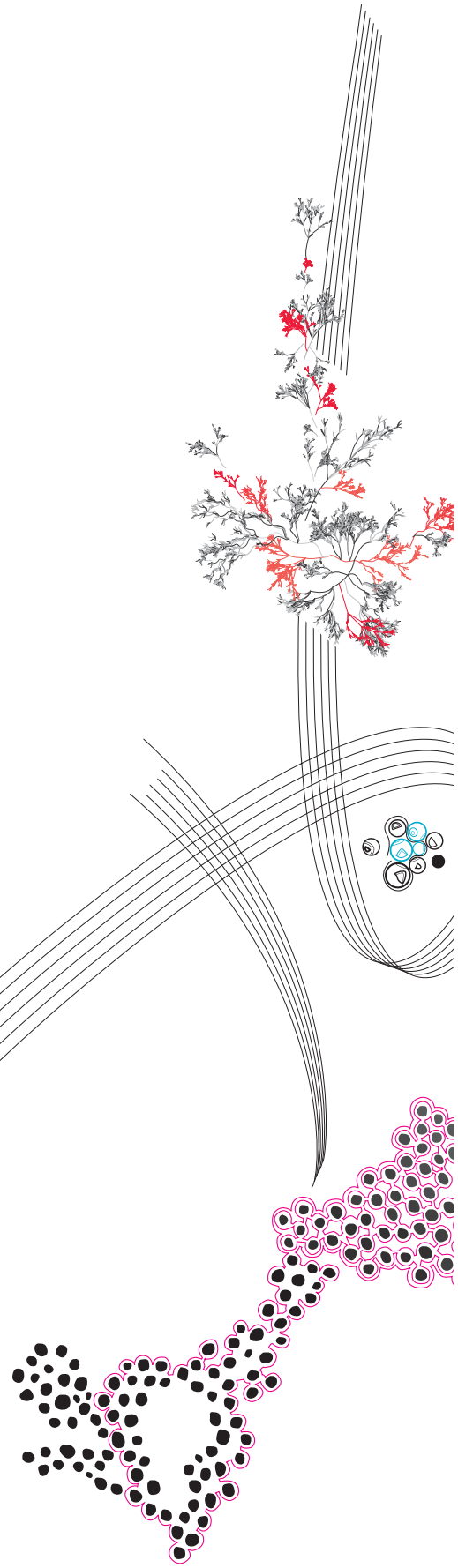

MSc Computer Science
Final Project

AValAnCHE:
Improving robustness of the VerCors verification toolset using fuzzing

Wander Nauta

Supervisors: prof.dr. Marieke Huisman, dr. Marcus Gerhold
External examiner: dr.ing. Kuan-Hsun Chen

June 9, 2025

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Abstract

VerCors [8] is a verification toolset for sequential and concurrent programs. Given a suitably annotated program, written in one of a variety of programming languages, the toolset can show both functional correctness and data race freedom.

The applicability and usability of the toolset has already been shown in a number of case studies [1]. However, VerCors cannot yet verify itself: it is written in the Scala programming language, which is not one of languages it supports. The toolset as a whole is also much larger and more complex than programs verified so far. Instead, automated tests are used to increase confidence in the correctness and robustness of VerCors, and so confidence in its output.

We have investigated whether adding fuzz testing to the VerCors testing strategy could aid in finding robustness issues in VerCors itself, especially in the face of unexpected inputs, such as those that may be tried by beginning users who are not yet familiar with the toolset.

In particular, we have compared a number of different fuzzing approaches. While purely random and coverage-guided fuzzing gave limited results, we have shown that grammar-based fuzzing can be successfully used to find robustness issues in VerCors. In total, we have found 28 confirmed new crashing bugs in the toolset.

To support our comparison of different fuzzing approaches and aid integration into the VerCors testing strategy, we have developed a fuzzing tool specifically tailored to VerCors itself. Such a fuzzing tool provides **a**dditional **val**idation by generating **an**notations and **c**ode in **h**igh-level languag**e**s, hence AValAnCHE.

Keywords: fuzzing, automated testing, robustness, code coverage, program verification

Acknowledgements

I owe much gratitude to my supervisors, prof.dr. Marieke Huisman and dr. Marcus Gerhold, for their patience, enthusiasm, understanding and support as I veered into many tangents and even switched topics partway through the process of writing a thesis.

Many thanks should also go out to the VerCors contributors, in particular Alexander Stekelenburg MSc, Bob Rubbens MSc, and Pieter Bos MSc, both for their important work on the toolset in general and for repairing and triaging many of the issues AValAnCHE has uncovered, sometimes within minutes of the issues being discovered.

I am also grateful to the fellow members of EEMCS Graduation Support Group 4 for their help in keeping me more or less on track during the entire process: Anissa, Daan, Pelle, Sanne, Tim, Wouter, and Youri; and to the group mentors, Ellen Wassink-Kamp MSc and Erik Bong MSc.

The Formal Methods and Tools group allowed me to present my in-progress research as part of their FMT Colloquium series and offered both interesting questions and useful suggestions and feedback, for which I am also very grateful.

Finally, I would like to thank the good folks at the Snoonet #thenetherlands IRC channel for the positive and productive discussions over the past years – never about fuzzing program verification tools, but productive discussions nonetheless.

Contents

Chapter 1

Introduction

Computerized decision-making is all around us. Over a span of decades, important choices, including those with large impacts on human lives and fortunes, have been increasingly left to software. Computer programs decide whether a self-driving car should brake or swerve, whether a specific delivery driver or post office worker should be suspected of fraud, which dose of therapeutic radiation is appropriate, which candidate is the best fit for a job opening, and which political campaign slogan should be shown to whom.

In our view, for humanity to safely build our world on top of software in this way, what is at least required is that the machine's actions match their operator's intentions. That is, a computer program should conform to the designer's formal or implied specifications. For instance, the program must not terminate unexpectedly and it must always give the expected result, whatever that means for the program in question.

This is a high bar to clear for complex software.[1]

A number of techniques exist for increasing our confidence in the belief that our programs will behave as we intend. Some of these techniques have been automated, at least in part, with automated testing and verification tools: here, a computerized decision-making process inspects a new version of the software, perhaps by executing it to determine its behavior, perhaps by analyzing its source code, and judges whether it is suitable for the task at hand.

Of course, we must now have confidence in our automation, which may be a complex piece of software in its own right. *It* must always give the expected result, rather than a false positive, or we risk deploying faulty software that we thought we had tested or verified to be correct. *It* must be robust and not itself crash when judging our programs, or the delivery of new versions of our software is held up unnecessarily.

The VerCors program verification toolset [8] is an example of such an automation. It is indeed a large and complex piece of software, with support for verifying programs in many programming languages. The VerCors project already uses a number of automated testing techniques; the goal of this research is to evaluate whether adding *fuzzing*, automated testing by generating random inputs, would be useful as an additional approach towards finding robustness issues, which can then be fixed.

Our hope is that further increasing the robustness of the VerCors toolset will then be a small step towards its wider application, finally leading to a world where the correctness of more and more software is formally verified.

[1] The author will readily admit not always having achieved this lofty goal even in his own programs. One such program managed to display "!bɹɹow ˌoɭɟəH" on screen before promptly crashing.

1.1 Terminology

To establish our usage of the terms, we will at this point introduce automated testing, fuzz testing, and verification in slightly more detail.

- *Automated testing* is executing programs with the intent of making them fail [48]. Testing can occur on many levels of granularity, from exercising individual functions (unit testing) to entire systems (integration testing). It does not require a formal specification of the program's behavior: a common approach is to give a table of example inputs and expected outputs, then automatically try each case. Test cases may be collected or written by developers or generated automatically. Testing is rarely exhaustive, and it gives no guarantees about inputs that were not tried.

- *Fuzz testing* is an automated testing approach where many (for instance, millions) test inputs to a function or system are randomly generated. The goal is not to exhaustively test every possible input, which is generally impossible; rather, the idea is to try a wide variety of inputs to elicit interesting (that is, incorrect or crashing) behavior. The output of the system may be compared to some known-good result, or we may just be interested in the fact that no fuzz input should cause the program to halt unexpectedly or commit a memory error. We will further discuss fuzz testing and give a short history of the approach in Chapter 2.

- *Program verification* is a formal analysis technique where certain properties about a program are stated formally and then proven, through logic-based proof rules and axioms which are applied to the structure of the program. Useful properties to prove could include general properties such as 'the program terminates, regardless of the input' or 'the program finds the largest integer in any array of integers'. Verification can give much stronger guarantees than testing. Chapter 3 examines program verification in more detail as part of its discussion of the VerCors toolset.

Deductive verification of programs can be done on paper, but a more productive and less error-prone approach is to use a verification tool such as VerCors. Even with such a tool, the effort required to specify the behavior of a program in such detail that it can be proven correct is in general much greater than the effort involved in adding basic test cases.

There would be a beautiful symmetry to being able use a verification tool like VerCors to prove that a verification tool like VerCors is perfect: that it verifies all correct programs (completeness), does not verify any programs that are incorrect (soundness), and that it never unexpectedly terminates (robustness).

This is, however, a large or perhaps impossible undertaking. VerCors is a substantial and complex piece of software, with dependencies that are in turn also complex, the behavior of which would need to be specified and either assumed correct or proven to be so. Further practical difficulties exist: for example, VerCors is written in the Scala programming language, which is not one of the many languages that the tool supports. An optimal strategy may therefore be to integrate multiple techniques [43].

In short, the aim of our research is to evaluate whether adding fuzz testing to the techniques used in the VerCors project would be advantageous, specifically focusing on finding robustness issues in VerCors itself. In a metaphorical sense, fuzz testing a verification tool is perhaps somewhat like hitting a fine surgeon's scalpel with a large rubber mallet to see whether it shatters under adverse conditions, then hitting it a few hundred thousand more times to be sure; lacking the beautiful elegance, perhaps, but very practical.

Here and throughout this thesis, we will use the term 'robustness' in the sense of some program not crashing for any input, including inputs that are erroneous in some sense. In [15], Dyck et al. studied the robustness of program verifiers using a different definition: whether the answer from a verification tool (verified, not verified, unknown) is robust to small changes to the program being verified, that is, unchanging for small changes that do not affect the semantics of the program. That level of detail is outside the scope of our research: we are only looking for cases that cause verification to fail entirely.

As deductive verification tools improve further, get easier to use, and barriers to adoption are lowered, the ideal strategy for ensuring the quality of software in general may start to gradually include more formal verification and less testing. The same should then apply to verification tools like VerCors. Until that time, adding some fuzz testing to the mix, in addition to the test cases written by VerCors developers, may be the best approach.

1.2 Overview

This thesis is structured as follows.

Chapter 2 introduces fuzz testing in a bit more detail. Our research builds on a long history of testing the robustness of programs by providing them with random inputs, then observing whether those inputs cause the programs to crash. A brief overview of that history, describing the use of fuzzing as a fault-finding tool in general, is given in this chapter. Of the many fuzzing approaches that are described in the literature, and the hundreds of fuzzing tools that are readily available as open source software, we describe some specific strategies and tools that we have evaluated for comparison in more detail.

Chapter 3 describes the VerCors verification toolset and its architecture, and briefly covers the theory of deductive verification and separation logic on which the VerCors toolset is based, although only in very broad strokes. We aim to give enough context to understand how the toolset can be used and what problems it is aimed at solving, as well as some insight into the software architecture of the tool itself. Readers who are already familiar with VerCors may skip, or perhaps particularly enjoy, this chapter.

An important part of our research is to compare the performance of different fuzzing approaches. The chapter on performance metrics, Chapter 4, describes the 'code coverage over time' metric we have used in this comparison, as well as our measurement setup.

The concurrent threads started by the previous chapters join in Chapter 5, which describes the design and inner working of the AValAnCHE tool that we have developed as part of this research. The tool runs VerCors on fuzzer-generated inputs, collecting both findings and performance measurements. Chapter 5 also expands on some of the technical details for each of the approaches being compared. In addition, we describe our proposal for integrating AValAnCHE-based fuzzing into the VerCors testing strategy in this chapter.

With both the performance metric we have selected and the measurement tool in hand, we can perform our measurements, the results of which we have included as Chapter 6.

Ours is not the first application of fuzz testing to program verification tools. Chapter 7 continues on from Chapter 2 to describe a number of efforts that are more closely related to our own research. Specifically, we discuss some projects that successfully applied fuzz testing and test case generation to the Dafny verification-aware programming language [29], and how the approach in these projects differs from our own.

We conclude in Chapter 8, where we also give a number of recommendations based on our results, both for the VerCors project and for other verification tools. Finally, interested readers may refer to appendix A for a complete listing of the VerCors bugs found and reported during the course of this research.

1.3 Research questions

In summary, we have set out to answer the following research questions:

RQ1 To what extent can we improve confidence in the robustness of the VerCors toolset by adding fuzz testing?

- RQ1.1 How can we best integrate fuzzing into the existing VerCors testing strategy?
- RQ1.2 What conclusions can we draw from a code coverage measurement that our fuzzing tool provides, and how does that measurement relate to confidence?
- RQ1.3 Of the fuzzing strategies introduced in Chapters 2 and 5, which perform the best when applied to the VerCors codebase?

RQ2 How would the conclusions from RQ1 translate to verification tools other than VerCors, and what recommendations would follow from this?

Chapter 2

Fuzzing and test case generation

In this chapter, we introduce the concept of fuzz testing and give an overview of a number of fuzzing strategies and tools that have been discussed in earlier research.

More closely related tools, specifically on applying fuzzing and test case generation to verification tools, is the subject of the chapter on related work, Chapter 7.

2.1 A short history of fuzzing

The concept of automatically generating possible program inputs and running programs on those inputs to find unexpected program states dates back to at least the 1970s, for instance in the work by Purdom [37]. Today, tools to aid in this process are often called fuzzing tools or fuzzers.

The purest form of fuzzer generates completely random inputs and, for each input, executes some code of the system under test, either through its standard interfaces or through a fuzzer-specific entry point. If an input is found that causes a crash (or some other unwanted condition that can be automatically detected), the crashing input is stored and reported, so that the program can be improved. Since there is generally an infinite number of possible inputs, the fuzzing process never completes, and it is impossible in general to predict whether and when the fuzzer will find a next crash.

The original fuzzing tool was simply called `fuzz`, by Miller et al. [31], and was used by the authors to find bugs in standard Unix utility programs like `uniq(1)` and `tsort(1)`.

The method was inspired by the weather: one of the authors noticed that noise on a phone line, caused by a rainstorm happening overhead, introduced spurious characters in the dial-up connection between their terminal and the Unix machine they were working on. Despite the occasionally garbled input or output, they continued their work. However, to their surprise, the unexpected input caused some basic Unix utilities to crash, rather than exiting with an error message as could perhaps be expected from well-known, well-tested programs written by professional Unix programmers.

Based on this experience, the `fuzz` tool blindly generates binary or ASCII data and feeds this 'noise' to the system being tested. Writing in 1990, the authors remark that 'to make a systematic statement about the correctness of a program, we should probably use some form of formal verification' but that, in the meantime, their fuzzing approach, while not a substitute, offered 'an inexpensive mechanism to identify bugs and increase overall system reliability,' aiming to 'complement, not replace, existing test procedures'.

The authors also repeated their study in 2021 [32], and discovered that the venerable `fuzz` still found crashing bugs in core Linux utilities.

Tool	Generates...	Based on and guided by...
`fuzz` [31]	Random ASCII strings or random printable ASCII strings	-
AFL++ [17]	Random bytes	Mutation of input corpus guided by branch coverage information
syzkaller [40]	Sequences of Linux system calls, as C programs and bug reports	Mutation of existing findings guided by Linux kernel code coverage
Grammarinator [22]	Syntactically correct programs	User-provided ANTLR grammar and optional user-provided weights
NAUTILUS [3]	Grammatically-correct programs	Mutating a corpus according to scriptable grammar, coverage data
Grimoire [5]	Mostly grammatical phrases	Learning the grammar at run time through code coverage information
Csmith [46]	C programs with defined behavior	Built-in deep knowledge of C grammar, semantics, and edge cases
Xsmith [20]	Programs in user-specified languages	User-provided syntax and semantic description
XDsmith [24]	Annotated XDafny programs and expected validation result	Knowledge of Dafny and execution of unannotated programs

TABLE 2.1: A small selection of fuzzing and test case generation tools, demonstrating the wide variety of approaches.

Already in the earlier work by Miller, fuzzers have been associated with security research, especially in the context of searching for security vulnerabilities in software written in memory-unsafe languages that interacts with (and thus needs to be robust in the face of) input from untrusted sources.

The article explicitly references the (recent, at the time!) Morris worm that exploited a buffer overflow in the *finger* network service to spread. To give a more recent example: fuzz testing is commonly used with image parsing code that is used in web browsers. No image included in a webpage should be able to cause the web browser to misbehave, write to out-of-bounds memory, or crash.

If the software being tested has a crashing bug, the number of potential inputs that can trigger the crash is often also infinite. To make the manual investigation of results feasible, inputs that lead to 'the same' crash (crashes that are caused by the same bug) should be grouped together, referred to as the 'fuzzer taming problem' by Chen et al. in [10]. A heuristic is required to perform this grouping, and different fuzzers make different trade-offs. The approach taken by the American Fuzzy Lop (AFL) fuzzer, for example, is based on the uniqueness of stack traces, and discussed in more detail in their technical whitepaper [47].

For the results of a fuzzing campaign to be useful for project developers, it is often useful to minimize the crashing inputs, that is, finding the shortest input that still reproduces the same crash. This minimization process can often be automated, either using functionality built into some fuzzers or by using a stand-alone test case reduction tool like Creduce [38].

Fuzzers may run unattended, either for a fixed duration or as long as compute resources are available. An example of such an approach is the syzkaller [40] and OSS-fuzz [2] projects by Google, which search for bugs in Linux kernel code and widely-used open-source libraries, respectively. In the former case, if a sequence of system calls is found that crashes the kernel, syzkaller's syzbot reports this to kernel development mailing lists automatically, including a C program that reproduces the issue. In the latter case, issues are automatically reported to a dedicated issue tracker and shared with project maintainers.

2.2 Strategies beyond random

Most target programs are robust enough that most invalid inputs are correctly rejected and do not lead to crashes. This means that a purely random 'brute-force' approach, while arguably elegant in its conceptual simplicity, is unlikely to find bugs in such programs. To improve on this, a number of strategies can be distinguished, with many tools using combinations of such strategies. An extensive survey, taxonomy and genealogy, covering over 60 distinct fuzzing tools, has been prepared by Manès et al. [30].

In each case, a balance exists: a more complicated ('smarter') fuzzer may introduce additional run-time overhead, slowing down either input generation or execution of the program under test. If this overhead is too large, the fuzzer may miss crashing inputs that a simpler, more brute-force approach would have found.

