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

Most large or critical software projects devote a significant fraction of their effort to testing; tests are the primary products of this effort, and are ubiquitous in real-world development. The research community has developed numerous techniques to automatically generate tests. Unfortunately, once tests are created, whether by manual effort or automated techniques, they are among the most inert of computational objects. At present, tests are stored, executed, and occasionally read by humans, but they are essentially treated as givens, amenable neither to modification (other than by hand), combination, separation, or amplification. For example, while there is a large body of research studying how to minimize, prioritize, and select tests from a test suite, it is only in the last three years that work has finally appeared that modifies individual tests within the suite in the pursuit of these goals. Tests are, at present, not first-class entities: they support only a few, limited, operations, lacking even such basic capabilities as composition and decomposition.

**Proposed Research** This project will investigate a practical theory of tests as entities that support operations such as composition, decomposition, normalization, and generalization. The two basic prongs of the research are: (1) how to represent tests so as to enable rich operations and (2) the development of algorithms for useful operations on tests. This research will be grounded in a set of realistic problems that require a better understanding of tests as first-class entities.

For example, a primary limitation of human-produced tests is that humans are not prolific test authors, nor are they typically imaginative enough to exercise the more obscure behaviors of a software system. However, automatically generated tests typically cannot check for complex properties of system output, because human understanding is needed to evaluate correctness in the absence of a complete formal specification. A sophisticated ability to compose tests should enable the "amplification" (in terms of coverage and complexity) of hand-crafted tests with automatically generated additional behavior that preserves the effectiveness of the human-produced checks on correct behavior. Effective test composition would also make it possible to automatically produce integration/system tests for systems, even heterogeneous ones, where the individual elements have effective tests but the interactions between the sub-systems are untested.

The ability to automatically decompose tests into smaller tests, each checking correctness of a smaller aspect of system behavior, such that the set of all such tests retains the power of the original test has many potential benefits: it can improve seeded test generation; reduce the high cost of regression testing by enabling a much finer-grained approach to selection, minimization, and prioritization of tests; reduce the chance of "flaky" test behavior; and support recently-proposed test-based approaches to self-adaptive software. Such tests are also likely to be easier to understand and use in debugging.

**Intellectual Merit** While tests are common objects of academic study, the very concept of a test is poorly defined. The commonalities and differences between types of tests (unit tests or API-call sequences vs. file inputs, for example) are poorly understood. The approach of this project is rooted in a novel conception of tests: tests can be understood as functions without inputs that return a test status indicating failure(s). Additionally, tests typically feature a linear (but possibly hierarchical) notion of components and a set of causal properties that define the reason for a given test's existence, and control its output given the state of the program under test. The idea of tests as programs of a restricted form should also enable the adaption of techniques from test analysis to more general program analysis, and vice versa.

**Broader Impacts** Improvements in testing for software systems will have a direct impact on the quality of systems critical to the economy, national defense, scientific research in other fields, and the quality of life for information consumers. Inadequate testing methods cost tens of billions of dollars each year, despite test budgets that often consume more than half of development costs. More effective test generation and debugging is critical to addressing this problem.

**Key Words:** software testing, regression testing, test generation, debugging

# 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 networked 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 described 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 and 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, some components are only verified using high-confidence, automated, but non-proof-based approaches, including model checking with unsound abstractions, or (model-driven) 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 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 approaches into a single (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 probabilistic or coverage-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 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 the 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 (also 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) 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 simple 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 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. The tested module's tests are valid under the assumption that the other module does not change values it accesses, and *this* assumption is guaranteed to hold. The model-checked system, on the other hand, is correct under the same assumption: that the other module does not change its' memory contents; this assumption is only probabilistic.

### 1.2 Expected Outcomes

## 2 Contributions to Formal Methods and the Field

## 3 Research Focus 1: Test Composition for Heterogeneous Systems

Modern critical software systems are often composed of a complex interacting set of layers. For example, cyber-physical systems (CPS) design employs layering in multiple domains, e.g., computation, networking, and the modeling of the embedding physical system and the system's sensors and actuators. However, current layering approaches do not capture non-functional system properties essential to CPS, e.g., timing and energy use, that emerge via testing. To manage design process complexity, iterative development is commonplace: while the long-term trend is refinement of abstract models, engineers often need to shift back and forth between implementation-level models and more abstract models to gather new data, gain knowledge and insight, and optimize system performance.

