Practical Aspects of Building a Constrained Random Test Framework for Safety-Critical Embedded Systems

Dongjiang You[1,2], Isaac Amundson[2], Scott A. Hareland[2], Sanjai Rayadurgam[1]
[1]Department of Computer Science and Engineering, University of Minnesota, USA
[2]Cardiac Rhythm Disease Management, Medtronic, Inc., USA
[djyou, rsanjai]@cs.umn.edu, iamundson@gmail.com, scott.hareland@medtronic.com

ABSTRACT

In the safety-critical embedded system industry, one of the key challenges is to demonstrate the robustness and dependability of the product prior to market release, which is typically done using various verification and validation (V&V) strategies. Directed verification testing is a common strategy that performs black-box testing at the system level; however, it only samples a small set of specific system behaviors and requires heavily manual effort. In this paper, we describe our experience and lessons learned of applying the concept of constrained random testing on safety-critical embedded systems as a complimentary testing methodology. Constrained random testing enables us to cover many more system behaviors through random input variations, random fault injections, and automatic output comparisons. Additionally, it can reduce manual effort and increase confidence on the dependability of both firmware and hardware.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability

Keywords

Safety-critical embedded systems, constrained random testing, practical experience

1. INTRODUCTION

Safety-critical embedded systems are embedded systems whose failure or malfunction could result in death or serious injury to people, property damage or loss, or damage to the environment. Examples of well-known safety-critical embedded systems are aircraft flight control systems, automotive braking systems, and implantable medical devices. Correct operation is of greatest concern to safety-critical embedded systems because their consequences of failure can be serious.

MoSEMInA '14, May 31, 2014, Hyderabad, India

One of the key challenges in the safety-critical embedded systems industry is demonstrating the robustness and dependability (e.g., reliability, safety, availability, etc.) of the product prior to market release [1]. This is generally done using a verification and validation (V&V) strategy that consists of testing, reviews, analysis via formal methods, and other techniques. Specifically, *verification testing* (VT) denotes the process that is performed to demonstrate that a system or subsystem (e.g., hardware, firmware, etc.) meets its requirements.

In order to perform VT on safety-critical embedded systems, a complete set of requirements and specifications should be available, which are typically written in a natural language (e.g., plain English) to describe the intended behavior of the device. Test engineers then write VT tests that exercise the device in a way that demonstrates whether a given requirement is satisfied. Each VT test contains concrete input values that trigger certain functionality and concrete output values that the device is expected to produce. These types of test inputs are commonly called *directed* test inputs, and these types of test oracles are commonly called *expected value* test oracles [2]. After VT tests are created, they are executed and will either return a pass or fail status. If a test passes, confidence is gained in the dependability of the system. If a test fails, the subsystem under test must be manually inspected to diagnose the problem.

This directed test process is sufficient to demonstrate system dependability; however, although it can be effective in finding faults, it can still be improved due to the following reasons:

- Only a small subset of the input space can be tested, and there is no sound basis for extrapolating from tested to untested cases.
- Heavily manual effort is still needed to not only write concrete test input values for certain scenarios, but also calculate the concrete expected output values to be asserted.
- Directed verification testing is usually performed in late development stages and its cost is substantial with respect to the overall development cost.

The concept of constrained random testing on safety-critical embedded systems, including both firmware and hardware, is a complimentary testing methodology that provides additional benefit. For firmware, constrained random testing enables equivalence testing of two different systems, or two different versions of the same system, by providing each with constrained random test inputs and checking whether

they produce consistent results. One system (e.g., a verified legacy system) serves as a source-of-truth or *oracle* and the other system (e.g., a new system or an incremental version) serves as the system under test. For hardware, random fault injection on constrained hardware blocks enables the evaluation of the architectural vulnerability factor (AVF), which is a measure of the effect of single event upsets (SEUs) on system behavior. Each SEU is simulated through a fault injection at a random hardware location at a random time, and is checked against an oracle that is provided by the original system. By performing constrained random fault injection testing, we can measure how SEUs impact the behavior of the system.

In this paper, we describe our experience of developing and applying this testing methodology. Specifically, we make the following contributions to the safety-critical embedded system community:

- We describe how constrained random testing can be performed on both firmware and hardware in practice.
- We focus on lessons learned from applying this methodology as well as important design decisions we made in order to balance cost-benefit.
- We describe the challenges encountered with implementing such a test framework, provide design suggestions, and highlight the obstacles that need further research from the test community.

The rest of this paper is organized as follows. We introduce background information on firmware and hardware verification testing in Section 2. We next describe our approach towards realizing a constrained random test framework for both firmware and hardware in Section 3. We discuss important design decisions, challenges, and obstacles in Section 4. We describe related work in Section 5, and present conclusions and future work in Section 6.

