Mutative Fuzzing for an Assembler Compiler

Aynel Gül and Vadim Zaytsev
Raincode Labs
Brussels, Belgium
aynel.gul@hotmail.com and vadim@grammarware.net

Abstract

Compilers play a crucial role in software development, and demands for their correctness are high. Testing them is a challenging process, involving many layers of complexity and many interwoven methodologies. One of the popular techniques that is known to increase the effectiveness of manually written test cases, is fuzzing: seeded with an initial test case, a fuzzer can mutate it to automatically infer multitudes of mutated test cases. Yet, not all is well when you try to apply fuzzing to test an industrial compiler of a legacy assembler-level language, since most research on fuzzing was done on C, and such results are not easy to generalise. In this paper, we explain the complications and show a number of mutation strategies that we successfully applied to fuzz HLASM code. So far we have tested 24000 mutated files and have been able to flag 621 of them as triggering a bug, and found and fixed at least one unique bug in Raincode Assembler Compiler.

Index terms— legacy software, compiler testing, random testing, fuzzing, mutation strategy

1 Introduction

Compilers are known to be among the most important, widely-used and complex software and we expect them to be correct. In practise, however, compilers are significantly vulnerable to bugs [1]. Bugs can lead to crashes or miscompilations [2], resulting in unexpected behavior. Unexpected behaviour can result in serious software failures [3]. A single erroneous compiler can produce many erroneous programs. Given the vital role of compilers in the creation of the code that is executed, it is important that confidence should be created in their correctness.

Compilers are mainly intensively tested by special test suites, which are are manually written and extended over time. However, this traditional way of testing often revolves around positive testing (i.e. testing that features work as specified), rather than negative testing (i.e. testing that the system does not do things that it is not supposed to do). Some errors cannot be found using only positive testing [4, 5]. In addition, it is known that human testers have blind spots. It has been shown that test cases are more effective when they have been written by someone other than the original programmer [5]. It is likely that a blind spot in the implementation also poses a blind spot in testing. Furthermore, as these tests are written by hand, they cost a substantial amount of time and effort, which makes manual tests expensive [5]. It is common to use fuzzing in addition to these test suites in an attempt to overcome these issues [4]. Fuzzing is an approach to software testing where the System Under Test (SUT) is bombarded with test cases generated by another program. The technique relies on providing unexpected or random input, such as randomly generated test programs. Fuzzing revolves around negative testing. As fuzzing focuses on random and unexpected input, it overcomes the blind spot problem. As the process can be automated, it can be more cost-efficient. Fuzzing has shown to be very effective at exposing non-obvious errors that had been missed by other testing techniques [4, 6].

One such well known and successful fuzzing tool is Csmith [7]. Csmith generates random C programs and has proven to be useful for testing compilers and other tools that process C code. The tool is based on differential testing, which implies that test programs

In: D. Di Nucci, C. De Roover (eds.): Proceedings of the 18th Belgium-Netherlands Software Evolution Workshop, Brussels, Belgium, 28-11-2019, published at http://ceur-ws.org

get compiled by different compilers and after their execution the results are compared. Csmith is a generative fuzzer, but there is also a lot of interest recently in research dedicated to mutative approaches, which do not require two or more comparable compilers, and in general require less human effort [5]. Such techniques apply equivalence transformations on existing test programs to automatically generate more test programs. An equivalence transformation is a transformation that is applied to a seed program and that does not change the output of the program. A seed program is a trivial but valid program that is used as the initial program in this method. These mutation strategies can be rather language specific and many different strategies are possible depending on the syntax and grammar.

2 Problem Context

Raincode Labs [8] is a compiler company that works with compilers, interpreters, grammars, refactoring tools, assemblers, etc. We often provide consulting services to other companies and provide compiler services such as migration off the mainframe platform. One of our tools to enable this migration is the Raincode Assembler Compiler [9, 10]. This tool is specifically designed for legacy code bases that contain, besides commonly encountered COBOL, PL/I and other "third generation languages", also some code in the IBM assembler. HLASM, or High Level Assembler, is IBM's low level programming language for z/Architecture mainframe computers. Just like other assembler languages, the assembler language supported by HLASM is specific to a particular computer architecture and operating system. As HLASM is an assembler language, it is very close to the machine language in both form and content. HLASM operates under the z/OS operating system, the CMS component of the z/VM operating system, the z/VSE operating system, and Linux for System z [11]. The Raincode Assembler Compiler is a relatively new tool and currently its integration testing infrastructure mainly consists of 40 manually written issue-driven test cases.

