Formal Methods in the Field: A Pathway for Combining Formal, Static, and Dynamic Analysis of Real-World Embedded Systems

1 Overview and Objectives

1.1 Problem Statement

The core problem we aim to address in this proposal is that *use of formal modeling, advanced static analysis, and advanced dynamic analysis tools in C and C++* for verification and validation, especially on critical, timing-dependent, embedded and cyber-physical systems, is prohibitively difficult and lacks sufficient synergy for effective application in real-world projects. This limitation applies even to systems built in an academic research context, unless the context is specifically that of using such systems (that is, unless the research is primarily *about* methods for verifying and testing systems, rather than work on an embdeded system for its own sake). Furthermore, even when use of these techniques to ensure correctness, reliability, or security *is* a focus of the project, such use is almost always limited to one type of effort—model checking, theorem proving, or dynamic analysis (e.g., automated test case generation). A major cause for this difficulty is the *lack of synergy* between these related efforts, the failure of effort in one context to transfer to another context. In short:

- Learning to use a formal modeling language and tool, such as UPPAAL [11], PRISM [59], or SPIN [56], provides help in discovering defects in a high-level, abstract formulation of a model or protocol, but seldom helps with implementation-level problems not directly modeled in the formalism.
- Many static analysis tools are primarily "bug detectors" (e.g, Coverity or CodeSonar), whose output is essentially limited to a list of possible problems. These tools, however, seldom provide any means for producing tests that can help distinguish false positives from real problems with the code.
- More powerful static analysis tools, such as FRAMA-C [57], provide proofs of correctness for limited aspects of a system, and a rich specification and annotation language. There is no connection between this annotation and either formal modeling or test generation using anything other than a concolic tool that is limited in functionality and scalability.
- There are a large variety of automated test generation tools; however, again, effort spent in learning one of these tools only partially applies to learning a different tool. Furthermore, many of the most powerful such tools (e.g., AFL [91]) are specialized to the problem of finding memory safety vulnerabilities, and provide no support for the kind of testing needed for other types of faults in e.g., communication protocols used in distributed embedded systems. Finally, none of these tools significantly leverage specification and verification effort from formal modeling or advanced static analysis.

Consider the case of an engineer working on a custom, low-energy consumption, communication protocol for use in a distributed system consisting of low-power sensors and actuators. If the engineer builds a formal model of the protocol, she will discover that this extensive effort provides no help, other than an improved concept of the system, in proving the correctness of the actual implementation, even if the property to be proved exists in the model. If the engineer begins instead by building an automated test generation harness she again discovers that despite having spent considerable time expressing pre-conditions and post-conditions for various functions in the implementation, to guide test generation, the work must be duplicated when she decides to try to formally prove the correctness of core functionality. Had she begun with the proofs, again, logically related (or even equivalent) information would have to be re-expressed, in a different language, to perform test generation. Not only must our engineer learn three tools, but effort spent in using one tool almost never carries over to another approach. In almost all cases, there is simply not enough time or energy available to make use of the full spectrum of available technology. In practice, *no advanced correctness technology may be used at all.* After all, it is hard to predict which technology will have the greatest payoff, or even work at all, so perhaps it is best to just put more effort into manual testing.

1.2 Proposed Solution

While allowing efforts from any form of formal or automated verification or validation attempt to maximally carry over to other forms (i.e., formal models to code annotations for static analysis, code annotations to test harnesses, test harnesses to code annotations, test harnesses to formal models, formal models to test harnesses, and code annotations to formal models) is the ideal goal, simply making it possible to follow *one* critical path to combine methods is feasible given current technologies in the sub-domains (formal modeling, static analysis, and dynamic analysis) and a set of specific advances in bridging the gap between the technologies.

Thus the question becomes: which path is most important to realize? Our approach is based in the reality of the embedded systems domain, where, while formal modeling is sometimes used, there is, in real-world efforts, *always* an implementation. The most basic obstacle to the adoption of formal methods in embedded systems work is that if the realities of development preclude extensive formal efforts, and there is only the usual informal design effort or adaptation of an existing implementation, formal methods are often simply inapplicable. By focusing on adding annotations to the actual implementation code, and exploiting those annotations to enable a set of potentially bug-finding or correctness-proving analyses, we promise to *always* give embedded systems engineers a reasonable payoff for their formal specification effort, often in a form (failing tests) that always provides practical benefit and can never be dismissed in critical embedded systems efforts.

This project therefore proposes to make it possible to introduce specifications into implementation code that can be directly checked using sophisticated automated test generation strategies, including symbolic execution, advanced fuzzing, explicit-state model checking, and SAT/SMT-based bounded model checking. Furthermore, these specifications can be directly exported to form the basis for formal models using, e.g., timed automata. In the long run, to benefit those developers who are more open to formal methods already, we hope that these annotations can be imported from a timed automata representation, but we begin where most embedded systems developers are, now, not where we hope they may be in some indeterminate future. To further improve the value of our approach, we focus on integrating static and dynamic analysis tools that are, themselves, frameworks/front-ends allowing application of multiple approaches. Our intellectual committment is to enabling *a maximum diversity of analysis methods* with *a minimum of specification and tool-learning effort*, to make formal methods attractive *simply because of their expected cost-benefit ratio*.

This project is specifically focused on communication protocol implementations in embedded systems where timing is critical to the modeling of behavior, but we expect that our solution will generalize to other critical C and C++ systems development scenarios. Note that we target the common, hard, case where our approach will be applied to systems with partial or complete existing implementations: typical legacy embedded C code. We expect developers to learn new tools, but not new programming paradigms or languages. The contribution to embedded systems design therefore is not a radical reworking of development methods, which, like many formal methods efforts in the past, is unlikely to achieve widespread adoption, but the introduction of, essentially, *an advanced form of unit testing, with more powerful methods for specifying "testing and verification problems" and more powerful, low-effort ways to try to solve those problems.* The end-result for embedded systems engineers will be an approach to development that works with legacy code, and does not require a fundamental shift in approach, but will *support* early adoption of formal modeling as, e.g., timed automata, if present, and, critically, will allow engineers to express the correctness properties in a full-featured way directly in code, while enabling use of a variety of methods to check those properties. This will modify development, in that design-for-testability and design-for-verifiability will become second nature. Just as the introduction of the possibility to express *and check* strong typing constraints changed Python development practices for high-value projects, the introduction of the possibility to express *and check* correctness properties will change embedded system development.

Summary: This project is based on the following core ideas:

1. **The primary obstacle to adoption of formal methods approaches in embedded systems development is not a lack of relevant methods and tools.**
2. **In particular, there are methods and tools that apply to the *implementation* of embedded systems in C and C++; every embedded software system requires an implementation.**
3. **However, learning and using any one of these tools may or may not "pay off" and the effort spent is only of limited application when applying another tool.**
4. **Therefore, to improve embedded systems development via formal methods we need:**
   (a) **an *implementation-focused common framework* for applying methods and tools and**
   (b) **a focus on *practically-inspired* improvements to the methods and tools encompassed by that framework enabling them to work better *within the framework*.**

1.2.1 PI Qualifications

PI Groce has a long history with formal methods, including involvement in design and development of well-known model checkers, and application of model checkers at NASA/JPL on flight software for the Mars rovers, including application of and development contributions to, the SPIN and CBMC model checkers. More recently, he has primarily focused on developing algorithms and tools for automated software testing; he is a core member of the DeepState [35, 36, 85] and TSTL [53] design and development teams. He brings to this project practical experience using verification and testing tools on critical real-world systems, and engineering such tools to be usable by engineers who are domain experts, not verification or testing experts.

Co-PI Nghiem has an extensive background in control and autonomous systems, and application of formal methods in control systems. He has long experience working with timed automata and the UPPAAL tool family. He developed, and was granted a U.S. patent for, methods for testing and verifying temporal logic specifications of hybrid systems—systems with both continuous and discrete behaviors. He brings to this project advanced knowledge of real-time embedded systems and practical experience applying verification methods and tools to them. He will also apply our methods and approaches to a multi-agent robotics system.

Co-PI Flikkema's current work includes research in energy-efficient embedded systems and networks, inference of the embedding environment, wireless sensor/actuator networks for monitoring and control of environmental and ecological systems, and cybersecurity with focus on IoT. Like Co-PI Nghiem, he ensures that developed approaches will be suitable for engineers whose primary focus is *building working systems*.