2. BACKGROUND

2.1 Verification Testing

Verification testing is a black-box testing strategy at either the system or subsystem level, and is performed to ensure that the system under test meets its requirements.

Testing is an expensive and time-consuming set of activities. In fact, the cost of testing can be more than 50% of overall development cost [3]. Because of the time and manual effort involved in creating and running tests, typically only a few tests are created for exercising each required behavior. To gain confidence in the validity of a test, the concrete input values are determined by test engineers to reflect likely use conditions. However, the vast majority of the input space is left untested, and there is no sound basis for extrapolating from tested to untested cases.

As illustrated in the top half of Figure 1, since we do not know where the faults are in our system, we essentially make educated guesses to choose the best input values and observe the test results. If the test passes, we assume that the test would pass for any combination of input values. However, this assumption is generally invalid because there can be hundreds of thousands of combinations of input values for each use scenario or functionality, all of which can lead to a large number of different system behaviors.

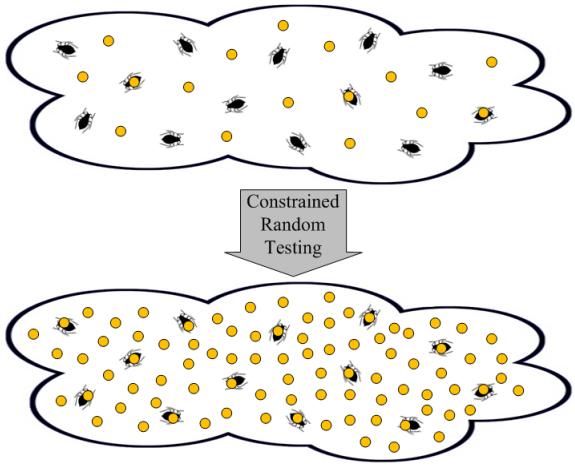

Figure 1: Coverage improvement through constrained random testing. The cloud represents an embedded system, bugs represent faults, and dots represent test cases

Despite the recent emergence of automated testing techniques, heavily manual effort is still needed to not only write concrete test input values for certain scenarios but also calculate the concrete expected output values to be asserted. Whenever there is a change in the requirements or design of the device, or even a change in the testing environment, test engineers have to re-write those concrete input values as well as re-compute the corresponding concrete expected output values.

Furthermore, directed verification testing is usually performed in late development stages and it can be extremely expensive. It has been recognized by both industry and academia that the cost of fixing bugs can increase significantly in later software development stages [4]. This fact has driven researchers and engineers to develop techniques to discover faults as early as possible by performing verification activities at early development stages.

Constrained random testing could be used as a complimentary test methodology to directed testing with the potential to provide additional confidence in the test results. Through random variations of input values, it would be possible to test a much larger part of the input space, as illustrated in the bottom half of Figure 1.

2.2 Hardware Fault Injection

Single event upsets (SEUs) are an important and well-known phenomenon in electronics, caused by energetic atmospheric neutrons, thermal neutrons, and alpha particles. There are numerous design techniques for safety-critical embedded systems that enable continuous operation in the presence of SEUs, specifically memory error detection and correction methods [5].

When an embedded system is subject to an SEU, a variety of consequences may result. At the simplest level, an SEU may change the logic state of a gate (e.g., flip-flop, register, etc.) and subsequently be over-written before the corrupted data is ever accessed. Such an event causes no disruption to the performance of a system and is never noted or observed by any system monitor or seen by the user. On the

other hand, if the corrupted data is subsequently accessed and read, there is a possibility that the faulty data may have zero impact to downstream logic due to logic masking. For example, *false* masks all other operands in an *and* operator and *true* masks all other operands in an *or* operator (see Figure 2). Otherwise, the SEU may cause an error in subsequent logic. The propagation of the error throughout the embedded system may result in no observable impact to the system performance, or it may trigger a more noticeable effect that may be observed by end users.

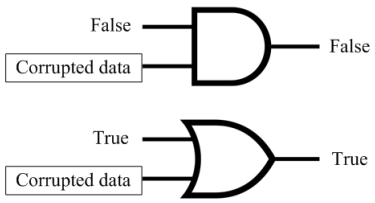

Figure 2: Corrupted data masked by logic operators

Thus, not all observable changes translate into system failures, and in many embedded systems, the traceability of SEU propagation to the visible output of the system is difficult to accurately determine based on simple analysis of circuit cross-sections, gate counts, and estimates of susceptibility. Simply assuming that all errors are bad errors may drive extensive work or effort that is not necessary. It is beneficial, therefore, to obtain a more accurate estimate of the fraction of these random SEUs that actually cause a change in the system behavior.

