in different benchmarks.

benchmark: The percentage of loops in $\mathcal{L}^{SB}$ is around 22%, except for WCET where it is at 44%. Some 10–15% more loops are only in $\mathcal{L}^{ST}$, which amounts to about 37% of all loops, except for WCET (55%). The weakest simple loop class $\mathcal{L}^{simple(\text{*wxw})}$ additionally contains 26% more loops on all benchmarks except coreutils, comprising about 65% of all loops in cBench and SPEC, and 82% in WCET. For coreutils, the number of simple loops is a bit smaller at 48% of all loops.

The percentage of loops in $\mathcal{L}^{SB}$ and $\mathcal{L}^{ST}$ is stable across all benchmarks except WCET. This can be explained by the single-path property of WCET that naturally fulfills the strong control flow constraint of syntactically terminating loops. Overall, the ratio of simple loops in WCET is consistent with the results of [58], where LOOPUS performed a lot better on WCET than on cBench, which suggests – consistent with our classification – that WCET is a less-demanding benchmark for automated program analysis.

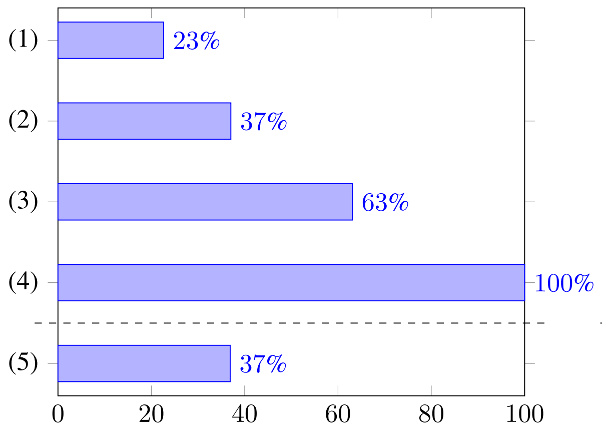
The small difference between $\mathcal{L}^{simple(\text{*wxw})}$ and $\mathcal{L}^{ST}$ in coreutils can be explained by a significant amount of loops taking the form

```
while (int i = syscall(c))
```

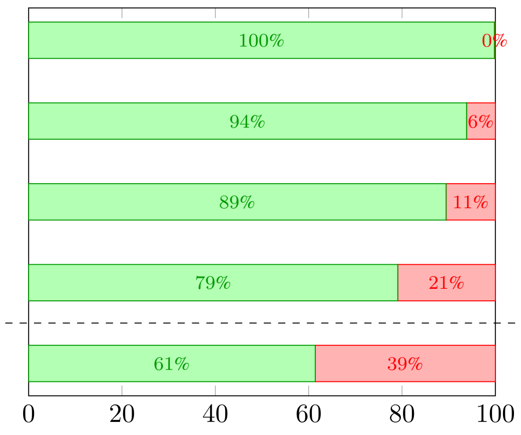
where `syscall` is a system call to the operating system, and `i` will take a value indicating success of that call.

Comparison with LOOPUS

As our central goal is to establish a relation between simple loops and the difficulty of automated program analysis, we now describe how well LOOPUS performs on our selection of loop classes: Figures 5.18a to 5.21a show the percentage of loops contained in each of these classes. Bars (1)–(4) are classes with increasingly less restrictive constraints. Bars (5) below the dashed line refer to loops not in any simple loop class, i.e. the complement of loops shown under (3). Figures 5.18b to 5.21b show the percentage of loops in the respective class for which LOOPUS succeeds and fails to compute a symbolic bound. As these are relative numbers, the percentage of positive results is equal to the Loopus performance measure used above.



(a) Percentage of loop classes (1) $\mathcal{L}^{SB}$, (2) $\mathcal{L}^{ST} = \mathcal{L}^{simple(\text{*pit})}$, (3) $\mathcal{L}^{simple(\text{*wxw})}$, (4) $\mathcal{L}$, and (5) $\mathcal{L} \setminus \mathcal{L}^{simple(\text{*wxw})}$ in the cBench benchmark suite.

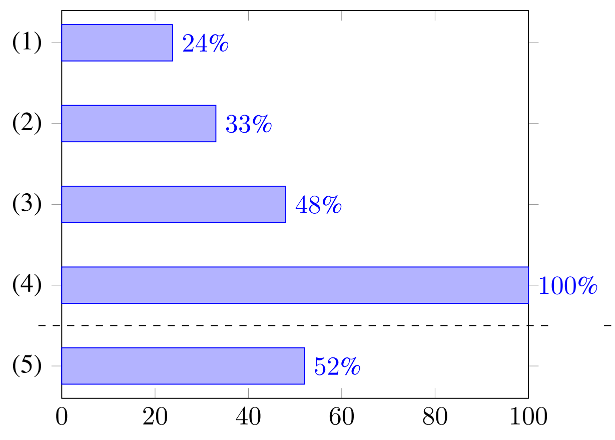


(b) Percentage of loops in the resp. class bounded (green) and not bounded (red) by LOOPUS.

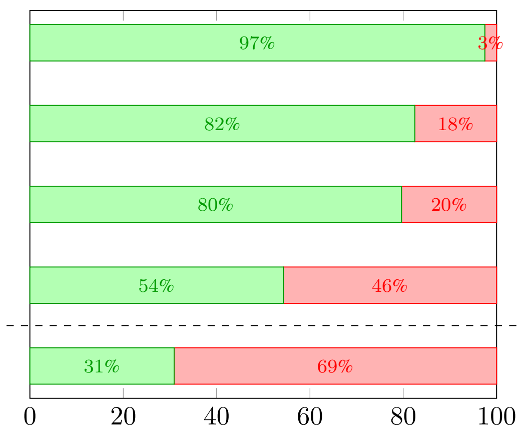
Figure 5.18: Comparison of various loop classes with LOOPUS results on cBench.

Discussion

Consistent with our expectation of simple loops, the simpler 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 difficult for an automated procedure to handle. Additionally, as expected, for any termination proof we obtain in $\mathcal{L}^{SB}$ on benchmarks that LOOPUS was optimized for (cBench and WCET), it also bounds the loop, i.e. both SLOOPY and LOOPUS agree on termination of this class. Random examination of those loops unbounded in $\mathcal{L}^{SB}$ on coreutils and SPEC suggests that this is caused by system calls unmodeled in LOOPUS, and is a good pointer for improvement. As a positive result for LOOPUS, while it performs worst on non-simple loops, it is still successful on about two thirds of these loops, except for coreutils which seems a much harder benchmark.

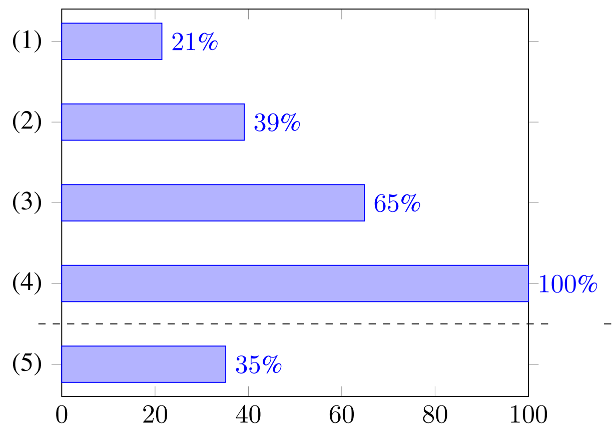


(a) Percentage of loop classes (1) $\mathcal{L}^{SB}$, (2) $\mathcal{L}^{ST} = \mathcal{L}^{simple(\text{*pit})}$, (3) $\mathcal{L}^{simple(\text{*wxw})}$, (4) $\mathcal{L}$, and (5) $\mathcal{L} \setminus \mathcal{L}^{simple(\text{*wxw})}$ in the coreutils benchmark suite.

