INL/EXT-18-52032

Specification of a Bounded Exhaustive Testing Study for a Software-based Embedded Digital Device

Dr. Carl Elks, Christopher Deloglos, Athira Jayakumar, Dr. Ashraf Tantawy, Rick Hite, and Smitha Guatham
Department of Electrical and Computer Engineering
Virginia Commonwealth University

November 2018

U.S. Department of Energy
Office of Nuclear Energy

INL/EXT-18-52032

Specification of a Bounded Exhaustive Testing Study for a Software-based Embedded Digital Device

Dr. Carl Elks, Christopher Deloglos, Athira Jayakumar,
Dr. Ashraf Tantawy, Rick Hite, and Smitha Guatham
Department of Electrical and Computer Engineering
Virginia Commonwealth University
601 W. Main Street
Richmond, Virginia

November 2018

Idaho National Laboratory
Idaho Falls, Idaho 83415

http://www.inl.gov

Prepared for the
U.S. Department of Energy
Office of Nuclear Engineering
Under DOE Idaho Operations Office
Contract DE-AC07-05ID14517

ABSTRACT

Under the Department of Energy's Light Water Reactor Sustainability Program, within the Plant Modernization research pathway, the Digital Instrumentation and Control (I&C) Qualification Project is identifying new methods that would be beneficial in qualifying digital I&C systems and devices for safety-related usage. One such method that would be useful in qualifying field components such as sensors and actuators is the concept of testability. The Nuclear Regulatory Commission (NRC) considers testability to be one of two design attributes sufficient to eliminate consideration of software-based or software logic-based common cause failure (the other being diversity). The NRC defines acceptable "testability" as follows:

Testability – A system is sufficiently simple such that every possible combination of inputs and every possible sequence of device states are tested and all outputs are verified for every case (100% tested). [NUREG 0800, Chapter 7, Branch Technical Position (BTP) 7-19]

This qualification method has never proven to be practical when viewing a very large number of combination of inputs and sequences for device states in a typical I&C device. However, many of these combinations are not unique in the sense that they represent the same state space or the state space that would not affect the critical design basis functions of the device. Therefore, the state space of interest might possibly be reduced to a manageable dimension through such analysis.

This project will focus on a representative I&C device similar in design, function, and complexity to the types of devices that would likely be deployed in nuclear power plants as digital or software-based sensors and actuators (e.g., smart sensors). Analysis will be conducted to determine the feasibility of testing this device in a manner consistent with the NRC definition.

This report describes the development of test process for bounded exhaustive testing with respect to combinatorial test methods. The report describes the candidate Embedded Digital Device - the Virginia Commonwealth University smart sensor, conceptual experimental methods for stated test objectives, description of the process, tools, resources, and computing. This information will be used to fully develop a detailed test plan (based on statistical measure needs) and test environment for conducting an I&C device testability demonstration study. The future planed experimental study of this project is to demonstrate digital qualification via bounded exhaustive testability with respect to common cause failure.

CONTENTS

FIGURES

TABLES

ACRONYMS

ACTS	Automated Combinatorial Testing System
API	Application Program Interface
ASCII	American Standard Code for Information Interchange
BET	Bounded Exhaustive Testing
BVA	Bounded Value Analysis
CATS	Constrained Array Test System
CCF	Common Cause Failure
CT	Combinatorial Testing
DC	decision coverage
DUT	device under test
EDD	Embedded Digital Device
FTDI	Future Technology Devices International
GCC	GNU Compiler Collection
GPU	Graphics Processing Unit
HAL	hardware abstraction level
HW	Hardware
I&C	Instrumentation and Control
I/O	Input/Output
INT	Integer
MC	multiple condition
NIST	National Institute of Standards and Technology
NRC	Nuclear Regulatory Commission
OA	orthogonal array
RTOS	Real-Time Operating System
SDD	Software Design Document
SRS	Software Requirements Specification
SUT	system under test
SW	Software
TCAS	traffic collision avoidance system
UART	Universal Asynchronous Receiver Transmitter.
UAV	Unmanned Aerial Vehicles
USB	universal serial bus
VCU	Virginia Commonwealth University

Specification of Bounded Exhaustive Testing Process for a Software-based Embedded Digital Device

1. Introduction and Purpose

As digital upgrades to nuclear power plants in the United States has increased, concerns related to potential software common cause failures (CCF) and potential unknown failure modes in these systems has come to the forefront. The U.S. Nuclear Regulatory Commission (NRC) identified two design methods that are acceptable for eliminating CCF concerns: (1) diversity or (2) testability (specifically, 100% testability) [1]. As pointed out in Ammann and Offutt's book [2], there is near universal consensus among computer scientists, practitioners, and software test engineers that exhaustive testing for modestly complex devices or software is infeasible [3,4], which is due to the enormous number of test vectors (i.e., all pairs of state and inputs) needed to effectively approach 100% coverage [1]. For this reason, diversity and defense-in-depth architectural methods for computer-based Instrumentation and Control (I&C) systems have been become conventional in the nuclear industry for addressing vulnerabilities associated with common-cause failures [5]. However, the disadvantages to large-scale diversity and defense-in-depth methods for architecting highly dependable systems are well known: significant implementation costs, increased system complexity, increased plant integration complexity, and very high validation costs. Without development of cost effective qualification methods to satisfy regulatory requirements and address the potential for CCF vulnerability associated with I&C digital devices, the nuclear power industry may not be able to realize the benefits of digital or computer-based technology achieved by other industries. However, even if the correctness of the software has been proven mathematically via analyses and was developed using a quality development process, no software system can be regarded as dependable if it has not been extensively tested. The issues for the nuclear industry at large are: (1) what types of Software testing provide very strong "coverage" of the state space, and (2) can these methods be effective in establishing credible evidence of software CCF reduction? The previous report identified several promising testing approaches that purport to provide strong "coverage." Namely, Combinatorial Testing (CT) methods can achieve "bounded" exhaustive testing under certain conditions [6]. Additionally, the definition for coverage will be elaborated in later sections, including several definitions used in the SW testing community. This document also defines and develops a specification for an empirical "study or test" to collect data on the efficacy of CT methods for accomplishing "bounded exhaustive" testing.

1.1 Background

Reducing the occurrence of design defects/errors in software-based systems is principally accomplished by design assurance methods, which are typically comprised of process, analysis, and testing methods. Process usually includes best practices, prevailing standards, and regulatory guidelines that govern the lifecycle development device software for a given level of assurance needed. Analysis encompasses the methods used to access the design and implementation of the device software with respect to a set of requirements and specifications. Testing aims to achieve discernable differences between intended and actual behaviors of a system (observable at the level of resolution required for assurance), or at gaining confidence that there are no discernible differences. The goal of testing is defect detection: finding impactful differences between the behavior of the implementation and the intended behavior of the system under test (SUT), as expressed by its requirements. Software testing is a broad term encompassing a wide spectrum of different activities: testing of a small piece of code by the developer (unit testing), to the customer validation of an installed system (acceptance testing), to the monitoring at run-time of a network-centric service-oriented application. In the various stages, the test cases could be devised to aim at different objectives, such as exposing deviations from user's requirements, assessing the conformance to a standard specification, evaluating robustness to stressful load conditions or to malicious inputs (fuzzing for security), etc.

This document focuses on a perspective with respect to coverage and testability. Nuclear industry's definition of testability is different from the testability definition used by the software testing community. The NRC defines acceptable "testability" as follows:

> *Testability – A system is sufficiently simple such that every possible combination of inputs and every possible sequence of device states are tested and all outputs are verified for every case (100% tested).* [1]

The NRC's definition is more closely aligned with hardware testability metrics, rather than software testability measures. The software testability-related definition is:

> *Software testability is the degree to which a software artifact (i.e., a software system, software module, requirements- or design document) supports testing in a given test context. If the testability of the software artifact is high, then finding faults in the system (if it has any) by means of testing is easier.* [7]

The issue with the NRC definition is that any modest microprocessor-based embedded device executing ordinary control software has an effective infinite state space, thus direct 100% testability by state enumeration is infeasible for most software systems. Accordingly, qualification methods based on these criteria are only applicable for extremely simple systems and have never proven to be practical in view of the very large number of combinations of inputs and sequences of device states for a typical I&C device. Another issue with the NRC definition is there is no given definition of "states," which can lead to different interpretations of states and requisite coverage. For example, one valid definition of "states" is from the automata model of computability [6]. Automata models are abstract models of computations (either SW or HW) and provide the underlying formal basis for computers. The state of a finite automata (representing software) includes not only the information about which discrete state the software is in (indicated by the bubble in the figure below), but also what values any variables have. The number of possible states can be very large, or countably infinite. If there are *n* discrete states (bubbles), and *m* variables each of which can have one of *p* possible values, the size of the state space is:

$$|\text{states}| = np^m$$

Or more simply, take two "bubble" states and six variables (assuming all variable are unique). Use 16-bit INT data types to each variable, it produces:

$$|\text{states}| = 2(2^{16})^6 = 5.01 \times 10^{30} \text{ enumerated states} \quad (1)$$

As shown, this definition of states is extremely conservative in defining "uniqueness" amongst the elements of a state set. However, this is a like a ground definition of states—abstractions of state space can be built up from this definition based on assumptions of groupings, equivalence, and conditions. As far as is known, the NRC definition of testability provides no guidance on reasonable theoretical abstractions as other industries have done (notably commercial air transportation, and railway industry).

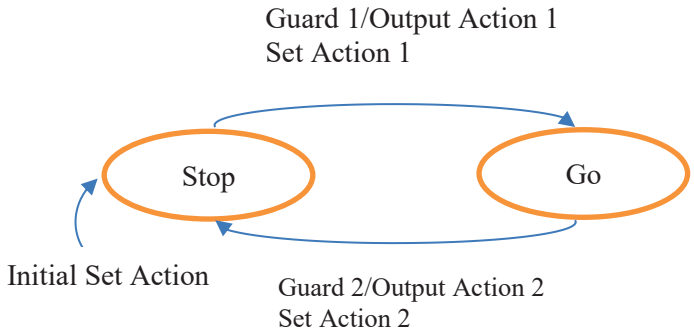

Figure 1. Extended finite automata.

For the above example, conducting exhaustive testing would take 10^{21} years to complete assuming one test per nanosecond. Obviously, this definition of states results in an impossibility of using testing to show a reduction of CCF. Another definition of "states" is related to the combinatorics of the variable and decision space of the digital behavior with respect to the software. A close cousin to the automata model is multiple condition model (that is, exhaustive testing from a condition evaluation aspect). Multiple condition model of testing defines states with respect to inputs and decisions points in the digital behavior of the software. That is, all possible combinations of inputs for each decision in the software. This ensures that the correct decision outcome is reached in all cases. Again, the problem with such testing is that for a decision with n inputs, 2^n tests are required. The multiple condition model is not doubly exponential as is the automata model, but it is still exponential in growth. In cases where n is small, running 2^n tests may be reasonable; running 2^n tests for large n is impracticable. As example, consider a fragment of code where 36 variables (conditions) are parameters to decisions statements (if-then-else, case statements, etc.). This is not an uncommon occurrence in control system software. At least 2^{36} tests would be needed to exhaustively test the decision conditions. How long would this take assuming 1000 tests/sec?

(236 tests)*(1 sec/1000 tests)*(1 minute/60 sec)*(l hour/60 min)*(1 day/24 hour)*(1 year/365 day) = Approximately 2.179 years

How much data space would be required to store this test result? If the test artifacts include a single line for the test results of each test case, with time, date, duration – then 10 bytes is reasonable for each test result.

2^{36} = 6.8719476736e+10 then 6.8719476736e+10 (10) =687.19476736 GB (2)

Again this is just for one decision code segment. This has to be repeated for each and every decision code segment.

The key to reducing the state space is recognizing that many of the states are equivalent in their behavior. In recent years, approaches to justifiably reduce the testable state space have made significant progress. These include methods based on equivalence partitioning: modified condition/decision coverage, t-way CT, and model-based testing. The state space of interest is reducible to a manageable dimension through such analysis methods. However, the degree to which these methods can provide coverage of critical code regions approaching "100%" needs further exploration—at least to the nuclear industry.