To increase the reliability of a compiler, it is common to apply complementary testing methods [12], as manually written tests presumably lack coverage. However, random testing of assemblers is uncharted territory. Tools like Orange4 [13] and Orion [14] are examples of successful random compiler testing tools that are based on mutative testing[1]. Their work suggests that the ideas behind methods of random compiler testing are generic. However, these tools are based on the syntax of the C programming language to specifically test C compilers. As different programming languages have different syntax, it is not possible to use the same tool for a syntactically different programming language. For this reason, it is unclear if and how random testing can be applied to HLASM.

In the next section we provide an overview of related work on testing compilers with generated test data, covering both generative (subsection 3.2) and mutative (subsection 3.1) approaches.

3 Related work

Despite active ongoing initiatives in compiler verification [17], testing is still the dominant technique that is used for assuring compiler quality [9, 12]. The most common compiler testing approach, also used at Raincode Labs, is manually creating, growing and maintaining a test suite for each compiler implementation. There are also popular commercial compiler test suites available, such as PlumHall [18] and SuperTest [19]. As test suites presumably lack coverage, other methods like random testing are used to provide extra testing coverage [5]. Random testing of compilers is not a replacement of test suites, but rather a complementary method. Other comparable methods aimed specifically at extending existing test suites, are called test plan augmentation [20] and test suite augmentation [21–24], or simply test augmentation [25]. Out of those, metamorphic testing appears to be the closest to our approach [26].

As suggested by McNally et al [5], we divide the related random testing work into two categories: mutative approaches and generative approaches. For a comprehensive overview of all compiler testing methods available before 2005, we refer to Kossatchev and Posypkin [27].

3.1 Mutative approaches

In our research, we explore possibilities for applying random compiler testing with equivalence transformation on HLASM. Equivalence transformation is a mutative testing method. At the moment we are unaware of research that does similar work related to equivalence transformation and HLASM. However, there are several projects that apply the same technique of mutative testing, which we also drew inspiration from. The initial approach to mutative compiler testing was presented by Tao et al. [28]. They present the Mettoc tool, which concentrates on open-source compilers for C. Mettoc was able to generate equivalent programs and to expose multiple bugs. A more recent tool, the Orange4 presented by Nakamura and

[1]Following McNally et al [5], we use the term "mutative testing", which is not to be confused with much researched nowadays "mutation testing" [15, 16]. Mutative testing techniques tweak test cases to produce more test cases to improve test suite quality. Mutation testing techniques tweak the code under test to measure test suite quality.

Ishiura [13], introduced a new mutation method for generating C programs. They start with a trivial initial program that includes only "`return 0;`" in the main function and repeatedly apply equivalence transformations. Examples are new variable declarations and if/for-statements. The Orange4 was able to detect bugs in the latest versions of GCC and LLVM. Their research is currently limited to C and to the above mentioned trivial seed program.

Le et al. [14] introduce Orion, which is based on a new technique called Equivalence Modulo Inputs, or in short: EMI. In contrast to Orange4, they take existing real-world code and transform it in a systematic way to produce equivalent variants of the original code. More precisely, they analyse real-life code by detecting dead code and then prune these sections stochastically. They have been able to find 147 new bugs in 3 big C compilers in 11 months. This technique is said to be generally applicable, but has only been tested with C compilers. Also, the only mutation strategy used is pruning. Moreover, the proposed technique only works on, and with, dead code. It does not include live code in the transformations, which means that the applied transformations are limited to deleting code fragments which are not on the execution path of the programs.

