INL/EXT-18-51521

Defining and Characterizing Methods, Tools, and Computing Resources to Support Pseudo Exhaustive Testability of Software Based I&C Devices

Carl R. Elks, Ashraf Tantawy, Rick Hite, Smitha Gautham, Athira V Jayakumar, Chris Deloglos

September 2018

U.S. Department of Energy
Office of Nuclear Energy

INL/EXT-18-51521

Defining and Characterizing Methods, Tools, and Computing Resources to Support Pseudo Exhaustive Testability of Software Based I&C

Carl R. Elks, Ashraf Tantawy, Rick Hite, Smitha Gauthem, Athira Jayakumar, and Chris Deloglos
Dept. of Electrical & Computer Engineering
Virginia Commonwealth University
601 West Main Street, Room 203
P.O. Box 843072
Richmond, Virginia 23284-3072

September 2018

Prepared for
Idaho National Labs
U.S. Department of Energy
Office of Nuclear Engineering

ABSTRACT

Under the Department of Energy's Light Water Reactor Sustainability Program, within the Plant Modernization research pathway, the Digital 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 in view of the very large number of combinations of inputs and sequences of device states for a typical I&C device. However, many of these combinations are not unique in the sense that they represent the same state space or in that that they represent 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 acceptable test methods, needed tools (existing or new), and computing resources to conduct this type of testing. This information will be used in the next milestone deliverable of the project to develop a test specification for I&C device testability. The test specification will, in turn, be used in a future phase of this project to demonstrate digital qualification with respect to common cause failure.

Table of Contents

List of Figures

1. Introduction

Much of the instrumentation and control (I&C) equipment in operating U.S. nuclear power plants (NPPs) is based on very mature, primarily analog technology that is trending toward obsolescence. Currently, all new reactor designs depend in large part on newer digital I&C systems and their software, and most planned I&C upgrades at existing units will employ software based digital technology. The assessment of the dependability of software in NPP safety systems is an essential and challenging aspect of the safety justification for digital systems. Experience in other industries has shown that digital technology can provide substantial benefits in terms of performance, reliability, and maintainability. For example, the avionics industry has seen large increases in sensor systems reliability (e.g., 1-2 orders of magnitude) by migrating to advanced sensors in both safety and non-safety applications. Because of the high demand for digital functionality in high-volume industries, the industrial I&C marketplace is dominated by digital technology such that it is increasingly difficult to acquire instrumentation that is not equipped with an embedded digital device intended to enhance its performance, reliability, and flexibility. Nevertheless, the nuclear power industry has been slow to adopt digital technology primarily because of regulatory uncertainty, implementation complexity, and limited availability of nuclear-qualified vendors and products.

In a draft Regulatory Issue Summary (RIS), the U.S. Nuclear Regulatory Commission (NRC) identified concerns about the impact of greater use of equipment with embedded digital devices [1]. Specifically, the NRC staff states that increased use of such devices "may increase a facility's vulnerability to a CCF". The prevailing NRC guidance on Diversity and Defense-in-Depth (D3) [2] identifies two design attributes that are acceptable for eliminating CCF concerns: (1) diversity or (2) testability (specifically, 100% testability). Either solution can result in high costs and remaining licensing uncertainty (how much diversity is enough? how to ensure test coverage of every possible sequence of device states?). Indeed, for increasingly complex digital devices, it is not practical to achieve exhaustive testing with conventional methods due to the enormous number of test vectors (e.g. all pairs of state and inputs) needed to effectively approach 100% test coverage for the device [3]. Consequently, many utilities and reactor designers have limited or avoided more extensive use of digital technology to minimize licensing, scheduling, and financial risk. Without development of cost-effective qualification methods to satisfy regulatory requirements and address the potential for CCF vulnerability associated with embedded digital devices, the nuclear power industry may not be able to realize the benefits of digital technology achieved by other industries.

Reducing the occurrence of design defects/errors in software-based systems is principally accomplished by *fault avoidance and fault removal methods*[4]**.** Fault avoidance methods (process oriented) seeks to prevent faults from being introduced into the software via structured design and development, formal specifications, or proofs of correctness. *Fault removal* methods imply verifying the system against requirements and verification conditions. Fault removal methods include *testing* which aims to show that the intended and actual behaviors of a system differ, or at gaining confidence that they do not. The goal of testing is failure detection: finding observable 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, from the testing of a small piece of code by the developer (unit testing), to the customer validation of a 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 aiming at different objectives, such as exposing deviations from user's requirements, or assessing the conformance to a standard specification, or evaluating robustness to stressful load conditions or to malicious inputs (fuzzing for security), and so on.

In this report we are focused on exploring the feasibility of testing software from a perspective of *testability*. The definition of testability that the nuclear industry has adopted 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).* [2]

The NRC 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) <u>supports</u> 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*[5].

1.1 Problem Definition

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 the very simplest of 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.

A simple illustrative example is a program modelled as a Finite Automata[6].

The state of an Automata [7] (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

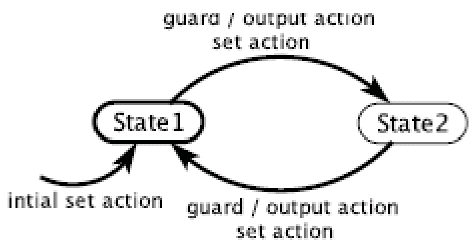

Figure 1:Finite Automata Model

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

A software module with 10 discrete states, 5 variables, and operating on 32-bit integers will have a total enumerated state space of

$$|\text{states}| = 10(2^{32})^{5} = 1.46 \text{ x}10^{48}$$

This is effectively infinity for explicit testing as would take 10^{31} years to complete assuming one test per nanosecond! The key to reducing the state space is recognizing that many of the states are equivalent in their behavior. In recent years, research approaches to justifiably reduce the testable state space have made progress. These include methods based on equivalence partitioning, program slicing, model based testing, identification of state space that has hazardous consequences, etc. Therefore, the state space of interest might possibly be reduced to a manageable dimension through such analysis methods. The degree to which these methods can approach "100%" is in question. Furthermore, there is an overarching driver to "100%" testability and that is to eliminate or reduce the possibility of CCF associated with software defects. Part this work is to explore software testing techniques, characterize them, and select some the most promising methods to see if there is "path" forward on the viability (or not) of an "bounded exhaustive" approach to software testability that is cost effective and practical. This interim report is "first look" at some of the candidate methods.

2. Approach to Research

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. We are mainly focused on methods that support or claim “pseudo-exhaustive” and bounded testing. By bounded/pseudo” exhaustive testing we mean:

> **Definition:** The term *pseudo-exhaustive* is used in relation to *combinatoric software* testing. Combinatoric testing is considered effectively exhaustive when well-grounded inference rules, covering arrays and equivalence partitioning reduce the state space to a testable set. The inference rules or assumption allows for reductions in the number of test interactions based on axiomatic evidence.

For example, Interaction testing at levels of 4-way to 6-way coverage have been suggested to be pseudo-exhaustive due to evidence that software defects involving 6 or more variable interactions are unusually rare [8]. Therefore, if we know from experience that t or fewer variables are involved in failures for a particular application type and we can test all t-way (or t+1 way) combinations of discrete variables, we have high confidence that the application will function correctly. Looked at another way, if we know in advance that all failures are triggered by t or fewer conditions, testing all t-way conditions is in some sense equivalent to exhaustive testing. Combinatoric or Interaction testing applies mainly to the *logical behavior* of the software, not so much to timing behaviors, or state sequences dependences behaviors – but there are testing methods to address these faults.

The approach we follow is to search, identify, and characterize explicit testing methods and technologies that can be applied to existing software articles that are representative of embedded digital devices. We review the methods and discuss applicability to the problem. As such we 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 sensors and actuators. In this case, we introduce the VCU smart Sensor below as preliminary representative device. Viability analysis will be conducted to determine the feasibility of testing this device in a manner consistent with the NRC definition (based on assumptions to reduce state space enumeration). If a viable approach is found, this information will then be used to develop a test specification for I&C device testability that can be used in a future phase of this project to demonstrate digital qualification with respect to common cause failure.

2.1 Assumptions

In this section we introduce assumptions based on “typical” software testing processes, artifacts and goals. The first assumption we make is that source is available to the testers. This assumption allows testers to consider white box or grey box testing approaches. The second assumption enables the testers to review requirements of the software so that test cases and oracles can be implemented to test the software. The third assumption is that testing the logical behavior of the code against a requirement or specification is the main the goal of the testing. There may be other tests required beyond logical (such as timing), but we only consider methods that apply to logical behavior of code. The fourth assumptions state we are not concerned with hardware faults that induce software misbehavior.

- A1: Source code as is and available...no modifications required to execute tests, applicable to COTS devices.
- A2: Design and Specification Documents available for review.
- A3: SW behavior under test- testing at state, state sequence, logical behavior is main focus
- A4: Not concerned with HW faults that induce software misbehavior
    - The hardware faults are independent and identically distributed random which would only fail one device at a time (not correlated across devices)

3. Representative Article VCU Smart Sensor

The VCU Smart Sensor Prototype is a barometric pressure and temperature measurement device that has been derived from a Part-23 (Non-Safety Related) VCU ARIES_2 Advanced Autonomous Autopilot Platform which consists of mature hardware design and software and has over 10,000 hours of tested flight time. The VCU smart sensor can be seen in Figure 2.