This report focuses on methods that support or claim high levels of "coverage" approaching exhaustive testing or bounded exhaustive testing. By bounded exhaustive testing we mean:

Definition: Bounded-exhaustive is used in relation to software testing. Software testing is considered bounded exhaustive when well-formed relations between input space and state space allow the testable

state space to be reduced, which enables a feasible testable set. The bounded aspect relates to the lower bound of contraction on space sets using a set of well-formed inference rules. Typical methods used (among others) to achieve state space reduction include boundary value analysis, covering arrays, and equivalence partitioning. The key assumption is that the state space reduction process must preserve the properties of and among the elements from the original state space [11,12,13].

Definition: Coverage refers to the extent to which a given verification activity has satisfied its objectives. Coverage measures can be applied to any verification activity, although they are most frequently applied to testing activities. Coverage is a measure, not a method or a test. As a measure, coverage is usually expressed as the percentage of an activity that is accomplished state space exercised or represented [4,9].

Testers of software prefer a metric that relates to coverage of the execution of source code, requirements, and its input domain. As example, requirements coverage analysis determines how well the requirements verified the implementation of the software requirements (IEC 61508, Section 3) [8], and establishes bi-traceability between the software requirements and the test cases. Structural coverage analysis determines how much of the code structure is linked to the requirements-based tests, and establishes traceability between the code structure and the test cases. Typically structural coverage criteria are divided into two types: data flow and control flow. Most structural coverage is control flow oriented; as such those will be discussed. For control flow criteria, the degree of structural coverage achieved is measured in terms of statement invocations, Boolean expressions evaluated, and control constructs exercised. The common types of coverage used today include statement coverage, decision coverage, condition coverage, single condition/decision coverage, multiple condition/decision coverage (MC/DC), t-way combinatorics, and multiple condition coverage. Table 1 below is an excellent reference on the ranking of coverage types.

Table 1. Types of structural coverage [9].

Coverage Criteria	Statement Coverage	Decision Coverage	Condition Coverage	Condition/ Decision Coverage	MC/DC	Multiple Condition Coverage
Every point of entry and exit in the program has been invoked at least once		•	•	•	•	•
Every statement in the program has been invoked at least once	•					
Every decision in the program has taken all possible outcomes at least once		•		•	•	•
Every condition in a decision in the program has taken all possible outcomes at least once			•	•	•	•
Every condition in a decision has been shown to independently affect that decision's outcome					•	•[8]
Every combination of condition outcomes within a decision has been invoked at least once						•

Table 1 gives the definitions of some common structural coverage measures based on control flow. A dot (.) indicates the criteria that applies to each type of coverage. The structural coverage measures in Table 1 range in order from the weakest, statement coverage, to the strongest, multiple conditions.

Note that the coverage measures above depend on access to program source code. CT, in contrast, can be a black box technique. Inputs are specified and expected results determined from some form of

specification. This aspect of CT is appealing because it is complementary to the "white box" coverage methods listed above.

The starting point is the state space of inputs. The following example taken from the Kuhn tutorial illustrates the principle [10]. Suppose there is a program that accepts two inputs, x and y, with 10 values each. Then the input state space consists of the $10^2 = 100$ pairs of x and y values, which can be pictured as a checkerboard square of 10 rows by 10 columns. With three inputs, x, y, and z, we would have a cube with $10^3 = 1{,}000$ points in its input state space. Extending the example to n inputs, would provide a (hard to visualize) hypercube of n dimensions with 10^n points. Exhaustive testing would require inputs of all combinations, but CT could be used to significantly reduce the size of the test set. Since, traditional coverage measures do not apply well to CT, the question remains what coverage measures are useful for CT? Kuhn et al. defines a number of measures in their 2010 NIST publication [9]. Namely, these include variable-value coverage, (t + k)-way combination coverage, and simple t-way combination coverage. Each of these measures has specific benefits depending on the goal of the testing. In this case, a desired outcome is to determine the practical limit of "stopping" t-way testing such that reasonable confidence can be claimed that all interaction faults have been uncovered or detected. Kuhn has collected statistics on a variety of domain applications and the data suggests the convergence of t-way testing is around 6-way interactions.

Alternatively, if credible empirical evidence is known in advance that observed failures in similar SW systems are triggered by t or fewer conditions, testing all t+1-way conditions is in some sense equivalent to exhaustive testing – for the class of interaction faults. There are other fault classes that may not be detectable by t-way testing (e.g., memory leaks, timing faults), but there are other testing methods that can be used for those fault classes. The t-way stopping rule is of course a heuristic rule, and should be validated before adopting. Finally, it is reasonable to ask how t-way combinatorial testing compares to other testing methods, such as testing with MC/DC criteria or mutation testing or random. Are there differences in terms of detection, cost, resources, and scalability? Are they complementary? Finally, no one software testing method is a silver bullet; but rather several testing methods used together and prudently can be far more effective toward CCF reduction and cost.

2. OBJECTIVE

The approaches, methods, and technologies described here within are mainly focused on testing actual software for embedded digital devices. That is, testing with actual inputs stimulating the software under test. The objective for this research is to develop a test specification to enable a study on the efficacy of t-way CT for embedded digital devices. To support this specification, questions will be developed to be tested by the study. These questions will be asserted in terms of statements that can be supported or refuted by the study.

- Question 1: Can t-way CT perform provide evidence that is congruent with exhaustive testing for an embedded digital device?
    - Under what assumptions and conditions for this claim to be true?
- Question 2: Can t-way combinatorial coverage criteria be comparatively contrasted to other coverage criteria (MC/DC, randomized) as to have some idea of the capabilities of CT?
- Question 3: Is t-way CT effective at discovering logical and execution-based flaws in nuclear power SW-based devices (device under test [DUT])?
- Question 4: Can t-way CT be facilitated by distributed computing and virtualized HW to reduce time on test, or accelerate testing?
- Question 5: Is t-way testing (in the context of Questions 1–4) cost effective for certifying safety critical SW in nuclear power applications?

2.1 SCOPE

The scope of this test specification is focused on t-way CT methods, technology, and supporting tools required to effectively carry out well-formed studies to answer Questions 1–5. It is recognized that dependencies in the critical path and early findings on the questions may limit the scope of the study, such as not locating a suitable DUT or finding that the Virginia Commonwealth University (VCU) smart sensor is not adequately representative of a NP smart sensor. In such cases, the program manager will be responsible for determining how the scope of the project is modified to answer the questions above.

3. PRIOR WORK

3.1 Bounded Exhaustive Testing

Exhaustive testing, testing a system's behavior for all combinations of inputs, is the ideal method for ensuring the dependability of a simple system. As was indicated Section 2.1, even for simple systems, the state space constituting all combinations of inputs and decision points is so large that exhaustive testing is unfeasible. Bounded Exhaustive Testing (BET) reduces this state space by applying boundaries on the test parameters of a system. An excellent survey and review of challenges with respect to CT methods is provided in [2]. While the majority of interest in BET has been for simpler systems [11,12] explored the viability of BET for a more complex dynamic fault tree modeling and analysis tool called Galileo. The principle of BET is that by observing only test cases that consider a specific number of inputs at any given time, the state space is reduced dramatically. Consider a system that takes 20 different events as arguments. This can be considered to have $2^{20} = 1{,}048{,}576$ possible input combinations. Now consider only the inputs combinations where six or less events may occur. The state space is reduced to 60,459 possible input combinations.

Recent work by Kuhn et al. [13] studied the effectiveness of CT in a variety of application domains, from critical systems (Traffic Collision Avoidance System, (TCAS) to web browsers). Their research has consistently showed that about 20–70% of software faults were triggered by single parameters, about 50–95% of faults were triggered by two or fewer parameters, and about 15% were triggered by three or more parameters. Thus, CT is effective in practice. Later they studied the fault interactions of large distributed systems, and discovered that the failure-triggering interactions of this kind of systems are mostly centered around 4 to 6 parameter interactions [14,15]. Kuhn et al.'s work shows that CT can be as effective as exhaustive testing in some cases, if all failures can be triggered by an interaction of 6 or fewer parameter values. Recent work by this group has considered how sequences can be tested via CT, rule-based systems, comparing t-way CT testing with random testing, and methods for generating test cases and oracles.

Bryce et al. made many contributions on test generation, failure diagnosis, and prioritization. Sherwood first introduced the CATS tool, which implemented a heuristic algorithm for pairwise coverage [16]. This group discussed two algebraic approaches to generate covering array, which could be used to build mixed covering array of Strength 2 and covering array of higher strength [17,18] introduced several greedy algorithms to construct covering arrays, mixed-level covering arrays, and biased covering arrays [19].

Cohen et al. worked on many areas of CT, including test generation, application, test prioritization, failure diagnosis, constraints, and evaluation. They first examined the need of variable strength covering array and proposed this new subject of research, after which they presented some computational methods, such as simulated annealing, to find variable strength array [20]. These researchers also explored a method for building covering array of strength three that combined algebraic constructions with computational search. This method leverages the computational efficiency and optimality of size obtained through algebraic constructions while benefiting from the generality of a heuristic search [21]. They used covering array to detect option-related defects and gave fault characterization in complex configuration spaces using the classification tree analysis [22].

Marinov [11] evaluated the effectiveness of BET by generating mutations of nine different software segments and observing the ability of BET to catch the mutants. Each of the software segments consisted of between 8 to 47 branches. This study confirmed that in all nine software segments, 90% of mutants were discovered using a bound of 7 or less branches in the testing scope. This density of faults at lower bounds suggests advantage of the BET method over that of random test-case generation, which equally observes test cases with large and small inputs. Coppit [12], ~~Sullivan~~ confirmed the scalability of this statement—that for more complex systems as well the density of faults is observed to be higher toward the lower bounds of inputs.

Performance limitations for BET are typically observed during the test case generation phase, while the test case evaluation phase goes much quicker. Recent work in optimizing the efficiency of BET includes implementing parallel algorithms for test-case generation [23,24], which has been shown to speed up the test generation process by 7.05x using a software test-case generator called Korat. Optimization of the Korat software for Graphics Processing Unit (GPU) implemmentation [24] has observed 17.46 speed up by contrast to Siddiqui's work [25], suggesting the multi-threaded GPU approach to have high potential for future work.

One of the primary limitations of the BET is that the statistical reliability of the system cannot be determined since the input values selected do not tend to be user domain profiles of inputs used in production, but this limitation is hardly unique to BET, many other testing techniques have this same limitation.

3.2 Combinatorial Testing as BET Method

As software is growing in size and complexity, testing the software that covers all the interactions between the data, environment, and the configuration is a challenging task. The studies conducted in National Institute of Standards and Technology (NIST) on software failures in Food and Drug Administration medical devices from 15 years of recall data concludes that the majority of software failures are due to interaction faults arising from the interaction of few parameters, mostly by two and three [12]. For National Aeronautics and Space Administration-distributed databases, 67% of the failures are triggered by a single parameter, 93% by 2-way interaction, and 98% by 3-way interaction. Several other applications studied also depicted similar results, shown in Butler and Finelli's 1993 article [3]. Applying the rule that the interaction between t or fewer variables are responsible for all the failures in software, testing all the t-way combinations of the variables can lead to "pseudo-exhaustive" testing of software. The combinatorial method, which involves selecting test cases that cover the different t-tuple combinations of input parameters, can lead to generating compact test sets that can be executed in considerably less time, while at the same time providing significant testability of the certain types of failures in software. Such failures are known as interaction failures because they are only exposed when two or more input values interact to cause the program to reach an incorrect result. CT is particularly suited to help detect problems like this early in the testing life cycle. The key insight underlying t-way CT is that not every parameter contributes to every failure, and most failures are triggered by a single parameter value or interactions between a relatively small number of parameters.

On the basis of experimental data collected by NIST on a variety of software applications, as shown in Butler and Finelli's 1993 article [3], it has been deduced that the cumulative percent of faults triggered in software reaches 100% when the number of parameters involved in the faults reaches six. This, in turn means that testing a software with all possible 6-tuple input parameter combinations can lead to tracking down all the bugs in the software. Exhaustive testing of four parameters with three values each covering all possible combinations will result in 81 test cases. If the combinatorial method that limits to a pairwise interaction level of parameters is used, the number of test cases can be reduced to nine. The combinatorial method thus renders a drastic reduction in test cases without compromising on the quality of testing [13].

