RandiCheck lit review

Tyler Swan

October 2024

[1] abstract: Property-Based Testing requires the programmer to write suitable generators, i.e., programs that generate (possibly in a random way) input values for which the program under test should be run. However, the process of writing generators is quite a costly, error-prone activity. In the context of Property-Based Testing of Erlang programs, we propose an approach to relieve the programmer from the task of writing generators. Our approach allows the automatic, efficient generation of input test values that satisfy a given specification. In particular, we have considered the case when the input values are data structures satisfying complex constraints. That generation is performed via the symbolic execution of the specification using constraint logic programming.

notes: this appears to be the closest to our idea and is done in Erlang using PropEr and CLP (Prolog) with 6 parts:

- a. A translator from PropEr to CLP, which converts types and functions used by Erlang to CLP
- b. A type-based generator, which implements a CLP predicate typeof(X,T) that generates datum X of any given (predefined or user-defined) type T
- c. A CLP interpreter for filter functions (part of erlangs list stdlib)
- d. A value generator, which takes as input a term produced by parts b and c and subjects them to constraints
- e. A translator from CLP to Erlang, which converts d to an Erlang value
- f. A property evaluator, which evaluates, using the Erlang system, the boolean Erlang expression Prop whose inputs are the values produced by e

code can be found here

[2] abstract: Property-based testing is the process of selecting test data from userspecified properties fro testing a program. Current automatic property-based testing techniques adopt direct generate-and-test approaches for this task,

consisting in generating first test data and then checking whether a property is satisfied or not. are generated at random and rejected when they do not satisfy selected coverage criteria. In this paper, we propose a technique and tool called FocalTest, which adopt a test-and-generate approach through the usage of constraint reasoning. Our technique utilizes the property to prune the search space during the test data generation process. A particular difficulty is the generation of test data satisfying MC/DC on the precondition of a property, when it contains function calls with pattern matching and high-order functions. Our experimental results show that a non-naive implementation of constraint reasoning on these constructions outperform traditional generation techniques when used to find test data for testing properties.

notes: Focalize is a functional language (similar to ML) that allows both programs and properties to be implemented into the same environment. The paper similar to the previous one also relies on using a prolog interpreter to parse specifications Focalize includes via program specifications in the language, for example:

```
let rec app(L, G) = match L with
              | [] -> G
              | H :: T -> H :: app(T, G);
let rec rev aux(L, LL) = match L with
              | [] -> LL
              | H :: T -> rev aux(T, H :: LL);
let rev(L) = rev aux(L, []);

property rev prop : all L L1 L2 :list(int),
L = app(L1, L2) -> rev(L) = app(rev(L2), rev(L1));
```

where the above is a program to reverse a list and the property component works as a specification to show that reversing is the same as appending backwards

[3] abstract: Fuzz testing is an effective technique for finding security vulnerabilities in software. Traditionally, fuzz testing tools apply random mutations to well-formed inputs of a program and test the resulting values. We present an alternative whitebox fuzz testing approach inspired by recent advances in symbolic execution and dynamic test generation. Our approach records an actual run of the program under test on a well-formed input, symbolically evaluates the recorded trace, and gathers constraints on inputs capturing how the program uses these. The collected constraints are then negated one by one and solved with a constraint solver, producing new inputs that exercise different control paths in the program. This process is repeated with the help of a code-coverage maximizing heuristic designed to find defects as fast as possible. We have implemented this algorithm in SAGE (Scalable, Automated, Guided Execution), a new tool employing x86 instruction-level tracing and emulation for

whitebox fuzzing of arbitrary file-reading Windows applications. We describe key optimizations needed to make dynamic test generation scale to large input files and long execution traces with hundreds of millions of instructions. We then present detailed experiments with several Windows applications. Notably, without any format-specific knowledge, SAGE detects the MS07-017 ANI vulnerability, which was missed by extensive blackbox fuzzing and static analysis tools. Furthermore, while still in an early stage of development, SAGE has already discovered 30+ new bugs in large shipped Windows applications including image processors, media players, and file decoders. Several of these bugs are potentially exploitable memory access violations.

notes: I think this paper is a bit too low level for a good comparison, considering it works via tracing x86 asm to figure out what inputs to generate in order to fufil coverage of a program, even to the point of creating absurd input data and leading to crashes that are impossible in the real world, I still found it an interesting paper.

