Project Summary
FMitF: Constructing Reliable Networked Systems via an Assembly of Formal Methods

Why composition of methods matters.

Proposed Research What we'll do about it.

Intellectual Merit Why it's deep science and novel.

Broader Impacts Improvements in verification and testing will obviously make sofware less inherently dangerous. As someone said, "software is eating the world... in the same sense that moths are eating your sweaters."

Key Words: some good key words

Data Management Plan
SHF:Small:Collaborative Research:Constructing Reliable Networked Systems via an Assembly of Formal Methods

The data in the proposed project is primarily of two kinds:

- source code for testing tools
- empirical data about testing experiments.

In the first case, TSTL and C-Reduce are already open source systems hosted on GitHub.

Curricular materials associated with the testing tools will also be stored in an open source repository, since in this project the primary educational benefits are linked to the use of testing tools. Using GitHub automatically provides us with excellent backup and archiving for the code and curricular material products of the project.

Empirical testing data will often be ephemeral and reproducible by re-running experiments with known seeds. If the computational burden to recreate data is excessively large we will store data in a compressed format on the open-source repository. Given the ever-increasing power of computers, we do not expect this to be the case for most data sets.

We have no unusual format or metadata requirements; test cases themselves are (in our setting) ususally source code files or text files readable by the tools, and testing results are standard formats produced by instrumentation tools or stored in a format such as CSV or XML files.

We do not anticipate the need to work with sensitive or confidential information; no extraordinary practices are required in order to conduct this research.

Project Description — FMitF: Constructing Reliable Networked Systems via an Assembly of Formal Methods

1 Introduction

Developing reliable complex systems under difficult engineering constraints, e.g., systems whose final endpoints are power-constrained, limited-bandwidth, and costly for humans to repair or even inspect, is a key challenge over the next decade; while the widely discussed "Internet of Things" may be the first thing that comes to mind given this description, the problem is also central to long-term scientific research efforts, ranging from the climate change and plant migration research platform used as a testbed in this proposal to efforts to understand the surfaces of other planets.

Formal methods have been used to verify aspects of such systems, including file systems, control elements, network protocols, Let's find some cites and add things here; however, the proofs of components are seldom, if ever, assembled into a coherent, quantified, or even fully *qualified* understanding of the extent to which the whole system has been verified. Because different components of many such systems are developed by disparate groups, sometimes collaborating internationally, even when formal methods are uniformly used, the emphasis is on the plural: formal methods, not a single formal method. The assumptions of one formal method/component are seldom formally yoked to the outputs of another formal method, with gaps in verification made clear. Finally, many components are only verified using high-confidence, automated, but non-proof-based approaches, including model checking with unsound abstractions, or automated test generation. When these kinds of "semi-formal" methods (relying on a formal model of inputs and system behavior, but not providing correctness proofs) are integrated into a design, there is almost never a characterization of the confidence thus provided. The outcome is either not used at all in other methods, or is assumed to be as solid as a full proof.

In this proposal, we aim to address this weakness by designing "glue" formal methods for assembling the results of a set of hetergenous formal and semi-formal approaches into a unified, coherent (if complex) result, able to identify both weaknesses and strengths in a system verification. E.g., if the proof of correctness of one component relies strongly on an assumption backed only by limited, manually constructed tests, this can be distinguished from the case where the assumption is backed by a proof of correctness in a different formalism, which can be distinguished from the case where it is backed by both model-based and symbolic-execution driven test generation, and a strong probabilistic, coverage-based, or mutation-based estimate of completeness is available. Moreover, the proposed approach will be able to distinguish the case where an entire verification result is tainted by a fundamental and non-proven assumption from the (common) case where different properties or behavioral paths have different degrees of reliabilty, ranging from complete independence from an uncertain assumption to confidence completely proportional to the strength of an unchecked assumption.

1.1 Problem Statement and PI Qualifications

The simplest form of problem statement in this case may be in the form of an example. Consider a networked system consisting of in-the-field power-constrained nodes with sensors and actuators, in-the-field base stations that collect data from and issue commands to these nodes, and a (possibly distributed) remote command-and-control center that handles permanent high-level data storage and analysis, coordination of activities across multiple field stations, and human oversight. The overall system might be (as in our primary case study and testbed platform) an ecological science platform performing experiments on plant migration and climate change impacts, or a NASA mission involving a swarm of robot explorers on another planet. The details of system purpose are irrelevant; what matters is that there is a complex networked system with heterogenous connectivity and computational capabilities, where failure is extremely costly, in terms of either scientific outcomes (a stuck irrigation ruins a five-year experiment on plant survival) or simply dollar value (a multi-million dollar space mission is lost).