A common improvement over random fuzzing is to use *dictionary-based fuzzing*. A dictionary of byte sequences is prepared that is expected to trigger behavior in the system under test, and the fuzzer prefers generating these sequences over others. As an example, an input containing the sequence `clams` is unlikely to trigger special logic in a Java parser, but the character sequence `class` might, when inserted at the right location. A dictionary-based fuzzer is not aware of the expected structure of the input and is not limited to only inputs that are 'valid' in some sense. The likelihood of finding a valid input, or an input that is valid enough to cause interesting behavior, is merely increased.

Another common approach, which forms the core of the popular American Fuzzy Lop (AFL) [47] fuzzer, its derivatives like AFL++, and many other tools, is *coverage-guided mutation-based fuzzing*. These fuzzers instrument the system under test to measure which code is executed by each input, often as a fraction of the total (the test coverage percentage), either on an instruction level or (as an optimization) on a branch level. An input that causes execution to take a different path than has been seen before, and which thus exercises new code and increases the coverage metric, is regarded as interesting and favored when producing future inputs. The inputs are then modified (sliced, damaged, shuffled, combined) in a number of ways. Mutations that cause a crash or other observable effect are kept.

Most mutation-based fuzzers require an initial corpus of inputs, and perform best when this corpus is varied enough to exercise many different features of the code under test. To create the image in Figure 2.1, a valid, undamaged JPEG image of the White Rabbit from a Lewis Carroll book was given as the initial corpus to the AFL fuzzer (the American Fuzzy Lop is a breed of rabbit). The frame shown here is one of the inputs that AFL generated. In this instance, the file is relatively unharmed, still parseable as a JPEG image, and even partially recognizable as a rabbit, but the goal is to generate varied inputs.

FIGURE 2.1: A frame from the American Fuzzy Lop logo, by Michal Załewski.

2.3 Fuzzing to test compilers

In the survey of compiler testing techniques by Chen et al. [9], the authors remark that 'manually constructed test programs have been used since the early days of compiler testing.' As we noted in Section 3.4, this is the approach taken for VerCors, and it is indeed very common: the test suites for the LLVM (clang)[1] and GNU (gcc)[2] C and C++ compilers contain many thousands of example inputs, each labeled with their expected result. However, the authors note that fuzzing and test case generation tools are also increasingly used, with some particular fuzzing strategies being especially applicable if the program under test is (similar to) a compiler.

Grammar-based fuzzing restricts the generated inputs to the subset that could be produced by (and, therefore, parsed by) some language grammar. The Grammarinator fuzzer by Hodován et al. [22] is an example of this approach, where based on a parser grammar in ANTLR syntax, the fuzzer generates only inputs (sentences) that conform to the grammar, with a weighted choice between grammar productions. Because only grammatically correct inputs are generated, it logically follows that with this tool, it is no longer possible to find crashes in the software under test related to the handling of invalid input. However, the inputs generated in this way are more likely to exercise (and so find bugs in) parts of the compiler that execute after the initial lexing and parsing phases.

The NAUTILUS fuzzer by Aschermann et al. [3] combines this grammar-based fuzzing approach with the coverage-guided mutation approach from the previous section. Rather than mutate inputs on a byte level, the tool can use its understanding of the input grammar to shorten, lengthen, or transplant entire branches of the parse tree. A corpus is not needed: instead, the tool uses the grammar to generate its own initial inputs.

Most grammar-based fuzzing tools require a specification of the grammar, which needs to be provided to the fuzzer. However, Blazytko et al. showed in [5] that such a grammar can also be synthesized by observing how code coverage changes as inputs are tried. Their Grimoire tool builds ('learns') an understanding of the structure of the input as it runs.

2.4 Test case generators

Gradually, we proceed from fuzzers that generate purely random inputs towards tools that generate inputs that follow an expected structure, that is, inputs that are lexically and grammatically valid. These inputs exercise later parts of the compiler, after the lexing and parsing phases. If we want to find even 'deeper' bugs, we must generate inputs that are also semantically valid and pass type checking phases in the compiler. Rather than fuzzers, tools that can generate these valid inputs are sometimes called just *randomized test case generation* tools.

The Csmith tool by Yang et al. [46] has been very successful in finding bugs in C compilers. At a high level, Csmith is like a grammar-based fuzzer, but rather than attempting to randomly cover the entire C grammar, it generates only a subset: programs that are semantically valid and have known, safe, defined behavior. Specifically, the generator avoids generating programs that contain undefined or implementation-defined behavior, and it avoids generating infinite loops. Because of these properties, all programs generated by Csmith can not only be successfully compiled but also successfully executed in a finite amount of time, and then should have identical behavior regardless of the compiler (or compiler version or compiler flag) used, as guaranteed by the C standard. If a difference

[1] Available at `https://github.com/llvm/llvm-test-suite`.
[2] Available at `https://gcc.gnu.org/git/?p=gcc.git;a=tree;f=gcc/testsuite`.

in behavior is detected in programs compiled with different compilers, a miscompilation bug has been found in either of those compilers.

As the name implies, Csmith is limited to generating only C programs, and it contains quite deep knowledge of not only C grammar but also of its semantics, the effects of expressions, and the guarantees given by the C language standard. The Xsmith [20] project is an effort to reduce the effort required to write Csmith-like tools that target different languages. Users of Xsmith write a description of their language's grammar and semantics in Racket, a dialect of LISP, and can reuse components that occur in multiple languages.

Chapter 3

The VerCors verification toolset

VerCors is a program verification toolset. Given the source code to a program, VerCors can guarantee that such a program is free of a number of classes of errors, such as data races and memory errors, including null pointer exceptions and array bound errors. It supports a large, and growing, number of programming languages in common use today, including Java, C++, C, and OpenCL. The toolset is especially suited to proving properties of concurrent programs, either using explicit threads or GPU-based parallelism. If functional specifications are provided, in the form of method contracts, VerCors can also show functional correctness, that is, it can prove that a program or method follows its specifications in every case. These specifications are provided by the specification writer as annotations embedded in the source code.

As a deductive verifier rather than a testing tool, VerCors works statically, without executing the program that is being verified. (Indeed, its own PVL Prototypal Verification Language can only be verified, never executed.) It checks that the program is correct for every possible input and every potential interleaving of threads of execution.

By default, the tool checks partial correctness: it shows that *if* the program (or part of the program) finishes, *then* it gives the correct result. It can additionally be asked to prove that a program terminates, showing total correctness.

Our own fuzz testing approach, which will be the subject of the chapters from Chapter 4 onwards, is a 'black-box' approach in the sense that it does not strictly depend on knowledge of the inner workings of VerCors. Therefore, this chapter is primarily intended to give enough context to introduce VerCors as our system under test, and to give the reader an impression of the capabilities of the toolset.

VerCors is a deductive verifier based on permission-based separation logic [19], an adaptation of concurrent separation logic [35], which is in turn an extension of Floyd-Hoare logic [18, 21]. Rather than introduce these logics in order, we will attempt to instead give a high-level overview by way of a simplified running example.

Our example program, the annotated source code of which is given in Figure 3.1, consists of a static Java method that returns the largest integer in an array of integers. For this method, VerCors can show that the contract holds: when given a non-null reference to a nonempty array of integers, the returned value is a largest element of that array, and the method arrives at that correct result in a finite number of steps. Furthermore, VerCors can prove that the method does not contain any data races.

VerCors is not an interactive proof assistant: it can prove the correctness of our example program from just the specification as given, without further human interaction.

```
class Numbers {
    //@ requires a != null;
    //@ requires a.length > 0;
    //@ requires Perm(a[*], 1\2);
    //@ ensures Perm(a[*], 1\2);
    //@ ensures (\forall int i; 0 <= i && i < a.length; \result >= a[i]);
    //@ ensures (\exists int i; 0 <= i && i < a.length; \result == a[i]);
    //@ decreases;
    static int largest(final int[] a) {
        int z = a[0];

        //@ loop_invariant Perm(a[*], 1\2);
        //@ loop_invariant 0 <= i && i <= a.length;
        //@ loop_invariant (\forall int j; 0 <= j && j < i; z >= a[j]);
        //@ loop_invariant (\exists int j; 0 <= j && j < i; z == a[j]);
        //@ decreases a.length - i;
        for (int i = 1; i < a.length; i++) {
            if (a[i] > z) {
                z = a[i];
            }
        }

        return z;
    }
}
```

FIGURE 3.1: A running example in Java that finds the largest integer in an array. Lines starting with //@ are VerCors annotations.

3.1 Theory by example

Our example method has a number of *preconditions*, labeled *requires*, and a number of *postconditions*, labeled *ensures*. Together, this block is referred to as a *method contract* between the method and its potential callers. The method promises that, if all preconditions are true when it is called, all postconditions will be true after it has finished.

Our method's precondition, which we refer to as P, specifies that the argument array a must be non-null and that its length must be positive. This avoids the 'empty array' case that would be confusing otherwise: among no integers, it is impossible to pick the largest.

Our method's promised postcondition R is then the conjunction of two first-order logic formulas that together describe the informal specification we gave above, namely that we want the largest element of the array:

1. The result of the method is larger or equal than all the elements in the array.
2. The result of the method is equal to at least one of the elements of the array.

VerCors must diligently verify both sides of this contract: checking that the method is only called when its preconditions are indeed met, and checking that the body of the method is sufficient to make all postconditions true in every possible case. For the latter task, using the triple notation from Floyd-Hoare logic, the goal is to prove that:

$$\{P\}\ \text{largest(...);}\ \{R\}$$

The body of our method is composed of three statements executed in sequence: an assignment, a loop, and a return statement. Therefore, those three statements must together make it so that R holds, provided that P holds beforehand. We can represent this as the beginning of a proof tree, with our conclusion at the bottom. (Reading top to bottom, the horizontal rule can be read as 'therefore'; bottom to top, it can be understood as 'because'. Horizontal space can be read as 'and also'.)

$$\frac{\{P\}\ \text{int z = a[0]; for (...) ...; return z}\ \{R\}}{\{P\}\ \text{largest(...);}\ \{R\}}$$

We can refer to Floyd-Hoare logic rules for composition tailored for our programming language, in this case Java, and as a first step split up the method body in the hope we can prove properties about the smaller parts. If we can prove that executing the assignment when P is true makes some proposition Q come true, and that executing the remaining statements when Q is true makes R come true, we have proven our original goal:

$$\frac{\dfrac{\{P\}\ \text{int z = a[0]}\ \{Q\} \qquad \{Q\}\ \text{for (...) ...; return z}\ \{R\}}{\{P\}\ \text{int z = a[0]; for (...) ...; return z}\ \{R\}}}{\{P\}\ \text{largest(...);}\ \{R\}}$$

Our assignment statement uses an array reference, which (in Java as in other languages) is evaluated before the actual assignment can happen. We need to show that evaluating $a[0]$ makes some Q_1 come true, then executing the assignment at that point makes some Q_2 come true, and executing the loop and returning when Q_2 is true makes R come true.

$$\frac{\dfrac{\dfrac{\{P\}\ \text{a[0]}\ \{Q_1\} \qquad \{Q_1\}\ \text{int z = ...}\ \{Q_2\}}{\{P\}\ \text{int z = a[0]}\ \{Q_2\}} \qquad \{Q_2\}\ \text{for (...) ...; return z}\ \{R\}}{\{P\}\ \text{int z = a[0]; for (...) ...; return z}\ \{R\}}}{\{P\}\ \text{largest(...);}\ \{R\}}$$

In Java, for $a[0]$ to evaluate successfully, a must be a non-null array of at least one element. Luckily, we already required as much in our precondition: it is part of P.

We will leave our unfinished proof here for now. It should be clear that even for relatively small units of a program, such as the method in Figure 3.1, paper-based proofs quickly become unwieldy and difficult (or at least laborious) to check for correctness, especially as the Ps and Qs represent more and more complex logical formulae.

In addition to these *requires* and *ensures* clauses, our example has a *decreases* clause related to loops, which we will discuss next, as well as pre- and postconditions on permissions, which we will discuss in Section 3.1.2.

3.1.1 Loop variants and invariants

For algorithms containing loops, each loop will generally run a finite number of iterations that is in some way dependent on the algorithm's input. In the case of our maximum-finding function, the number of iterations is equal to the number of elements in the input array. To avoid formulating a different proof for every possible length of the input array, we instead define a *loop invariant*.

Loop invariants are first-order logic formula that must hold before the beginning of the loop and at the end of each iteration, regardless of the number of iterations. The loop body must ensure that the loop invariant becomes true.

Again referring to the example program in Figure 3.1, the latter three loop invariants specified have the following meaning:

1. The current index i must be between 0 and the length of the array, inclusive.
2. The maximum z must be larger or equal than all array elements before index i.
3. The maximum z must be present somewhere in the part of the array before index i.

Before the beginning of the loop, z has been set equal to the first element of the array, at index 0, and i is 1. All invariants hold: 1 is between 0 and the length of the array, inclusive, since we require that the length is at least 1; we have only seen one value, which is our highest so far. After each iteration, the invariants still hold: if looking at the next i meant that we have seen a new highest value, that has now been assigned to z. If we did not see a higher number, then z is still the highest among the numbers we have seen so far. In all these cases, z exists in the array. Finally, after the last iteration, the invariants still hold:

1. We have checked the entire input array: i is now equal to the length of the array.
2. z is larger or equal than all array elements before i, which is now all array elements.
3. z is present in the array.

The last statement of our method returns z. With that value as the result, we see that what we earlier called R now holds: z corresponds to the largest number in the array. We have shown our postcondition and proven the method correct for all non-empty arrays.

VerCors both checks and uses the loop invariant: it checks that the loop invariant is true before the loop and after each iteration, and it uses the fact that that means it is true after the loop has finished. At present, VerCors does not generate loop invariants; these must instead be given by the specification writer, and specifying invariants that are both provable and useful can be a challenge for complicated loops.

Finally, the *decreases* clause on the loop indicates a *loop variant*. Recall from the introduction to this chapter that VerCors defaults to showing partial correctness, proving that if the function terminates, its result is correct. We can additionally prove that the function *will* terminate, showing total correctness. To do this, we give a decreasing measure: i is a natural number that becomes larger every loop iteration, and therefore $a.length - i$ is a natural number that becomes smaller every iteration. Since a natural number can only become smaller a finite number of times, VerCors can use this to show that the loop runs for some finite number of iterations, and thus for some finite amount of time. Furthermore, since our *largest* method only contains this one loop, this also means that we can conclude (and VerCors can prove) that the entire method is finite.

3.1.2 Threads and separation logic

Classical Floyd-Hoare logic rules assume that there is a single thread of control throughout the program that is being reasoned about, and our small proof sketches above already assumed as much. However, many programs (including all Java programs) have multiple threads of control that run concurrently and share a single memory heap.

In practice, this means that another thread in possession of a reference to the same array a could change the integer elements of that array while our method was iterating over it. The *largest* method would now no longer always return the largest value. Instead, the result would depend on whether or not the other thread modified the array, in what way, and which operation on which thread happened to happen first: a race condition.

VerCors is based on *concurrent separation logic* [35] adding a system of fractional permissions [19] to explicitly represent ownership of potentially shared heap locations, like the array in our example. A fractional permission of 1, the maximum, corresponds to a write permission on some particular location in memory, while any fractional between 0 and 1 represents a read permission. A thread is granted write permission to an element on the heap when it creates that element itself, for example:

```
Object foo = new Object();
```

Our method *largest* does not create heap objects, but states in its precondition (*requires*) that it needs to receive, from its caller, read permission over all elements of array a, and in its postcondition (*ensures*) that it returns this read permission to its caller after it is done. It also specifies that the same read permission as a loop invariant.

This is enough for *largest* to iterate over the array while preventing the scenario described above. Because *largest* has read permission on a's elements, we can be certain that no other code has write permissions at the same time.

VerCors allows both splitting (using division, \) and merging (using addition, +) permissions. In the latter case, it must be proven that the permissions being merged are entirely distinct. Here, the separating conjunction is useful: $P * * Q$ specifies that P is true, Q is true, and the ownership of the different parts of P and Q is such that adding them together would not cause any permission to exceed 1.

By verifying that reading from and writing to the heap always happens with proper permissions and ownership rules are always followed, VerCors can prove that race conditions such as the one described above cannot occur. This makes it possible to verify and reason about concurrent programs, and to prove the functional correctness of functions like the *largest* function also in the presence of other threads.

In principle, specifying the permissions that different methods require is up to the specification writer. In some cases, where the ownership is straightforward, VerCors can also generate permission specifications automatically.

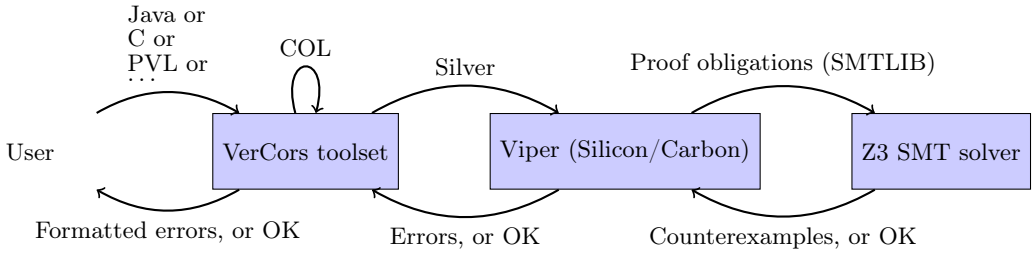

FIGURE 3.2: A high-level overview of the architecture of the VerCors toolset including the Viper back-end, showing the different intermediate representations in use (above) and the results that can be expected (below).

3.2 Overall design of VerCors

The overall architecture of VerCors has similarities to that of a compiler, and we will introduce the toolset in that manner, both because the reader may be familiar with compilers and to relate VerCors to existing work in compiler testing, which we have referred to in Chapter 2. The toolset is like a compiler in the sense that it parses annotated program code, rewrites the program in a number of passes, and then outputs a simpler representation of the program. Unlike other compilers, that simpler representation (the Silver intermediate language) is not intended to be executed. Instead, it is passed to a verifier. Figure 3.2 shows an overview of this arrangement.

3.2.1 Front-end

The VerCors toolset contains a wide variety of parsers for general-purpose programming languages including Java, C and C++, as well as for more domain-specific languages like OpenCL. To each of these languages, VerCors adds its own specification language, which is inspired by JML but is tailored to the specific host language: for instance, specifications for Java methods can refer to other methods using the familiar Java syntax. The intended workflow is that existing programs written in these languages can be annotated with their specifications and then verified, without changing the implementation. VerCors uses a syntax for annotations where specifications begin with '//@' or '/*@', a convention which is also used by OpenJML and other tools. This choice lets the same code be verified as well as compiled and executed: the annotations are understood as comments and ignored by the regular compiler.