[4] abstract: We present a new symbolic execution tool, KLEE, capable of automatically generating tests that achieve high coverage on a diverse set of complex and environmentally-intensive programs. We used KLEE to thoroughly check all 89 stand-alone programs in the GNU COREUTILS utility suite, which form the core user-level environment installed on millions of Unix systems, and arguably are the single most heavily tested set of open-source programs in existence. KLEE-generated tests achieve high line coverage — on average over 90% per tool (median: over 94%) — and significantly beat the coverage of the developers' own hand-written test suite. When we did the same for 75 equivalent tools in the BUSYBOX embedded system suite, results were even better, including 100% coverage on 31 of them.

notes: This is more search based than constraint based, it compiles programs to llvm asm and then interprets them, using heuristics to guess what steps to take in order to increase code coverage

[5] abstract: How can we generate valid system inputs? Grammar-based fuzzers are highly efficient in producing syntactically valid system inputs. However, programs will often reject inputs that are semantically invalid. We introduce ISLa, a declarative specification language for context-sensitive properties of structured system inputs based on context-free grammars. With ISLa, it is possible to specify input constraints like a variable has to be defined before it is used, the 'file name' block must be 100 bytes long, or the number of columns in all CSV rows must be identical. Such constraints go into the ISLa fuzzer, which leverages the power of solvers like Z3 to solve semantic constraints and, on top, handles quantifiers and predicates over grammar structure. We show that a

few ISLa constraints suffice to produce 100% semantically valid inputs while still maintaining input diversity. ISLa can also parse and precisely validate inputs against semantic constraints. ISLa constraints can be mined from existing input samples. For this, our ISLearn prototype uses a catalog of common patterns, instantiates these over input elements, and retains those candidates that hold for the inputs observed and whose instantiations are fully accepted by input-processing programs. The resulting constraints can then again be used for fuzzing and parsing.

Notes: This paper relies on being able to firstly have a correct and properly formatted grammar for a lanugage, and secondly either creating your own constraints or debugging self learned ones. while interesting and theoretically language agnostic, it does require an awful lot of effort to get to a good point for use as opposed to the tailored ones previously that 'just work' for target platforms.

[6] abstract: Property-based random testing, exemplified by frameworks such as Haskell's QuickCheck, works by testing an executable predicate (a property) on a stream of randomly generated inputs. Property testing works very well in many cases, but not always. Some properties are conditioned on the input satisfying demanding semantic invariants that are not consequences of its syntactic structure—e.g., that an input list must be sorted or have no duplicates. Most randomly generated inputs fail to satisfy properties with such sparse preconditions, and so are simply discarded. As a result, much of the target system may go untested. We address this issue with a novel technique called coverage guided, property based testing (CGPT). Our approach is inspired by the related area of coverage guided fuzzing, exemplified by tools like AFL. Rather than just generating a fresh random input at each iteration, CGPT can also produce new inputs by mutating previous ones using type-aware, generic mutator operators. The target program is instrumented to track which control flow branches are executed during a run and inputs whose runs expand control-flow coverage are retained for future mutations. This means that, when sparse conditions in the target are satisfied and new coverage is observed, the input that triggered them will be retained and used as a springboard to go further. We have implemented CGPT as an extension to the QuickChick property testing tool for Coq programs; we call our implementation FuzzChick. We evaluate FuzzChick on two Coq developments for abstract machines that aim to enforce flavors of noninterference, which has a (very) sparse precondition. We systematically inject bugs in the machines' checking rules and use FuzzChick to look for counterexamples to the claim that they satisfy a standard noninterference property. We find that vanilla QuickChick almost always fails to find any bugs after a long period of time, as does an earlier proposal for combining property testing and fuzzing. In contrast, FuzzChick often finds them within seconds to minutes. Moreover, FuzzChick is almost fully automatic; although highly tuned, hand-written generators can find the bugs faster, they require substantial amounts of insight and manual effort.

