SHF: Medium: 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 about methods for verifying systems, not just about the systems themselves). 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 — modeling, static analysis (including proof of correctness for limited components of a system), or dynamic analysis (including automated test case generation). At heart, we believe the 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 [69], or SPIN [63], provides help in discovering defects in a high-level, abstract formulation of a model or protocol, but usually does not translate, in any way, into assistance with implementation-level problems that are 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 [65], provide proofs of correctness for limited aspects of a system, and a rich specification and annotation language. Yet, again, there is no connection between this annotation and either formal modeling or most test generation tools. Use of dynamic analysis with FRAMA-C is limited to its own concolic execution tool, and lacks support for other generation methods such as fuzzing.
- 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 [103]) are specialized to the problem of generating files or packets for use in security vulnerability detection, and provide no support for the kind of testing needed to detect and understand 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, or even knowledge gained in these efforts.

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 part of the actual implementation, even if the property to be proved has a representation in the model. If the engineer begins instead by building a test generation harness, using an automated test generation tool, she again discovers that despite having spent considerable time expressing pre-conditions and post-conditions for various functions in the implementation, in order to guide test generation, this work must be duplicated when she decides to try to formally prove the correctness of certain core functionality. Had she begun with the proofs, again, the logically related (or even equivalent)

information about the code would have to be re-expressed, in a different language, in order to proceed with test case generation. Not only must our engineer learn three tools, but effort spent in using one tool essentially never carries over to another part of the effort. In almost all cases, there is simply not enough time or energy available to make use of the full spectrum of available technology for avoiding defects and ensuring reliability of the system. One approach may be selected, if an engineer is familiar with that method, or, in practice, no advanced correctness technology may be used at all. After all, it is hard to predict whether formal modeling, static analysis, or dynamic analysis will have the greatest payoff, or even work well, and so perhaps it is best to just put somewhat more effort into scenario design and 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. This project proposes to make it possible 1) to transfer efforts to build a formal protocol or system model using timed automata into code annotations making static analysis and proof of implementation correctness possible and 2) to transfer specification and verification effort from advanced static analysis tools to the automated generation of tests using a variety of advanced tools, including symbolic execution tools and highly effective gray-box fuzzers. Enabling this path also enables, indirectly, the use of automated test generation tools on the implementation of a system represented by a timed automata model, even if no effort is made to prove the implementation correct. 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 within those domains. This project is specifically addressed to 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. Additionally, we target the common, hard, case where our approach may be used to guide creation of new code, but will be applied mostly to systems with partial or complete (but not sufficiently reliable and verified) implementations, in typical embedded C code. We expect developers to learn new tools, but not new programming paradigms or even languages.

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. More recently, he has primarily focused on developing algorithms and tools for automated software testing, including front-ends allowing a user to specify a testing problem once and use multiple back-ends; he is a core member of the DeepState [42, 43, 96] design and development team. He brings to this project practical experience using advanced verification and testing tools on real systems, and engineering such tools to be useable by engineers who are domain experts, not verification or testing experts.

Co-PI Loulergue has a long experience in designing parallel programming languages and libraries based on formal semantics. About ten years ago, he started using the COQ proof assistant in his research on programming language semantics and on the development of parallel programs correct by construction. More recently he has begun a collaboration with colleagues at *Commissariat à l'énergie atomique et aux énergies alternatives* (CEA). This line of work focuses on extending FRAMA-C with new features, always starting from needs coming from analysis and verification tasks on real-world code. He brings to this project a strong expertise on static analysis and deductive verification of C programs with the FRAMA-C framework, and on specification and proof engineering on real-world code.

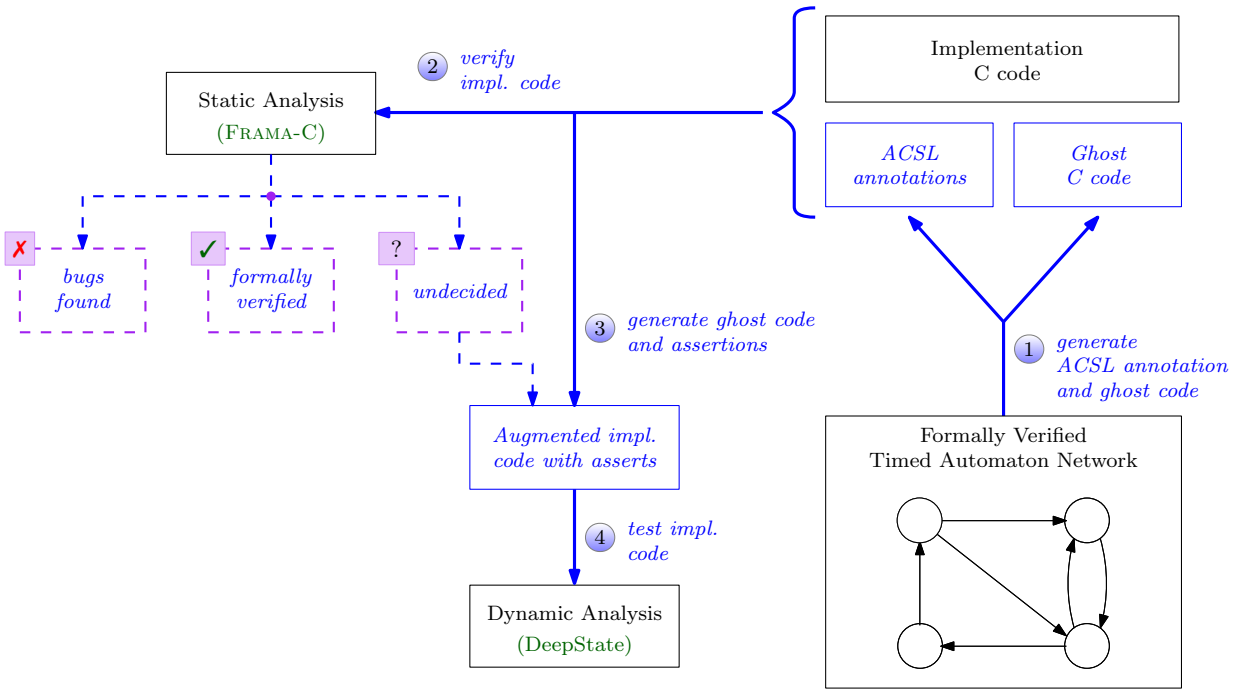

Figure 1: Overview of the proposed research.

