INL/EXT-19-54096
Revision 0

Light Water Reactor Sustainability Program

# REALIZATION OF AN AUTOMATED T-WAY COMBINATORIAL SOFTWARE TESTING APPROACH FOR A SAFETY CRITICAL EMBEDDED DIGITAL DEVICE

June 2019

U.S. Department of Energy

Office of Nuclear Energy

INL/EXT-19-54096
Revision 0

# REALIZATION OF AN AUTOMATED T-WAY COMBINATORIAL SOFTWARE TESTING APPROACH FOR A SAFETY CRITICAL EMBEDDED DIGITAL DEVICE

**Dr. Carl Elks, Athira Jayakumar, Aidan Collins, Christopher Deloglos, Brandon Simmons, Dr. Ashraf Tantawy and Smitha Gautham**

**June 2019**

**Prepared for**
**Idaho National Laboratory**
**U.S. Department of Energy**
**Office of Nuclear Energy**

# 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 the detailed workflow for conducting testing and the testbed to support Bounded Exhaustive Testing with respect to Combinatorial Test (CT) methods. The report primarily describes the description of the process workflow, testbed architecture, tools, resources, and computing needed to conduct an automated testing process. This information will be used to fully realize the testbed and conduct the experimental study – which is to demonstrate the efficacy of digital qualification via Bounded Exhaustive Testing with respect to Common Cause Failure assessment.

# TABLE OF CONTENTS



# TABLE OF FIGURES

# ACRONYMS

| | |
|---|---|
| BVA | Boundary Value Analysis |
| CCF | common-cause failure |
| CCM | Combinational Coverage Measurement |
| CT | combinatorial test |
| CTM | Combinatorial Tree Method |
| DUT | device under test |
| I&C | instrumentation and control |
| I/O | input/output |
| JML | Java modeling language |
| MC/DC | modified condition/decision coverage |
| NASA | National Aeronautics and Space Administration |
| NIST | Nation Institute of Standards and Technology |
| NPP | nuclear power plant |
| NRC | Nuclear Regulatory Commission |
| PIL | processor in the loop |
| RTOS | real-time operating system |
| SUT | software under test |
| VCU | Virginia Commonwealth University |

# Realization of an Automated T-Way Combinatorial Software Testing Approach for a Safety Critical Embedded Digital Device

## 1. Introduction

As digital upgrades to U.S. nuclear power plants (NPPs) 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) identifies two design methods that are acceptable for eliminating CCF concerns: (1) diversity or (2) testability (specifically, 100% testability as identified in BTP 7-19, Rev. 7, "Guidance for Evaluation of D3 in Digital Computer-Based Instrumentation and Control Systems," U.S. NRC, August 2016 - Accession No. ML16019A344). As pointed out in [1], there is near universal consensus among computer scientists, practitioners, and software test engineers that exhaustive testing for modestly complex devices or software is infeasible [2], [3] – this is due to the enormous number of test vectors (e.g. all pairs of state and inputs) needed to effectively approach 100% coverage [4]. For this reason, diversity and defense-in-depth architectural methods for computer-based I&C systems have been become the norm 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 SW testing provide very strong "coverage" of the state space, and (2) can these methods be effective in establishing credible evidence of software CCF reduction. In the previous report, we identified several promising testing approaches that purport to provide strong "coverage". Namely, among the methods reviewed were Combinatorial Testing (CT) methods that can achieve "bounded" exhaustive testing under certain conditions [6]. Additionally, the term coverage requires some farther elaboration as to classify among the several definitions used in the SW testing community – and we provide that discussion in latter sections.

In this document, we define the test approach for an empirical study or test to collect data on the efficacy of CT methods for accomplishing bounded-exhaustive testing.

### 1.1 Purpose

In this document we are focused on describing the realization of a testbed for conducting automated empirical software testing of embedded digital devices with respect to bounded exhaustive testing. We are mainly focused on methods that support or claim high levels of "coverage" approaching exhaustive testing or bounded exhaustive testing. By bounded exhaustive testing we mean:

> **Definition:** The term *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 – enabling 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 [7]–[9].

**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 [1], [10].

## 1.2 Test Objectives

The approaches, methods, and technologies described herein are mainly focused on testing actual software for embedded digital devices—i.e., testing with actual inputs stimulating the software under test. Our objective for this research is to develop a test approach and methodology to enable a study on the efficacy of t-way combinatorial testing for embedded digital devices. To support this specification, we will develop questions 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 combinatorial testing 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: Is t-way combinatorial testing effective at discovering logical and execution-based flaws in nuclear power software-based devices (device under test [DUT])?

## 1.3 Scope

The scope of this test specification is focused on t-way combinatorial testing methods, technology and supporting tools required to effectively carry out a well-formed study to answer Questions 1 and 2.

# 2. Background: Combinatorial Testing

As software grows in size and complexity, software tests covering all the interactions between the data, environment, and configuration is challenging. The studies conducted by the National Institute of Standards and Technology (NIST) [**Error! Bookmark not defined.**] on software failures in 15 years of Food and Drug Administration Medical Device recall data concludes that a majority of the software failures are due to interaction faults arising from the interaction of a few parameters, mostly by two and three. According to the National Aeronautics and Space Administration (NASA)-distributed database, 67% of 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 **Error! Reference source not found.**. Applying the rule that the interaction between t or fewer variables is 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. Combinatorial testing is particularly suited to help detect problems like this early in the testing life cycle.