Complementary to Orion, Proteus [29], Athena [30] and Hermes [31] are other examples of recent random test tools that have proven to be successful. These three tools are also based on equivalence transformation by generating single-file test programs from existing ones. It transforms a seed program into multiple compilation units and randomly assigns each of them to an optimisation level. In addition to Orion they support both deletion and insertion. However, the transformations of Athena and Proteus are still and all limited to code fragments that are not on the execution paths of the programs. Hermes, however, also applies equivalence transformations on live code.

3.2 Generative approaches

A very successful compiler testing tool, known as Csmith [7], has been able to find several hundred bugs in C compilers GCC and LLVM. Their method is based on differential testing, thus their generated programs are compared across compilers and compiler versions to detect potential deviant behavior. Csmith covers a broad range of syntax, including arrays, function calls, loop statements, etc. They avoid undefined behaviour by placing conservative restrictions on the syntax of the generated test programs [13]. Csmith is limited to C and their technique is limited to cases that have two or more comparable compilers.

Another generative approach, presented by Lindig [32], concentrates on generating C programs with complex data structures. These are loaded with constant values and passed to functions that check the received values. These received values are compared to the expected values. Any inconsistencies are exposed by an assertion failure. The tool, named Quest, is not based on differential testing as their tests are self-checking. Quest was mainly used to test GCC, LCC and ICC and they were able to find 13 bugs.

Alternatively, generation of test programs can revolve around extensive coverage of the language—or, as a proxy, coverage of the grammar of the language. For instance, there is a famous algorithm of generating a small set of short test sentences from a context-free grammar, initially proposed by Purdom [33] and later extended with backtracking [34], actions [35], controlled coverage [36], etc. Just as with mundane software testing, in parser testing rule coverage is known to be important but fundamentally insufficient [37]. Coverage can be seen as two-dimensional [38] with a syntactic axis (nonterminals, rules, etc.) and a semantic one (computations, calculations, attributes, etc). Fischer et al. [39] use grammar-based test data generators to compare several alternative grammars for reportedly "the same" language.

As for making the next step and going from the test case exposing a possible defect, to the place where defect can be found and fixed, recently it has been shown that spectrum-based fault localisation methods can help to identify problematic grammar rules corresponding to test case failures [40]. The applicability of this method to HLASM remains to be investigated, since it requires instrumentation of the parser to collect grammar spectra.

4 Equivalence transformation

4.1 Overview

In this section, we present a method for automatically generating equivalent programs using a mutative approach. Valid HLASM programs are generated starting from a trivial seed program and by repeatedly applying mutations.

When a seed program P is fed to the compiler C, it produces an executable E (1).

$$c : P \to E \tag{1}$$

The behaviour of the executable $c(P)$ ought to behave exactly as the semantic of the seed program P prescribes. As follows, if a seed program satisfies a certain property, then the executable should satisfy the same property (2) [28].

$$R(P_1, P_2, \ldots, P_n) \Rightarrow R(c(P_1), c(P_2), \ldots, c(P_n)) \tag{2}$$

When we create equivalent programs, we expect them to behave the same as the seed program when they have the same input (3) [28].

$$\begin{gathered} P_1 \equiv P_2 \equiv \cdots \equiv P_n \\ \Rightarrow \\ c(P_1) \equiv c(P_2) \equiv c(\ldots) \equiv c(P_n) \end{gathered} \tag{3}$$

The HLASM language consists of different types of instructions, namely machine instructions, assembler instructions and macro instructions. As our method is mutative and the results should be equivalent, it is very important to preserve the semantics of the seed program. As the assembler language lies very close to the machine language, every little change can have a big impact as you can work on byte- or even bit level. This is especially true for machine instructions and assembler instructions. These given instructions provide the opportunity to, for example, make jumps in the program specified by bytes rather than symbolic addresses. However, the third type of instructions, macro instructions, have shown a lot more mutative testing potential by providing an environment that is less prone to semantic changes. As macro instructions are more high-level than machine- and assembler instructions, it provides better means for mutative testing. With macro instructions we were able to apply mutations that created valid programs that did not alter the semantics of the program.

As for the seed programs, we use Raincode's 40 issue-driven test cases, as these test cases have an expected output.

4.2 Mutation strategies

All mutation strategies are restricted to strategies that preserve the semantics of the seed program. Below we give four examples of mutation strategies that we have already implemented and deployed.

