SHF: Small: Combining Formal, Static, and Dynamic Analysis to Verify and Validate Real-World Embedded Systems

1 Problem Statement, Overview, and Objectives

Sophisticated embedded systems can be found in many safety-critical systems, such as automobiles, airplanes, and medical devices. Ensuring the correctness and reliability of these systems is crucial to avoid potentially catastrophic consequences, for which disparate methods and tools have been developed. The **core problem** we aim to address in this proposal is that **use of formal modeling, advanced static analysis, and advanced dynamic analysis for verification and validation, especially on critical embedded systems, is prohibitively difficult and lacks sufficient synergy for cost-effective application**. This is true even of systems built in an academic research context: that is, unless the research is primarily *about* methods for verifying and testing systems, rather than work on an embedded system for its own sake, these methods are hard to apply. 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 automated test generation. A major cause for this difficulty is the lack of synergy between these related efforts, that is the failure of effort in one context to transfer to another context. In short:

- Learning to use a formal modeling language and tool provides help in discovering defects in a high-level model or protocol, but seldom helps with implementation-level bugs not thus modeled.
- Many static analysis tools are primarily "bug detectors" (e.g, Coverity or CodeSonar), whose output is essentially limited to a list of possible problems; devising test inputs to reject false positives is hard.
- More powerful static analysis tools, such as FRAMA-C [1], provide proofs of correctness for limited aspects of a system, and a rich specification and annotation language. However, there is little or no connection between this annotation and either formal modeling or state-of-the-art test generation.
- There are a large variety of automated test generation tools; however, effort spent learning one tool only partially transfers to another tool. Many tools, e.g., AFL [2], focus on finding crashes, and do not leverage other types of specification.

Consider the case of an embedded systems engineer working on a custom, low-energy consumption, communication protocol for use in a network of low-power sensors and actuators. If she builds a formal model of the protocol, she will likely find that this extensive effort provides no help, other than an improved concept of the system, in proving the correctness of her implementation. If she begins instead by building an automated test generation harness she will find that, despite having spent considerable time expressing pre-conditions and post-conditions for various functions in the implementation, the work must be duplicated when she decides to try to formally prove the correctness of core functionality. Had she begun with proofs, again, logically related (or even equivalent) information would have to be re-expressed, in a different language, to perform test generation. Effort spent in using one tool almost never carries over to another approach. There is simply not enough time or energy available to make use of the available technology. Consequently, in practice, no advanced correctness technology may be used at all. Since it is hard to predict which one(s) will have the greatest payoff, or will even work, perhaps it is best to just focus on manual testing.

Proposed Solution: The overarching goal of this project is to develop a **practical approach for verifying and validating real-world embedded systems**. This consists of *novel methods for combining formal, static, and dynamic analysis* as well as an *open-source software implementation* of these methods. In addition, case studies of real-world embedded systems in different application domains will be conducted to evaluate the effectiveness and practicality of the developed methods and tools.

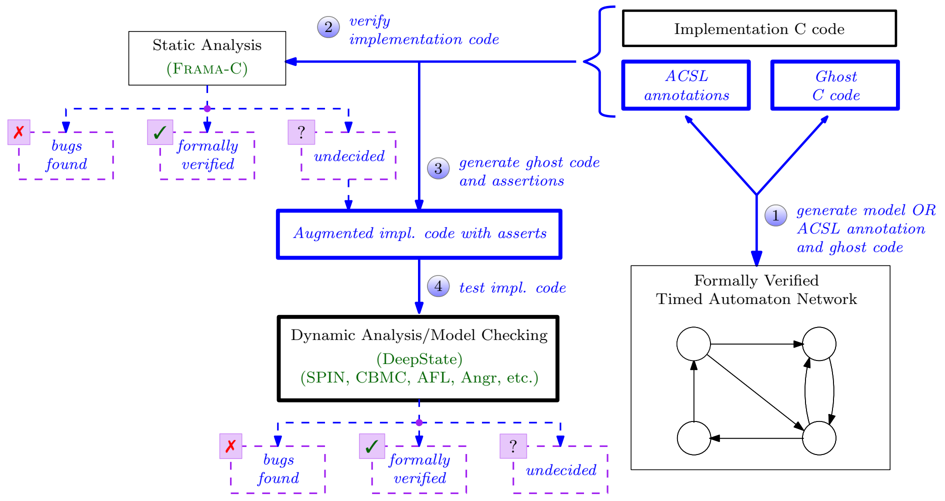
While allowing efforts from any form of formal or automated verification or validation attempt to carry over to other forms is the ideal goal, simply making it possible to follow one critical path to combine methods is feasible and desirable. Which path is most important to realize? Our approach is based in the reality of the embedded systems domain, where, while formal modeling is sometimes used, there is always an implementation. The most basic obstacle to the adoption of formal methods in embedded systems is that if there is only the usual informal design effort or the adaptation of a legacy implementation, formal methods are often inapplicable. By focusing on adding annotations to implementation code, and exploiting those annotations to enable analyses, we promise to always give embedded systems engineers a reasonable payoff.