1.3 Intellectual Merit

The aim of this proposal is to (1) identify a set of principles for the analysis (formal, static, and dynamic) of communication protocols and their implementations in embedded systems; (2) match these theoretical principles with tools usable by engineers developing such systems; and (3) enable the synergistic use of enhanced versions of these tools in two real applications through a common framework with minimal duplication of effort and maximal extraction of information from shared annotations. In the first case study, we will analyze networks of wireless sensor/actuator nodes deployed in the Southwest Experimental Garden Array (SEGA) [89, 31], a distributed facility for examining climatic, genetic, and environmental factors in plant ecology. The second case study will use the framework to formally verify and dynamically test the distributed coordination code of multiple autonomous ground and aerial robots in a lab setting.

Figure 1 shows the overall concept. The core open research problems addressed are represented by two sets of arrows. First, an engineering design, expressed as C code, is provided, and annotated with correctness properties, information about the expected environment (constraints on sensor values, etc.), and hints to guide heuristic application of tools ranging from fuzzers to symbolic execution engines to model checkers. While it is not the focus of this project, code to apply advanced static analysis or generation of timed automata (TA) models can also be automatically generated. Our goal is to certify whether the C code truly implements the correctness properties in question, with a focus on dynamic approaches:

1. First, the TA model can be used to check high-level properties of the design, ignoring many low-level implementation details. If core timing properties are wrong, finding bugs in the implementation is best left

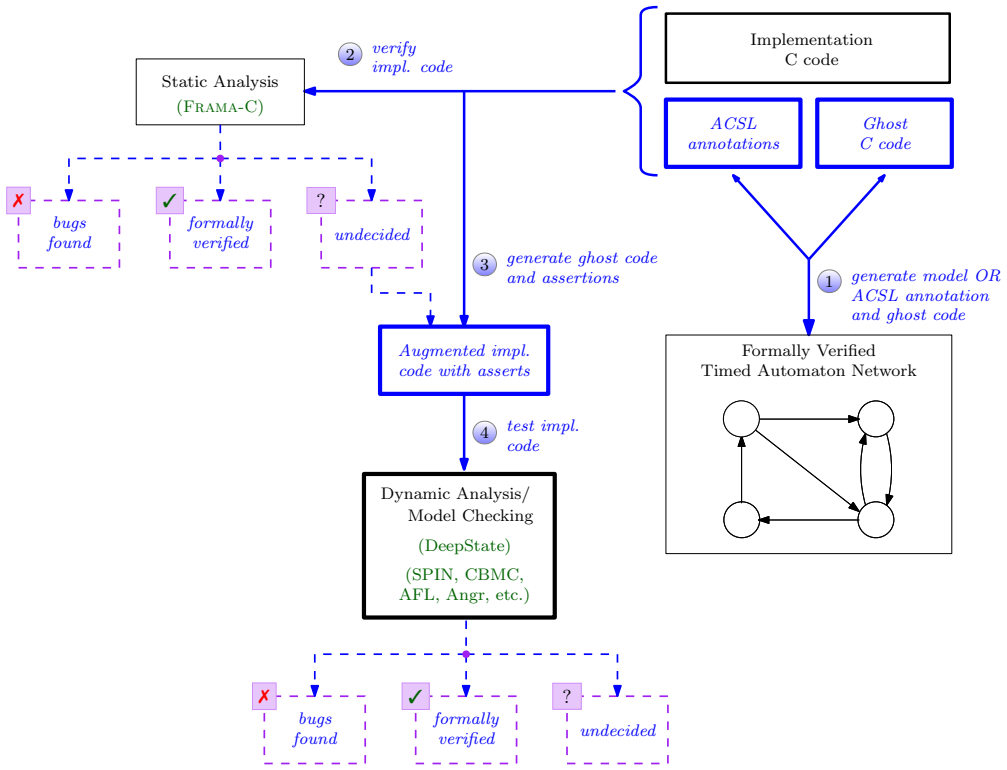

Figure 1: Overview of the proposed research.

until the basic design is correct. However, this step may be skipped.

2. The implementation code with the ACSL annotations and ghost code is checked by a static analysis tool, such as FRAMA-C. There are three possible outcomes: **verified:** the implementation can be formally verified, which means that it meets the design specifications that can be checked statically; **bugs found:** non-spurious bugs are found in the implementation, showing that it can violate the specification, and the bugs must be fixed; or **undecided:** the static analysis tool is unable to prove or disprove correctness of the implementation, which will often be the case. Again, this step can be skipped (though the overhead of running FRAMA-C, given we adopt its core specification language, is unlikely to be prohibitive.
3. Finally, the focus of our efforts is a multi-pronged attempt to refute correctness (or increase our confidence in it) via dynamic analysis—automated test generation—and implementation-level model checking. Ghost code, additional runtime assertions, and needed test-harnesses are *automatically generated* from the annotated code.
4. The augmented implementation code is then analyzed using the DeepState [35] framework, which serves as a front-end to highly scalable fuzzers, as well as to symbolic execution tools and explicit-state and SAT/SMT-based bounded model checkers that can detect more bugs at the cost of restricted applicability or scalability. Generated test cases may prove the system faulty, or they may leave us more confident the system is correct.

Our focus is on providing a unified specification method that can be applied to source code itself, and on enabling and enhancing the dynamic analysis and explicit-state and SAT/SMT bounded model checking aspects of the approach. We believe these approaches are currently the most difficult to apply, and the most likely to dramatically improve the ability to detect bugs in complex embedded systems designs.

Principles: Assuming that a communication protocol can in theory be described as (probabilistic) timed automata [6], which satisfies temporal logic formulas [16], and implemented as a set of imperative programs that realize these timing constraints and other correctness conditions, we ask:

- Given a set of annotated programs, how can we best automatically find bugs in those programs (and, in some circumstances, for some properties, prove correctness), based on an annotated specification?
- Can we generate a skelton of the timed automata model from annotated programs, in order to facilitate analysis of the design, in order to increase adoption of design-level analysis by traditional embedded systems developers?

Note that these problems differ considerably from the more studied, but more limited, synthesis problem. We are not assuming that system development will involve first producing a formal model, then using that model to automatically generate an implementation; rather, we consider the typical real-world scenario, where modeling is a separate activity, either undertaken after implementation due to concerns about reliability, or an activity during design that only indirectly informs the implementation. That is, the more studied problem is producing a runtime semantics for a model; we address the problem of reconciling a runtime semantics with a model semantics, without unrealistic burden on engineers. We begin with the implementation side, because while realtively few embedded systems engineers currently use formal models, all must implement their systems. Furthermore, to the extent that developers do more rigorous testing and analysis of their systems, they primarily use unit tests, simple fuzzing, and static analysis. Any method that aims to increase the adoption of formal methods and more powerful automated test generation approaches, including implementation-level model checking and symbolic execution, is likely to be successful to the extent that it enters the design and implementation process via this existing "bridge to engineers and developers." Engineers at Google have referred to this approach as "meeting developers where they are" [73].

Tools: We will focus on C code, using DeepState [35] as a front-end for dynamic and implementation-level model checking approaches, and UPPAAL [11] and PRISM [59] for the analysis of protocols; FRAMA-C will provide a powerful static analysis framework, and we well adopt the ACSL language developed for FRAMA-C as a basis for our specification language. The primary open research questions here are numerous, and include: (1) how to extend existing specification languages to support timing and uncertainty; (2) how to assign the same meaning to a specification construct in various contexts, ranging from fuzzing to symbolic execution to explict-state model checking to bounded model checking in the "dynamic" DeepState world, and including a static context for FRAMA-C and a modeling conext for timed automata; (3) how to minimize annotation burden while allowing developers to include information that can be exploited by those methods: e.g., to make intelligent use of pre-conditions in fuzzing, to automatically derive loop bounds in bounded model checking, and to restrict branching factors in explicit-state model checking; (4) how to handle intra-program parallelism; (5) how to effectively translate a failed proof effort in FRAMA-C into a representation of a testing problem (to find counterexamples refuting that proof could be possible) in a dynamic setting; and (6) how to ensure that the methods are sufficiently automatic and behave in ways engineers (not formal modeling, static, or dynamic analysis experts) will expect.

Our focus will be on *practical* solutions, guided by the embedded domain experts, rather than on purely theoretical approaches that do not scale to real systems. Practical solutions here require fundamental contributions to system and specification design and semantics and dynamic analysis and model checking approaches.