The same multiplicity of heterogeneous layers also often applies to existing tests for complex systems: often components of a system, such as a file system or actuators, are tested by one set of engineers, and using completely different methods than are used to produce tests at the high level of either control software with humans-in-the-loop or autonomous control systems. The core functionality of a system is usually written in low-level, embedded systems languages, such as C. In the ideal case, such systems are developed using both formal specification and verification and sophisticated automated testing. In some cases the formal specification is used to generate executable tests to ensure the real system matches the formal models; in other cases there is at least a very determined test generation effort, including efforts to produce very high-coverage tests. In contrast, user-centric or high-level autonomy layers are often developed in higher-level languages, such as Java, and with a much more informal approach to testing and verification. Increasingly, mission- and safety-critical low-level elements interact with users or high-level autonomy. Even when the behavior of each system, in isolation, is valid, the composition may compromise performance, user experience, security and privacy, or (in the worst case) safety.

Existing tests for different layers are highly valuable; however, they typically fail to cover the interaction of components effectively. This problem is made worse by a fundamental limitation: tests do not compose. Even within a single system, executing test `A` followed by test `B` seldom produces the desired union of behaviors (e.g., even covering all code covered by `A` or `B`). The actions of `A` often interfere with those of `B` (or vice versa): that is, some action in `A` is either illegal in a composed context, causing `B` (and thus the entire test) to become invalid, or disables some behavior of `B`, lowering test effectiveness. The ordering of test operations also matters: e.g., some actions in `A` must be before some actions in `B` to produce interaction, while other actions must be after some `B` action. For compositions of safety-critical and user-centric systems, and heterogeneous systems in general, it would be highly desirable to be able to automatically produce tests that are valid, have as little interference as possible, and maximize the sum of behaviors from the composed tests. In order to achieve this goal, we propose a novel operation of *test composition*, using the existing operation of test minimization [31,37,90] to remove portions of a constructed *hypothesis composition* to produce a test that has more behavior than a naïve composition [34].

### 3.1 Motivating Example: Mars Rover File System Testing

As an example of the problem, consider the following situation, a simplified generalization of testing efforts performed by PI Groce and others at NASA's Jet Propulsion Laboratory, during the development of the Curiosity Mars Rover (Mars Science Laboratory project) [35, 38, 39]. The file system for the rover can be considered from two points of view. There is the low-level, embedded flash file system, implemented as (essentially) a library in C. There is also a higher-level file catalog process, through which other components of the Curiosity software interact with the file system, and which is directly accessed by ground operations teams. The low-level file system was extensively tested using both model checking and random testing, by a team of formal verification and software engineering researchers. The high-level catalog process was also tested extensively — but only manually, by systems engineers and the Curiosity QA team, using less formal and intensive approaches. Every catalog test also tests the underlying file system, of course, but file system tests do not test the catalog. In practice, the two sets of tests exist completely separately: the catalog tests as Python scripts to issue commands, and the file-system tests as C programs. In operation, some faults related to interaction of the catalog and the file system were discovered. **We hypothesize that being able to compose high-level catalog tests and low-level file system tests might have detected some of these faults.**

Naïve composition of tests for the file system and the catalog will not work. Low-level file system tests may include operations that change the file system state in a way that the catalog, which has sole control of the contents of the file system in most directories during normal rover operation, cannot handle. Consider the composition of test `A`, for the file system, and test `B` for the catalog. Neither `A+B` (composition with `A` followed by `B`) nor `B+A` will provide the desired testing functionality. If we execute `A` then `B`, there may be little interaction between systems, and `A` may produce an initial state that the catalog cannot handle. If we execute `B` then `A`, the much more extensive testing of the underlying flash file system provided by `A` will not impact the catalog behavior at all, resulting in even less interaction. How to interleave the behaviors, while avoiding actions in `A` that violate catalog constraints, is a challenge even for engineers well-versed in both systems. We therefore construct a new test, $\texttt{(A+B)}\times k$, consisting of `A` followed by `B`, repeated $k$ times. This test will also, due to interference (let us assume `A` violates a catalog constraint) tend to fail immediately without exposing a real fault. How can we avoid these problems?

