Expanding the Scope of Grammar-Based Enumerative Testing

THEA U. KJELDSMARK, University of California Irvine, USA

Enumerated testing has emerged as a systematic approach to generate test cases and address the limitations of traditional fuzzing techniques. One of these tools is ET, a grammar-based exhaustive enumerator used to find bugs in SMT solvers using context-free grammars. In this work, we present ET++, an enhanced version of the original ET tool. ET++ expands grammar-based exhaustive enumerative testing to support context-free grammars for any language. By integrating ANTLR parsing and improving grammar translation, ET++ addresses the limitations of its predecessor, enabling it to handle more complex and varied grammar definitions across languages. Using ET++, we identified 14 new bugs in SMT solvers and the JavaScript interpreter Espruino, emphasizing ET++'s effectiveness in uncovering bugs even with small test cases.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Grammar-based enumeration, SMT solvers, JavaScript environments

1 Introduction

Ensuring the correctness and robustness of programming languages and satisfiability modulo theory (SMT) solvers is crucial, as they are foundational tools in software development and verification processes. Fuzzing has been instrumental in uncovering vulnerabilities in languages like JavaScript [11, 12, 20] and bugs in SMT solvers such as Z3 and CVC [4, 16, 23]. However, fuzzing is often unsystematic and random, leading to potentially missed bugs. The inherent randomness of fuzzing can also generate large test cases, which often complicates the process of identifying and isolating the root cause of detected bugs for developers.

Instead, previous research has proposed enumerated testing to generate test cases systematically. One of these tools is ET [22], a grammar-based enumeration tool used to find bugs in SMT solvers. ET leverages FEAT [7], a functional enumeration library, to generate test cases based on context-free grammars. Given a number of tests N and a grammar, the generation starts at the smallest possible test case and continues to the N-smallest input for the specified grammar. This allows ET to take advantage of the *small scope hypothesis* [1], which states that most bugs can be found by testing small inputs. Despite its effectiveness, ET focuses only on SMT solvers, necessitating additional expansion for broader applicability. In this work, we present ET++ that supports context-free grammars for any language, evaluate its effectiveness using SMT and JavaScript grammars, and demonstrate its capability by uncovering 14 new bugs.

2 Our Approach

Design. Given a context-free grammar in the ANTLR format, ET++ first compiles it to algebraic data types in Haskell. Then, the FEAT enumerator uses these types to realize a test generator for differential testing. In contrast to ET's simple string parsing of input grammars, ET++ leverages ANTLR's lexer and parser to parse the input grammar. The enhanced translator then converts the resulting output from the context-free grammar to a regular tree grammar. One challenge with generating test cases from grammars is the possibility of encountering recursive rules that can lead to infinite generation paths. To address this issue, ET++ uses a combination of depth-first search to identify recursive paths and FEAT's "pay" feature to make these paths more costly. Figure 1 shows an example of the generated Haskell types for the simple recursive plus statement in Figure 2.

The improved parsing and translation approach enables the use of grammars for any language and overcomes ET's limitation of not allowing grammar rules with more than seven arguments.

Author's Contact Information: Thea U. Kjeldsmark, tkjeldsm@uci.edu, University of California Irvine, USA
Research Advisor: Dominik Winterer, dominik.winterer@inf.ethz.ch.

```
start: plus_stmt;
var_name: 'INT_VAR';
plus_stmt: var_name | plus_stmt Plus plus_stmt;
Plus: '+';
```

Fig. 1. Grammar for a simple plus statement grammar.

```
data Plus_stmt = C0_plus_stmt Var_name | C1_plus_stmt Plus_stmt Plus Plus_stmt

instance Show Plus_stmt where
  show (C0_plus_stmt var_name) = show var_name
  show (C1_plus_stmt plus_stmt plus plus_stmt2) =
    show plus_stmt ++ " " ++ show plus ++ " " ++ show plus_stmt2

instance Enumerable Plus_stmt where
  enumerate = share $ aconcat [c1 C0_plus_stmt, pay(c3 C1_plus_stmt)]
```

Fig. 2. Haskell code generated for the grammar shown in Figure 1.

Furthermore, it enables the use of ANTLR's operators, such as optional (?), repetition (*), and alternatives (|), allowing for more flexible grammar definitions.

Grammar Engineering. A key aspect of ET++ are the input grammars. For this work, we derived grammars for SMT and JavaScript using the ANTLR grammar syntax. For SMT, we focused on grammars that mix types, including arrays/integers, arrays/reals, bit vectors/floats, and real/integers, since the original ET work extensively tested grammars with one type. For JavaScript, we developed both general and specific grammars. We used the respective grammars from the ANTLR repository[1] for the general grammars and adjusted them to remove unsupported rule formats. These grammars allow us to test various language features in the same test generation. However, unless the number of tests N is sufficiently large, the generated test cases are often very simple due to potential combinatorial blow-up of the many rule options in a general grammar. To address this, we also develop specific grammars that target a particular part of the language. For example, for JavaScript, we have grammars that focus on language aspects such as arrays, conversions, functions, and classes.