This project 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, fuzzing, and 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 expect that these annotations can also be imported from a timed automata representation, but we begin where most embedded systems developers are, now, not where we hope they may be, someday. We additionally focus on using frameworks/front-ends allowing application of multiple approaches. Our commitment is to enabling **a maximum diversity of analysis methods** with **a minimum of specification and tool-learning effort**, to **make formal methods attractive in terms of cost-benefit ratio**.

This project is specifically focused on embedded and networked systems, but we anticipate that our solution will generalize to other application domains with similar characteristics and challenges. We expect developers to learn new tools, but not new programming paradigms or languages. The proposed contribution to embedded systems design is not a radical reworking of development methods, which, like many formal methods efforts in the past, would be unlikely to achieve widespread adoption, but the introduction of *an advanced form of unit testing* that works with, e.g., legacy code, with more powerful methods for specification and checking of correctness. This will modify development, in that design-for-testability and design-for-verifiability will become second nature. This project is therefore 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 comprehensive approach with:**
   (a) **an *implementation-focused common framework* for applying methods and tools, and**
   (b) **a focus on *practically-inspired* improvements to methods and tools.**

Case study applications: The proposed research is motivated by applications in embedded and networked systems, in particular wireless sensor networks and multi-robot systems. A wireless sensor network (WSN) consists of multiple, often many, sensor nodes that communicate via a wireless network with servers and/or other sensor nodes. In a WSN, each sensor monitors some physical quantities, such as air temperature, humidity, and CO_2 concentration level, and exchanges its measurement data with other entities for specific objectives, such as for monitoring and forecasting wildfires. Each sensor node typically has some limited computing power (an embedded microprocessor) and limited energy provided by a battery and/or a renewable energy source. Furthermore, wireless communication in a WSN is often subject to frequent packet drops and other failures. Consequently, some of the major challenges for the reliability and correctness of WSNs include timing and communication uncertainty, and computing and energy constraints. In addition, testing and verifying code for sensor nodes faces another significant hurdle due to the nature of interacting directly with the physical world, which involves another level of uncertainty and is not always easily replicable in a software testing or verification tool. A multi-robot system (MRS) consists of multiple robots, possibly of different types such as terrestrial robots and aerial drones, which coordinate to perform certain tasks, such as package delivery, disaster rescue, and surveillance. An MRS carries similar major challenges as a WSN. Due to the complexity of their dynamics and interactions with the physical world, testing and verifying software for robots, especially cooperative robots in an MRS, is particularly challenging. In summary, this project focuses on three major challenges of these representative applications of embedded and networked systems: (1) timing and communication uncertainty, (2) complex dynamic behaviors, and (3) interactions with the physical world. Real-world case studies will be used for validating and demonstrating our proposed approach.



Static Analysis
(Frama-C)
2 verify implementation code
Implementation C code
ACSL annotations
Ghost C code
✗ bugs found
✓ formally verified
? undecided
3 generate ghost code and assertions
1 generate model OR ACSL annotation and ghost code
Augmented impl. code with asserts
4 test impl. code
Formally Verified Timed Automaton Network
Dynamic Analysis/Model Checking
(DeepState)
(SPIN, CBMC, AFL, Angr, etc.)
✗ bugs found
✓ formally verified
? undecided

Figure 1: Overview of the proposed methods and software tools.

1.1 PI Qualifications

See the collaboration plan for an extensive examination of PI Qualifications. In brief, PI Groce has extensive history with formal methods and testing tool development, and practical application to real-world embedded systems. Co-PI Nghiem is an expert in control and autonomy for robotics, including use of formal methods, and has extensive experience with deployed real-world embedded systems and networks.

1.2 Intellectual Merit

The aim of this proposal is to (1) identify a set of principles for the analysis (formal, static, and dynamic) of implementations of embedded and networked systems; (2) match these theoretical principles with tools usable by engineers developing such systems; (3) enable the synergistic use of enhanced versions of these tools in real applications through a common framework with minimal duplication of effort and maximal extraction of information from shared annotations; and (4) implement the developed methods and framework in a practical, user-friendly software tool. These components constitute the proposed framework for a method which will be validated and demonstrated in two case studies. In the first case study, we will analyze networks of wireless sensor nodes deployed in the Distributed Sensing & Computing Over Sparse Environments (DISCOVER) Platform – an NSF CCRI project that develops a large-scale and diverse testbed for wireless sensor networks and multi-robot systems. The second case study will use the framework to formally verify and dynamically test implementations of autonomous control and coordination of multiple autonomous terrestrial and aerial robots on the DISCOVER platform.