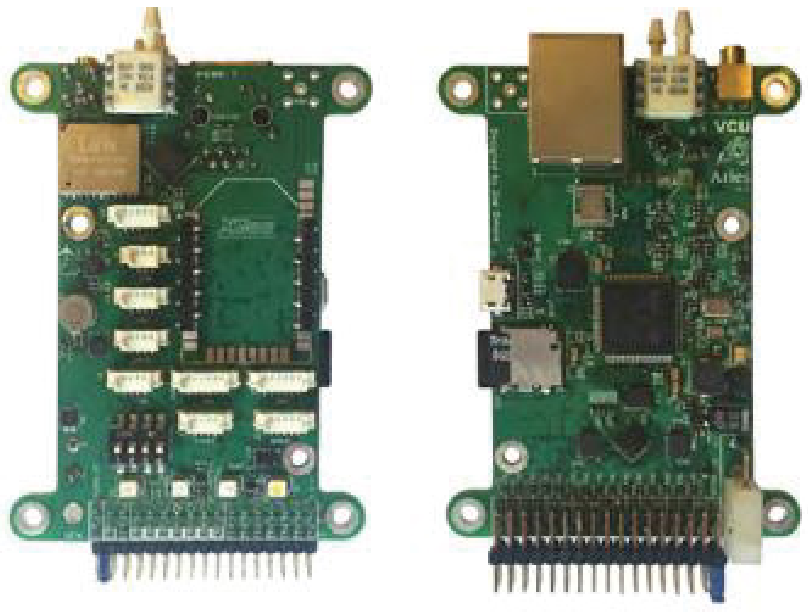

Figure 2. VCU Smart Sensor

The Smart Sensor software and hardware architecture design is representative in terms of form, function, and complexity of an avionics smart sensor, and by extension a nuclear power plant smart sensor. The Smart Sensor is capable of accurate temperature and pressure measurement and connection configuration options, which support various output formats. The Smart Sensor also includes deterministic real-time multi-threaded scheduling for multiple sensors, a functionality of the underlying real-time operating system, ChibiOS. Other specific functional features of the Smart Sensor include digital to analog (DAC) conversion, calibration and communication configurations. The major functions of the on-board software of the VCU Smart Sensor include the collection, display, and communication of measurement data, device configuration, the execution of diagnostic routines, the storage of calibration information, and the performance of sensor functions. The performance of sensor functions includes re-ranging, temperature-effect compensation calculations and conversions, transmitter calibration, and compensation/averaging via a first order Kalman filter and a moving average filter. Board-specific configuration and calibration data is provided built into the sensors from their factory defaults.

The OS software for the smart sensor is built around ChibiOS which is real-time deterministic OS for embedded devices and applications[9]. 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 on-board 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.

Figure 3 shows the program data flow of the software components of the VCU Smart Sensor including the threads and communication protocols used to transmit data between modules. Figure 4 shows the major logic of the system, including the separate threads and the functions associated with each.

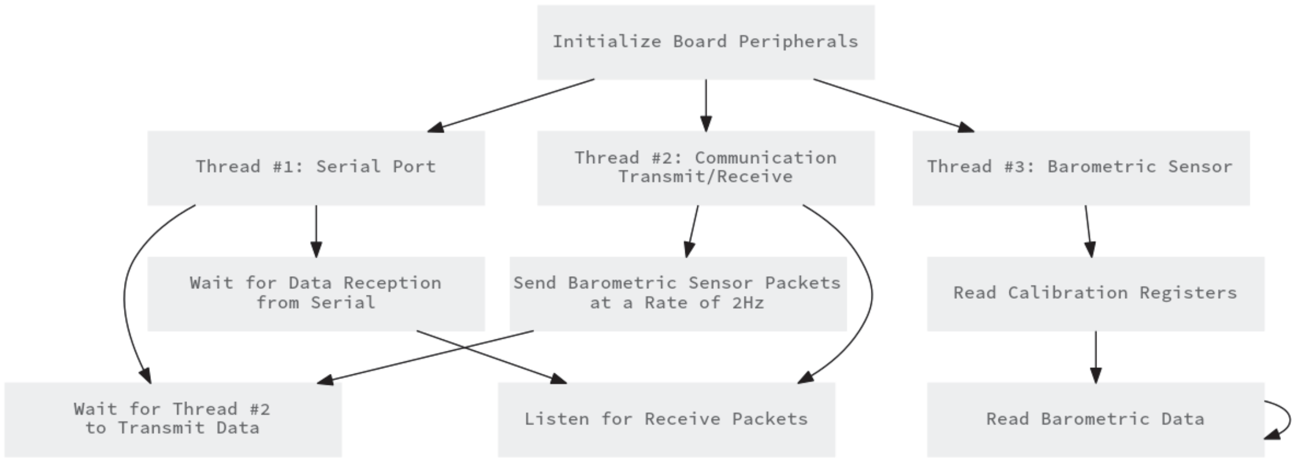
Initialize Board Peripherals
Thread #1: Serial Port
Thread #2: Communication Transmit/Receive
Thread #3: Barometric Sensor
Wait for Data Reception from Serial
Send Barometric Sensor Packets at a Rate of 2Hz
Read Calibration Registers
Wait for Thread #2 to Transmit Data
Listen for Receive Packets
Read Barometric Data

Figure 3. Major Task Interactions in Smart Sensor

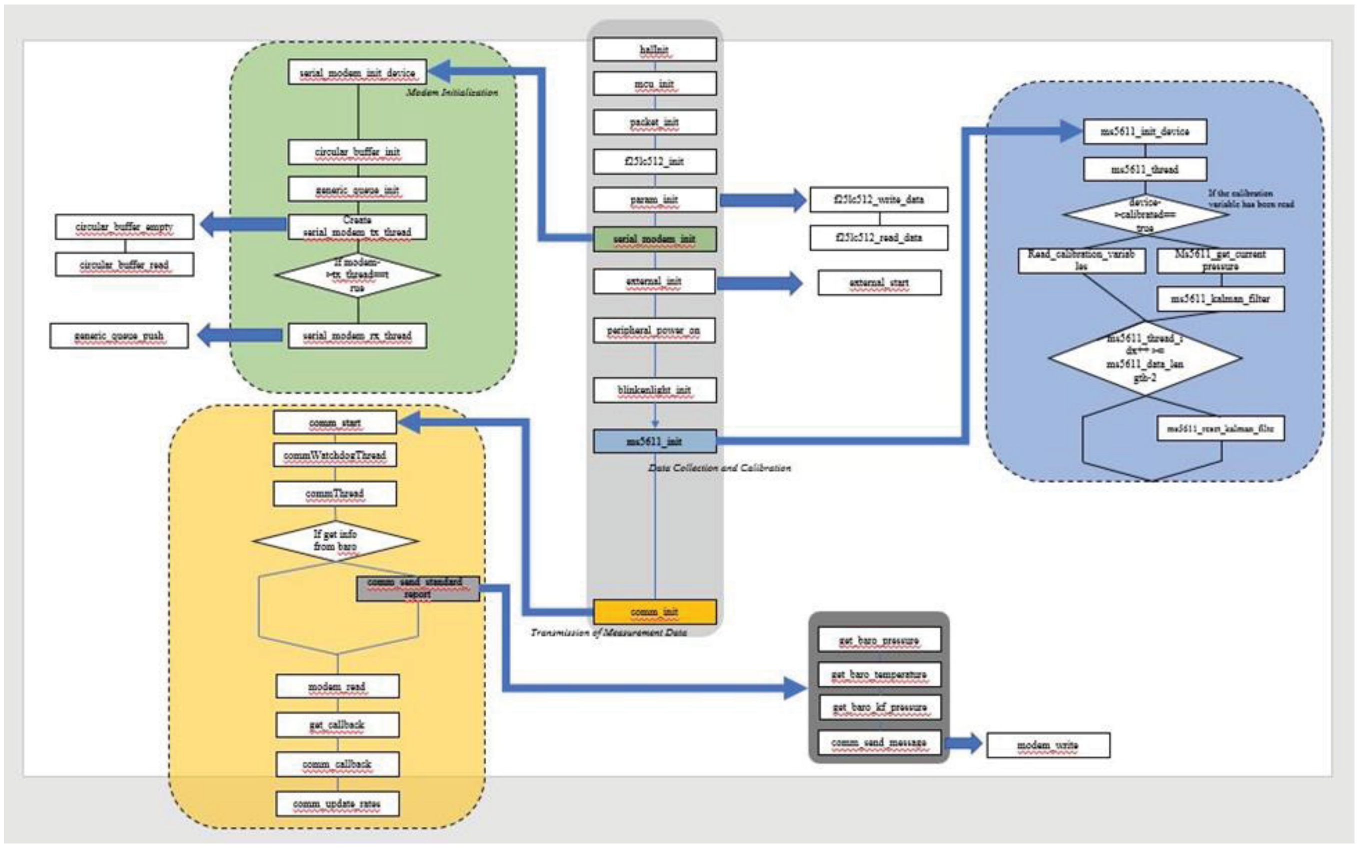

Figure 4. Program Logic Flow of the Smart Sensor

The VCU smart sensor code segment size is approximately 50 Kbytes of program- which is relatively compact. The use of real time operating systems may be excessive for embedded device with limited real time threads, however, it is not an uncommon occurrence.