When failure is sufficiently costly, verification efforts are (now and, we hope, increasingly in the future) proportionally intense. However, complex systems such as these are seldom developed by one team, in one place, with a coordinated approach to ensuring software correctness. Instead, a variety of methods may be applied, even by teams all seeking the same goal of high reliability using state-of-the-art methods.

For example, two software modules may run on the same embedded system without operating-system-level memory protection. One module may have been "verified" using high-coverage, high-quality, automatically generated tests. The other module may have been verified using bounded model checking, with a depth that can be shown to exceed the diameter of system executions under valid inputs. The tested module's testing results are valid under the assumption that the other module does not change values it accesses, and *this* assumption is guaranteed to hold, because the model-checked system is shown to never access memory in unsafe ways. The model-checked component, on the other hand, is correct under the symmetric assumption: that the tested module does not change its' memory contents; this assumption is only partially guaranteed, for a limited set of executions. This is a simple case, but even here the relationship can be used to guide engineering. If we insert runtime guards on memory accesses, these only need be applied to the tested module, not the model-checked one, to guarantee memory boundary protection for the combined system. However, if the model-checked system essentially relies on validity of outputs from the tested system for correctness, and these cannot be checked at runtime, the model-checking result itself is inherently only known to be as realiable as the tested system. The measure of the strength of testing, whether it is code coverage, expected mean-time-to-failure under a realistic operational profile of inputs, or mutation score, propagates to the "verified" module as well.

A basic aspect of the proposed work is understanding the implications of four kinds of assumption relationships:

1. First, there is the case where a formal method producing complete proofs has an assumption that is discharged by another formal method providing proofs. In this case, the core effort is providing translations between logics that allow us to understand when these assumptions are fully discharged, and to propagate any assumptions of the formal method used to guarantee the assumption to the formal method making the assumption.

2. Second, there is the case where a formal method producing complete proofs has an assumption discharged by testing or another incomplete method.

3. Third, there is the case where testing relies on an assumption from a formal method.

4. Fourth, there is the case where testing relies on an assumption validated by another testing approach.

Everything past this point is from a previous proposal, ignore it. My prior NSF may be partially valid.

1.2 Expected Outcomes

2 Related Work

Some of the related work is the general literature on software testing problems [5, 55] that is relevant, which is cited throughout this proposal, e.g. recent work on regression testing [7, 16, 21, 60, 68], seeded or parameterized test generation [39, 40, 50, 57, 63, 64, 69], flaky tests [19, 42, 48, 51, 52, 56] and compiler testing [1, 18, 32, 44, 67, 74]. This section focuses on more specific precursors to our ideas in this proposal, including our own efforts towards the idea of novel test manipulations.

The idea of algorithms that operate on tests, as such, is primarily represented in the literature by the work on delta-debugging or test reduction in general: [9, 33, 46, 47, 53, 59, 61, 70–72]. Sai proposed a very limited, ad hoc version of semantic minimization that aims to go beyond the simplifications possible with conventional delta-debugging [75]. Work on automatically producing readable tests [12, 13] is also related, in that it aims to "simplify" tests. Test case purification [65] is a kind of limited (in approach and in goal)

Year	Both Institutions	NAU	Utah
1	test representations; operation definitions	Python composition; heterogenous composition	"little fuzzers" for C and C++; start C/C++ composition work
2	composition heuristics; oracle composition	grammar composition; Python decomposition	continue C/C++ composition; start C/C++ decomposition work
3	behavior+oracle comp.; regression toolchain; decomposition for seeding	grammar decomposition; Java decomposition	continue C/C++ decomposition; study critical compositions

Table 1: Some core elements of proposed work plan

decomposition, as are some efforts to produce unit from system tests [15,41,54,62]. To our knowledge, our own proposal on test composition [26] is the only significant effort towards automated test composition.