Co-PI Nghiem has an extensive background in control theory and application of formal methods in control systems. He has long experience working with timed automata and the UPPAAL tool family. He developed methods for testing and verifying temporal logic specifications of hybrid systems – systems that exhibit both continuous and discrete behaviors – and was granted a U.S. patent for his methods. He brings to this project advanced knowledge about real-time embedded systems and practical experience applying verification methods and tools on those systems.

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*, not verification or testing theory.

1.3 Intellectual Merit

The aim of this proposal is to:

- identify a set of principles for the analysis (formal, static, and dynamic) of communication protocols and their implementations in embedded systems,
- implement these theoretical principles in tools usable by engineers developing such systems,
- use these tools to formally, statically, and dynamically analyze networks of wireless sensor/actuator nodes deployed in the Southwest Experimental Garden Array [102, 38], a distributed facility for examining climatic, genetic, and environmental factors in plant ecology.

Figure 1 shows the overall concept. The core open research problems addressed are represented by two sets of arrows. First, an engineering design, modeled as a network of timed automata (TA), is formally verified with respect to a set of temporal logic specifications. The timed automaton modeling paradigm together with temporal logics for system requirements are rich enough for expressing many practical engineering system designs, including but not limited to communication protocols and supervisory controls (including those relevant to sensor networks and IoT systems). Existing C code that is supposed to implement the design, and

hence satisfy the design specifications, is provided and needs to be verified. Our goal is to certify whether the C code truly implements the complete TA-based design and, if not, find any bugs in the code. This is achieved in a sequence of steps:

1. First, from the TA, ACSL annotations[1] and possibly ghost C code (code that does not contribute to runtime semantics, but assists in proof construction) are automatically generated to augment the implementation code. The annotations together with the ghost code sufficiently describe the semantics of the TA; in other words, if the annotated specification can be verified then the TA design is implemented correctly by the code.
2. The implementation code with the ACSL annotations and ghost code is verified by a static analysis tool, in our case FRAMA-C. There are three possible outcomes:
   verified The implementation can be formally verified, which means that it correctly implements the TA design and therefore meets the design specifications.
   bugs found Bugs are found in the implementation, showing that it can violate the specifications. In this case, the bugs need to be fixed and the process repeated.
   undecided The static analysis tool is unable to prove or disprove correctness of the implementation. In this case, we continue with the next step.
3. In the event of an undecided outcome from the static analysis step, we attempt to refute correctness (or increase our confidence in it) via dynamic analysis — test case generation. Ghost code and runtime assertions as well as a test-harness driver are automatically generated from the annotated implementation code to augment the implementation code.
4. The augmented implementation code is then analyzed using the DeepState [42] framework to apply symbolic execution and graybox fuzzing to generate test cases. These test cases may refute the correctness of the design, or they may fail to do so, leaving us more confident the system is correct.

Principles: Assuming that a communication protocol is described as timed automata [6] or probabilistic timed automata, which satisfy some temporal logic formulas [20], and that it is implemented as a set of imperative programs, the two main questions are:

- Given the timed automata and a set of programs supposed to implement them, how can we annotate the programs to be able to check that they correctly implement the protocol, or to find bugs?
- Given a set of annotated programs, how can we automatically generate a harness (efficiently) representing the test generation problem for that system?

For the first problem, we will consider a toy imperative language with the usual control structures, unbounded integers, addresses, and non-recursive procedures.

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 model semantics and a runtime semantics, without unrealistic burden on engineers.

Tools: In this part we will consider the C programming language, FRAMA-C [65] and DeepState [42] for the analysis of programs, and UPPAAL [2] and PRISM [69]) for the analysis of protocols. The primary open research questions here are numerous, and include:

- How to handle C constructs that are not part of the toy language
- How to extend existing specification languages to support timing and uncertainty

[1] ACSL is the specification language of FRAMA-C.

[2] `http://www.uppaal.org`

- How to assign the same meaning to a specification construct in the static FRAMA-C context and the dynamic DeepState context
- How to handle intra-program parallelism
- 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
- How to ensure that the methods are sufficiently automatic and behave in ways engineers (not 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. In terms of both principles and tools, this project aims to make fundamental contributions to both system design and (specifically) static and dynamic analysis tool design and specification language design and semantics.

2 Background and Preliminary Research

2.1 Static Analysis and Deductive Verification with FRAMA-C

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, and is currently 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, and runtime verification for parts of code that cannot be (or are not yet) proved using deductive verification, or parts of code that contain warnings detected by static analysis that need further analysis to determine whether the warning is an error or not.

This project will use FRAMA-C[3] [65], a widely-used source code analysis platform that aims at conducting 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 their compliance to a common specification language: ACSL [9]. ACSL, for ANSI/ISO C Specification Language, is based on the notion of contract like in JML. It allows users to specify functional properties of programs through pre/post-condition, and provides different ways to define predicates and logic functions. Some useful built-in predicates and logic functions are provided, to handle for example pointer validity or separation.

Value analysis is a program analysis technique that computes a set of possible values for every program variable at each program point. It is based on the *abstract interpretation* technique proposed by Cousot and Cousot in the 1970's [31]. Its main idea is to compute an abstract view of values of variables in the form of *abstract domains*. For example, a usual abstract view for a number value is an interval. Value analysis can be very useful to detect potential runtime errors or prove their absence. Typical examples include invalid pointers, invalid array indices, arithmetic overflows or division by zero. It can also help to prove other properties for which domain-based reasoning can be efficient.

Since the FRAMA-C Aluminium release, FRAMA-C offers a new value analysis plugin EVA (Evolved Value Analysis) [21]. It implements value analysis as a generic extendable analysis parameterized by cooperating abstract domains. Different, highly optimized domains are used to represent integers, floating-point numbers and pointers. EVA is strongly integrated into the FRAMA-C ecosystem and offers a basis for many other derived plugins that reuse the results of EVA (see [65]).