4. Methods, tools, and computing resources to support bounded exhaustive testing

This Section of the report examines several promising technologies and methods related Pseudo Exhaustive testing. These technologies were reviewed related to their capacity to address Pseudo Exhaustive testing's maturity, and tool support.

4.1 Combinatorial testing

As software is growing in size and complexity, testing the software covering all the interactions between the data, environment and the configuration is a challenging task. The studies conducted in Nation Institute of Standards and Technology (NIST) [10] on Software failures in 15 years of FDA Medical Device recall data concludes that majority of the software failures are due to interaction faults arising from the interaction of few parameters, mostly by two and three. For NASA Distributed database, 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 Figure 5. Applying this 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 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 guaranteeing the confidence of a bug free software. On the basis of experimental data collected by NIST on a variety of software applications as shown in Figure 5, it has been deduced that the cumulative percent of faults triggered in a software reaches 100% when the number of parameters involved in the faults reaches 6. 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 4 parameters with 3 values each covering all possible combinations will result in 81 testcases. If the combinatorial method which limits to a pairwise interaction level of parameters is used, the number of testcases can be reduced to 9. The combinatorial method thus renders a drastic reduction in testcases without compromising on quality of testing[10].

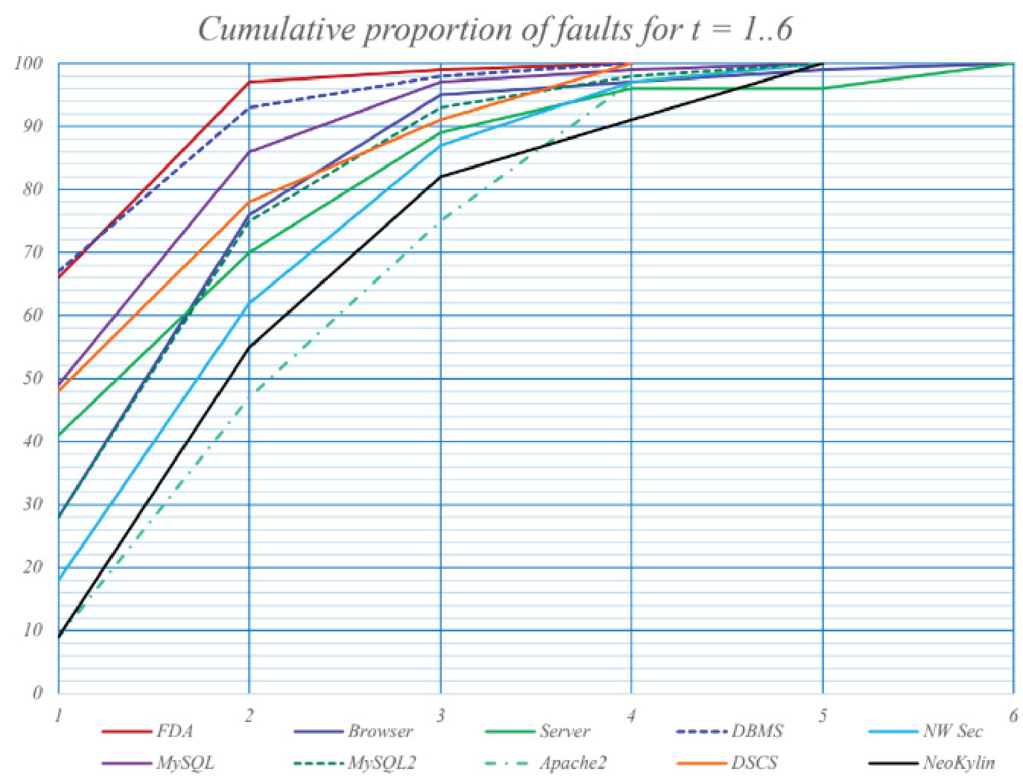

Figure 5: Cumulative proportion of faults for t (number of parameters) = 1...6

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[11].

The first step for creating a test model is to identify all the relevant parameters and this should include the user and environment interface parameters and the configuration parameters. 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, we need to 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 [12].

Richard Kuhn and Vadim Okun’s work on “Pseudo Exhaustive testing for Software” [10] 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 testcase 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 and will cause the software model to fail thus causing the model checker to generate counterexamples that can be used as testcases. By using t-way coverage of the variables as the test criterion, we can derive the combinatorial testcases. 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 variable combinations that satisfy R. The experiment conducted by Richard et al in using Symbolic model checker to create pairwise to 6-way combinatorial testcases 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 testcases were found to reduce as the input interaction coverage (t) increases.

Richard Kuhn (NIST) and James M Higdon (US Air Force) ’s research work on extending the application of combinatorial testing to event driven systems, described in the paper “Combinatorial Methods for Event Sequence Testing” [12], also proves to be extremely valuable. Some faults in the software gets activated only when there is a particular sequence of events happening. Sequence covering arrays can be used to test all the t-way order of t events in a software. The basic concept of sequence covering is that, if we have a 2-way event testing, there should be a testcase 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 in 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 in tracking down all the event sequence-based issues in software thereby improving the efficiency of testing. A tool for combinatorial parameter and sequence-based testing named ACTS is described in detail in a later section in this report.

There has also been a lot of research in the field of studying and developing various algorithms for covering array test suite generation which include Mathematical methods, Greedy algorithms and Heuristic methods. Bryce et al's greedy algorithm for testcase generation in [13], which takes user inputs on the priorities of the interactions to be covered and which also allows for seeding of fixed testcases into the test set, is identified as another important work in the field of combinatorial testing.

4.2 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. For complex systems, the state space constituting all combinations of inputs 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. While the majority of interest in BET has been for simpler systems[14], [15]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 6 or less events may occur. The state space is reduced to 60,459 possible input combinations.

Marinov[14] 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 8-47 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. 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 testcase 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 [16] [17], 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 GPU implemmentation[17] has observed 17.46 speed up by contrast to Siddiqui's work[18], suggesting the multi-threaded GPU approach to have high potential for future work.

The primary limitation of the BET is that the statistical reliability of the system cannot be determined since the input values selected do not tend to be valid samples of inputs used in production.

4.3 Unit testing

Unit testing is a method of testing where the design is divided into modules or units and each unit is tested to check if it meets the requirements. The aim of unit testing is to isolate each unit and analyze them individually and address any errors that may occur at each smallest testable part of the design. Unit testing ensures functional correctness of the design, helps identify defects early and makes the integration testing with all the units together easier[19].

Unit testing can be white box testing or grey box testing based on the structure of the program and the test execution and checking for each unit is repeatable and automated [18]. When an automated unit testing is performed in an environment which is not natural to the unit under test, dependencies are revealed between the unit under test and other modules. This helps redesign and remove any unnecessary dependencies in the design[20].

All errors cannot be identified with unit testing as you cannot assure the correct output for all execution paths and all input combinations. Another challenge is to setup realistic and useful test cases. Unit testing has a combinatorial problem where each line of code or a Boolean statement would require two test cases, one for a true outcome and one for a false outcome which would make manual test case generation very difficult. Therefore automated test generation tools such as Junit are used[20].

4.4 Model-Based Testing Methods

Model-Based Testing (MBT) dates back to the 1970's when it was known as specification-based testing. MBT is considered one type of a lightweight formal method to verify the system behavior [21]. MBT relies on the behavior of the System Under Test (SUT) and its environment that are expressed in explicit behavioral models. The idea behind MBT is that explicitly encoded artifacts mitigate traditionally-derived tests which are typically unstructured, unreproducible, undocumented, lacking rationale, and are heavily-reliant upon the acuity of test engineers. MBT consists of the processes and techniques that potentially allow for automatic extraction of test cases from the model and provision of generating test oracles. Typical models used are UML, SysML, model checkers, finite state machines, and mathematical formalisms.

Figure 6 illustrates the process of MBT, as presented in[22]. There are 5 steps in the MBT workflow that starts with the development of a test model of the system under test (SUT). This test model focuses on directly linking with test objectives at an abstracted level from requirements and specifications. Also, from requirements come test selection criteria that produce the test case specification. The test model and test case specifications then produce test cases that are embedded in a test script and are used to stimulate the SUT which ultimately produces verdicts on the test cases.