Test case reduction [90] works due to the high probability that contiguous parts of a test are related: removing *chunks* of a test can eliminate many behaviors that are irrelevant or interfering. Given a test `a1.a2.a3.a4.a5.a6.a7.a8` that fails, delta-debugging might first determine if either of `a1.a2.a3.a4` or `a5.a6.a7.a8` fails; if so, it proceeds from either. If not, it increases the granularity of reduction, and considers additional candidates, until no single component can be removed without the test no longer failing.

Cause reduction [31, 32] extends delta-debugging and other minimization approaches to reduce tests with respect to an arbitrary property, not just failure. For example to produce very fast regression tests (called "quick tests"), automated tests can be minimized to find smaller tests that retain full code coverage. Cause reduction traditionally requires as input a test that satisfies the property of interest, e.g., a failing test or one with certain coverage. However, our concept of tests implies this is not necessary. Given a test that does not fail (or provide some other useful property), cause reduction defines a search, based on removal of components, for a test that *does* meet the criteria. We first construct hypothesis compositions of tests (that do not provide useful testing), and then use cause reduction to search for a test that *does* provide useful composition of the tests. The search has a potential to succeed because in most cases the reason composition fails is interference, which can be avoided by removing the interfering parts of a test, leaving a good interleaving of test actions, made possible by the $k$ repetitions in the hypothesis composition.

A concrete application of our approach to the NASA Mars rover file system testing would work as follows. First, construct the test $\texttt{(A+B)}\times k$. If $k$ is at least one more than the max of the lengths of `A` and `B`, then there is a possibility (though not a guarantee) for cause reduction to produce *any* needed interleaving of actions: removing all but the needed actions from each copy yields all interleavings of a single copy of `A` and `B`. The extra copy is required so that the interleaving can start with either of `A` or `B`. We search for a reduction of $\texttt{(A+B)}\times k$ that: 1) does not violate catalog invariants, so is a valid test, since a core problem of composition is the creation of invalid tests and 2) covers at least the union of code coverage for test `A` and

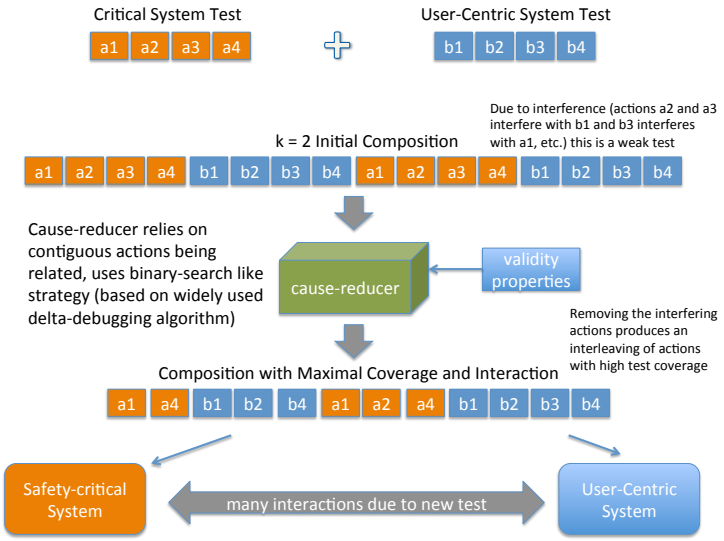


Figure 1: Composition of heterogeneous system test cases.

test `B`, and maximizes additional coverage due to interaction.

To understand the concept, consider the simple case where `A = a1.a2.a3.a4 and B = b1.b2.b3.b4`, with $k = 3$. If `a1` interferes with `B`, causing the catalog to fail with an invariant violated at action `b3`, then our approach can produce a test such as: `a2.a3.a4.b1.b2.b3.a1.a2.a3.a4.b1.b2.b3.b4.a1.a2.a3. a4`. Here, `a1` is removed from the copy of `A` before any `b3`, but remains in the final version, from which all `B` actions are removed, because it adds new code coverage of the low-level file system. One `b4` instance is removed, because it causes the low-level file system code to be in a state such that the second copy of `B` exercises less code (it forces an early garbage collection of flash blocks). Using gains in coverage to change the base test (a variation of cause reduction we refer to as "amplification"), we can direct the reduction toward this high-coverage, valid, composed test without human intervention. Figure 1 graphically shows the workflow of automated test case composition for a different set of tests.