The key insight underlying *t*-way combinatorial testing 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 **Error! Reference source not found.**, 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 software with all possible 6-tuple input parameter combinations can lead to tracking down all 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, which 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 quality of testing [**Error! Bookmark not defined.**].

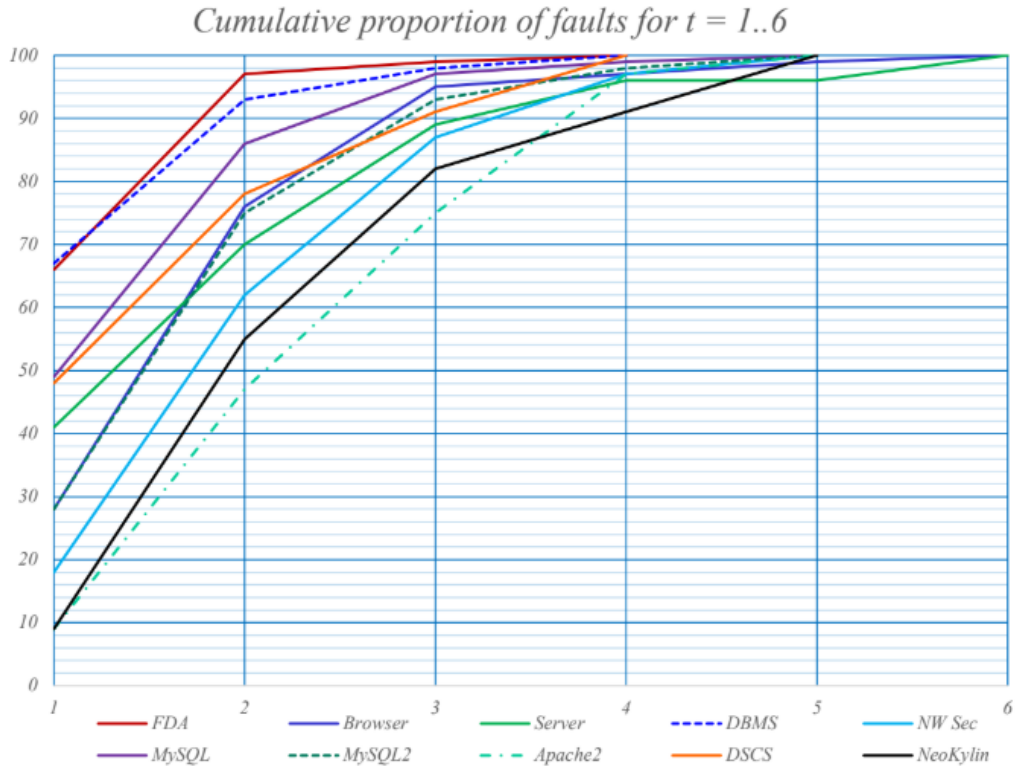


Figure 1. Cumulative proportion of faults for t (number of parameters) = 1 . . . 6 *[7]*

The two most-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 have 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 relevant parameters. This should include the user- and environment-interface parameters and the configuration parameters. The second step is to determine values for these parameters. Using the entire set of values for all 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: equivalence partitioning, boundary-value analysis, category partitioning, and domain testing. As a 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 constraints on the interactions, which define the set of impossible parameter interactions, is also vital for obtaining the expected software coverage [12].

R. Kuhn and V. Okun's work on "Pseudo-Exhaustive Testing for Software" [**Error! Bookmark not defined.**][12] 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 described 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 test cases. By using t-way coverage of the variables as the test criterion, we can derive the combinatorial test cases. Temporal logic expressions in the form AG(v1 & v2 & ... & vt -> AX !(R)) which direct 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 Kuhn et al in using a 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.

R. Kuhn's (of NIST) and J. Higdon's research work on extending the application of combinatorial testing to event-driven systems, described in the paper, "Combinatorial Methods for Event Sequence Testing," [**Error! Bookmark not defined.**], also proves to be noteworthy for systems of the type found in NPPs. Some faults in the software are 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 the t-way order of t events in a software. The basic concept of sequence covering is that, if we have a two-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 in detecting most 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.

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 greedy algorithms and heuristic methods. Bryce et al.'s greedy algorithm for test-case 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.

# 3. Conceptual Experiment Process

Figure *2* 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, and n. Each of the variable space parameters (v and n) defines the input model. The input model is pre-analyzed (e.g. BVA) to determine equivalence partitions. The input model define is used define the "list" of experiments – and this can be done parametrically (one factor at a time), or by Design of Experiment methods. The list of "covering" test vectors is generated to produce a reduced optimal set of test vectors. 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 under test 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: Set of pass/fail, coverage metric (% of covering array), and a metric related to % of state space examined. The process continues until there are no more variations on the parameter sets 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). Prior to the experiments being conducted it is often good practice to execute tests on baseline fault sets to determine the capacity of the method to detect faults. The key point is that the testbed implementation must be designed from experimental viewpoint with respect to the test objectives. Below we discuss our proposed testbed environment and process to support the execution of experiments.