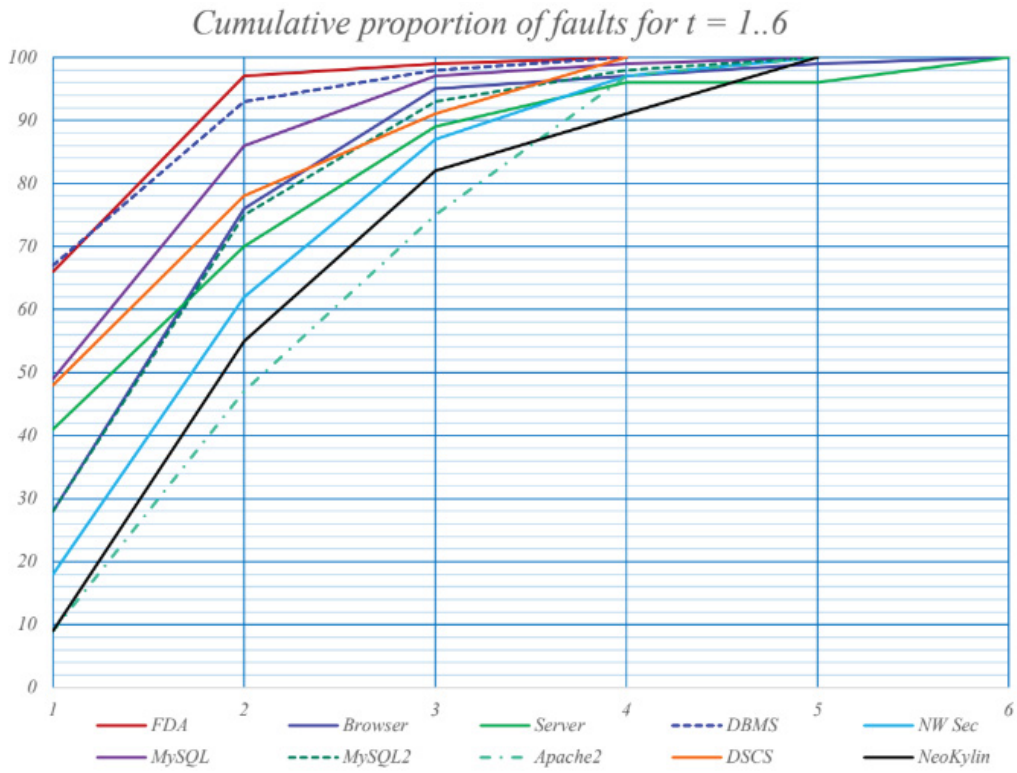

Figure 2. Cumulative proportion of faults for T (number of parameters) = 1...6 [13].

For combinatorial test set generation, the two mainly used combination arrays are covering arrays and orthogonal arrays. Covering arrays CA (N, t, k) are arrays of N test cases, which has all the t-tuple combinations of the k parameters covered at least a given number of times (which is usually 1). Orthogonal arrays (OA) (N; t, k) are covering arrays with a constraint that all the t-tuple combinations of the k parameters should be covered the same number of times. The major elements of a combinatorial test model are parameters, values, interactions, and constraints [26].

The first step for creating a test model is to identify all of the relevant parameters, which should include the user and environment interface parameters and the configuration parameters. The second step is to determine the values for these parameters. Using the entire set of values for all the parameters would lead to unmanageable test suites and testing. Hence, to confine the values of the parameters to a necessary and tractable set, apply the various value partitioning techniques like equivalence partitioning, boundary value analysis, category partitioning, and domain testing. As the third step, interactions between the parameters must be analyzed in order to generate an efficient set of test cases. Defining the valid parameter interactions and their strengths in the test model can aid in avoiding test cases involving interactions between parameters that actually never interact in the software and also in prioritizing test cases for closely interacting parameters. Specifying the "constraints" on the interactions, which define the set of impossible parameter interactions, is also vital for obtaining the expected software coverage [27].

R. Kuhn and V. Okun's work on "Pseudo Exhaustive Testing for Software" [13] discusses the concept of integrating combinatorial methods with model checking and presents the results of applying this technique on an experimental system. Model checking can be used for automatic test case generation. The requirement to be tested is identified and a temporal logic formula is formulated in such a way that the requirement is not satisfied. This formulation of the negative requirement will be the test criterion, which will cause the software model to fail, thus causing the model checker to generate counterexamples that can be used as test cases. By using t-way coverage of the variables as the test criterion, the

combinatorial test cases can be derived. Temporal logic expressions in the form AG(v1 & v2 & ... & vt -> AX !(R)), which directs that for the input variable combination (v1, v2…vt), the condition R should be false in the next step, has to be fed as input to the model checker tools. Thus, the model checker will generate counter examples that cover all the variable combinations that satisfy R. The experiment conducted by Kuhn et al. in using a symbolic model checker to create pairwise to 6-way combinatorial test cases for a Traffic Collision Avoidance System gives supporting results. It shows a 100% error detection rate with 6-way combinatorial coverage of inputs. Although there were more counterexamples generated by the model checker than the actual t-way combinations needed, the number of redundant test cases were found to reduce as the input interaction coverage (t) increases.

R. Kuhn (NIST)) and J. Higdon's (U.S. Air Force) research work on extending the application of CT to event driven systems, described in the paper "Combinatorial Methods for Event Sequence Testing" [27], also proves to be noteworthy for systems of the type found in NPP. Some faults in the software become activated only when there is a particular sequence of events happening—a very relevant condition related to NPP operations. Sequence covering arrays can be used to test all of the t-way order of t events in a software. The basic concept of sequence covering is that if there is a 2-way event testing, there should be a test case with x...y, such that y event occurs after x event. And there should also be a reverse order of the event occurrence y...x where x occurs after y. Testing the forward and reverse order of occurrences for all the events with respect to all other events can help when detecting most of the event-driven failures in the software. The research paper provides mathematical proof that the number of tests only grows logarithmically with respect to the number of events. This combinatorial sequence-based testing helps when tracking down all the event sequence-based issues in software, thereby improving the efficiency of testing.

A lot of research has also been done in the field of studying and developing various algorithms for covering array test suite generation, including greedy algorithms, and heuristic methods. Bryce et al.'s greedy algorithm for test case generation in Bryce and Colbourn's 2006 article [28], which takes user inputs on the priorities of the interactions to be covered and which also allows for seeding of fixed test cases into the test set, is identified as another important work in the field of CT.

4. REPRESENTATIVE SMART SENSOR DEVICE TO BE TESTED

4.1 VCU Open Source Smart Sensor

The VCU Smart Sensor is a barometric pressure and temperature sensing device that originates from the VCU Unmanned Aerial Vehicles (UAV) Laboratory. The device is derived from a Part 23 (non-safety related) VCU ARIES_2 Advanced Autopilot Platform [7,9], which consists of mature design and code, and has over 10,000 hours of tested flight time. The VCU Smart Sensor is comprised of both hardware and software articles, which are described more in-depth in the following sections. The definitive descriptions of the VCU Smart Sensor software are the VCU Software Requirements Specification and Software Design Document—both found on the VCU Github repository. The VCU Github repository (see link below) contains all software, documentation, fault files, and testing setups. All software for the VCU Smart Sensor is written in GNU11 C programming language for the application code and compiled and executed by the GNU Compiler Collection (GCC) Version 7.3 to run on top of the ChibiOS Version 17.6.4 Real-Time Operating System (RTOS). The VCU Smart Sensor aims to aid in the qualification and licensing of Embedded Digital Devices (EDDs) in the Nuclear Digital I&C Domain, where the tests performed thereafter will serve as a benchmark for the originally planned CCF measurements and tests.

4.1.1 General Matter

The VCU Smart Sensor software consists of several threads executing periodically in a real-time operating system. The following generalities are mentioned to place the software development process and documentation in context.

- Development - Several developmental tools were evaluated and used to generate the VCU Smart Sensor and the associated supporting documentation files. The primary development tools used include the GNU11 C programming language, the GNU GCC Version 7.3.
- Code support - the graphic visualization tools code2flow, Doxygen, and Graphviz. The entire structure and functionality of the VCU Smart Sensor is comprised of source code, written in the GNU11 C programming language, which is readily available to the user for further inspection or external testing. The GNU GCC compiler is the standard compiler used to compile and execute the application code.
- Function maps - The associated function-call maps, which are found in the Software Requirements System (SRS) and Software Design Document (SDD) in the VCU Smart Sensor Github repository, are generated directly from the VCU Smart Sensor application code using the online interactive code to flowchart converter, code2flow. Additionally, the open source tools Doxygen and Graphviz were used to create visual call graphs of the software. Doxygen is the standard tool for generating documentation from annotated application code sources, and Graphviz is an open source graph visualization software.

4.1.2 User Documentation

The following documents are provided for the user for more in-depth information:

- The VCU Software Requirements Specification Document (Github)
- The VCU Software Design Document (GitHub)
- Product Specifications Document (Datasheet) for ST STM32F405xx and SM32F407xx ARM Cortex-M4, 2016.
- Reference Manual for ST STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 Advanced ARM®-Based 32-Bit MCUs, 2017.
- Product Specifications Document (Datasheet) for TE Connectivity MS4525DO PCB Mounted Digital Output Transducer, Combination Differential, Gage, Absolute, Compound, & Vacuum Temperature and Pressure Sensor with I^2C or SPI Protocol, 2016.
- Product Specifications Document (Datasheet) for TE Connectivity Sensor Solutions MS4525DO PCB Mounted Digital Output Transducer, 2016.
- I2C and SPI Interface Specifications Document (Datasheet) for TE Connectivity Sensors, Interfacing to MEAS Digital Pressure Modules, 2016.
- Product Specifications Document (Datasheet) for TE Connectivity Sensor Solutions MS5611-01BA03 Barometric Pressure Sensor, with stainless steel cap, 2017.

4.2 VCU SMART SENSOR COMPONENTS

4.2.1 Hardware Architecture

The hardware architecture of the VCU Smart Sensor is shown in Figure 3. The main components of the hardware architecture include the STM32F4 ARM Cotrex-M4 168 MHz microcontroller, the MS4525DO absolute and differential pressure sensors, onboard memory components, multiple peripheral options, dedicated buses for networking, and components for the communications interfaces. Since the VCU Smart Sensor is based on the pre-existing and vigorously tested VCU ARIES_2 Advanced Autopilot Platform, using the ARM-based processor was an easy decision. Additionally, the ARM STM32FM407 System on a Chip is a very widely used chip in the embedded systems world and in safety-related embedded systems, and there is a vast array of supporting documentation for the STM32FM407 microcontroller. The ARIES_2 was originally designed as a generic hardware and

software platform, with a specific emphasis on ease of extendibility as a general computing device for embedded applications that require sensory input. More generally, ARIES_2 was designed for further platform modifications and testing.

The sensor heads for pressure and temperature measurements are integrated through the I2C bus. The physical sensor head, Measurement Specialties MS4525DO offers both absolute and differential pressure sensing capabilities. The MA4525DO sensor head was chosen during design so that the user could choose between using either of the different sensor types, absolute or differential. Additionally, a static pressure sensor, the Freescale MP3H6115A, is included for altitude measurement, as well as a dynamic pressure sensor, the Freescale MP3V5004DP, included for airspeed measurement with sub-knot precision. The differential pressure sensor type is used currently to calculate the altitude for the VCU Smart Sensor, which is then mapped into an digital value using the associated Analog to Digital Conversion channels.

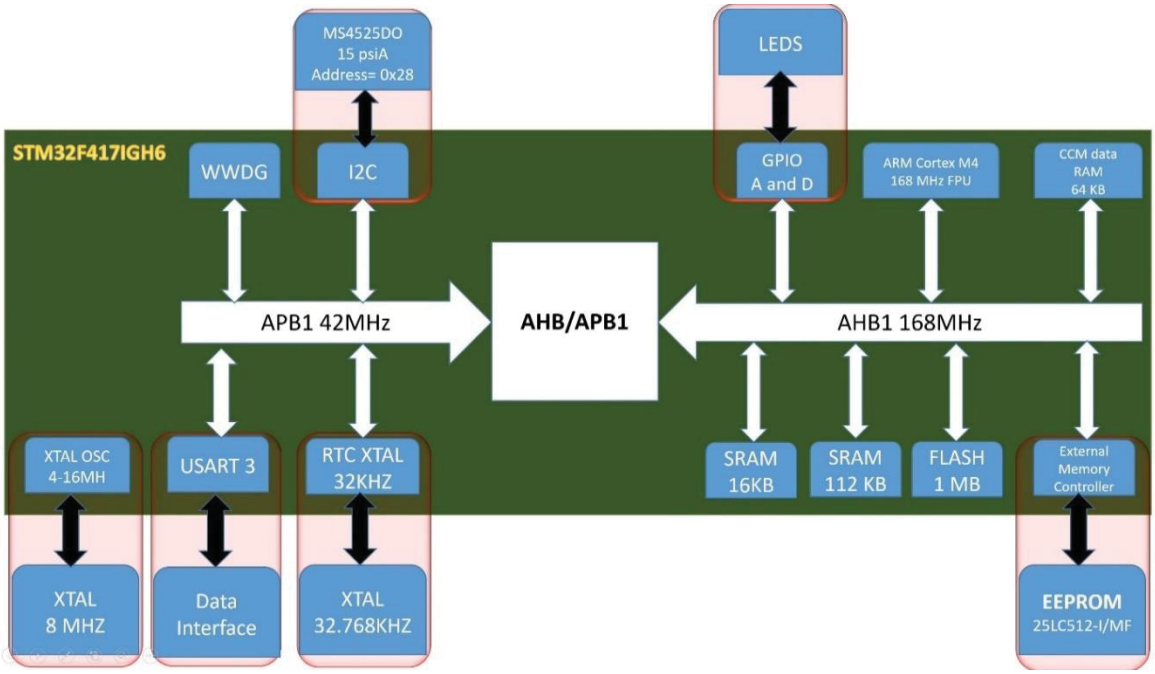

Figure 3. Hardware architecture of the VCU Smart Sensor.