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 they are not the focus of this project, code to apply advanced static analysis or timed automata (TA) model skeletons can also be automatically generated:

1. A (generated) TA model can be used to check high-level properties of the design, ignoring many low-level implementation details. However, this step is often skipped in practice.
2. The implementation code with the ACSL annotations and ghost code can be checked by a static analysis tool, such as FRAMA-C. In some cases this will verify the code, and in other cases a definite bug will be found; but often the result will be "undecided" and further analysis required to see if a bug is spurious or real. Again, this step can be skipped, though it is likely low-cost and beneficial.
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 assertions, and needed test-harnesses are *automatically generated* from annotated code.

4. The augmented implementation code is then analyzed using the DeepState [3] framework, which serves as a front-end to highly scalable fuzzers, as well as to symbolic execution tools and model checkers.

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 model checking aspects of the approach. We believe these approaches are currently difficult to apply, but likely to dramatically improve the ability to detect bugs.

Principles: Assuming that an algorithm for an embedded and networked system, such as a communication protocol, can in theory be described as (probabilistic) timed automata [4], which satisfies temporal logic formulas [5], and implemented as a set of imperative programs, 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 skeleton of a timed automata model from annotated programs, in order to facilitate 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. Any effort to increase the adoption of formal methods and automated test generation approaches is likely to be successful only to the extent that it enters embedded systems engineering practice via this existing pathway. Engineers at Google have referred to this approach as "meeting developers where they are" [6].

Tools: We will focus on C code, using DeepState [3] as a front-end for dynamic and implementation-level model checking approaches, and UPPAAL [7] and PRISM [8] for the analysis of protocols; FRAMA-C will provide a powerful static analysis framework, and we will adopt its 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, interrupts, and uncertainty; (2) how to assign the same meaning to a specification construct in various contexts, ranging from fuzzing to symbolic execution to explicit-state model checking to bounded model checking in the "dynamic" DeepState world, and including a static context for FRAMA-C and a modeling context 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 and store state in explicit-state model checking; (4) how to handle intra-program parallelism; (5) how to ensure that the methods are sufficiently automatic and behave in ways engineers will expect. Our focus will be on *practical* solutions, guided by domain experts, rather than on purely theoretical approaches that do not scale. Practical solutions here require fundamental contributions to system and specification design and dynamic analysis and model checking technologies.

Applications: This project will contribute significantly to the application domains of embedded and networked systems, especially in safety-critical applications where correctness and reliability are vital. The methods and tools developed by our team will enable domain experts without formal training in formal methods and software testing to test and verify their code to not only reduce bugs but also gain confidence in their systems.

2 Background and Preliminary Research

A Foundation for Implementation-Level Specification: Ideally, the development of critical embedded systems should rely on a combination of formal methods to achieve an appropriate degree of guarantee:

automatic static analysis to ensure the absence of some 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 that generated *warnings* from static analysis requiring confirmation of a problem or refutation of the feasibility of an error. A core question is how to represent a specification that is intimately tied to real implementation code.

This project will therefore make use of the ideas developed in the ongoing work on the FRAMA-C [1] tool as a foundation for a specification and annotation language. FRAMA-C is a widely-used source code analysis platform that aims at enabling verification of industrial-size programs, and supports combinations of different approaches by providing its users with a collection of *plugins* for analyses. 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 [10]: ACSL allows users to specify functional properties of programs through pre/post-conditions. Many built-in predicates and logic functions are provided, to handle, for example, pointer validity and separation. Using ACSL/FRAMA-C means that while focusing on dynamic analysis and model checking, our approach automatically provides an additional benefit for embedded systems engineers: access to the current set of powerful FRAMA-C static analyses. FRAMA-C provides both abstract interpretation [11] based analysis plugins and deductive verification plugins based on a weakest precondition calculus; the latter have recently been improved to make proof without interacting with a theorem prover easier for engineers [12].

FRAMA-C was designed as a static analysis platform, but has been extended with limited plugins for dynamic analysis. One of these plugins is E-ACSL, which supports runtime assertion checking [13]. 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 plugin E-ACSL is used to translate a subset of FRAMA-C assertions into executable C code. However, the E-ACSL plugin does not support all the specification constructs we need, or assist developers in the most difficult part of dynamic analysis: constructing a set of tests that exercise the checks. The only assistance provided by FRAMA-C for this is very limited in capability. Rather than "re-inventing the wheel" and offering a solution that lacks a strong static aspect we therefore extend ACSL and E-ACSL.

Dynamic Analysis 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 [14], that aims to produce a unit test for a single function, using concolic testing (dynamic symbolic execution [15]). 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. Only actual concrete inputs can be executed on real hardware, and satisfy regulatory requirements on code coverage such as those imposed on civilian avionics by DO-178B, etc. [16]. Furthermore, only testing can prove faults are not spurious, the result of imprecise abstraction or weak assumptions.