For each language, the subset that can be verified with VerCors varies: for example, there is some support for Java generic types, but C++ templates are not yet supported. Generally, the front-end reports an error if unsupported features are used.

In addition to existing programming languages, VerCors supports the VerCors-specific Prototypal Verification Language (PVL), which is an object-oriented language intended for examples, tests, teaching, and research related to the toolset. New VerCors features are often exposed first in the PVL front-end. It is not intended for general programming: programs written in PVL can not be executed, they can only be verified.[1]

For the Java, PVL and C-like frontends, as well as for the specification syntax, the parser is generated by the ANTLR parser generator [36]. This has made it possible to reuse existing ANTLR grammars for those languages.

[1]There is some code for converting PVL to Java, which could then be executed, but this feature is not complete: it does not cover all of PVL.

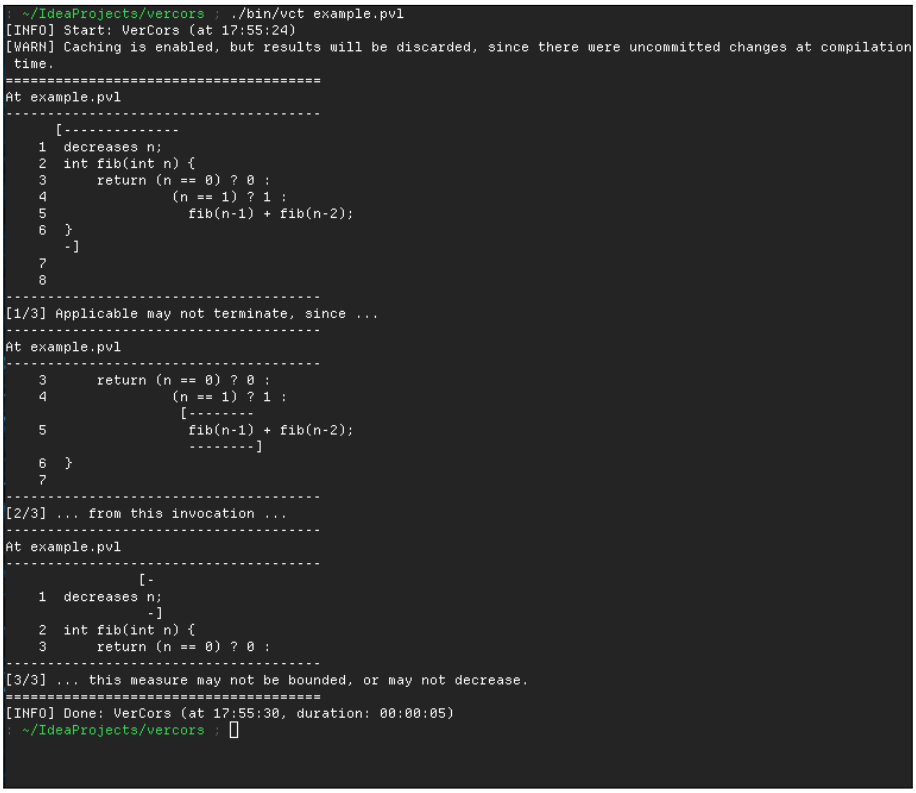

FIGURE 3.3: An imprecise specification for a recursive Fibonacci function fails to verify, and the failure is blamed on the relevant parts of the program.

3.2.2 Rewriting

The syntax trees that the different language front-ends derive are translated to Common Object Language (COL), a VerCors-internal abstract representation that forms a superset of all the supported language subsets, including the specification language.

From there, a large number of rewriting phases translate and simplify the annotated program. Constructs that only exist in some languages are flattened to a common representation: for instance, *PVLConstructor* is transformed into a *Constructor* by explicitly encoding the features specific to PVL separately. A later pass then transforms all *Constructor*s (originating from any language) into regular *Procedure*s by encoding what is special about constructors, and so on. Different ways to represent the same program semantics are also unified, for example transforming *for* loops into *while* loops and replacing *continue* statements with labeled *break* statements. In this way, later passes no longer need to know about *continue*.

3.2.3 Back-end

The last rewriting pass finally simplifies the COL to the language of the configured back-end. As illustrated in Figure 3.2, in the current version of VerCors the target is always either the Silicon verifier or the Carbon verifier. Both verifiers are developed as part of the Viper project [34] and both use the Silver intermediate verification language as their input language. The Silicon verifier performs symbolic execution, using the Z3 SMT solver [13] to answer logical queries. The Carbon verifier uses an approach based on verification condition generation via a translation to the Boogie intermediate language and the Boogie tool [4],

which in turn also uses the Z3 solver. Users of VerCors can choose between the two using a command-line flag. Historically, VerCors has used other back-ends as well [8].

An important feature of VerCors is that the different rewrites and translations keep track of the location of fragments in the original annotated program. If the program fails to verify, or an error occurs, this information and the message from the back-end can be 'translated backwards', and the failure can be *blamed* on a specific expression in the input. Such a blame can indicate that, for example, some expression may be false or some array index may exceed the array's bounds.

Most blames also include column and line numbers, which are then used to format error messages. This information is very useful in aiding VerCors users to either improve either their implementation so that it matches the specification, or perhaps improving the specification so that it matches the program.

An example of a blame is given in Figure 3.3. The *fib* function is a recursive Fibonacci function, and we would like to prove that its recursion is bounded. However, VerCors correctly tells us that the function does not always terminate. Adding a *requires* clause demanding that $n \geq 0$ is sufficient to show that *fib* is finite.

3.3 Categorizing errors

For reference throughout the remainder of this thesis, and to further contextualize our research, we want to distinguish some classes of issues: unsoundness issues, robustness issues (verifier crashes and back-end crashes), language limitations, and finally completeness issues.

3.3.1 Unsoundness and semantic errors

VerCors is designed to be a *sound* deductive verification tool: if a program verifies, it is guaranteed to be correct with regard to its specification, for every input and in every case. An unsoundness error therefore occurs when VerCors verifies that a given program is correct, while it actually is not: it contains a data race, commits a memory error, or computes some incorrect result in some (perhaps rare) situation.

Unsoundness errors can be introduced by a faulty rewriting phase, if the transformation is not behavior-preserving and the behavior of the transformed program no longer matches the original program. It can also occur if VerCors makes an assumption of the semantics or execution environment of the program that does not necessarily hold.

As an example of an unsoundness error, some versions of VerCors contained method contracts that specified that the C *malloc* function always returns a non-NULL result, but this is not guaranteed by the specification: *malloc* returns NULL if there is no virtual memory left to allocate, which is unlikely on many systems but not guaranteed. This issue since been repaired.[2] Another example, and a known limitation, is that VerCors always assumes that all integers have infinite range and do not overflow.[3]

Soundness errors are the most serious errors, since they breaks the trust placed in the tool: the strong guarantees that a verification tool can give depend on the tool being sound. It is also the most difficult category of error to detect automatically. A transformation can be shown to be *un*sound by showing that VerCors verifies a program that should not verify. However, automatically finding such programs is nontrivial.

[2] `https://github.com/utwente-fmt/vercors/pull/1239`
[3] `https://github.com/utwente-fmt/vercors/issues/396`

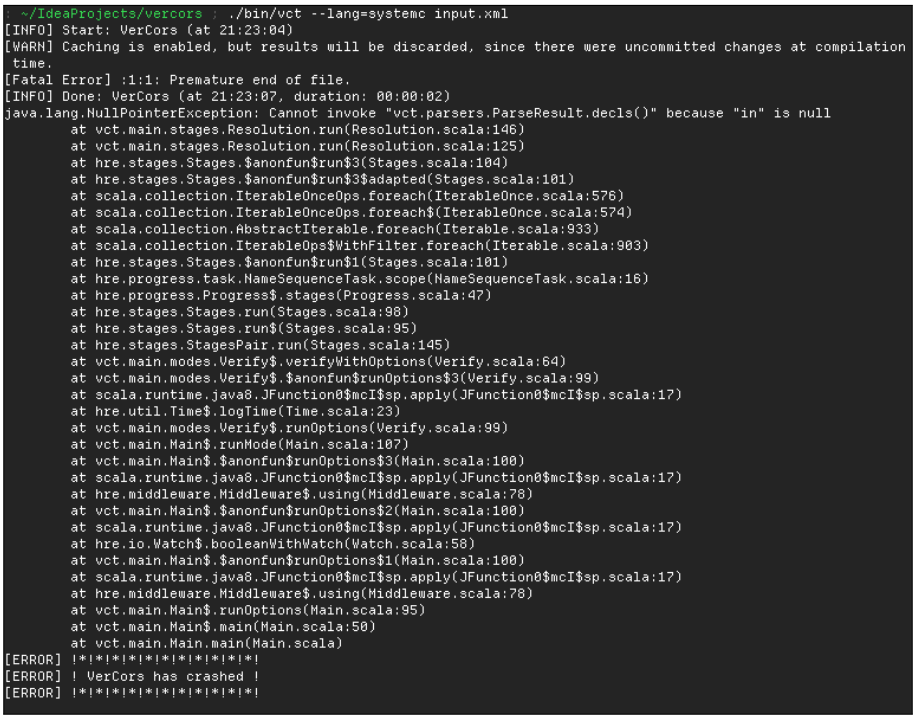

FIGURE 3.4: A fatal error when trying to parse a SystemC AST file has caused VerCors to throw a NullPointerException and crash; this has been reported as GitHub issue #1344.

3.3.2 Verifier crash

Verifier crashes are those that can be traced to bugs in VerCors itself, that is, errors occurring after the (ANTLR-generated) parser, but before the (Viper-provided) back-end.

An uncaught exception can cause VerCors to stop abruptly. Internal consistency checks, including type checks, are done throughout and after each rewrite pass, and errors from these checks are surfaced as exceptions, also crashing VerCors. The current version is configured to show a stack trace and a 'VerCors has crashed!' message when this occurs.

This category of error is less serious than unsoundness, especially because VerCors can in some cases use the 'blame' functionality to indicate which part of the user's input caused the crash, but it can still be frustrating for users, especially in cases where a blame is not available, in which cases users have to deduce from the stack trace what went wrong.

Since our research focuses on robustness, this is the primary error category that we look for. AValAnCHE catches all otherwise uncaught exceptions and reports them as crashes, and our approach is based on the principle that no input may cause VerCors to crash. This category of error makes a good test oracle: it is unambiguous whether some input leads to a crash or not.

3.3.3 Back-end crash

As described in Section 3.2.3, VerCors generates Silver intermediate language code that is consumed by a Carbon or Silicon back-end. In principle, our research treats these back-end crashes in the same way as verifier crashes. If a user program causes VerCors to generate input that does not conform to the Silver language definition or does not meet the Viper project's expectations, this is a bug in VerCors, with similar effects as in the previous

Language	Supported	Unsupported
C	OpenMP, structs, malloc/free, ...	Unions, sized integer types, ...
C++	SYCL, CUDA, namespaces, ...	Templates, STL, exceptions, ...
Java	Classes, interfaces, exceptions, arrays, ...	Containers, reflection, ...
LLVM IR	Functions, labels, instructions, ...	Magic wand operator, ...
OpenCL	Barriers, fences, local memory, ...	Images, samplers, rounding modes...
PVL	Axiomatic data types, parallel blocks, ...	–

TABLE 3.1: Programming languages for which VerCors front-end support exists, and some assorted examples of supported and unsupported features.

section. Of course, a back-end crash could also be caused by a problem in one of the Viper verifiers themselves, but we have not encountered such a problem.

3.3.4 Language limitation

The eventual goal for VerCors is to support entire programming languages, not just subsets of programming languages [7]. However, at the time of writing none of the existing languages are supported in their entirety, with the front-end for PVL being the most complete, owing to PVL being specifically designed for VerCors and to its role as the language where new features are often added first.

For cases where a feature is explicitly unsupported, a message like "This construct is syntactically valid, but not supported by VerCors" is logged, requiring the user to find another way to express their program. While these messages are technically error messages, they are not crashes, and it would not be very useful to treat them as new bugs in VerCors. The cases where they occur are already known and can be found by looking for the '??()' function that raises these errors. We therefore leave these cases outside of the scope of our research. An overview with a selection of example features that are supported and unsupported, respectively, for each frontend is given in Table 3.1.

3.3.5 Non-errors

The VerCors toolset is intended to be sound, but it is not *complete*: a program that is correct does not necessarily verify. The program may just need some additional annotations, such as the loop invariants in the listing in Figure 3.1, or the program may be correct in some way that is difficult to prove.

An active area of research, as surveyed by Lathouwers et al. in [27], is to automatically generate such annotations where possible. As mentioned in Section 3.1.2, VerCors can already generate permission annotations for some programs that have straightforward memory ownership semantics, and in general, approaches exist in the literature to automatically suggest suitable invariants and pre- and postconditions for a given program. For example, the Daikon [16] system can dynamically infer useful invariants that are likely to hold, by running a program and continuously observing the values that different variables take. This reduces the amount of invariants that have to be specified by hand. However, for our purposes, it is not an error in VerCors if a program does not verify while it 'should'.

Relatedly, it is also possible to write or specify a program in such a way that either the rewriting (inside VerCors) or proving (inside the back-end) takes a long time. Again, this can be legitimate – it may just be difficult to prove that the program is correct – and so we also do not treat this as an error. It can be argued that it should not be possible to write an input program that causes VerCors to run forever, but we do not try to distinguish

this case. As part of our experimentation, we did find and report[4] one case of VerCors 'getting stuck': recursively self-referential generic Java classes would cause VerCors to not terminate. Since this issue was not found using fuzzing, it is not included in the list of bugs in appendix A.

Finally, to avoid all doubt: VerCors will report a message when it is asked to verify programs that have syntactical or semantic errors or that do not verify, indicating the faulty or unprovable part of the input program if possible. Figure 3.3 shows such a message. This is an error from the user's point of view, but not for our research: from our perspective, the issue has been successfully handled by telling the user to improve their program.

3.4 Existing test suite

Currently, testing VerCors is primarily done by means of a set of automated integration tests. Each integration test attempts to validate an example program or snippet, then confirms that the result matches the expectation, which can either be pass (the program verifies), fail (it does not verify), or a specific error code. The integration tests are taken from a number of sources:

- Example programs written specifically for the test suite that exercise and demonstrate specific concepts or combinations of features.
- Programs from VerCors-related publications and case studies.
- Solutions to previous iterations of the VerifyThis verification competition [45].
- Regression tests showing that an earlier issue no longer occurs.

The test harness is based on ScalaTest[5], the de-facto standard testing tool in the Scala language ecosystem. GitHub Actions are used to run the tests on GitHub's cloud infrastructure on every commit and pull request, and pull requests that do not pass tests are not merged. The test suite is normally executed on the Linux platform. A subset of basic tests are also executed on Mac and Windows platforms, also using GitHub Actions.

At present, the project does not track a test coverage metric. Of the supported languages, the Java and PVL front-ends are the most extensively tested.

Some unit tests, covering and checking a small portion of the VerCors codebase, are also included in the test suite.

[4] `https://github.com/utwente-fmt/vercors/issues/1279`
[5] `https://www.scalatest.org/`

Chapter 4

Performance metrics

To answer our Research Question 1.3, we have compared the relative performance of a number of different fuzzing strategies when specifically applied to finding robustness issues in the VerCors verification toolset. In this chapter, we introduce the metric that we have used for this comparison, and describe some practicalities around our benchmark setup.

4.1 Coverage over time

As a metric to compare the different approaches described in Section 2.2, we use code coverage over time. This is also the metric that is most widely used in the literature. Often, a number of new findings discovered using the tool is also given, as we have done in our abstract. There does not seem to be a consensus for how much time is reasonable:

- In [14], code coverage (there reported as percentages of lines and branches covered) is used to compare the new fuzz-d and DafnyFuzz approaches to each other, to the earlier XDsmith tool, and to Dafny's existing set of unit and integration tests. Both tools ran for 8 hours each.
- Describing the PolyGlot fuzzer, Chen et al. in [11] compare the performance of their approach against other fuzzers by measuring the edge coverage achieved over 24 hours.
- When comparing different modes of running their NAUTILUS fuzzer, Aschermann et al. [3] use the percentage of new branches covered after 20 times 24 hours as a metric.

In our specific case, we measure the number of instrumentation points that were covered in a fixed amount of time, on the same environment, and on the same machine. We found that in our case, a time limit of 5 minutes was already sufficient to see a marked performance difference between the different approaches.

AValAnCHE uses the Jazzer [12] fuzzer for code coverage instrumentation, which in turn uses the JaCoCo [23] code coverage library. JaCoCo works by instrumenting the Java Virtual Machine (JVM) bytecode of Java and Scala classes as they are loaded, inserting bytecode into a number of points in each method. Each instrumentation point (or 'probe', or 'coverage counter'), when executed, sets its own index to true in a large per-class array of boolean values. Summing the number of set booleans for all classes when execution has finished then gives the total number of covered probes.

JaCoCo, and so Jazzer, applies a number of optimizations to reduce the number of probes required. This reduces the run-time and code size overhead of instrumentation: the authors note that instrumented classes are 'about 30%' larger and 'typically less than 10%'

slower. Rather than instrumenting every instruction, the optimized logic instruments every *return* or *throws* instruction, as well as every *jump* instruction. Straight-line code does not need probes: if we arrive at a *return* at the end of a straight-line Java method, we know that the other lines must have been executed.[1]

These optimizations do mean that the number of probes covered does not directly correspond to a number of JVM instructions, JVM basic blocks, branches, or lines of code. Those metrics can be recovered from the booleans in the array, and this is commonly done for human-readable coverage reports, the generation of which is the usual reason JaCoCo is used. However, this accounting and report generation is a relatively expensive operation, and so we instead use the counts directly: the number of probe points covered ('hit') and the number of points instrumented. The former is then a measure of how many paths through the code have been tried, and the latter a measure of how many classes have been loaded and thus the variety of VerCors features we have exercised so far, even though it doesn't strictly equal either of those measures.

The version of VerCors that we are measuring is based on the most recently released version (2.3.0). If this version were entirely instrumented, this would require 358.985 probes in total. This includes both probes in hand-written code and in machine-generated code, for example classes based on AST node definitions or ANTLR grammars. Probes in code that is dynamically unreachable (for instance, implementations of the *toString* method that are not actually used, or code related to front-ends that we are not currently testing) are also included in this count.[2]

4.2 Alternative: number of issues

The 'code coverage over time' metric is intended as a proxy for how well a specific strategy exercises different behaviors of the system under test (VerCors, in this case). It is, however, imperfect as an indirection: we are actually interested in finding crashing bugs, rather than increasing test coverage *per se*. Put differently, our metric requires the reader to agree that executing code (and perhaps finding a crash) is better than not executing it, and that covering more code is always better, even if we do not find a crash, and regardless of which part of the code we are covering.