WP is a *deductive verification* plugin provided with FRAMA-C. It is based on 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. A combination of automatic and interactive

[3] See https://frama-c.com

proofs often offers a good trade-off for a complete proof. Indeed, some properties can only be defined recursively, and in this case, SMT solvers often become inefficient, trying to unroll them. By using inductive or axiomatically defined functions, we can prevent this behavior but reasoning about them still requires induction, a task that SMT solvers are not good at. Thus, the last step is generally to state lemmas that can be directly instantiated by SMT solvers.

FRAMA-C was initially designed as a static analysis platform, but it was later extended with plugins for dynamic analysis. One of these plugins is E-ACSL, a runtime verification tool. E-ACSL supports runtime assertion checking [30]. Assertions are very convenient for detecting errors and providing information about their locations. It is the case even when such an error does not result in a failure during execution. 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 C code that can then be executed. In this case the assertions are checked *dynamically*.

2.2 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 [100], that aims to produce a unit test for a single function, using concolic testing (that is, dynamic symbolic execution [41]). In cases where concolic execution fails to scale, PathCrawler will also fail. Furthermore, PathCrawler is tuned to the problem of testing a single function, not producing more complex scenario-based tests of, e.g., an set of API calls to a library, or even a set of multiple functions that must coordinate access to a data structure. Finally, PathCrawler is not an open source, extensible system, but a proprietary system that may be costly to acquire and use, and is obviously difficult 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 [86]. Furthermore, only testing guarantee that faults detected will not be spurious, resulting from imprecise abstraction or weak assumptions. In other words, testing (that is, dynamic analysis including generation of inputs) is both able to detect faults that escape modeling and proof, and the only method guaranteed to produce real faults that actually exist in the system as implemented and executed on top of a real hardware and software substrate, an especially critical concern in embedded and cyberphysical domains.

DeepState [42] is a dynamic analysis tool that aims to provide a single, usable, flexible front-end to a wide variety of back-end systems for test-case generation. Most developers do not know how to use symbolic execution tools; developers seldom even know how to use less challenging tools such as greybox fuzzers, even relatively push-button ones such as AFL [103]. Even those developers whose primary code 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 aims to address this problem in two ways. First, developers *do*, often know how to use unit testing frameworks, such as JUnit [39] or Google Test [2]. DeepState makes it possible to write parameterized unit tests [94] in a GoogleTest-like framework, and automatically produce tests using angr [89, 91, 88], Manticore [80], American Fuzzy Lop [103] (or another file-based fuzzer), or libFuzzer [87]. DeepState also targets the same space as property-based testing tools such as QuickCheck [28], ScalaCheck [83], Hypothesis [77], and TSTL [47, 62], but DeepState's test harnesses look like C/C++ unit tests. The major difference from previous tools is that DeepState aims to provide a front-end that can make use of a growing variety of back-end methods for test generation, including (already) multiple binary analysis engines and fuzzers. 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 [93, 95] or KLEE [22], while similar on the surface in some ways, always have a single back-end for test generation. DeepState's flexibility is evident: in the last few months, DeepState added support for the Eclipser [27], Angora [24], and Honggfuzz [3] fuzzers, as well as an ensemble [26] mode supporting the use of multiple fuzzers at once [23].

In effect, DeepState targets the same space (providing the technology and translation between different semantics neccessary to use different verification/bug-detection technologies) as this proposal, but narrowed to the domain of generating test inputs. Moreover, DeepState goes beyond other parameterized unit testing tools by providing an interface that extends the typical GoogleTest interface with constructs making it easy to generate tests for sets of functions, or entire libraries. DeepState has already been used to test (and find bugs in) a user-mode ext3-like file system developed at the University of Toronto [92, 44]—not to test each individual function, but to test the combination of all file-system operations, with the environment for each function defined as the set of states reachable by calling any of the file-system modifying functions. DeepState additionally provides "smart" state-of-the-art test case reduction [55], using its knowledge of deep test structure to efficiently construct easy-to-understand test cases from the complex inputs often generated by fuzzers. Such reduction has also been shown to be useful in improving the performance of seeded symbolic testing [104]. DeepState is being considered as a basis for automatic testing for components in NASA's open source flight software framework FPrime [16, 81]. DeepState is a fully open source system [96], supported by Trail of Bits (a New York based security analysis company), and is being considered as a future extension to the core GoogleTest framework. Although only released in early 2018, DeepState is already one of the most popular property-based testing and fuzzing projects on GitHub.

3 Research Plan

3.1 From Timed Automata to FRAMA-C

It is notoriously difficult to design correct and secure communication protocols. One of the most famous example is the Needham Schroeder Public Key protocol [82]: it took 18 years to discover a flaw in this protocol [76], and it was done using formal methods. Networks of timed automata are formalisms suitable for the formalization of protocols. They are the basis of model checkers such as UPPAAL [32] and PRISM [69] that have been used successfully in the verification of network protocols [105, 58, 64, 90]. These tools help in finding issues with an abstract formulation of a protocol, but they cannot help with implementation details that are not directly modeled in the formalism.

But the *implementation* details of a correct protocol also matter as the Heartbleed vulnerability in the OpenSSL implementation shows. In the case of Heartbleed, the problem was a C runtime error: an access to an invalid memory region, and it was due to an implicit assumption on the input of a function that was actually false. Combination of static and dynamic analyses can detect such vulnerabilities [66] because they are not due to complex interactions related to the protocol.

The methodology we envision combines the use of model-checkers such as UPPAAL and PRISM for the verification of protocols, with frameworks for static and dynamic analysis of C programs, namely FRAMA-C and DeepState, for the verification of the *implementations* of protocols.