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 [2]. 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 [3, 17] addresses these problems. First, developers *do*, usually, know how to use unit testing frameworks, such as JUnit [18] or Google Test [19]. DeepState makes it possible to write parameterized unit tests [20] in a GoogleTest-like framework, and automatically produce tests using symbolic execution tools [21, 22, 23, 24], or fuzzers like AFL [2] or libFuzzer [25] (as well as Eclipser [26], Angora [27], and Honggfuzz [28]). DeepState targets the same space as property-based testing tools such as QuickCheck [29], ScalaCheck [30], Hypothesis [31], and TSTL [32, 33], but for 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. Most property-based tools only provide random testing, and symbolic execution tools such as Pex [34, 35] or KLEE [36] offer only a single back-end. DeepState has already been used to test (and find bugs in) an ext3-like file system [37, 38] and a widely used compression library, and is being actively explored as a basis for automatic testing for in NASA's open source flight software framework FPrime [39, 40]. 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 the GoogleTest [19] platform. PI Groce is at present the lead developer for DeepState.

3 Research Plan

The core outcome of this project consists of practical methods and a framework for combining formal, static, and dynamic analysis for embedded system software written in C and C++, as well as an open-source software implementation and two real-world case studies. This section will detail these efforts.

3.1 DeepState and Automated Test Generation

Applying DeepState to real embedded systems requires us to meet many challenges:

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 of (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). Similar, but in some ways even more complex, challenges are posed by the ubiquity of *interrupts* in embedded code, a problem addressed by very little previous work in fuzzing [41].
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` failures (invalid tests) and `ASSERT` failures (bugs).
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

```
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(), s2_bv = acquire_s2();
  update_state(s, s1_bv);  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();
  DeepState_Timeout(
    [&]{update_state(s, DeepState_UInt64());},
    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

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 [1]. 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. We propose to let users annotate *sets* of functions to be tested as an API-call-sequence group, extending recent work exploring this concept [42, 43].
5. Finally, DeepState and, in fact, general-purpose fuzzers such as AFL, have, to date, been exclusively (to our knowledge) used in what might be deemed conventional environments. As recently noted, "the tight coupling between hardware and firmware and the diversity found in embedded systems makes it hard to perform dynamic analysis on firmware" and existing mainstream fuzzing tools offer almost no support to embedded developers for simulation and emulation [44].

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

[1]There are some Frama-C plugins related to properties over groups of functions, but none apply to the most general form of state problems easily.

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).

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 POSIX operations [45, 46]. 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 [27] and Eclipser [26], 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. Fuzzing of individual functions or sets of functions is a highly promising area: most fuzzing is applied at the whole-program level, where input generation can simply be too hard. By focusing on a middle-ground between unit testing and whole-program fuzzing–using fuzzer technology to drive property-driven testing–the problem is made tractable. Prioritizing paths that include more than just input validation is an explicit goal of, e.g., AFLFast [47], but it must work with an implicit definition based on path frequencies, while we have access to ground truth. Given the complexity of state validity checks, there may be hard-to-reach—but uninteresting—ways to create invalid input; AFLFast will *prioritize* such paths, while we will (correctly) avoid them.

However, it is possible to be more aggressive with preconditions that flow directly from the test harness to a function. Namely, in a large number of cases, it is possible to *actually produce values satisfying a precondition from a fuzzer-chosen value* rather than simply abort the run. This does not violate the semantics of the SUT; it merely transforms one arbitrary input into another, with a fixed mapping so that the fuzzer can still learn from the pattern of inputs, *as transformed*. Consider the simple case of ranges. If a function begins with `ASSUME ((input > 10) and (input < 256))` where `input` is an integer parameter to the function, and this assumption takes place before any assignments to `input` in the function, then in the case where the fuzzer directly generats a value for `input`, we can replace the assumption with the code:

```
if ((input <= 10) || (input >= 256)) input = (abs(input) % 245) + 11
```

Rather than testing if `input` is in a range, this simply maps values that violate the assumption into valid values, that could have been provided by the fuzzer. In such simple cases, of course, the developer of the test harness could have written `DeepState_IntInRange(11, 245))`but when the values in an assumption are dynamically computed inside the called function and/or involve state values not visible to the test harness, or are simple more complex constraints than a simple range check, this is often impractical and sometimes almost impossible. We propose to provide forms of `ASSUME` that enable automatic mapping of inputs, without developer action (other than using our variants of ASSUME), using a mix of guaranteed transformations such as in the range case and "search-based" approaches that find a nearby value of an input that satisfies an assumption (particularly useful in case of, e.g., parity checks).

As far as we are aware, the problem of mapping inputs (rather than producing inputs) to satisfy a predicate has not been explored in the literature. Note that a "mapping" may in the most general case be a *search*:

while also amenable to a solution like that above, a parity check can be satisfied simply by incrementing a fuzzer input byte until it satisfies a check. This most-general approach may be useful in some cases, such as complex checksums with a small size, where a complete solution cannot be produced. On average the time to produce a 16 bit checksum by brute force search from random bytes, for example, may be an acceptable overhead to fuzzing.

We note that this approach to preconditions is not limited to DeepState and/or embedded systems; a manual transformation of this sort is given as essential advice for users of the Echidna smart-contract fuzzer [48]. However, while Trail of Bits even provides an API for the simple range case, the complexity of correctly using the shift from assumption to mapping is such that it is seldom done for more complex cases.

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 can apply to coverage of data structures, for fuzzing data-driven code such as machine-learning algorithms, where much behavior is implicit—e.g., the route taken through a forest of decision trees. In general we aim to extend the work [47, 49, 50, 51, 52], that prioritizes certain program paths in an intelligent way, by exploiting our extended ACSL/E-ACSL.

Finally, these elements must be tied to the problem of applying fuzzing and related methods in embedded-relevant execution environments. We plan to investigate multiple potential solutions, initially focusing on integrating fuzzing with the HALucinator tool [44] for virtualizing firmware via the Hardware Abstraction Layer, which is likely to add emulator-specific notions of coverage and path relevance.

3.2 Other DeepState Extensions

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). Fuzzing or binary symbolic analysis often has trouble finding exact bit-values; it is well known that, e.g., inverting even non-cryptographic hashes is hard. Translation to SAT or SMT, however, often easily handles such problems.

CBMC, the C Bounded Model Checker [53] 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 [54], in analysis of Amazon Web Services infrastructure [55], and in the analysis of flight software systems at NASA's Jet Propulsion Laboratory [46]. 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 other complex configuration options.

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 CBMC at "no cost."

Moreover, because choosing loop unwinding bounds imposes a serious burden on embedded engineers, we will investigate their automatic determinations. One approach is to instrument fuzzer or symbolic-execution engine generated tests to record iterations of loops, and then use the maximum bound observed. 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 [56] may work. 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 fuzzers by alternative "coverage" 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.

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 [46].

The SPIN model checker [57] offers execution of C code with backtracking [58, 59]. DeepState's `OneOf` construct has a semantics that can be matched with the SPIN nondeterministic choice, which in part inspired the DeepState construct [60, 61]. However, integrating SPIN as a back-end for DeepState is even more challenging than integrating CBMC. With CBMC, the mapping from DeepState to CBMC semantics may be performed by changing included headers so that CBMC-specific constructs have differing implementations (but not semantics); SPIN however executes 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; PI Groce's previous work [60] can serve as a foundation. A more fundamental problem is that while CBMC and DeepState can share a semantics for, e.g., `DeepState_Int64()`, a PROMELA model with a branching factor of, e.g., 2^{64} will not work. Solutions range from using results from fuzzing to choose a limited range, to translating "flat" bit-value selection into a sequence of choices with a larger range but bias towards certain values, to using SPIN to control a seed and deterministically choosing random values [60], a hybrid approach.

Timed Automata Model Skeleton Generation: As noted above, one of our core assumptions is that timed automata can model the underlying protocols in many embedded systems. However, writing timed automata models using UPPAAL [7] and PRISM [8] 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 propose 1) to enable DeepState to generate *traces* of the annotations related to timing that are covered during a run and 2) to build a tool to combine and reconcile these traces into a skeleton model for UPPAAL [7] or PRISM [8] (as has been done to some extent for Java [62]). The structure of code (function locations of DeepState annotations) will be used to form the structure of the model. Additional annotations for, e.g. probabilities, may need to be added if not present in the code annotations, though DeepState already has a primitive support for expressing probabilities that we plan to extend.

3.3 Dissemination of Software