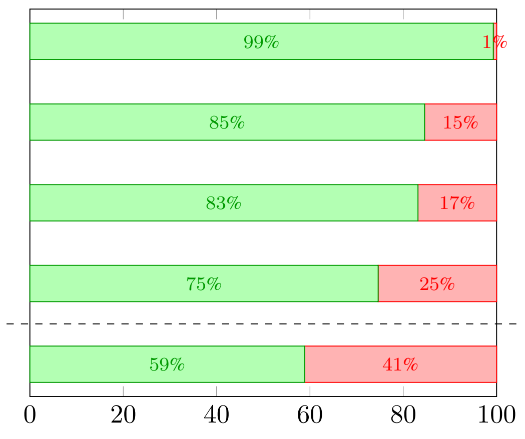


(b) Percentage of loops in the resp. class bounded (green) and not bounded (red) by LOOPUS.

Figure 5.19: Comparison of various loop classes with LOOPUS results on coreutils.



(a) Percentage of loop classes (1) $\mathcal{L}^{SB}$, (2) $\mathcal{L}^{ST} = \mathcal{L}^{simple(\text{*pit})}$, (3) $\mathcal{L}^{simple(\text{*wxw})}$, (4) $\mathcal{L}$, and (5) $\mathcal{L} \setminus \mathcal{L}^{simple(\text{*wxw})}$ in the SPEC benchmark suite.



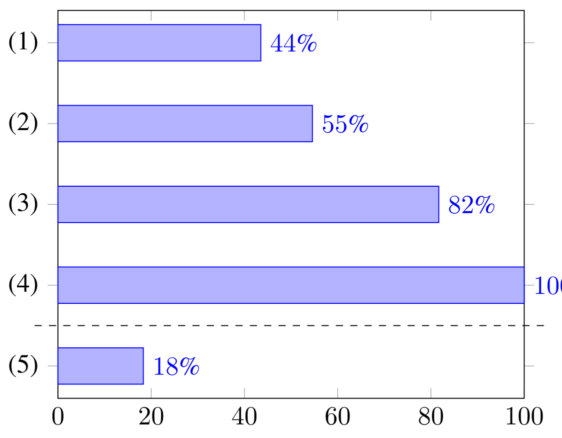
(b) Percentage of loops in the resp. class bounded (green) and not bounded (red) by LOOPUS.

Figure 5.20: Comparison of various loop classes with LOOPUS results on SPEC.

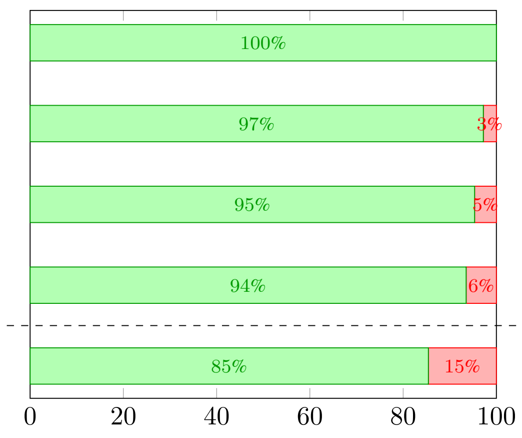
5.4 Effectivity of Simple Loop Constraint Transformers

Up until now, we have only considered the strongest and weakest simple loops classes $\mathcal{L}^{ST} = \mathcal{L}^{simple(\text{*pit})}$ and $\mathcal{L}^{simple(\text{*wxw})}$. To evaluate the usefulness of each simple loop class, we now study the *effectivity* of simple loop constraint transformers (SLCTs):

Definition 29 (Effectivity of simple loop constraint transformers)**.** Given an SLCT t, its effectivity E_t on a constraint C_1 is the relative increase of loops covered by the obtained constraint



(a) Percentage of loop classes (1) $\mathcal{L}^{SB}$, (2) $\mathcal{L}^{ST} = \mathcal{L}^{simple(\text{*pit})}$, (3) $\mathcal{L}^{simple(\text{*wxw})}$, (4) $\mathcal{L}$, and (5) $\mathcal{L} \setminus \mathcal{L}^{simple(\text{*wxw})}$ in the WCET benchmark suite.



(b) Percentage of loops in the resp. class bounded (green) and not bounded (red) by LOOPUS.

Figure 5.21: Comparison of various loop classes with LOOPUS results on WCET.

$C_2 = t(C_1)$, i.e.

$$E_t = \frac{|\mathcal{L}^{simple(C_2)}| - |\mathcal{L}^{simple(C_1)}|}{|\mathcal{L}|} \qquad \text{where } C_2 = t(C_1)$$

We tabulate the effectivity of each SLCT t in Tables 5.5 to 5.7: The first row for each benchmark lists the percentage of simple loops covered by the stronger constraint C_1 given in the respective header column. The second row for each benchmark lists the percentage of simple loops covered by the weaker constraint C_2 obtained by applying t to C_1. The third row shows the difference between these two percentages, i.e. the effectivity E_t.

We define a significance level of $E_t \geq 1.5\%$. From our experience, this value serves as a clean cutoff point between transformers that include a considerable amount of additional loops and those that do not. For transformer applications that fulfill this significance criterion, the value of E_t is set in bold in Tables 5.5 to 5.7. In summary, we see that for

Proved control flow $\rightarrow$ strong control flow (cf_1) This transformer's effectivity is only significant on cBench for all four of the possible applications, and on SPEC for two of the four applications. Effectivity on well-formed loop classes ($W =$ w) is slightly higher than on terminating loop classes ($W =$ t).

Strong control flow $\rightarrow$ weak control flow (cf_2) Effectivity of the cf_2 transformer is significant on all benchmarks, and with the exception of coreutils even for a significance level of $E_t \geq 8.8$. Again, effectivity is slightly higher on well-formed loop classes. Effectivity on coreutils is clearly below all other benchmarks.

Checking loop-invariance $\rightarrow$ not checking loop-invariance (inv) Effectivity of this transformer is not significant on any benchmarks, with E_t usually ranging from 0 to 0.4, and three cases with slightly higher effectivity (0.8, 1.1, 1.1) on WCET.

Terminating predicate $\rightarrow$ well-formed predicate (wf) This transformer's effectivity is significant on all benchmarks. Empirically, its effectivity increases with weaker control flow constraints.

Single-exit $\rightarrow$ any-exit (ex) Effectivity of ex is as well significant on all benchmarks. Additionally, we measured higher effectivity on well-formed ($W = \mathsf{w}$) than on terminating ($T = \mathsf{t}$) loops.

Omitting the empirically insignificant transformers cf_1 and inv and replacing cf_2 with a (significant) new transformer $cf = cf_2 \circ cf_1$, we can obtain a subset of simple loop classes as illustrated in Figure 5.22.

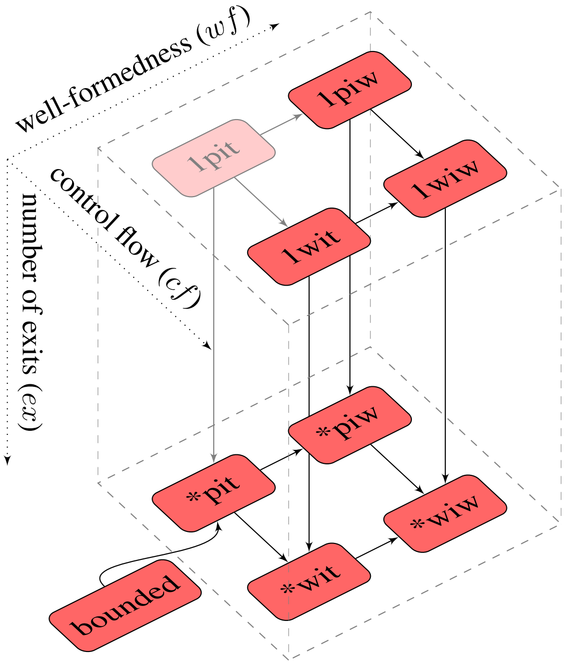


Figure 5.22: Hasse diagram of the reduced simple loop poset.