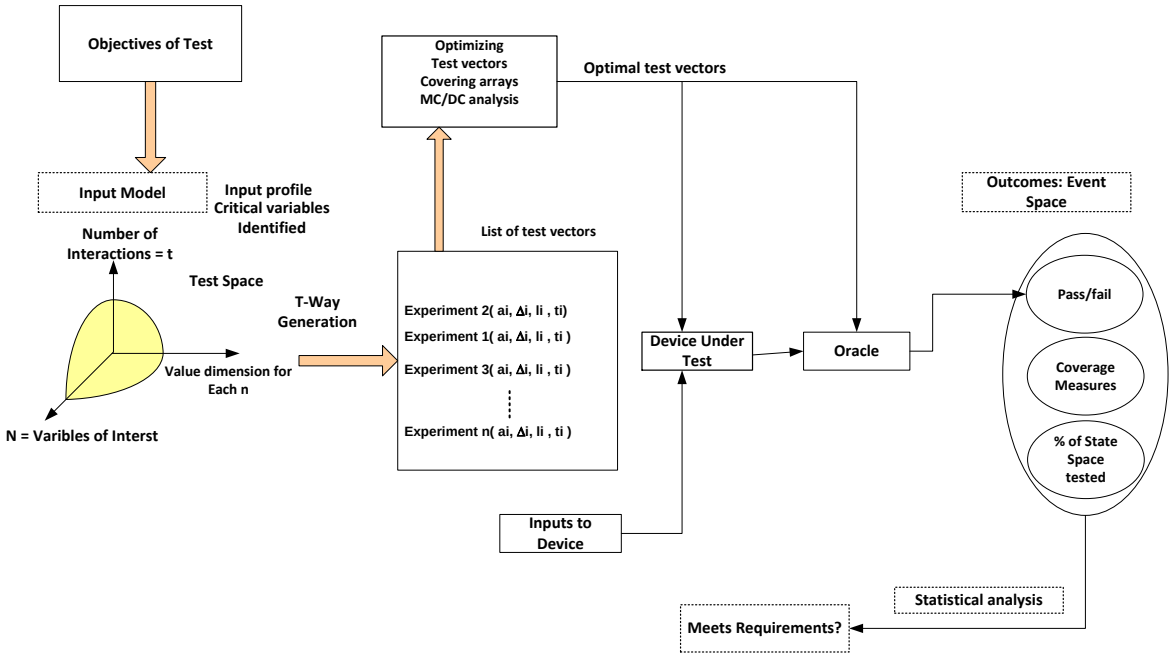


Figure 2. Conceptual view of a bounded exhaustive testing process

## 4. Testbed Architecture and Workflow

In this section, we present an implementation perspective of how to realize an automated test environment to support conceptual T-way testing on a real time embedded digital device. Note what is described in this section is our preliminary design and it is still evolving. **Error! Reference source not found.** presents the basic test environment architecture with respect to tools, systems and components needed to support the BET experiment process. There are three "SW tools" that figure prominently in our testbed; (1) Razorcat's TESSY, (2) NIST ACTS, and (3) Keil Interactive Debugger. The TESSY automated testing tool is developed by Razorcat (https://www.razorcat.com/en/product-tessy.html). TESSY's primary purpose is for testing safety critical embedded system software. It provides a number of features to assist in automation, unit testing, integration testing, and regression. TESSY supports test management, including requirements, coverage measurement, and traceability. For our purposes, the automation features of TESSY are of significant interest. The NIST ACTS tools provide a full set of

features and capabilities to support T-way combinatorial testing. This includes support for BVA, efficient T-way test vector, sequences, fault location and covering array generation. Finally, the Keil interactive debugger tool allows real time interfacing to the target device through the OCD ports of the device (e.g. JTAG, SWD, ETM, etc..). The "debugger" interface allows; (1) test vectors to be directly interfaced to DUT, (2) unobtrusive extraction and monitoring of all types of data including I/O data, internal variables, intermediate variables, state variables, conditionals and guards.