Our initial intuition was to directly use the bug-finding capacity of the different approaches, by simply counting the number of bugs discovered by each fuzzing approach. However, to correctly compare the approaches amongst each other, all approaches should be given fair chances, or at least the chance to run for the same (long) duration. Guaranteeing this was difficult in practice:

- We ran early versions of our implementation for each approach during development, and reported findings that surfaced from these runs. This would have proven difficult to make fair: should running a grammar-based fuzzer for a few minutes 'count' those minutes if it turns out after those minutes that the grammar was actually incorrect and should be fixed? Does a bug found by a fuzzer using a faulty grammar 'count' for the number of bugs found?

- Running the approaches that were quick to develop, but that we did not expect to yield any results, for the same duration as the others would have been wasteful. In

[1] More details and diagrams of instrumented bytecode can be found in the JaCoCo documentation: `https://www.jacoco.org/jacoco/trunk/doc/flow.html`

[2] Since instrumentation happens lazily, at class load time, the 'Total probes' number that is shown in the AValAnCHE web interface will be lower than this number.

particular, this would refer to blind fuzzing. We were surprised that blind fuzzing found any issue at all: the crash that became GitHub issue 1330 (names with many trailing digits crash VerCors, see appendix A) was discovered using blind fuzzing.

- A challenge arose as soon as the first crashing bug found by an early version of our tool was fixed: now that the bug was fixed, and could no longer be found by the approach that first discovered it, should its 'score' be negatively affected on the new version? Taking the definition strictly, it would be. Then, the historical score of each approach over time, as new versions of VerCors were released, would depend on project decisions on which crashing bugs to prioritize. In the hypothetical case that no crashing bugs in VerCors would remain and perfect robustness would be achieved, all scores would become zero. This did not seem reasonable.

4.3 Alternative: mutation score

Besides metrics based directly on code coverage, it is possible to use *mutation testing* [26] to measure the thoroughness of test suites. Mutation testing is most commonly applied with developer-written test suites, but the technique can also be used with test suites generated by a fuzzer-like tool. In summary, mutation testing follows a process similar to the following:

1. It is assumed that the unmodified system under test passes the test suite.
2. The source code of the system under test is mutated, that is, bugs are automatically introduced. Common mutations are to flip boolean literals from *true* to *false*, invert conditionals and logical operators, and to replace string literals. The test suite is not changed.
3. The test suite is then executed to test the mutated system.
4. If the test suite now fails, correctly detecting that the system has been mutated, the mutant is said to be 'killed'; if the test suite is not sensitive enough to find that the program has been broken, the mutant 'survives'.
5. The *mutation score* is the fraction of mutants killed.

In [14], the authors use the mutation score as a secondary metric, also comparing the mutation score of the different fuzzing-based approaches to that of the developer-written test suite. To make the comparison fair, fuzzer-generated tests are run for the same duration as the existing test suite.

We decided against using mutation score as a metric because of our focus on robustness issues. As described in Section 3.3, we are interested specifically in verifier and back-end crashes, and do not look for soundness or precision errors. With mutation testing, this would mean that we would be measuring how well each fuzzing approach can detect crashing bugs *that have been introduced by mutation*, and using this result as a proxy for how well it can detect crashing bugs that are already present. We thus again would need an assumption, namely that the 'real' crashing bugs are in some sense similar to those introduced by the mutator.

Chapter 5

The AValAnCHE tool

As part of our research, we have developed the AValAnCHE tool, which is intended both as a vehicle to compare the performance of a number of fuzzing strategies when applied to the VerCors codebase, and (as a side benefit to our research but the main benefit to the VerCors project) directly as a tool to aid the VerCors contributors in their work.[1]

The tool continuously runs VerCors on test inputs from a fuzzer, and watches for crashes, which are collated to remove duplicates. Newly found crashes are sent by email, for example to a developer mailing list. A web interface to watch the tool's progress as it tries different inputs is also offered, which shows both statistical information (number of inputs attempted so far, code coverage statistics, and so on) as well as a list of findings that is automatically updated as more inputs are tried and more crashes are discovered.

AValAnCHE is designed to be generic in the fuzzing strategy that is employed, which is the main direction of its extensibility. The intention is that, if a better strategy is found than the strategies compared in the present research, this can be integrated without having to modify the tool itself. Beyond this, the tool has few dependencies and additional requirements beyond those needed to run VerCors. Instructions to run each of the strategies below as a Docker container are included with the tool.

5.1 Strategies implemented

The current version of AValAnCHE can use either coverage-guided fuzzing, grammar-based generation, grammar-based generation with coverage feedback, and grammar-based generation of a verifiable subset. The different strategies we are aiming to compare are based on those described in Section 2.2, and build on top of earlier work mentioned in that section. In this chapter, we describe some specifics of the AValAnCHE implementation and our experiences in using the strategies with VerCors specifically.

In addition to the strategies mentioned, AValAnCHE integrates the Radamsa[2] general-purpose blind fuzzer. This integration is intended to demonstrate how to extend AValAnCHE and as a benchmark for the interface between the generator and AValAnCHE itself; it is not a contender in our performance comparisons. In all cases, AValAnCHE runs VerCors with the Viper back-end disabled: for AValAnCHE, we are not interested in bugs in Viper's Carbon or Silicon verifiers, and so skipping those verification steps increases performance and so the number of attempts that can be done in a given amount of time.

[1] The source code and documentation for AValAnCHE is publicly available at `https://github.com/wandernauta/vercors`, and we are planning on working with the VerCors group to integrate with the main VerCors repository at `https://github.com/utwente-fmt/vercors`.

[2] Radamsa is available at `https://gitlab.com/akihe/radamsa`.

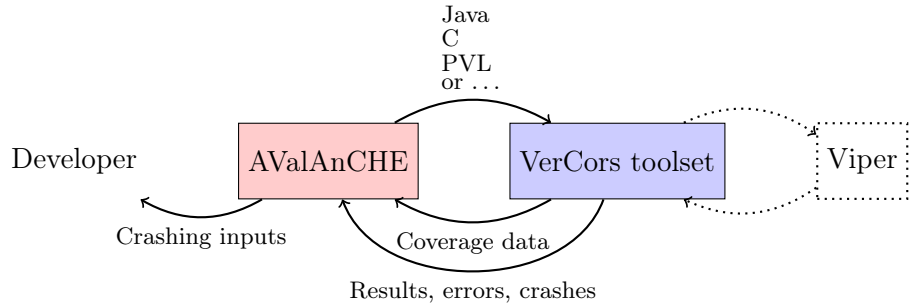

FIGURE 5.1: AValAnCHE runs unattended whenever otherwise unused compute resources are available, notifying developers when it finds an input that crashes VerCors. The Viper back-end is skipped for performance reasons.

5.1.1 Coverage-guided fuzzing

AValAnCHE supports coverage-guided fuzzing (without grammars) through the Jazzer fuzzer, [12] which is written in Java. This approach is not specific to compilers, and is intended as a baseline to compare the other approaches against.

Jazzer is an in-process fuzzer that instruments classes on the fly and as-needed, using logic from the JaCoCo[3] code coverage library and mutation logic from LLVM's libFuzzer[4]. As we have described in Section 4.1, the instrumentation adds *coverage points* to measure which parts of the system under test are executed ('covered') by tests.

As a coverage-guided fuzzer, Jazzer uses the coverage information to discover which bytes in the input cause the execution path (and thus the behavior) of the program being tested to change, and which do not. The mutations from libFuzzer operate on byte array inputs: bytes can be shuffled, erased, changed and inserted, bits can be flipped, and a 'crossover' operation exists that can combine multiple inputs. Besides arrays of bytes, Jazzer can also generate random Java data structures, but this feature is not used by AValAnCHE.

In the AValAnCHE documentation, this approach is referred to as 'Direct', since it directly feeds Jazzer's generated input to VerCors.

As a practical matter, the AValAnCHE tool uses Jazzer's code coverage instrumentation to measure fuzzing performance in every mode, not just when used with fuzzers that are coverage-guided. This supports measurement of the performance metric we described in Chapter 4.

5.1.2 Grammar-based generation

We use the Grammarinator fuzzer by Hodován et al. [22], already introduced in Section 2.3, as the base of the grammar-based generation strategy. The goal is to generate inputs that randomly cover the entire grammar of the input language. Grammarinator takes an ANTLR [36] grammar as input, and generates a fuzzer based on this grammar. Both the tool and the generated fuzzer are written in Python.

We have ported the PVL, C, C++, and Java ANTLR grammars from VerCors to Grammarinator, to be used by AValAnCHE. The grammar descriptions were mostly usable as-is, but some smaller changes to the different VerCors ANTLR front-end grammars were required to use them with the Grammarinator tool:

[3] `https://www.eclemma.org/jacoco/`
[4] `https://llvm.org/docs/LibFuzzer.html`

- VerCors splits lexer and parser grammars into seperate files, as well as seperating the grammars for the specification and host language syntax (Java, C++, and so on). This structure is not yet understood by Grammarinator, and so needed to be flattened.
- Support for comments was removed from all front-end languages, since parser states are not well-supported. In theory, this would mean that we cannot find bugs related to the handling of grammatically correct comments, but we expect such bugs to be unlikely.

All inputs generated by this method are grammatically correct, by construction, and therefore pass the parsing stage. However, the grammar-based fuzzer is not aware of the semantics of the programs it generates, the scopes of variables, or the types of expressions. Therefore, in almost all cases (except the ones that cause a crash), the result from VerCors is an error in a resolution or rewriting phase: a reference to a class or variable that does not exist, a type error, or a message that some statement or expression is not allowed in some context. For the C and C++ front-ends, the language limitation error (see Section 3.3.4) is also very common, since the grammar is not limited to only those features that are understood by VerCors.

5.1.3 Grammar-based generation with coverage feedback

Since we wanted to be able to reuse the infrastructure for grammar-based generation described above, we use the same Grammarinator-generated grammars for this strategy, but add coverage feedback by using the Jazzer-generated input as a seed.

While Jazzer is under the impression that it is generating inputs for VerCors, it is actually generating inputs for the grammar-based fuzzer described in the previous section. Normally, the fuzzer makes entirely random choices on which grammar production to generate, which letter to use in an identifier, or how many repetitions of a certain element should be generated. In the seeded mode, the choices are instead made based on the seed input. If no seed input remains, the generated program is ended as quickly as possible while still remaining within the grammatical constraints of the language being generated.

The process is both deterministic (given the same seed input, the same program is generated) and predictable (changing one byte in the Jazzer input changes one choice in the output). As in the previous section, the construction ensures that test inputs generated by this strategy are always grammatically correct, and like in the previous section, the inputs therefore are generally rejected by VerCors in the resolving stage.

Some example outputs of this process are given in Figure 5.2. The first example verifies; the second example fails because *return* is grammatically allowed but semantically forbidden inside *vesuv_ entry*. The third and fourth example fail in a resolving phase because no class 'a' exists.

Example seed	Generated PVL
`_`	`vesuv_entry { }`
`_A`	`vesuv_entry { return; }`
`_A3`	`vesuv_entry { if (new a()) return; }`
`_A3x`	`vesuv_entry { if (with return; new a()) return; }`

FIGURE 5.2: Generating grammatically correct PVL code from random seed bytes. The result is deterministic for a given version of the grammar; longer seeds give longer programs.

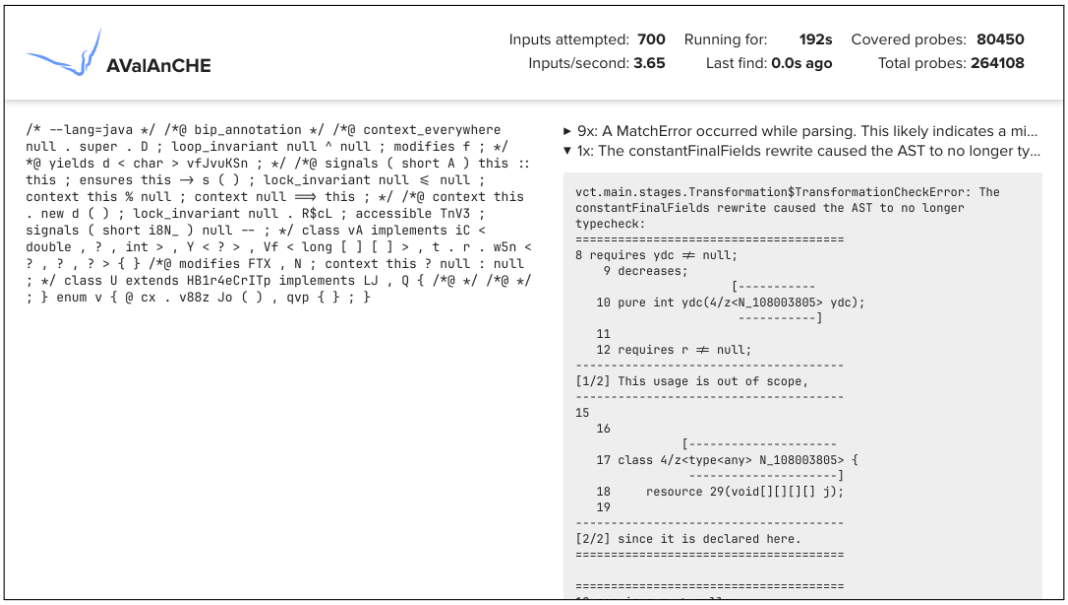

FIGURE 5.3: A screenshot of the AValAnCHE web interface. The most recently attempted input is shown on the left (here: a Java program generated by Grammarinator). On the right, the results found so far are shown: 10 crashes have been found in 700 inputs, which the collation process has reduced to two distinct bugs.

5.1.4 Grammar-based generation of a verifiable subset

This approach, which has also been called test generation elsewhere in this thesis, uses the Xsmith [20] tool already introduced in Section 2.4. The Xsmith tool is written in, and expects its grammars to be written in, the Racket language. There is no way of converting ANTLR grammars into Xsmith grammars, and the grammars used by Xsmith are more involved than with the earlier approaches, since they also partially describe the type system of the programming language being generated.

The Xsmith-based generator at present only generates PVL, as there are no grammars for the other programming languages supported by VerCors. However, in contrast to the Grammarinator-based approach above, it generates PVL that can not only be successfully parsed but that can also be type-checked and could successfully be passed to a backend verifier, if one was enabled. Similarly, the generator never generates an unknown variable name, since it knows how to insert variable declarations with matching names into the functions it generates.

The Xsmith library works by randomly generating an abstract syntax tree (AST) that represents a program but also has 'holes'. These holes are then recursively filled, by generating a random concrete AST node that would syntactically fit the hole. At this point, type checking occurs: if the generated AST does not conform to the type rules described by the grammar, the newly generated node is discarded and another node is generated to fill the same hole. This approach means that as the type rules in the grammar become more specific, and more closely represent the type system of the language being generated, more and more attempted nodes need to be discarded.

Very specific type rules thus make the generated programs more varied, using more of the functionality of VerCors and covering more of its codebase, but generation becomes still slower, at some point becoming the bottleneck in the fuzz testing process: more CPU time is spent in generation of inputs than in verification.

For AValAnCHE, we use the built-in HTTP server feature of Xsmith to avoid starting the generation program for every new input, which avoids some overhead. However, even with that optimization, test case generation is the most complex method supported by the tool, and by far the slowest.

5.2 Collating crashes

In Section 2.1, we discussed that if a crashing input is found by a fuzzer, it is likely that an infinite number of potential inputs would trigger what is essentially the same crash, or at least the same bug, in VerCors. To avoid overloading the mailing list with duplicate reports, AValAnCHE attempts to combine crashing inputs by inspecting the stack trace of the exception that caused the crash. For each stack frame, a hash is computed over the different fields, especially the class name, method name, file name, and line number. Crashes that hash to the same value are treated as duplicates. For exceptions that have a 'cause', only the original cause is inspected in this way. The exception message, human-readable error logging, and other values are ignored in the collation process, since these will likely be different for each crashing input.

AValAnCHE does not attempt to combine crashes across different versions of VerCors or between different runs of the tool. For instance, if a version of VerCors is released that does not fix a particular crash, and AValAnCHE finds essentially the same crash in the new version, it will be reported again. This could be partially resolved in a future version by having AValAnCHE record the crashes it has reported to persistent storage. However this would not be a complete solution: since the tool would still combine crashes by their stack trace, which includes information like line numbers, it seems likely that any change to the VerCors source code would cause crashes recorded with an old version to no longer match crashes recorded with the new version, defeating the purpose.

5.3 Minimizing test cases

The AValAnCHE tool's output takes the form of an input that triggers a crash in VerCors. For the tool's user to go from that input to finding the cause of the crash, it is useful if the crashing input is as short as possible, with no code that is irrelevant to the crash; in other words, that it it minimized. We also briefly touched on this in Section 2.1.

The current version of the AValAnCHE tool does not minimize crashing test cases automatically. For the issues we have found so far, all of which are listed in appendix A, we have had good success minimizing the test cases by hand, guided by VerCors's error messages and 'blame' feature. The test cases generated by AValAnCHE are relatively short, and often the stack trace from the crash already gives a good idea of a likely cause. If not, a simple manual approach that has worked well is to cut the program in half lengthwise, remove one of the halves, and check if VerCors still crashes with the same stack trace.

In addition, we have also successfully used the Shrinkray test-case reducer by David R. MacIver[5] to reduce crashes found by AValAnCHE. Shrinkray is not, and does not need to be, aware of the structure of the test cases it is minimizing: it works by attempting to remove bytes from the input file and rerunning an 'interestingness test' to see whether the same crash reoccurs. The automated approach works best for crashes that occur early in the verification process (for example, problems in one of VerCors's language front-ends)

[5] `https://github.com/DRMacIver/shrinkray`

and less well if every run of VerCors (and so every interestingness test) takes a long time to execute. In a sense, it is an automation of the manual process described above.

We are planning on extending AValAnCHE so that it can generate an interestingness test for each crash, which can then be used with Shrinkray on a developer's own machine. This would allow Shrinkray to take advantage of many-core CPUs common in such hardware, and would also mean the tool runs with human oversight that can stop the shrinking process once the input has been sufficiently reduced.

5.4 Integration

The AValAnCHE tool is intended to be integrated into the VerCors development process. Practically, it is written in Scala (like the rest of VerCors) and offers a web interface to view the current status of the fuzzing campaign. The web interface automatically updates with new inputs, findings, and statistics as they become available, using the built-in EventStream API. A screenshot of the web interface is shown in Figure 5.3.