The proposed methods and framework will be incorporated into a usable open source tool. While individual software components will be developed and evaluated in tandem with the research efforts described above, we dedicate a task (Task T3 in Section 3.5) for consolidating all these components in a practical, user-friendly software tool since adoption by real-world users is a primary scientific goal of this project. To maximize impact, the software will be made publicly available using the MIT open source software license. We are committed to making software open source as demonstrated by previously released code [63, 64, 65, 66].

3.4 Case Studies

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. At heart, however, we aim to allow case studies to prioritize our efforts, and are certain that other challenges will arise during these efforts. The studies informing this research are the embedded software of wireless sensor nodes and mobile robots in the Distributed Sensing & Computing Over Sparse Environments (DISCOVER) Platform.

3.4.1 Overview of the DISCOVER Platform

DISCOVER is a cyber-infrastructure testbed for remote, rural, and sparsely populated areas. The project is funded by NSF and led by NAU, whose team includes co-PI Nghiem (co-PI of DISCOVER). DISCOVER consists of a fabric of highly configurable Internet-of-Things (IoT) sensor nodes, autonomous and highly capable terrestrial robots and drones, and a heterogeneous wireless network. DISCOVER sites will be located at the campuses of NAU, Navajo Technical University, and Clemsom University, as well as several remote sites. The platform will enable focused research in many domains, including data science and machine learning, heterogeneous networked services, distributed computing and AI, control, autonomous robots, and in-network computation, among many others. We will use DISCOVER for the case studies in this project.

3.4.2 Case Study 1: Embedded Software of Wireless Sensor Nodes in DISCOVER

Overview and challenges: As a community research platform, DISCOVER will allow users, who are researchers in relevant field domains, to develop software code and experiments for the DISCOVER stationary nodes (i.e., sensor nodes) and mobile nodes (i.e., terrestrial robots and drones). A critical step of the process supported by DISCOVER is automatic verification and testing of the embedded code submitted by users for live experiments. We expect that submitted code is developed by researchers who are not trained in computer science and who do not usually apply best practices in software engineering. We also expect that submitted code will have a wide spectrum of code quality and may be malicious, either by chance or intentionally. Automatic functional testing of user code will therefore be of critical importance for the operation and sustainability of DISCOVER. Another challenge is the fact that the code to be tested and verified is for embedded systems that have highly complex physical dynamics and interactions with the physical world and with other physical systems, and are constrained by limited computing power and energy. In addition, wireless communication in a WSN is often subject to frequent packet drops and other failures. Such physical aspects cannot be easily described in code for the purpose of software verification and testing. Therefore, sophisticated software-based and hardware-in-the-loop simulations are necessary, for which several options will be developed by the DISCOVER team.

Plan: Following our proposed workflow, we will first annotate the implementation with specifications of correctness properties. We will use DeepState, driven by harnesses automatically generated by our tools, to generate tests of the implementation components in question, using fuzzing at first, followed by CBMC and SPIN model checking once prototype back-ends are available. For the purpose of verification and testing, the uncertainty inherent in the physical environment and in wireless communication will be modeled by probabilistic timed automata. The above workflow will be conducted by an Embedded System Engineering student, who is familiar with the DISCOVER wireless sensor nodes 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 usage.

3.4.3 Case Study 2: Distributed Coordination in Multi-Robot Systems

Overview: Coordinated operation of multiple autonomous robots has many important real-world applications [67, 68], e.g., in rescue, security, or disaster response missions. 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. Such coordination is essential in real-world applications where the environment is constantly and unexpectedly changing. One of the most critical challenges of this application is to guarantee 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. The terrestrial robots and aerial drones of the DISCOVER platform will be used for this case study.

Challenge: 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-world tests with robots. Even when a method is validated by proofs and/or simulations, it often fails in real tests 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 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) [69], and checked against the model using verification and testing tools such as UPPAAL or S-TaLiRo [70]. While we do not expect actual user code to be accompanied by formal models, in our case study, this step ensures that the original coordination plan has no subtle flaws, and helps us determine properties that need formulation at the implementation level. An implementation of the algorithm in C code, distributed among the robots, will be developed by a robotics/control student. The implementation will be annotated with a specification in our extended ACSL/E-ACSL. We will then use DeepState harnesses to generate tests of the implementation components using fuzzing, symbolic execution, and both bounded SAT/SMT based and explicit-state model checking. Finally, we will determine if a timed automata skeleton extracted from the implementation code corresponds to and would help create a full specification such as we developed before beginning implementation. The very different nature and complexity of this study, compared to stationary sensor nodes, will ensure that our methods and tools work in a variety of kinds of real systems. To overcome the challenge stemming from the complex physical dynamics and interactions of the robots, we will utilize a sophisticated robot simulation environment, based on the Robot Operating System (ROS) [71], with a rich set of predefined scenarios, developed by the DISCOVER team (specifically by the group of co-PI Nghiem). An interface between the robot simulation software and the tools developed in this project will be created to enable seamless verification and testing of the robotic code.