The bus-bridge in the center of Figure 3 separates the two types of operations within the hardware architecture of the VCU Smart Sensor. The right side of the AHB/APB1 bus-bridge includes components used for high-speed operations, including the processor core and memory components, instruction and data buses, and memory buses, all connected through the 168 MHz AHB1 bus. The left side of the AHB/APB1 bus-bridge includes components used for low-speed operations, including external and internal communication peripherals such as the serial UART and I2C interfaces, for the communication of data and instructions. The MS4525DO sensor heads are included in the low-speed operations components. All low-speed operation components are connected through the 42-MHz APB1 bus. The external power supply requirements are covered in Section 4.2.1.2, External Power Interfaces.

4.2.2 Software Stack Model

The software stack model for the VCU Smart Sensor is seen in Figure 4. The VCU Smart Sensor software incorporates the ARIES_2 software, which has an integrated configuration system that allows for the runtime configuration of most low-level and high-level drivers, for onboard peripheral configuration.

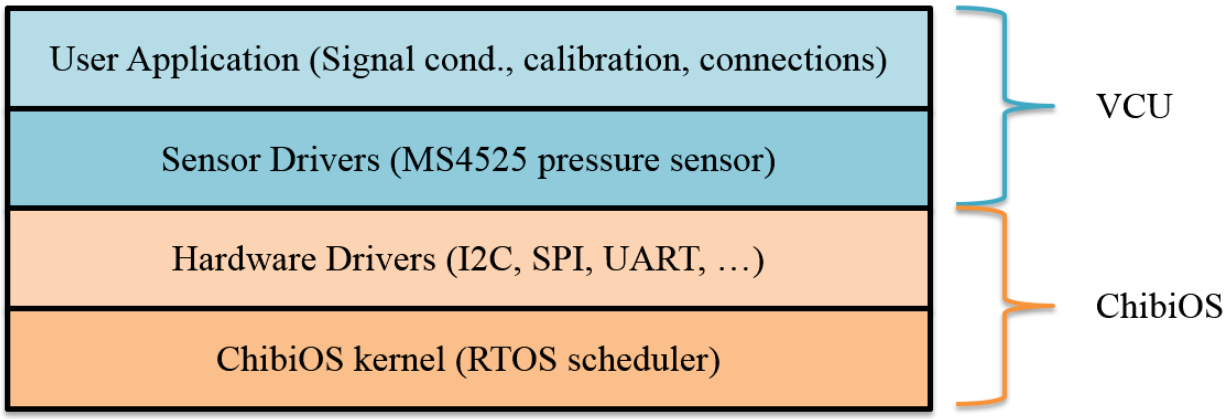

Figure 4. VCU smart sensor software stack model.

The software layers of the VCU Smart Sensor include the user application layer and the sensor drivers layer, which are original articles generated by the VCU UAV Laboratory, and modified for this current project for adjustment to the appropriate context and functionality required. The sensor drivers layer includes the drivers for the MS4525DO pressure sensor. The MS4525DO pressure and temperature transducers are managed by the user application layer to obtain pressure and temperature information, including altitude, speed, and offset. The VCU Smart Sensor is built around the ChibiOS real-time operating system (RTOS). ChibiOS provides the hardware drivers layer and the ChibiOS kernel layer, which include the drivers used for I2C and serial UART communication, and the RTOS scheduler, respectively.

All of the various software layers communicate with each other via I2C communication, and are managed by the full stack RTOS ChibiOS. The combination of the application code and ChibiOS ensure the proper scheduling and execution of all periodic tasks within the system, which are handled by a priority-based queue system. The software stack model has been designed and adjusted through years of implementation to ensure that it is a modular software design that may be adjusted or modified for future work as necessary.

4.2.3 Real-Time Operating System – ChibiOS

The software provided by VCU is built around the ChibiOS complete development environment for embedded applications. The ChibiOS development environment includes a RTOS, a hardware abstraction level (HAL), various peripheral drivers, support files and tools. ChibiOS is a free, open source RTOS, which includes many standard APIs used for most common peripherals. Additionally, ChibiOS supports the STM32FM4 and all onboard peripherals, which was the primary reason for the original design choice of using the ChibiOS development environment. Since the VCU Smart Sensor originates from the VCU ARIES_2 Advanced Autopilot Platform, which is also built around the ChibiOS development environment, all protocols for the accurate interfacing of software using ChibiOS within the VCU Smart Sensor are already in place and have been tested extensively. The architecture model for ChibiOS is seen in Figure 5.

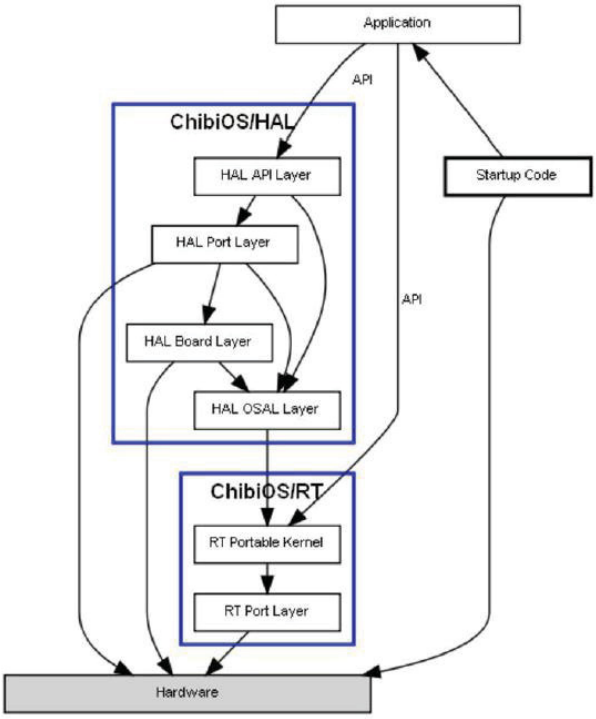

Figure 5. ChibiOS architecture model.

ChibiOS is a static rate-based multi-threaded RTOS, which allows for deterministic behavior. The ChibiOS architecture is composed of an application model, startup code, ChibiOS/RT, and ChibiOS/HAL. The application model is a single application with multiple threads, consisting of a trusted runtime environment and multiple threads that share the same address. The original RTOS scheduler has been replaced by a thread-based protocol, which generates threads during platform initialization. The generated threads are awoken as needed, either by various VCU Smart Sensor functions, or on a periodic basis using internal timers, depending on the thread. The application and operating system are linked together into a single memory image (a single program). The startup code is executed after the reset, and is responsible for core, stack, and runtime initializations, as well as the calling of the main function of the application. ChibiOS/HAL is the hardware abstraction layer, which includes a set of device drivers for the peripherals most commonly found in microcontrollers.

In ChibiOS, the startup code is provided with the operating system for the various supported architectures and compilers. Scatter files and any other necessary files required for system startup are also provided with the operating system. ChibiOS is meant to be used in 8, 16, and 32-bit microcontrollers starting from 2 KB of RAM and 16 KB of Flash. Additionally, ChibiOS can be ported to any CPU architecture as long as it includes a real stack pointer. More information on the ChibiOS open source development environment may be found at http://www.chibios.org/dokuwiki/doku.php.

4.3 High Level Description of Smart Sensor

The VCU Smart Sensor will run as a single interface application. On startup, the following will occur:

- Input/Output (I/O) Initialization – A user-defined American Standard Code for Information Interchange (ASCII)-formatted input file for the sensor head will be fed into the Arduino from a host computer via a universal serial bus (USB) connection. This will be discussed more thoroughly in Section 4.2.5, Testing Interface.

- Thread Initialization – The ChibiOS Kernel will be started and the main program will become a thread. Various threads exist for specific functions within the source code.
- Serial Initialization – The Serial I/O interface will be started between the host computer and the Arduino via a USB connection. This will be discussed more thoroughly in Section 4.2.5, Testing Interface.
- Timer Initialization – The system timer will be initialized and started, counting the system time from system startup.

4.3.1 Data Flow

Figure 6 shows the program data flow of the software components of the VCU Smart Sensor in its testing environment context, including the threads and communication protocols used to transmit data between modules.

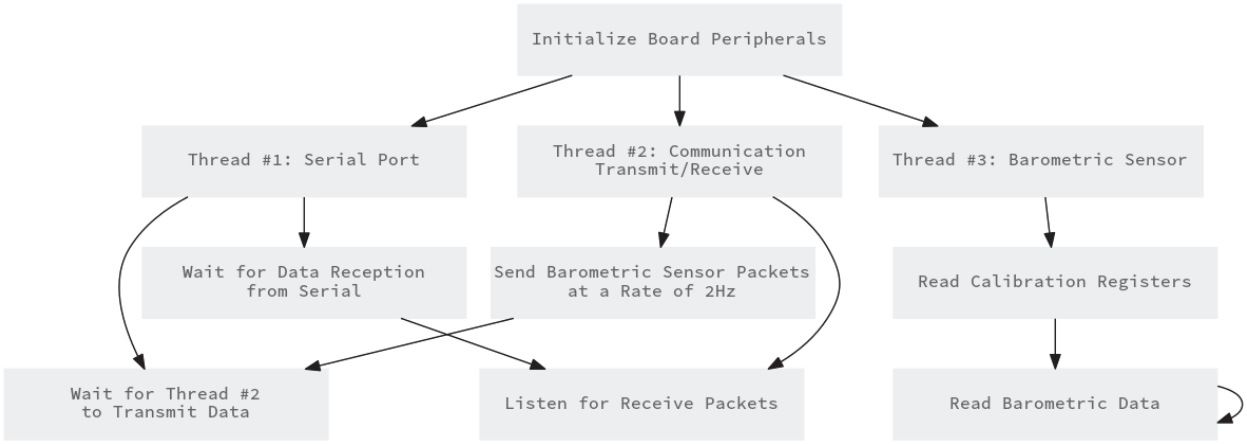

Figure 6. Program data flow.

As seen in Figure 6, the board peripherals are initialized prior to any other actions. Following the board peripheral initializations, three threads shall be generated:

- Thread 1: Serial Port
- Thread 2: Communication Transmit/Receive
- Thread 3: Barometric Sensor.

Following the generation of the three respective threads, various data transmission, receive, and wait functions shall be utilized to read/write barometric data/packets (at a rate of 2 Hz), calibration registers, and receive packets using serial communication protocols.

4.3.2 Software Interfaces

The VCU Smart Sensor shall interface with software to enable the user to communicate externally via software. The user shall interact with the PC-based console window. The PC-based console window shall communicate with the VCU Smart Sensor via a serial port (UART) on the VCU Smart Sensor. Data will be formatted as an ASCII text transmitted via RS-232 protocol to the PC-based command line prompt shell. Another software interface is the Windows/Linux operating system. Specific API calls shall be employed during the programming and operation of the VCU Smart Sensor.

4.3.3 Communications Interfaces

All I2C communication is performed using standard I2C protocol; that is, all I2C communication is event-driven and uses write-on requests. All communication within the VCU Smart Sensor, including the

different layers of the software stack and the implementation of high-level communications functions for the API, which shall be performed using the standard I2C protocol. Only one I2C bus is used within the VCU Smart Sensor, which shall perform all peripheral communication and driver communication for hardware interfacing purposes. The communication interface to the VCU Smart Sensor from an external user will be a serial port using a serial monitor application at 57600 Baud (bits per second).