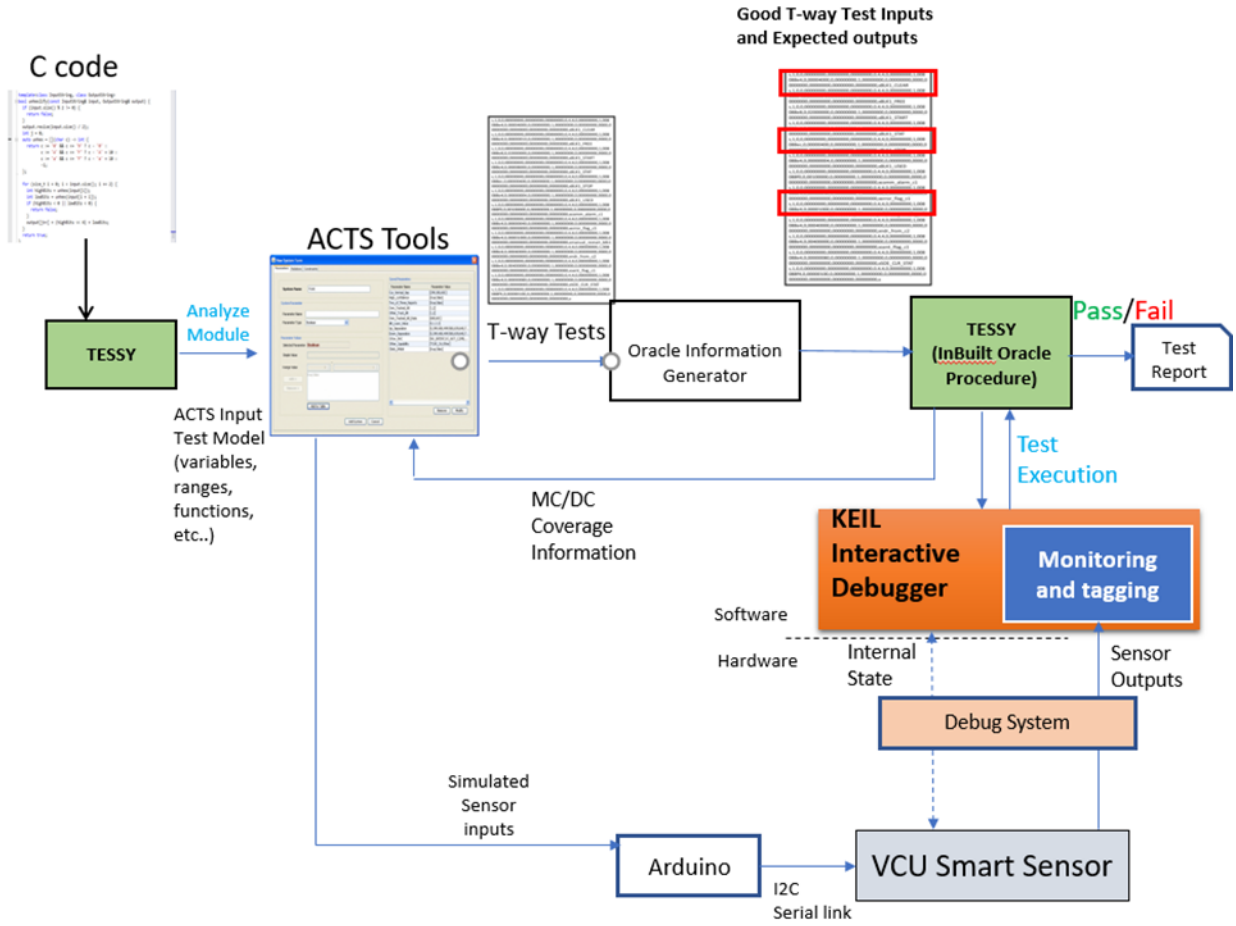


Figure 3. Baseline testbed architecture

Partially or fully automated testing schemes are preferable to optimize testing efficiency. Automation in this context refers to following items:

- Translation of ACTS Test-Case data from text to code Smart-Sensor
- Recording of Smart-Sensor state/output variables on a per-test-case cycle
- Appending test-case value with time-stamped smart-sensor data
- Execution of test
- Comparison of Smart-Sensor results to Oracle results

## 4.1 Automated Test Environment and Workflow

The preceding sections discussed the process steps of how to realize or build a bounded-exhaustive testing study. This section presents an implementation perspective on how to realize a test-environment setup. Note that what is described in this section is just one of many ways to realize the process outlined in Section 3. This section describes the basic workflow of the automated testbed shown in figure 3.

1. **Development of the Input model**. The first step is to parse the code to reveal the variables instances, data structures, parameters, and constants the code embodies. This will be done with the TESSY tool. Input-Output model for each function in c file can be exported from the Test Interface Editor in TESSY. This export can be into any of the following formats: tcx, xls, xlsx, script, csv, txt or yml.

2. **Boundary Value Analysis** (BVA). Second step is to determine the representative values for the input model – this is the v parameter. Using the entire set of values would lead to infeasible 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. In step 2 we can use Input Partitioning and Boundary value analysis either in TESSY or in ACTS.

3. **TESSY-ACTS interface**. If using the TESSY tool for BVA on the input model, we must export the Input model to ACTS. At present we convert the input model to format that ACTS Tools understands using a script file.

4. **Generate T-way Test vectors (ACTS)**. Given the input model, ACTS will generate sets of T-way test vectors. The user decides on specific experimental parameters to control (e.g. unit test, t, n, and sequence). This first pass through ACTS generates T-way tests without MC/DC coverage information. A second pass will factor in MC/DC coverage information to refine test vectors.

5. **Test Case import**. The complete test cases are then imported into TESSY in one of the following formats: tcx, xls, xlsx, script, csv, txt or yml.

6. **Oracles** - Even with efficient tools and procedures to produce input models, 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's going to do it. In "*Automated Software Test Implementation Guide for Managers and Practitioners, STAT COE-Report-05-2018*" - appendix E defines a number of approaches for developing oracles for SW testing. We are currently reviewing these methods to determine a feasible sub-set of approaches.