When a new crashing input is discovered that is unlike a known crash, AValAnCHE automatically sends a notification by email to a mailing list. The notification contains the generated input as an attachment, as well as the crash backtrace, and version information, to make triaging the issue as straightforward as possible.

The different generators (Jazzer, Radamsa, Grammarinator and Xsmith) use entirely different technology stacks, and we anticipate that other generators may be added in the future that use still other programming languages and runtime environments. To this end, each generator is shipped as its own Docker container, which packages GCC, Python and Racket as needed in each specific case. The generators are self-contained in that they do not depend on AValAnCHE; they offer a standardized HTTP-based API on a known port, which AValAnCHE then consumes.

The intention is that AValAnCHE will run on compute resources that would otherwise be unused and that are 'spinning anyway', similar to Google's OSS-Fuzz effort [2], or else on inexpensive commodity (virtual) hardware during off-peak hours, when there is no other demand. This way, there is no great loss if the robustness of VerCors is improved to such an extent that AValAnCHE no longer finds bugs.

5.5 Using AValAnCHE with other tools

Rather than a generic fuzzing solution, AValAnCHE is tailored to VerCors specifically: it consists of the tool itself, which is integrated into VerCors, and the different grammar-based generators which have been provided with grammars that describe the VerCors-specific Prototypal Verification Language or else a common programming language but with VerCors-specific specification (annotation) syntax.

AValAnCHE could also be integrated with verification tools other than VerCors, as long as those tools are also written in Scala or some other JVM-based programming language. The following changes to the tool would be required:

- Updates to grammars, as described above, in the likely case that the specification syntax is different.
- An updated selection of which packages to include and exclude. For VerCors, we include the *vct* and *hre* packages, which corresponds to the entirety of VerCors's Scala code but not, for instance, any back-end code from Carbon or Silicon.

- Some entry point would need to be chosen that can be called and then reliably raises an exception to indicate a crash. In AValAnCHE, this is just the `Main` command-line entry point to VerCors.

If AValAnCHE were to be used with a verification tool written in a non-JVM language, there would also need to be some new way for AValAnCHE to measure the achieved code coverage; as used by AValAnCHE, Jazzer will only compute coverage over Scala/Java code. However, at this point, not much of the tool's existing code would actually be reused.

Chapter 6

Measurements and results

Having introduced our context and our tool in earlier chapters, our aim in this chapter is twofold.

First, we aim to present the results of comparing the different fuzzing strategies that were introduced in Chapter 2 and Chapter 5 as they apply to VerCors, ranking their relative performance in service of Research Question 1.3. For this, we take advantage of the AValAnCHE tool introduced in the previous chapter and the coverage-over-time metric explained in more detail in Chapter 4. In the process, we will also highlight a few of the issues we have found in VerCors using the different approaches.

Second, we describe our experience using the different fuzzing strategies on a number of tools outside the VerCors toolset. Since the AValAnCHE tool is specifically intended to be used with VerCors, as we have described in the previous chapter, the same metric can not be used here. Instead, we look at whether we can find not-yet-reported issues at all using the approaches we are comparing for VerCors, answering our Research Question 2.

6.1 AValAnCHE applied to VerCors

To visualize the performance of the different approaches, we will use a number of scatter plots where the x axis represents time and the y axis represents the number of coverage counters. Each plot point corresponds to an execution of VerCors where some previously uncovered code path was either instrumented (blue crosses) or covered (red pluses), the latter being our chosen performance metric. We will give the results for each of the different approaches individually, and then give a combined result in figure 6.7.

All measurements were done on a Lenovo ThinkPad P16v Gen 2, Intel Core Ultra 7 155H (x86_64), running NixOS, with VerCors development version 2bd3bcaed, which is based on released version 2.2.0. The Linux CPU performance governor was set to 'performance', clocked at a maximum of 4.8 GHz.

We avoid exercising code that is not part of VerCors by running the tool with the `--skip-backend` flag. This makes it so that inputs that can be succesfully parsed, type-checked and rewritten by VerCors (primarily those from the XSmith generation strategy) are not passed to the Silver backend. Of course, this does mean that bugs that would only appear in that stage can not be discovered.

As we have described in Chapter 4, the performance metric has been selected in such a way that it does not depend on specific issues being either found or fixed. A complete list of the issues discovered using AValAnCHE is given in appendix A.

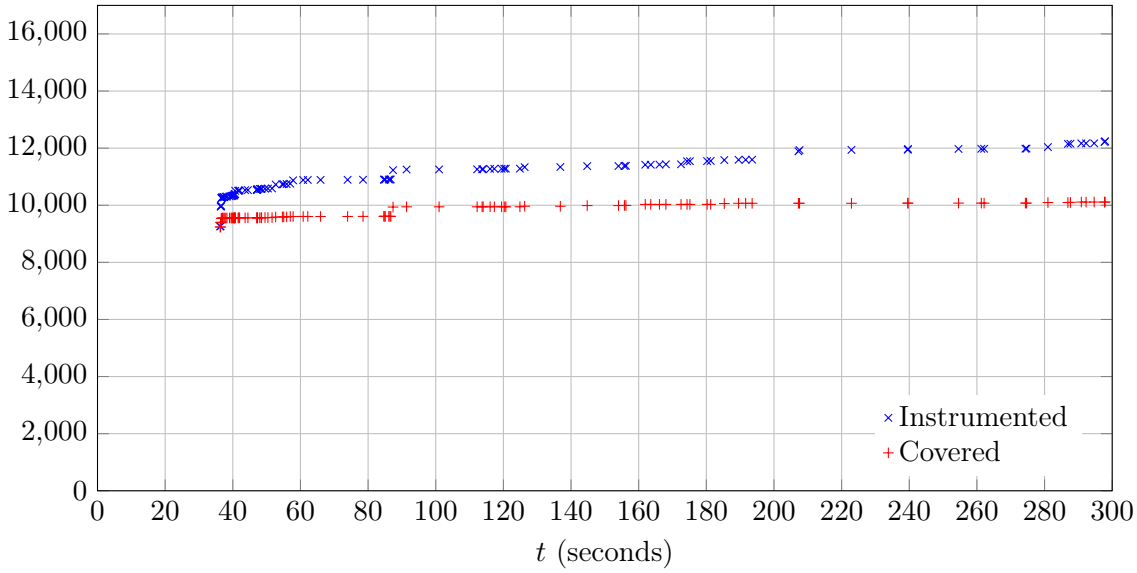

FIGURE 6.1: Coverage-guided fuzzing without a grammar or corpus.

6.1.1 Coverage-guided fuzzing

In this measurement, which uses the AValAnCHE 'Direct' mode, the generated inputs from the Jazzer coverage-guided fuzzer are fed directly to VerCors. Jazzer is used here as a fuzzer that generates inputs solely based on coverage information: the fuzzer is run without a corpus of example PVL inputs and so starts 'from scratch'. This measurement is intended as a base of comparison against the approaches in the following sections, each of which require more information about the expected shape of the inputs, expressed as a grammar.

The plot in Figure 6.1 shows the number of coverage counters instrumented and covered every time either number changes. Time is measured from the beginning of the entire process, and therefore includes a startup delay as different Scala and Java classes are instrumented. This startup delay exists for all approaches.

The coverage achieved using this method is low and barely grows over time, indicating that the approach is not able to find interesting inputs in the time allotted. The number of instrumented coverage counters also does not grow, implying that a large part of the VerCors codebase is not exercised by the inputs being generated.

Manual inspection of some of the generated inputs shows that the fuzzer primarily finds inputs that are immediately rejected by the ANTLR-based PVL parser. It appears that different syntactically invalid inputs follow different code paths in the parser, to format different error messages. This variation leads the fuzzer to treat these inputs as interesting even though it is not interesting for our purposes.

The fuzzer also finds semantically empty inputs, that is, inputs consisting entirely of whitespace or comments. These inputs succeed, which causes the code path corresponding to successful verification of an empty program to be covered. This is a much better result than an input that causes a syntax error, but an empty program still does not exercise much of the VerCors codebase.

In theory, and given enough time, a coverage-guided fuzzer can 'learn' or recover part of the parser grammar from coverage information, as was shown with the Grimoire tool in [5], but this did not succeed in our case in the number of attempts that were done.

No bugs were found in VerCors using this approach.

6.1.2 Grammar-based fuzzing (PVL)

Going from coverage-guided fuzzing to grammar-based fuzzing gives a large improvement, as can be seen in Figure 6.2. (Note the 10-fold increase in the y axis from before.) Again, we plot the number of coverage counters instrumented and covered. The inputs generated are varied enough to find and instrument new parts of the VerCors codebase, reflected in the growth of the number of points instrumented. The number of paths actually covered also grows, albeit more slowly.

Some examples of problems found using this approach include GitHub issue 1273 and issue 1293, example inputs for which can be found in appendix A.

6.1.3 Grammar-based fuzzing (PVL, Java, C++, C)

This approach corresponds to grammar-based generation of a syntactically valid input in a random choice of either PVL, Java, C++ or C. No attempt is made to limit the inputs to those language features that are supported by VerCors (see Section 3.3.4), so many generated inputs return that the input is "syntactically valid, but not supported by VerCors" when verified. Nevertheless, code that is specific to those front-ends is now also instrumented and covered. The performance of this approach is plotted in Figure 6.3.

It can be seen that the amount of points instrumented and covered is much improved compared to the previous figure, and indeed this approach performs the best in our chosen metric among all the approaches tried, as shown in Figure 6.7.

6.1.4 Grammar-based fuzzing, coverage feedback (PVL)

Performance in this metric is markedly worse than in Figure 6.2; a number of attempts, and therefore testing time, is spent on trying to find which parts of the input change VerCors's behavior. However, not much information is gained this way: *every* byte in the seed input can drastically change the shape of the program being generated, but whether it does depends on the preceding bytes, not on the value of each individual byte. The attempts by the fuzzer to confirm this therefore amount to trying a largely same-shaped input again. This time is effectively wasted.

It is not entirely clear to us whether the worse performance is inherent to our strategy of generating based on a seed, or whether it is a problem with our implementation.

6.1.5 Grammar-based fuzzing, coverage feedback (PVL, Java, C++, C)

As can be seen in Figure 6.5, generating a larger variety of input languages is an improvement over generating only PVL input, also in the coverage-guided case. However, the results are still clearly worse than running the fuzzer without coverage guidance and generating random inputs, which are apparently more varied and therefore trigger more unique behaviors, and therefore cover more of the codebase.

6.1.6 Grammar-based generation of a verifiable subset

Our Xsmith-based generation of a verifiable subset of PVL start off strong, as can be seen in Figure 6.6. Because we generate only programs that can be parsed and resolved, the initial test (at $t = 40$) already exercises a significant part of the PVL front-end and a number of rewrite phases, which together cover a sizable chunk of the codebase. In particular, rewrite phases after variable resolution are covered with each input. Where generated programs from the earlier approaches generally trigger 'no such variable' or 'no such class' errors, this

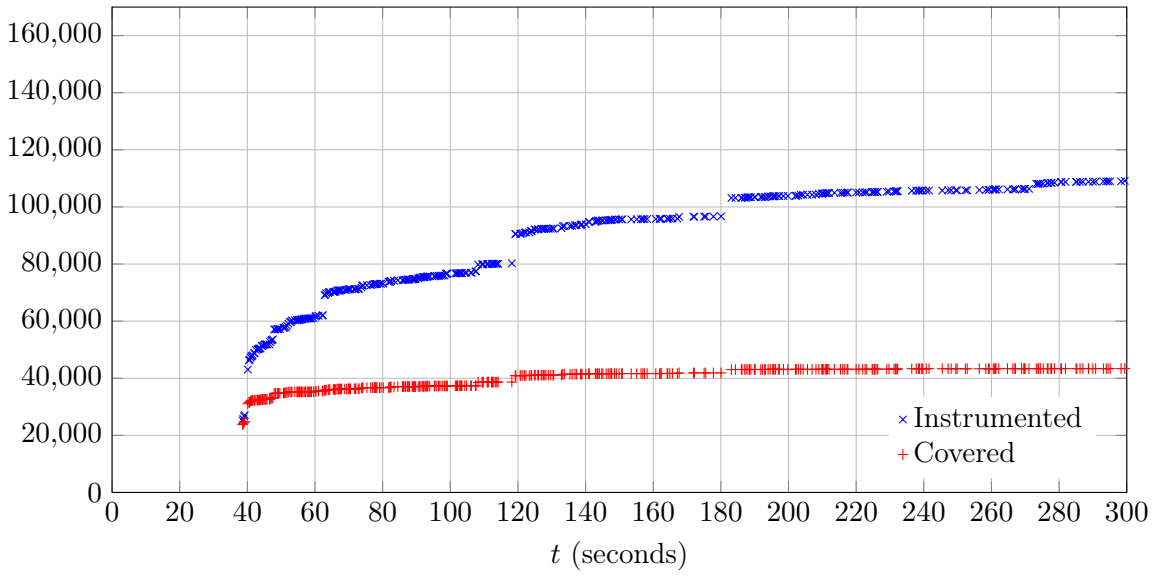

FIGURE 6.2: Grammar-based generation of syntactically valid PVL.

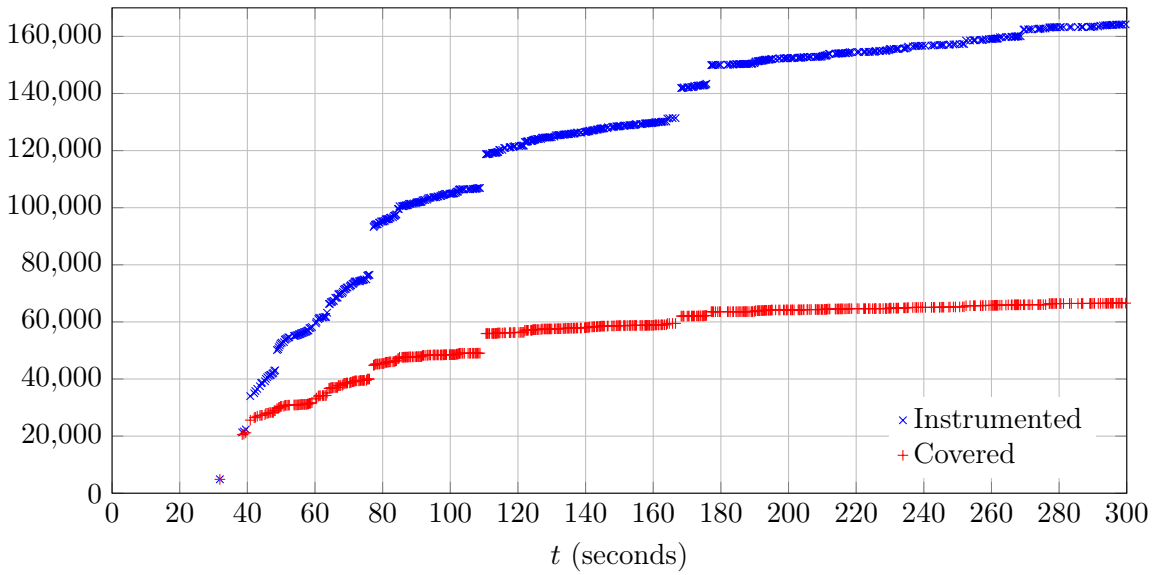

FIGURE 6.3: Grammar-based generation of PVL, Java, C++ and C.

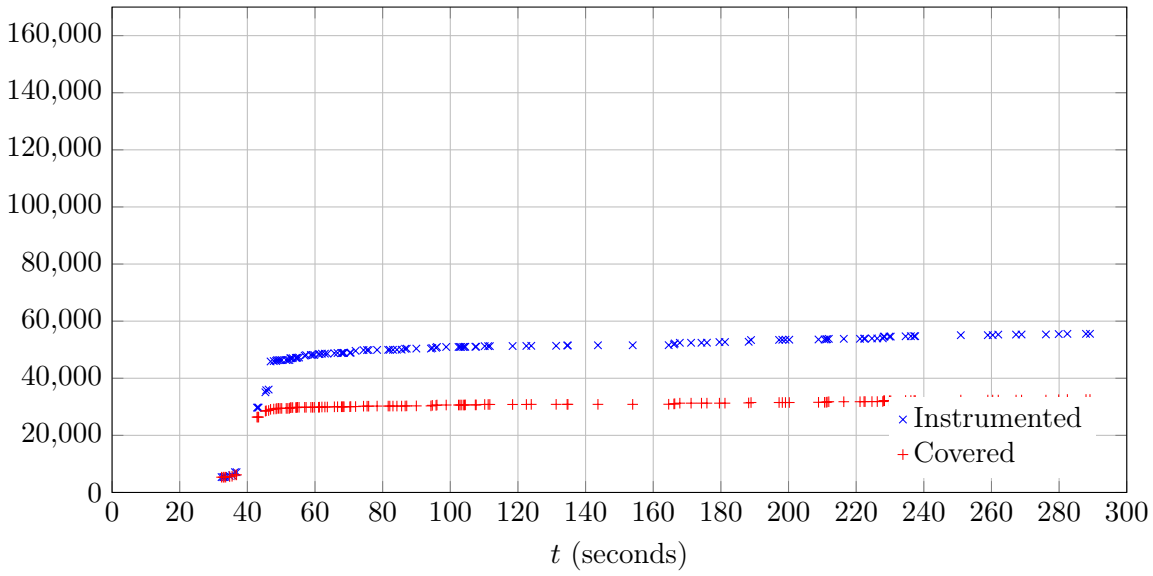

FIGURE 6.4: Coverage-guided generation of syntactically valid PVL.