2 Background and Preliminary Research

2.1 A Foundation for Implementation-Level Specification

While the correctness of an implementation with respect to a formal functional specification provides a very strong form of guarantee, it can be very costly to achieve. Currently it is mostly reserved to domains where it is required by regulations or offers a competitive advantage. In practice, it is very useful to rely on a combination of formal methods to achieve an appropriate degree of guarantee: automatic static analysis to ensure the absence of runtime errors, deductive verification to prove functional correctness of aspects of the code, and runtime verification for parts of the code that cannot be (or are not yet) proved using deductive verification, or parts of code that generated *warnings* from static analysis requiring confirmation of a problem or refutation of the feasibility of an error.

A core question in our approach is how to represent a specification in a way that is intimately tied to real implementation code. However, as development of a specification language is not our primary focus, we prefer to build on existing work in the area, extending it when required.

This project will therefore make use of the ideas developed in the ongoing work on the FRAMA-C (`https://frama-c.com`) [57] tool as a foundation for the specification and annotation language. It is a widely-used source code analysis platform that aims at enabling verification of industrial-size programs written in ISO C99 source code. FRAMA-C fully supports combinations of different approaches, by providing its users with a collection of *plugins* for static and dynamic analyses of safety- and security-critical software. Moreover, collaborative verification across cooperating plugins is enabled by their integration on top of a shared kernel, and, most critically for our purposes, their compliance to a common specification language: ACSL [9]. ACSL, the ANSI/ISO C Specification Language, is based on the notion of a contract as in JML. the Java Modeling Language [64]. ACSL allows users to specify functional properties of programs through pre/post-conditions, and provides different ways to define predicates and logic functions. Many built-in predicates and logic functions are provided, to handle, for example, pointer validity or separation. FRAMA-C is very appropriate for the verification of typical legacy embedded C code. Its specification language is rich but easy to understand: ACSL is essentially a typed first-order logic that contains C expressions.

Using ACSL/FRAMA-C as a basis also means that while focusing on our dynamic and model checking tasks, use of our methods automatically provides an additional benefit for embedded systems engineers: access to the current set of FRAMA-C analyses, which are themselves powerful and useful. For example:

- *Value analysis* is a program analysis technique that computes a set of possible values for every program variable at each program point, based on *abstract interpretation* [28]. The core idea is to compute an abstract view of values of variables in the form of *abstract domains*. Abstracting variable values, e.g., as intervals, can detect invalid pointers, invalid array indices, arithmetic overflows or divisions by zero. The EVA (Evolved Value Analysis) plugin is well integrated with the FRAMA-C ecosystem, and is the basis for many other derived plugins (see [57]).
- WP is a *deductive verification* plugin provided with FRAMA-C, based on a weakest precondition calculus. Given a C program annotated in ACSL, WP generates the corresponding proof obligations that can be discharged by SMT solvers or with interactive proof. Recent work showed how to avoid using an interactive theorem prover in some cases even for inductive lemmas [12], using function contracts in ACSL and loop annotations (verified using SMT solvers), making it more possible for engineers to produce some proofs.

FRAMA-C was initially designed as a static analysis platform, but it was later extended with (limited) plugins for dynamic analysis. One of these plugins is E-ACSL, a runtime verification tool. E-ACSL supports runtime assertion checking [26]. Assertions are very convenient for detecting errors and providing information about how a failure occurred. In FRAMA-C, E-ACSL is both the name of the assertion language and the name of a plugin that generates C code to check these assertions at runtime. E-ACSL is a subset of ACSL: the specifications written in this subset can therefore be used both by WP and E-ACSL. WP tries to prove the correctness of these assertions *statically* using automated provers, while the plugin E-ACSL is used to translate these assertions into executable C code. In this case the assertions are checked *dynamically*.

However, the E-ACSL plugin does not assist developers in the most difficult part of dynamic analysis: constructing a set of tests that exercise runtime checks to find as many bugs as possible or build confidence in the correctness of the system. The only assistance provided by FRAMA-C in this task is, as discussed in the next section, very limited in capability. Rather than "re-inventing the wheel" and offering a solution that lacks a strong static aspect (given our focus on dynamic analysis and model checking), we choose to address this limitation, and extend ACSL and E-ACSL, providing a strong static aspect to our approach, and strengthening our focus on common frameworks and cross-tool usability.

2.2 Dynamic Analysis and Model Checking with DeepState

While FRAMA-C provides powerful tools for static detection of program faults and generation of runtime checks for properties that cannot be discharged by formal proof or sound static analysis, it provides only

limited, and difficult-to-scale, ability to generate program inputs to exercise runtime checks, limited to one tool, PathCrawler [87], that aims to produce a unit test for a single function, using concolic testing (dynamic symbolic execution [34]). In cases where this fails to scale, PathCrawler will fail. Furthermore, PathCrawler is tuned to the problem of testing a single function, not producing more complex scenario-based tests of a set of functions that must coordinate state changes. Finally, PathCrawler is not an open source, extensible system, may be costly to acquire and use, and is arguably impossible to extend.

The limitation of dynamic analysis tools to PathCrawler is a major weakness of FRAMA-C from the perspective of a user. Scalability of symbolic-execution-based test generation methods is extremely difficult to predict, and producing complete and exhaustive preconditions that allow a function to be tested entirely in isolation is often either too time-consuming or essentially impossible, because the actual environment is only represented by the set of states reachable using a set of coordinating functions or a library. These problems are pressing, for several reasons. First, full formal proof of correctness is, at present, impractical for most realistic systems. The actual work of fault detection and validation of software still relies, fundamentally, on effective testing. Moreover, modeling and even static approaches often must rest on a basis of numerous un-examined assumptions about the behavior of hardware systems and low-level system behavior (e.g., what operating system calls actually return). Only actual concrete inputs—tests—can be executed in a completely realistic environment, including real hardware. Only tests can satisfy regulatory requirements on code coverage such as those imposed on civilian avionics by DO-178B and its successors [75]. Furthermore, only testing can prove faults are not spurious, the result of imprecise abstraction or weak assumptions. In sum, dynamic analysis can detect otherwise invisible faults, and confirm the reality of statically detected faults.

Most developers do not know how to use symbolic execution tools; developers seldom even know how to use less challenging tools such as gray-box fuzzers, even relatively push-button ones such as AFL [91]. Even those developers whose primary focus is critical security infrastructure such as OpenSSL are often not users, much less expert users, of such tools. Furthermore, different tools find different faults, have different scalability limitations, and even have different show-stopping bugs that prevent them from being applied to specific testing problems. DeepState [35] addresses these problems. First, developers *do*, usually, know how to use unit testing frameworks, such as JUnit [32] or Google Test [2]. DeepState makes it possible to write parameterized unit tests [83] in a GoogleTest-like framework, and automatically produce tests using symbolic execution tools [78, 79, 77, 68], or fuzzers like AFL [91] or libFuzzer [76]. DeepState targets the same space as property-based testing tools such as QuickCheck [24], ScalaCheck [70], Hypothesis [66], and TSTL [40, 53], but with harnesses that look like C/C++ unit tests. DeepState is, most importantly, the first tool to provide a front-end that can make use of a growing variety of back-ends for test generation. Developers who write tests using DeepState can expect that DeepState will let them, without rewriting their tests, make use of new symbolic execution or fuzzing advances. The harness/test definition remains the same, but the method(s) used to generate tests may change over time. In contrast, most property-based tools only provide random testing, and symbolic execution based approaches such as Pex [82, 84] or KLEE [18], while similar on the surface in some ways, have a single back-end for test generation. DeepState's flexibility is evident: in the last year, DeepState added support for the Eclipser [23], Angora [20], and Honggfuzz [3] fuzzers, as well as an ensemble [22] mode supporting the use of multiple fuzzers at once [19].

In effect, DeepState targets the same space (providing the technology and translation between different semantics necessary to use different verification/bug-detection technologies) as this proposal, but narrowed to the domain of generating test inputs. DeepState has already been used to test (and find bugs in) a user-mode ext3-like file system developed at the University of Toronto [80, 37]. DeepState is being considered as a basis for automatic testing for components in NASA's open source flight software framework FPrime [14, 69], and is a fully open source system [85], supported by Trail of Bits (a New York based security analysis company, which uses DeepState in security audits). Although only released in early 2018, DeepState is already one of the most popular property-based testing and fuzzing projects on GitHub, and has been used internally by both startups and well-established companies and in security audits by Trail of Bits. There have even been informal discussions of integrating DeepState, once matured, into a future release of Google's GoogleTest

platform, one of the most widely used unit testing frameworks in the world.