7. **Oracle Integration**. Once an Oracle approach has been identified, TESSY has basic features built in to support oracles in an automated testing framework. At the basic level, TESSY allows one to submit already generated t-way test vectors into an Oracle Information Generator which analyzes all test vectors and adds in the expect output values for all the test input vectors. This can be automated using some scripts or worst case can be done manually in TESSY itself (probably not feasible for large test sets. need to figure out). Important note: TESSY provides "hooks" to include oracles in the automated testing process. The design of the oracle is left up to the testers to realize. This is an open issue at the moment.

8. **Test case execution**. – Executing the tests requires an automated test environment where test vectors are submitted to the smart sensor, and results are cataloged. The key aspect is to collect data in manner that is tractable and supports the test objectives. TESSY runs test cases directly on the actual Smart sensor target (STM32F4 uC) by interfacing with the Keil debugger. http://www2.keil.com/mdk5/debug. TESSY in concert with the Keil debugger ID's and time stamps all

data. TESSY reads in the variable information within the smart sensor code via the Keil debugger interface.

9. **Outcome evaluation**. Assuming we have designed and integrated an oracle procedure into TESSY, we can compare expected outputs against actual outputs while running test cases. Test Results are generated – Passed/Failed.

10. **Refining Test cases**. After the initial test execution, MC/DC coverage data can be generated via TESSY. This coverage data can be fed back into the ACTS tools (if possible) or analyzed manually to refine tests in TESSY to improve the test coverage on the code. Also, Automatic Test generator and Scenario based Test method in TESSY can also be used to improve coverage. Scenario Test cases in TESSY can be used to cover for scenario based test cases that do not involve input value combinations but instead involve sequential invocation of various functions in a particular order. ACTS also has a similar feature to generate sequence based CT tests vectors.

11. **Analysis of 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 combinatorial testing to exhaustive testing is desirable, we need to have as many t-way interactions as possible. Selection and analysis of metrics at the beginning of the experiment is important to ensure that experiment can support calculation of the metrics at post analysis phase.

## 4.2 TESSY as the Automation Engine

Since TESSY is somewhat new to the team and is proposed to figure prominently in the test bed, this section describes some basic capabilities.

### 4.2.1 Testing on Target Device

As shown in figure 4, TESSY can execute unit /component level tests directly on the microcontroller. TESSY uses a third-party debugger (specific for the microcontroller) as interface to the test execution environment. TESSY controls the debugger software during the tests [14].

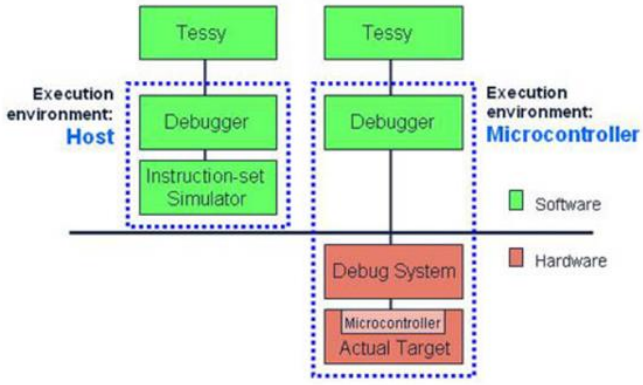


Figure 4. Using TESSY for controlling testing on embedded device.

As shown in figure 5, TESSY also allows one to set breakpoint when the Test object is entered during test execution and further use debugger features like line steps, watch windows, etc to evaluate variable

values. This aids in easily finding the reasons behind deviations in output values from their expected values that caused test case failures [15].

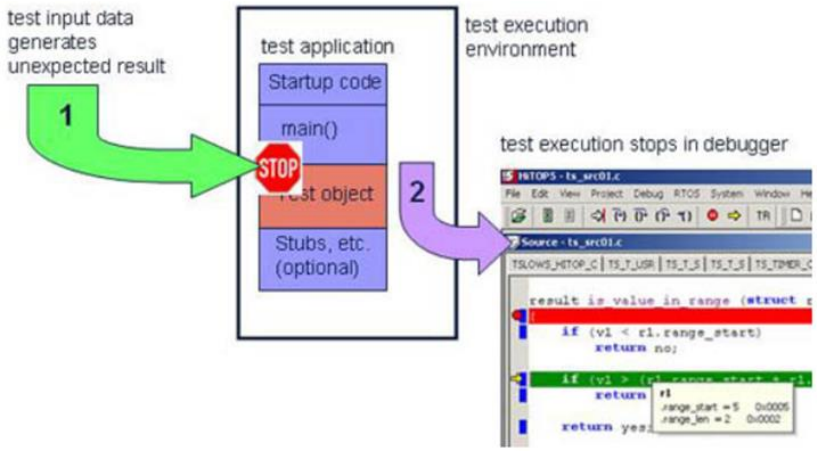


Figure 5. Breakpoint Feature - A failed test case can be debugged by using the breakpoint feature of TESSY

### 4.2.2 Automatic analysis of functions

TESSY analyzes the passing directions of functions automatically and stores its findings in the interface database. Passing direction of function parameters/variables that reflects the kind of usage while testing the test object can be of the type IN, OUT, INOUT, EXTERN or IRRELEVANT. Variables passed into or received from low level driver or ChibiOS functions if not considered in the scope of testing can be marked as IRRELEVANT type so that these elements will not be visible for testing activities.