We consider FRAMA-C, and its specification language ACSL pivotal in this approach. The research challenge here is to translate networks of timed automata into ACSL annotations and C ghost code for enabling the verification of the C code implementing the protocol modeled by the timed automata. This problem can further be broken down into the following key-subproblems:

1. The translation of networks of automata into annotations to be used within the FRAMA-C code analyzer.
   Previous work co-authored by co-PI Loulergue on the Contiki [34] lightweight operating system for the Internet of Things showed that various approaches can be applied to the verification of the same code. For checking the correctness of the linked list API of Contiki, it includes the use of ghost arrays [14]. Ghost code is a part of a program that is added for the purpose of specification. Such code should not interfere with regular code. Erasing it should make no observable difference in the program results. This approach made it possible to perform most proofs automatically using the FRAMA-C/WP tool, only a small number of auxiliary lemmas being proved interactively in the COQ proof assistant. This work relied on an elegant segment-based reasoning over the companion array developed for the proof.
   This approach, however, is expressed in parts of the ACSL language that cannot be translated to executable C code, i.e. that do not belong to the E-ACSL subset. In a broader verification context, especially as long as the whole system is not yet formally verified, it is very useful to rely on runtime verification, in particular to test client modules that use the list module. Another work [75] showed a variant of the list module specification that belongs to the executable subset E-ACSL of ACSL and can be transformed into executable C code. A newer approach [15] relies on logic lists: they are part of the ACSL standard library of inductively defined logical data structures. In the case of Contiki, a logic list provides a convenient high-level view of the linked list. The specifications of all functions are now proved faster and almost all automatically, only a small number of auxiliary lemmas and a couple of assertions being proved interactively in COQ.
   We expect several translations of networks of automata to be considered: some may be easier to understand to C programmers not familiar with formal specifications (ghost code), some more efficient for deductive verification (logical data structures), and some suited for dynamic verification while still being amenable to deductive verification.
   Such translations will be implemented as FRAMA-C plugins. The alphabet (events) and states of the automata will need to be associated with respectively specific execution events and memory states of the C programs. We expect to experiment manually with this mapping in case studies before enhancing the plugin to provide support for it.
2. Although deductive verification about *algorithmic complexity* is possible from source code [97, 84, 56], such a formal way is not appropriate for this project, essentially because these approaches deal with complexity rather than execution time, and there is no precise enough translation from one to the other.
   Time constraints will be translated into new ACSL annotations. Then we will rely on static analysis tools for worst-case execution time estimation. There are recent projects [78] that explore taking advantage of semantics information to improve WCET estimation, as well as preliminary work showing the benefits of such an approach [19].
   A C program with ACSL annotations very often provides a large variety of semantic information including intervals for variable values, and information about loops such as relations between the number of iterations and other variables. Exploiting this information will require us to be able to modify the WCET tool. This requirement excludes the best WCET estimator, aiT [35], a closed source commercial tool. Heptane [59] and `OTAWA` [8] are two actively developed open source projects with software architectures designed to ease extending the tools. Heptane is focused on cache analysis while `OTAWA` supports more processor architectures. For our case study, `OTAWA` seems the best alternative for verifying if the bound obtained by

WCET estimation on the code indeed satisfies the time constraints obtained for the timed automata.

3. The main part of the code where the specification of the automata will be used should be a kind of event loop. However this loop may be incomplete in the sense that it may not consider all the possible events, or even may not be structured as an event loop in the case most events are handled through interrupts. Although FRAMA-C does not directly handle concurrency, its CONC2SEQ [12] plugin allows for the analysis of parallel compositions of C programs through program transformation to sequential C programs [13]. The main part of the simulating program is a loop that handles control switch among the various threads. This loop can be used as the main event loop in a concurrency context.

3.2 From FRAMA-C to DeepState

While FRAMA-C allows, ideally, for proof of correctness of annotated code, in many real-world instances it will be impossible to prove correctness, either because the proof is too hard to construct or because the code is not in fact correct. While FRAMA-C provides some mechanisms for generating possible counterexamples to a proof, and limited test case generation, it is far from ideal in this setting. A full workflow for verification of realistic systems, therefore, requires a first-class *dynamic* analysis component. Furthermore, such a component should not limit itself to a single method for generating test cases, as with tools such as KLEE [22], Pex [93], or the test generation tools provided by FRAMA-C [18]. Predicting which test generation methods will discover a fault in code, or simply scale to provide effective exploration of code paths, is notoriously difficult. Furthermore, most tools are research prototypes and have known bugs that prevent application to some subset of programs. We therefore aim to provide a *flexible* dynamic front-end that provides a one-stop solution to the problem of dynamic analysis, either to provide confidence in code that cannot be proven correct, or to discover a counterexample showing that the code is not correct. DeepState [42] provides such a front-end.

The research challenge is to translate ACSL-annotated code for use in FRAMA-C into a full-featured DeepState test harness. This problem can further be broken down into four key-subproblems:

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, because DeepState provides more back-ends, some executable specifications may need to be modified to be efficiently handled when the DeepState back-end is a symbolic execution tool. Additionally, it is important to instrument the executable code that support specifications in order to make the coverage of the specification itself visible to fuzzer back-ends, such as libFuzzer. Moreover, DeepState's nature as a test generation tool means that it support constructs, such as ForAll, Minimum, and Maximum, that are not normally 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, including nondeterministic expression of the timing constraints on calls, and introducing ghost branches that make timing impacts visible to coverage-driven fuzzers.
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. Additionally, the same optimizations and visibility-to-fuzzer improvements as for the specification must be provided.
3. The inputs to a function must be translated into code controlling the input values that DeepState must generate in a test, including ranges and types. When input types are simple, this process is straightforward; however, when functions take, e.g., arbitrarily sized arrays or linked lists, or other complex structures, this becomes a problem of constructing a test harness that (1) makes fuzzing and symbolic execution scalable

```
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

but (2) allows 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.

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 this is no longer true 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 described by the precondition for a function may be too large to explore with a fuzzer or symbolic execution tool. In such cases, exploring the space described by valid calls of other functions has two benefits: first, the space described by a sequence of calls may be much smaller, and easier to explore, than the full set of possible input values to a function. Second, errors in this portion of the input space are more clearly realistic scenarios. 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 never appear in practice. 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. 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 approach to the problem of handling pre-conditions/assumption semantics, and second, an investigation of how to let fuzzers take advantage of the much greater structure involved in property-based testing than in traditional unstructured fuzzing; this includes specification structure, so is inherently tied to the first problem. Consider the code in Figure 2. This defines two different DeepState tests of (largely omitted) software that reads sensor values and incorporates them into a system state structure. The two tests check two different properties: `UpdateNeverSlow` ensures that no matter what valid inputs are given to it, updating the sensor is never too slow. The property 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, independent, test, `AvoidCrashes` simply starts the system up in some valid state, and repeatedly either reads both sensors or reads only sensor one. There is no explicit property here, only the expectation that the system will not crash; tests can be executed using, e.g. LLVM sanitizers to further check for integer overflow, other undefined behavior, and so forth. Generating such harnesses automatically from ACSL specifications is a significant challenge, but our research agenda also includes solving problems that would appear even if the harness were generated by hand. First, 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 but without indicating that it failed to the fuzzer. 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 a failed test. We aim to synthesize code to make assumptions automatically take on the proper semantics during test execution, without the user having to redefine the behavior. The solution must also encode the difference between a path to search for and a constraint to satisfy for symbolic execution of tests.

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, with respect to fuzzing. By default, 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 the `acquire` functions. However, from the point of view of testing, this is not ideal. It is often possible to enumerate a vast number of input paths that only define invalid inputs, and so produce very little real testing despite a major computational effort. 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 [49, 53]. DeepState knows which branches are pre-conditions, and so can help a fuzzer 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 [24] and Eclipser [27] that perform lightweight constraint-solving to cover branches, the process can be even 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 validity check) should improve Eclipser's performance in property-based fuzzing. Devising effective heuristics to tune both "classic" mutation fuzzers and concolic gray-box fuzzers promises to improve test generation not only in the context of our workflow, 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, and faults detected during unit development, not after a working program is available. Prioritizing paths that are not input validation is an explicit goal of, e.g., AFLFast [17], but it must work with an implicit definition based on path frequences, 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 properly 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 bitvectors 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 [17], FairFuzz [74], VUzzer [85], and other efforts [106, 7], 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.

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

The case study informing this research is the embedded software used in large collection of operational wireless sensor/actuator networks for monitoring and control of ecological systems [29, 40, 10]. The 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 heterogenous nodes that incorporate a wide range of sensing, actuation, and sensing+actuation capabilities. Our implementation of the architecture emphasizes energy efficiency [36, 37]. For example, all satellite processors are power-gated via central processor control; ensuring that satellite processors are depowered prevents satellite sleep-mode energy leakage. The power subsystem provides multiple power buses at different voltages, including an optically-isolated high-power bus for actuation. A variety of energy supplies are also supported, including battery-backed photovoltaic sources [38, 68].

The nodes synchronously interact with neighbors in a multi-hop, self-organizing/healing network; communication synchronization is implemented as scheduled rendezvous in time slots; slot boundaries are in turn managed by a lightweight global time synchronization protocol that is integrated with low-level communication synchronization. This approach, implemented using a time-triggered architecture, minimizes communication energy cost, which dominates the overall energy consumption in these applications. 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, we have found in our long-term deployments (at dozens of field sites over years of operation) that very 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.

Because 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. We therefore aim to use SEGA (and in particular the protocol in question and its implementation) as a case study for our methods. This will enable us to apply our approach in a practical setting, and increase the chances that what we produce is actually usable by engineers of real systems.

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 specification extracted from the specification of the timed automata model, and attempt to prove components of the code faithfully represent the intended behavior. Either of these steps may, of course, 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 (as we believe is unlikely) FRAMA-C is able to prove most individual functions "correct" the DeepState testing may expose faults that are not part of the specification.