3 Research Plan

3.1 DeepState and Automated Test Generation

As discussed above, a full workflow for verification of realistic systems requires a first-class, flexible, *dynamic* analysis component, such as DeepState [35]. The research challenge is to translate ACSL-annotated code for use in FRAMA-C into a full-featured DeepState test harness:

1. The *specification* of correctness must be translated into an executable form. To some extent, the existence of the E-ACSL executable subset of ACSL, and libraries for runtime checking of properties satisfies this condition. DeepState can support any C/C++ executable method of checking for correctness. However, some executable specifications need to be modified to be efficiently handled when the DeepState back-end is a symbolic execution tool. DeepState's nature as a test generation tool means that it supports constructs, such as `Minimum`, `Maximum`, and `Pump`, not usually available in executable specifications. Tailoring E-ACSL usage for DeepState therefore requires a custom effort, including extending the semantics of executable specifications and optimizing the implementation for symbolic execution and fuzzing. Finally, because our domain critically involves timing, we need to implement DeepState handling (and E-ACSL representations for) deadlines, and specification of function-level deadlines including arbitrary, specified, "runtimes" for code that operates via simulation rather than real hardware (or in symbolic execution).
2. The *assumptions* that control which tests are considered valid must be translated in the same way; normally, E-ACSL simply translates these into further assertions (as pre-conditions to check at runtime), but in DeepState, we need to distinguish between `ASSUME` constructs where failure indicates an invalid test and `ASSERT` constructs where failure indicates a failing test.
3. The inputs to a function must be translated into code controlling the input values that DeepState provides, including ranges and types. When input types are simple, this process is straightforward; however, when functions take, e.g., arbitrarily sized arrays, linked lists, or other complex structures, this becomes a problem of constructing a test harness that (1) makes fuzzing and symbolic execution scalable but (2) uses large enough structures to expose subtle bugs. Moreover, because DeepState supports strategies for input generation, such as forking concrete states for values too complex for symbolic execution using the `Pump` construct, the translation must determine when such strategies are appropriate, and apply them.
4. In many cases, checking a single function may not be an effective way to detect faults; only a sequence of API calls can expose a problem in a system (e.g., that a function produce a state that causes another function to violate an invariant). ACSL annotations provide enough information for a fully-automated translation to a harness enabling dynamic analysis in the case of proving properties of a single function, but not for groups of functions. Moreover, even in cases where the violation of a specification can, in theory, be discovered without calling multiple functions, the state space may be too large to explore with a fuzzer or symbolic execution tool. In such cases, exploring only states produced by valid call sequences has two benefits: first, the space itself may be much smaller, and easier to explore, than the full set of possible input values. Second, errors in this part of the input space are more important. Even if a precondition is not sufficiently restrictive to guarantee correct behavior, if the "bad" inputs are never, in practice, generated by the functions that modify system state, the fault may not matter. In cases where constructing a sufficiently exact precondition is difficult for engineers, such "in-use" verification may be the only avenue to system assurance; proof is impossible without a restrictive enough precondition, and dynamic methods may scale very poorly to, e.g., a large unstructured byte buffer such as a hash table. We propose to let users annotate (in an extension of ACSL) sets of functions to be tested as an API-call-sequence group, extending recent work exploring this concept [13, 74]. E.g., annotating a set of file system functions (`mkdir`, `rmdir`, `readdir`, etc.) as such a group could allow the automatic generation of a DeepState harness that checks for cases where a sequence of valid function calls can violate a precondition or cause a fault despite preconditions being satisfied.

   These goals require significant advances in two areas of dynamic analysis: first, a complete and principled

```
void update_state(struct state_t *s, uint64_t bv) {
  ASSUME(valid_state(s));
  ASSUME(valid_bv(bv));
  ...
}
void process_both_sensor_readings(struct state_t *s) {
  ASSUME(valid_state(s));
  unit64_t s1_bv = acquire_s1();
  update_state(s, s1_bv);
  unit64_t s2_bv = acquire_s2();
  update_state(s, s2_bv);
}
void process_one_sensor_reading(struct state_t *s) {
  ASSUME(valid_state(s));
  unit64_t s1_bv = acquire_s1();
  update_state(s, s1_bv);
}
```

```
struct state_t *NewState() {
  return
    DeepState_Malloc(sizeof(struct state_t));
}
TEST(SensorReading, UpdateNeverSlow) {
  struct state_t *s = NewState();
  uint64_t bv = DeepState_UInt64();
  DeepState_Timeout(
    [&]{update_state(s, bv);},
    MAX_EXPECTED_UPDATE_TIME);
}
TEST(SensorReading, AvoidCrashes) {
  struct state_t *s = NewState();
  for(int i = 0; i < TEST_LENGTH; i++) {
    OneOf(
        [&]{process_both_sensor_readings(s);},
        [&]{process_one_sensor_reading(s);});
  }
}
```

Figure 2: Sensor reading code and DeepState test harness

approach to the problem of handling pre-conditions/assumption semantics, and second, an investigation of how to let fuzzers take advantage of the significant additional structure provided by property-based testing, including such assumptions. Consider the code in Figure 2. This defines two different tests of software that reads sensor values and incorporates them into a system state. The two tests check two different properties: `UpdateNeverSlow` ensures that updating the sensor is never too slow. It is checked, potentially, over *all* valid inputs, not just ones produced by the actual sensor reading code in `acquire_s1` and `acquire_s2`. The second test, `AvoidCrashes` starts the system up in some valid state, and repeatedly either reads both sensors or only sensor one. There is no explicit property, only the expectation that the system will not crash; tests can be executed using LLVM sanitizers to check for integer overflow and other undefined behavior. Generating such harnesses automatically from ACSL specifications is a significant challenge, but our research agenda also includes solving problems that would appear even for manual harnesses. For example, what is the proper semantics of the `ASSUME` in `update_state`? It depends on the test. In `UpdateNeverSlow`, a fuzzer will often generate an input value that violates the (possibly complex) requirements on valid states and sensor readings. These invalid inputs should not be flagged as bugs (the default behavior of E-ACSL), but instead the test should be abandoned without indicating that it failed. However, in `AvoidCrashes`, since we are not directly generating state values, that is, `update_state` is not an *entry point* for the test, assumption violations should result in failed tests. We aim to synthesize code to make assumptions automatically take on the proper semantics during test execution (including symbolic execution using constraint solvers), without the user having to redefine the behavior.

This point about preconditions/`ASSUME` brings up a second point. Preconditions, when they have an `ASSUME` semantics, are fundamentally different than other branches in code. A fuzzer will attempt to explore the behavior of branches in `valid_state` and `valid_bv` just as it explores branches in `update_state` or `acquire`. However, it is often possible to enumerate a vast number of paths that differentiate only invalid inputs, and so produce very little real testing. A classic example is "testing" a file system by producing a huge variety of unmountable file system images, rather than actually executing any POSIX operations at all [41, 45]. DeepState knows which branches are pre-conditions, and so can help avoid this problem. In some fuzzers, this means prioritizing inputs to mutate based on whether they execute any code other than validity checks; but in fuzzers such as Angora [20] and Eclipser [23] that use lightweight constraint-solving to cover branches, the process can be more sophisticated. We have begun discussions with the Eclipser team, and they confirm that identifying precondition code and devising suitable heuristics to handle it (e.g., never solve for a negation of a passed check) should improve performance. Devising effective heuristics to tune both "classic" mutation fuzzers and concolic gray-box fuzzers promises to improve test generation not only in our context, but in general. Fuzzing of individual functions or sets of functions is a highly promising

area: most fuzzing is applied at the whole-program level, where learning to produce interesting inputs can be an overwhelming problem. By focusing on a middle-ground between unit testing and whole-program fuzzing, that is by using modern fuzzer technology to drive property-driven testing, the problem may be more tractable. Prioritizing paths that are not input validation is an explicit goal of, e.g., AFLFast [15], but it must work with an implicit definition based on path frequencies, while we have access to more precise data. Given the possible complexity of a state validity check, there may be hard-to-reach—but fundamentally uninteresting—ways to create invalid input; AFLFast will prioritize such paths, while our approach will (correctly) avoid them.