### 4.2.3 Classification Tree Method

The basic concept of the Classification Tree Method is to first partition the set of possible inputs for the test object separately and from different aspects, and then to combine them to obtain redundancy-free test cases covering the complete input domain. Boundary Value Analysis (BVA) attempts to discover program errors for 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 separate test cases. The boundary values are determined after the equivalence classes are constructed, and could be tested independently and in parallel with the other test cases. Values at the borders of a range of values are better suited to form error-sensitive test cases than values in the middle. Boundary values can be expressed in Classification Tree Method (CTM).

In the figure 6 below, a classification tree created in TESSY gives a typical example of how input domain can be partitioned into equivalence classes and further into boundary values. It shows an integer variable ‘start’ being divided broadly into 3 classes positive, negative and zero. In the ‘positive’ class, start is assigned a value 3, in the ‘negative’ class start is assigned a value -3 and zero class with value 0. Further the classification tree is extended with boundary class. The class start is further subdivided according to the size into two classes normal positive and maximal positive, such that one class denotes normal positive values and other class denotes maximum positive value. The highest positive value existing in the given integer range is used for start in the ‘maximal positive’ class. Using extreme values (or boundary values) in test cases is well-suited to produce error-sensitive (or interesting) test cases. This satisfies mathematical completeness. Similarly the negative class is also partitioned into normal negative and maximal negative classes. The input variables length and position are also divided into equivalence and boundary classes.

Afterwards test cases are specified in the so-called combination table below the Classification Tree. A line in the combination table depicts a test case. The test case is specified by selecting leaf classes, and by setting markers in the line of the respective test cases in the combination table. The objective of the CTM is to find a minimal, non-redundant but sufficient set of test cases by trying to cover several aspects in a single test case, whenever possible. Here it shows 14 test cases created by marking various combinations of input classes.

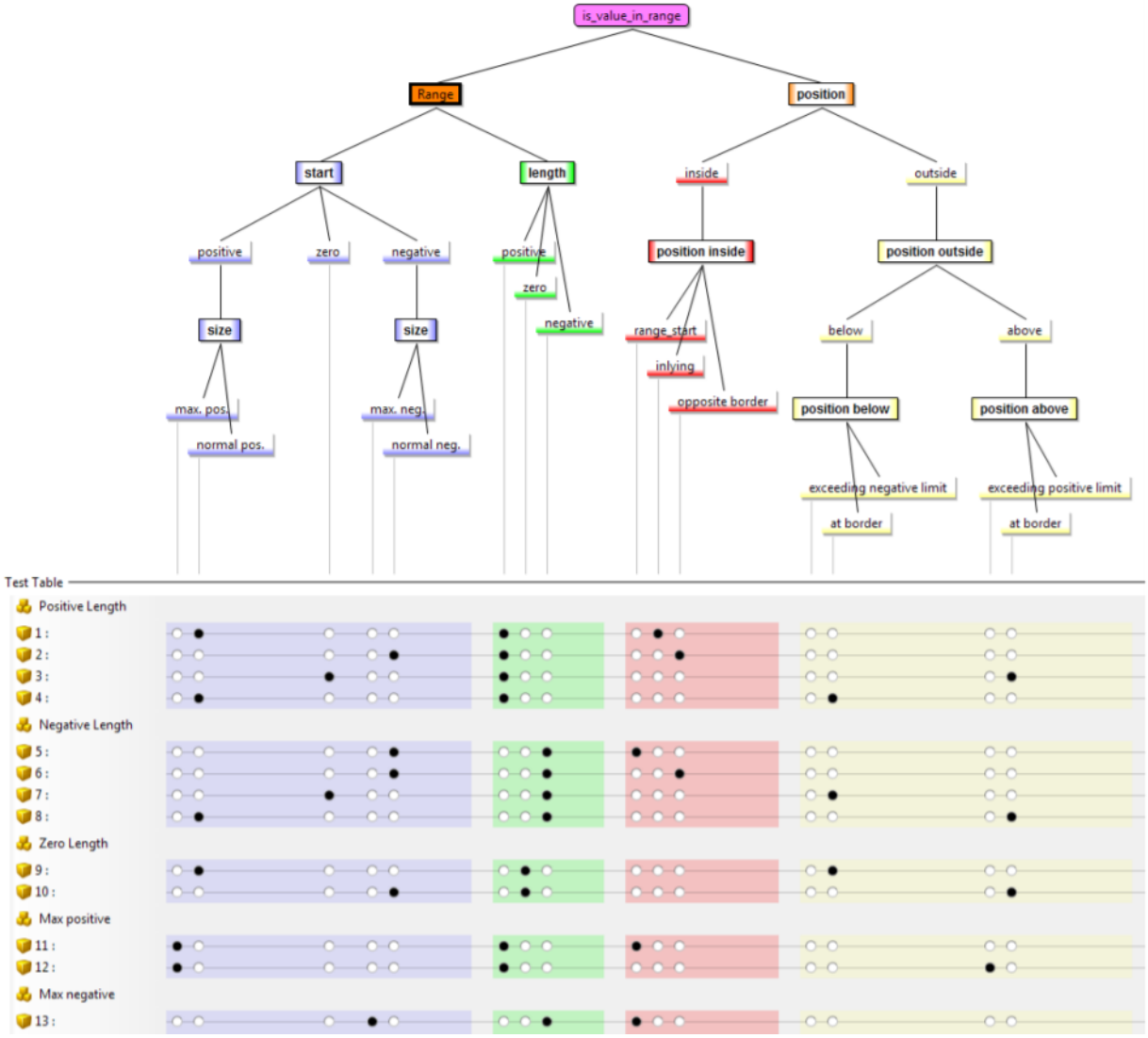

Figure 6. Classification Tree Editor [15]