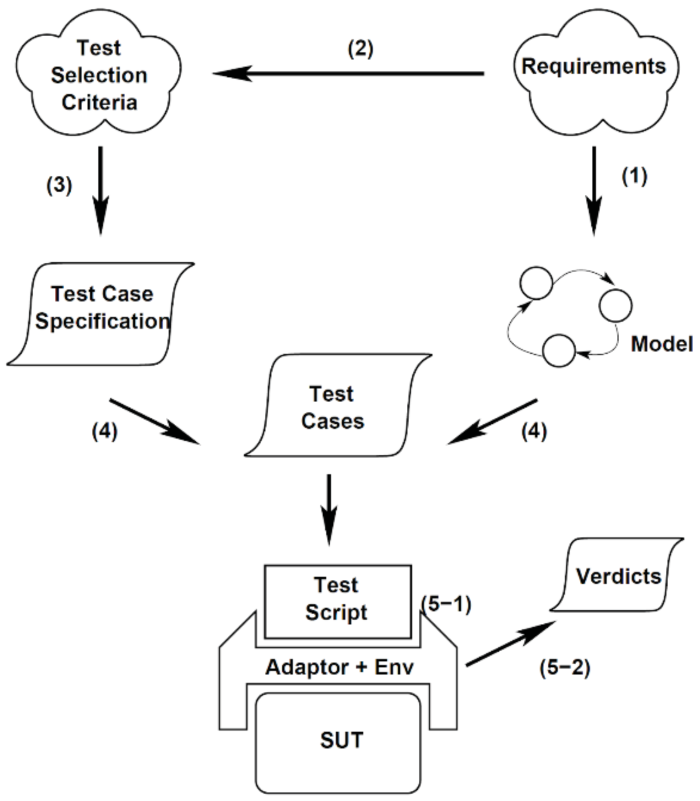

Figure 6: The Process of Model-Based Testing

One real-world problem in software testing is determining the correct output for a given set of input profiles, generally referred to as the test oracle problem. The key benefit of MBT for exhaustive testing is the semi-automatic generation of test oracles. If human manual intercession is required to check outputs, the efficiency of the testing will be extremely limited to a few hundred test cases. Therefore, automation is a necessary condition for testing hundreds of thousands of test cases – which is not uncommon for pseudo-exhaustive testing. One approach to automated generation of test oracles *is model checking*[23], which uses a formal specification to compute expected output for input values and events.

Model Checking is based on exhaustive state exploration of a mathematical model of the software requirement or specification. The properties of the SW are often described in temporal logic format such as Linear Temporal Logic (LTL), Computational Tree Logic (CTL). Input to a model checker has two parts. First is a state machine model of the SW defined in terms of variables, initial values for the variables, environmental assumptions, and a description of the conditions under which variables may change value. Once the model has been constructed, the temporal logic expressions are searched over states and execution paths. Conceptually, a model checker visits all reachable states and verifies that the temporal logic expressions (e.g. which defines what the SW is supposed to do) are satisfied over all paths. If an expression is not satisfied, the model checker attempts to generate a counterexample in the form of a sequence of states. In model-based testing, model checking is used to generate tests in the negative – tests that attempt to refute the requirements of the code. Another advantage of model checking is that it is fully automatic. However, the primary disadvantage is that it does not scale to large systems, partially due to the full formalization of both the specification and the design of the SW. We discuss this is more detail in the tools section.

The key value of model-based testing to any pseudo-exhaustive testing paradigm is that it (1) provides a means to create oracles for comparing the behavior of the software under test, and (2) generate test sequences for variable interactions. Both of these are important for large scale testing.

Key papers for understanding this technology are: [24], [25], and [26].

4.5 Distributed computing to facilitate efficiency/feasibility of testing

With expanding software and the resulting growth of the test suite size, the time taken for execution of the test suites increases and hence the productivity and efficiency of testing reduces. Distributed computing is one of the solutions to eliminate the overhead of testing large and complex software in industries. Computational Grids can be used as a testbed for performing unit testing and integration testing on large Software projects.

The entire software or parts of the software to be tested are transferred to all the computers that are part of the software testing grid network and separate test suites are concurrently and independently executed on the grid computers. In this way, unit testing of functions or components in the entire software can be distributed among the grid resources and thus ensured to be completed in a far lesser time. There are various grid-based software testing tools that enable the users to create software test suites for each individual unit, assign the tasks to the different computers, schedule them and collect the test results. The grid based test framework developed by Yaohang Li et al and discussed in the research paper "Using Grid Computing for Distributed Software Testing" also supports for integration testing of the software in a grid computing network by allowing the testers to create workflows for scheduling closely related testing units in an order defined by the software architecture[27].

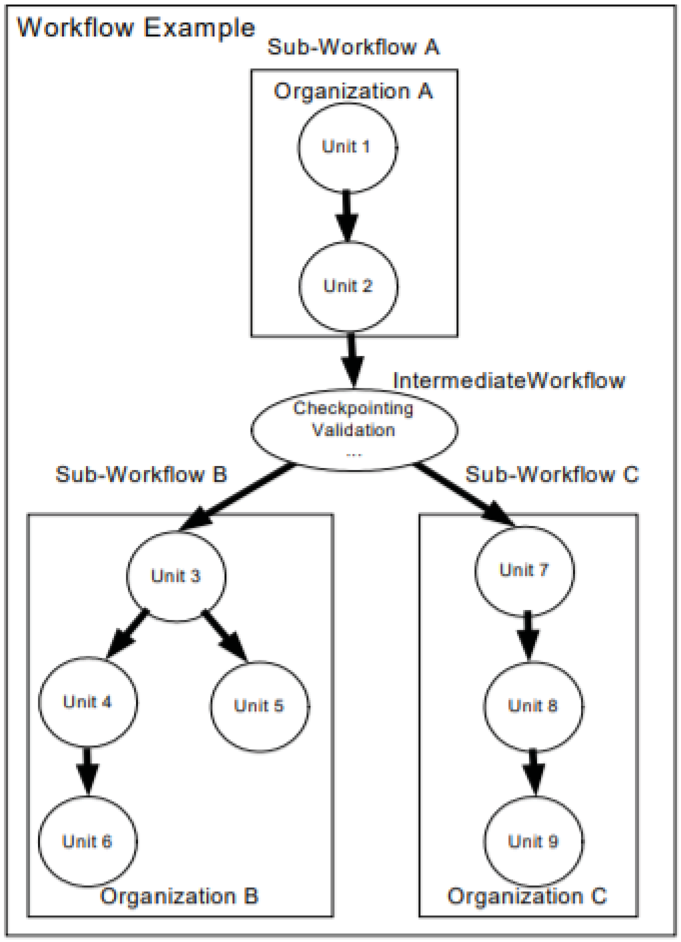

Figure 7: Example of a Grid-based Integration Testing Workflow

Research paper by Alexandre Duarte et al "GridUnit: Software Testing on the Grid" [28] provides insight into the advantages of Grid computing and introduces a tool for utilizing the grid network for software testing, which is described in detail in the later sections. The parallelism that grid network offers helps in faster test execution thus lowering the testing cost and increasing the testing efficiency. More importantly considering input domain and software size, state space, even after applying various techniques such as input domain partitioning, equivalence class partitioning and combinatorial testing methods, the test suites might end up being bigger than that can be handled on a single computer. In such cases, Grid computing could be the ultimate solution [28].

There are also few drawbacks of the grid computing that needs to be handled or traded off when considering this approach for testing. In Grid network nodes could be joining in and leaving dynamically. File transfer delays could be an additional overhead and it will depend on the network speed. As the processing capacity of each node varies, the job completion times of the nodes will also be different. There could also be security threats on the software under test imposed by untrustworthy or unreliable nodes in the grid[27]. For the purpose of Software testing, creating private grid networks with centralized control could be a solution to handle majority of the above issues and making the system stable.

Figure 8 [27] shows the flow of distributing software testing task and collecting test results via a grid middleware.