This effort also connects to a second fuzzing research thrust: making specification elements that do not correspond to simple code coverage visible to a fuzzer. In this example, consider the `DeepState_Timeout` check (note that this itself is functionality we will develop as part of handing timing constraints in FRAMA-C and DeepState). Unless we break down the timing analysis explicitly using a set of conditional branches, coverage-driven fuzzers cannot distinguish an execution that is very slow (close to violating the constraint) from one that has the minimum execution time possible. We propose to make timing of such specified events visible to a fuzzer, by modifying coverage bit-vectors to incorporate bucketing of execution time. Once we add such novel coverage measures, and introduce distinctions between coverage classes (as with preconditions), we will research how to balance competing priorities in more complex notions of coverage. In addition to implicit execution properties such as timing, this effort can apply to coverage of data structures, which is also a critical problem in fuzzing data-driven code such as machine-learning algorithms, where much of the behavior is implicit in, e.g., the route taken through a forest of decision trees. We assume that as we investigate real world examples, further challenging research problems in property-based fuzzing will arise and require improving fuzzer science and art. In essence, this proposal aims to extend the work, including AFLFast [15], FairFuzz [65], VUzzer [72], and other efforts [93, 8], that prioritizes certain program paths over others in an intelligent way, by specializing that set of approaches to the case of property-based testing with a stronger specification and understanding of interacting functions and data structures.

3.2 DeepState and SAT/SMT-Based Bounded Model Checking

While automated test generation by fuzzing or binary-level symbolic execution can be highly effective as a means for finding bugs in code, other approaches are also needed to handle the kinds of code especially common in embedded contexts. In particular, embedded software often includes a large number of functions that perform complex low-level bit operations, especially for interacting with hardware and "parsing" network packets (from traditional wireless or RF-derived signals) communicating in very low-level protocols. Fuzzing or binary symbolic analysis often fails to find exact bit-values that violate correctness properties for such code; it is well known that, e.g., inverting hashes is hard for either approach. Translation to bounded SAT from the source code, without going through compilation of the overheads of binary analysis, however, can often find problematic inputs even for such code.

CBMC, the C Bounded Model Checker [58] is a well-known tool that analyzes C programs using a translation to SAT or SMT queries based on a bounded unrolling of loops. CBMC is an actively developed project, and has been used extensively in real-world development for years, including in automotive/embedded code development at Bosch and General Electric [81], in analysis of Amazon Web Services infrastructure [27], and in the analysis of flight software systems at NASA's Jet Propulsion Laboratory [45]. Using CBMC requires writing custom test harnesses using CBMC's API for expressing nondeterminism, and running the tool with a specified bound on loop executions, in addition to a number of other configuration options, which are often non-obvious (for example, which safety properties of code are checked by default has changed with some releases of the tool, sometimes causing silent failures of "working" CBMC harness code).

We propose to allow CBMC to be used as a backend for verification by DeepState, with a seamless interface, just as DeepState currently supports symbolic analysis engines such as angr and Manticore. It is notoriously hard to guess when a SAT/SMT based approach to code analysis will work well and when it will fail to scale; using a DeepState harness will allow users to try both CBMC and other test generation approaches without the effort of writing multiple harnesses.

Moreover, because choosing a loop unwinding bound imposes a serious burden on embedded engineers, we will investigate the automatic determination of reasonable loop bounds. One approach is to instrument fuzzer or symbolic-execution engine generated tests to record iterations of loops, and then use the maximum bound from those runs. Additionally, for small functions (the most likely targets for DeepState-CBMC: complex but compact bit-manipulation code), the mutation-based approach proposed by Groce et. al [49] may be useful. Finally, in some cases CBMC may be able to find interesting bugs for cases where the loop unrollings are limited, but cannot scale to larger depth limits. Using the same instrumentation that we use to estimate loop bounds, we will use the ability to guide DeepState by more kinds of coverage discussed above to focus fuzzer runs on executions with more loop iterations than the bound explored by CBMC. This will offer engineers a true partnership between verification methods.

3.3 DeepState and Explicit-State Model Checking

Just as some functions are best analyed using bounded model checking, some dynamic analysis problems are best handled by explicit-state model checking that actually executes C code, like a fuzzer, but with the capability to store states and backtrack, in order to exhaustively explore a state space, using either actual comparison of stored states or comparison of abstractions of states to guide exploration. This approach is particularly attractive for exploring sequences of API calls; this kind of test generation was used in efforts that uncovered dozens of errors in file systems at NASA/JPL [45] and in a widely-used mock file system for Python testing used by over 1,000 projects [47].

The SPIN model checker [56] offers execution of C code with backtracking [54, 55]. DeepState's `OneOf` construct has a semantics that can be matched with the SPIN nondeterministic choice, which in part inspired the DeepState construct [39, 38]. However, integrating SPIN as a back-end for DeepState is even more challenging than integrating CBMC. With CBMC, it is plausible that the mapping from DeepState to CBMC semantics can be performed entirely in terms of changing included headers so that CBMC-specific constructs have differing implementations (but not semantics); SPIN however executed C code in the context of a PROMELA model, which requires rewriting a DeepState model to embed test choices inside SPIN's constructs. This also means "lifting" DeepState API calls to the PROMELA level outside the C code, and bridging between nondeterminism visible to SPIN and determinism within C code. Such a translation is not the core research challenge however, and a substantial elaboration of techniques develop by PI Groce as long as as 2008 at JPL citeWODA08 can serve as a foundation for solving this problem. The more fundamental problem is that while CBMC and DeepState can share a semantics for, e.g., `DeepState_Int()`, allowing all values of the appropriate bit-size, a PROMELA model with a branching factor of, e.g., 2^{64} is completely useless (and, indeed, will not work). There are multiple possible solutions, ranging from using results from fuzzing to choose a limited range, to translating "flat" bit-value selection into a sequence of choices with a strong bias to a set of initial values, but an in-principle unlimited range, to using SPIN to control a seed and deterministically choosing random values, a hybrid fuzzing/explicit-state model checking approach. All of these approaches have potential problems (e.g., sequences that construct numeric values cause significant state-space explosion), but all may be needed to handle different embedded systems verification problems.

3.4 Generation of Timed Automata Model Skeletons

As noted above, one of our core assumptions is that timed automata can model the underlying protocols in many embedded systems. However, analysis of timed automata models using UPPAAL [11] and PRISM [59] is at present a skill only a small number of embedded engineers have mastered. In order to encourage more engineers to make use of these powerful formalisms, we will also develop a method to extract time automata models for both of these model checkers from annotated C code. By providing some additional information with the timing constraints that are included for checking during verification of the implementation, engineers can also make use of these tools.

3.5 Case-Study-Driven Problem Determination

The above briefly introduces a number of problems that we know in advance must be dealt with in order to enable a pathway for combining formal, static, and dynamic analysis. However, we aim to allow the case

studies described below prioritize our work, to a large extent, and are certain that other challenges will arise from having real engineers apply our approach to real systems.

3.6 Case Study: Sensor/Actuator Networks for Ecological Monitoring and Control

Overview: The first case study informing this research is the embedded software used in the Southwest Experimental Garden Array (SEGA) [25, 33, 10]. SEGA is a large collection of operational wireless sensor/actuator networks for monitoring and control of ecological systems, located at 17 sites in the states of Arizona and California. Currently, SEGA consists of 138 wireless nodes and is planned to expand to a total of 154 nodes at 21 sites in the coming years. As a genetics-based climate change research platform, SEGA allows scientists to quantify the ecological and evolutionary responses of species to changing climate conditions. Multiple long-term and large-scale scientific experiments are conducted at SEGA sites.

The SEGA nodes use a multi-processor architecture. A central processor provides services, including scheduling and dispatch of tasks, storage, and a message-passing interface for wireless networking. Plug-in satellite processors handle transducer sampling, actuation, and related computational tasks. In addition to allowing true parallelism, this architecture enables hardware-level improvements in energy efficiency, since each satellite can be optimized for its specific task. More practically, it admits the rapid implementation of highly heterogeneous nodes that incorporate a wide range of sensing and actuation capabilities. The nodes synchronously interact with neighbors in a multi-hop, self-organizing/healing network; synchronization is implemented as scheduled rendezvous in time slots; slot boundaries are managed by a lightweight global time synchronization protocol that is integrated with low-level communication synchronization. The nodes use a custom time-triggered RTOS tightly integrated with a time/frequency-hopped PHY/MAC protocol. This approach minimizes communication energy cost, which dominates the overall energy consumption.