In our own preliminary work, we went beyond these ideas to explicitly define novel *operations* for *normalization* and *generalization* [27]. Normalization is an operation that, like delta-debugging, aims to preserve some predicate describing a test's purpose (and validity). However, rather than simply carving out subsets from a test, normalization uses term rewriting to balance the goal of (1) preserving some continuity of behavior, reducing the risk of slippage [34] with (2) the goals of transforming multiple, redundant failures into a single failing test, and achieving significant additional test size reduction beyond delta-debugging. The approach has been shown to reduce the number of redundant failing tests by more than an order of magnitude, and often provide additional size reduction by a factor of two or more. Generalization [27] uses a similar rewriting approach to produce a family of related tests with the same behavior, which makes understanding the accidental and essential elements of a failure much easier. Pike proposed a limited test generalization that applies to tests that consist of Haskell data values [58].

The idea of representing the generality of tests with the goal of enabling complex operations has been relatively unexplored, to our knowledge [25]. Andrews et al. presented a pool-based concept as part of an effort to analyze bounded exhaustive testing vs. random testing [6], but did not base any idea of test operations on this approach. The UDITA language [20] considers related ideas, but focuses only on generation approaches, not operations on existing tests. The TSTL language [28,35] is a concrete embodiment of tests as selections between finite transition operations, informed by the approach to model-checking C code used in the SPIN model-checker [36–38].

3 Plan of Work and Evaluation

We propose to structure the work into three lines of effort (Table 1). First, there is core work on tests as first-class entities: developing representations, core algorithm development, and so forth. This will be performed jointly, with a basic three year plan consisting of first focusing on representations and code/data test integration, then focusing on operation definitions and methods that apply to multiple operations (e.g., using normalization or validity preservation as a tool in composition), and finally focusing on integrating operations into workflows such as regression testing or test generation that are more complex than single operations, and require extensive effort to evaluate (though we will also undertake preliminary work on these applications to keep our methods focused on practical ends).

Concurrently, NAU will focus on TSTL-based aspects of the work, both integration and evaluation of operations and using data generation in a primarily code-based testing setting. Similarly, at Utah, an as-yet-unnamed platform for composing compiler test cases will be developed.

Year 1:

- Implement basic composition approach, evaluate increase in test throughput, evaluate new coverage due to feature interactions in the composed test cases.

- Start investigating the idea that highly sophisticated tests for very difficult input spaces (such as C++ programs) can be generated by starting with high-coverage test cases generated by relatively simple "little fuzzers," and composing them.
- Initial framework for composition of tests for *heterogeneous* systems.

Year 2:

- Define basic automatic decomposition approach: not just test reduction, but carving a test into sub-tests that for behaviors naturally covered together, with minimal or no loss of interaction behavior.
- Produce a composition approach that takes test independence information and other heuristics (e.g. simple bridge code and automatic variable renamings) into account.
- Work on a generic grammar-based (and raw-binary) composition tool.
- Start on techniques for extracting oracles from human unit tests and composing them with automatically generated tests, and taking property-based specification [10,49] in automatically-generated tests and composing properties with human tests; attempt to automatically (heuristically, with false positives) reject invalid tests.

Year 3:

- Investigate more aggressive search-based composition approaches to try to find critical compositions of features requiring generation of new test elements.
- Produce tools that can combine both behavior and oracles from human and automatically generated tests, evaluate by improved test effectiveness and validity of tests systems, including hardware/software systems, test on NASA systems (PI Groce has contacts on both MSL and upcoming small CubeSAT architecture aiming at testability).
- Produce composition/decomposition aware regression tools, able to produce custom compositions and decompositions of tests to enable high-speed effective regression testing based on changes: combine traditional dependency analysis with on-the-fly adjustment of test granularities.
- Investigate ability of decomposed tests to improve performance of seeded test generation.

Evaluation: While our ability to compose and decompose test cases can be evaluated (and we will do so, using reasonable metrics), our main goal is to further the community's understanding of the character of defects in real systems, and of the test cases that reveal them. Thus, our most important metrics will be coverage of, and ability to discover interesting defects in, real programs such as file systems, embedded and cyber-physical systems, and compilers. We have working relationships with major projects in all of these fields (e.g., PI Regehr is a member of the LLVM Foundation's Board of Directors and PI Groce has already begun discussion of applications of test composition to the Southwest Experimental Garden Array (SEGA) [2,11]). Specific evaluation metrics that we will use include:

- Can automatically decomposed tests be as good as unit tests written by humans? We do not propose to evaluate this via user studies, but rather to determine if the maintainers of important open source projects will allow these test cases into their curated unit test suites.
- Are automatically composed test cases effective in increasing coverage beyond the summed coverage of the test cases before composition? Can they hit bugs that are the result of critical compositions of features that are otherwise very difficult to hit?
- Can automatically composed test cases provide significant increases in testing throughput?