1. **Label mutation** simply changes the name of a label in the program along with all its occurrences. If this is done consistently, the behaviour of all branching instructions will not change.

2. **Simple code injection** involves adding new valid instructions and expressions to the program. Examples are casting instructions and declarations. For HLASM, this is only possible to perform on the level of macro instructions, since injecting actual assembler instructions will change the byte code representation of the compiled program, and semantic preservation cannot be guaranteed.

3. **Expression derivation** uses mathematical properties to generate or mutate arithmetic expressions. It uses a target value as a starting point and applies mathematical operations that evaluate to the target value. In our current implementation these mathematical operations can be addition, subtraction or division, selected at random. The target value is randomly selected from what can be guaranteed by the current state of the program.

4. **Control structure mutation** allows to control the sequence in which the statements of the program are processed by the assembler. By adding certain macro branching instructions, we are able to mutate the control flow without changing the byte code representation of the program, and thus preserving its behaviour. Examples are unconditional branches or branches depending on logical expressions. No original lines are skipped in the process.

5 Experimental results

We fed the mutated files generated by our tool to the Raincode Compiler Tester (a proprietary regression testing tool we use to test all our compilers) and investigated if we were able to detect any bugs. For this experiment we generated HLASM programs with 3 different numbers of mutation rounds: 150, 300 and 600 (the higher the number, the farther the result from the seed). With each seed program we generated 200 equivalent programs. All mutation strategies were included during the mutation phase.

In addition, we generated HLASM programs with single mutation strategies to be able to compare them. We generated 1000 programs for each strategy. Each program underwent 150 mutations.

In total, the Raincode Compiler Tester flagged 621 programs out of the 24000 mutated programs produced with our tool. Per round of 8000 mutated programs, an average of 207 mutated programs were flagged. An overview of these results can be found in Table 1. An important side note to these results is that not all flagged files indicate unique bugs, as one bug can trigger multiple miscompilations.

When running experiments with single mutation strategies, we were able to flag 30 mutated files with the simple code injection strategy. We were not able to flag any mutated files with the other three mutation strategies. An overview of these results can be found in Table 2.

Table 1: Overview of first experiment results

Mutations	**Mutated files**	**Flagged files**	**% flagged files**
150	8000	171	2.14
300	8000	224	2.80
600	8000	226	2.83

```
NONTERMINAL CLASS CDuplicationFactor;
  INHERITS BaseNonTerminal;
  GRAMMAR  LPar TheCount RPar;
  VAR      TheCount: CArithExpression;
  METHOD Apply(scope: Scope; string: STRING): STRING;
  VAR    count: INTEGER;
  BEGIN
    count := TheCount.Evaluate(scope);
    RESULT.CREATE(count*string.SIZE);
    FOR I := 0 TO count - 1 DO
      FOR J := 0 TO string.SIZE DO
      FOR J := 0 TO string.SIZE - 1 DO
        RESULT[I*string.SIZE+J] := string[J];
      END;
    END;
  END Apply;
END CDuplicationFactor;
```

Figure 1: An example of a bug acknowledged by other Raincode engineers and fixed. The red marking indicates the line of code that contains the bug. This line was removed when the bug was fixed. The green marking indicates the fix. This line was added when the bug was fixed.

One flagged file has already been chosen, analysed and acknowledged as a bug (see Figure 1). The bug was fixed in the process. The acknowledged and fixed bug clearly falls into an off-by-one category, since it iterates one too many times over an array and produces an out of bound exception that terminates the compilation process abnormally.