3.5 Work Plan

The project will be organized into two phases, described by work packages. In the first phase, T4.1 will be conducted along with and inform T1.1 (see Figure 3). In the second phase, the focus will be on the application of tools in T1.2 in tandem with T2. Tasks related to the case studies will help refine the developed tools especially in the final phases of the project.

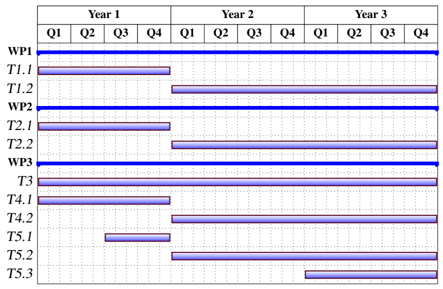


Figure 3: Project schedule.

Work Package 1 (WP1): This work package concerns the development of and use of ACSL and E-ACSL extensions.

• T1.1: This task will consider needed extensions for handling real-world embedded systems. In particular, there will be a focus on a study of the formal semantics of timed automaton networks defined in UPPAAL and PRISM, to determine the extent to which shared semantics can be assigned making it possible to carry implementation annotations into such formal models.

• T1.2: This task will take feedback from applications of tools to generate tests and proofs (T2) into account, to add annotations that are focused on heuristic guidance for tools, not correctness per se.

One Ph.D. student will conduct this work, which will last for the entire duration of the project. Because this aspect is directly tied to ACSL and E-ACSL, and compatibility with Frama-C, we have allocated money for travel to France to meet with Frederic Loulergue, a previous collaborator of the PIs, who has expertise in using Frama-C for Internet of Things applications, and full proof automation in Frama-C.

Evaluation: Evaluation of WP1 will be determined by ability of embedded engineers to agree that the key properties, including those related to timed automata models, to be checked are (1) all representable by the annotations (2) easy to construct (3) easy to read when produced by others and (4) maintainable after introduction.

Work Package 2 (WP2): This work package covers automatically translation of ACSL/E-ACSL-annotated code into a DeepState test harness (Section 3.1), development of back-ends for CBMC and SPIN, and improvements to fuzzers:

- T2.1: This task will optimize the implementation of symbolic execution and fuzzing in DeepState, so that ACSL/E-ACSL annotations and extensions from WP1 can be used effectively.
- T2.2: This task will develop DeepState back-ends for CBMC and SPIN, inform annotations needed to handle loop bounds, memory tracking and matching, and make use of feedback from fuzzing.

The execution of this work package will also span the entire duration of the project. Because the tasks in this package are also based on developing verification and test generation tools (thus formal methods expertise), the same Ph.D. student will work on WP1 and WP2. We separate the WPs primarily to emphasize that specification extensions and tool support are somewhat orthogonal concerns, and evaluated differently.

Evaluation: Evaluation of WP2 will be determined by the application of DeepState harnesses to generate tests for realistic systems. We will use benchmarks and simple examples to some extent, but primarily rely on our connection to case studies. For test generation, we will use coverage and faults detected as measures [72].

Work Package 3 (WP3): This work package will focus on consolidating the software developed in the other work packages in an open-source software tool, usable by embedded software engineers, and on the case studies described in Section 3.4, as a way to inform the methodology and tool developments in the other tasks. WP3 includes the following components:

Open-source software tool. Task T3, that develops the open-source software tool, will span the entire duration of the project, in coordination with the software development tasks in the other work packages. All the students in this project will contribute to task T3.

Wireless sensor network (WSN) case study on DISCOVER. This is divided into two tasks:

- T4.1: In this task, the wireless sensor node systems will be studied thoroughly to extract the key requirements and characteristics of the embedded system implementations. The system information and models resulting from this task will inform the semantics design and method developments in WP1 and WP2. As time allows, we will extend this work to include sensing elements.
- T4.2: This task will apply the tools developed in WP1 and WP2 to the WSN systems, to detect and fix bugs in the embedded software implementations. It will also provide feedback to the other work packages to refine and improve our tools.

Multi-robot system case study on DISCOVER. This study is divided into three tasks:

- T5.1: In this task, a standard multi-robot coordination algorithm will be modeled as a network of timed automata. Using our insights into the robotics application, we will express its performance specifications, particularly its safety requirements, in temporal logics and formally verify or test them in tools like UPPAAL, PRISM, or S-TaLiRo. This task will extend the developed semantics and methods to applications beyond communication protocols, to identify further needed runtime extensions and semantic connections between timed automata theory, implementation annotations, and runtime checks.
- T5.2: This task will apply the tools developed in WP1 and WP2, and the robot simulation environment of the DISCOVER platform, to the coordinated multi-robot system, in order to validate the implementation code, detect and fix possible bugs, and improve the tools developed in this project.
- T5.3: This task will aim to apply the DeepState-trace driven route to produce timed automata skeletons.

As the tasks in this work package are conducted in tandem with WP1 and WP2, to form a feedback loop with the developments in other work packages, it will last for the entire duration of the project. We expect that groups of undergraduate students, in collaboration with an embedded systems Ph.D. student and the Ph.D. students in WP1 and WP2, will perform the work. Close collaboration with the DISCOVER team, led by Co-PI Nghiem, is expected.

Evaluation: In essence, this task is the evaluation aspect of our project, which forms one of the major thrusts of the project. The successful application of WP1 and WP2 tools to the case studies is essentially the driving factor in determining our success in the project. The measure of success is: (1) faults detected and corrected; (2) functionality proven correct using CBMC, symbolic execution engines, or SPIN; (3) coverage and other measures of generated tests; and (4) reported usability and value by engineers, particularly students. For T5.3, evaluation will be based on comparison of extracted skeletons with independently developed full models.