- Can automatically decomposed seed tests for AFL, KLEE, EvoSuite, and TSTL provide significant increases in testing effectiveness (faults detected and code covered) over using fewer, larger and more complex, seed tests?
- Can composition and decomposition be used to improve regression operations, as measured by traditional measures such as average percent faults detected (APFD) [17]? Can it reduce test flakiness?

4 Broader Impacts and Education

The proposed research yields several opportunities for enhancing CS education, recruiting new CS majors, and retaining CS students, particularly members of underrepresented groups. Our education activities directly relate to the following broader impact goals: (1) Development of a globally competitive STEM workforce; (2) Increased participation of women, persons with disabilities, and underrepresented minorities in STEM; (3) Improved pre-K-12 STEM education; and (4) Improved undergraduate STEM education.

4.1 Teaching Testing to Non-Computer-Scientists

PI Regehr views testing as very under-emphasized in the typical computer science curriculum and he makes testing an explicit area of focus in every course that he teaches, whether it is compilers, operating systems, or embedded software. In the winter and spring of 2017 he organized an after-school "coding club" for the 5th and 6th grade classes at Whittier Elementary, the public school in Salt Lake City that his son attended, which is located in a somewhat disadvantaged neighborhood (median annual household income $\$35,446 \pm \$11,944$, according to 2014 census data).

He taught the students Python, starting with turtle graphics and using that as the basis for teaching fundamentals like variables, loops, functional abstraction, and recursion. As the students grappled with these concepts, he also emphasized methods for creating "glitch free" code (kids this age seem to have an intuitive grasp of software bugs, especially in games, and call them "glitches"), such as trying to exercise corner cases in the code and trying random inputs.

In spring 2018 PI Regehr will be teaching a software engineering course, focused mainly on testing, as part of a new professional MS program at the University of Utah. The students will not have CS degrees, but rather come from diverse backgrounds. His course will emphasize the fundamental issues, that it hardly makes sense to create software unless we have a precise understanding of what it is supposed to do, and that once we have that understanding, it can be operationalized using tests.

4.2 Excursions in Testing

PI Groce will be working with the NAU Student ACM Chapter to present a series of "excursions in testing" that use automated testing to introduce popular or exciting Python libraries to undergraduates. In particular, some of these excursions will focus on media-related libraries or bioinformatics libraries. The work of Guzdial [31] has shown that media computation is a potentially effective way to both recruit and retain female and under-represented minority students in computer science. More than 55% of biology majors are female; bioinformatics is a bridge between STEM majors that are not lacking in female students and those majors (such as CS) that continue to lag in recruitment and retention of women. Testing excursions introduce a Python topic and allow students to participate in possibly finding and reporting bugs in real software, which forms a strong connection to real, ongoing open-source development efforts, in same cases even before students have strong programming mastery.

4.3 Improving Software System Reliability

A key element of broader outreach will be to report bugs discovered during our testing experiments, and contribute test suites to open source projects. To that end, we will primarily target real world systems with

our experiments, in hopes of improving their quality, reliability, and test suites. While we expect to develop more examples, our current infrastructure includes automated testing for Google and Mozilla JavaScript engines, a variety of C compilers (including GCC and LLVM), the YAFFS2 [66] embedded flash file system, Google's Go compiler, a large set of Unix utilities, various Python libraries (including the most widely used library, and key scientific and numeric analysis packages), and a set of Android applications. In previous work, discussions with working test engineers at Mozilla, Google, and NASA have significantly informed our research progress, and we expect this to continue. In addition to simply reporting bugs, we aim to provide enhancements to tools (C-Reduce and TSTL) that engineers can use, and in the case of TSTL, easily modified packages for testing Python libraries that can be tuned to developer needs.

5 Results From Prior NSF Support

The most relevant prior NSF support for PIs Groce and Regehr is CCF-1217824, "Diversity and Feedback in Random Testing for Systems Software," with a total budget of $491,280 from 9/2012 until 9/2015, a collaborative proposal between the PIs.