The effects of SEUs can be evaluated by means of the architectural vulnerability factor (AVF), which is a measure of how random SEUs impact the performance of the system. Knowing the AVF enables us to determine if design changes are necessary to meet customer dependability expectations. If the AVF is small (i.e., when the fraction of SEUs that cause significant errors is small), additional design effort may not be required. However, if the AVF is large, it may drive more extensive design mitigation techniques to obtain the required level of acceptability.

3. APPROACH

In order to evaluate the constrained random testing methodology, we built a unified framework for both firmware and hardware testing. This testing framework can automatically: (1) generate and execute constrained random VT tests in multiple test environments; (2) inject single event upsets at random hardware locations and random times; and (3) analyze test results and identify important cases for further inspection. In this section, we present our design decisions for building such a testing framework in detail.

3.1 Testing Framework at a High-level

The constrained random testing framework concept is illustrated in Figure 3. Test inputs are automatically generated with random variations based on constraints. The constraints are helpful for narrowing the range of possible input values and providing more meaningful test inputs to reach hard-to-cover or interesting system behaviors. Test results will identify risk areas and drive appropriate design changes. This automated and continuous approach can maximize the number of tested execution pathways. Hundreds of thousands of random tests can be continuously run, and because test coverage increases with time, this type of testing also enables discovery of real reliability growth threshold [6].

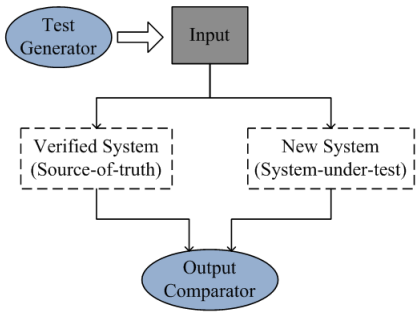

Figure 3: Testing framework overview

The test framework contains three major components: (1) a test generator that generates random test inputs based on constraints (for both hardware and firmware); (2) a test environment for executing the tests; and (3) an output comparator that compares outputs from two systems and scores the level of differences.

3.2 Constrained Random Test Generation

In order to create such a testing framework, we need a test generator. The top part of Figure 4 shows the process of generating constrained random test inputs.

Ideally, the test generator should take information from implementation-independent sources such as formalized requirements or test descriptions. However, such an ideal artifact is not always available and there is a gap between requirements (written in plain English) and VT tests (written in computer programming/scripting languages), all of which prevent us from applying our testing methodology in an ideal way. On the other hand, a large amount of existing VT tests are usually available, and for quality reasons they are typically well-formatted and well-documented.

We therefore have the following options for creating constrained random tests: (1) formalizing requirements; (2) creating implementation-independent test descriptions; and (3) using existing tests to extract useful information. We compared the cost-benefit of each of these options and concluded that option (3) was optimal due to the following reasons:

- It can be relatively easy and inexpensive to automatically extract information from existing VT tests using static code parsing. Whether the existing VT tests were created by test engineers or generated automatically, they still follow certain rules and formats, which make them machine readable.
- Although requirement formalization and an implementation-independent test description language can be potentially more efficient for generating new tests, these are not trivial tools to construct, and would require a significant amount of time and effort to implement and qualify.

Therefore, starting with existing VT tests, we wrote a simple parser to extract all statements that set input parameter values and expected output values for the system.

We then created a simple constraint description language that supports constraint specification for input parameters. Parameter ranges can be specified as continuous or discrete values and they provide bounds for generating random values. Parameter dependencies can be specified through arithmetic relations. Due to the real-time nature of safety-critical embedded systems, the same parameter may need different constraints at different test steps. In those cases, we made it possible to assign unique tags to those parameter instances. The constraints could then also be specified using those tags. All constraints are specified in comments within the test script using pre-defined formats and keywords. Specifying constraints in comments has the benefit that the original VT tests can still be used elsewhere without any side effects.

After extracting all applicable parameter information from the original VT test, the test generator starts generating random values for new tests. Uniformly distributed random values[1] are generated for each designated parameter based on their ranges. A test may have multiple instances of a parameter, each with different ranges and dependencies. If there are any dependencies among parameters, they will be checked: (1) if all dependencies are satisfied, we have a new random test; (2) otherwise, we generate another set of random values until we satisfy all constraints.