Problem: Because timing is critical and is determined by the embedded system hardware and software, most testing has occurred at the network level, with extensive in-lab testing with small networks and instrumented field tests. However, it has been found in long-term deployments that occasionally the networking fails and nodes become isolated—we think due to a complex set of subtle bugs rooted in different levels of timing abstraction. When such a failure occurs, it often spreads from one node to others, causing nodes to seek to rejoin and expend high levels of energy for radio operation and eventually deplete their energy sources. Eventually, subnets, or sometimes the entire site, are disabled and humans must visit the site to reboot it. Such failures could affect or even destroy (e.g., via over-watering), long-running scientific experiments.

Since access to SEGA installations can be difficult, and in the long run many may be located so remotely that it is cost-prohibitive to send humans to address problems, discovering the source of these in-operation faults, identifying other faults, and generally improving the reliability of the system is critical. We therefore aim to use SEGA (in particular the protocol in question and its implementation) as our primary case study. This will enable us to apply our approach in a practical setting, and ensure that what we produce is actually usable by engineers of real systems. SEGA is an ideal case study for several reasons. First, the above mentioned network problem enables exploring how to design, prove, and test time-critical systems in a way that does no harm: human life is not affected in this application, and data is not lost since all sensed information is logged as a local back-up. On the other hand, reliable operation is important, and failure costly. Finally, this application uses common data structures for task control blocks, and the operating system at each node schedules and dispatches both periodic and pseudo-randomly scheduled tasks. Thus the system is representative of general applications of scheduling and synchronization in time-critical systems. The embedded code is written in C, enabling the use of both FRAMA-C and DeepState. SEGA will help us understand how our theory and tools can improve the correctness, reliability, and safety of cyber-physical and IoT applications.

Plan: First, we will model the protocol itself as a timed automata in UPPAAL or PRISM, in order to ensure that there is not a subtle flaw in the protocol itself, and to model our expectations of behavior in the real system. Then, following our proposed workflow, we will automatically annotate the implementation with specifications extracted from the timed automata model and attempt to prove components of the code faithfully

represent the intended behavior. Either of these steps may expose the source of the mysterious networking failures. Whether at this point the current problem is identified or not, we will finally use DeepState, driven by harnesses automatically generated by our tools, to generate tests of the implementation components in question. Even if FRAMA-C is able to prove most individual functions correct, which we believe is unlikely, the DeepState testing may expose faults that are not part of the specification. The above workflow will be conducted by an Embedded System Engineering student, who is familiar with the SEGA IoT system but does not have expertise in software verification and testing, using the software tools developed in this project. Feedback from the engineer in this case study will inform us how to develop and improve the theory and tools for practical usages by non-expert users in real applications.

3.7 Case Study: Distributed Coordination in Multi-Robot Systems

Overview: Coordinated operation of multiple autonomous robots (multi-robot systems) has many important real-world applications [17, 90]. For example, in a rescue, security, or disaster response mission, several autonomous aerial robots can coordinate to survey an area, monitor some target objects or activities, and guide ground robots or vehicles. In such applications, each robot is autonomous but has the capability to coordinate efficiently and safely with other robots to complete a shared mission, often in a distributed manner without any central coordinator. Such coordination is essential in real-world applications where the environment is constantly and unexpectedly changing, but is also very challenging. The Intelligent Control Systems (ICONS) Lab at NAU, directed by co-PI Nghiem, is developing distributed control and planning methods for multi-robot systems on a platform that includes quadcopters and four fully autonomous vehicles. One of the most critical challenges of this research the guarantee of the safety, of a coordination plan, which is typically implemented in C code on the embedded computers of the robots and usually involves wireless inter-robot communication, sensing, and actuation.

Problem: Validation of a distributed coordination method for a multi-robot system is currently performed using a mix of theoretical proof (for limited settings), extensive computer-based simulations, simulation-based falsification techniques, and real experiments. Even when a method is validated by proofs and/or simulations, it often fails in real experiments due to discrepancies between models and reality/implementation. The methods and tools proposed in this project will help control and robotics researchers, who usually do not have expertise in software verification and testing, overcome this challenge.

Plan: First, we will model a coordination plan/algorithm for multiple robots as a (potentially very complex) network of timed automata. Performance specifications will be expressed in temporal logics, e.g., the Signal Temporal Logic (STL) [30], and checked against the model using verification and testing tools such as UPPAAL or S-TaLiRo [7]. This step ensures that the original coordination algorithm has no subtle flaws. An implementation of the algorithm in C code, distributed among the robots, will be developed by a robotics/control student. The implementation will then be automatically annotated with a specification using our tools. Next, we will attempt to prove that components of the code faithfully represent the coordination algorithm. Given the complexity of multi-robot coordination algorithms, we do not expect FRAMA-C to be able to prove all the components of the implementation correct. We will thus use DeepState harnesses automatically generated by our tools to generate tests of the implementation components not yet verified by FRAMA-C. Finally, we will perform experiments with aerial and ground robots in the ICONS Lab. The very different nature and complexity of this study, compared to SEGA, will ensure that our methods and tools work in a variety of kinds of real systems.

3.8 Contributions to Formal Methods and the Field

The contributions to formal methods proposed include:

- Fundamental contributions to integrating formal specification languages developed for use in static analysis and theorem proving with dynamic analysis, producing a common semantics for formal, static, and dynamic checking of correctness.

- Enhanced ability of fuzzing and other test generation methods to make use of information from formal specifications, and integrate feedback about, e.g., specification coverage into test generation heuristics.
- Common semantics and a framework for fuzzing, symbolic execution, SAT/SMT-based bounded model checking, and explicit-state model checking.
- Approaches to using feedback from fuzzing to guide bounded or explicit-state model checking, and vice-versa.
- Translations from implementation-level specification to (probabilistic) timed automata models.

The contributions to the field include:

- New development and design methods that focus on implementation-level specification of correctness of code as a guiding principle for embedded systems.
- Tactics and strategies for incorporating the above methods into legacy efforts, where existing code bases require additional specification and annotation.
- Best-practices for using formal, static, and dynamic tools in debugging known, poorly-understood problems in legacy systems.

3.8.1 Evaluation Approach

4 Related Work

A fundamental goal of this project is to reduce both user effort and the opportunity for user effort by allowing minimizing (ideally to one) the number of times a user must specify an aspect of system correctness. The principle that important information should have a "single point of truth" is widely accepted in software engineering, even in such foundational early advances as avoiding repeated magic numbers by the use of named constants. Such a principle can be extended to specification and definition of test harnesses. Early work emphasizing this goal of both reducing work and chance of error in specification and test generation included the effort by Groce and Joshi to use a single harness for both model-checking and random testing, in the verification of the Mars Science Laboratory's file system [39, 41, 45]. In later work, Groce and Erwig extended this idea to propose development of a single language with a unified semantics for a wide variety of dynamic test generation tools [38]; this approach is essentially realized in the DeepState [35] system. Indeed, FRAMA-C and ACSL [9] and DeepState are both arguably limited instantiations of this goal: providing a single language, interface, and semantics that is applied to a variety of methods (static or dynamic) for checking that a specification holds. This project aims to further extend this goal by extending it to include a formal timed-automata model and to connect the primarily static and dynamic approaches.

There are several work on the implementation and verification of distributed systems, and code extraction from automata modeling in the proof assistant COQ [86, 88, 71] Systems developed in this way are executable thanks to COQ's extraction mechanism (to OCaml [67] code). While being very interesting, these proposals require that the developers master COQ, and start from the modeling activity to generate code. They are therefore not applicable in the context of the verification of legacy embedded C code.