Intellectual Merit The results of CCF-1217824 include a preliminary exploration of how to "tame" fuzzer output, a problem also considered in this proposal [8]. In previous work, the goal was to find an algorithm for using hand-chosen distance metrics to identify bugs in tests; in this proposal, other methods for taming fuzzers are addressed. A key related result from CCF-1217824 is the development of a strategy for creating very "quick tests" from time-consuming randomized test suite by minimizing each test with respect to its code coverage [23], which won the Best Paper award at the 2014 International Conference on Software Testing. This work showed that tests reduced with respect to code coverage can serve as effective regression tests or seeds for symbolic execution [24, 73]. Moreover, we showed that the benefits of such reduction do not depend on 100% preservation of a property, when that property is quantitative (e.g. coverage) rather than qualitative [4]. Cause reduction, with and without complete preservation of properties, is a core component of our approach to test operations, and a primary demonstration of our conceptual framework, where two tests that satisfy the same key properties are assumed to be interchangeable.

CCF-1217824 also contributed to the design and development of the TSTL tool [28, 35], which supports fully automatic swarm testing, based on the actions defined in a test harness, and which is a core element of work for this proposal.

Other results from this project include an overview of the value of coverage in testing experiments [22] and exploration of how individual test features impact the coverage and fault detection statistics of random tests [30]. The basic swarm testing approach has been extended to allow production of focused random tests targeting particular code [3]. At the broader level, CCF-1217824 has produced a general set of results that focus on making automated random testing usable by practitioners, and using symbolic execution on larger, realistic software, in particular understanding how code coverage and test content interact, either through minimization or through statistical analysis. Publications resulting from this grant are numerous, including ones [3, 4, 8, 22–24, 29, 30, 35, 73] cited above with relevant results.

Broader Impact The results of CCF-1217824 have been used in teaching software engineering to undergraduates at Oregon State University. At the University of Utah PI Regehr developed a new course "Writing Solid Code" during this time period, that focused almost wholly on software testing, based in part on research driven by this grant. Work done by both PIs has contributed to the discovery of previously unknown faults in multiple open source and commercial software systems. The further development of the swarm testing techniques the proposal centers on have in particular furthered the effort to improve the quality of compilers, including LLVM and GCC, and to test language tools in general [14, 43–45]. Source code for tools resulting is available online at GitHub in the `swarmed_tools` repository, and data (too large for hosting services) is available upon request; TSTL is available on GitHub at `https://github.com/agroce/tstl`, and key data from fuzzer taming is available at `https://github.com/agroce/mutants16/tree/master/tests`.

References

[1] kcc. `https://github.com/kframework/c-semantics`.

[2] Southwest Experimental Garden Array: An instrumented field site array for climate change research.

[3] Mohammad Amin Alipour, Alex Groce, Rahul Gopinath, and Arpit Christi. Generating focused random tests using directed swarm testing. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 70–81, New York, NY, USA, 2016. ACM.

[4] Mohammad Amin Alipour, August Shi, Rahul Gopinath, Darko Marinov, and Alex Groce. Evaluating non-adequate test-case reduction. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 16–26, 2016.

[5] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[6] Jamie Andrews, Yihao Ross Zhang, and Alex Groce. Comparing automated unit testing strategies. Technical Report 736, Department of Computer Science, University of Western Ontario, December 2010.

[7] John Bible, Gregg Rothermel, and David S. Rosenblum. A comparative study of coarse- and fine-grained safe regression test-selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):149–183, 2001.

[8] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *Programming Language Design and Implementation*, pages 197–208, 2013.

[9] J. Choi and A. Zeller. Isolating failure-inducing thread schedules. In *International Symposium on Software Testing and Analysis*, pages 210–220, 2002.

[10] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279, 2000.

[11] J. S. Clark, P. K. Agrawal, D. M. Bell, P. G. Flikkema, A. Gelfand, X. Nguyen, E. Ward, and J. Yang. Inferential ecosystem models, from network data to prediction. *Ecological Applications*, 21(5):1523–1536, 2011.

[12] Ermira Daka, José Campos, Jonathan Dorn, Gordon Fraser, and Westley Weimer. Generating readable unit tests for Guava. In *Search-Based Software Engineering - 7th International Symposium, SSBSE 2015, Bergamo, Italy, September 5-7, 2015, Proceedings*, pages 235–241, 2015.

[13] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. Modeling readability to improve unit tests. In *Foundations of Software Engineering, ESEC/FSE*, pages 107–118, 2015.