	*pit	*piw	*pxt	*pxw
cBench				
cf_1 ς	37.0	46.6	37.2	47.0
	39.3	52.4	39.5	52.8
E_t	**2.2**	**5.7**	**2.3**	**5.8**
coreutils				
cf_1 ς	33.7	40.7	33.7	40.8
	35.0	44.3	35.1	44.5
E_t	1.3	**3.6**	1.4	**3.7**
SPEC				
cf_1 ς	39.2	45.9	39.3	46.0
	40.0	47.0	40.2	47.2
E_t	0.8	1.1	0.8	1.2
WCET				
cf_1 ς	54.6	66.8	55.0	67.6
	54.6	67.2	55.0	68.3
E_t	0.0	0.4	0.0	0.8

(a) Effectivity of the cf_1 transformer.

	*sit	*siw	*sxt	*sxw
cBench				
cf_2 ς	39.3	52.4	39.5	52.8
	48.0	62.7	48.3	63.1
E_t	**8.8**	**10.3**	**8.8**	**10.3**
coreutils				
cf_2 ς	35.0	44.3	35.1	44.5
	37.5	48.3	37.6	48.5
E_t	**2.5**	**4.0**	**2.5**	**4.0**
SPEC				
cf_2 ς	40.0	47.0	40.2	47.2
	56.0	64.5	56.2	64.7
E_t	**16.0**	**17.5**	**16.1**	**17.5**
WCET				
cf_2 ς	54.6	67.2	55.0	68.3
	67.2	80.5	67.6	81.7
E_t	**12.6**	**13.4**	**12.6**	**13.4**

(b) Effectivity of the cf_2 transformer.

Table 5.5: Effectivity of the cf_1 and cf_2 transformers.

	*pit	*piw	*sit	*siw	*wit	*wiw
cBench						
$inv\ \varsigma$	37.0	46.6	39.3	52.4	48.0	62.7
	37.2	47.0	39.5	52.8	48.3	63.1
E_t	0.1	0.3	0.2	0.4	0.2	0.4
coreutils						
$inv\ \varsigma$	33.7	40.7	35.0	44.3	37.5	48.3
	33.7	40.8	35.1	44.5	37.6	48.5
E_t	0.0	0.1	0.1	0.2	0.1	0.2
SPEC						
$inv\ \varsigma$	39.2	45.9	40.0	47.0	56.0	64.5
	39.3	46.0	40.2	47.2	56.2	64.7
E_t	0.1	0.2	0.2	0.2	0.2	0.3
WCET						
$inv\ \varsigma$	54.6	66.8	54.6	67.2	67.2	80.5
	55.0	67.6	55.0	68.3	67.6	81.7
E_t	0.4	0.8	0.4	1.1	0.4	1.1

(a) Effectivity of the inv transformer.

	*pit	*pxt	*sit	*sxt	*wit	*wxt
cBench						
$wf\ \varsigma$	37.0	37.2	39.3	39.5	48.0	48.3
	46.6	47.0	52.4	52.8	62.7	63.1
E_t	**9.6**	**9.8**	**13.1**	**13.3**	**14.7**	**14.8**
coreutils						
$wf\ \varsigma$	33.7	33.7	35.0	35.1	37.5	37.6
	40.7	40.8	44.3	44.5	48.3	48.5
E_t	**7.0**	**7.1**	**9.3**	**9.4**	**10.8**	**10.9**
SPEC						
$wf\ \varsigma$	39.2	39.3	40.0	40.2	56.0	56.2
	45.9	46.0	47.0	47.2	64.5	64.7
E_t	**6.7**	**6.7**	**7.0**	**7.1**	**8.5**	**8.5**
WCET						
$wf\ \varsigma$	54.6	55.0	54.6	55.0	67.2	67.6
	66.8	67.6	67.2	68.3	80.5	81.7
E_t	**12.2**	**12.6**	**12.6**	**13.4**	**13.4**	**14.1**

(b) Effectivity of the wf transformer.

Table 5.6: Effectivity of the inv and wf transformers.

	1pit	1piw	1pxt	1pxw	1sit	1siw	1sxt	1sxw	1wit	1wiw	1wxt	1wxw
cBench												
$ex\,\varsigma$	29.1	35.3	29.2	35.6	30.2	38.9	30.3	39.2	36.8	46.2	36.9	46.5
	37.0	46.6	37.2	47.0	39.3	52.4	39.5	52.8	48.0	62.7	48.3	63.1
E_t	**7.9**	**11.3**	**8.0**	**11.4**	**9.0**	**13.5**	**9.1**	**13.6**	**11.2**	**16.6**	**11.3**	**16.6**
coreutils												
$ex\,\varsigma$	22.6	26.5	22.6	26.6	23.2	28.0	23.2	28.1	24.8	30.7	24.8	30.8
	33.7	40.7	33.7	40.8	35.0	44.3	35.1	44.5	37.5	48.3	37.6	48.5
E_t	**11.1**	**14.2**	**11.1**	**14.2**	**11.8**	**16.3**	**11.9**	**16.4**	**12.7**	**17.6**	**12.8**	**17.7**
SPEC												
$ex\,\varsigma$	31.3	35.0	31.4	35.2	31.8	35.6	31.9	35.7	45.6	50.5	45.8	50.7
	39.2	45.9	39.3	46.0	40.0	47.0	40.2	47.2	56.0	64.5	56.2	64.7
E_t	**7.9**	**10.9**	**7.9**	**10.9**	**8.2**	**11.4**	**8.2**	**11.5**	**10.4**	**14.0**	**10.4**	**14.1**
WCET												
$ex\,\varsigma$	51.9	64.1	52.3	64.9	51.9	64.1	52.3	65.3	62.2	75.2	62.6	76.3
	54.6	66.8	55.0	67.6	54.6	67.2	55.0	68.3	67.2	80.5	67.6	81.7
E_t	**2.7**	**2.7**	**2.7**	**2.7**	**2.7**	**3.1**	**2.7**	**3.1**	**5.0**	**5.3**	**5.0**	**5.3**

Table 5.7: Effectivity of the ex transformer.