6 Conclusion

In this short paper we have presented a complementary method for HLASM compiler random testing based on equivalence transformation of seed test programs. To the best of our knowledge, this is the first attempt to apply mutative testing to a compiler of an assembler-level language such as HLASM. The main problem with the inapplicability of straightforward C-level equivalence transformations to HLASM, lies in the fact that assembler-level programs depend on their own bytecode representation: use code as data and data as code, modify themselves at runtime, perform relative jumps, etc. After noticing this inapplicability, we still found a way to apply mutative compiler testing to HLASM, if operating on the level of the macro language. Thus, our prototype is limited to macro code, but it might be a fundamental limitation that can only be lifted by imposing heavy and uncheckable restrictions on the seed programs, which would make it possible to fuzz machine instructions and assembler instructions in some contexts. We have presented a list of possible mutation strategies and explored their added value: the list is open for extension, and one should read it as a list of examples, as well as the list of strategies that have already been implemented and shown to work. With the described infrastructure, we were able to find at least one unique bug, which has then been easily fixed. The finding of more unique bugs is considered possible, as many flagged files have to be analysed in the nearest future.

Table 2: Overview of second experiment results

Mutation strategy	**Mutated files**	**Flagged files**
Label mutation	150	0 (0%)
Simple code injection	150	30 (3%)
Expression derivation	150	0 (0%)
Control structure mutation	150	0 (0%)

References

[1] Anna Zaks and Amir Pnueli. Covac: Compiler validation by program analysis of the cross-product. In *International Symposium on Formal Methods*, pages 35–51. Springer, 2008.

[2] Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. In *ACM SIGPLAN Notices*, volume 51, pages 849–863. ACM, 2016.

[3] Xavier Leroy. Formally Verifying a Compiler: Why? How? How Far? In *Proceedings of the 9th International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 2011.

[4] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. *IEEE Trans. Reliability*, 67(3):1199–1218, 2018.

[5] Richard McNally, Ken Yiu, Duncan Grove, and Damien Gerhardy. Fuzzing: the State of the Art. Technical report, Defence Science and Technology Organisation Edinburgh (Australia), 2012.

[6] Barton P Miller, Louis Fredriksen, and Bryan So. An Empirical Study of the Reliability of

UNIX Utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[7] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In Mary W. Hall and David A. Padua, editors, *PLDI*, pages 283–294. ACM, 2011.

[8] Raincode Labs. http://www.raincodelabs.com, 2016.

[9] Volodymyr Blagodarov, Ynes Jaradin, and Vadim Zaytsev. Raincode Assembler Compiler. In Tijs van der Storm, Emilie Balland, and Dániel Varró, editors, *Proceedings of the Ninth International Conference on Software Language Engineering (SLE)*, pages 221–225, 2016.

[10] Raincode. The Raincode ASM370 compiler for .NET and .NET Core. https://www.raincode.com/technical-landscape/asm370/, 2016.

[11] SC26-4940-06. High Level Assembler for z/OS & z/VM & z/VSE. Language Reference. Version 1 Release 6. IBM, http://publibz.boulder.ibm.com/epubs/pdf/asmr1021.pdf, 2013.

[12] Vadim Zaytsev. An Industrial Case Study in Compiler Testing. In David J. Pearce, Tanja Mayerhofer, and Friedrich Steimann, editors, *Proceedings of the 11th International Conference on Software Language Engineering (SLE)*, pages 97–102. ACM, 2018.

[13] Kazuhiro Nakamura and Nagisa Ishiura. Random testing of C compilers based on test program generation by equivalence transformation. In *APCCAS*, pages 676–679. IEEE, 2016.

[14] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In Michael F. P. O'Boyle and Keshav Pingali, editors, *Proceedings of the 35th Conference on Programming Language Design and Implementation*, page 25. ACM, 2014.

[15] Pedro Reales Mateo, Macario Polo, José Luis Fernández Alemán, Ambrosio Toval, and Mario Piattini. Mutation testing. *IEEE Software*, 31(3):30–35, 2014.

[16] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation Testing Advances: An Analysis and Survey. *Advances in Computers*, 112:275–378, 2019.

[17] Maulik A. Dave. Compiler verification: A bibliography. *ACM SIGSOFT Software Engineering Notes*, 28(6):2, 2003.

[18] Plum Hall, Inc. The Plum Hall Validation Suite for C. http://www.plumhall.com/stec.html.

[19] ACE. SuperTest compiler test and validation suite. http://www.ace.nl/compiler/supertest.html.

[20] Wen Chen, Kuo-Kai Hsieh, Li-Chung Wang, and Jayanta Bhadra. Data-Driven Test Plan Augmentation for Platform Verification. *IEEE Design & Test*, 34(5):23–29, 2017.