## 3.2 Initial Results and Research Problems

We implemented a simple version of our approach in the TSTL tool for Python testing [40, 49], and applied it to compose tests for the `pyfakefs` file system [3]. We generated a large set of tests containing only file operations and a large set of tests containing only directory operations, and then applied both a näive composition approach and a simple version of our approach (without amplification), with increasing $k$. Directory-only tests had a mean branch coverage of 343.3 branches, and file-only tests had a mean branch coverage of 612.03 branches. Näive composition improved this to 651.87 mean branches, but $k = 2, 3, 4$ composition improved the means to 693.23, 698.3 and 714.37 mean branches, respectively. All results were statistically significant, and at $k = 4$, produced tests were highly complex and using chunks of original tests as "components" of complex file system operations not present in either set of simpler tests.

Because this approach suggests that test composition is essentially a search problem, an obvious question is why we use cause reduction/delta-debugging rather than a more traditional search-based evolutionary or genetic algorithm approach [6, 26, 67]. First, we believe that removal of operations is the only mutation of interest in this context: crossover or random change in test actions is likely to introduce invalid test behavior. Our assumption is that tests to be composed are valid in isolation, and only nearby behaviors are of interest (or likely to maintain single-component validity). Second, many search-based techniques expect access to branch distances and other intrusive instrumentation. This may not be feasible for embedded systems; we can use cause reduction with instrumentation only for user-centric code or, in the worst case,

for neither system. This necessitates guidance by other means.

The sketch of a composition approach here is, of course, highly incomplete. We must determine, for example, how best to choose initial values for $k$, how to identify and make modifications to delta-debugging that speed termination of the search or allow for better parallelization [47], and how to devise heuristics for abandoning a failing search and using a larger $k$. Other basic research questions are numerous: e.g., is it best to first amplify for coverage, then search for faults, or is this inefficient?

As an example of the kind of research question each new operation for tests introduces, consider the fact that for some systems composition of more than two tests may be essential to exploring interactions. However, composing more than 4-5 tests may make cause reduction prohibitively expensive. How to choose only tests likely to yield critical interactions is a novel and interesting test selection [86] problem; does composition compose? That is, for truly first-class tests we would expect that `(A+B) + (C+D) = A+B+C+D` but this is far from clear in the context of test composition, where `C` and `B` may have unavoidable interference. Another key research issue is how to extend the number of cases where composition can succeed. In some cases, composition of two tests cannot proceed without "bridging" actions contained in neither test. Some bridging actions may be possible to generate automatically, through static analysis of tests (variable renamings, object creation and deletion, etc.), or via the same term rewriting approach taken in normalization. Others may require generating random actions [7,11,44,73] that cause reduction can discard if they are not needed or are interfering. We expect that other complexities will arise during experimentation.

We plan to use test composition as an initial benchmark operation to refine our ideas of tests as first-class entities, and the building blocks required to build complex operations over tests in general settings.

# 4 Research Focus 2: Test Composition for Compilers

Although widely-used compilers such as GCC and LLVM typically work well in the common case, many bugs lurk in dark corners of C and C++ that are exercised by relatively few programs, and where the standards shed little light. It also tends to be very easy to trigger compiler bugs when using esoteric compiler options and when targeting new or little-used chipsets.

GCC and LLVM each come with thousands of unit tests that are somewhat effective at preventing new bugs from being introduced into the code base. These compilers, which we have extensive experience testing [85], will be ideal testbeds for first class test operations. The issue of validity (freedom from undefined behavior) is particularly tricky for test cases in C and C++ [25], in contrast to the simple guard-based approach of TSTL. Validity is undecidable in principle and difficult to establish in practice — as evidenced by the difficulty that the computing community has had in eliminating undefined behaviors in Internet-facing software. On the other hand, unit tests are small and either take no inputs or else take fixed inputs; the validity of this kind of program is usually easy to establish using a heavyweight checking interpreter such as tis-interpreter [5] or kcc [1,25,45].

Test case composition will be particularly useful in the compiler domain, since compiler bugs often stem from feature interactions. Consider this simple C function:

```
int foo (char x) {
  char y = x;
  return ++x > y;
}
```

It can be viewed as the composition of a unit test for the autoincrement operator, `++`, and for the integer promotion rules: in this case, the implicit promotion from `char` to `int` prior to performing the comparison. When we first ran across this function several years ago, we found that all of Clang/LLVM, GCC, the Intel compiler, and CompCert emitted incorrect code for it (the issue in CompCert was not in the verified part, but rather in an incorrect assumption about the contents of a Linux header file).