### 4.2.4 Test Oracle

As we mentioned earlier, oracles can be integrated into TESSY. A test oracle is a mechanism that determines whether software executed correctly for a test case. Test oracle can be defined to contain two essential parts: oracle information that represents expected output; and an oracle procedure that compares the oracle information with the actual output [16]. In TESSY, we can provide the **oracle information** by manually writing in the expected values for the output variables for each test-case. In the below figure 7, 'result' variable is the return value for the function and this return value being a Boolean variable can be

expected to be 0(no) or 1(yes) which automatically appears in the drop down menu. Based on the test inputs and the knowledge of the algorithm in the function, the expected value of the result variable is written in the Test editor for all the test-cases. The **Test Oracle procedure** inbuilt in TESSY compares the provided expected output values in the Test Data Editor, against the actual outputs while executing the test cases and result in Passed or Failed test-cases.

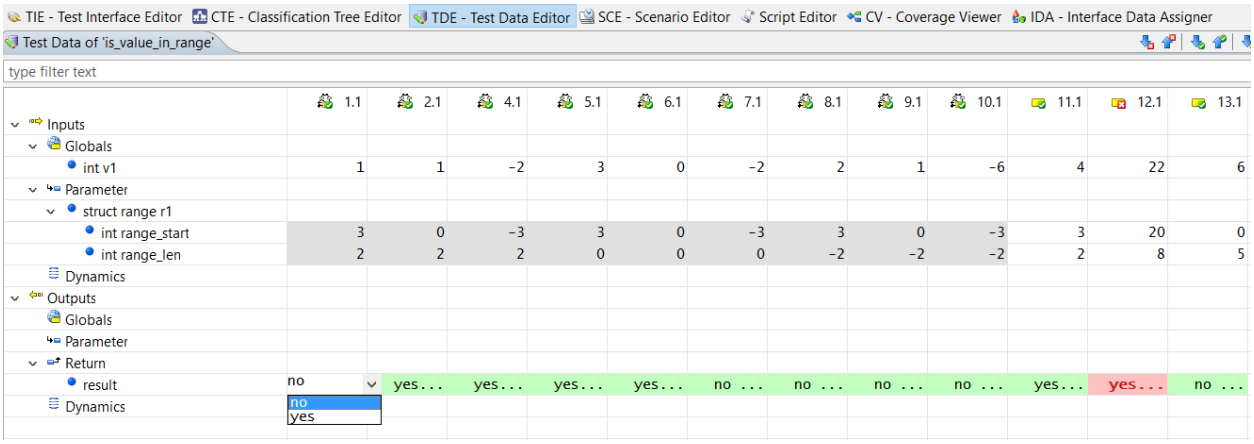


Figure 7. Test Data Editor

In figure 8 below shows the basic TESSY oracle results window. If indicates Pass/Fail after executing the three test cases on the source code. It also shows which output variable has deviated from its expected value in the failed test case.

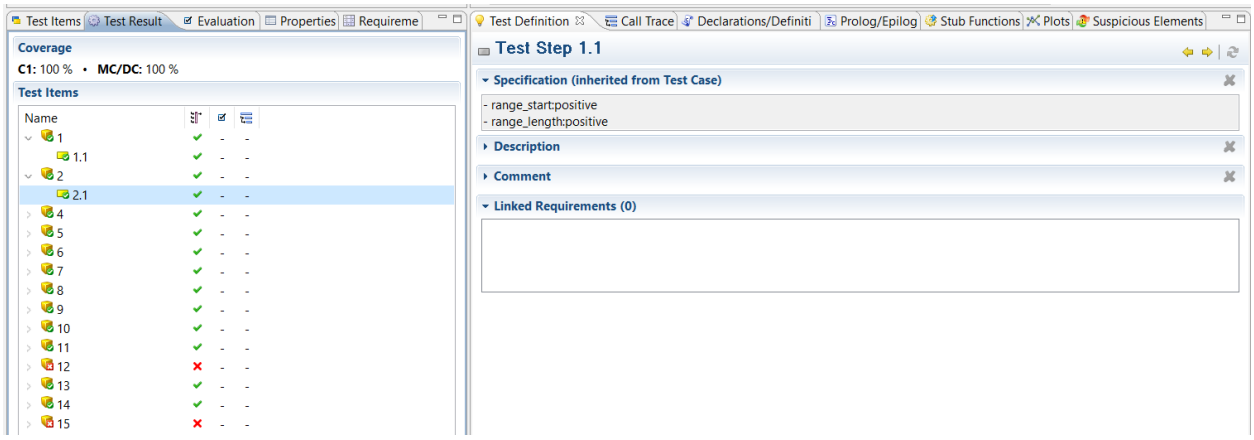


Figure 8. Test Oracle Results Window

### 4.2.5 Scenario editor

Sequence dependent tests are often required for real time control applications. Both ACTS and TESSY provide the ability to generate sequence dependent test cases. In TESSY, scenarios can be designed to test the temporal behavior of the component. In TESSY, dependent test sequences are created by writing time based sequence steps. Periodical invocation of functions can be simulated by specifying the tick rate of the main function under test. [15]