For instance, using libFuzzer with DeepState we can use LLVM's Undefined Behavior sanitized to catch some classes of undefined behavior that FRAMA-C does not take into account. Furthermore, FRAMA-C's ability to prove properties about interactions of multiple functions operating in arbitrary sequence is often limited; such proofs are notoriously hard to construct in general. DeepState allows us to hope to detect faults when we cannot prove correctness. DeepState's ability to use symbolic execution as a back-end will be most useful for verifying single functions that are hard to verify with FRAMA-C, while state-of-the-art fuzzers will be most useful for sets of functions, or cases where symbolic execution fails to scale.

This system is ideal as a case study for several reasons. First, the deployed networks "fail" in way that 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 locally at each node as a back-up. On the other hand, reliable operation is important; as noted above, access to manually fix problems can be problematic even with current installations, and in the future this problem will only grow. Second, some failure modes could damage or even destroy (through e.g. over-watering) long-running scientific experiments. Secondly, 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 a good example of the highly general applications of scheduling and synchronization used in myriad time-critical systems. The embedded code is written in C, enabling the use of both FRAMA-C and DeepState. More broadly, it will inform how the developed theory and tools can improve the correctness, reliability, and safety of cyber-physical and IoT applications.

3.4 Work Plan and Evaluation Approach

The proposed work will be organized in three work packages, corresponding to the directions described above. Because in this project evaluation is essentially a core thrust of the project itself, we describe evaluation as part of the work plan.

3.4.1 Work Package 1 (WP1)

This work package will develop the principles and extensions to FRAMA-C for automatic translation of networks of timed automata into ACSL and ghost code, as described in the first research thrust in Section 3.1. WP1 consists of the following major tasks:

- T1.1: This task will study the formal semantics of timed automaton networks defined in UPPAAL and PRISM, and define an appropriate set of the timed automaton semantics considered in this project. The semantics would be rich enough for modeling the application systems considered in the project, specifically in WP3, while simple enough for the translation into annotations usable by static analysis tools (FRAMA-C) and dynamic analysis tools (DeepState). As such, there will be close collaboration and iterative design steps between this task and the other work packages.
- T1.2: This task will develop several methods for automatic translation of timed automaton networks into ACSL annotations and/or ghost code. Depending on the target usage, whether for statis analysis or dynamic analysis or being inspected by programmers, a suitable method will be developed.
- T1.3: In this task, we will implement the methods developed in T1.2 as FRAMA-C plugins. The developed tools will be tested on the case study application in WP3, whose results will inform the tool development process.

One Ph.D. student will be needed to conduct the work in this work package, which will last for the entire duration of the project. The student will collaborate closely with students in WP2 and WP3, and will partially participate in the application in WP3.

Evaluation: Evaluation of Work Package 1 will be determined by the set of timed automata networks that can be translated into ACSL and ghost code, and the ability to use that code to prove properties. We will use benchmark examples (realistic code found on GitHub or in embedded systems textbooks) to some extent, and simple test cases for constructs, but primarily rely on the SEGA protocol and implementation code to ensure that our efforts are successful.

3.4.2 Work Package 2 (WP2)

This work package will develop methods and tools to automatically translate ACSL-annotated code in FRAMA-C into a DeepState test harness, as described in the second research thrust in Section 3.2. WP2 is further divided into the following tasks:

- T2.1: This task will extend the E-ACSL semantics and optimize the implementation of symbolic execution and fuzzing, so that ACSL annotations generated from timed automaton networks (in WP1) can be used in DeepState to test implementation code.
- T2.2: This task will develop methods to translate annotations and ghost code generated for static analysis in WP1 to test specifications in DeepState. It will be carried out concurrently with tasks T1.1 and T1.2 in WP1, and will inform the design and method development in those tasks.
- T2.3: In this task, we will implement the methods developed in T2.1 as extensions to DeepState. The tools will be applied to the case study in WP3, whose results will inform the development of the tools.

The execution of this work package will span the entire duration of the project. One Ph.D. student will work on the tasks in WP2, and collaborate with the students in WP1 and WP3.

Evaluation: Evaluation of Work Package 2 will be determined by the successful generation of DeepState harnesses for ACSL annotated code, and the successful application of those harnesses to generate tests for realistic systems. We will again use benchmarks and simple examples to some extent, but primarily rely on our connection to the large, real-world SEGA project. In the case of test generation, in addition to faults detected, we will use code coverage and other standard benchmarks, including methods proposed for evaluating fuzzers in high quality research [67].

3.4.3 Work Package 3 (WP3)

This work package will focus on the application described in Section 3.3, as both a way to inform the methodology and tool developments in the other work packages and a case study to validate our methods and tools. WP3 is divided into two tasks:

- T3.1: In this task, the existing application system will by studied thoroughly to extract the key requirments and characteristics of the embedded system implementation. Timed automaton models of the communication protocol used in the system, at different levels of abstraction, will be developed and formally verified in UPPAAL and/or PRISM. The system information and models resulted from this task will inform the semantics design and method developments in WP1 and WP2. As time allows, and pending results with communication protocol, we will extend this to include sensing and control elements of SEGA.
- T3.2: This task will apply the tools developed in WP1 and WP2 to the application system, in order to detect and fix bugs in the current implementation that cause the intermittent failures mentioned in Section 3.3. It will also provide feedback to the other work packages to refine and improve the tools developed in this project.

As the tasks in this work package are conducted in tandem with WP1 and WP2, to form an informative feedback loop with the developments in other work packages, it will last for the entire duration of the project. We expect that a group of undergraduate students, in collaboration with the Ph.D. students in WP1 and WP2, will do the tasks in this work package.

Evaluation: In essence, this task *is* the evaluation aspect of our project, which forms one of the three major thrusts of the project. The successful application of WP1 and WP2 tools to SEGA is essentially the driving factor in determining our success in the project, and the key feedback to drive changes to our research priorities or technical choices. The measure of "successful application" is (1) faults detected and corrected (2) functionality proven correct in FRAMA-C and (3) coverage and other standard measures of generated tests.

3.4.4 Timeline

The project will be organized in three phases. In the first phase, T3.1 will be conducted along with T1.1 and T2.1, and will inform the development in these tasks. In the second phase, the focus will be on the