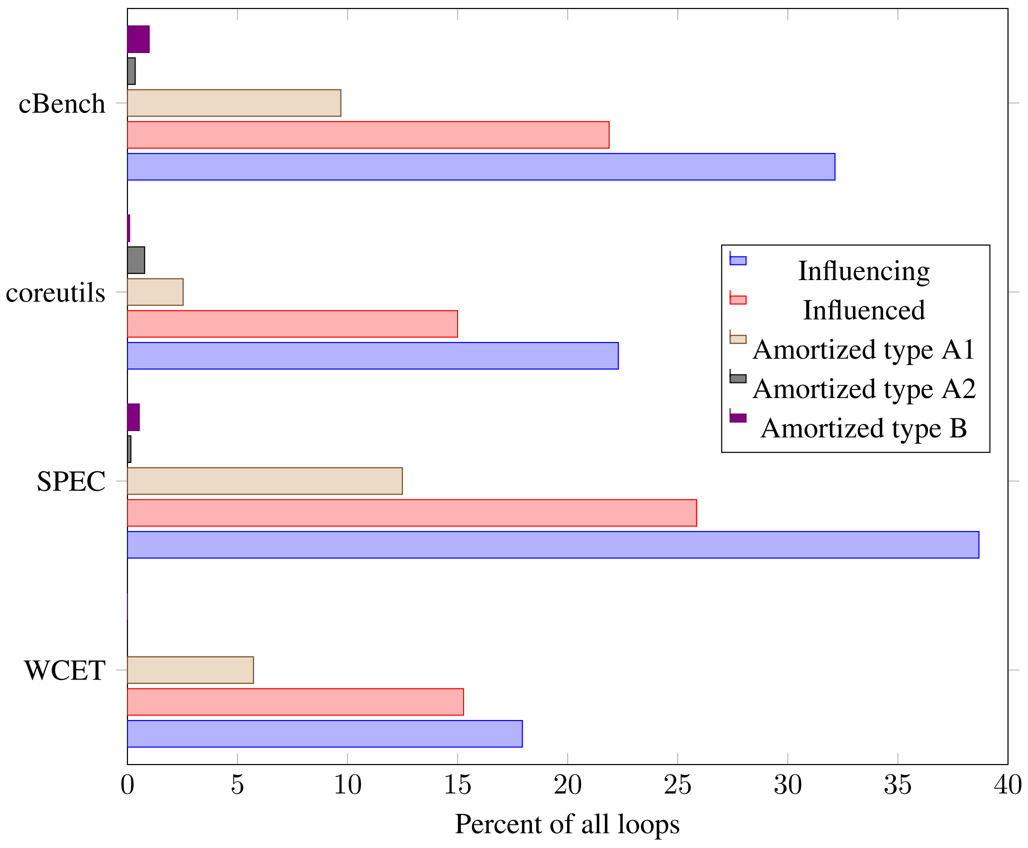


Figure 5.23: Nested loop classes.

5.5 Nested Loops

In Chapter 4 we discussed the use of simple loops to define classes of nested loops influencing each other. We also described classes of amortized loops, that may exhibit non-trivial symbolic bounds. Table 5.8 shows the number of loops found by our analysis in the respective classes, and Figure 5.23 illustrates the relative occurence of these classes: Clearly, there is a significant amount of influencing and influenced loops across all benchmarks. While amortized loops of type A2 and B are rather uncommon, amortized loops of type A1 occur with a rate of 3–12%, thus making a rewarding objective for automated bound computation.

5.6 Statement Usage

An important aspect of programming language design is which 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 any of the statements in C capable of expressing loops can be equivalently rewritten using one of the others, as illustrated below by the equivalent programs P, Q, and R:

	$\mathcal{L}$	$\mathcal{L}^{influencing}$	$\mathcal{L}^{influenced}$	$\mathcal{L}^{AmortA1}$	$\mathcal{L}^{AmortA2}$	$\mathcal{L}^{AmortB}$
cBench	4250	1366	930	412	15	42
coreutils	1027	229	154	26	8	1
SPEC	15 437	5971	3991	1928	24	83
WCET	262	47	40	15	0	0

Table 5.8: Nested loop classes.

Listing 5.1 Example program P.

```
for (int j = 0; j < off; j++) {
  p += primes_diff[i + j];
}
```

Listing 5.2 Example program Q.

```
{
  int j = 0;
  while (j < off) {
    p += primes_diff[i + j];
    j++;
  }
}
```

Listing 5.3 Example program R.

```
{
  int j = 0;
loop:
  if (! (j < off) )
    goto afterloop;
  p += primes_diff[i + j];
  j++;
  goto loop;
}
afterloop: ;
```

To map a natural loop L to a loop statement, we introduce a function *stmt* : $\mathcal{L} \rightarrow \{$FOR, WHILE, DO, GOTO, UF$\}$. If L's backedge is induced by an iteration statement (for, while, do), *stmt*(L) evaluates to FOR, WHILE, and DO, respectively. If the backedge is induced by a continue statement, *stmt*(L) evaluates to *stmt*(M), where M is the inner-most loop containing L whose backedge is induced by an iteration statement. If the backedge is induced by a goto statement, then *stmt*(L) = GOTO. In any other case, *stmt*(L) = UF (UF stands for unstructured flow).

While the former should all be clear, UF loops are rather uncommon and may require some explanation. We give an example representative of UF loops in our benchmarks in Listing 5.4: Flow of control enters the loop at a single location, i.e. the label Header. The loop's body is comprised of two function calls a() and b(), after which the the loop continues its next iteration from Header. In other words, control flow from b() to Header forms the loop's backedge, but there is no statement explicitly directing flow of control. Rather, the backedge is caused by normal order of execution of a compound statement.

Listing 5.4 A `UF` loop.

```
  goto Header;
A:
  b();
Header:
  a();
  goto A;
```

	FOR		WHILE		DO		GOTO		UF	
	n	%	n	%	n	%	n	%	n	%
cBench	2700	63.5	1195	28.1	269	6.3	85	2.0	1	0.0
coreutils	470	45.8	482	46.9	73	7.1	2	0.2	0	0.0
SPEC	12 594	81.6	2508	16.2	256	1.7	75	0.5	4	0.0
WCET	232	88.5	27	10.3	2	0.8	1	0.4	0	0.0

Table 5.9: C statements used to express loops.

Discussion

Table 5.9 and Figure 5.24 show the occurence of statements determined by $stmt$ in our selection of loop classes: Overall, the `for` statement is used a lot more than other statements, even though the number varies greatly between more than 80% of all loops $\mathcal{L}$ (row (4) in the table) for SPEC and WCET and less than 50% of $\mathcal{L}$ for coreutils. The next-most used statement is `while`, with an equally wide range from $\approx 10\%$ on WCET to $\approx 47\%$ 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. Loops with unstructured flow (`UF`) are extremely rare: There are only five specimen in total, one in cBench and four in SPEC.

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 occurence of `FOR` in $\mathcal{L}$, the simpler the loop class, the higher the percentage of loops expressed using a `for` statement. At the same time, the less simple a loop, the higher the percentage of `while`, `do`, `goto` statements, and `UF` loops. This correlation is especially strong for `do` and `goto` statements, where virtually no such loops are in $\mathcal{L}^{ST}$. Unstructured flow loops seem even less simple, although our sample size is extremely small: all occurances of `UF` loops are non-simple, i.e. in $\mathcal{L} \setminus \mathcal{L}^{simple(\text{*wxw})}$.