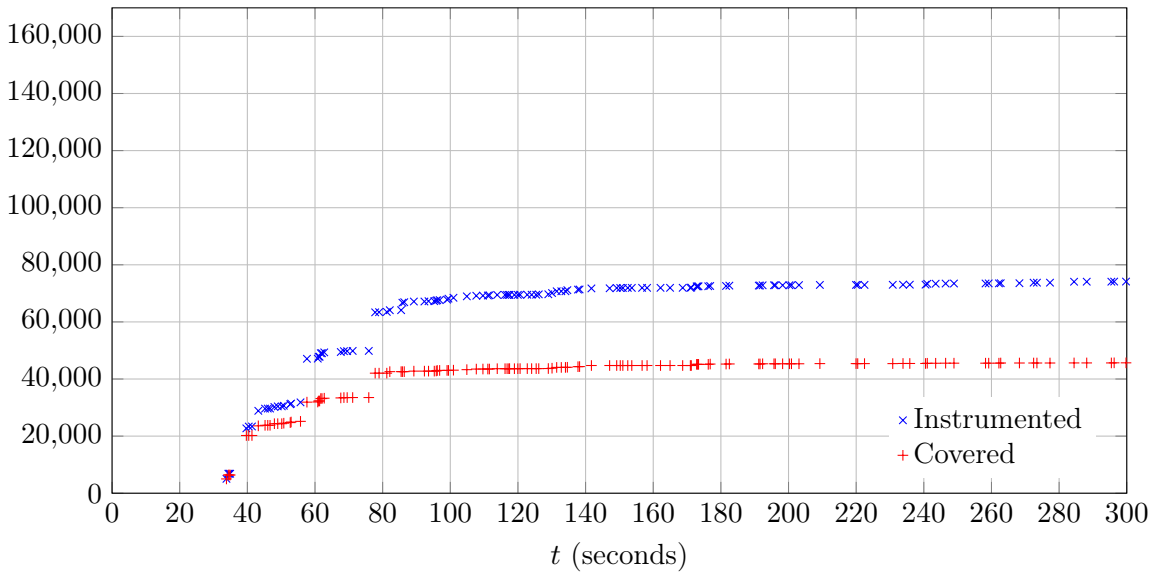

FIGURE 6.5: Coverage-guided generation of PVL, Java, C++ and C.

approach does not: programs do really make it all the way to the back-end (which is then skipped).

However, this success in achieving interesting depth is not captured by our chosen metric. Only a subset of PVL is generated, so code in the other front-ends is not touched, and the inputs are not very varied. The actual coverage metric does not increase much beyond the initial attempts. On the whole, the performance on our metric is thus worse than in Figure 6.2.

6.1.7 A flavor of findings

In general, and in our opinion, the bugs that we have discovered in VerCors have been relatively shallow, in a number of dimensions:

1. The crashing inputs are relatively short after minimization.

2. The cause of the crash is clear from the input and stack trace.

In our view, our findings are partially orthogonal to other approaches of improving confidence in software, including automated testing with unit tests and code review.

A number of issues that were found would be unlikely to be caught in either code review or tests. One of these issues, GitHub issue #1294 (C++ front-end fails to parse specific names, see appendix A), amounted to a parse error that only appeared in the C++ front-end, and only for names that consisting entirely of the letters u and l, case insensitive. For the bug to trigger, it was not enough for names to simply contain these letters. It is difficult to imagine a unit test that would find this issue.

However, for other crashes it is not difficult to imagine that automated testing could have caught the bug. Issue 1248 (empty 'enum' block crashes VerCors) and issue 1299 (empty 'sequential' block crashes VerCors), both also listed in appendix A, would be examples of such cases. A convention of always writing unit or integration tests for empty cases would have caught these errors.

It is difficult to estimate whether the issues that were found would have been encountered by a beginning user, of the kind that we hypothesized in our abstract. Some of the minimized crashing inputs in appendix A are grammatically valid 'accidentally' and would not be likely to be attempted even by a beginner; for instance, issue 1261 (writing 'bip_annotation' in the wrong location crashes VerCors) seems unlikely. However, there are also issues that a studious reader of the manual could run into, like issue 1303 (exponentiation expression crashes VerCors), or that could conceivably flow from a misunderstanding, like issue 1302 ('old' expression in requires clause crashes VerCors).

6.2 Approaches applied to other tools

In addition to comparing how the different approaches compare, we have investigated whether the coverage-guided and grammar-based fuzzing approaches we use here can discover bugs in verification tools other than VerCors. In this case, we are looking for new bugs that have not yet been reported to the project issue tracker. Since the other tools are not integrated into AValAnCHE, we do not have code coverage measurements and thus cannot use the metric we have defined earlier.

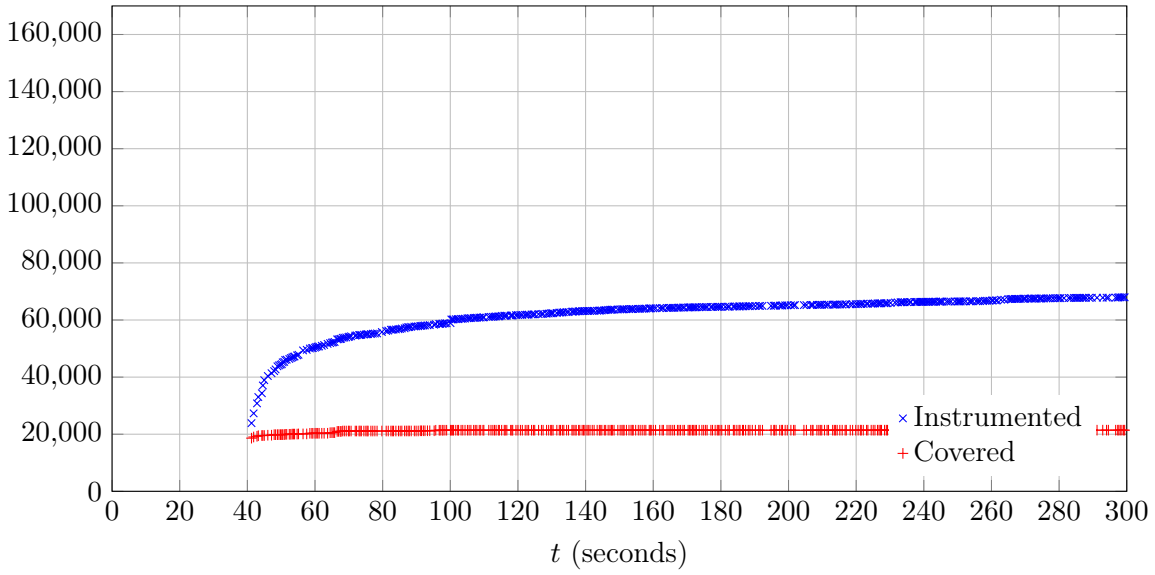

FIGURE 6.6: Generation of a verifiable subset of PVL.

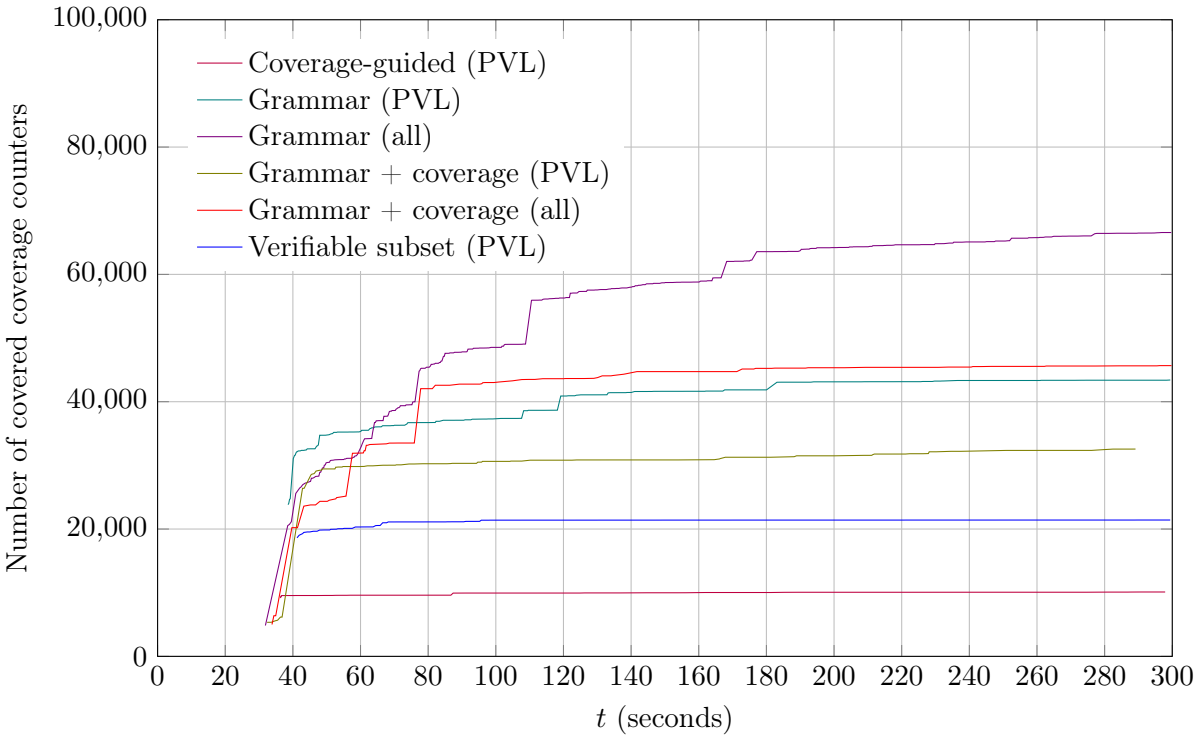

FIGURE 6.7: Performance of the different approaches compared.

6.2.1 VeriFast

VeriFast [25] is a verification tool for C, Rust and Java programs. Except for the Rust frontend, the tool itself is written in OCaml. Its authors describe it as a research prototype. Like VerCors, it supports both single-threaded and multi-threaded programs and is ultimately based on separation logic. Using a coverage-guided (not grammar-based) fuzzer, we have found three issues in VeriFast, of which two were new (not yet reported in the project issue tracker):

- Declaring an *enum* variable or type inside a C function causes VeriFast to exit with a fatal error. This has already been reported[1] in the VeriFast issue tracker as issue 296, that is, it is a known limitation.
- An unclosed block comment at the end of a C or Java file causes the lexer to exit with an "index out of bounds" error. This has been reported[2] as issue 731 and has been resolved (the same evening!) by the project's main author.
- Syntax errors with an incorrect exponential number literal (like *float f = 0e;*) confuse the lexer, causing VeriFast to report a fatal error. Since this happens at the lexing stage, a (syntactically incorrect) declaration like *int 4est();* also triggers the issue. This has been reported[3] as issue 739 and resolved soon after.

We could not use AValAnCHE or our metric from Section 6.1, since AValAnCHE is set up to measure code coverage in VerCors specifically and JVM-based verification tools more generally. Instead, we used the AFL++ (improved American Fuzzy Lop) fuzzer, using the set of examples shipped with VeriFast as the initial corpus. VeriFast is partially written in the OCaml programming language, which catches unexpected exceptions and shows a backtrace rather than crashing outright, which is the signal that AFL++ looks for. To resolve this, we inserted an illegal instruction (*ud2*) byte into the VeriFast binary at the correct location so that unexpected exceptions would immediately crash the process.

6.2.2 Dafny

Dafny [29] is a verification toolset like VerCors, albeit with a different practical approach: where VerCors verifies (existing, annotated) code in a number of languages, Dafny is a verification-aware programming language that can be compiled to a number of target languages. The C#, Go, Python, Java and JavaScript languages are supported. The verifier itself is written in C#.

We have looked at whether the grammar-based fuzzing approach from Section 5.1.2 can be used with the Dafny verification language and implementation. This is indeed possible; the grammar used by Dafny can be translated into an ANTLR grammar, which in turn can be used with the same tools we applied above. However, we have not been able to find any crashes in Dafny either through this approach, or with AFL++.

We have not investigated the test case generation approach from Section 5.1.4 with Dafny, since this has already been succesfully demonstrated by Irfan et al. [24] and other projects that build on that effort. In general, the likely explanation appears to be that the lowest-hanging bugs in Dafny have already been found.

[1] `https://github.com/verifast/verifast/issues/296`
[2] `https://github.com/verifast/verifast/issues/731`
[3] `https://github.com/verifast/verifast/issues/739`

6.2.3 Carbon and Silicon

The Carbon and Silicon verifiers, both part of the Viper project from the ETH Zürich [34], have already been introduced in Section 3.2.3 as backends for VerCors. Since they are also verifiers in their own right, verifying programs in the Silver intermediate language, our testing approach can be used for (and by) these projects as well. We have attempted both coverage-guided and grammar-based fuzzing.

For the coverage-guided fuzzing, we have again used the AFL++ fuzzer. Using this fuzzer, we discovered that an input containing bytes that are invalid in the UTF-8 encoding would cause the Silver parser to crash. This could happen for example if a file with a comment containing the Danish letter ä was saved in a Latin1 characters encoding. However, this issue was not new - it had already been reported[4] to the issue tracker as issue 499, by another researcher, who incidentally was also using a fuzzer.

Finally, we also applied the grammar-based approach from AValAnCHE to both Carbon and Silicon, by writing a Grammarinator-compatible ANTLR grammar for the Silver intermediate language based on the grammar description that is included in the Silver documentation, then running both verifiers on at least 5000 randomly generated programs each. However, we did not find any crashes using this method; all inputs were correctly accepted or rejected without throwing an exception.

[4] `https://github.com/viperproject/silver/issues/499`

Chapter 7

Related work

In Chapter 2, we introduced a number of different fuzzing tools that inpired or formed the basis of our own approach. In this chapter, we will describe a small sample of efforts to apply fuzzing specifically to program verification tools.

Especially the Dafny verification-aware programming language [29], introduced in the previous section, has been quite extensively fuzzed using a number of different tools, three of which we discuss in this chapter.

7.1 XDsmith

To find correctness issues in Dafny's different compilers, Irfan et al. [24] built XDsmith. XDsmith uses the Xsmith library described in Section 2.4 to generate Dafny programs (including annotations) that are not only syntactically correct, but that also have a verification outcome that is already known during generation. This allows the tool to find both soundness and precision errors, that is, programs that are verified incorrectly, as well as programs that are rejected incorrectly.

XDsmith generates a 'carefully chosen subset' of the Dafny language, including annotations, which is sequential, deterministic, and free of heap mutations, and where there is always one execution path through the program. Loops and recursion are not supported, which implies that that execution path is finite and generated programs always terminate. Because of this, XDsmith can use the generated program itself to find the values of function arguments and return values, by executing the generated code. The annotations generated are specific rather than general, for instance requiring a specific argument value in the precondition and static a specific result value in the postcondition.

With the generated annotations in place, the program should verify, or else a precision bug is found; conversely, if the generated annotations are changed even slightly, the program should no longer verify.

In addition to comparing the verification result from Dafny against the known correct result, XDsmith uses the verified generated programs to do differential testing (described earlier in Section 2.4), looking for bugs in the different compilers that translate Dafny to a target language. Since the Dafny program was verified, the visible behavior should be the same for each of the programs compiled from that Dafny program. If it is not, or if a compiled program crashes, a bug has been discovered.

Running XDsmith on a cluster of 100 machines over a three-month period uncovered 31 new Dafny bugs in total, both in the verifier and in the compilers that translate Dafny code to other languages.

7.2 fuzz-d and DafnyFuzz

fuzz-d [42] and DafnyFuzz [14] are also compiler testing tools targeting Dafny [29].

In contrast to our own research, the primary focus of both tools is to look for miscompilations: that is, instances where the semantic meaning of the Dafny program is not preserved after transformation into the target language. However, both tools can also find, and have found, instances where an input causes the Dafny compiler to crash - which the authors describe as problematic since running into an easy-to-trigger crashing bug would make it difficult to find the correctness issues both groups were actually looking for. In total, the authors report that 24 new bugs were found by the two tools, of which 9 were soundness bugs.

In broad strokes, both fuzz-d and DafnyFuzz are test case generators in the same category as XDsmith (and so similar to our own application of Xsmith), though using their own generation code based on an ANTLR [36] grammar. Both projects were started to add to, and overcome limitations in, the XDsmith approach described in the previous section. In particular, XDsmith's approach of only generating test programs that are finite excludes generation of test programs containing loops or recursion. Both of the newer tools will generate such programs; in addition, fuzz-d will for example also generate code that uses Dafny's object-oriented features.

Since both tools look for miscompilations rather than tool crashes, a test oracle is required to distinguish passing from failing tests. In general, the tools use differential testing, compiling Dafny programs to different targets and asserting that all target programs output the same values. Both fuzz-d and DafnyFuzz can optionally also use a *self-checking oracle*: here, the generated programs 'know' what their output should be, and compare these to the values computed at runtime.

Similar to XDsmith, a goal of these tools is to generate only programs that have completely defined behavior at runtime. The fuzz-d tool uses two kinds of *reconditioning* to avoid generating programs that would fail: programs that divide integers by zero, access arrays at out-of-bounds indexes, and so on. The first kind, referred to as *standard reconditioning* by the authors, is inspired by Csmith [46]: all division and index operators are replaced with operations that safely come up with a value, regardless of the inputs. The second, named *advanced reconditioning*, performs this reconditioning only where needed, by considering all possible locations at first, executing the program to see which safe versions of operators are actually used, then removing the remainder. In their evaluation of the advanced reconditioning feature, the authors write that the addition of this approach did not lead to additional findings, but that having this feature enabled did result in inputs that were easier to reduce to an understandable test case.

The standout feature of the DafnyFuzz tool is that it performs value tracking: as it generates programs, DafnyFuzz remembers the values that variables should take. This avoids the need for reconditioning: for instance, DafnyFuzz can avoid generating division operators on variables that it knows are zero. In addition, the tracked values can be used to build a test oracle: if the program, when executed, computes different values than the ones tracked by DafnyFuzz, this indicates a bug, either in Dafny or in DafnyFuzz.

DafnyFuzz and fuzz-d are both integrated with the Perses test case reducer [41] to automatically minimize interesting inputs. Perses is grammar-aware: it knows the Dafny grammar and can therefore attempt removing structural parts of the program. Nevertheless, the fuzz-d authors report that in some cases, manually reducing test cases was faster than running an automated reducer. This aligns with our own experience.

Chapter 8

Conclusions

We will now return to our research questions, summarize our findings, and make some recommendations for VerCors and similar program verification tools based on our experience.

8.1 Confidence

Our main research question (RQ1) concerned the extent to which we can improve confidence in the robustness of the VerCors toolset by adding fuzz testing, and in RQ1.2 we asked what conclusions we can draw from our measurements and results.

Compared to the hard guarantees sought and provided by formal verification, these conclusions are somewhat limited.

Since our approach is based on testing and not verification, as already introduced in Section 1.1, an essential limitation of our approach is that we can make no claims about inputs or situations that we have not tried. There is no way to be 'done' with fuzzing; no moment where we are certain that all crashes have been found, or that a certain fraction of bugs have been discovered.

What we can state instead is that we have found the 'first few hours' of fuzzer-findable robustness bugs in VerCors, that is, the 'low-hanging fruit'. Furthermore, we have automated an approach that allows continuing to find such 'fruit' in the future, if need be.

From our investigation in Section 6.2 for RQ2, it appears likely that there is indeed a class of bugs that is amenable to being found by fuzzing. In that section, we have shown that using fuzzing approaches is both possible and worthwhile for other verification tools. We have found new (not yet reported) bugs in those tools using approaches similar to the ones used in AValAnCHE for VerCors, in addition to rediscovering already-known issues. In the projects that had already been subject to any form of fuzzing or test case generation, specifically Dafny and Viper (Carbon/Silicon), we did not find any new bugs. This strengthens our belief that fuzzing is best suited at finding a certain category of (shallow) bugs easily, and that the type of fuzzing is of secondary importance. For VerCors, that class or category consists then at least of the bugs listed in appendix A.