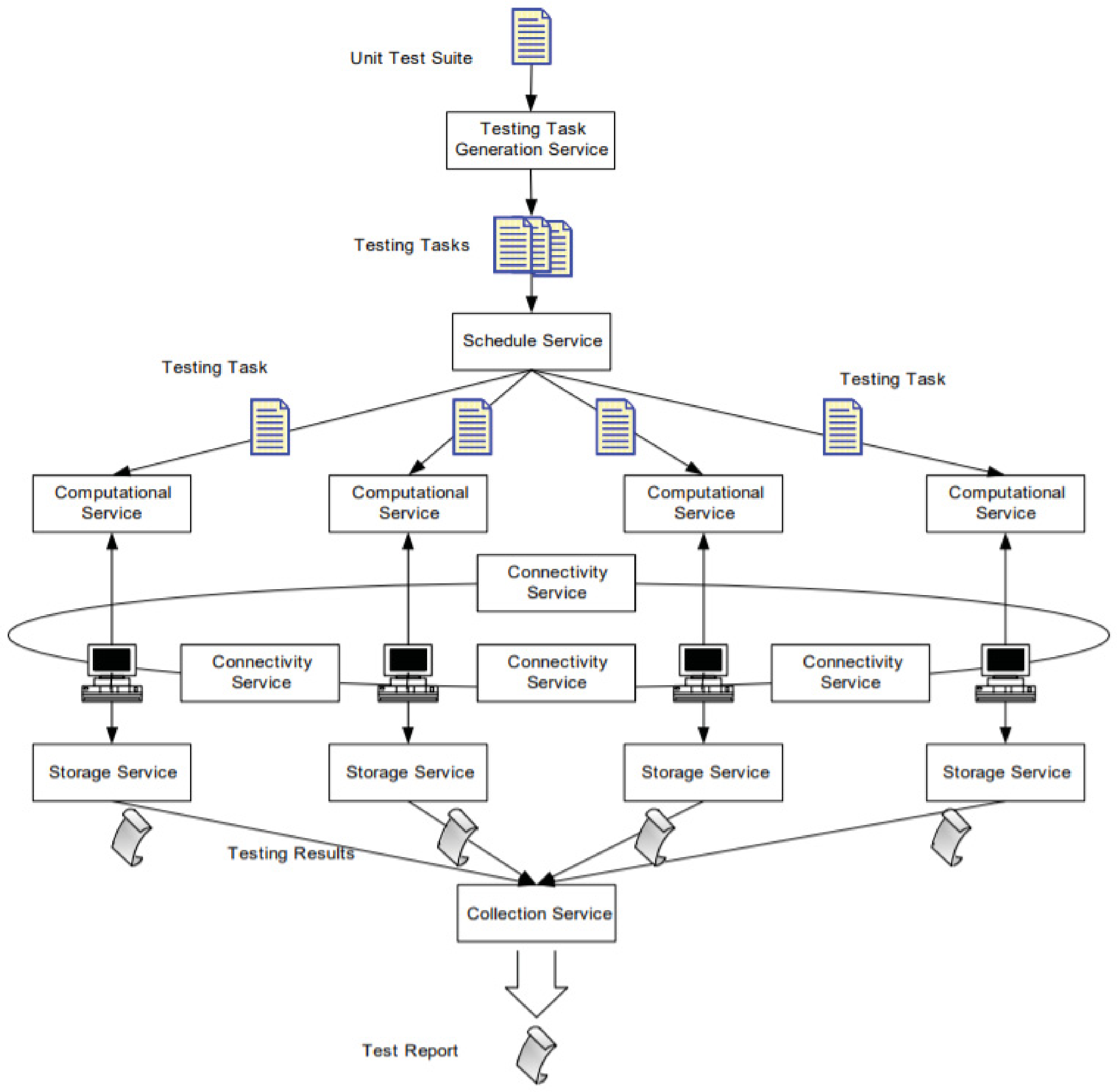

Figure 8: Running Software Test Tasks on the Grid

4.6 Static decomposition and analysis to help identify modules

Static testing is a method of testing performed without executing the code. Initially the requirements, design and test cases are verified to ensure that there is no ambiguity. Next, structural analysis of the software is performed to ensure that there is no inconsistency between modules that are connected and the design is checked for unused variables, dead code and syntax errors. Static testing also ensures that the variables are always initialized, as a default null value assigned to an uninitialized variable can cause null pointer exception[29]. If there is a requirement for the code to comply with a safety standard, static testing checks and ensures that this condition is satisfied. Security vulnerabilities are also identified by static testing, thereby making the design more reliable. For example, Mathworks Polyspace® performs static code analysis and addresses security challenges by exposing critical vulnerabilities in the code. It also ensures that the code complies with standards such as CWE and CERT C[30]. Static analysis is most useful at finding bugs related to syntactical processes in SW development. Static analysis can be used independently on dynamic testing methods (like Pseudo Exhaustive testing) to find different bugs (e.g. array bounds).

4.7 Enabling technology – distributed test management, distributed compute clusters, and paradigms

This section mentions distributed frameworks and programming paradigms relevant to the research approach in a distributed computing environment.

4.7.1 Hadoop

Hadoop is a cluster infrastructure framework for distributed computation and storage of data in a heterogenous cluster environment the operates on the premise that calculations should happened near the data. Hadoop frameworks can be used to support distributed testing. The framework uses a Map/Reduce approach, discussed later, a master/slave architecture, and a distributed file system known as a Hadoop Distributed File System (HDFS). The HDFS is a triple redundant (default) fault-tolerant file system that is able to detect and correct data faults in the file storage system. The HDFS possesses traits such as high bandwidth, high availability, and scalability making Hadoop a desirable system for data intensive computations[31].

Please reference Figure 9 for the architectural description in this paragraph. In the Hadoop architecture resides a namenode that manages the file space across the distributed system. The namende holds the directory structure, file-to-node mapping, and permissions of the data file system. The namenode directs its clients to the location of the datanode in which the requested informations resides. Other operations of the namenode include block replication and rebalancing. The job submission node is responsible for tasking jobs to the tasktracker of individual slave nodes[32].

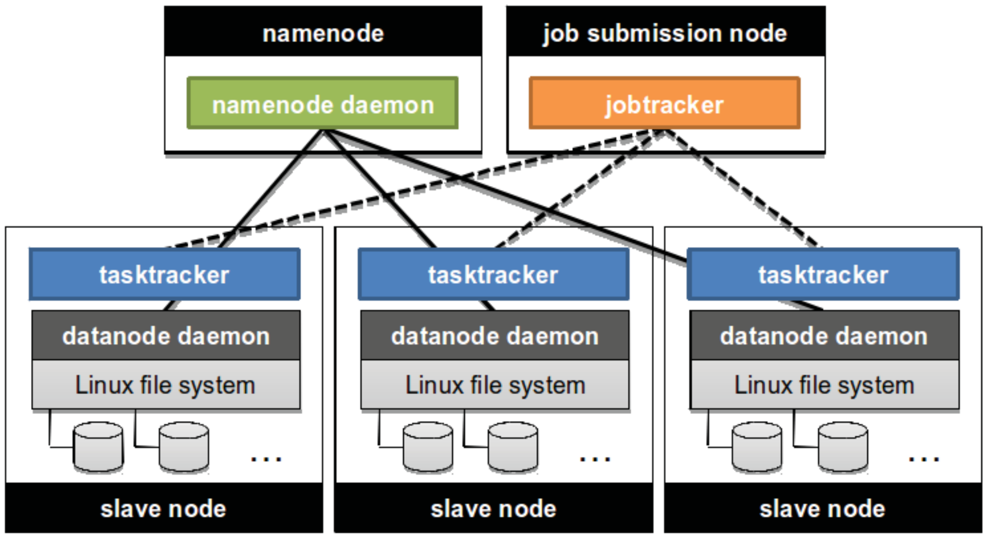

Figure 9: Apache Hadoop and HDFS Architecture[32]

4.7.2 MapReduce Paradigm

Reference Figure 10 for this paragraph. MapReduce is conducted in this way. Portions, splits, of an input HDFS file are distributed across the Hadoop cluster and is fed to a mapper. The mapper then sorts and maps the input to a key-value pair. The key-value pair is then copied to a reduce node where keys are sorted and merged, and reduced into an output HDFS that is split into parts[32].

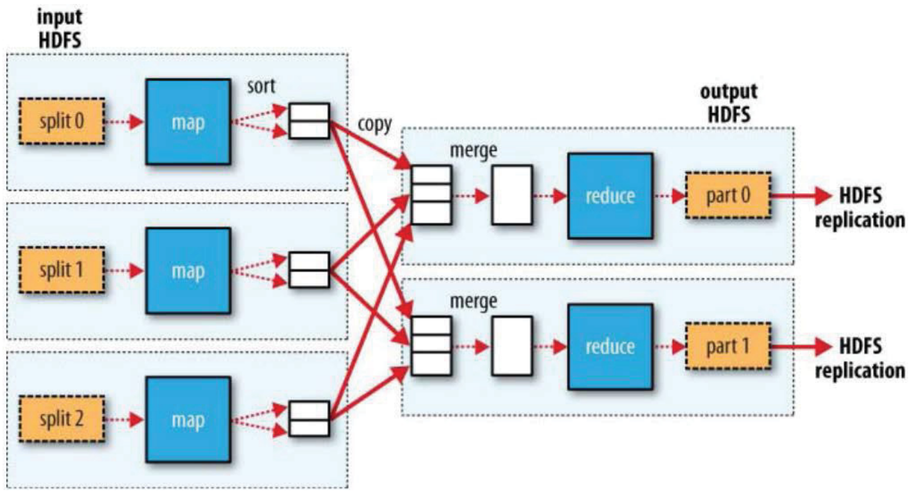

Figure 10: MapReduce Data flow[32]

4.7.3 GridUnit and Ingrid

The GridUnit developed by Alexandre Duarte et al[28] is an open source Java based framework that can automatically distribute the execution of a Junit test suite over any grid. This grid-based tool runs on top of the grid middleware that provides the services and protocols required for the interoperability between the networked computers. The GridUnit tool architecture composes of a GUI displaying information about the testing execution, a Grid TestRunner for creating job description from the test suite and scheduling the jobs in the grid network and a Grid TestMonitor for providing centralized control for monitoring the execution of testcases.

The GridUnit can be applied for applications that needs to be tested faster and for applications that needs to be tested in a variety of environments. Experiments conducted with the GridUnit gives promising results on the reduction of the test execution time. A synthetic application when tested on a single machine with a test suite composed of 288 test cases, takes 24 hours. When the testing was distributed by GridUnit on a free to join public grid, the 'OurGrid', it reduced the time of testing to 20 minutes with 111 grid machines. This is almost 70 times speed-up [28] compared to the time taken to run on a single computer[33].