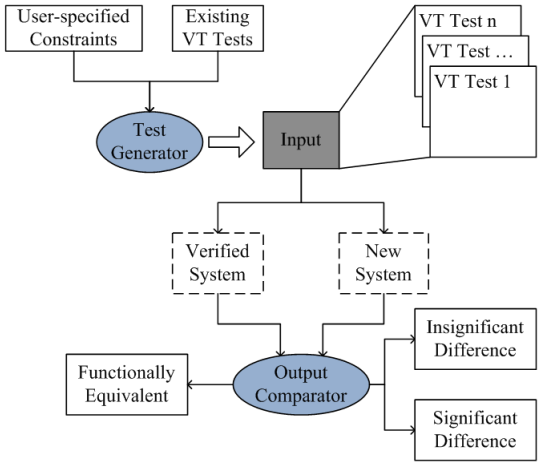

Figure 4: Test generator and output comparator

This "brute-force" approach is simple and effective, although it may be inefficient in some cases. On average it took less than 1 second to generate a new test, and several seconds in the worst case. Using a constraint solver would likely make the test generation more efficient, but it would require additional time for us to study, for example, the distribution of random values a solver would produce or what specific theory a given solver uses to generate random values. Instead of generating values in an unknown way, we decided to use the current approach.

As discussed earlier, existing VT tests contain expected outputs written as assertions. When they are being executed, actual outputs are checked against expected outputs. If the actual outputs match expected outputs, the test passes. Otherwise, the test fails. However, given randomized test inputs in generated new tests, the expected outputs in the original tests are no longer valid because expected outputs are tightly coupled with their respective inputs. Therefore, in the process of generating new tests, all statements that check outputs were replaced with statements that record the output values to a log file. Those output variables and expressions were only recorded if they were asserted in the original tests. This setting enables the new tests to have the same strength as the original tests from the oracle perspective, and will be further discussed in Section 3.5.

[1]As randomness might be a concern in random testing literature, we generated seeded "pseudo-random" values by standard deterministic methods rather than purely nondeterministic random values, but we still use the term "random" in order to be consistent with published work.

3.3 Hardware Fault Injection

It is a common approach to estimate the AVF of an embedded system by performing hardware fault injection studies [7].

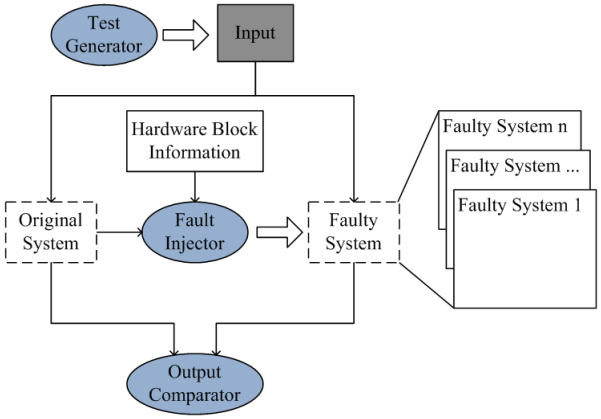

Figure 5: Hardware fault injection

For these kinds of studies, the two comparison systems include the original system and the system with a random injected fault, as shown in Figure 5. To be more specific, we can establish a baseline consisting of a variety of use scenarios (by executing a set of VT tests) with no introduced faults, and store their output as the source of truth. In small systems, a complete simulation of the state space can be modeled, but typically the system will be large enough to make it impractical to apply this approach. Thus, we apply a sampling method that randomly injects faults (i.e., random bit flips) into the hardware model to reflect the random nature of SEUs. We can execute the same tests and compare the outputs of the system under test with the original baseline source of truth. In the system under test, the fault injector injects one random bit flip at a random time. The random bit flip is selected from user-specified hardware blocks, or all blocks by default. The random time when this bit flip happens is a number between 0 and the total time needed to finish the original test, which can be measured when executing the baseline VT tests.

We can then obtain an important estimate of the AVF of our system by repeating this process for a large number of random faults and evaluating whether they lead to visible effects. The output comparator compares the output produced by the baseline source of truth and the output produced by the faulty system, and groups the results into

different categories. For example, if 1000 random bit flips are injected in 1000 separate executions and we observed 30 significant effects on output, then it would be concluded that the AVF is 3% +/- statistical uncertainty. Knowing the AVF provides quantified data for the analysis of potential design solutions necessary to achieve system dependability goals.

3.4 Test Environment

Because we are using existing VT tests, the test environment that executes them already exists. The constrained random test framework executes the generated tests according to the interface provided by the VT environment. Therefore, the only thing we need to handle here is the generation of automated scripts (e.g., shell scripts, batch scripts, etc.) to enable automatic execution of a large number of generated random tests.