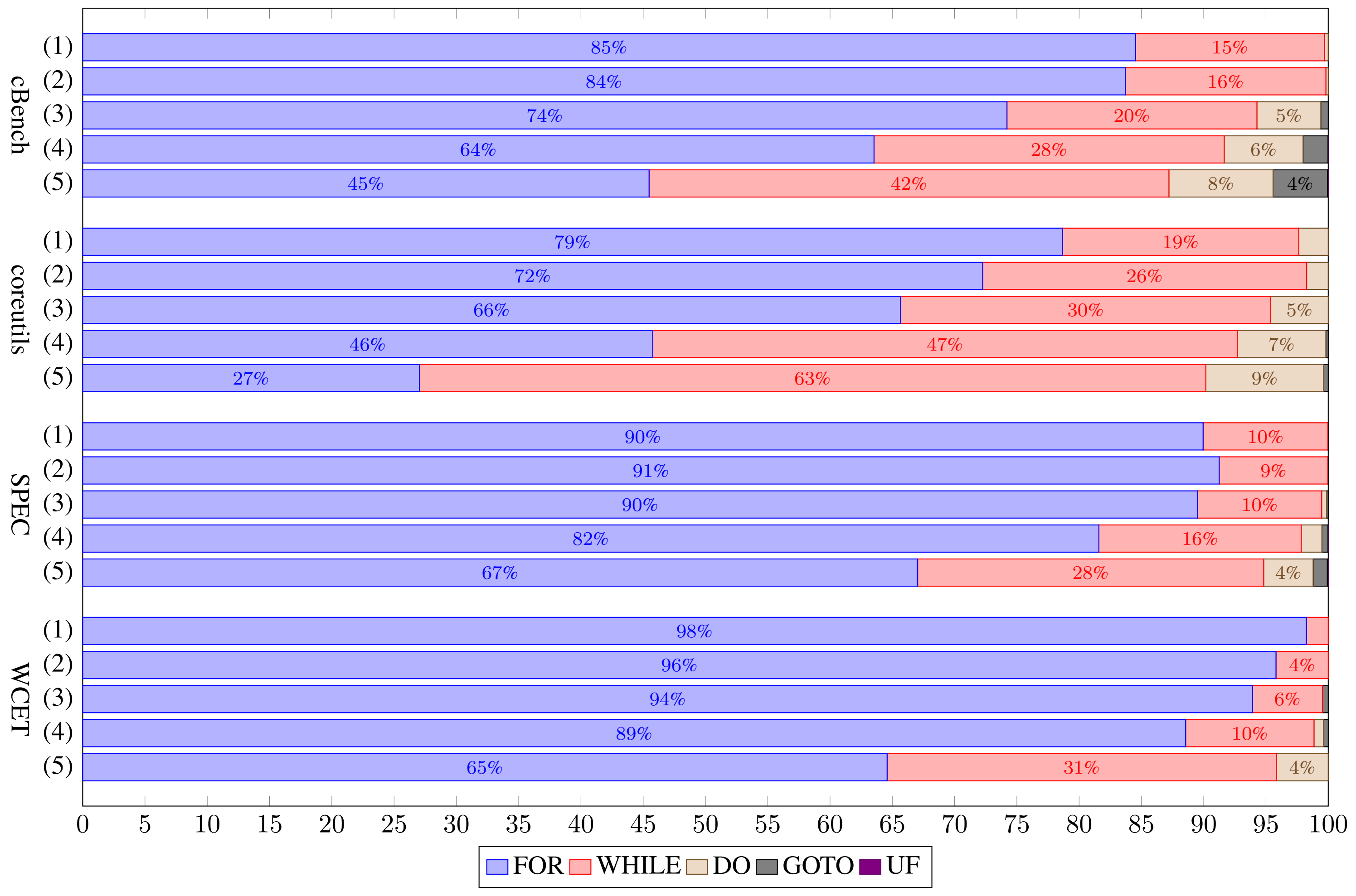


Figure 5.24: C statements used to express loops from loop classes (1) $\mathcal{L}^{SB}$, (2) $\mathcal{L}^{ST}$, (3) $\mathcal{L}^{simple(\text{*wxw})}$, (4) $\mathcal{L}$, and (5) $\mathcal{L} \setminus \mathcal{L}^{simple(\text{*wxw})}$.

Figures 5.25b to 5.28b show LOOPUS' performance on loops partitioned by their assigned statement. Consistent with the observation that a high percentage of simple loops are expressed by a `for` statement, LOOPUS performs significantly better on these with $\approx 80\%$ of `FOR` loops bounded. Fortunately, `FOR` loops also make up the biggest portion of loops in all benchmarks ($\geq 46\%$ as shown in Figures 5.25a to 5.28a) and thus contribute to the overall performance above average. Performance on `WHILE` and `DO` loops is about the same but significantly lower than on `FOR`, and is remarkably low for `GOTO` at 0–18%. LOOPUS fails to bound the single `UF` loop it discovers.

Discussion

We suggest possible explanations for the increased use of `for` statements to express simple loops:

Readability Arguably, `for` statements can be used to achieve better readability, because variable declaration and initialization, a controlling expression, and an advancing expression may be placed together in the statement's header. However, the programmer may not rely on the assumption that termination is only influenced by the statement's header, as arbitrary operations may occur in the statement's body. Thus, favoring the `for` statement can be beneficial for many simple loops – precisely those in which a single increment occurs as the last statement in the loop body, or which can be rewritten to an equivalent subprogram of such form.

Intended well-behavior A more subtle explanation is that programmers may mark loops as "well-behaved" by using the `for` statement. This appeals to follow Dijkstra's notion that "the programmer should let correctness proof and program grow hand in hand" [21]: If the programmer has a correctness proof in mind and finds it trivial, he writes the loop using a `for` statement, thus implicitly marking it as "low priority" for further quality-assuring measures, such as code reviews. In other cases, where the loop exhibits non-obvious flow of control, or he is not certain of his idea of a correctness proof, he may choose to use a `while` statement, thus marking it as "tricky", for the QA process and also to subsequent readers and editors of his own code.

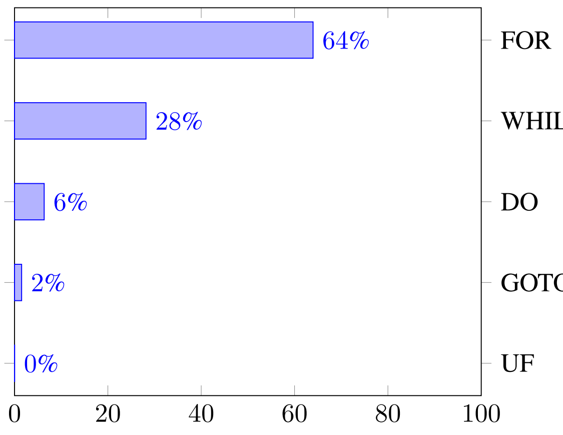
Assumed well-behavior Unexperienced programmers may even assume that the `for` statement imposes restrictions ensuring certain well-behavior, although this is not the case, as we briefly illustrated at the beginning of this section. Other programming languages, such as Ada and Fortran, provide loop statements with some guaranteed properties. Listing 5.5 illustrates such a statement: The Ada languages prevents assignment to the counter of `for` loops and the compiler enforces this property.

Although a strict guarantee as mentioned in the last paragraph is – depending on its implementation – either in general undecidable or very restrictive[1] for a C-like language, given the large number ($> 30\%$, c.f. Section 5.3) of syntactically terminating loops we found, we believe it is advisable to extend the C programming language with either:

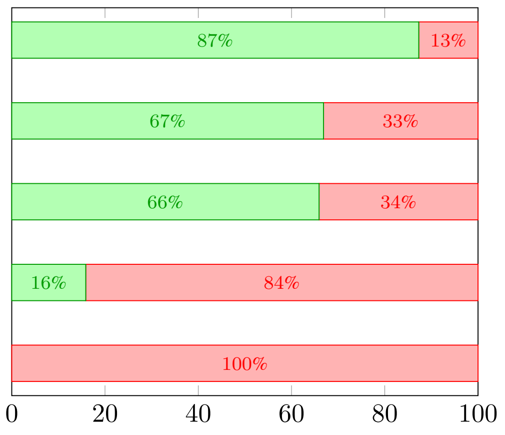
[1]by allowing only a small subset of the language in the loop body

Listing 5.5 A Ada `for` loop.

```
for i in 0..N loop
    -- not allowed:
    i := 42;
end loop;
```



(a) Percentage of statements in the cBench benchmark suite.



(b) Percentage of loops expressed by the resp. statement bounded (green) and not bounded (red) by LOOPUS.

Figure 5.25: Comparison of loop statements with LOOPUS results on cBench.

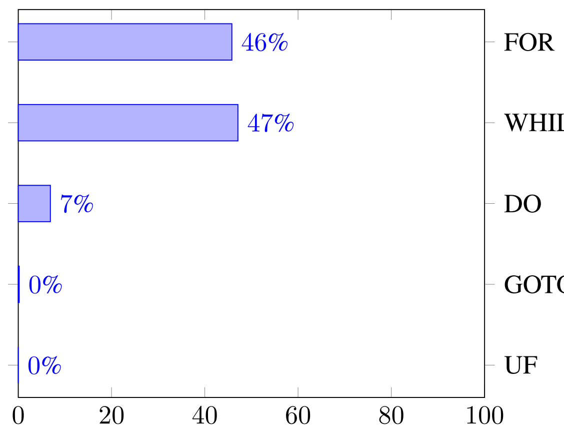
- Syntactic sugar supporting the pattern followed by syntactically terminating loops, thus clearly marking such loops as belonging to this class, but not enforcing particular restrictions. Such a statement is likely to allow expression of a wider class of loops than just $\mathcal{L}^{ST}$, similar to our notion of simple loops.

- Or, a restricted loop statement with some guaranteed properties, which are enforced by the compiler, thus restricting these loops to a well-defined subset such as $\mathcal{L}^{ST}$.