Of course, in a strict sense, the robustness of VerCors is only improved if the issues discovered are actually resolved. Indeed, some of the issues have already been fixed by the VerCors developers, and both new issues in VeriFast were also quickly repaired.

It is important to note that the AValAnCHE tool does not prioritize the issues it finds, a task that is left entirely to its users, the VerCors developers. Not all crashing code patterns have the same likelihood of appearing in user's code. For example, it seems reasonable to assume that GitHub issue 1312 (postfix increment without permission crashes VerCors, see appendix A) is more likely to trigger for a beginning user than issue 1313 (label in

Java initializer block crashes VerCors), just because initializer blocks are rare and labels in initializer blocks even more so. It will remain up to the VerCors project developers to decide how to prioritize fixing these issues relative to all the other engineering and research work that could be done, like adding additional features and supporting more, or bigger subsets of, languages.

8.2 Integration into VerCors project

Since the development of the AValAnCHE tool itself is not yet complete at the time of writing, and has not yet been integrated into the VerCors codebase or shared with the VerCors project developers, we can not yet report on any practical experiences in using the tool. This makes it difficult to claim with certainty that the integration we are envisioning, with a OSS-Fuzz [2] style fuzzing machine quietly spinning away on some already-running infrastructure, is the best possible approach (RQ1.1).

8.3 Comparing strategies

Regarding our Research Question 1.3, among the fuzzing strategies that we have attempted and with our selected performance metric, the strategy that gives the best results for VerCors is clearly grammar-based generation of as wide a scope of input languages as possible. This would be the strategy to recommend going forward, both for VerCors and for similar verification tools.

As expected, the purely random approach of the original fuzz(1) tool is not an effective method of finding crashing bugs in VerCors: the ANTLR-based parsers are robust, battle-tested, and correctly report syntax errors rather than crashing, for all of the many hundreds of thousands of inputs we have found with our various strategies. We were surprised that the Radamsa-based blind fuzzer managed to find a bug, and crucially, it was not a bug in the generated parser.

Coverage-guided fuzzing without a corpus did not, in general, find interesting inputs. Adding coverage guidance to the grammar-based fuzzer did not improve its performance, although we cannot say with certainty whether this is an inherent issue or one related to our implementation.

Of the strategies being compared, the XSmith-based test case generation method generated the most realistic programs and discovered the 'deepest' and most interesting errors. However, it is also by far the slowest, both at generating inputs and in integrating the approach into a project. XSmith is geared toward differential testing more than crash testing; in the way we have used it, generating test inputs becomes a bottleneck in itself. In addition, the existing ANTLR grammars can not be reused for the XSmith approach. Additional care is needed to make sure that the fuzzer understands enough of the targeted language's type system to generate correctly-typed programs, although the XSmith 'canned components' are very helpful in this regard.

8.4 Challenges and limitations

A technical limitation of our current implementation of the approach in the AValAnCHE tool is that issues that are found, reported, but not yet fixed will likely be found again. While there is a deduplication process that ensures crashes with the same stack trace are not reported multiple times, this process is not perfect, and crashes from multiple locations

in the code can still have the same root cause. There is also not a good way for the tool to discover that a crash that was found has the same root cause as another crash in a previous version of VerCors. This makes the email notification feature described in Section 5.4 less useful than it could be.

In our abstract, we describe our goal as finding the kinds of crashing bugs that new users of VerCors might trigger. Unfortunately, there is no guarantee that the crashing inputs we have found are representative of the kind of input that would be tried by a beginning user. For at least one of the issues found by AValAnCHE (using an 'old'-expression in a context clause, issue 1302) we have understood that this was at some point attempted by a real user, but this is an anecdote. For other issues, it seems likely that a new user might accidentally run into the issue (exponentiation expression, issue 1303; postfix increment, issue 1312), but this is of course an assumption, which is arguably worse than an anecdote.

8.5 Summary and recommendations

For verification tools like VerCors that use automated testing and code review, but not yet fuzzing, we can answer the question implied by our Research Question 2 in the positive. We would strongly recommend adding some form of fuzzing, especially grammar-based fuzzing, to the testing strategy of all program verification tools. Especially if the input grammar is already represented in a way that can be used by fuzzing tools, perhaps after some simplification, we have found that grammar-based fuzzing is an effective way of finding errors, and that those errors are at least in part orthogonal to those that would be found using unit testing or integration testing.

Especially for coverage-guided fuzzing, but also for grammar-based fuzzing, the performance and resource use is heavily dependent on the performance of the system being tested. In short, making a verifier faster also makes it easier and more rewarding to fuzz.

Chapter 9

Future work

In this final chapter we suggest some possible avenues for future research and engineering work.

9.1 Soundness and precision errors

In our research, our focus was on finding robustness errors in VerCors. However, as we saw in Chapter 7, approaches similar to our own can also be used to find soundness problems, which as we discussed in Section 3.3.1 can have much more serious consequences. (Recall from that section that a soundness error is when a tool verifies a program where it should not; a precision error is when a tool rejects a program that should verify.) With care, programs can be generated that 'know' their own expected output, an approach referred to in [14] as 'self-checking oracles' and used for the fuzz-d and DafnyFuzz tools described in that paper.

A variant of this approach would be to generate programs that perform a computation and then purposely crash at runtime if the result does not match the predetermined expectations. If such a strategy was used, the AValAnCHE tool could be used to find some classes of soundness errors as well, without large changes to the tool itself.

It would be interesting to see whether a fuzzing approach such as the one used in the present research, and a differential testing approach as used in XDafny, could be combined to test verification tools against each other, for instance verifying the same C code in VerCors and VeriFast and seeing whether the verifiers agree. If they differ, this would point to either a soundness error or a precision error in either verifier. For this approach to work, it must be possible to generate inputs in such a way that the specification annotations can be understood by, or translated to be understood by, both tools with the same meaning.

As we have discussed, for VerCors a precision error is not currently treated as an error; instead, the suggestion is to add the required additional annotation. However, quantifying the precision of VerCors in some way would be interesting. Ideally, over time, the amount of annotations required to verify programs would trend downward.

9.2 Sharing grammars

In our recommendations, we encouraged other verification tools to also take advantage of fuzzing as a problem-finding instrument, if they are not doing so already. The approach of reusing the grammar specification that is already used for the parser, as used by us and supported by Grammarinator, already lowers the barrier to adoption when it comes to adopting grammar-based fuzzing as a testing strategy. Other verification tools that

use ANTLR [36] parsers and grammars can use a tool like Grammarinator; if they do not use ANTLR, but verify a language similar to PVL, they could use AValAnCHE's fuzzer grammar as a starting point.

For what we called test case generation, or generation of a verifiable subset, such reuse is also possible. As mentioned, the XSmith project already offers a number of reusable ('canned') components that describe features common to many programming languages: for instance, many languages support integer addition or array literals, with similar type system rules if perhaps different syntax, so the description of these features can be reused, and is shipped with XSmith itself. In our view, it would be useful to offer similar reusable tools for common elements of verification languages, to support others who want to use XSmith with a verification tool.

Either the VerCors PVL Xsmith grammar or the Dafny XDsmith grammar could serve as a useful base for this effort.

9.3 Better generators, faster testing

As mentioned in Chapter 2, many hundreds of fuzzing tools exist. Just a few have been selected for comparison and experimentation as part of this research. With that in mind, the AValAnCHE tool has been built with the intention that fuzzers and their grammars can be easily swapped out as new, improved, or just plain better ones are found. Specifically, the poor performance (in our coverage-over-time metric) of the seeded Grammarinator fuzzer may be specific to our approach and not due to any issue with that approach in general. Improvements to the Xsmith grammar may improve generation performance as well. If both these cases, it should be relatively easy to swap in a better implementation in its place.

Currently, AValAnCHE does not enjoy any kind of privileged access to VerCors: it executes the entire verifier, excluding the backend. In the pursuit of faster (and thus more effective) fuzz testing, a possible avenue to explore would be to experiment with skipping more parts of VerCors, for example circumventing the different parsers. This could be done by directly generating valid abstract syntax trees (ASTs), or perhaps generating valid Protobuf binary blobs describing ASTs, instead of generating valid PVL, Java, C or C++ program text. This would avoid spending CPU cycles on generating and then parsing program syntax. Without additional tooling, however, it may make it more difficult for users of the AValAnCHE tool to reproduce the crashes the tool finds, and care must be taken that only ASTs that could result from a possible program are tried to avoid false positives.

Of course, making VerCors faster in general (as has been done in [33], for instance) would also be advantageous, for users but also for fuzz testers: every strategy we have tried benefits from being able to do more verification runs with fewer computational resources.

9.4 LLVM and Pallas

At the time of writing, a project is underway to add support for LLVM [28] intermediate representation (IR) to VerCors, originally under the name VCLLVM [44] but now as Pallas.

Since many programming languages use the LLVM toolchain and compiler framework, and compile to LLVM IR, this would widen the applicability of VerCors to these languages without VerCors having to include a specific front-end to support the language.

As part of this research, we have shown that the approaches described in Chapter 5 apply to the VCLLVM/Pallas frontend in principle: GitHub issue 1346 (nonexistent variables

in LLVM binops crash VerCors, see appendix A) has been found by grammar-based generation of method contracts in LLVM IR assembly code. However, as the project is still under development, the precise syntax is still in flux. The expectation is that the specific *VC.contract* construct being exercised in this issue will be deprecated, with another (as yet undocumented) method contract syntax taking its place.

It would be interesting to see whether generating LLVM assembly code, LLVM bitcode, or C code performs best when it comes to fuzzing, exercising the VerCors codebase and finding bugs in VerCors in general and Pallas in particular.

Bibliography

[1] Lukas Armborst, Pieter Bos, Lars B. van den Haak, Marieke Huisman, Robert Rubbens, Ömer Şakar, and Philip Tasche. "The VerCors Verifier: A Progress Report". In: *Computer Aided Verification*. Ed. by Arie Gurfinkel and Vijay Ganesh. Cham: Springer Nature Switzerland, 2024, pp. 3–18. ISBN: 978-3-031-65630-9. DOI: 10.1007/978-3-031-65630-9_1.

[2] Abhishek Arya, Oliver Chang, Jonathan Metzman, Kostya Serebryany, and Dongge Liu. *OSS-fuzz*. URL: https://github.com/google/oss-fuzz.

[3] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. "NAUTILUS: Fishing for Deep Bugs with Grammars". In: *Proceedings 2019 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA: Internet Society, 2019. ISBN: 978-1-891562-55-6. DOI: 10.14722/ndss.2019.23412. URL: https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_04A-3_Aschermann_paper.pdf (visited on October 10, 2024).

[4] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. "Boogie: A Modular Reusable Verifier for Object-Oriented Programs". In: *Formal Methods for Components and Objects*. Ed. by Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 364–387. ISBN: 978-3-540-36750-5.

[5] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. "GRIMOIRE: Synthesizing Structure While Fuzzing". In: *Proceedings of the 28th USENIX Conference on Security Symposium*. SEC'19. USA: USENIX Association, August 14, 2019, pp. 1985–2002. ISBN: 978-1-939133-06-9.

[6] Simon Bliudze, Petra van den Bos, Marieke Huisman, Robert Rubbens, and Larisa Safina. "JavaBIP Meets VerCors: Towards the Safety of Concurrent Software Systems in Java". In: *Fundamental Approaches to Software Engineering*. Ed. by Leen Lambers and Sebastián Uchitel. Vol. 13991. Cham: Springer Nature Switzerland, 2023, pp. 143–150. ISBN: 978-3-031-30825-3 978-3-031-30826-0. DOI: 10.1007/978-3-031-30826-0_8. URL: https://link.springer.com/10.1007/978-3-031-30826-0_8 (visited on April 29, 2025).

[7] Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. "The VerCors Tool Set: Verification of Parallel and Concurrent Software". In: *Integrated Formal Methods*. Ed. by Nadia Polikarpova and Steve Schneider. Cham: Springer International Publishing, 2017, pp. 102–110. ISBN: 978-3-319-66845-1. DOI: 10.1007/978-3-319-66845-1_7.

[8] Stefan Blom and Marieke Huisman. "The VerCors Tool for Verification of Concurrent Programs". In: *FM 2014: Formal Methods*. Ed. by Cliff Jones, Pekka Pihlajasaari, and Jun Sun. Cham: Springer International Publishing, 2014, pp. 127–131. ISBN: 978-3-319-06410-9. DOI: 10.1007/978-3-319-06410-9_9.

[9] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. "A Survey of Compiler Testing". In: *ACM Computing Surveys* 53.1 (January 31, 2021), pp. 1–36. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3363562. URL: https://dl.acm.org/doi/10.1145/3363562 (visited on November 1, 2024).

[10] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. "Taming Compiler Fuzzers". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. New York, NY, USA: Association for Computing Machinery, June 16, 2013, pp. 197–208. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2491956.2462173. URL: https://dl.acm.org/doi/10.1145/2491956.2462173 (visited on November 8, 2024).

[11] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. "One Engine to Fuzz 'em All: Generic Language Processor Testing with Semantic Validation". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021 IEEE Symposium on Security and Privacy (SP). May 2021, pp. 642–658. DOI: 10.1109/SP40001.2021.00071. URL: https://ieeexplore.ieee.org/document/9519403/?arnumber=9519403 (visited on November 1, 2024).

[12] CodeIntelligence. *Jazzer: Coverage-guided, in-Process Fuzzing for the JVM*. URL: https://github.com/CodeIntelligenceTesting/jazzer (visited on May 6, 2025).

[13] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3.

[14] Alastair F. Donaldson, Dilan Sheth, Jean-Baptiste Tristan, and Alex Usher. "Randomised Testing of the Compiler for a Verification-Aware Programming Language". In: *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 2024 IEEE Conference on Software Testing, Verification and Validation (ICST). Toronto, ON, Canada: IEEE, May 27, 2024, pp. 407–418. ISBN: 979-8-3503-0818-1. DOI: 10.1109/ICST60714.2024.00044. URL: https://ieeexplore.ieee.org/document/10638596/ (visited on May 5, 2025).

[15] Florian Dyck, Cedric Richter, and Heike Wehrheim. "Robustness Testing of Software Verifiers". In: *Software Engineering and Formal Methods*. Ed. by Carla Ferreira and Tim A. C. Willemse. Vol. 14323. Cham: Springer Nature Switzerland, 2023, pp. 66–84. ISBN: 978-3-031-47114-8 978-3-031-47115-5. DOI: 10.1007/978-3-031-47115-5_5. URL: https://link.springer.com/10.1007/978-3-031-47115-5_5 (visited on April 20, 2025).

[16] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. "The Daikon System for Dynamic Detection of Likely Invariants". In: *Science of Computer Programming* 69.1–3 (December 2007), pp. 35–45. ISSN: 01676423. DOI: 10.1016/j.scico.2007.01.015. URL: https://linkinghub.elsevier.com/retrieve/pii/S016764230700161X (visited on April 29, 2025).

[17] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. "AFL++: Combining Incremental Steps of Fuzzing Research". In: *Proceedings of the 14th USENIX Conference on Offensive Technologies*. WOOT'20. USA: USENIX Association, August 11, 2020, p. 10.

[18] Robert W. Floyd. "Assigning Meanings to Programs". In: *Program Verification: Fundamental Issues in Computer Science*. Ed. by Timothy R. Colburn, James H. Fetzer, and Terry L. Rankin. Dordrecht: Springer Netherlands, 1993, pp. 65–81. ISBN: 978-94-011-1793-7. DOI: 10.1007/978-94-011-1793-7_4. URL: https://doi.org/10.1007/978-94-011-1793-7_4.

[19] Christian Haack, Marieke Huisman, Clément Hurlin, and Afshin Amighi. "Permission-Based Separation Logic for Multithreaded Java Programs". In: *Logical Methods in Computer Science* Volume 11, Issue 1 (February 27, 2015), p. 998. ISSN: 1860-5974. DOI: 10.2168/LMCS-11(1:2)2015. URL: https://lmcs.episciences.org/998 (visited on May 6, 2025).

[20] William Hatch, Pierce Darragh, Sorawee Porncharoenwase, Guy Watson, and Eric Eide. "Generating Conforming Programs with Xsmith". In: *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE '23: 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences. Cascais Portugal: ACM, October 22, 2023, pp. 86–99. ISBN: 979-8-4007-0406-2. DOI: 10.1145/3624007.3624056. URL: https://dl.acm.org/doi/10.1145/3624007.3624056 (visited on November 1, 2024).

[21] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Communications of The Acm* 12.10 (October 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259.

[22] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. "Grammarinator: A Grammar-Based Open Source Fuzzer". In: *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. ESEC/FSE '18: 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Lake Buena Vista FL USA: ACM, November 5, 2018, pp. 45–48. ISBN: 978-1-4503-6053-1. DOI: 10.1145/3278186.3278193. URL: https://dl.acm.org/doi/10.1145/3278186.3278193 (visited on September 24, 2024).

[23] Marc R. Hoffman, Evgeny Mandrikov, and Mirko Friedenhagen. *JaCoCo Java Code Coverage Library*. URL: https://www.jacoco.org/jacoco/ (visited on May 6, 2025).

[24] Ahmed Irfan, Sorawee Porncharoenwase, Zvonimir Rakamarić, Neha Rungta, and Emina Torlak. "Testing Dafny (Experience Paper)". In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. Virtual South Korea: ACM, July 18, 2022, pp. 556–567. ISBN: 978-1-4503-9379-9. DOI: 10.1145/3533767.3534382. URL: https://dl.acm.org/doi/10.1145/3533767.3534382 (visited on November 1, 2024).

[25] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. "VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java". In: *NASA Formal Methods*. Ed. by Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 41–55. ISBN: 978-3-642-20398-5.

[26] Yue Jia and Mark Harman. "An Analysis and Survey of the Development of Mutation Testing". In: *IEEE Transactions on Software Engineering* 37.5 (2011), pp. 649–678. DOI: 10.1109/TSE.2010.62.

[27] Sophie Lathouwers and Marieke Huisman. "Survey of Annotation Generators for Deductive Verifiers". In: *Journal of Systems and Software* 211 (May 2024), p. 111972. ISSN: 01641212. DOI: 10.1016/j.jss.2024.111972. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121224000153 (visited on October 22, 2024).

[28] C. Lattner and V. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* International Symposium on Code Generation and Optimization, 2004. CGO 2004. San Jose, CA, USA: IEEE, 2004, pp. 75–86. ISBN: 978-0-7695-2102-2. DOI: 10.1109/CGO.2004.1281665. URL: http://ieeexplore.ieee.org/document/1281665/ (visited on April 29, 2025).