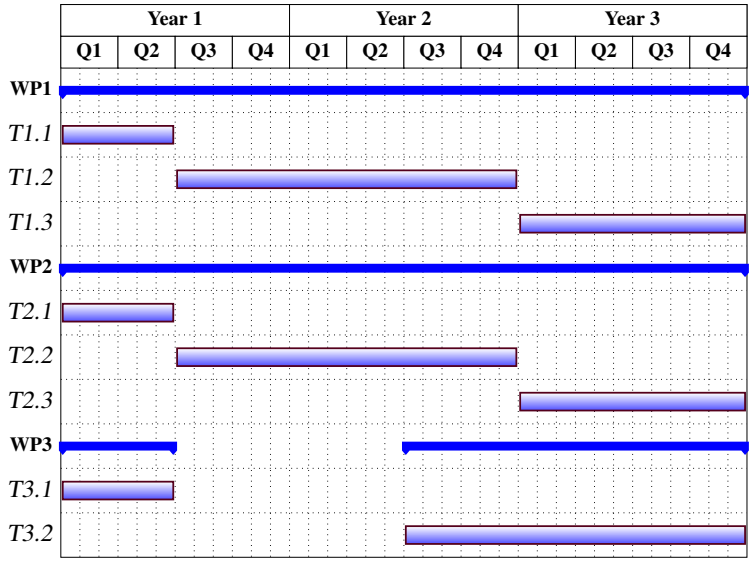

Figure 3: Project schedule.

development of translation methods in T1.2 and T2.2. In the last phase, software tools will be developed (tasks T1.3 and T2.3) and applied to the case study (task T3.2), whose results and feedback will help refine the developed tools. It is important that the tasks of the work packages are carried out in tandem and in close collaboration. The timeline of the tasks will be structured as shown in Figure 3.

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 [46, 49, 53]. 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 [45]; this approach is essentially realized in the DeepState [42] 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 approaches of FRAMA-C and the dynamic approaches of DeepState.

Verdi [99, 101] is a framework for the implementation and verification of distributed systems using COQ. The strength of this framework is to allow users to design and prove the correctness of a system without considering faults, and apply what is called a verification system transformer to modify the system into a system that handles faults. Systems developed in this way are executable thanks to COQ's extraction mechanism (to OCaml [79] code). Verdi could not be applied to our case study: time is not considered by Verdi and although it is possible to port variants of OCaml on micro-controllers [98], there is currently no such variant for the micro-controllers deployed in SEGA. Using a framework such as Verdi would also mean writing new code using a language unfamiliar to SEGA's developers. This proposal instead focuses on the

analysis of legacy code.

Testing real-time systems modeled by networks of timed automata was investigated by the authors of the tool UPPAAL [61, 60, 71] and implemented in the tools UPPAAL-TRON[4] and UPPAAL-COVER[5]. These tools generate tests, either offline or online, for comformance 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, it is possible that our proposed approach using a static analysis tool like FRAMA-C can provide such formal 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 [16, 81] 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 reliabliity 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; currently, engineers in real-world projects seldom use *any* of these approaches. One barrier to entry is that it is seldom clear, in a particular project, which of these approaches will provide the most benefit; formal modeling can detect design flaws, but seldom provides any help with the most frequent source of failure, implementation-level flaws; static analysis either tends to miss bugs, overwhelm users with false positives, or, when extended to proving correctness, simply fail to prove many components; and, finally, dynamic analysis is ad hoc and misses subtle flaws. By increasing the synergy of these methods, we hope to make use of *all three methods* in conjunction much more economical (and less error prone), greatly enhancing the likelyhood of a payoff in terms of fault detection and prevention.

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 [57] has shown that media computation is a potentially effective way to both recruit and retain female and under-represented minority students in computer science.

[4] UPPAAL-TRON: `http://people.cs.aau.dk/~marius/tron/index.html`

[5] UPPAAL-COVER: `http://www.hessel.nu/CoVer/index.php`

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., one question investigated was how to "tame" fuzzer output, in order to more effectively discover unique bugs [25]. A second major result was the development of a strategy for creating "quick tests" from a long-running randomized test suite by minimizing each test with respect to its code coverage [52], which 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, in particular understanding how code coverage and test content interact. Publications resulting from this grant were numerous [51, 25, 104, 52, 50, 4, 48, 62, 54, 5]. **Broader Impact:** The results of CCF-1217824 have been used in teaching software engineering to undergraduates at Northern Arizona University and Oregon State University. Work from the project contributed to the discovery of previously unknown faults in multiple open source and commercial software systems. The further development of the swarm testing techniques the proposal centered on have furthered the effort to improve the quality of compilers, including LLVM and GCC, and to test language tools in general [72, 70, 33, 73]. 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].

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] 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.

[8] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: An open toolbox for adaptive WCET analysis. In *Proceedings of the 8th IFIP WG 10.2 International Conference on Software Technologies for Embedded and Ubiquitous Systems (SEUS)*, pages 35–46, Berlin, Heidelberg, 2010. Springer-Verlag. doi:10.1007/978-3-642-16256-5_6.

[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, Nikolai Kosmatov, Matthieu Lemerre, and Frédéric Loulergue. conc2seq: A Frama-C plugin for verification of parallel compositions of C programs. In *16th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 67–72, Raleigh, NC, USA, 2016. IEEE. doi:10.1109/SCAM.2016.18.

[13] Allan Blanchard, Frédéric Loulergue, and Nikolai Kosmatov. From Concurrent Programs to Simulating Sequential Programs: Correctness of a Transformation. In Alexei Lisitsa, Andrei P. Nemytykh, and Maurizio Proietti, editors, *Proceedings Fifth International Workshop on Verification and Program Transformation, Uppsala, Sweden, 29th April 2017*, volume 253 of *Electronic Proceedings in Theoretical Computer Science*, pages 109–123. Open Publishing Association, 2017. doi:10.4204/EPTCS.253.9.

[14] Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue. Ghosts for Lists: A Critical Module of Contiki Verified in Frama-C. In *Nasa Formal Methods*, number 10811 in LNCS, pages 37–53. Springer, 2018. doi:10.1007/978-3-319-77935-5_3.

[15] Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue. Logic against ghosts: Comparison of two proof approaches for a list module. submitted, 2018.

[16] 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.

[17] 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.

[18] Bernard Botella, Mickaël Delahaye, Stéphane Hong Tuan Ha, Nikolai Kosmatov, Patricia Mouy, Muriel Roger, and Nicky Williams. Automating structural testing of C programs: Experience with PathCrawler. In *AST*, 2009.

[19] Remy Boutonnet and Mihail Asavoae. The WCET Analysis using Counters - A Preliminary Assessment. In *Proceedings of the 8th Junior Researcher Workshop on Real-Time Computing (JRWRTC)*, pages 21–24, Versailles, France, October 2014. URL `http://www.cister.isep.ipp.pt/jrwrtc2014`.

[20] 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.

[21] David Bühler, Pascal Cuoq, and Boris Yakobowski. *EVA - The Evolved Value Analysis plug-in*. URL `http://frama-c.com/download/frama-c-value-analysis.pdf`.

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

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

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. Uppaal smc tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415, Aug 2015. ISSN 1433-2787. doi:10.1007/s10009-014-0361-y.

[33] 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.

[34] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - A lightweight and flexible operating system for tiny networked sensors. In *Proc. of the 29th Annual IEEE Conference on Local Computer Networks (LCN 2004)*, pages 455–462. IEEE Computer Society, 2004. doi:10.1109/LCN.2004.38.

[35] C. Ferdinand. Worst case execution time prediction by static program analysis. In *18th In-*

ternational Parallel and Distributed Processing Symposium, 2004. Proceedings., April 2004. doi:10.1109/IPDPS.2004.1303088.

[36] Paul G. Flikkema. Energy-efficient model inference in wireless sensing: Asymmetric data processing. In *Ninth Annual IEEE Conference on Sensors (SENSORS 2010)*, 2010.