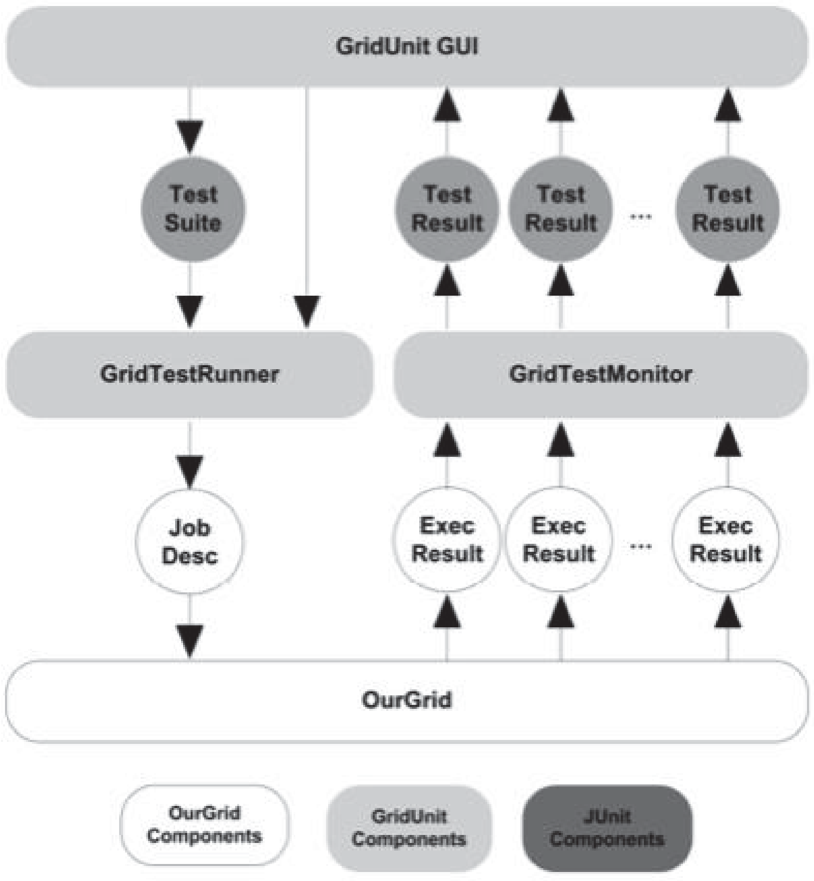

Figure 11: GridUnit High Level Architecture

There is another tool Ingrid (Incremental Grid Deployer) described in described in the paper "Multi-environment software testing on the Grid" [33], that was developed to supplement the GridUnit with additional feature of allowing the user to describe and deploy the environments in the grid machines. Using Ingrid users will be able to provide the description of environment in terms of the applications, data and configurations in which they want the software to be tested in the different grid machines. This will help the testing or verification team to test the same software on multiple and well-defined configurations in parallel, without consuming extra time[33].

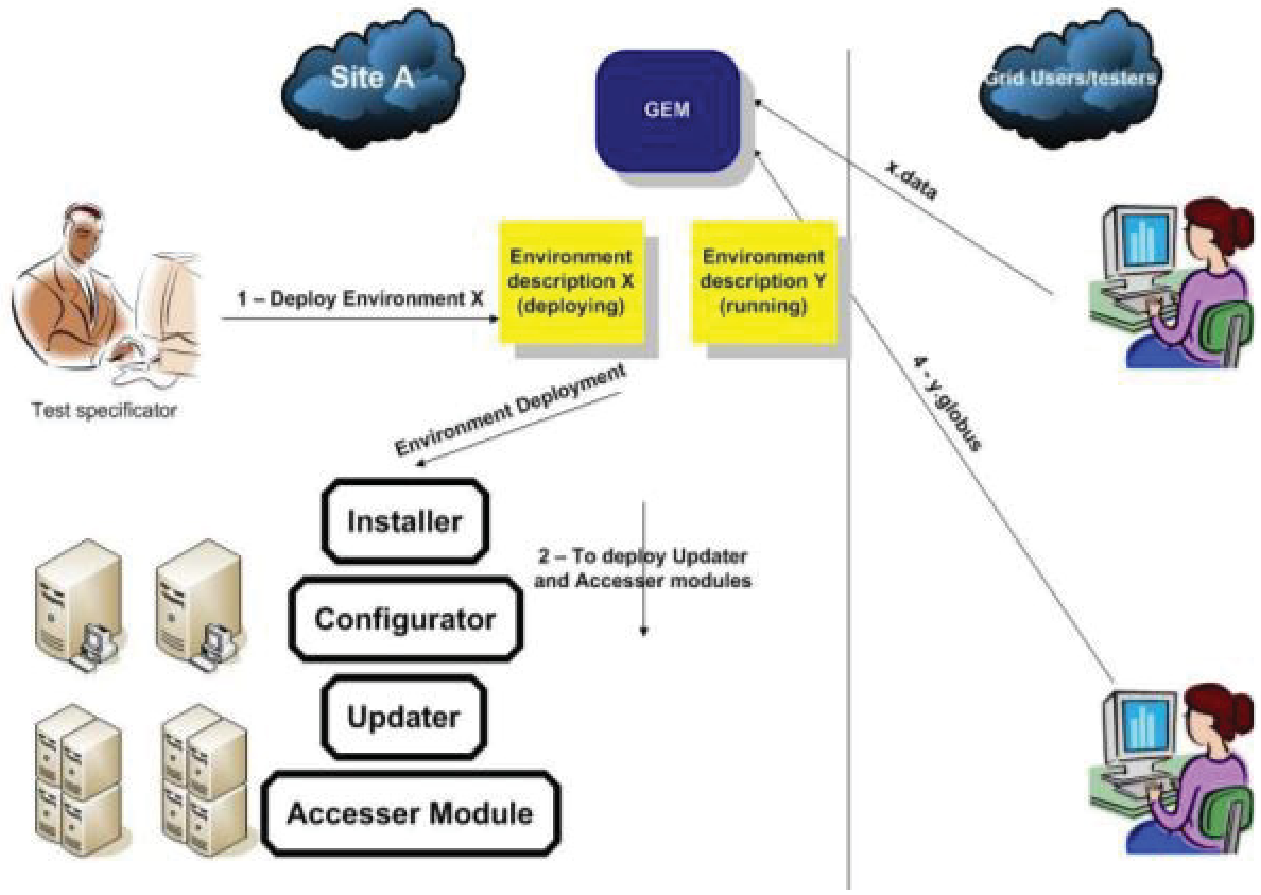

Figure 12: InGrid High level Architecture

4.8 Tools - State of the practices and Art to assist testing methods

Here we discuss some of the tools used for combinatorial testing and model checking that were discussed earlier.

4.8.1 Combinatorial Testing tools

Automated Combinatorial Testing for Software (ACTS) - ACTS is a combinatorial testing tool developed by US National Institute of Standards and Technology (NIST) and the University of Texas at Arlington. It is written in Java and can be used on any operating system including Windows, Linux and Mac.

ACTS supports t way combinatorial testing. The main idea behind this is the faults in a system with k parameters can be detected by combinatorial testing of fewer parameters t. All the combinations of t parameters are tested at least once and with correct modeling of the test parameters, all the faults can be detected. In-Parameter-Order-General (IPOG) and several other combinatorial test generation algorithms are used by ACTS. IPOG balances the size of the test set and the time to generate test cases. ACTS supports certain advanced testing features such as mixed strength test generation and constraint handling which are supported by the IPOG algorithm [34]. One of the important criteria while testing is to cover the sequence of events. For example, while testing an event driven software, a critical condition for failure at the current state may be if a particular event has occurred in the past. To address this issue, ACTS ensures that t events are tested in all combination of t way order, using a sequence covering array for testing [12]. Further improvement of the tool is in progress, with new features such as ability to include negative testing and improve constraint handling.

4.8.2 Model Checker tools

Java PathFinder (JPF) – JPF is an open source model checker developed by NASA Ames Research Center to analyze and formally verify programs written in high level programming languages. JPF consists of a Java virtual machine which reads the Java code with assertion statements and checks for race conditions, detect dead locks and violations of assertions. JPF handles big states by state compression and uses techniques such as slicing, symmetry reduction etc. to reduce state space. It is widely used in the avionics industry including the real time avionics operating system developed by [11]. One of the limitations of JPF is it can have combinatorial explosion despite using state reduction techniques such as partial order reduction[11].

Berkeley Lazy Abstraction Software Verification Tool (BLAST) – is a model checker for verifying C programs. BLAST checks temporal safety properties in C programs and if there is a violation of the property, it gives the execution path in the program that causes the violation. BLAST performs property verification by creating abstraction of the program in state space using lazy predicate abstraction where it first tries to verify the property on a coarse abstraction of the program which includes few relations or predicates. If the property fails, the execution path that violates the property is given. If there is no violation of property, it may be due to the coarse abstraction of the program, and therefore BLAST incudes more predicates in the program abstraction and reiterates the abstraction process multiple times until it finds a violation or has a precise abstraction to prove the absence of a bug. In [35] case studies were performed on BLAST to prove safety properties to check memory operations and to generate test conditions to get full coverage for a given predicate. Although BLAST was successfully used in applications such as device drivers and protocols, it doesn't work on large programs with heap data structures. Creating precise abstractions for heap analysis is being explored[35].

C*PAChecker* – CPAChecker is an open source model checker for verifying C programs. It uses lazy abstraction like BLAST but it can perform checking the program on the go without needing a separate block for abstraction. This allows for greater flexibility of the program and effective algorithms[36]. CPAChecker performs a new way of model checking called configurable software verification where it performs the model checking and analyzing the properties with one formalism.

Distributed Real-time Embedded Analysis Method (Dream) – Dream is a verification tool for formal distributed real time embedded systems. It performs scheduling of events based on timed automata and uses UPPAL or Verimag IF model checker for real time analysis. DREAM can perform a performance estimation of the distributed real-time embedded (DRE) systems [37]. Figure 13 shows the model-based verification of DRE systems by Dream. Domain-specific Model (DSM) is a high-level specification written in languages such as Architecture Description Language (ADL) or textual specification and they describe the properties of the design, design behavior and constraints to be satisfied. The DSM is translated to executable formalisms ensuring that there is right level of abstraction to avoid inaccurate results or state explosion problem. DRE Semantic Domain captures multiple properties of the DRE system. Simulators are used with the analysis tool so that if a property is violated, the simulator can run the execution trace that caused the violation. The semantic model is mapped to heterogeneous models of computation to analyze for three issues namely formal verification, performance estimation and real-time verification. This analysis provides partial simulation results if model checking cannot be performed exhaustively and fails due to state explosion problem [38].