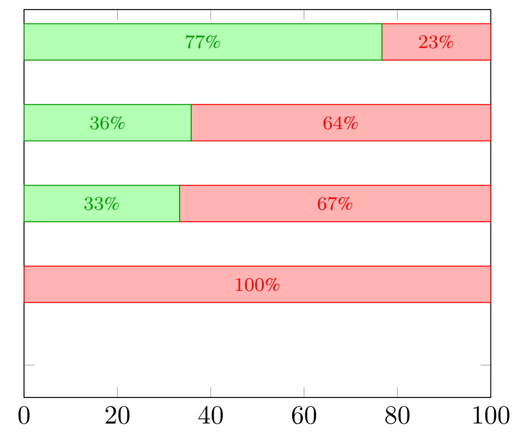
Both approaches – the first one more informally, the second more formally – would provide a way for both software engineers and automated program analysis tools to easily extract such implicit notions and patterns followed by the programmer from the source code. Of course, even without explicit language support, our tool SLOOPY can be used in a pre-analysis run to obtain this classification.

5.7 Finding Bugs with SLOOPY

Finally, we present an unexpected application of SLOOPY: As stated in Section 3.4, obtaining a syntactic termination proof by **increase** or **decrease** predicates over signed integers assuming two's complement overflow is an indicator of a buggy loop. Our tool SLOOPY encountered one

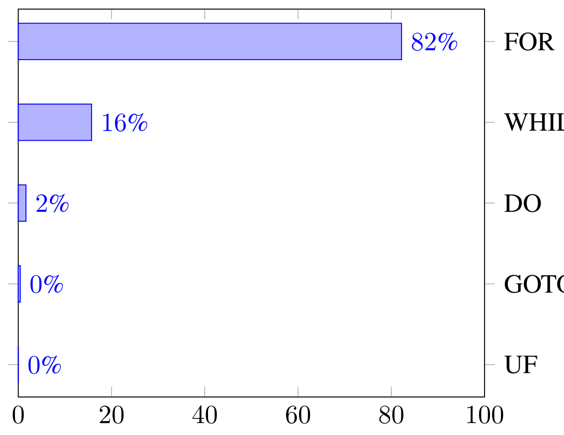


(a) Percentage of statements in the coreutils benchmark suite.

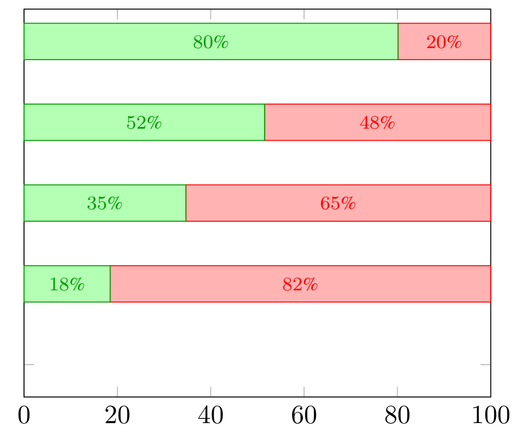


(b) Percentage of loops expressed by the resp. statement bounded (green) and not bounded (red) by LOOPUS.

Figure 5.26: Comparison of loop statements with LOOPUS results on coreutils.

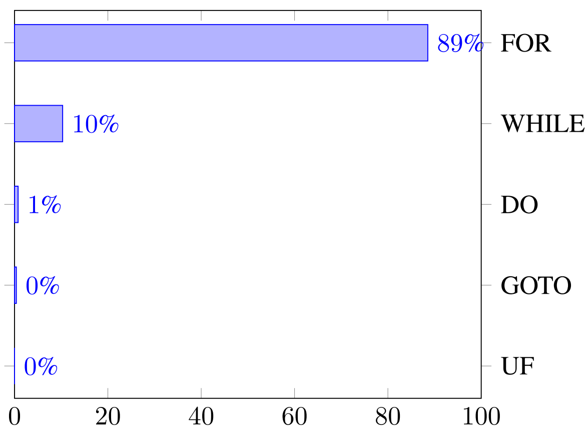


(a) Percentage of statements in the SPEC benchmark suite.

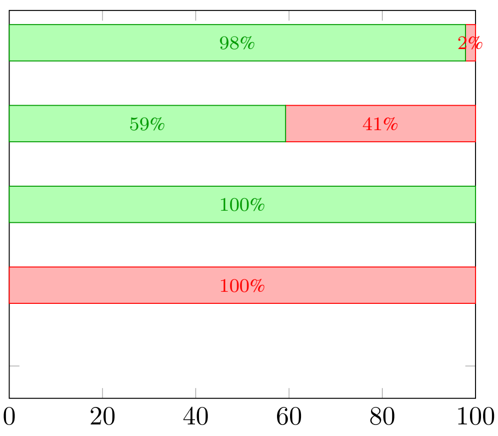


(b) Percentage of loops expressed by the resp. statement bounded (green) and not bounded (red) by LOOPUS.

Figure 5.27: Comparison of loop statements with LOOPUS results on SPEC.



(a) Percentage of statements in the WCET benchmark suite.



(b) Percentage of loops expressed by the resp. statement bounded (green) and not bounded (red) by LOOPUS.

Figure 5.28: Comparison of loop statements with LOOPUS results on WCET.

such bug in the Huffman encoder of GPL Ghostscript, which is part of the cBench suite. The malfunctioning code has been introduced in 1997, and to our knowledge has gone undetected for sixteen years. We raised the issue with the GPL Ghostscript developers, and the buggy code was subsequently removed.

Let us study the affected loop, whose source code is given in Listing 5.6: Variable `n` counts from its initial value to `MAX_INT`, and then overflows. From the context, it is clear that the loop was actually intended to enumerate the range $[0, n-1]$, i.e. the increment was supposed to read `j++`.

Using Google to search for the exact expression "`for (i = 0; i < n; n++)`" returns 387 000 results, which shows that this kind of typo-based bug is not uncommon. We believe SLOOPY is an effective and efficient tool for detecting this kind of bug. It may even suggest possible corrections by trying to prove a modified loop statement which substitutes the loop counter with other visible variables.

Listing 5.6 A buggy loop in GPL Ghostscript.

```
void hc_definition_from_bytes() {

    /* omitted for brevity */

    int n = (dbytes[i] >> 4) + 1;

    for (int j = 0; j < n; n++) {    // buggy loop
        def->values[counts[l]++] = v++;
    }

    /* omitted for brevity */
}
```

CHAPTER 6

Related Work

6.1 Definite Iteration

Definite iteration describes an iteration over all elements of a set, such as the nodes of a linked list, the elements of an array, or an integer sequence. Stavely [51] gives a general definition. Some languages provide definite iteration as restricted loop statements, such as the Fortran `do` statement or the Ada `for` statement. Hoare discusses a few special cases in [33]. However, none of the loop statements in C guarantees this property, and there is no recent study determining how many loops take this restricted form.

A study by Stavely [50] published in 1993 considers similar classes of definite iteration as our approach. The author determines iteration over high-level, *abstract structures* such as sets, sequences or finite mappings. Because adequately determining these abstract data types is by itself a difficult task, classification was performed manually by the author's reasoning. Our work takes an algorithmic approach, which yields a concise definition of the studied classes, and – as automated – is easily extended to other programs.

6.2 Programming Clichés and Plans

Rich [47] introduced the notion of *programming clichés*, which he defines as standard forms used in engineering problem solving. In the programming domain, such standard forms are be commonly used data structures or algorithms, such as *doubly-linked lists* or *bubble sort*. An important aspect of clichés is that they can be implemented as syntactically and semantically quite different programs. Consider, for example, the *string search* cliché:

String search cliché. Given a character string, the string is searched for all characters for which some predicate holds. For each of these elements, some action is performed.