4.3.4 External Interface Design

This section describes the five types of external interfaces: user interfaces, hardware interfaces, software interfaces, communications interfaces, and test and debug port interfaces.

4.3.5 User Programming Interface

Programming methods currently exist on Windows or Linux operating systems. The testing method preferred by VCU uses the Linux operating system, where the methods have been tested on the Ubuntu 16.04 LTS 64-bit version, and all user programming operations are performed via the command line interface. Users are encouraged to program the VCU Smart Sensor using the ST-Link Utility.

The steps for user programming using the ST-Link Utility are as follows:

- The ST-Link software for programming may be downloaded at http://www.st.com/content/st_com/en/products/development-tools/software-development-tools/stm32-software-development-tools/stm32-utilities/stsw-link009.html. The user must unzip and run the stlink-winusb-install.bat file, followed by a machine restart after the installation finishes.
- From the start menu, the user must run the STM32 ST-Link Utility.
- From the bar at the top, the user must click “target,” then click “connect.” The text at the bottom should say “SWD frequency 4 MHZ, device family STM32F405xx, etc.”
- From the target menu, the user must click “program” and “verify.”
- The user must click “browse” from the “File Path” menu, and navigate to the compiled aries.bin file in the “build” folder of aries_rt.
- The user must ensure that “verify while programming” or “verify after programming” is selected. The user must also select “reset after programming.” The user then must click “start.” An example to this point is shown in Figure 7.

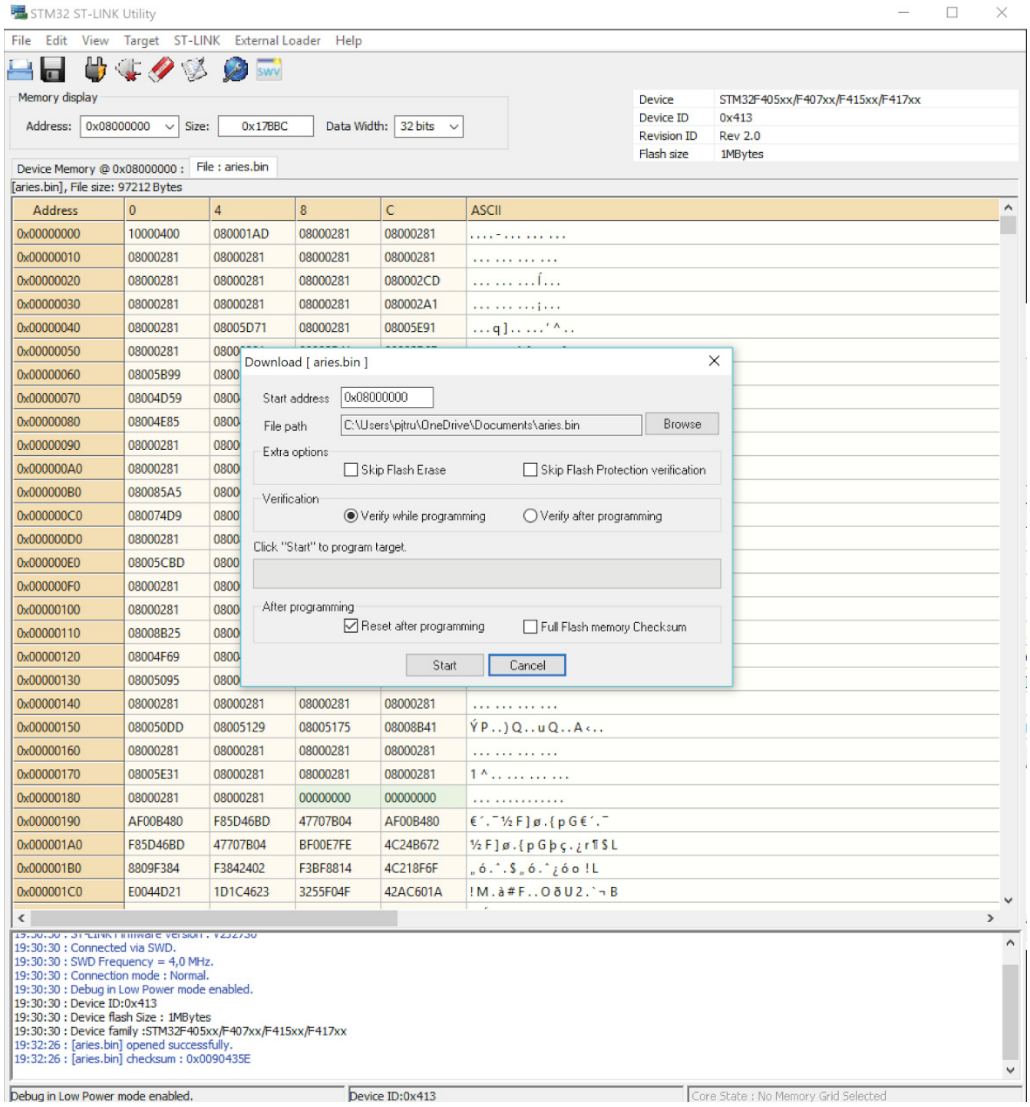

Figure 7. ST-Link utility programming process example.

- The program window should exit and the bottom of the screen should say “Verification… OK.”
- If the user reaches this point, then the VCU Smart Sensor is successfully programmed. An example to this point is shown in Figure 8.

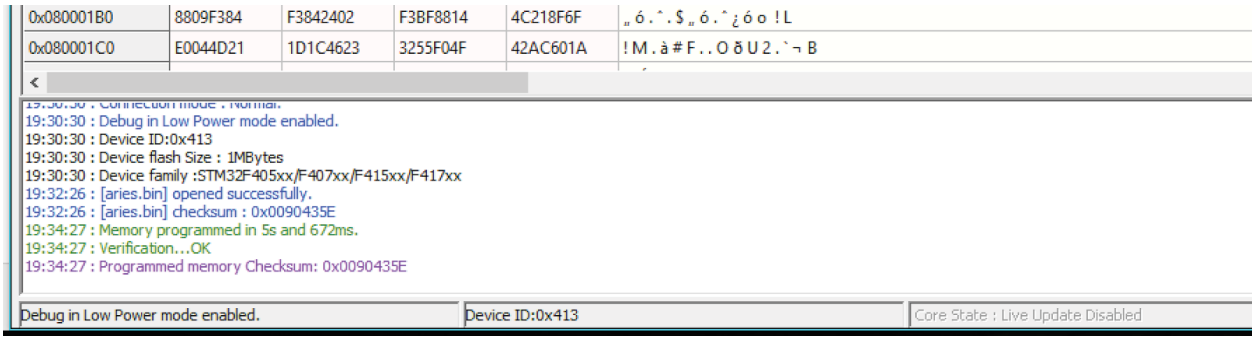

Figure 8. ST-Link utility programming success example.

After programming is complete, the purple light on the VCU Smart Sensor should begin to blink. If the purple light does not blink, the user must unplug the programmer from the VCU Smart Sensor and power cycle the smart sensor.

4.3.6 Operational User Interface

The VCU Smart Sensor is configured to continuously convert pressure and temperature samples, and to transmit the data over serial communication. The "Small Red" board included is a Sparkfun Future Technology Devices International (FTD)I Basic 3.3V, which converts the serial signal used by the VCU Smart Sensor to USB that can be used by the host computer. A cable shall be included which connects the FTDI to the port labeled "MDM" on the VCU Smart Sensor. The cable should be a 6-position connector with three pins populated. The user shall plug this cable into the FTDI adapter such that the black wire is connected to the position labeled "GND" in the FTDI adapter. The other two pins should be connected to the "RXI" and "TXO" pins on the FTDI adapter.

The user shall connect the VCU Smart Sensor and FTDI adapter to the host computer with a microUSB and miniUSB cable, respectively. The red light on the VCU Smart Sensor should turn on, and the blue and purple lights should blink continuously. The user shall open the serial port using a serial monitor application at 57600 Baud (bits per second). On Ubuntu Linux, the user may use the command line prompt "Screen/dev/ttyUSBx 57600" from the terminal where "x" is the name of the serial adapter. The user can view the available serial adapters by typing "ls/dev." Usually the device will appear as "/dev/ttyUSB0." If the user has connected correctly, they should see pressure, temperature, and Kalman-filtered pressure displayed as key value triples of the format "pre:1.000000,tem:10.000000,kf_pre:4.799696," for example. Pressure shall be displayed in Pascals, temperature shall be displayed in degrees, Celsius, and Kalman-filtered pressure shall be displayed in Pascals.

4.3.7 User Data Logging Interface

The VCU Smart Sensor shall provide a means for the logging of raw sensor data, the viewing of the data, and the downloading of the data. This interface shall be implemented via a serial port (UART) on the VCU Smart Sensor. Data will be formatted as ASCII text transmitted via RS-232 protocol to a PC-based command line prompt shell. The commands for interrogating the data are as follows:

- Initiate Data Stream
- Stop Data Stream
- Change Rate of Data Stream.

4.3.8 Debug and Test Port Interface

The VCU Smart Sensor shall provide a debug and testing port to allow for real-time monitoring of execution behavior of the VCU Smart Sensor. The VCU smart sensor will use ARM CoreSight Debug and Trace debug standard for this purpose. At a minimum, the VCU Smart Sensor will use the Serial Wire Debugger port for communicating test and debug information to commercial debug environments. A variety of debugger SW tools exist for the testing and debugging of the VCU Smart Sensor via Serial Wire Debugger. The options include the GNU GDB (GNU Debugger) (https://www.gnu.org/software/gdb/), the ARM Keil Microcontroller Development Kit Toolset (http://www2.keil.com/mdk5/), the ARM CoreSight Debug and Trace – Serial Wire Debugger (https://developer.arm.com/products/system-ip/coresight-debug-and-trace/coresight-architecture/serial-wire-debug), and the Atollic Serial Wire Viewer (http://blog.atollic.com/cortex-m-debugging-introduction-to-serial-wire-viewer-swv-event-and-data-tracing).

4.3.9 External Power Interfaces

The software requires that the testing board uses a 3.3-V input voltage, which is currently provided via a USB connection, in order to perform specific limits calculations. Details on the miniUSB and microUSB connectors are given in Section 4.2.1.2, Operational User Interface.

4.3.10 Hardware Interfaces

The VCU Smart Sensor shall interface with several hardware articles during operation. The first hardware interface is the serial modem, used to communicate data information between the PC-based console window and the VCU Smart Sensor. The PC-based console window is responsible for a combination of inputs to the Arduino (subsequently transmitted to the VCU Smart Sensor) and readouts from the VCU Smart Sensor. The PC-based console window shall communicate with the Arduino via a USB connection and with the VCU Smart Sensor via a serial port (UART) on the VCU Smart Sensor. The Arduino shall communicate with the VCU Smart Sensor via a serial port (UART) on the VCU Smart Sensor. Data will be formatted as an ASCII text transmitted via RS-232 protocol to the PC-based command line prompt shell. The same serial port shall be used to output data to the host computer for logging purposes. The output data will be formatted as an ASCII text as well, including pressure, temperature, and Kalman-filtered pressure values.

Only one I2C bus is used within the VCU Smart Sensor, which shall perform all peripheral communication and driver communication, for hardware interfacing purposes. Both digital barometric sensors shall interface to the main processor over the single I2C bus, and the communication shall be handled by the underlying operating system of the VCU Smart Sensor, ChibiOS. Specific protocols are already in place for the accurate communication of data between the transmitter and microcontroller within the VCU Smart Sensor, due to the original software protocols used within the VCU ARIES_2 Advanced Autopilot Platform. These protocols have been tested extensively. A microUSB and miniUSB connector shall be used with the FTDI adapter to power and operate the VCU Smart Sensor. Further details on the miniUSB and microUSB are given in Section 4.2.1.2, Operational User Interface.

5. TEST METHODOLOGY AND PROCESS

This section outlines a general framework for designing a set of studies to address test objectives.

5.1 Prioritization of Test Objectives

The test methodology to be developed shall be designed to address the five test objectives listed in Section 2 of this document. The following definitions describe the set of desirable goals that a comprehensive software testing method (in the spirit of the NRC testability definition) endeavors to achieve.