[14] Kyle Dewey, Jared Roesch, and Ben Hardekopf. Fuzzing the rust typechecker using clp (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 482–493. IEEE, 2015.

[15] Sebastian Elbaum, Hui Nee Chin, Matthew B. Dwyer, and Jonathan Dokulil. Carving differential unit test cases from system test cases. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 253–264, New York, NY, USA, 2006. ACM.

[16] Sebastian Elbaum, Praveen Kallakuri, Alexey Malishevsky, Gregg Rothermel, and Satya Kanduri. Understanding the effects of changes on the cost-effectiveness of regression testing techniques. *Software Testing, Verification and Reliability*, 13(2):65–83, 2003.

[17] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. *SIGSOFT Softw. Eng. Notes*, 25(5):102–112, August 2000.

[18] Chucky Ellison and Grigore Rosu. An executable formal semantics of c with applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 533–544, New York, NY, USA, 2012. ACM.

[19] Zebao Gao, Yalan Liang, Myra B. Cohen, Atif M. Memon, and Zhen Wang. Making system user interactive tests repeatable: When and what should we control? In *International Conference on Software Engineering*, ICSE '15, pages 55–65. IEEE, 2015.

[20] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test generation through programming in UDITA. In *International Conference on Software Engineering*, pages 225–234, 2010.

[21] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):184–208, April 2001.

[22] Alex Groce, Mohammad Amin Alipour, and Rahul Gopinath. Coverage and its discontents. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 255–268, New York, NY, USA, 2014. ACM.

[23] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. Cause reduction for quick testing. In *IEEE International Conference on Software Testing, Verification and Validation*, pages 243–252. IEEE, 2014.

[24] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. Cause reduction: Delta-debugging, even without bugs. *Journal of Software Testing, Verification, and Reliability*, 26(1):40–68, 2016.

[25] Alex Groce and Martin Erwig. Finding common ground: choose, assert, and assume. In *Workshop on Dynamic Analysis*, pages 12–17, 2012.

[26] Alex Groce, Paul Flikkema, and Josie Holmes. Towards automated composition of heterogeneous tests for cyber-physical systems. In *Workshop on Testing Embedded and Cyber-Physical Systems*, pages 12–15, 2017.

[27] Alex Groce, Josie Holmes, and Kevin Kellar. One test to rule them all. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 1–11, New York, NY, USA, 2017. ACM.

[28] Alex Groce and Jervis Pinto. A little language for testing. In *NASA Formal Methods Symposium*, pages 204–218, 2015.

[29] Alex Groce, Jervis Pinto, Pooria Azimi, and Pranjal Mittal. TSTL: a language and tool for testing (demo). In *ACM International Symposium on Software Testing and Analysis*, pages 414–417, 2015.

[30] Alex Groce, Chaoqiang Zhang, Mohammed Amin Alipour, Eric Eide, Yang Chen, and John Regehr. Help, help, I'm being suppressed! the significance of suppressors in software testing. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 390–399. IEEE, 2013.

[31] Mark Guzdial. A media computation course for non-majors. In *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '03, pages 104–108, New York, NY, USA, 2003. ACM.

[32] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the undefinedness of C. *SIGPLAN Not.*, 50(6):336–345, June 2015.

[33] Ralf Hildebrandt and Andreas Zeller. Simplifying failure-inducing input. In *International Symposium on Software Testing and Analysis*, pages 135–145, 2000.

[34] Josie Holmes, Alex Groce, and Mohammad Amin Alipour. Mitigating (and exploiting) test reduction slippage. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*, A-TEST 2016, pages 66–69, New York, NY, USA, 2016. ACM.

[35] Josie Holmes, Alex Groce, Jervis Pinto, Pranjal Mittal, Pooria Azimi, Kevin Kellar, and James O'Brien. TSTL: the template scripting testing language. *International Journal on Software Tools for Technology Transfer*, 2017. Accepted for publication.

[36] Gerard Holzmann and Rajeev Joshi. Model-driven software verification. In *SPIN Workshop on Model Checking of Software*, pages 76–91, 2004.

[37] Gerard Holzmann, Rajeev Joshi, and Alex Groce. New challenges in model checking. In *Symposium on 25 Years of Model Checking*, pages 65–76, 2008.

[38] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[39] Wei Jin and Alessandro Orso. Bugredux: Reproducing field failures for in-house debugging. In *International Conference on Software Engineering*, pages 474–484, 2012.