[37] Paul G. Flikkema. How much information per joule? Measuring the energy efficiency of inferential wireless sensing. In *Communications Workshops, IEEE Int'l Conference on Communications (ICC 2011)*, pages 1–5, June 2011.

[38] 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.

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

[40] 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.

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

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

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

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

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

[46] 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.

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

[49] 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.

[50] 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.

[51] 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`.

[52] 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.

[53] 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.

[54] 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.

[55] Alex Groce, Josie Holmes, and Kevin Kellar. One test to rule them all. In *International Symposium on Software Testing and Analysis*, pages 1–11, 2017.

[56] Armaël Guéneau, Arthur Charguéraud, and François Pottier. A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In Amal Ahmed, editor, *Programming Languages and Systems (ESOP)*, volume 10801 of *LNCS*, pages 533–560. Springer International Publishing, 2018. doi:10.1007/978-3-319-89884-1_19.

[57] 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`.

[58] F. Hanssen, A. Mader, and P. G. Jansen. Verifying the distributed real-time network protocol rtnet using uppaal. In *14th IEEE International Symposium on Modeling, Analysis, and Simulation*, pages 239–246, Sept 2006. doi:10.1109/MASCOTS.2006.52.

[59] Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. The Heptane Static Worst-Case Execution Time Estimation Tool. In Jan Reineke, editor, *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, volume 57 of *OpenAccess Series in Informatics (OASIcs)*, pages 8:1–8:12, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2017.8.

[60] 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.

[61] 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.

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

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

[64] Xiaowan Huang, Anu Singh, and Scott A. Smolka. Using integer clocks to verify the timing-sync sensor network protocol. In *Second NASA Formal Methods Symposium - NFM 2010, Washington D.C., USA, April 13-15, 2010. Proceedings*, pages 77–86, 2010.

[65] 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.

[66] Balázs Kiss, Nikolai Kosmatov, Dillon Pariente, and Armand Puccetti. Combining static and dynamic analyses for vulnerability detection: Illustration on Heartbleed. In *Hardware and Software: Verification and Testing (HVC)*, volume 9434 of *LNCS*, pages 39–50. Springer, 2015. doi:10.1007/978-3-319-26287-1_3.

[67] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. *arXiv preprint arXiv:1808.09700*, 2018.

[68] J. Knapp and P. G. Flikkema. Design and implementation of an energy-neutral solar energy system for wireless sensor-actuator nodes. In *2017 IEEE Global IoT Summit (GIoTS-2017)*, June 2017.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] Frédéric Loulergue, Allan Blanchard, and Nikolai Kosmatov. Ghosts for lists: from axiomatic to executable specifications. In *Tests and Proofs (TAP)*, volume 10889 of *LNCS*, pages 177–184. Springer, 2018. doi:10.1007/978-3-319-92994-1_11.

[76] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using FDR. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 1055 of *LNCS*, pages 147–166. Springer, 1996. ISBN 3-540-61042-1. doi:10.1007/3-540-61042-1_43.

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

[78] Claire Maiza, Pascal Raymond, Catherine Parent-Vigouroux, Armelle Bonenfant, Fabienne Carrier, Hugues Cassé, Philippe Cuenot, Denis Claraz, Nicolas Halbwachs, Erwan Jahier, Hanbing Li, Marianne de Michiel, Vincent Mussot, Isabelle Puaut, Christine Rochange, Erven Rohou, Jordy Ruiz, Pascal Sotin, and Wei-Tsun Sun. The W-SEPT Project: Towards Semantic-Aware WCET Estimation. In Jan Reineke, editor, *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, volume 57 of *OpenAccess Series in Informatics (OASIcs)*, pages 9:1–9:13, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.WCET.2017.9.

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

[80] Mark Mossberg. `https://blog.trailofbits.com/2017/04/27/manticore-symbolic-execution-for-humans/`, April 2017.

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

[82] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978. doi:10.1145/359657.359659.

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

[84] Mário Pereira and Simão Melo de Sousa. Complexity checking of ARM programs, by deduction. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC)*, pages 1309–1314, New York, NY, USA, 2014. ACM. doi:10.1145/2554850.2555012.

[85] 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.

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

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

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

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

[89] 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.

[90] Admar Ajith Kumar Somappa, Andreas Prinz, and Lars Michael Kristensen. Model-based verification of the DMAMAC protocol for real-time process control. In Belgacem Ben Hedia and Florin Popentiu Vladicescu, editors, *Proceedings of the 9th Workshop on Verification and Evaluation of Computer and Communication Systems, VECoS 2015, Bucharest, Romania, September 10-11, 2015.*, volume 1431 of *CEUR Workshop Proceedings*, pages 81–96. CEUR-WS.org, 2015. URL `http://ceur-ws.org/Vol-1431/paper9.pdf`.

[91] 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.

[92] 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.

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

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

[95] 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.

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

[97] E. van der Weegen and J. McKinna. A Machine-checked Proof of the Average-case Complexity of Quicksort in Coq. In Stefano Berardi, Ferruccio Damiani, and Ugo de'Liguoro, editors, *Types for Proofs and Programs, International Conference (TYPES 2008)*, LNCS 5497, pages 256–271. Springer, 2008. doi:10.1007/978-3-642-02444-3.

[98] Benoît Vaugon, Philippe Wang, and Emmanuel Chailloux. Programming microcontrollers in OCaml: The OCaPIC project. In *Practical Aspects of Declarative Languages (PADL)*, volume 9131 of *LNCS*, pages 132–148. Springer, 2015. doi:10.1007/978-3-319-19686-2_10.

[99] 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.

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

[101] 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.

[102] K. Yamamoto, Y. He, P. Heinrich, A. Orange, B. Ruggeri, H. Wilberger, and P.G. Flikkema. WiSARD-Net 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.

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

[104] 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.

[105] Fengling Zhang, Lei Bu, Linzhang Wang, Jianhua Zhao, Xin Chen, Tian Zhang, and Xuandong Li. Modeling and evaluation of wireless sensor network protocols by stochastic timed automata. *Electronic Notes in Theoretical Computer Science*, 296:261–277, 2013. ISSN 1571-0661. doi:https://doi.org/10.1016/j.entcs.2013.09.001. URL `http://www.sciencedirect.com/science/article/pii/S1571066113000479`. Proceedings the Sixth International Workshop on the Practical Application of Stochastic Modelling (PASM) and the Eleventh International Workshop on Parallel and Distributed Methods in Verification (PDMC).

[106] 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.