[21] Zhihong Xu. Directed test suite augmentation. In Richard N. Taylor, Harald Gall, and Nenad Medvidović, editors, *Proceedings of the 33rd International Conference on Software Engineering*, pages 1110–1113. ACM, 2011.

[22] Tingting Yu. TACO: test suite augmentation for concurrent programs. In *ESEC-FSE*, pages 918–921. ACM, 2015.

[23] Zhihong Xu, Yunho Kim, Moonzoo Kim, Myra B. Cohen, and Gregg Rothermel. Directed test suite augmentation: an empirical investigation. *Software Testing, Verification & Reliability*, 25(2):77–114, 2015.

[24] Raúl A. Santelices, Pavan Kumar Chittimalli, Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Test-Suite Augmentation for Evolving Software. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 218–227. IEEE, 2008.

[25] Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Andy Zaidman, Martin Monperrus, and Benoit Baudry. A Snowballing Literature Study on Test Amplification. *Journal of Systems and Software*, 157, 2019.

[26] Zhi Quan Zhou and Liqun Sun. Metamorphic Testing of Driverless Cars. *Communications of the ACM*, 62(3):61–67, February 2019.

[27] A. S. Kossatchev and M. A. Posypkin. Survey of Compiler Testing Methods. *Programming and Computing Software*, 31:10–19, January 2005.

[28] Qiuming Tao, Wei Wu, Chen Zhao, and Wuwei Shen. An automatic testing approach for compiler based on metamorphic testing technique. In *Asia Pacific Software Engineering Conference*, pages 270–279. IEEE, 2010.

[29] Vu Le, Chengnian Sun, and Zhendong Su. Randomized Stress-testing of Link-time Optimizers. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 327–337. ACM, 2015.

[30] Vu Le, Chengnian Sun, and Zhendong Su. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *ACM SIGPLAN Notices*, volume 50, pages 386–399. ACM, 2015.

[31] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458. USENIX, 2012.

[32] Christian Lindig. Random Testing of C Calling Conventions. In *Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging*, pages 3–12. ACM, 2005.

[33] Paul Purdom. A Sentence Generator for Testing Parsers. *BIT Numerical Mathematics*, 12(3):366–375, 1972.

[34] Uwe Kastens. Studie zur Erzeugung von Testprogrammen für Übersetzer. Bericht 12/80, Institut für Informatik II, University Karlsruhe, 1980.

[35] Augusto Celentano, S. Crespi Reghizzi, P. Della Vigna, C. Ghezzi, G. Granata, and F. Savoretti. Compiler Testing Using a Sentence Generator. *Software: Practice and Experience*, 10(11):897–918, 1980.

[36] Ralf Lämmel and Wolfram Schulte. Controllable Combinatorial Coverage in Grammar-Based Testing. In Umit Uyar, Mariusz Fecko, and Ali Duale, editors, *Proceedings of the 18th IFIP TC6/WG6.1 International Conference on Testing of Communicating Systems (TestCom'06)*, volume 3964 of *LNCS*, pages 19–38. Springer Verlag, 2006.

[37] Ralf Lämmel. Grammar Testing. In *Proceedings of the Fourth International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 2029 of *LNCS*, pages 201–216. Springer, 2001.

[38] Jörg Harm and Ralf Lämmel. Two-dimensional Approximation Coverage. *Informatica Journal*, 24(3), 2000.

[39] Bernd Fischer, Ralf Lämmel, and Vadim Zaytsev. Comparison of Context-free Grammars Based on Parsing Generated Test Data. In Uwe Aßmann and Anthony Sloane, editors, *Post-proceedings of the Fourth International Conference on Software Language Engineering (SLE 2011)*, volume 6940 of *LNCS*, pages 324–343. Springer, 2012.

[40] Moeketsi Raselimo and Bernd Fischer. Spectrum-based fault localization for context-free grammars. In Oscar Nierstrasz, Jeff Gray, and Bruno C. d. S. Oliveira, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering (SLE)*, pages 15–28. ACM, 2019.