We illustrate possible implementations of this cliché by giving two examples:

Listing 6.1 Index-based implementation of the string search cliché.

```
#define N 12
const char a[N] = "...";

for (int i = 0; i < N; i++) {
    if (predicate(a[i])) {
        // statement
    }
}
```

Listing 6.2 Pointer-based implementation of the string search cliché.

```
#define N 12
const char a[N] = "...";

for (char *c = a; c != 0; c++) {
    if (predicate(*c)) {
        // statement
    }
}
```

While the two implementations differ in their syntax and semantics, they encode the same basic cliché, searching for an element in an array (in this case, a character in a string). In order to summarize these different implementations in a common representation, Rich [47] introduced *programming plans*. Such plans describe constraints on the program's control flow and data flow. Programs fulfilling these constraints implement the cliché represented by the plan. Rich [47] gives a formal language for such plan diagrams.

In contrast to *simple loops*, the programming plans capture high-level tasks such as "search in a binary tree", "item search in data structure", or "compute running total", whereas simple loops are only concerned with the more general concept of termination.

6.3 Software Metrics

Two widely used metrics for software programs are lines of code (LOC), a pure syntactical count of lines, and cyclomatic complexity (CC), measuring the complexity of the control-flow structure of the program. While metrics considering data-flow inside the program exist [32, 55, 6], we are not aware of one that relates data-flow to influencing the control-flow of loops, as our approach does. Additionally, software metrics are usually used as "predictors of reliability (the frequency of operational defects)" [24], while our classification should not indicate defects but difficulty to prove termination. Whether the metrics introduced in this work can also be used to predict defects is an interesting research question (c.f. Chapter 7).

6.4 Iterators

Iterators have been introduced to provide sequential access to the elements of an aggregate, without exposing its underlying implementation. As such, they are related to the simple loops we study in this thesis. Various programming languages provide built-in support for implementing and looping over iterators, such as Alphard [48] (introduced in the 1970s), CLU [41] (early ’80s), and Sather [44] (around 1990). However, mainstream adoption only occured recently, e.g. in C# (2002) and Java, which provided the iterator pattern for library collection classes in J2SE 1.2 (1998), and introduced syntactic sugar for looping over iterators in J2SE 5.0 (2004) [16].

The Gang of Four book [27] describes iterators as an object-oriented pattern: The iterator provides operations for checking if the iterator may be advanced, for accessing the current element, and for advancing the iterator. All mentioned languages except CLU implement variations of this pattern.

As an abstraction of this approach, the language may provide a `yield` statement: Iterators work like routines, except that they additionally may yield: Executing a `yield` statement suspends operation of the iterator and returns control to the invoking loop statement, passing the `yield` statement’s arguments. Following the execution of the loop body, execution of the iterator is resumed. CLU and Sather directly implement this approach, and C# provides the `yield` statement as syntactic sugar for implementing the iterator pattern described above.

Language providing pointer arithmetic such as C++ may rely on another iteration concept: pointers i and end to the beginning and end of a data structure are acquired. Pointer i is then iteratively advanced through the data structure as far as end.

Iterators of the first kind are typically employed for well-behaved iteration similar to the one expressed by our simple loops. However, while simple loops prescribe this well-behavedness, iterators can actually implement arbitrarily complex iteration – they just provide a high-level interface to it. In this sense, simple loops describe a form of iteration that engineers usually expect from an iterator interface, but that is not guaranteed. The C++ kind of iterators is structurally closer to our work, but only feasible for data structures (i.e. not for integer-based iteration we studied here). We discuss extension of our work to data structures in Chapter 7.

6.5 Induction Variables

Induction variables are variables whose value is a function of the loop iteration count. Linear induction variables take values $ai + b$, where i is the count of loop iterations and a, b are loop-invariant [1]. Modern compilers implement analyses to discover induction variables and based on the discovered variables perform optimizations such as strength reduction or induction variable elimination.

While our approach to accumulated increments shares similarities with linear induction variables, we do not require counter values to be a function of the loop iteration. Rather, there may be arbitrary updates as long as the accumulated increment later implies eventual truth of an exit predicate (e.g. if only positive integers are added). In this respect, our analysis shares more similarities with interval abstract domains [17] than with induction variables.

CHAPTER 7

Research Directions

In this work we have introduced a pattern-based approach to classify loops with respect to their difficulty for automated program analysis and empirically shown that the respective classes indeed capture this difficulty. There are a number of research directions that naturally build on this work and that will benefit from further investigation:

Extension of simple loops to data structures. The simple loop classes presented in this thesis cover a majority of the loops in our benchmarks. However, these classes are solely based on integer and pointer arithmetic, i.e. they represent loops that iterate a range of integers or the contents of an array. It is still interesting to investigate other types of iteration, especially of data structures such as null-terminated lists, to find termination criteria similar to syntactically terminating loops and subsequently derive approximations as we did for simple loops. Clearly, the occurrence of these classes in practical examples should be studied in a fashion similar to this work.

Quality assurance in software engineering. While many methods such as testing or code reviews augment the software engineering process, in many settings software engineering is subject to economic constraints. A notion of difficulty can be used to allocate resources to the more complex cases. Additionally, application of automated program analysis is usually subject to high resource utilization, e.g. CPU time or memory consumption. If access to these resources is restricted, it makes sense to allocate them in a way where most impact is expected. In this context, it is also interesting to investigate applicability of this work as a classic software metric, i.e. to indicate likely defects, for example by measuring the number of non-simple loops in a module.

Internal heuristic for automated program analysis. Similar to the previous point, but at a lower level, we believe that a systematic treatment of loops can yield results to guide formal methods-based tools internally, e.g. as a heuristic for selecting appropriate abstraction functions [19], as input to invariant generation [38], or for preselecting loops that should benefit from additional preprocessing such as loop acceleration [39].

Programming language design. Programming languages should ideally be designed to make the software engineering process efficient. To improve programming language design, it is essential to study how programmers employ existing language constructs. Studying the applicability of our results to programming language design, and making concrete suggestions based on this data opens a further research direction in addition to the contributions to program analysis we have discussed in this work.

APPENDIX A

Further Resources

A.1 Source Code

The source code of the SLOOPY tool is available from the author's website at `http://forsyte.at/~pani/sloopy/`.

A.2 Experimental Results

Extensive datasets of the experimental evaluations presented in this thesis are available from the author's website at `http://forsyte.at/~pani/sloopy/`.

A.3 Loop Exchange Format

For comparison of experimental results between SLOOPY and LOOPUS we introduced a simple format for referring to loops in C programs: We use tuples $(path, line, property_1, property_2, \ldots)$ to refer to a natural loop's backedge, where $path$ is the relative filesystem path in the benchmark, and $line$ is the statement line number for back edges induced by unconditional jump statements (e.g. `continue` or `goto`), or the line number of the loop statement header for backedges induced by loop statements (`do`, `for`, `while`). The tuples can be output as comma-separated values to communicate among different tools.

As our references are line-based, we first preprocess the source code to obtain a single statement per line normal form. This can easily be accomplished using the C preprocessor (to inline macros) and a pretty-printing tool such as GNU INDENT. LOOPUS processes the frontend-translated LLVM intermediate representation (LLVM IR). To obtain file path and line number, we translate the C code with debug information, which generates appropriate metadata nodes in the LLVM IR. To make this metadata consistent with our definition from above, we employ a patched version of CLANG available from `https://github.com/thpani/clang`.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Princiles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. "Thorough static analysis of device drivers". In: *EuroSys*. 2006, pp. 73–85.

[3] Josh Berdine, Byron Cook, and Samin Ishtiaq. "SLAyer: Memory Safety for Systems-Level Code". In: *CAV*. 2011, pp. 178–183.

[4] Dirk Beyer. "Competition on Software Verification - (SV-COMP)". In: *TACAS*. 2012, pp. 504–524.