3 Evaluation

To evaluate ET++, we used the developed grammars to generate test cases. We employed differential testing for SMT solvers and JavaScript to compare outputs across multiple implementations. Using this approach, we found 6 SMT bugs and 8 bugs in the JavaScript interpreter Espruino, summarized in Table 1. The SMT testing uses a differential oracle with the solvers Z3 [5] and cvc5 [3]. For JavaScript testing, we employed a differential oracle approach across six environments: Node.js [17], Hermes [13], Escargot [8], QuickJS [19], Espruino [9], and Deno [6]. Besides looking for crashes and unexpected behaviors, we also compared the output of potential operations using `console.log()` and `print()` when applicable. This allows us to identify discrepancies that might indicate potential bugs or non-standard implementations in specific engines.

One example of a test case that led to a bug is shown in Figure 3, where Espruino returned a different value than the other environments. The issue stems from an incorrect handling of the evaluation order with `a[0]` being treated as a location value instead of immediately evaluating it

[1]https://github.com/antlr/grammars-v4

Table 1. Bug Status and Types

Status	Z3	cvc5	Espruino	Total
Reported	1	5	8	14
Confirmed	0	4	4	8
Fixed	0	2	4	6
Duplicate	0	0	1	1
Won't fix	0	0	3	3

(a)

Type	Z3	cvc5	Espruino	Total
Correctness	1	0	8	9
Crash	0	5	0	5

(b)

```
>var a = [1]; a.unshift(2); console.log((2 / (a[0] / a.reverse().length)));
=4 // expected 2 from (2 / (2 / 2))
```

Fig. 3. Bug #2547 in Espruino found using a test case generated by ET++.

as a read value. Thus, by evaluating `a.reverse()` first, `a[0]` evaluated to the incorrect value. The developers confirmed the issue as a bug and have fixed it since our report[2].

4 Related Work

Prior work has proposed various approaches to automated testing of programming languages through execution-based methods, with many incorporating grammar-based techniques. These can be classified as white box, grey box, or black box, depending on the level of internal knowledge of the system under test. Grammar-based white box fuzzing often uses symbolic execution to create constraints on the program inputs [10, 18]. Grammar-based grey box fuzzing has some knowledge of the system, such as code coverage [2, 21]. However, having full or partial knowledge of the tested system might not be feasible. White box approaches may struggle with large systems and grey box techniques may rely on heuristics that can miss certain test cases. In contrast, grammar-based black box fuzzing, which does not require any knowledge about the source code, has been applied to compilers [24] and interpreters for languages like JavaScript [14, 15]. While ET++ classifies as black box, our enumeration approach is bounded and nonrandom, allowing for a more systematic exploration of the input space.

5 Conclusion and Future Work

ET++ expands the scope of grammar-based exhaustive testing to support grammars for any context-free language. By leveraging ANTLR's parsing capabilities and introducing a more flexible grammar translation process, we overcome the major limitations of the original ET. Our evaluation of ET++ on SMT solvers and JavaScript environments demonstrates its ability to uncover bugs even from small test cases. Future work includes searching for more bugs in SMT solvers, JavaScript, and other programming language implementations. Additionally, we aim to optimize our grammar engineering to ensure a balance between the small scope hypothesis and including the complexity that might be necessary for mature languages or engines.

References

[1] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. 2003. Evaluating the "Small Scope Hypothesis". *Proceedings of ACM Symposium on the Principles of Programming Languages (POPL)* (2003).

[2]https://github.com/espruino/Espruino/issues/2547

[2] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)* (2019).

[3] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS*.

[4] Robert Brummayer and Armin Biere. 2009. Fuzzing and delta-debugging SMT solvers. In *Proceedings of the International Workshop on Satisfiability Modulo Theories (SMT)*.

[5] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS/ETAPS)*.

[6] deno [n. d.]. Deno. https://deno.com/. [Online; accessed 20-Oct-2024].

[7] Jonas Duregård, Patrik Jansson, and Meng Wang. 2012. Feat: functional enumeration of algebraic types. In *Proceedings of the 2012 Haskell Symposium*.

[8] escargot [n. d.]. Escargot. https://github.com/Samsung/escargot. [Online; accessed 20-Oct-2024].

[9] espruino [n. d.]. Espruino. https://www.espruino.com/. [Online; accessed 20-Oct-2024].

[10] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[11] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*.

[12] Xiaoyu He, Xiaofei Xie, Yuekang Li, Jianwen Sun, Feng Li, Wei Zou, Yang Liu, Lei Yu, Jianhua Zhou, Wenchang Shi, and Wei Huo. 2021. SoFi: Reflection-Augmented Fuzzing for JavaScript Engines. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[13] hermes [n. d.]. Hermes. https://github.com/facebook/hermes. [Online; accessed 20-Oct-2024].

[14] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST)*.

[15] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *Proceedings of the USENIX Conference on Security Symposium*.

[16] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[17] nodejs [n. d.]. Node.js. https://nodejs.org/. [Online; accessed 20-Oct-2024].

[18] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2016. Model-based whitebox fuzzing for program binaries. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[19] quick [n. d.]. QuickJs. https://bellard.org/quickjs/. [Online; accessed 20-Oct-2024].

[20] Ye Tian, Xiaojun Qin, and Shuitao Gan. 2021. Research on Fuzzing Technology for JavaScript Engines. In *Proceedings of the International Conference on Computer Science and Application Engineering (CSAE)*.

[21] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: grammar-aware greybox fuzzing. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[22] Dominik Winterer and Zhendong Su. 2024. Validating SMT Solvers for Correctness and Performance via Grammar-Based Enumeration. *Proceedings of the ACM on Programming Languages (PACMPL)* (2024).

[23] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT solvers via semantic fusion. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[24] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.