Testing real-time systems modeled by networks of timed automata was investigated by the authors of the tool UPPAAL [52, 51, 61] and implemented in the tools UPPAAL-TRON (`http://people.cs.aau.dk/~marius/tron/index.html`) and UPPAAL-COVER (`http://www.hessel.nu/CoVer/index.php`). These tools generate tests, either offline or online, for conformance testing of a real-time system with respect to its model and an environment model, both as timed automaton networks. In both cases, the real-time system is considered a black-box with an input/output interface through which the test generator or monitor can change the system inputs and observe the system outputs. The actual implementation code is not considered and is in fact hidden from the testing tools. While this approach is general, it has several drawbacks. It requires a centralized input/output interface accessible to the testing tools. Such an interface is not always available in all systems, especially in large-scale distributed systems like the sensor/actuator networks considered in our case study. Furthermore, by considering only the (timed) input/output behavior of a system, this approach may not be able to test internal system behaviors and therefore miss opportunities for a better test coverage. Finally, regardless of how thorough the tests are, generally they will not be able to provide a formal guarantee

of the conformance between the implementation and the model / specifications. On the contrary, if the system code is available, our proposed approach can potentially provide such a guarantee.

5 Broader Impacts

Improving Software System Reliability: A key element of our approach is to focus on realistically deployable techniques. Part of this effort is concentrated in our internal effort to apply our methods to the SEGA project. However, we also plan to aim for early integration with NASA's FPrime [14, 69] open source flight software architecture and platform; PI Groce is already in discussion with engineers at NASA's Jet Propulsion Laboratory, and engaged in producing tests for the FPrime autocoder using DeepState. This integration will allow our methods to be applied to CubeSat missions (and other flight software systems), leading to improved reliability for low-budget space-based scientific efforts. We expect, in the long run, that our approaches will lead to more reliable and robust development in many embedded and cyberphysical systems domains, and contribute to a more secure and reliable Internet of Things. One key goal of this project is to increase the synergy between formal modeling, heavyweight static analysis, and advanced dynamic analysis using automated test generation tools, and thus adoption of all three methods.

Education and Outreach: The proposed research yields several opportunities for enhancing CS education, recruiting new CS majors, and retaining CS students, particularly members of underrepresented groups. In addition to the activities discussed at length in the Broadening Participation in Computing plan, PI Groce will work with the NAU Student ACM Chapter to present a series of "excursions in testing" that introduce automated testing to students, using DeepState to find bugs in real world code, including code from media player libraries; in advanced meetings, integrating DeepState with FRAMA-C will be demonstrated as well. The work of Guzdial [50] has shown that media computation is a potentially effective way to both recruit and retain female and under-represented minority students in computer science. Groce is also teaching a class on automated testing of embedded systems to graduate and undergraduate students. Co-PI Nghiem is preparing a new graduate-level course on autonomous vehicles, based on the F1/10 platform (`http://f1tenth.org/`), to be offered to EE and CS students at NAU. To prepare students in addressing one of the greatest challenges of autonomous driving, namely safety guarantee, the course will incorporate the methods and tools developed in this project to teach students about safety, verification, and testing. Each year, Loulergue teaches CS451/551 Mechanized Reasoning about Programs to about 40 students. This class is based on FRAMA-C, and Loulergue transitions his research on formal methods into this class. Loulergue has presented many FRAMA-C tutorials in conference such as ISSRE, ACM SAC, FM, SecDev, *etc.*, and will continue to do so.

6 Results From Prior NSF Support

PI Groce: The most relevant prior NSF support for PI Groce 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 with John Regehr of the University of Utah. **Intellectual Merit:** The results of CCF-1217824 included a number of advances to practical automated generation and use of tests, a key focus of this proposal as well. E.g., an approach to creating "quick tests" from a test suite by minimizing each test with respect to its code coverage [44], won the Best Paper award at the 2014 International Conference on Software Testing. CCF-1217824 produced a general set of results focused on making automated random testing usable by practitioners, and using symbolic execution on larger, realistic software. Publications resulting from this grant were numerous [43, 21, 92, 44, 42, 4, 48, 53, 46, 5]. **Broader Impact:** The results of CCF-1217824 have been used in teaching software engineering classes. Work from the project contributed to the discovery of previously unknown faults in important software systems, including LLVM and GCC, and is widely used in compiler testing [62, 60, 29, 63]. Tools and data sets from CCF-1217824 are available via GitHub in multiple repositories and projects (TSTL, Csmith, CReduce, etc.).

Co-PI Flikkema: Flikkema is co-PI on the Southwest Experimental Garden Array (SEGA) funded by an NSF development MRI (DEB-1126840), with a total budget of $2,848,876 from 10/2011 until 9/2017.

Intellectual Merit: SEGA is a facility distributed across a 1615m elevation gradient in Arizona that supports long-term research to increase understanding of and mitigate climate change using knowledge of genetic variation in species of concern. It consists an array of eleven gardens and supporting distributed monitoring and control cyberinfrastructure for the study of gene-by-environment interactions and enabling development of strategies to best manage for future climates. **Broader Impact:** With 9 successfully completed projects to date, SEGA currently supports 11 experiments and has resulted in over 35 publications and 20 conference presentations. SEGA results are available online [1].

Co-PI Loulergue & Co-PI Nghiem: have not received any NSF support.

References

[1] Southwest Experimental Garden Array. https://sega.nau.edu/home.

[2] Google Test. https://github.com/google/googletest, 2008.

[3] honggfuzz. https://github.com/google/honggfuzz, 2010.

[4] 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. ISBN 978-1-4503-4390-9. doi:10.1145/2931037.2931056. URL http://doi.acm.org/10.1145/2931037.2931056.

[5] 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.

[6] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor Comput Sci*, 126(2):183 – 235, 1994. ISSN 0304-3975. doi:https://doi.org/10.1016/0304-3975(94)90010-8.

[7] Yashwanth Annpureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. S-taliro: A tool for temporal logic falsification for hybrid systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 254–257. Springer, 2011.

[8] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: fuzzing with input-to-state correspondence. In *NDSS (Network and Distributed Security Symposium)*, 2019.

[9] Patrick Baudin, Jean C. Filliâtre, Thierry Hubert, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*, February 2011. http://frama-c.cea.fr/acsl.html.

[10] David M Bell, Eric J Ward, A Christopher Oishi, Ram Oren, Paul G Flikkema, and James S Clark. A state-space modeling approach to estimating canopy conductance and associated uncertainties from sap flux density data. *Tree Physiology*, 2015.

[11] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal—a tool suite for automatic verification of real-time systems. In *International Hybrid Systems Workshop*, pages 232–243. Springer, 1995.

[12] Allan Blanchard, Frédéric Loulergue, and Nikolai Kosmatov. Towards Full Proof Automation in Frama-C using Auto-Active Verification. In *Nasa Formal Methods*, LNCS, pages 88–105. Springer, 2019. doi:10.1007/978-3-030-20652-9_6.

[13] Lionel Blatter, Nikolai Kosmatov, Pascale Le Gall, Virgile Prevosto, and Guillaume Petiot. Static and dynamic verification of relational properties on self-composed C code, 2018.

[14] Robert Bocchino, Timothy Canham, Garth Watney, Leonard Reder, and Jeffrey Levison. F prime: An open-source framework for small-scale flight software systems. In *Small Satellite Conference*, 2018.

[15] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as Markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

[16] Patricia Bouyer, François Laroussinie, Nicolas Markey, Joël Ouaknine, and James Worrell. Timed temporal logics. In *Models, Algorithms, Logics and Tools - Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday*, volume 10460 of *LNCS*, pages 211–230. Springer, 2017. doi:10.1007/978-3-319-63121-9_11.

[17] W. Burgard, M. Moors, C. Stachniss, and F. E. Schneider. Coordinated multi-robot exploration. *IEEE Transactions on Robotics*, 21(3):376–386, June 2005. doi:10.1109/TRO.2004.839232.

[18] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[19] Alan Cao. DeepState now supports ensemble fuzzing. https://blog.trailofbits.com/2019/09/03/deepstate-now-supports-ensemble-fuzzing/, August 2019.

[20] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725, 2018.

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

[22] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. EnFuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *USENIX Security Symposium (USENIX Security 19)*, 2019.

[23] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. Grey-box concolic testing on binary code. In *International Conference on Software Engineering*, pages 736–747, 2019.

[24] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming*, pages 268–279, 2000.

[25] J.S. Clark, P.K. Agarwal, 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), July 2011. In press.

[26] Lori A. Clarke and David S. Rosenblum. A historical perspective on runtime assertion checking in software development. *SIGSOFT Softw. Eng. Notes*, 31(3):25–37, May 2006. doi:10.1145/1127878.1127900.

[27] Byron Cook, Kareem Khazem, Daniel Kroening, Serdar Tasiran, Michael Tautschnig, and Mark R Tuttle. Model checking boot code from aws data centers. *Formal Methods in System Design*, pages 1–19, 2020.