- Goal 1: The method is unambiguous and can be applied to a wide variety inputs data types, logical expressions, and configurations in most (if not all) types of safety critical software
- Goal 2: The method has a basis on rigorous mathematical foundations, with well-defined assumptions and constraints
- Goal 3: The number of tests to achieve “bounded exhaustive” testing is tractable (e.g., ideally linear or logarithmic) with respect to the number of terms (and interactions) in the expressions
- Goal 4: **All** the variables interactions, conditions, and configurations (or terms) in the expressions can expressions are observable
- Goal 5: Complicated expressions can receive more testing than simple expressions
- Goal 6: The method is shown to have a high probability of detecting errors.

Whether all of these goals can be achieved in total or partially for combinatorial t-way testing is an open question, especially with constraints on resources, time, and cost. The purpose of the test methodology is to provide objective evidence on some these goals. Specifically, Goals 3, 4, and 6 are of particular interest.

The restated research test objectives in context of goals are given in Table 2 below.

Table 2. Test objective and goals.

	Test Objective	Supports Goals	Requires
1	Can t-way combinatorial testing provide evidence that is congruent with exhaustive testing for an embedded digital device?	Goals 3 and 4	Representative DUT SW, tools to conduct t-way combinatorial testing, design of experiments (studies) to achieve comparative results.
2	Can t-way combinatorial coverage criteria be comparatively contrasted to other coverage criteria (MC/DC, randomized) as to have some idea of the capabilities of combinatorial testing?	Supports Goals 2 and 5	Representative DUT SW, in addition to conducting t-way combinatorial testing must conduct testing with respect to MC/DC criteria
3	Is t-way combinatorial testing effective at discovering logical- and execution-based flaws in nuclear power SW-based devices?	Supports Goal 6	Representative DUT SW, faulted versions of the DUT SW, Design of Experiments study to determine statistical power of the testing
4	Can t-way combinatorial testing be facilitated by distributed computing and virtualized HW to reduce time on test, or accelerate testing?	Supports Goals 3 and 4	Representative DUT SW, faulted versions of the DUT SW, distributed computing clusters, processor to function mapping (HADOOP), maybe virtualized HW.
5	Is t-way testing (in the context of Questions 1–4) cost effective for certifying safety critical SW in nuclear power applications?	Supports Goals 1 and 2	All of the above, PLUS manpower estimates in time and effort, resources required to estimate certification costs.

Table 2 provides the details to examine goals in terms of "things" required to answer the questions of the objectives. These "things" roughly relate directly to expected resources, level of effort, and person-effort. For this research effort, test Objective 1 and 3 have been identified as essential, in that order. Others, while important must be placed on a second tier of priority. Accordingly, the following subsections will focus on test concepts for addressing test Objectives 1 and 3.

5.2 Test Objectives 1 and 3

- T1: Can t-way CT provide evidence that is congruent with exhaustive testing for an embedded digital device?
- T3: Is t-way CT effective at discovering logical- and execution-based flaws in nuclear power SW-based digital devices?

5.3 Preliminary Concepts

To fully develop the idea behind this study, we first describe some essential material related to state space and interaction t-way CT. Efficient generation of test suites to cover all t-way combinations is a difficult mathematical problem (NP hard). Additionally, contemporary software in most embedded digital devices is a combination of data types representing continuous variables (fixed

point, floats), integers, Booleans which have possible values in a very large range. For effective reduction to a testable state space, the range of these values must be mapped to a much smaller range, possibly a few values. This is usually done though equivalence partitioning and sampling methods—another non-trivial problem. Most evident of all is the problem of determining the correct result that should be expected from the system under test for each set of test inputs. This is the oracle problem—how to determine when something is correct. Fortunately, most of these challenges have been addressed to the point where practical methods and tools supporting t-way CT allow credible reduction of the input and state space. Nonetheless, there are still open research issues associated with t-way CT, and they are actively being addressed, notably the creation of effective test oracles.

Beginning with the generation of tests, generally, the number of t-way combinatorial tests that will be required is proportional to $v^t \log n$, for n parameters with v possible values each. The key parameter in these equations is v and t. Keeping v and t small reduces the "parameter state space." t is a function of the logical behavior of the software. v is a function of the data type space in terms of range of the data type. Normally, creating partitions for each v is minimally sufficient for testing. For example, a variable whose range was -10 to +10 might create a partition with the set {-10, -1, 0, 1, +10}—five representative values. This case provides the min/max values, values close to 0, and 0. To exhaustively test this range, the full span of values would be needed is (21). The issue in the design of this experiment is that the full span of variables cannot be used with a large range for comparative exhaustive testing. Another way must be found. One idea is to look at how the variable is used in the decision logic of the program. If the variable is a part of a condition or guard expression, then selecting a range of values on the condition and on either side of the condition might be sufficient for testing interactions. This is called boundary value analysis, to select test values at each boundary and at the smallest possible unit on either side of the boundary, for three values per boundary. The intuition, backed by empirical research, is that errors are more likely at boundary conditions because errors in programming may be made at these points. Additionally, the boundary analysis partition can now be expanded to include more representative elements. This becomes the basis for comparing to an "exhaustive set." The bounded partition is defensible because every important element of the set is represented at least once and the smallest units are used at the boundaries.

5.3.1 Number of Tests

From [9], The goal to find covering arrays is to find the smallest possible array that covers all configurations of t variables. If every new test generated covered all previously uncovered combinations, then the number of tests needed would be:

$$\frac{v^t\binom{n}{t}}{\binom{n}{t}} = v^t \tag{3}$$

Since this is not generally possible, the covering array will be significantly larger than v^t but still a reasonable number for testing. It can be shown that the number of tests in a t-way covering array will be proportional to:

$$number\ of\ tests \triangleq v^t \log n \tag{4}$$

Where v is the value span of the input variables or parameter (n)

n is the number of inputs parameters

t is the number of interactions between parameters.

First, note that the number of tests grows exponentially with the interaction strength t, but logarithmic with the number of input parameters (n). The value span of v determines the base of the value, which can have a growth effect of the number tests. Table 3 below provides an indication on the relationship between v and t and the number of tests. Since its contributions is logarithmic, n was ignored. Although the number of tests required for high-strength CT can be very large (as illustrated

below), with advanced distributed processing clusters and mapping software (like gridunit or Hadoop) it is not out of reach.

Table 3. Relationship between v and t for covering array tests.

v↓ t⟶	2	3	4	5	6
2	4	8	16	32	64
6	36	64	256	1024	4096
10	100	1000	10,000	100,000	1,000,000
12	144	1728	20736	248832	2,985.984
16	256	4096	65536	1048576	16,777.216

For illustrative purposes suppose the following subset of variables are taken from the VCU smart sensor:

- 20 Boolean variables - each variable takes on (T,F)
- 10 continuous time variables (float) – by Boundary Value Analysis (BVA)and equivalence partitioning each variable is represented by 12 values
- 10 integer variables - by BVA and equivalence partitioning each variable is represented by 10 values.

What would be the expected number of tests for a 4-way cover array?

- $BOOL\ number\ of\ tests\ \triangleq v^t \log n = 2^4\ Log40 = 26$
- $INT\ number\ of\ tests\ \triangleq v^t \log n = 8^4\ Log40 = 6562$
- $FLOAT\ number\ of\ tests\ \triangleq v^t \log n = 12^4\ Log40 = 33220.$

Total = 39,808 tests.

Percentage of tests with respect to exhaustive testing (with respect to the defined equivalence partitions)

$$\frac{Number\ of\ CT\ tests}{v^{N\_bool}+v^{N\_INT}+v^{N\_float}} \triangleq \frac{2^4}{2^{20}} + \frac{8^4}{8^{10}} + \frac{12^4}{12^{10}} \triangleq .000019\% \quad (5)$$

This low-state space coverage result can be interpreted as follows. If the equivalence and BVA partitions are well-formed for the program, and the covering arrays generate tests that cover all combinations, then a very percentage of well-formed test vectors is needed to perform as well as brute force exhaustive testing—the essence of bounded exhaustive testing. This is the power of the test. The key assumptions are that reduction methods like BVA and equivalent partitions are well-formed, and 4-way interactions are sufficient. In the case where 4-way interactions is found not to be sufficient, then performing t+1 (5) interactions is required. This would roughly have 6-fold increase in the number of tests to ~282,000.

For a study, where the purpose of the study is to affirm or refute, the capacity of a given SW testing method to achieve bounded exhaustive testing or (pseudo exhaustive testing) then increasing t and v to levels well beyond where no faults are observed, could require significant computational resources, time and effort. This should be noted early as a significant factor in the study.

5.3.2 Conceptual Experiment Process

Figure 9 below shows conceptually the experiment process required to achieve the objectives. The first step is to define all of the relevant parameters required for the test objectives. In this case the critical parameters are t, v, n, and the faulted versions of code; there are more parameters (like time, IDs) but these are critical. Each of these parameters must be pre-analyzed (e.g., by Boundary Value Analysis) to determine their equivalence partitions. With tool assistance, a "covering array" of the parameter space is used to define the list of experiments; this can be done parametrically (one factor at a time), or by Design of Experiment methods. The list of "covering" test vectors is then used to define the experiments. One-way experiments can be designed is by varying the t variable for a given set of experiments, increasing t incrementally. The same can be done with the v parameter. These experimental test vectors are applied to the DUT. For each experiment executed, the DUT must start from a known good state. This usually requires the experiment automation instrumentation to issue reset before each experiment. Once the DUT is operational, the test vectors are applied. The outcomes of the DUT are observed by the test oracle or by assertions (maybe code based). The oracle makes notations on pass/fail, collects data for statistics, etc. The outcomes will belong to three sets: detected/undetected faults, coverage metric (percentage of covering array), and a metric related to the percentage of state space examined. This process is repeated for each "fault seeded" version of the code. The process continues until there are no more variations on the parameter sets for any fault-seeded versions OR the computational complexity exceeds the processing power to carry out the experiments. Data is post processed from the outcome space to determine if the experiments yielded evidence to support (or refute) the claims (test objectives). Implementation of this experiment method or process can be accomplished a number of ways. The key point is that the experiment process implementation must be designed to accurately collect data for the test objectives. The following subsection discusses a step-by-step outline on issues and choices for experimenters.

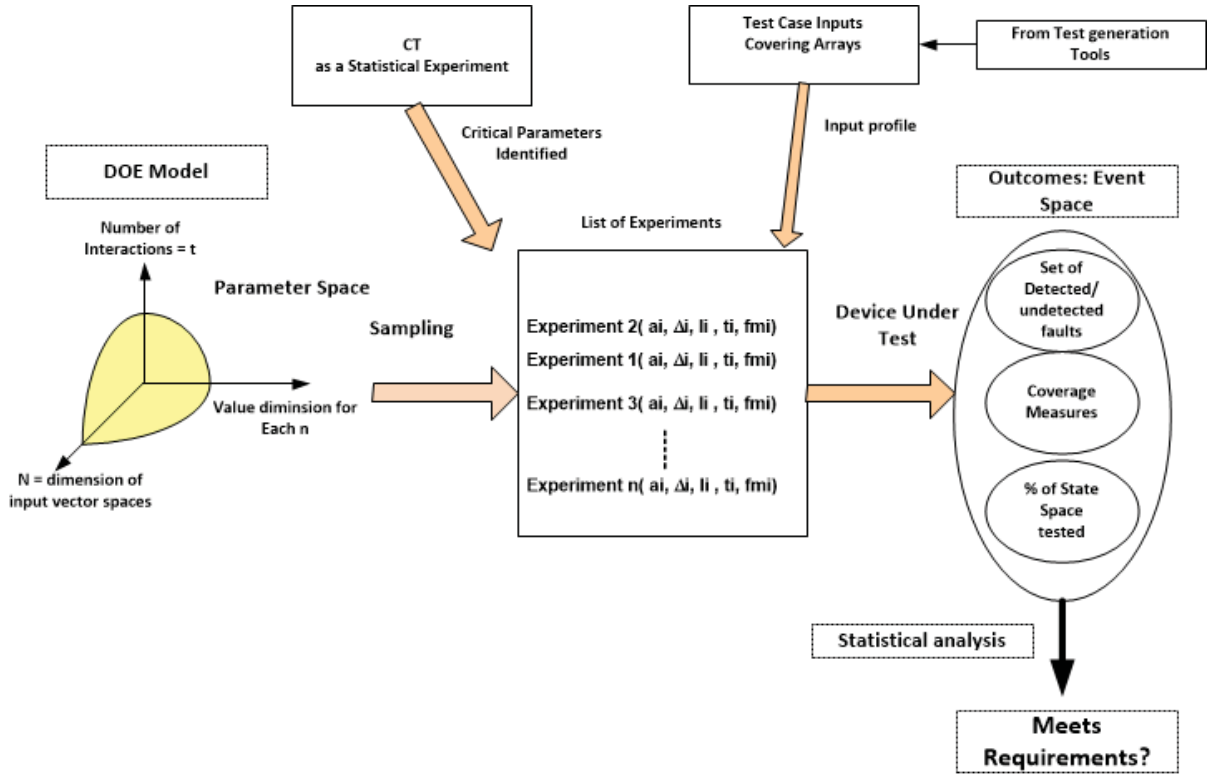

Figure 9. Conceptual view of the bounded exhaustive testing process.