[29] Rustan Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning.* LPAR'10. Berlin, Heidelberg: Springer-Verlag, April 25, 2010, pp. 348–370. ISBN: 978-3-642-17510-7.

[30] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. "The Art, Science, and Engineering of Fuzzing: A Survey". In: *IEEE Transactions on Software Engineering* 47.11 (November 2021), pp. 2312–2331. ISSN: 1939-3520. DOI: 10.1109/TSE.2019.2946563. URL: https://ieeexplore.ieee.org/document/8863940 (visited on November 11, 2024).

[31] Barton P. Miller, Lars Fredriksen, and Bryan So. "An Empirical Study of the Reliability of UNIX Utilities". In: *Commun. ACM* 33.12 (December 1, 1990), pp. 32–44. ISSN: 0001-0782. DOI: 10.1145/96267.96279. URL: https://dl.acm.org/doi/10.1145/96267.96279 (visited on November 8, 2024).

[32] Barton P. Miller, Mengxiao Zhang, and Elisa R. Heymann. "The Relevance of Classic Fuzz Testing: Have We Solved This One?" In: *IEEE Transactions on Software Engineering* 48.6 (June 2022), pp. 2028–2039. ISSN: 1939-3520. DOI: 10.1109/TSE.2020.3047766. URL: https://ieeexplore.ieee.org/document/9309406 (visited on November 8, 2024).

[33] Henk Mulder, Marieke Huisman, and Sebastiaan Joosten. "Improving Performance of the VerCors Program Verifier". In: *Deductive Software Verification: Future Perspectives: Reflections on the Occasion of 20 Years of KeY.* Ed. by Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, and Mattias Ulbrich. Cham: Springer International Publishing, 2020, pp. 65–82. ISBN: 978-3-030-64354-6. DOI: 10.1007/978-3-030-64354-6_3. URL: https://doi.org/10.1007/978-3-030-64354-6_3 (visited on September 19, 2024).

[34] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: *Verification, Model Checking, and Abstract Interpretation.* Ed. by Barbara Jobstmann and K. Rustan M. Leino. Vol. 9583. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 41–62. ISBN: 978-3-662-49121-8 978-3-662-49122-5. DOI: 10.1007/978-3-662-49122-5_2. URL: http://link.springer.com/10.1007/978-3-662-49122-5_2 (visited on November 11, 2024).

[35] Peter W. O'Hearn. "Resources, Concurrency, and Local Reasoning". In: *Theoretical Computer Science* 375.1 (2007), pp. 271–307. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2006.12.035. URL: https://www.sciencedirect.com/science/article/pii/S030439750600925X.

[36] T. J. Parr and R. W. Quong. "ANTLR: A Predicated- *LL* (*k*) Parser Generator". In: *Software: Practice and Experience* 25.7 (July 1995), pp. 789–810. ISSN: 0038-0644, 1097-024X. DOI: 10.1002/spe.4380250705. URL: https://onlinelibrary.wiley.com/doi/10.1002/spe.4380250705 (visited on April 25, 2025).

[37] Paul Purdom. "A Sentence Generator for Testing Parsers". In: *BIT Numerical Mathematics* 12.3 (September 1, 1972), pp. 366–375. ISSN: 1572-9125. DOI: 10.1007/BF01932308. URL: https://doi.org/10.1007/BF01932308 (visited on November 8, 2024).

[38] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. "Test-Case Reduction for C Compiler Bugs". In: *ACM SIGPLAN Notices* 47.6 (August 6, 2012), pp. 335–346. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/2345156.2254104. URL: https://dl.acm.org/doi/10.1145/2345156.2254104 (visited on October 23, 2024).

[39] Robert Rubbens, Petra van den Bos, and Marieke Huisman. "VeyMont: Choreography-Based Generation of Correct Concurrent Programs with Shared Memory". In: *Integrated Formal Methods*. Ed. by Nikolai Kosmatov and Laura Kovács. Vol. 15234. Cham: Springer Nature Switzerland, 2025, pp. 217–236. ISBN: 978-3-031-76553-7 978-3-031-76554-4. DOI: 10.1007/978-3-031-76554-4_12. URL: https://link.springer.com/10.1007/978-3-031-76554-4_12 (visited on April 29, 2025).

[40] Jukka Ruohonen and Kalle Rindell. "Empirical Notes on the Interaction Between Continuous Kernel Fuzzing and Development". In: *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). Berlin, Germany: IEEE, October 2019, pp. 276–281. ISBN: 978-1-7281-5138-0. DOI: 10.1109/ISSREW.2019.00084. URL: https://ieeexplore.ieee.org/document/8990271/ (visited on October 7, 2024).

[41] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. "Perses: Syntax-Guided Program Reduction". In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18: 40th International Conference on Software Engineering. Gothenburg Sweden: ACM, May 27, 2018, pp. 361–371. ISBN: 978-1-4503-5638-1. DOI: 10.1145/3180155.3180236. URL: https://dl.acm.org/doi/10.1145/3180155.3180236 (visited on May 5, 2025).

[42] Alex Usher. "Fuzz-d : Random Program Generation for Testing Dafny". Imperial College London, 2023. URL: https://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/2223-ug-projects/fuzz-d-Random-Program-Generation-for-Testing-Dafny.pdf.

[43] Petra van den Bos and Marieke Huisman. "The Integration of Testing and Program Verification". In: *A Journey from Process Algebra via Timed Automata to Model Learning*. Ed. by Nils Jansen, Mariëlle Stoelinga, and Petra van den Bos. Cham: Springer Nature Switzerland, 2022, pp. 524–538. ISBN: 978-3-031-15629-8. DOI: 10.1007/978-3-031-15629-8_28. URL: https://doi.org/10.1007/978-3-031-15629-8_28 (visited on September 19, 2024).

[44] Dré Van Oorschot, Marieke Huisman, and Ömer Şakar. "First Steps towards Deductive Verification of LLVM IR". In: *Fundamental Approaches to Software Engineering*. Ed. by Dirk Beyer and Ana Cavalcanti. Vol. 14573. Cham: Springer Nature Switzerland, 2024, pp. 290–303. ISBN: 978-3-031-57258-6 978-3-031-57259-3. DOI: 10.1007/978-3-031-57259-3_15. URL: https://link.springer.com/10.1007/978-3-031-57259-3_15 (visited on April 29, 2025).

[45] *VerifyThis Long-Term Challenge*. VerifyThis Long-Term Challenge. URL: https://verifythis.github.io/ (visited on November 11, 2024).

[46] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. "Finding and Understanding Bugs in C Compilers". In: *SIGPLAN Not.* 46.6 (June 4, 2011), pp. 283–294. ISSN: 0362-1340. DOI: 10.1145/1993316.1993532. URL: https://dl.acm.org/doi/10.1145/1993316.1993532 (visited on November 8, 2024).

[47] Michał Zalewski. *Technical Whitepaper for Afl-Fuzz*. URL: https://lcamtuf.coredump.cx/afl/technical_details.txt (visited on October 10, 2024).

[48] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., September 2005. ISBN: 978-1-55860-866-5.

During the preparation of this work the author did not use any artificial intelligence tools.

Appendix A

Issues discovered

The following issues in VerCors were discovered using different versions of the AValAnCHE tool, including the prototype version of the tool developed as part of the preparation for this research. The issues are listed in chronological order of discovery and are numbered for easy reference with the VerCors repository, `https://github.com/utwente-fmt/vercors`.

In each case, a (manually) minimized input is given as an example, rather than the complete input as generated by the fuzzer. The inputs are not intended to be examples of actual, useful programs, but they are grammatically correct.

The repository's issue tracker has more details for each issue, including a change history pointing at the VerCors version where an issue was fixed, if applicable.

Issue 1248: Enum with no members crashes VerCors

The following PVL input program caused VerCors to crash:

```
enum Empty {
}
```

This program is now rejected with a parse error.

Issue 1260: All-underscores name crashes VerCors

Splitting and combining names consisting of only underscores, such as in the following PVL program, caused VerCors to crash:

```
void ___() {
}
```

This has been resolved by passing along such names as-is.

Issue 1261: Unexpected bip_annotation keyword crashes VerCors

The *bip_annotation* keyword is specific to the JavaBIP verification feature [6], but it was also parsed in the C and PVL front-ends, where it caused a crash:

```
bip_annotation
void foo() {
}
```

The occurrence of the *bip_annotation* keyword is now correctly marked as an error.

Issue 1263: Label declarations outside expected scopes crash VerCors

The PVL language has some constructs that are similar to method bodies, including the *run* block, the *constructor* block, and the *vesuv_entry* block. In these blocks, labels (for use with the *goto* statement) were syntactically allowed, but placing them there caused VerCors to crash.

```
class Three {
    run {
        label sixty;
    }
}
```

This has been resolved.

Issue 1264: Special expressions outside channel invariant crash VerCors

As part of the VeyMont [39] support for verifying choreographies, VerCors supports 'channel invariant expressions', specifically including the *\msg* (message), *\sender* and *\receiver* keywords. These expressions only have a meaning inside a *channel_invariant* statement, but were also grammatically accepted in other places, where they caused a crash.

```
requires \msg;
void foo() {
}
```

This is now correctly reported as an error.

Issue 1265: Type of `\type`, `\typeof` expressions is inconsistent

There is some support in the PVL and Java front-ends for referring to the types of expressions, to support the Java *instanceof* operator among other features. However, the type of these expressions is not documented, and different rewriting phases disagreed whether it should be some form of *type<>* or an integer, which meant that the following programs both failed with an error message suggesting the other would succeed:

```
class A {
}

type<A> foo() {
   return \type(A);
}
```

```
class A {
}

int foo() {
   return \type(A);
}
```

The *\type* and *\typeof* expressions are not believed to be widely used.

Issue 1266: Right plus operator overload syntax error crashes VerCors

PVL supports operator overloading, specifically for the plus (addition) operator. This is intended for cases like the string class, where the plus operator is overloaded to represent string concatenation. The class author has the option to decide whether the operator they define should be left-associative by defining *+* or right-associative by defining *right+*. However, the PVL grammar also accepted other keywords, which would then cause a crash:

```
class Th {
    ensures \result;
    bool wrong+(Th a) {
        return true;
    }
}
```

This is now an error.

Issue 1267: Returning a value of type `resource` crashes VerCors

The *resource* type is a 'Boolean-like type' that is intended to be used with VerCors separating conjunction expression, a concept from separation logic (see Section 3.1.2). However, in some cases using this type with regular Boolean expressions in a *return* statement would cause a crash:

```
resource bar() {
    return true;
}

void foo() {
    assert bar();
}
```

This has been worked around by marking returning *resource*s as temporarily unsupported. The input no longer crashes.

Issue 1268: Neglecting to specify type variable crashes VerCors

Attempting to create (using the *new* expression) an instance of a generic type without specifying type variables would cause VerCors to crash:

```
class X<T> {
}

void main() {
    new X();
}
```

This is now reported as a type error that says that the *new* expression only supports non-generic classes; not being able to instantiate generic classes is therefore a language limitation, and no longer a crash.

Issue 1273: Generic classes with final fields crash VerCors

A PVL program like the following would cause a crash when a rewriting phase caused the abstract syntax tree to no longer typecheck:

```
class C <T> {
    final int f;
}
```

Issue 1290: Prover types and prover functions outside PVL crash VerCors

The *prover_type* and *prover_function* spec keywords allow adding definitions directly to the underlying SMT solver, for instance defining a function directly in SMTLIB syntax. This was only supported in PVL, but included in other language grammars as well, which meant that attempting to verify the following as e.g. C code would hit a match error:

```
/*@ prover_function int two() \smtlib `(+ 1 1)`; */
```

Issue 1291: C union declarations are not yet supported

There is no support yet for unions in the C implementation. Normally, this would count as a language limitation (see Section 3.3.4) and so not as an error; however, in this case the omission causes a ParseMatchError, which does cause a crash in our definition.

```
union onion {
    float goat;
};
```

Issue 1292: Non-method members of Java @interfaces not yet supported

Java annotation interfaces are allowed to declare nested classes, interfaces, enums and constrants, but this is not yet supported by VerCors, also triggering a *ParseMatchError*. The following example from the Java Language Specification[1] shows a possible use of this language feature, which appears to be quite rarely used otherwise:

```
@interface Quality {
    enum Level { BAD, INDIFFERENT, GOOD }
    Level value();
}
```

Issue 1293: Non-inline thread-local predicates are not rewritten

The VerCors specification language (here demonstrated using PVL) allows declaring and defining predicates: functions returning 'resources', which are similar to booleans but are used to represent a read or write permission. These predicates can optionally be marked *inline*, *thread_local*, or both. The case where a predicate was marked *thread_local* but not *inline* was not handled by VerCors, which would cause an error just before the program was handed off to the Silver backend:

```
thread_local resource fox();
```

[1] `https://docs.oracle.com/javase/specs/jls/se23/html/`

Issue 1294: C++ front-end fails to parse specific names

A bug in the lexer grammar meant that the following input did not parse as C++:

```
namespace u {
}
```

Issue 1298: lock/unlock statement with literal null crashes VerCors

The *lock* statement takes a non-null expression. This is checked, but the check does not correctly handle a null literal, triggering a UnreachableAfterTypeCheck error.

```
void example() {
    lock null;
}
```

Issue 1299: Empty sequential block triggers crashes VerCors

The VerCors toolset is especially geared towards verifying the correctness of concurrent programs. The *par* (parallel) and *sequential* blocks in PVL allow specifying programs that may involve multiple threads and proving their correctness. However, VerCors expects these *sequential* blocks to be nonempty, triggering a 'tail of empty list' exception.

```
void x() {
    sequential {
    }
}
```

Issue 1300: fork/join statement with literal null crashes VerCors

The PVL *fork* statement expects an object-typed expression of a class that has a run method, but the grammar also allows a literal *null* keyword. This passes at least some type checks, but then causes a ClassCastException in the code that checks whether the expression is a runnable.

```
void spork() {
    fork null;
}
```

Issue 1302: 'old' expression in context or requires clause crashes VerCors

The *\old(expression)* syntax is 'typically used in postconditions (ensures) or loop invariants'. However, in PVL it is also grammatically accepted in other places where an expression is expected, which then causes VerCors to generate an abstract syntax tree that is rejected by the Viper back-end. The rejection is treated as a crash.

```
requires \old(true);
void example() {}
```

Issue 1303: Exponentiation/power expression crashes VerCors

In the documentation for PVL, an exponentiation (power) operator is described, consisting of two caret characters. Attempting to use this operator causes VerCors to crash.

```
void zap() {
    int i = 2 ^^ 3;
}
```

Issue 1304: Simplification rule in program crashes VerCors

VerCors contains a number of axioms (rules) that can be used to simplify inputs. For instance, if an expression $i-i$ appears in a program, with i some number, it can be simplified to 0. These rules are specified in PVL syntax in the file *simplify.pvl*, which is bundled with VerCors. The issue here is that including a simplification rule in the program being verified (so outside *simplify.pvl*) was accepted by the grammar but not correctly handled by VerCors, causing a *ColToSilver$NotSupported* exception.

```
axiom add { 2 + 2 == 4 }
```

Issue 1306: Type error in context_everywhere crashes VerCors

The *context_ everywhere* statement, which is a shorthand to introduce a single expression as a precondition, a postcondition, and a loop invariant for all loops in a method. The expression is expected to be of type *resource* (often a boolean). If some other type of expression is given, this causes an internal type error, specifically an *CoercingRewriter$Incoercible* exception.

```
context_everywhere 1;
void one() {
}
```

Issue 1307: Referring to sibling par inside parallel block can crash VerCors

The following PVL program, which uses the *parallel*, *par* and *barrier* constructs incorrectly, caused VerCors to throw a *TimeTravel* exception.

```
void x() {
    parallel {
        par ty {}
        par barrier(ty) {}
    }
}
```

Issue 1308: Function/predicate arguments of type type<...> crash VerCors

The *type<...>* generic type represents a type; it is the result of the *\typeof* operator (see also issue 1265). However, declaring a function, method or predicate that accepts such a value triggers *ColToSilver$NotSupported*. In PVL, this looks as follows:

```
void x(type<void> y) {
}
```

Issue 1312: Postfix inc/dec without permission crashes VerCors

The following program should result in a permission error, since the 'acters' method accesses the field 'erior' without read or write permissions. Instead, a *BlameUnreachable* exception is thrown: some code asserts that 'assigning to a field should trigger an error on the assignment, and not on the dereference', an assertion that apparently trips on this case as well. The issue occurs in PVL and Java, but not in C or C++. In PVL, it looks as follows:

```
class room {
    int erior;

    char acters() {
        erior++;
    }
}
```

Issue 1313: Label in Java initializer block crashes VerCors

In Java, classes may contain blocks of statements to initialize instance or static members. These blocks are supported by VerCors, but attempting to insert a label into such a block would trigger a *Scopes$NoScope* exception. This issue is similar to issue 1263, above, but in the more rarely used initializer block context. An example input is:

```
class Z {
    { m:; }
}
```

Issue 1316: string keyword in C and C++ spec crashes VerCors

The C and C++ string types (*char** and *std::string*, respectively) are not yet well-supported by VerCors. However, for both languages, the *string* keyword is recognized in the grammar, apparently referring to some specification-level string type that has not been implemented yet. When an input containing a specification-level *string* keyword is parsed and converted to COL, this results in a *ParseMatchError*, as follows:

```
//@ pure string cheese() = "mozzarella";
```

Issue 1330: Names with many trailing digits crash VerCors

In some instances, VerCors will append or strip a trailing numeric suffix from a declaration or label name to disambiguate between different instances. Some code in the *Namer* class assumed that such a suffix would always be parseable as a Java Integer, but this is not the case. A suffix that is too large would raise a *NumberFormatException*, like so:

```
void func_2147483648() {
}
```

Since the *Namer* class is used for all front-ends, the issue applied to all languages supported by VerCors. It has since been fixed.

Issue 1346: Nonexistent variables in LLVM binops crash VerCors

At the time of testing, VerCors supports a preliminary syntax for embedding method contracts in LLVM assembly language files, with the *VC.contract* syntax. Referring to nonexistent variables (here 'a') in a *requires* clause is correctly handled, except in the case where the variable is used in a LLVM binary operator. In this scenario, the usage triggers a *NoSuchElementException* and a crash:

```
define void @example()
!VC.contract !{ !" requires and(%a, %a); " }
{
    ret void
}
```

However, this method contract syntax is slated to be deprecated in the future, in favor of a new contract format that has not been publicly documented yet. Some details about the VerCors integration with LLVM can be found in Section 9.4.