[28] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th ACM Symposium on Principles of Programming Languages (POPL 1977)*, pages 238–252. ACM, 1977.

[29] 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.

[30] Alexandre Donzé and Oded Maler. Robust satisfaction of temporal logic over real-valued signals. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 92–106. Springer, 2010.

[31] P.G. Flikkema, K.R. Yamamoto, S. Boegli, C. Porter, and P. Heinrich. Towards cyber-eco systems: Networked sensing, inference and control for distributed ecological experiments. In *2012 IEEE International Conference on Green Computing and Communications (GreenCom)*, pages 372–381, Nov 2012. doi:10.1109/GreenCom.2012.61.

[32] Eric Gamma and Kent Beck. JUnit 5. `http://junit.org/junit5/`.

[33] Souparno Ghosh, David M. Bell, James S Clark, Alan E. Gelfand, and Paul G. Flikkema. Process modeling for soil moisture using sensor network data. *Statistical Methodology*, 17:99–112, 2014.

[34] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Programming Language Design and Implementation*, pages 213–223, 2005.

[35] Peter Goodman and Alex Groce. DeepState: Symbolic unit testing for C and C++. In *NDSS Workshop on Binary Analysis Research*, 2018.

[36] Peter Goodman, Gustavo Greico, and Alex Groce. Tutorial: DeepState: Bringing vulnerability detection tools into the development cycle. In *IEEE Cybersecurity Development Conference (SECDEV)*, 2018.

[37] Alex Groce. Test harness for testfs. `https://github.com/agroce/testfs`, 2018.

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

[39] Alex Groce and Rajeev Joshi. Random testing and model checking: Building a common framework for nondeterministic exploration. In *Workshop on Dynamic Analysis*, pages 22–28, 2008.

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

[41] 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.

[42] Alex Groce, Chaoqiang Zhang, Mohammad Amin Alipour, Eric Eide, Yang Chen, and John Regehr. Help, help, I'm being suppressed! the significance of suppressors in software testing. In *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013*, pages 390–399, 2013.

[43] 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. ISBN 978-1-4503-3210-1. doi:10.1145/2661136.2661157. URL `http://doi.acm.org/10.1145/2661136.2661157`.

[44] 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.

[45] 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.

[46] 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.

[47] Alex Groce, Jervis Pinto, Pooria Azimi, Pranjal Mittal, Josie Holmes, and Kevin Kellar. TSTL: the template scripting testing language. `https://github.com/agroce/tstl`, May 2015.

[48] 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, January 2016.

[49] Alex Groce, Iftekhar Ahmed, Carlos Jensen, Paul E McKenney, and Josie Holmes. How verified (or tested) is my code? falsification-driven verification and testing. *Automated Software Engineering Journal*, 2018. Accepted for publication.

[50] 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. ISBN 1-58113-672-2. doi:10.1145/961511.961542. URL `http://doi.acm.org/10.1145/961511.961542`.

[51] Anders Hessel and Paul Pettersson. CoVer - a real-time test case generation tool. In *19th IFIP International Conference on Testing of Communicating Systems and 7th International Workshop on Formal Approaches to Testing of Software*, 2007.

[52] Anders Hessel, Kim Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing Real-Time systems using UPPAAL. In *Formal Methods and Testing*, pages 77–117. 2008.

[53] 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*, 20(1):57–78, February 2018.

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

[55] Gerard Holzmann, Rajeev Joshi, and Alex Groce. Model driven code checking. *Automated Software Engineering*, 15(3–4):283–297, 2008.

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

[57] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015. doi:10.1007/s00165-014-0326-7.

[58] Daniel Kroening, Edmund M. Clarke, and Flavio Lerda. A tool for checking ANSI-C programs. In

Tools and Algorithms for the Construction and Analysis of Systems, pages 168–176, 2004.

[59] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[60] 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.

[61] Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. Testing real-time embedded software using UPPAAL-TRON: an industrial case study. In *the 5th ACM international conference on Embedded software*, pages 299 – 306. ACM Press New York, NY, USA, September 18–22 2005.

[62] 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.

[63] 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.

[64] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: a Java Modeling Language. In *Formal Underpinnings of Java Workshop (at OOPSLA '98)*, October 1998. `http://www-dse.doc.ic.ac.uk/~sue/oopsla/cfp.html`.

[65] Caroline Lemieux and Koushik Sen. FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In *International Conference on Automated Software Engineering*, pages 475–485, 2018.

[66] David R. MacIver. Hypothesis: Test faster, fix more. `http://hypothesis.works/`, March 2013.

[67] Yaron Minsky. OCaml for the masses. *Commun. ACM*, 54(11):53–58, 2011. doi:10.1145/2018396.2018413.

[68] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 1186–1189. IEEE, 2019. doi:10.1109/ASE.2019.00133. URL `https://doi.org/10.1109/ASE.2019.00133`.

[69] NASA. F prime: A flight-proven, multi-platform, open-source flight software framework. `https://github.com/nasa/fprime`, 2018.

[70] Rickard Nilsson, Shane Auckland, Mark Sumner, and Sanjiv Sahayam. ScalaCheck user guide. `https://github.com/rickynils/scalacheck/blob/master/doc/UserGuide.md`, September 2016.

[71] Christine Paulin-Mohring. Modelisation of timed automata in coq. In *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001, Sendai, Japan, October 29-31, 2001, Proceedings*, volume 2215 of *LNCS*, pages 298–315. Springer, 2001. doi:10.1007/3-540-45500-0_15.

[72] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: application-aware evolutionary fuzzing. In *NDSS (Network and Distributed Security Symposium)*, 2017.

[73] Alastair Reid, Luke Church, Shaked Flur, Sarah de Haas, Maritza Johnson, and Ben Laurie. Towards making formal methods normal: meeting developers where they are, 2020.

[74] Virgile Robles, Nikolai Kosmatov, Virgile Prevosto, Louis Rilling, and Pascale Le Gall. MetAcsl: Specification and verification of high-level properties. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 358–364, Cham, 2019. Springer International Publishing. ISBN 978-3-030-17462-0.

[75] RTCA Special Committee 167. Software considerations in airborne systems and equipment certification. Technical Report DO-178-B, RTCA, Inc., 1992.

[76] Kostya Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *Cybersecurity, Develop-*

ment (SecDev), IEEE, pages 157–157. IEEE, 2016.

[77] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, 2015.

[78] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[79] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Network and Distributed System Security Symposium*, 2016.

[80] Jack Sun, Daniel Fryer, Ashvin Goel, and Angela Demke Brown. Using declarative invariants for protecting file-system integrity. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, page 6. ACM, 2011.

[81] Andreas Tiemeyer, Tom Melham, Daniel Kroening, and John O'Leary. Crest: Hardware formal verification with ansi-c reference specifications, 2019.

[82] Nikolai Tillmann and Jonathan De Halleux. Pex–white box test generation for .NET. In *Tests and Proofs*, pages 134–153, 2008.

[83] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 253–262, 2005.

[84] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests with Unit Meister. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 241–244, 2005.

[85] Trail of Bits. DeepState: A unit test-like interface for fuzzing and symbolic execution. `https://github.com/trailofbits/deepstate`, 2018.

[86] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 357–368, New York, NY, USA, 2015. ACM. doi:10.1145/2737924.2737958.

[87] N. Williams, B. Marre, P. Mouy, and M. Roger. PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In *EDCC*, 2005.

[88] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP)*, pages 154–165, New York, NY, USA, 2016. ACM. doi:10.1145/2854065.2854081.

[89] K. Yamamoto, Y. He, P. Heinrich, A. Orange, B. Ruggeri, H. Wilberger, and P.G. Flikkema. WiSARDNet field-to-desktop: Building a wireless cyberinfrastructure for environmental monitoring. In Charles van Riper III, Brian F. Wakeling, and Thomas D. Sisk, editors, *The Colorado Plateau IV: Shaping Conservation Through Science and Management*, pages 101–108. The University of Arizona Press, 2010.

[90] Zhi Yan, Nicolas Jouandeau, and Arab Ali Cherif. A survey and analysis of multi-robot coordination. *International Journal of Advanced Robotic Systems*, 10(12):399, 2013. doi:10.5772/57313.

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

[92] 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.

[93] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *NDSS (Network and Distributed Security Symposium)*, 2019.