5.3.3 Step by Step Outline

Step 1: The first step for creating an input test model is to identify relevant input parameters, which should include the user and environment interface parameters and the configuration parameters—the *n* parameter. For the smart sensor this includes as a minimum all input data variables, calibration variables, intermediate function variables, filtering algorithm variables, I/O handling variables, and device configuration settings. This set is around 35 to 40 parameters of various data structure types. The exact number will be determined via engineering analysis of important parameters. RTOS functions and parameters, drivers, and lower level service functions are excluded at this time. This step only identifies the candidate parameters.

Step 2: The second step determines the values for these parameters—the *v* parameter. Using the entire set of values for all the parameters would lead to infeasible test suites and testing. Hence, to confine the values of the parameters to a necessary and tractable set, the various value partitioning techniques like equivalence partitioning, boundary value analysis, category partitioning and domain testing need to be applied. This step requires some analysis and tool support to define the ranges and domains of the parameter set for the test objectives.

Step 3: As the third step, interactions between the parameters must be analyzed in order to generate an efficient set of test cases—this is the *t* value. Defining the valid parameter interactions and their strengths in the test model can aid in avoiding test cases involving interactions between parameters that actually never interact in the software and also in prioritizing test cases for closely interacting parameters. Specifying the constraints on the interactions is necessary to create a searchable state space. As noted in Kuhn et al.'s 2010 article and Kuhn and Okum's 2006 article [10,13], the input data range could be constrained by the problem domain or implementation aspects combined with the data representation format. As an example, speed measurement always belongs to the input domain $s \geq 0$. If the speed input is a signed integer, then the input domain is reduced by half. As another example, if the speed sensor maximum output is 90, $s \in [0,90]$, represented by a 16-bit unsigned integer format, and the software input is a 32-bit unsigned integer, then the input domain is reduced by 216, as these combinations will not be produced by the sensor.

Step 4: The fourth step generates test cases for the DUT, which is one of the more challenging aspects of SW testing, and it is no different for combinatorial t-way testing. Most methods use a *combination strategy* which selects test cases based on some combinatorial strategy [29]. Combinational strategies involves four elements: (1) covering array specifying the specific kind of test suite to be used; (2) seeding to assign some specific user test cases in advance; (3) considering constraints in the test generation; and (4) using methods to generate test cases. The general approach, once the above steps have been concluded, is to build a set of test vectors that support an experiment list. Note most of combination strategy generator methods are supported by open source tools (such as NIST ACTS), but require expert domain knowledge to effectively use. While the elements of combinational strategy are encompassed in most tools, the two most important elements are covering arrays and test sequence generation. Most testing can be accomplished with these two methods.

- *Covering array* - The two mainly used combination arrays for combinatorial test set generation are covering arrays and orthogonal arrays. Covering arrays CA (N, t, k) are arrays of N test cases, which has all the t-tuple combinations of the k parameters covered at least a given number of times (which is usually 1). Orthogonal arrays OA (N; t, k) are covering arrays with a constraint that all the t-tuple combinations of the k parameters should be covered the same number of times. The major elements of a combinatorial test model are parameters, values, interactions, and constraints [26]. Even using covering arrays, a large number of combinations will be required, but far fewer than fully exhaustive testing. For the small example in Kuhn et al.'s 2010 article [10], exhaustive coverage would have required 230,400 combinations, but all 4-way combinations were covered with 1,450, all 5-way with 4,347, and all 6-way with 10,902.

- *Seeding* - Seeding means to assign some specific test cases or some specific schema in testing. Seeding is used to guarantee inclusion of favorite test cases by specifying them as seed tests. Seeding has two practical applications. (1) Seeding allows explicit specification of important combinations. For example, if a tester is aware of combinations that are likely to be used in the field, the tester can specify a test suite to contain these combinations. (2) It can be used to minimize change in the test suite when the test domain description is modified and a new test suite regenerated.
- *Constraints*. Constraints occur naturally in most systems. The typical situation is that some combinations of parameter values are invalid. Existence of constraints increase the difficulty in applying CT, as most existing test generation methods have limited ways to deal with constraints. With the NIST ACTS tool one can specify constraints, which inform the tool not to include specified combinations in the generated test configurations from the covering arrays. ACTS supports a set of commonly used logic and arithmetic operators to specify constraints.
- *Test sequence generation* - Test case generation for t-way CT is a very active research area, and thus there are many options for generating test sequences. The following website provides a list of tools that are used to generate testing sequences (http://www.pairwise.org/tools.asp). Greedy algorithms have been the most widely used method for test suite generation for CT. They construct a set of tests such that each test covers as many uncovered combinations as possible. Recent research has focused on using model checking with test sequence generation to automatically generate tests and oracles together. Model checking is applied to test generation in the following way. One first chooses a test criterion, that is, decides on a philosophy about what properties of a specification must be exercised to constitute a thorough test. When the model checker finds that a requirement is inconsistent, it produces a counterexample. These counterexamples are used as stimulus to the SW.

Step 5: The fifth step is generating oracles. Even with efficient algorithms to produce covering arrays, test sequences, the oracle problem is critical. Testing requires both test data and results that should be expected for each data input. Much care should be given early and often on the "whats and hows" of the oracle; that is define what you want the oracle to do, and how it is going to do it. Approaches to solving the oracle problem for CT include:

- *Crash testing*: The easiest and least expensive approach is to simply run tests against SUT to check whether any unusual combination of input values causes a crash or other easily detectable failure. This is essentially the same procedure used in "fuzz testing," which sends random values against the SUT. Crash testing is the weakest form of oracle testing.
- *Embedded assertions*: An increasingly popular "light-weight formal methods" technique is to embed assertions within code to ensure proper relationships between data, for example as preconditions, post-conditions, or input value checks. Tools such as the Java modeling language or Frama-C, can be used to introduce very complex assertions, effectively embedding a formal specification within the code. The embedded assertions serve as an executable form of the specification, thus providing an oracle for the testing phase. With embedded assertions, exercising the application with all t-way combinations can provide reasonable assurance that the code works correctly across a very wide range of inputs. Of course the DUT SW language must accept embedded assertions, and this requires access to the source code. It is not known at this time how difficult it would be to instrument the VCU Smart Sensor code with embedded assertions. It would have to be recompiled with the Frama-c compiler.
- *Model based test generation* uses a mathematical model of the SUT and a simulator or model checker to generate expected results for each input. If a simulator can be used, expected results can be generated directly from the simulation, but model checkers are widely available and can also be used to prove properties, in addition to generating tests. What makes a model checker particularly valuable is that if the claim is false, the model checker not only reports this, but also provides a "counterexample" showing how the claim can be shown false. If the claim is false, the model checker

indicates this and provides a trace of parameter input values and states that will prove it is false, which can be submitted to the DUT for fault verification and identification.

- *Model Based Simulation* – With tools like Simulink, complex models of the algorithms (representing SW functions) can be functionally captured and simulated to provide a comparison to the device under test. This is called model in the loop simulation. Of course, the model has to be validated.

Step 6: The sixth step develops faulty versions of code. To test the effectiveness of the fault detection capabilities of the testing methods, faulty code is needed. Generated faulty versions can be accomplished several ways. First, real bugs from the development and operational history of the software can be used as faulty versions. Second, mutated versions of the code can be created using code mutation tools. Both ways are acceptable means to producing reference cases.

Step 7: The seventh step executes the tests. Executing the tests require an automated test environment where test vectors are submitted to the DUT and results are cataloged. Most of the time these automated test environments are built from commercial instrumentation environments such as LabVIEW. The key aspect of these environments is to capture all operational steps necessary to collect data to support the test objectives. The data includes things like sequencing test cases with time stamps and IDs so that results can be matched to inputs. Additionally, data processing requires that test results be marshaled in a way that allows faulty versions be tracked so that test results can show how many tests were required to detect the fault, how much state space was exercised, etc. To support coverage metrics we need to keep track of the number of test case interactions that have been processed need to be tracked so that comparative analysis to exhaustive testing can be made. One of the golden rules of CT from [10] *start testing using 2-way (pairwise) combinations, continue increasing the interaction strength t until no errors are detected by the t-way tests, then (optionally) try t+1 and ensure that no additional errors are detected*. As with other aspects of software development, this guideline is also dependent on resources, time constraints, and cost-benefit considerations.

Step 8: The eighth step analyzes the results. After all test cases have been executed, then post-analysis can proceed to compute various metrics on the efficacy of the testing. Since a comparative analysis of t-way CT to exhaustive testing is desirable, multiple t-way interactions are needed. Also, a simpler function is needed rather than the entire code base of the Smart Sensor to achieve some comparative results, or the computational complexity will overwhelm the processing resources. Selection and analysis of metrics at the beginning of the experiment is important to ensure that experiment can support calculation of the metrics at the post-analysis phase.

5.3.4 Functional Test Environment Perspective

In the preceding sections, (Section 5.3.3) it discussed the process steps of how to realize or "build" a BET study. In this section, an implementation perspective is presented of how to realize a test environment setup. Note what is described in this section is just one of many ways to realize the process outlined in Section 5.3.3. Figure 10 presents a functional test environment diagram with respect to tools, systems and components needed to support the BET experiment process. The calls out numbers on the diagram roughly represent the "steps" associated with the process outline in Section 5.3.3. The first phase is to parse the code to reveal the variables instances, data structures, parameters, and constants the code embodies. This is typically found in the *.map file from the compiler. The variables of interest are entered into the NIST ACTS and CCM tools to produce groupings of test vectors with respect to the experiment process. That is, experiment parameters (t and v) are parametrically varied to produce a table of t-way tests. Through the configuration and setup instrumentation, the DUT is configured to operate in manner that is consistent with its operating requirements. From the ACTS tool, the test vectors are systematically loaded into the DUT or alternatively through a test sequencer, which loads the test vectors into DUT. For each test vector, the DUT responds to that specific test vector with a set of readouts (outputs). These outputs may be combinations of pure outputs, states, or conditions. The state monitoring function observes these readouts and performs time stamping, instance tagging, and ordering to facilitate

post analysis and fault location identification. These readouts are then forwarded to the oracle where decisions are made on the validity of the readout. The oracle validity data field is then appended (or associated) to the readout. Finally the readouts, oracle validity results are marshaled into a data base (like MySQL) so that queries can be made on the data for post analysis. In the upper-right corner of Figure 10 is an alternative DUT configuration. Realizing that working with a small set of smart sensor devices (possibly 1 or 2), will constrain the efficiency of the experiment process. VCU with Imperas Technologies have developed a high-fidelity virtualized HW model of the smart sensor [30] that can run on the OVPsim platform simulator [30]. The advantage to this approach is that multiple VM instances of the smart sensor can be distributed across a compute cluster or a server cluster to significantly accelerate testing as was done in [31]. The disadvantage to this approach is that the "experiment process management" complexity is much greater that the single article test environment. It would be judicious to first implement the "single" article test environment while planning to move to a distributed test environment.