3.5 Output Comparison

The bottom part of Figure 4 shows the process of comparing outputs from two systems.

In the generated tests, we have replaced all expected value test oracles (i.e., concrete expected output values that determine whether a test passes or fails) with statements that simply record actual output values into log files. This approach enables us to run tests without determining whether they pass or fail, which is necessary because we cannot determine whether a random test passes or fails without an oracle. We therefore compare actual output values from two different systems (or two different versions of the same system) and determine whether they are consistent for a given test execution. The output comparator maintains at least the same oracle data [8] as the original tests and thus has the same fault-finding capability from the oracle perspective.

To complicate matters, the system that is being used as the source of truth may operate in a different test environment[2] (including OS, IDE, HW breadboard, etc.) from the system under test. For testing a system against a previous version, or testing a system with injected faults against its unaltered version, there is usually one single testing environment. In these cases, our testing framework is able to compare not only output values but also any intermediate states that are recorded by the original tests. This enables our testing framework to have stronger fault-finding capability because triggered faults might not propagate or be observable in the output.

However, if we would like to test a new system against a similar legacy system, each of which uses a different testing environment, the output comparator can only compare output values. This is because, although the same output values can be recorded, different intermediate results will be recorded in different ways. There are two difficulties here: (1) we cannot add code to record intermediate results because it would change system behaviors such as timing; (2) it is extremely difficult to align information from two different testing environments. We will discuss these issues further in Section 4.

Finally, the output comparator groups results into three categories: (1) *Functionally Equivalent*, if two systems produce exactly the same results (including intermediate results when used in the same testing environment); (2) *Insignificant Difference*, if two systems produce the same outputs but different intermediate results (in case of different testing environments, there is no such category); (3) *Significant Difference*, if two systems produce different outputs (i.e., the test fails from the traditional testing perspective).

By taking this approach, we do not need to calculate expected outputs by manual effort. We feed random test inputs to the "correct" system and check whether the "system under test" produces the same output. This approach can improve test productivity and reduce the amount of engineer resources required for testing.

[2]By testing environment, we refer to the system itself and all (simulated) environmental elements that make a test executable.

4. DISCUSSION

4.1 Randomness and Constraints

There are many studies in the literature on the merits of random testing versus directed testing. Some works state that random testing cannot be as effective as directed testing because many interesting hard-to-cover system behaviors have little chance of being tested [9]. Other works suggest that random testing can indeed be as effective as directed testing [10]. By constraining the random test input, we combine the benefits of both types of testing, but it can be challenging to maintain and control an appropriate level of randomness.

On the one hand, if the randomness is too high, the generated tests may not represent meaningful system behaviors. When a system is provided invalid inputs, the system may simply reject those inputs by entering a safe state in order to avoid damage. For example, a typical microwave oven does not do anything if the "Start" button is pressed when the door is open. In these cases, "do-nothing" and "reset-system-state" are *correct* system behaviors for invalid inputs, but otherwise provide little valuable information on the dependability of this system. On the other hand, if the randomness is too low, the generated tests tend to be similar to directed testing and thus suffer from the same problems that directed testing has (e.g., only a small number of specific input values are tested heavily and most of the input space is left untested).

Therefore, constraints are used to narrow the possible input values and control the level of randomness. Constraints will initially be specified by users rather than being automatically generated. For example, parameter ranges are typically defined in requirements and design specification documents. Dependencies among parameters are usually defined implicitly in requirements, which are written in plain English and thus difficult to parse using software tools. Furthermore, due to the real-time nature of safety-critical embedded systems, the same parameter may need different constraints at different test steps, which makes it even more difficult to automatically generate the constraints.

On the other hand, when users write tests manually, it is a natural process to write input values according to the constraints in mind in order to cover certain system behaviors. It is then a matter of writing the constraints down in a specific format, which does not require much additional effort. Once constraints are specified, they can also be reused to generate many new random tests.

Regarding randomness and constraints in this test methodology, we have the following lessons learned:

It can be an effective and efficient approach to generate a large number of constrained random tests that cover more system behaviors. Although constraints have to be manually specified by users, they are more precise and reusable, and can be written within a reasonable amount of time.

4.2 Oracles

The source of oracles is usually not a problem because, when a testing approach is proposed and evaluated, it is common to evaluate its fault-finding effectiveness by means of a fault injection study, in which a system with injected faults is compared with its original version. For example, mutation testing is widely used in evaluating software test generation techniques, and it has been shown that generated mutants based on a set of mutation operators can be used to predict the effectiveness of finding real faults [11]. In these cases, the original system serves as the baseline source-of-truth and the system with injected faults serves as the system-under-test. Similarly with hardware fault injection, we can use the original system to generate oracles, against which the system with injected single event upsets is tested.