[40] Wei Jin and Alessandro Orso. BugRedux: reproducing field failures for in-house debugging. In *Int. Conf. on Software Engineering*, pages 474–484, 2012.

[41] Matthew Jorde, Sebastian G. Elbaum, and Matthew B. Dwyer. Increasing test granularity by aggregating unit tests. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, pages 9–18, 2008.

[42] Wing Lam, Sai Zhang, and Michael D. Ernst. When tests collide: Evaluating and coping with the impact of test dependence. Technical Report UW-CSE-15-03-01, University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, March 2015.

[43] Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C Pierce, and Li-yao Xia. Beginner's luck: A language for property-based generators. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, 2017.

[44] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 216–226, 2014.

[45] 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.

[46] Yong Lei and James H. Andrews. Minimization of randomized unit test cases. In *International Symposium on Software Reliability Engineering*, pages 267–276, 2005.

[47] Andreas Leitner, Manuel Oriol, Andreas Zeller, Ilinca Ciupa, and Bertrand Meyer. Efficient unit test case minimization. In *International Conference on Automated Software Engineering*, pages 417–420, 2007.

[48] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 643–653. ACM, 2014.

[49] David R. MacIver. Hypothesis: Test faster, fix more. `http://hypothesis.works/`.

[50] Paul Dan Marinescu and Cristian Cadar. make test-zesti: a symbolic execution solution for improving regression testing. In *International Conference on Software Engineering*, pages 716–726, 2012.

[51] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. Taming Google-scale continuous testing. In *International Conference on Software Engineering*, pages 233–242. IEEE, 2017.

[52] John Micco. Flaky tests at Google and how we mitigate them. `https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html`, May 2016.

[53] Ghassan Misherghi and Zhendong Su. HDD: hierarchical delta debugging. In *International Conference on Software engineering*, pages 142–151, 2006.

[54] Alessandro Orso and Bryan Kennedy. Selective capture and replay of program executions. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[55] Alessandro Orso and Gregg Rothermel. Software testing: A research travelogue (2000–2014). In *Proceedings of the on Future of Software Engineering*, FOSE 2014, pages 117–132, 2014.

[56] Fabio Palomba and Andy Zaidman. Does refactoring of test smells induce fixing flaky tests? In *IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2017.

[57] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 504–515, 2011.

[58] Lee Pike. SmartCheck: automatic and efficient counterexample reduction and generalization. In *ACM SIGPLAN Symposium on Haskell*, pages 53–64, 2014.

[59] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *Programming Language Design and Implementation*, pages 335–346, 2012.

[60] Gregg Rothermel, Roland Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization. *Trans. Softw. Eng.*, 27:929–948, 2001.

[61] Jesse Ruderman. Bug 329066 - Lithium, a testcase reduction tool (delta debugger). `https://bugzilla.mozilla.org/show_bug.cgi?id=329066`, 2006.

[62] David Saff, Shay Artzi, Jeff H. Perkins, and Michael D. Ernst. Automatic test factoring for java. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 114–123, New York, NY, USA, 2005. ACM.

[63] Suresh Thummalapenta, Madhuri R. Marri, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. Retrofitting unit tests for parameterized unit testing. In *Fundamental Approaches to Software Engineering - 14th International Conference, FASE 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, pages 294–309, 2011.

[64] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 253–262. ACM, 2005.

[65] Jifeng Xuan and Martin Monperrus. Test case purification for improving fault localization. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 52–63, 2014.

[66] YAFFS: A flash file system for embedded use. `http://www.yaffs.net/`.

[67] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 283–294, 2011.

[68] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, March 2012.

[69] Michal Zalewski. american fuzzy lop (2.35b). `http://lcamtuf.coredump.cx/afl/`, November 2014.

[70] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC / SIGSOFT Foundations of Software Engineering*, pages 253–267, 1999.

[71] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2005.

[72] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on*, 28(2):183–200, 2002.

[73] Chaoqiang Zhang, Alex Groce, and Mohammad Amin Alipour. Using test case reduction and prioritization to improve symbolic execution. In *International Symposium on Software Testing and Analysis*, pages 160–170, 2014.

[74] Qirun Zhang, Chengnian Sun, and Zhendong Su. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 347–361, 2017.

[75] Sai Zhang. Practical semantic test simplification. In *International Conference on Software Engineering*, pages 1173–1176, 2013.