Our motivation for working on automated test case composition for compiler testing is concrete and is based on years of experience building Csmith, a monolithic test case generator for C. Csmith is around 30,000 lines of code and took at least five person-years to create; it is tightly-coupled and difficult to modify.

Basically, we do not want to create something like Csmith again: it was too difficult; there must be a better way to produce the same results.

An alternative to writing a monolithic test-case generator like Csmith is to write a lightweight generator that targets a much smaller subset of the language. We have done a few of these by hand (for example, for a small subset of C++ that targets the part of the compiler that processes templates). However, we believe that a better alternative is to integrate the new generators with a modern coverage-driven fuzzing framework such as AFL [87] or libFuzzer [2]—these tools have powerful heuristics for skewing the generated test cases towards those that will induce new coverage in the system under test. For example, recently a fuzzer for C++ expressions and loops was developed[1] (not by us) for libFuzzer. The core of this generator is 79 lines of code, plus an additional 140 lines of code to convert the internal representation into C++ code. This extremely simple C++ generator, along with the coverage-driven test harness, found seven LLVM bugs.

The proposed work is to:

- Implement additional random test-case generators for small (and therefore manageable) subsets of programming languages. For example, for C++ we could individually stress:
  - templates
  - class hierarchies
  - constructors and initializers
  - run-time type information
  - exception handling
  - nested loops such as those used in linear-algebra codes
  - constexprs
- Use these generators to find bugs in open-source compilers and other tools that process C and C++, such as static analyzers. We expect this to work, but not necessarily to produce amazing results.
- Automatically compose generated test cases—again leveraging a coverage-driven fuzzing framework—to exercise parts of compilers that process interactions between language features.
- Develop the idea of "critical composition" of test case features where the features semantically interact, as opposed to merely being present in the same test.

This application of test composition will succeed if the composed test cases can find interesting compiler bugs and cover significantly more code in the compilers under test than can test cases from the individual generators. The mechanism that we believe will allow this method to succeed is that we are dividing up the difficult job of testing feature interactions in programming language implementations into several easier tasks: generating the features individually, decomposing the smaller tests, and then integrating them into new, composed tests.

# 5 Research Focus 3: Automatic Test Decomposition

Automatically and manually generated tests often do more than one thing. In some contexts (test startup costs, ease of generation/writing), this is beneficial; however, in other contexts, such as ease of understanding, granularity of regression analysis [86], or likelyhood of being flaky [63,74], tests that combine multiple purposes are problematic.

We propose that such complex tests can automatically decomposed into multiple (often overlapping) tests that, taken together, accomplish the same goal. The core idea of the approach is to analyze the causal structure of the test in reverse, using test reduction algorithms (delta-debugging [90] or normalization [37])

[1] `https://github.com/llvm-mirror/clang/blob/master/tools/clang-fuzzer/cxx_proto.proto`

```
ORIGINAL TEST:
component0 = "b"
os0 = newFakeOS()
path0 = "/Volumes/ramdisk/test"
path0 += "/" + component0
path1 = "/Volumes/ramdisk/test"
os0.link(path1,path0)
component0 = "e"
os0.makedirs(path0)
component2 = "c"
path0 += "/" + component2
result = os0.readlink(path0)
path1 += "/" + component0
os0.makedirs(path1)
os0.link(path0,path0)
os0.rename(path1,path0)
```

```
DECOMPOSITION 1:
os0 = newFakeOS()
path0 = "/Volumes/ramdisk/test"
path1 = "/Volumes/ramdisk/test"
component0 = "e"
component1 = "c"
path0 += "/" + component1
path1 += "/" + component0
os0.makedirs(path1)
os0.rename(path1,path0)

DECOMPOSITION 2:
os0 = newFakeOS()
path0 = "/Volumes/ramdisk/test"
component0 = "c"
path0 += "/" + component0
os0.link(path0,path0)
```

```
DECOMPOSITION 3:
os0 = newFakeOS()
path0 = "/Volumes/ramdisk/test"
component0 = "c"
path0 += "/" + component0
result = os0.readlink(path0)

DECOMPOSITION 4:
component0 = "b"
os0 = newFakeOS()
path0 = "/Volumes/ramdisk/test"
path0 += "/" + component0
path1 = "/Volumes/ramdisk/test"
os0.link(path1,path0)
```