[5] Dirk Beyer. "Second Competition on Software Verification - (Summary of SV-COMP 2013)". In: *TACAS*. 2013, pp. 594–609.

[6] Dirk Beyer and Ashgan Fararooy. "A Simple and Effective Measure for Complex Low-Level Dependencies". In: *ICPC*. 2010, pp. 80–83.

[7] Dirk Beyer and M. Erkan Keremoglu. "CPAchecker: A Tool for Configurable Software Verification". In: *CAV*. 2011, pp. 184–190.

[8] Dirk Beyer, M. Erkan Keremoglu, and Philipp Wendler. "Predicate abstraction with adjustable-block encoding". In: *FMCAD*. 2010, pp. 189–197.

[9] Armin Biere. "PicoSAT Essentials". In: *JSAT* 4.2-4 (2008), pp. 75–97.

[10] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. "Symbolic Model Checking: 10^{20} States and Beyond". In: *LICS*. 1990, pp. 428–439.

[11] Busybox. *BusyBox: The Swiss Army Knife of Embedded Linux*. URL: `http://www.busybox.net/`.

[12] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. "Bounded Model Checking Using Satisfiability Solving". In: *Formal Methods in System Design* 19.1 (2001), pp. 7–34.

[13] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach". In: *POPL*. 1983, pp. 117–126.

[14] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. “Counterexample-Guided Abstraction Refinement”. In: *CAV*. 2000, pp. 154–169.

[15] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. “Proving program termination”. In: *Commun. ACM* 54.5 (2011), pp. 88–98.

[16] Sun Corporation. *JSR 201: Extending the JavaTM Programming Language with Enumerations, Autoboxing, Enhanced for loops and Static Import*. 2004. URL: `http://jcp.org/en/jsr/detail?id=201`.

[17] Patrick Cousot and Radhia Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *POPL*. 1977, pp. 238–252.

[18] cTuning. *Collective Benchmark (cBench)*. URL: `http://cTuning.org/cbench`.

[19] Yulia Demyanova, Helmut Veith, and Florian Zuleger. “On the Concept of Variable Roles and its Use in Software Analysis”. In: *FMCAD*. 2013, pp. 226–229.

[20] Will Dietz, Peng Li, John Regehr, and Vikram S. Adve. “Understanding integer overflow in C/C++”. In: *ICSE*. 2012, pp. 760–770.

[21] Edsger W. Dijkstra. “The Humble Programmer”. In: *Commun. ACM* 15.10 (1972), pp. 859–866.

[22] Kamil Dudka, Petr Peringer, and Tomás Vojnar. “Byte-Precise Verification of Low-Level List Manipulation”. In: *SAS*. 2013, pp. 215–237.

[23] Kamil Dudka, Petr Peringer, and Tomás Vojnar. “Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic”. In: *CAV*. 2011, pp. 372–378.

[24] Norman E. Fenton and Martin Neil. “Software metrics: successes, failures and new directions”. In: *Journal of Systems and Software* 47.2-3 (1999), pp. 149–157.

[25] Free Software Foundation, Inc. *Coreutils - GNU core utilities*. URL: `http://www.gnu.org/software/coreutils/`.

[26] Andy Galloway, Gerald Lüttgen, Jan Tobias Mühlberg, and Radu Siminiceanu. “Model-Checking the Linux Virtual File System”. In: *VMCAI*. 2009, pp. 74–88.

[27] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[28] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. “Synthesizing software verifiers from proof rules”. In: *PLDI*. 2012, pp. 405–416.

[29] Sumit Gulwani and Florian Zuleger. “The reachability-bound problem”. In: *PLDI*. 2010, pp. 292–304.

[30] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. “The Mälardalen WCET Benchmarks: Past, Present And Future”. In: *WCET*. 2010, pp. 136–146.

[31] John L. Henning. “SPEC CPU2006 benchmark descriptions”. In: *SIGARCH Computer Architecture News* 34.4 (2006), pp. 1–17.

[32] Sallie M. Henry and Dennis G. Kafura. “Software Structure Metrics Based on Information Flow”. In: *IEEE Trans. Software Eng.* 7.5 (1981), pp. 510–518.

[33] C.A.R. Hoare. “A note on the for statement”. In: *BIT Numerical Mathematics* 12.3 (1972), pp. 334–341.

[34] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. “FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement”. In: *CAV*. 2008, pp. 209–213.

[35] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. “How did you specify your test suite”. In: *ASE*. 2010, pp. 407–416.

[36] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. Dec. 8, 2011.

[37] Gary A. Kildall. “A Unified Approach to Global Program Optimization”. In: *POPL*. 1973, pp. 194–206.

[38] Laura Kovács and Andrei Voronkov. “Finding Loop Invariants for Programs over Arrays Using a Theorem Prover”. In: *FASE*. 2009, pp. 470–485.

[39] Daniel Kroening, Matt Lewis, and Georg Weissenbacher. “Under-Approximating Loops in C Programs for Fast Counterexample Detection”. In: *CAV*. 2013, pp. 381–396.

[40] Daniel Kroening and Georg Weissenbacher. “Interpolation-Based Software Verification with Wolverine”. In: *CAV*. 2011, pp. 573–578.

[41] Barbara Liskov, Russell R. Atkinson, Toby Bloom, J. Eliot B. Moss, Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*. Vol. 114. Lecture Notes in Computer Science. Springer, 1981.

[42] Florian Merz, Stephan Falke, and Carsten Sinz. “LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR”. In: *VSTTE*. 2012, pp. 146–161.

[43] Leonardo Mendonça de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *TACAS*. 2008, pp. 337–340.

[44] Stephan Murer, Stephen M. Omohundro, David Stoutamire, and Clemens A. Szyperski. “Iteration Abstraction in Sather”. In: *ACM Trans. Program. Lang. Syst.* 18.1 (1996), pp. 1–15.

[45] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C, 2nd Edition*. Cambridge University Press, 1992.

[46] Jean-Pierre Queille and Joseph Sifakis. “Specification and verification of concurrent systems in CESAR”. In: *Symposium on Programming*. 1982, pp. 337–351.

[47] Charles Rich. “A Formal Representation for Plans in the Programmers Apprentice”. In: *IJCAI*. 1981, pp. 1044–1052.

[48] Mary Shaw, William A. Wulf, and Ralph L. London. “Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators”. In: *Commun. ACM* 20.8 (1977), pp. 553–564.

[49] James Stanier and Des Watson. “A study of irreducibility in C programs”. In: *Softw., Pract. Exper.* 42.1 (2012), pp. 117–130.

[50] Allan M. Stavely. “An empirical study of iteration in applications software”. In: *Journal of Systems and Software* 22.3 (1993), pp. 167–177.

[51] Allan M. Stavely. “Verifying Definite Iteration Over Data Structures”. In: *IEEE Trans. Software Eng.* 21.6 (1995), pp. 506–514.

[52] Alan M Turing. “On computable numbers, with an application to the Entscheidungsproblem”. In: *Proceedings of the London mathematical society* 42.2 (1936), pp. 230–265.

[53] Leslie G. Valiant. “The Complexity of Enumeration and Reliability Problems”. In: *SIAM J. Comput.* 8.3 (1979), pp. 410–421.

[54] Mark Weiser. “Program Slicing”. In: *ICSE*. 1981, pp. 439–449.

[55] Elaine J. Weyuker. “Evaluating Software Complexity Measures”. In: *IEEE Trans. Software Eng.* 14.9 (1988), pp. 1357–1365.

[56] Thomas Witkowski, Nicolas Blanc, Daniel Kroening, and Georg Weissenbacher. “Model checking concurrent linux device drivers”. In: *ASE*. 2007, pp. 501–504.

[57] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O’Hearn. “Scalable Shape Analysis for Systems Code”. In: *CAV*. 2008, pp. 385–398.

[58] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. “Bound Analysis of Imperative Programs with the Size-Change Abstraction”. In: *SAS*. 2011, pp. 280–297.