One of the major obstacles of applying automated testing techniques in practice is that it is still difficult to automatically generate oracles. Although, for example, in model-based testing, test oracles can be generated from the formal model of the system, it will not work in situations where a formal specification is not available. The most common approach then is to use manual effort to review whether the actual output matches the expected output from each test execution of the embedded system. This situation is less than ideal when we apply constrained random testing because the exact test input values are unpredictable from the perspective of the user, so users cannot calculate the expected output values until the random tests are generated.

The benefit of manual inspection is obvious. Test engineers are able to provide more accurate and reliable comparisons, and once the comparison is done, it is usually easier to trace back to the root cause in the case of a mismatch. However, manual inspection is not efficient. The time it can take a test engineer to write a complete VT test, including concrete input and expected output values, can be measured in days for some complex safety-critical embedded systems. In the case of constrained random testing, this time would essentially be multiplied by the number of randomized tests to be run. Furthermore, diagnosing a mismatch may eventually turn out to be a minor issue such as a simple variation due to non-determinism.

However, it is possible to generate oracles from a verified legacy system or a previous version of the same system, provided that they exist and most of the functionality remains the same. For example, if some functionality or component is added to or removed from the original system, it should not affect the intended behavior of other components (unless it is supposed to do so). There may also be new designs or optimizations that make the system more efficient or operate with fewer resources but should otherwise not change the behaviors of the system.

The advantage of automated constrained random testing is apparent when there exist completely tested and verified legacy systems or previous incremental versions available that can be used to generate oracles for random test inputs. This is usually the case for safety-critical embedded systems because they are often developed as a family of products. Since it is our goal to apply this test methodology as a supplementary process to improve test effectiveness, the majority of the test results from the two systems can be compared and analyzed automatically, and a significant amount of manual effort is reduced. Only a small number of inconsistent test results are stored for further manual inspection.

Regarding test oracles in practice, we have the following lessons learned:

The oracle is an important but overlooked aspect of testing in practice. Manual inspection is typically used, but is highly inefficient. We can generate oracles from verified systems or previous versions as a supplementary approach to manual inspection to reduce manual effort significantly.

4.3 Non-determinism

The impact of non-determinism on testing cannot be overlooked. For safety-critical embedded systems, specifically real-time reactive systems, sources of non-determinism such as timing can greatly alter the effectiveness of test oracles, as well as complicate the alignment of intermediate results.

A good oracle can be characterized on two aspects, low false positives and low false negatives. We would like our test framework to have an equal or lower false negative rate than our existing test methods, but it may have an equal or higher false positive rate because two different outputs can both be acceptable if their difference is within a specific tolerance. For example, suppose a device is supposed to emit a signal every second. When the device is running in reality, it is impossible for it to release signals at exact one-second intervals due to reasons such as hardware overhead and clock inaccuracies. However, it could still be acceptable for the system to release signals within a +/- 10 ms tolerance.

When engineers write tests, they include tolerances when appropriate. We attempted to apply general rules (e.g., certain kinds of parameters may have a similar tolerance), to filter false positives (i.e., a mismatch between two outputs that turns out to be acceptable). However, we found that it was not feasible to summarize a set of tolerances, and so they are instead configured case-by-case. In the comparison results, we ranked possible faulty output values by their magnitude difference in order to assist users in finding true faults.

Regarding non-determinism, we have the following experience:

Timing is a challenge in testing real-time embedded systems and it can reduce the effectiveness of test oracles. More advanced research and techniques are needed, but it is fairly straight-forward to sort differences in order for users to filter false positives.

4.4 Limited Output Information

One of our implicit goals when applying this test methodology was to test the system without changing its original behavior. As mentioned in Section 3, we replaced expected value test oracles with statements that record actual output values in the generated random tests. This change should not interfere with the system during testing (and thus not affect its behavior) because the original VT test would obtain the exactly same information from the system or testing environment.

Additionally, we can use any intermediate information that the system or the original test provides. Therefore, we have the following two situations: (1) when the two systems for comparison in our testing framework run in the same test environment, we can compare both output and intermediate information; and (2) when the two systems are in different testing environments, we can still compare output information, but we cannot compare intermediate information because different testing environments may record different information in different ways and at different times.