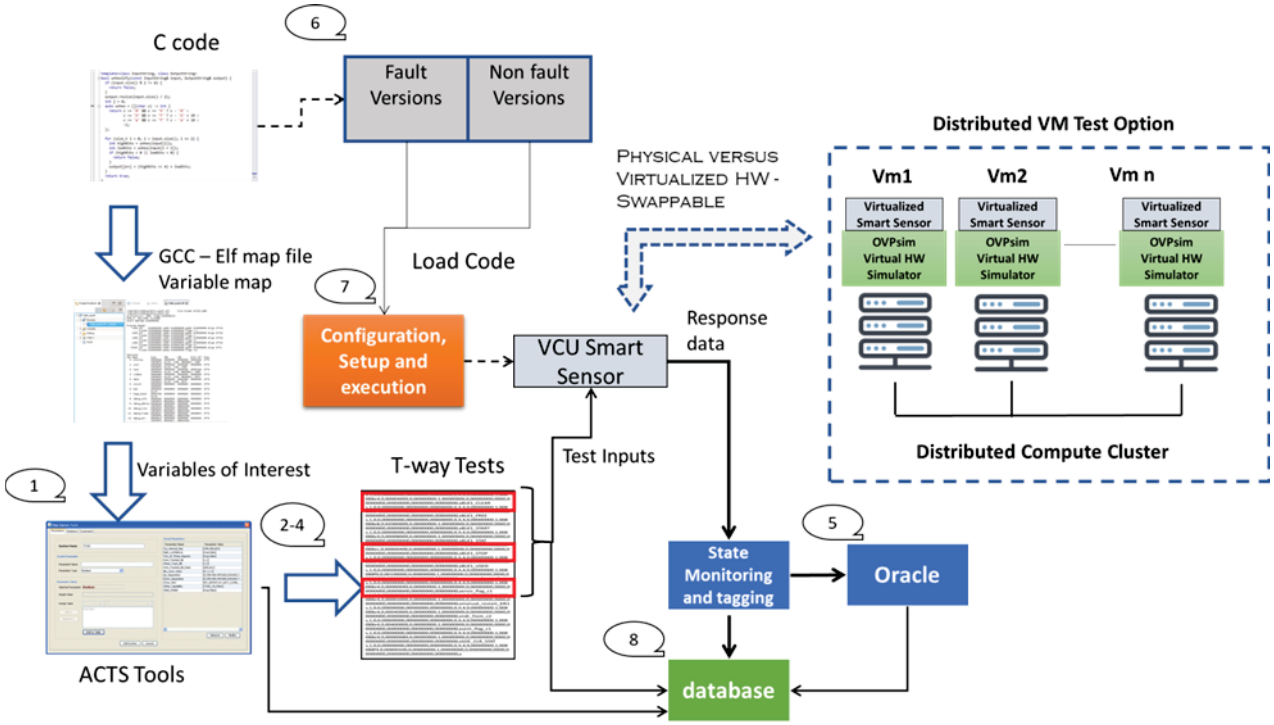

FIGURE 10 EXPERIMENTAL FUNCTIONAL DIAGRAM

6. TOOLS AND RESOURCES

Numerous commercial and open source tools are available to assist the experimenter in conducting the study. These can be found at (http://www.pairwise.org/tools.asp). However, the tools produced by NIST would be a good choice for a variety of reasons. Namely, they have been used in the analysis of critical systems. Key Resources to Support Test

The resources listed below are the essential items, tools, and resources to support the functional experimental diagram in figure 10. This list is not considered definitive nor complete, but suggests the essential items and resources to conduct an experimental evaluation.

1. Device under Test Software – VCU Smart Sensor SW code basis.
2. ST-Link debug software or similar (see section 4.3.8)
3. ST STM32F405 – ARMM4 Cortex processor development board.
4. Ubuntu 16.04 LTS 64 Linux
5. GNU11 C programming language, the GNU GCC Compiler Version 7.3.
6. The VCU Software Requirements Specification Document (Github)
7. The VCU Software Design Document (GitHub)
8. LabVIEW development environment
9. Host computers, servers and database software
10. USB-8451 I²C/SPI Interface Device (maybe)
11. Requirements for generating a test oracle – Specification of required functionality.
12. Instrumentation to observe results (data logging) – State monitoring function
13. Optional: Virtualized smart sensor model, OVPsim, compute clusters, test management SW, etc…

NIST Combinatorial Testing tools:

1. ACTS Covering array generator – basic tool for test input or configurations;
2. ACTS Input modeling tool – design inputs to covering array generator using classification tree editor; useful for partitioning input variable values
3. ACTS Fault Location Tool – identify combinations and sections of code likely to cause problem
4. Sequence covering array generator – new concept; applies combinatorial methods to event sequence testing
5. ACTS CCM Combinatorial coverage measurement – detailed analysis of combination coverage; automated generation of supplemental tests; helpful for integrating c/t with existing test methods

7. TEST PLAN

To execute the experiments, a test plan should be created. Since this is a research-based effort, full compliance to a standard is not required; however, the key elements of IEEE 829-2008 [33] would be a good choice to incorporate (Standard for Software Test Documentation), 829 is an IEEE standard that specifies the form of a set of documents for use in defined stages of software testing. The main 829 articles that would be useful for this study would be:

- Test objectives
- SW test items
- Assumptions
- SW features to be tested

- SW features not to be tested
- Technical approach or process
- Event outcome space, and pass/fail criteria
- When to "stop" criteria
- Test analysis deliverable.

8. POTENTIAL CHALLENGES AND NEXT STEPS

The potential challenges to the proposed research are listed below:

- Inability to determine if the VCU Smart Sensor is representative of a typical nuclear power plant smart sensor.
- Complexity of SW testing exceeds given time and effort to conduct sufficient experiments to support or refute test objectives.
- Development of oracles – somewhat unknown at this point with respect to the best approach to follow.
- Unfamiliarity with the tools – currently, very little experience is available with the tools required to facilitate the experiment.
- Time schedule – the given time (9 months) to prepare, conduct, execute, and process data is a very rapid pace.

Next steps are to fully develop the details of the experimental process steps in the context of the tool support and the VCU Smart Sensor software. The first step is to determine what functions in the Smart Sensor (provided it is representative) will be selected for testing. From this starting point, each step in the above process needs to be fully examined in the context of supporting the test objectives. Decisions will be made in the next month or so about what tools to use, the maximum extent of parameter experiment space, what oracle designs are appropriate, the amount of data expected to be processed, etc. Most of these decisions can be resolved quickly once the functions to be tested are identified, and the extent and dimensions of the testing are considered in context of test objectives.

9. REFERENCES

[1] U.S. Nuclear Regulatory Commision, *Guidance for Evaluation of Diversity and Defense-In-Depth in Digital Computer-Based Instrumentation and Control Systems Review Responsibilities*, Accession No. ML16019A344, BTP 7-19, Rev. 7, August 2016.

[2] P. Ammann and J. Offutt, *Introduction to Software Testing.* Cambridge: Cambridge University Press, 2008.

[3] R. W. Butler and G. B. Finelli, "The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software," *IEEE Trans. Softw. Eng.*, Vol. 19, No. 1, pp. 3–12, 1993.

[4] J. B. Goodenough and S. L. Gerhart, "Toward a Theory of Test Data Selection," *IEEE Trans. Softw. Eng.*, Vol. SE-1, No. 2, pp. 156–173, 1975.

[5] International Atomic Energy Agency, *Protecting against Common Cause Failures in Digital I & C Systems of Nuclear Power Plants*, Vienna: International Atomic Energy Agency, 2009.

[6] C. Elks, A. Tantawy, R. Hite, A. Jayakumar, and S. Gautham, "Defining and Characterizing Methods, Tools, and Computing Resources to Support Pseudo Exhaustive Testability of Software Based I&C Devices," INL/EXT-18-51521, Idaho National Laboratory, 2018.

[7] J. M. Voas and K. W. Miller, "Software Testability: The New Verification," *IEEE Softw.*, Vol. 12, No. 3, pp. 17–28, May 1995.

[8] 8 IEC 61508, "Functional Safety," Section 3, International Electrotechnical Commission.

[9] K. J. Hayhurst, J. J. Chilenski, and L. K. Rierson, *A Practical Tutorial Decision Coverage on Modified Condition*, NASA/TM-2001-210876, NASA, 2001.

[10] D. R. Kuhn, Y. Lei, and R. N. Kacker, "Practical Combinatorial Testing," *IT Professional*, Vol. 10, No. 3, pp. 19–23, 2010.
[11] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard, *An Evaluation of Exhaustive Testing for Data Structures*, MIT CSAIL, 2003.
[12] D. Coppit, J. Yang, S. Khurshid, W. Le, and K. Sullivan, "Software assurance by bounded exhaustive testing," *IEEE Trans. Softw. Eng.*, Vol. 31, No. 4, pp. 328–339, 2005.
[13] D. R. Kuhn and V. Okum, "Pseudo-exhaustive testing for software," in *Software Engineering Workshop, 2006. SEW'06. 30th Annual IEEE/NASA*, 2006, pp. 153–158.
[14] D. R. Kuhn and M. J. Reilly, "An investigation of the applicability of design of experiments to software testing," in *27th Annual NASA Goddard/IEEE Software Engineering Workshop, 2002. Proceedings.*, pp. 91–95.
[15] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software fault interactions and implications for software testing," *IEEE Trans. Softw. Eng.*, Vol. 30, No. 6, pp. 418–421, June 2004.
[16] G. B. Sherwood, "Effective Testing of Factor Combinations," in *3rd International Conference on Software Testing, Analysis, and Review (STAR94)*, 1994, pp. 151–166.
[17] C. J. Colbourn, S. S. Martirosyan, G. L. Mullen, D. Shasha, G. B. Sherwood, and J. L. Yucas, "Products of mixed covering arrays of strength two," *J. Comb. Des.*, Vol. 14, No. 2, pp. 124–138, March 2006.
[18] R. C. Bryce and C. J. Colbourn, "A density-based greedy algorithm for higher strength covering arrays," *Softw. Test. Verif. Reliab.*, Vol. 19, No. 1, pp. 37–53, Mar. 2009.
[19] R. C. Turban, "Algorithms for covering arrays," Arizona State University, 2006.
[20] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, C. J. Colbourn, and J. S. Collofello, "A variable strength interaction testing of components," in *27th Annual International Computer Software and Applications Conference. COMPAC 2003*, 2003, pp. 413–418.
[21] M. B. Cohen, "Designing Test Suites for Software Interaction Testing," The University of Auckland, 2004.
[22] C. Yilmaz, M. B. Cohen, and A. A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *IEEE Trans. Softw. Eng.*, Vol. 32, No. 1, pp. 20–34, January 2006.
[23] S. Misailović, A. Milićević, N. Petrovic, S. Khurshid, and D. Marinov, "Parallel Test Generation and Execution with Korat," in *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2007, pp. 135–144.
[24] J. H. Siddiqui and S. Khurshid, "PKorat: Parallel generation of structurally complex test inputs," in *Proceedings - 2nd International Conference on Software Testing, Verification, and Validation, ICST 2009*, 2009, pp. 250–259.
[25] A. Celik, S. Pai, S. Khurshid, and M. Gligoric, "Bounded Exhaustive Test-Input Generation on GPUs," *Proc. ACM Program. Lang.*, Vol. 1, 2017.
[26] W. Visser, K. Havelund, G. Brat, S. Park, and L. Flavio, "Model Checking Programs," *Autom. Softw. Eng.*, Vol. 10, No. 2, pp. 1–36, 2002.
[27] D. R. Kuhn, J. M. Higdon, J. F. Lawrence, R. N. Kacker, and Y. Lei, "Combinatorial methods for event sequence testing," in *Proceedings - IEEE 5th International Conference on Software Testing, Verification and Validation, ICST 2012*, 2012, pp. 601–609.
[28] R. C. Bryce and C. J. Colbourn, "Prioritized interaction testing for pair-wise coverage with seeding and constraints," *Inf. Softw. Technol.*, Vol. 48, No. 10, pp. 960–970, 2006.
[29] M. Grindal, J. Offutt, and S. F. Andler, "Combination testing strategies: A survey," *Softw. Test. Verif. Reliab.*, Vol. 15, No. 3, pp. 167–199, September 2005.
[30] F. E. Derenthal IV, C. R. Elks, T. Bakker, and M. Fotouhi, "Virtualized Hardware Environments for Supporting Digital I & C Verification," in *11th Nuclear Plant Instrumentation, Control and Human-Machine Interface Technologies*, 2017, pp. 1658–1670.
[31] D. Aarno and J. Engblom, "Software and System Development using Virtual Platforms: Full-System Simulation with Wind River Simics," *Elsevier Science*, 2014.
[32] A. Duarte, G. Wagner, F. Brasileiro, and W. Cirne, "Multi-environment software testing on the grid," in *Proceeding of the 2006 workshop on Parallel and distributed systems testing and debugging PADTAD 06*, 2006,

Vol. 2006, pp. 61–69.
[33] IEEE Computer Society, Software & Systems Engineering Standards Committee., Institute of Electrical and Electronics Engineers., and IEEE-SA Standards Board., "IEEE standard for software and system test documentation," Institute for Electrical and Electronics Engineers, New York, New York, USA, 2008.