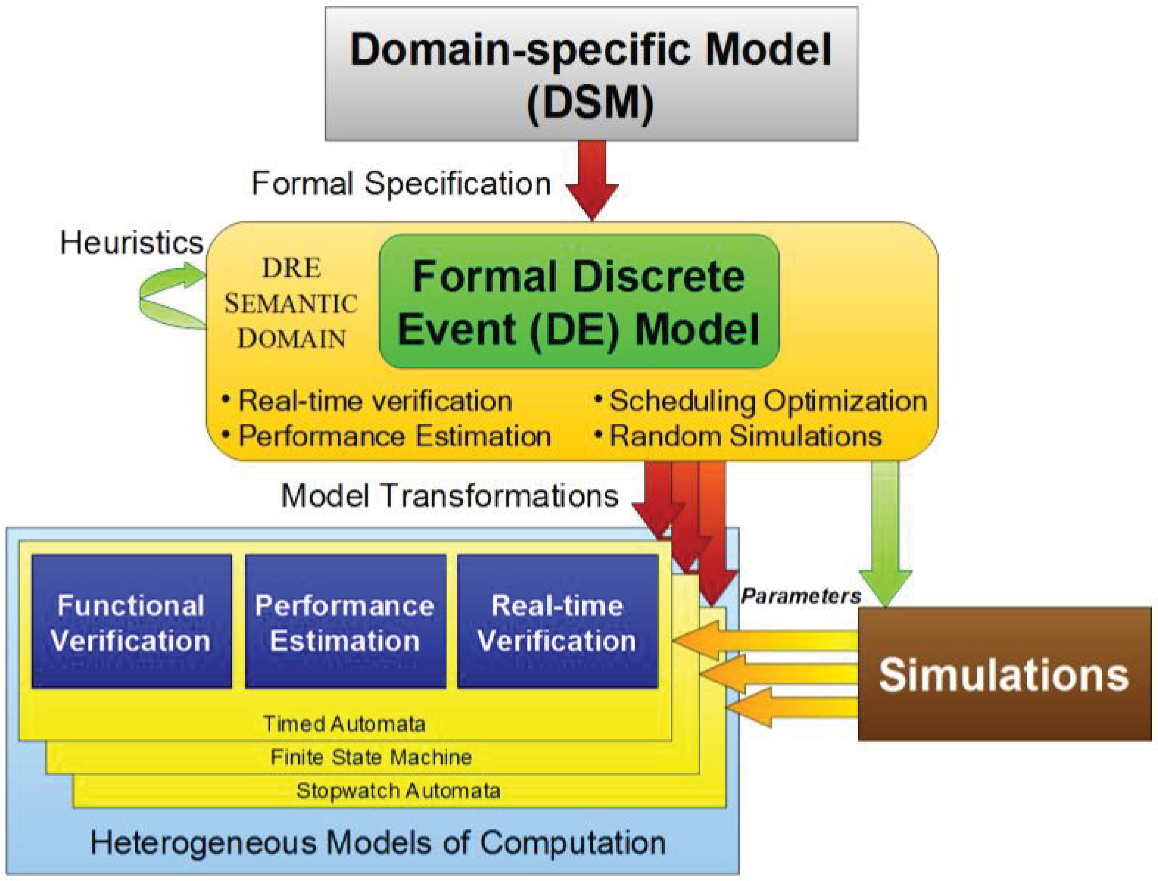

Figure 13: Model-based Analysis of Distributed Real-time Embedded Systems [37]

4.9 Methods for which pseudo exhaustive leverages

4.9.1 Modularity

State of the art software applications follow a modular design approach where the program could be decomposed into a number of interacting modules. From dataflow diagrams, different software modules could be isolated taking into account data dependency. The purpose of this technique is to breakdown the software into smaller manageable pieces of code that could be exhaustively-tested in parallel. Figure 14 shows a first-level breakdown for typical data acquisition and communication functions. Further breakdowns are carried out to the level of primitive functions that do not need verification.

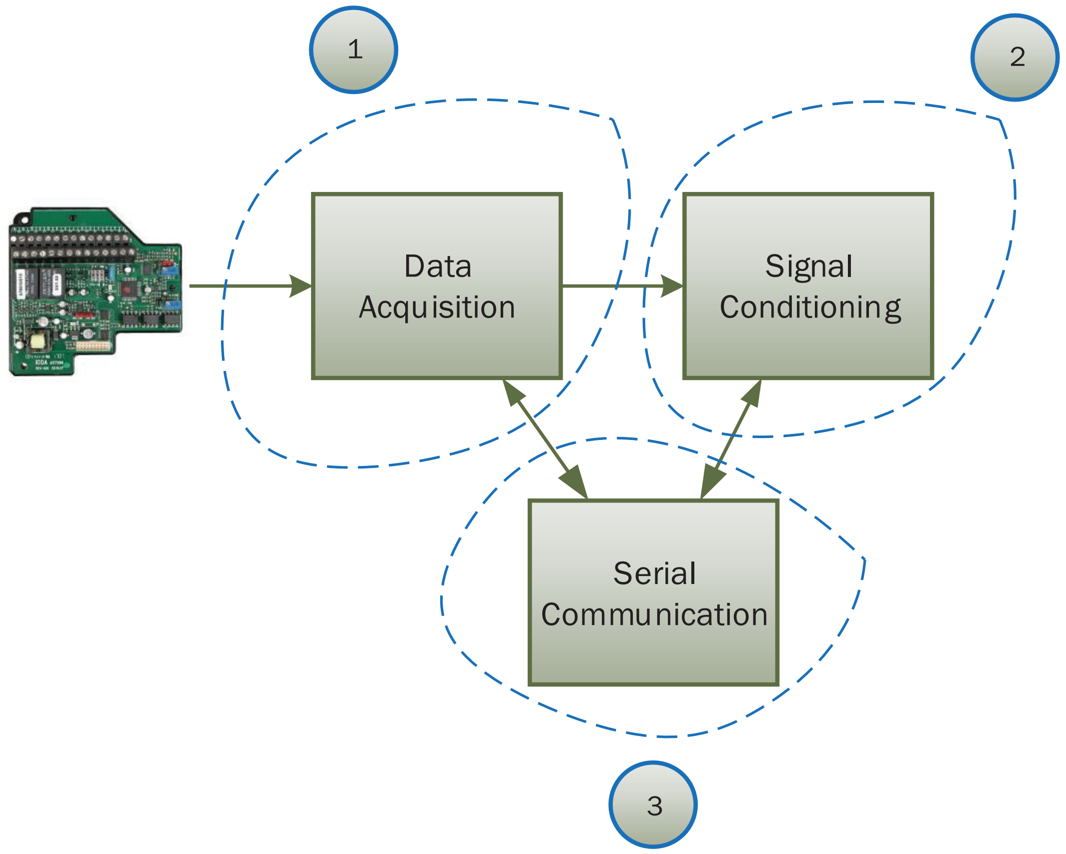

Figure 14: Software Modular Breakdown for Testing

4.9.2 Input Domain Partitioning

The purpose of input domain partitioning techniques is to utilize mathematical approaches to reduce the domain of possible inputs. In this report, we consider range constraints, equivalence class partitioning, boundary value analysis, and pairwise testing.

4.9.3 Input Data Range Constraints

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. These examples are typical for instrumentation and control applications. Although this technique is application-dependent, the nature of the nuclear application and the instrumentation required to measure various physical phenomena suggests that the adoption of this technique can considerably reduce the input domain.

To properly apply this approach, any data out of range will be assumed either not permitted by the data source or handled by a software exception handling mechanism implemented by the module under test. This should not be considered a limitation of the technique as contemporary software design includes input checking and error handling as standard features.

4.9.4 Equivalence Class Partitioning

Given a program *P* in the form of a Control Flow Graph (CFG), the program could be divided into separate execution paths. Each execution path contains only basic blocks, which contain sequences of consecutive statements with no conditional instructions. For each path, there is a path condition that is used to decide whether the path will be executed. All program inputs that satisfy the path condition constitute the *path domain*. The path domain, therefore, is a subset of the program's input domain[39].

For bounded exhaustive testing, we can use equivalence classes to further divide the input domain into separate path domains. Each program path along with its path domain will be treated as a separate program that could be tested exhaustively on a separate computer and in parallel with other program paths.

This technique, however, does not come without its own caveats. For example, software loops (e.g. multiple iterations through a section of code) through a increase the number of paths significantly. Every traversal of the loop body adds a condition to the program, hence increases the number of paths by at least one (think of unfolding the loop iterations). This makes exhaustive testing difficult. We propose further investigation of two techniques to overcome this problem: (1) explore the domain knowledge for sensors and actuators, and (2) investigate mathematical techniques, e.g. formal proofs, to use sample test cases only.

Finally, boundary value analysis is often used in conjunction with equivalence partitioning to discover edge case input values at the boundary of equivalence classes. Boundary value analysis does not contribute to the reduction of the possible input vectors, but it is essential to treat boundary values as important test cases.