Figure 2: Automatic decomposition of a complex file system test into easier to understand (and use to generate new tests) component tests with equivalent total code coverge.

as a building block. The algorithm, at a high level, is to first execute the test and observe all causally meaningful events. The principles in Section **??** allow us to limit the set of events to a much more manageable set than in traditional execution capture and replay: typically only binary coverage of statements and branches (or more complex coverage such as paths), plus assertion statements, are relevant. Consider a test $t =$ `a b c d e f g` that produces ordered events $e_1.e_2.e_3.e_4.e_5$. Decomposition aims to produce a *set* of tests that also produce $e_1$-$e_5$. We begin by reducing $t$ with respect to the property that it still produces at least $e_5$. Suppose this yields test $t_1 =$ `a b e g` that produces events $e_1.e_3.e_5$. The next reduction will take $t$ again but this time with the criteria that the reduction produce $e_4$. If the result is $t_2 =$ `a c d f` and events $e_1.e_2.e_4$, then decomposition is complete, with two tests. Decomposition begins from the last events in a sequence, on the expectation that many earlier events will be required to produce those events, in order to limit the number of reduction queries and produce "natural" tests, essentially pulling apart a tangled web of casusality into multiple, independent (but with possible duplication of both test steps and events) threads. As a concrete example, consider the file system test and its decomposition, shown in Figure 5, produced by our initial implementation of decomposition in TSTL. Decomposition preserves actual interactions that produce novel behavior (code coverage), but removes accidental interactions.

There are many challenges in the problem of decomposition: our TSTL implementation supports only code coverage events, not correctness checks such as assertions and differential comparisons, or stress events such as system load or free memory. Decomposition for multi-threaded or multi-process tests, where event ordering in a particular run may be arbitrary is also essential for scaling to real-world system tests, where the impact is greatest.

We believe that test decomposition will be similarly useful for compiler testing; for example, decompositions can inform novel metamorphic [13] compiler testing approaches [59] or make it possible (along with normalization) to extract complex canonical sub-programs [92] from complex real world code. Decomposed test cases will be more likely to be amenable to composition operators as well, enabling composition-based test generation.

More generally, automatic causally-sound decomposition of tests is extremely promising for *all* test generation techniques that use *seed* tests to generate new tests [36]. We propose to extend causal decomposition to produce better seed tests for AFL, seeded versions of KLEE, EvoSuite, and TSTL itself. In previous work, we showed that reducing the size of seed tests for KLEE could greatly improve performance of symbolic execution [32, 91], and we expect similar (or better) gains should be possible with decomposed tests, which can decrease test size greatly beyond what is possible with cause reduction alone.

Decomposition is also a potentially powerful tool in the avoidance of flaky tests [64, 69, 74]. Flaky tests

are tests that sometimes fail for non-deterministic reasons (not related to the faultiness of the underlying code), and pose a major problem for Google-scale testing [68, 69]. Analysis by Google engineers suggests that sheer test size [63] is a major factor contributing to flaky behavior. We have discussed (and submitted a Google Faculty Research Proposal sponsored by John Micco based on) the idea of using decomposition to reduce the size, and thus potential for flaky behavior, of tests.

Finally, decomposition supports reducing the granularity and thus improving the effectiveness of methods we have recently proposed [16], as part of the DARPA BRASS program [53], for using tests as a basis for specifying resource-adaptation specifications for use in self-adapting software.

# 6 Related Work

Some of the related work is the general literature on software testing problems [9, 72] that is relevant, which is cited throughout this proposal, e.g. recent work on regression testing [12, 23, 29, 78, 86], seeded or parameterized test generation [54, 55, 66, 75, 81, 82, 87], flaky tests [27, 57, 64, 68, 69, 74] and compiler testing [1, 25, 45, 59, 85, 92]. 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: [15, 46, 61, 62, 70, 77, 79, 88–90]. Sai proposed a very limited, ad hoc version of semantic minimization that aims to go beyond the simplifications possible with conventional delta-debugging [93]. Work on automatically producing readable tests [19, 20] is also related, in that it aims to "simplify" tests. Test case purification [83] is a kind of limited (in approach and in goal) decomposition, as are some efforts to produce unit from system tests [22, 56, 71, 80]. To our knowledge, our own proposal on test composition [34] 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* [37]. 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 [48] 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 [37] 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 [76].