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 [60, 45, 46]. 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 [61]; this approach is essentially realized in the DeepState [3] 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.

The implementation and verification of distributed systems, and code extraction from automata modeling in the proof assistant COQ [73, 74, 75] is a topic of some previous work. Such 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, the common case in the real world.

Testing real-time systems modeled by networks of timed automata was investigated by the authors of the tool UPPAAL [76, 77, 78] and implemented in the tools UPPAAL-TRON and UPPAAL-COVER. 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.

5 Broader Impacts

Improving Software System Reliability: A key element of our approach is to focus on realistically deployable techniques. We aim for early integration with NASA's FPrime [39, 40] 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. Our engagement with interested Galois engineers ensures the applicability of our methods will be as wide as possible, and impact tools outside our initial scope.

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. The work of Guzdial [79] has shown that media computation is an effective way to both recruit and retain female and underrepresented students in computer science. Groce is also teaching a class on automated testing of embedded systems. Co-PI Nghiem has developed a course on autonomous vehicles, based on the F1/10 platform (funded by NSF). To prepare students for addressing safety in autonomous driving, future offerings of the course will incorporate the methods and tools developed in this project.

Broadening Participation in Computing (BPC): The goal of the BPC component of this project is to *increase the number of females who are involved or choose careers in computing, at NAU and in the local community of Flagstaff, Arizona.* Our plan carefully integrates active learning experiences designed for female students at both the undergraduate and middle school/junior high levels. **Undergraduate Education Experience -** We will reach female students in two degree programs at the 2nd-year level: Computer Science and Electrical and Computer Engineering. In CS, we will target CS 200 Introduction to Computer Organization; in ECE, we will target EE 215 Microprocessors. We will integrate a new project in which teams of female (and possibly male, due to the current lack of females in ECE and CS) students imagine and create exciting and meaningful one-day active learning experiences and projects for female student teams in grades 7-9. We will provide full support to these teams, especially female students, and design the project so that female students will take leadership roles to gain confidence. In both courses, we will bring in expert speakers to facilitate development of students' understanding how to design these projects so they are marker events in the students' lives. **Outreach to grades 7-9 -** As noted above, the undergraduate teams will develop active learning and design project "Build Events" for girls in grades 7-9. We will recruit female undergraduates who have taken CS 200/EE 215 to become mentors in the one-day events for the grade 7-9 students. We will schedule these events as part of the annual Flagstaff Festival of Science, and plan them for Saturdays to avoid conflicts with school schedules, maximizing participation. The Flagstaff Festival of Science, now in its 32nd year and enjoying wide financial and participatory support in the community, holds over 100 events for all ages over a 10-day period in the Fall, and is an ideal venue. **Facilities and Support -** By scheduling the grade 7-9 Build Events on Saturdays, we will be able to use the educational laboratories of the School of Informatics, Computing & Cyber Systems (SICCS) for the Flagstaff Festival of Science events. We have requested $2,000 for each in years 2 and 3 for materials (primarily embedded development boards) for these experiences. **Assessment -** We will conduct focused feedback sessions and administer short surveys of the participants to aid continuous improvement of the activities over the course of the project.

6 Results From Prior NSF Support

PI Groce: The most relevant prior NSF support for PI Groce is CCF- CCF-2129446, "Feedback-Driven Mutation Testing for Any Language," with a total budget of $500,000 from 9/2021 until 8/2024, a collaborative proposal with Claire Le Goues of Carnegie Mellon University. **Intellectual Merit:** This project focuses on a synergistic approach for allowing developers to improve testing by using mutation testing to identify weaknesses in tests and to generate tests. Though in its first year, this project has already resulted in three publications [80, 81, 82]. **Broader Impact:** Work from this project has already resulted in the reporting and correction of mulitple bugs in software systems, including production compilers for smart contracts.

Co-PI Nghiem is a co-PI, on the Distributed Sensing & Computing Over Sparse Environments (DISCOVER) Platform funded by an NSF CCRI grant (2120485), with a total budget of $1,366,513 from 10/2021 until 9/2024. **Intellectual Merit:** DISCOVER is a cyberinfrastructure testbed for remote, rural, and sparsely populated areas, consisting of a fabric of highly configurable fixed and mobile sensor nodes and a heterogeneous wireless network. It will enable focused research on new breeds of algorithms that address a range of challenges around prioritization and optimization of computation and communication in remote, less populous and rural areas. **Broader Impact:** DISCOVER will provide a research platform for investigation of distributed computing, networking, security, control and coordination solutions in a heterogeneous configurable cyber-physical system infrastructure that will provide critical services for areas and populations at increasing risk of being underserved. The education and outreach impacts of this project include training and research opportunities for undergraduate students, engaging underrepresented minority students, and developing hands-on research experiments for K-12 students.