In both cases, we can only compare the information that is provided to us, which is specified by the system or in the original test and carefully handled by engineers. For example, the test should be designed to give enough time to print a message between two events. Otherwise, printing the message would delay the event following it. We may not be allowed to record additional intermediate information because it would change the original system behavior. The embedded system is different from traditional systems due to its limited resources. Thus, we can only compare intermediate results if they are already available to us, and this prevents us from using certain testing techniques and instrumentation because it may change timing and hence change system behavior. Therefore, we have limited options of what data we can record as the output. For similar reasons, we cannot substantially modify or control intermediate information if we would like to compare systems on different testing environments.

The second case is even more challenging because, (1) it is difficult to make two test environments produce the same kind of information without changing the system behavior (i.e., we have to calculate carefully if we want to print another message between two events); and (2) it is time-prohibitive for us to align information from two different environments (i.e., map outputs from one environment to outputs of another environment).

Given those difficulties, we still believe that our testing framework has at least the same fault-finding capability as the traditional tests from the oracle perspective because the same output variables and expressions can always be compared in any case.

It is also worth noting that, although intermediate results can potentially provide valuable insights and more information on test executions, they can be expensive to compare. Therefore, this test framework would benefit from applying new techniques for selecting an appropriate set of intermediate results to compare [12] or actively propagating intermediate results to the output [13].

Regarding limited output information, we have the following challenges:

> Since resources are limited in safety-critical embedded systems, it is difficult to control and obtain intermediate results. However, we can maintain at least the same effectiveness as traditional testing and there are potential approaches currently under investigation that may improve this situation.

5. RELATED WORK

5.1 Random Testing

Random testing has received a lot of attention recently because it is simple to implement, can be effective in many cases, and there is no bias in generated and selected tests. Random testing has also been used as a baseline for evaluating test generation and selection techniques with the implication that a testing technique should achieve at least the same effectiveness as random testing [14]. Random testing is used for testing a wide range of applications, including Java programs [15], Haskell programs [16], and for GUI systems [17].

Some works have pointed out that random testing is weak compared with systematic testing, because many interesting hard-to-cover system behaviors have little chance of being tested [9]. However, other works show random testing in a more favorable light, and suggest that it can indeed be as effective [10, 18].

Apart from a large body of work on random testing, there is a set of work that combines random testing and systematic testing, which is similar to our constrained random testing methodology (as opposed to purely random testing). Specifically, a test generation technique has been proposed that starts with executing a random input on the program under test, and then systematically modifies the input such that it covers different paths [9]. Another approach, called DART, integrates random input generation with symbolic execution [19]. A tool called RANDOOP has also been created, which is primarily a random input generator, but applies techniques to systematically search input to make it more effective [15]. Furthermore, there are approaches based on the Design-by-Contract development approach to generate black-box random tests for Java methods [20, 21]. They generate test inputs by random testing and solving constraints on the pre-conditions, and generate expected outputs by executing post-conditions.

The major (and most significant) difference between the above techniques and our approach is that they are all techniques that support unit testing rather than black-box system testing. On the one hand, unit testing is effective in finding faults in each isolated component and there are scalable approaches that make unit testing automatic. On the other hand, unit testing is completely different from system testing due to reasons including (1) unit testing does not address communications and interactions among modules and subsystems, which are sources of serious faults that are difficult to test and expensive to fix; (2) techniques for unit testing may not be feasible or scalable for system testing; and (3) black-box system testing is usually carried out by system test engineers who are often not familiar with the system design, although they know the domain very well.

Adaptive random testing (ART) is a testing approach that improves on random testing [22]. In ART, random test inputs are generated or selected from an existing test pool in a way that achieves diversity and covers failure patterns. The underlining concept of ART is similar to our testing methodology, which is that diversity plays a fundamental role in tests and the implicit goal is to cover more system behaviors.

5.2 Testing Real-time Embedded Systems

A modern approach to testing real-time embedded systems is to perform conformance testing between the system and its specification [23]. The specification is usually a formal model of the system and it can also be used to generate tests. Although black-box conformance testing can be an effective approach, it will not work in situations in which

a formal specification is not available, and it typically does not scale very well for large and complex systems.

The difficulties and uniqueness of black-box system testing for real-time embedded systems are addressed in [24]. The authors point out that black-box system testing relies on the external environment rather than the internal design of the system. They further compare the effectiveness of random testing, adaptive random testing, and search-based testing with their modeling of real-time embedded system environment.

5.3 Oracles

There are two key artifacts necessary for software testing, the test inputs and the test oracles. Substantial research has been done to develop test input generation techniques. The test oracle problem, as described in [25], is the problem of constructing efficient and robust test oracles. Despite the increased attention and research on test oracles, they are still a major challenge not only for safety-critical embedded system testing, but also for software testing in general.