The idea of representing the generality of tests with the goal of enabling complex operations has been relatively unexplored, to our knowledge [33]. Andrews et al. presented a pool-based concept as part of an effort to analyze bounded exhaustive testing vs. random testing [10], but did not base any idea of test operations on this approach. The UDITA language [28] considers related ideas, but focuses only on generation approaches, not operations on existing tests. The TSTL language [40, 49] 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 [50–52].

# 7 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

| 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

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 [17,65] 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) [4, 18]). 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) [24]? Can it reduce test flakiness?

# 8 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.

## 8.1 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.

### 8.2 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 [43] 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.

### 8.3 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 [84] 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.

## 9 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 [14]. 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 [31], 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 [32, 91]. 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 [8]. 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 [40,49], 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 [30] and exploration of how individual test features impact the coverage and fault detection statistics of random tests [42]. The basic swarm testing approach has been extended to allow production of focused random tests targeting particular code [7]. 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 [7,8,14,30–32,41,42,49,91] 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 [21,58–60]. 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] libfuzzer - a library for coverage-guided fuzz testing. `https://llvm.org/docs/LibFuzzer.html`.

[3] pyfakefs. `https://github.com/jmcgeheeiv/pyfakefs`.

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

[5] tis-interpreter. `http://trust-in-soft.com/tis-interpreter/`.

[6] Shaukat Ali, Lionel C. Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering*, 36:742–762, 2010.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] Andrea Arcuri, Muhammad Zohaib Z. Iqbal, and Lionel C. Briand. Formal analysis of the effectiveness and predictability of random testing. In *International Symposium on Software Testing and Analysis*, pages 219–230, 2010.

[12] 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.

[13] T. Y. Chen, S. C. Cheung, and S. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, Hong Kong Univ. Sci. Tech., 1998.

[14] 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.

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

[16] Arpit Christi, Alex Groce, and Rahul Gopinath. Resource adaptation via test-based software minimization. In *IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, pages 61–70, 2017.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.

[26] Gordon Fraser and Andrea Arcuri. EvoSuite: automatic test suite generation for object-oriented software. In *ACM SIGSOFT Symposium/European Conference on Foundations of Software Engineering*, pages 416–419, 2011.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

[35] Alex Groce, Klaus Havelund, Gerard Holzmann, Rajeev Joshi, and Ru-Gang Xu. Establishing flight software reliability: Testing, model checking, constraint-solving, monitoring and learning. *Annals of Mathematics and Artificial Intelligence*, 70(4):315–349, 2014.

[36] Alex Groce and Josie Holmes. Provenance and pseudo-provenance for seeded learning-based automated test generation. In *NIPS 2017 Interpretable ML Symposium*, 2017. Accepted for publication.

[37] 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.

[38] Alex Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *International Conference on Software Engineering*, pages 621–631, 2007.

[39] Alex Groce, Gerard Holzmann, Rajeev Joshi, and Ru-Gang Xu. Putting flight software through the paces with testing, model checking, and constraint-solving. In *Workshop on Constraints in Formal Verification*, pages 1–15, 2008.

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

[41] 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.

[42] 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.

[43] 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.

[44] Richard Hamlet. When only random testing will do. In *International Workshop on Random Testing*, pages 1–9, 2006.

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

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

[47] Renáta Hodován and Ákos Kiss. Practical improvements to the minimizing delta debugging algorithm. In *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 1: ICSOFT-EA, Lisbon, Portugal, July 24 - 26, 2016.*, pages 241–248, 2016.

[48] 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.

[49] 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.

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

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

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

[53] Jeffrey Hughes, Cassandra Sparks, Alley Stoughton, Rinku Parikh, Albert Reuther, and Suresh Jagannathan. Building resource adaptive software systems (BRASS): Objectives and system evaluation. *SIGSOFT Softw. Eng. Notes*, 41(1):1–2, February 2016.

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

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

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

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

[62] 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.

[63] Jeff Listfield. Where do our flaky tests come from? `https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html`, April 2017.

[64] 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.

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

[66] 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.

[67] Phil McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14:105–156, 2004.

[68] 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.

[69] 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.

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

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

[72] 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.

[73] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *International Conference on Software Engineering*, pages 75–84, 2007.

[74] 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.

[75] 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.

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

[77] 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.

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

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

[80] 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.

[81] 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.

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

[83] 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.

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

[85] 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.

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

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

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

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

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

[91] 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.

[92] 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.

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