Notes: This doesn't have much in the way of constraint programming, however the related work section appears promising and has some papers I haven't yet collated from a brief look.

the rest of the papers I list here I haven't had a proper read of yet, but will update this when I have (along with structuring more proper notes for the above)
[7] is interesting as it's a phd dissertation, however it is targeting uml models rather than code.
[8] [9] [10] [11] [12]

References

[1] Emanuele De Angelis et al. "Property-Based Test Case Generators for Free". In: *Tests and Proofs*. Ed. by Dirk Beyer and Chantal Keller. Vol. 11823. Cham: Springer International Publishing, 2019, pp. 186–206. ISBN: 978-3-030-31156-8 978-3-030-31157-5. DOI: 10.1007/978-3-030-31157-5_12. URL: http://link.springer.com/10.1007/978-3-030-31157-5_12.

[2] Matthieu Carlier, Catherine Dubois, and Arnaud Gotlieb. "FocalTest: A Constraint Programming Approach for Property-Based Testing". In: *Software and Data Technologies*. Ed. by José Cordeiro, Maria Virvou, and Boris Shishkov. Vol. 170. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 140–155. ISBN: 978-3-642-29577-5 978-3-642-29578-2. DOI: 10.1007/978-3-642-29578-2_9. URL: http://link.springer.com/10.1007/978-3-642-29578-2_9.

[3] Patrice Godefroid, Michael Y Levin, and David Molnar. "Automated Whitebox Fuzz Testing". In: *Network and Distributed System Security Symposium* (2008).

[4] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (2008), pp. 209–224.

[5] Dominic Steinhöfel and Andreas Zeller. "Input Invariants". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE '22: 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Singapore Singapore: ACM, Nov. 7, 2022. DOI: 10.1145/3540250.3549139. URL: https://dl.acm.org/doi/10.1145/3540250.3549139 (visited on 09/17/2024).

[6] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. "Coverage Guided, Property Based Testing". In: *Proceedings of the ACM on Programming Languages* 3 (OOPSLA Oct. 10, 2019), pp. 1–29. ISSN: 2475-1421. DOI: 10.1145/3360607. URL: https://dl.acm.org/doi/10.1145/3360607.

[7] Elisabeth Jobstl. “Model-Based Mutation Testing with Constraint and SMT Solvers”. In: ().

[8] Bruno Marre and Benjamin Blanc. “Test Selection Strategies for Lustre Descriptions in GATeL”. In: *Electronic Notes in Theoretical Computer Science* 111 (Jan. 2005), pp. 93–111. ISSN: 15710661. DOI: 10.1016/j.entcs.2004.12.010. URL: https://linkinghub.elsevier.com/retrieve/pii/S157106610405234X.

[9] Xavier Gillard, Pierre Schaus, and Yves Deville. “SolverCheck: Declarative Testing of Constraints”. In: *Principles and Practice of Constraint Programming*. Ed. by Thomas Schiex and Simon De Givry. Vol. 11802. Cham: Springer International Publishing, 2019, pp. 565–582. ISBN: 978-3-030-30047-0 978-3-030-30048-7. DOI: 10.1007/978-3-030-30048-7_33. URL: http://link.springer.com/10.1007/978-3-030-30048-7_33.

[10] Arnaud Gotlieb. “Euclide: A Constraint-Based Testing Framework for Critical C Programs”. In: *2009 International Conference on Software Testing Verification and Validation*. 2009 International Conference on Software Testing Verification and Validation. Apr. 2009, pp. 151–160. DOI: 10.1109/ICST.2009.10. URL: https://ieeexplore.ieee.org/document/4815347/?arnumber=4815347.

[11] Arnaud Gotlieb. “Automatic Test Data Generation Using Constraint Solving Techniques”. In: *ACM SIGSOFT Software Engineering Notes* 23.2 (1998), pp. 53–62. DOI: 10.1145/271775.271790.

[12] Cormac Flanagan. “Automatic Software Model Checking via Constraint Logic”. In: *Science of Computer Programming* 50.1-3 (Mar. 2004), pp. 253–270. ISSN: 01676423. DOI: 10.1016/j.scico.2004.01.006. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167642304000073.