4.9.5 Relationship Between Software Testing and Formal Methods

Physical testing and simulation-based testing have been the mainstay for software testing for many years. In the past 15-20 years, the discipline of formal methods and verification has emerged as a complementary companion to testing. In a nutshell, formal methods provide a framework within which people can specify, develop, and verify systems in a systematic manner. A method is formal if it has a sound mathematical basis, typically given by a formal (logic) specification language. This basis provides the means of precisely defining notions like consistency, completeness and correctness. It provides the means of proving that a specification is realizable, proving that a system has been implemented correctly, and proving properties of a system without necessarily running exhaustive tests to determine its complete behavior.

The most noticeable difference between testing and formal verification is that the former requires input vectors and the latter does not. The mind-set in test-based verification is first to generate input vectors (a test bench) and then to derive reference outputs (from specification and requirements), and then observe outputs to determine compliance. The thinking process is reversed in the formal verification process. The user starts out by stating what output behavior is desirable (or not desirable) and then lets the formal verification checker proves or disproves it. Users do not concern themselves with specific input stimuli at all. In a way, the simulation-based or test methodology is input driven and the formal methodology is output driven.

The strength of formal verification sometimes leads to the misconception that once a design is verified formally, the design is 100% free of bugs, which could be illustrated by viewing the formal verification method from the perspective of the output. Simulating a vector can be conceptually viewed as verifying a point in the input space. With this view, simulation-based verification can be seen as verification through input space sampling. Unless all points are sampled, there exists a possibility that an error escapes verification. As opposed to working at the point level, formal verification works at the property level. Given a property, formal verification exhaustively searches all possible input and state conditions for failures. If viewed from the perspective of output, simulation-based verification checks one output point at a time; formal verification checks a group of output points at a time (a group of output points make up a property).

A major disadvantage of formal verification methods is that for complex systems or software, it may not be able to finish before a decision is reached. As a result, designers and test engineers must use sound engineering judgement on when to employ formal verification, on what parts of the system to apply it to, what type of tool is most applicable and what types of strategies to use when trying to prove properties about a system. In many ways, formal methods now offer the promise of significant improvements in verification and validation. But it is widely recognized that these methods are expensive, and their use has been limited largely to high-risk areas such as security and safety.

5. Findings and Summary

While this report and its findings are preliminary, we have identified several methods, technologies that are synergistic toward large scale pseudo-exhaustive testing of software components commensurate with digital embedded devices. Methods to pre-analyze, reduce state space and execute testing include combinatoric testing, combinatoric sequence testing, model-based testing, and model checking for generating complex inputs for infrequent state actions. Technologies related to acceleration of testing, improving efficiency of testing, and managing tests were investigated as well. These methods and technologies allow testing to be carried out at large scale dimensions if needed. Distributed testing enables significant reduction in time. While there are methods and tools for managing distributed SW unit testing across private clouds, grid computer clusters we do not know at this time how these methods map to testing embedded computing applications. While these findings are preliminary, there is guarded optimism that the nexus of these state of the art methods and technologies can afford solutions to pseudo exhaustive testing of digital I&C software. State space reduction methods (inherent to most of our reviewed methods) is almost assuredly necessary to address the problem. How to justify state space reduction is important such that no impactful SW defect is removed. Based on the preliminary investigation of the challenges, methods, tools, and resources the research is positive (optimistic finding).

6. References

[1] "Embedded Digital Devices in Safety-Related Systems," *Federal Register*, 05-Jun-2014. [Online]. Available: https://www.federalregister.gov/documents/2014/06/05/2014-13087/embedded-digital-devices-in-safety-related-systems. [Accessed: 26-Sep-2018].

[2] "ML110550791.pdf." [Online]. Available: https://www.nrc.gov/docs/ML1105/ML110550791.pdf. [Accessed: 26-Sep-2018].

[3] D. R. Kuhn, R. N. Kacker, and Y. Lei, "SP 800-142. Practical Combinatorial Testing," 2010.

[4] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Secure Comput.*, vol. 1, no. 1, pp. 11–33, Jan. 2004.

[5] J. M. Voas and K. W. Miller, "Software testability: The new verification," *IEEE Softw.*, vol. 12, no. 3, pp. 17–28, 1995.

[6] J. E. Hopcroft and J. D. Ullman, *Formal Languages and Their Relation to Automata*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1969.

[7] J. E. Hopcroft, R. Motwani, and J. D. Ullman, "Introduction to automata theory, languages, and computation," *Acm Sigact News*, vol. 32, no. 1, pp. 60–65, 2001.

[8] 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, Jun. 2004.

[9] "ChibiOS free embedded RTOS - ChibiOS Homepage." [Online]. Available: http://www.chibios.org/dokuwiki/doku.php. [Accessed: 26-Sep-2018].

[10] 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.

[11] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Autom. Softw. Eng.*, vol. 10, no. 2, pp. 203–232, 2003.

[12] D. R. Kuhn, J. M. Higdon, J. F. Lawrence, R. N. Kacker, and Y. Lei, "Combinatorial methods for event sequence testing," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, 2012, pp. 601–609.

[13] 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.

[14] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard, "An Evaluation of Exhaustive Testing for Data Structures," p. 18.

[15] 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, Apr. 2005.

[16] S. Misailovic, A. Milicevic, N. Petrovic, S. Khurshid, and D. Marinov, “Parallel test generation and execution with Korat,” in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering - ESEC-FSE ’07*, Dubrovnik, Croatia, 2007, p. 135.

[17] J. H. Siddiqui and S. Khurshid, “PKorat: Parallel Generation of Structurally Complex Test Inputs,” in *2009 International Conference on Software Testing Verification and Validation*, Denver, CO, USA, 2009, pp. 250–259.

[18] A. Celik, S. Pai, S. Khurshid, and M. Gligoric, “Bounded exhaustive test-input generation on GPUs,” *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 1–25, Oct. 2017.

[19] P. Runeson, “A survey of unit testing practices,” *IEEE Softw.*, vol. 23, no. 4, pp. 22–29, 2006.

[20] “Unit Testing | Benefits Unit Testing | Limitations of unit testing | Rules of Unit Testing | Software Testing Information.” [Online]. Available: http://www.onestoptesting.com/unit-testing/. [Accessed: 26-Sep-2018].

[21] S. R. Dalal *et al.*, “Model-based testing in practice,” in *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, 1999, pp. 285–294.

[22] M. Utting, A. Pretschner, and B. Legeard, “A taxonomy of model-based testing approaches,” *Softw. Test. Verification Reliab.*, vol. 22, no. 5, pp. 297–312, Aug. 2012.

[23] J. Callahan, F. Schneider, and S. Easterbrook, “Automated software testing using model-checking,” in *Proceedings 1996 SPIN workshop*, 1996, vol. 353.

[24] G. Fraser, F. Wotawa, and P. E. Ammann, “Testing with model checkers: a survey,” *Softw. Test. Verification Reliab.*, vol. 19, no. 3, pp. 215–261, 2009.

[25] G. Hamon, L. De Moura, and J. Rushby, “Generating efficient test sets with a model checker,” in *Proceedings of the Second International Conference on Software Engineering and Formal Methods, 2004. SEFM 2004.*, 2004, pp. 261–270.

[26] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou, “Testing real-time systems using UPPAAL,” in *Formal methods and testing*, Springer, 2008, pp. 77–117.

[27] Y. Li, T. Dong, and Y.-D. Song, “Using Grid Computing for Distributed Software Testing.”

[28] A. Duarte, W. Cirne, F. Brasileiro, and P. Machado, “Gridunit: software testing on the grid,” in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 779–782.

[29] “What is Static Testing? - Software Testing Class.” [Online]. Available: http://www.softwaretestingclass.com/static-testing/. [Accessed: 26-Sep-2018].

[30] “Embedded Security.” [Online]. Available: https://www.mathworks.com/products/polyspace/application-security.html. [Accessed: 26-Sep-2018].

[31] "FrontPage - Hadoop Wiki." [Online]. Available: https://wiki.apache.org/hadoop. [Accessed: 26-Sep-2018].

[32] "CMSC 603 High Performance Distributed Systems: Apache Hadoop." VCU, Spring-2017.

[33] A. Duarte, G. Wagner, F. Brasileiro, and W. Cirne, "Multi-environment software testing on the Grid," in *Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*, 2006, pp. 61–68.

[34] L. Yu, Y. Lei, R. N. Kacker, and D. R. Kuhn, "ACTS: A Combinatorial Test Generation Tool," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, Luxembourg, Luxembourg, 2013, pp. 370–375.

[35] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker b last," *Int. J. Softw. Tools Technol. Transf.*, vol. 9, no. 5–6, pp. 505–525, 2007.

[36] D. Beyer and M. E. Keremoglu, "CPAchecker: A tool for configurable software verification," in *International Conference on Computer Aided Verification*, 2011, pp. 184–190.

[37] G. Madl and S. Abdelwahed, "Model-based analysis of distributed real-time embedded system composition," in *Proceedings of the 5th ACM international conference on Embedded software*, 2005, pp. 371–374.

[38] G. Madl and N. Dutt, "Model-based analysis of event-driven distributed real-time embedded systems," PhD Thesis, University of California, Irvine, 2009.

[39] A. P. Mathur, *Foundations of software testing, 2/e*. Pearson Education India, 2013.