As classified in [26], there are four categories of oracles: specified oracles, derived oracles, implicit oracles, and human oracles. Following their definition, *specified oracles* can be generated from, for example, a formal specification [27]. In the event a formal specification is not available, *human oracles*, i.e., expected value test oracles, are usually used.

6. CONCLUSIONS AND FUTURE WORK

We have documented our experience of applying constrained random testing, including hardware fault injection, on safety-critical embedded systems and described the lessons learned from designing such a testing framework.

The constrained random test methodology permits us to (1) demonstrate the dependability of both firmware and hardware prior to market release; (2) perform fault injection testing that provides insight into the potential impacts of single event upsets on system performance; and (3) drive proactive, robust, appropriate firmware and hardware design that meets customer expectations.

We hope to explore how this test framework can be improved moving forward by investigating the following:

- Formalize requirements, create system models, or use test description languages to provide implementation-independent source-of-truth in place of a verified legacy system or a verified version.
- Apply advanced techniques such as solvers to generate random parameter values based on constraints. This would make our testing framework more efficient in generating random tests.
- Compare both output and intermediate results or apply more advanced oracle techniques such as oracle selection and oracle steering to improve the output comparison.

7. ACKNOWLEDGEMENTS

We would like to thank James Erickson, Joel Artmann, and Mats Heimdahl for their support and feedback in finishing this project.

8. REFERENCES

[1] John C Knight. Safety critical systems: challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering*, pages 547–550, 2002.

[2] Matt Staats, Michael W Whalen, and Mats PE Heimdahl. Better testing through oracle selection (NIER track). In *Proceedings of the 33rd International Conference on Software Engineering*, pages 892–895, 2011.

[3] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[4] Barry Boehm and Victor R Basili. Software defect reduction top 10 list. *IEEE Computer*, 34(1):135–137, 2001.

[5] Shu Lin and Daniel J Costello. *Error control coding*, volume 123. Prentice-hall Englewood Cliffs, 2004.

[6] Bev Littlewood. The problems of assessing software reliability... when you really need to depend on it. In *Proceedings of the 8th Safety-Critical System Symposium*, 2000.

[7] Shubhendu S. Mukherjee, Chris Weaver, Joel S. Emer, Steven K. Reinhardt, and Todd M. Austin. Measuring architectural vulnerability factors. *IEEE Micro*, 23(6):70–75, 2003.

[8] Matt Staats, Michael W Whalen, and Mats Per Erik Heimdahl. Programs, tests, and oracles: the foundations of testing revisited. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 391–400, 2011.

[9] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, 1996.

[10] Joe W Duran and Simeon C Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, (4):438–444, 1984.

[11] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, 2006.

[12] Matt Staats, Gregory Gay, and Mats PE Heimdahl. Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing. In *Proceedings of the 34th International Conference on Software Engineering*, pages 870–880, 2012.

[13] Michael Whalen, Gregory Gay, Dongjiang You, Mats PE Heimdahl, and Matt Staats. Observable modified condition/decision coverage. In *Proceedings of the 35th International Conference on Software Engineering*, pages 102–111, 2013.

[14] Richard Hamlet. Random testing. *Encyclopedia of Software Engineering*, 1994.

[15] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, pages 75–84, 2007.

[16] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 268–279, 2000.

[17] Justin E Forrester and Barton P Miller. An empirical study of the robustness of windows NT applications using random testing. In *Proceedings of the 4th USENIX Windows System Symposium*, pages 59–68, 2000.

[18] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. Formal analysis of the effectiveness and predictability of random testing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pages 219–230, 2010.

[19] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[20] Yoonsik Cheon, Antonio Cortes, Martine Ceberio, and Gary T Leavens. Integrating random testing with constraints for improved efficiency and diversity. In *Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering*, pages 861–866, 2008.

[21] Yi-Tin Hu and Nai-Wei Lin. Automatic black-box method-level test case generation based on constraint logic programming. In *2010 International Computer Symposium (ICS)*, pages 977–982, 2010.

[22] Tsong Yueh Chen, Fei-Ching Kuo, Robert G Merkel, and TH Tse. Adaptive random testing: The ART of test case diversity. *Journal of Systems and Software*, 83(1):60–66, 2010.

[23] Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, 2009.

[24] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. Black-box system testing of real-time embedded systems using random and search-based testing. In *Testing Software and Systems*, pages 95–110. 2010.

[25] Elaine J Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[26] Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. A comprehensive survey of trends in oracles for software testing. Technical report, CS-13-01, Department of Computer Science, University of Sheffield, 2013.

[27] Debra J Richardson, Stephanie Leif Aha, and T Owen O'malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th International Conference on Software Engineering*, pages 105–118, 1992.