A scenario test in TESSY involves two tasks:

1. Stimulating the component under test by adding calls to external functions that stimulates the functionality we are testing in the component.
2. Checking the reaction of the component after the stimulation which is basically verifying the expected behavior of the component/function under test.[15]

Checking expected behavior of the component under test is done by verifying return values of functions, or global variable values or checking if external functions are being called at right instances, or checking the parameters in calls to external functions.

In the example figure 9 below, a 'tick' function is set to Work Task (main function under test) and set to a cycle time of 10ms. Time steps when added automatically spaces out to 10ms steps. Set_sensor_door function is called at 10ms. When the set_sensor_door function call with parameter d=closed is called at 10ms, the LightOn external function is expected to be called by the component under test. Checking of calls to underlying external functions can also be added to the desired time step by drag and drop. The LightOn() [10-20ms] indicate that the LightOn function call is expected to happen any time between 10 and 20ms.Thus we are extending the time frame of the reaction by 10ms. Whereas, LightOff() function is expected to be called exactly at 40ms.

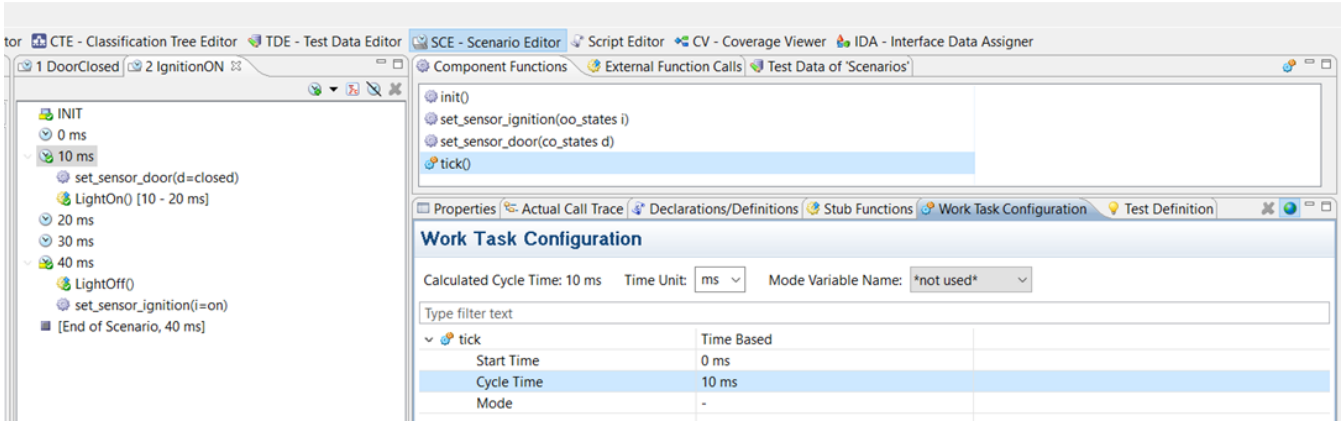


Figure 9. Scenario Editor

### 4.2.6 Stub functions

TESSY supports unit testing, as such, it can create stubs for external or local functions that the function under test is calling. For the VCU Smart Sensor these would be external functions in ChibiOS operating systems or low level drivers. Some of these functions are not relevant for testing the application (green fiSles) can be stubbed with empty functions or dummy code.

### 4.2.7 Coverage Viewer

A software test object consists of program constructs like blocks of code, logic, branches, conditions, guards, etc. Code coverage measures how many of these constructs were exercised during the tests. This metric is related to the reachability of total number of items and is usually expressed as a percentage. TESSY supports seven different coverage measurements. Coverage results are visualized in a graphical flow chart linked with colorized source code views as well as in textual form. Navigation through the flow chart easily reveals uncovered branches and conditions [14], [15]. Below are examples of coverage metrics supported in TESSY.

1. C0 (Statement Coverage) - It is a metric, which is used to calculate and measure the number of statements in the source code which have been executed.
2. C1 (Branch Coverage) - Branch coverage measures the coverage of both conditional and unconditional branches.

3. DC (Decision Coverage) - Decision coverage measures the coverage of conditional branches

4. MC/DC (Modified Condition / Decision Coverage) – This coverage criterion requires each condition constituting the decision to independently change the decision outcome during the test execution.
5. MCC (Multiple Condition Coverage) - In Multiple Condition Coverage for each decision, all the combinations of conditions should be evaluated.
6. EPC (Entry Point Coverage) - only for unit tests
7. FC (Function Coverage) - only for component tests – Checks if all functions are covered during test execution.

In figure 10 below shows the MC/DC Coverage for in a c file and its control flow graph. Red indicates decisions with not taken paths and green indicates fully covered decisions.

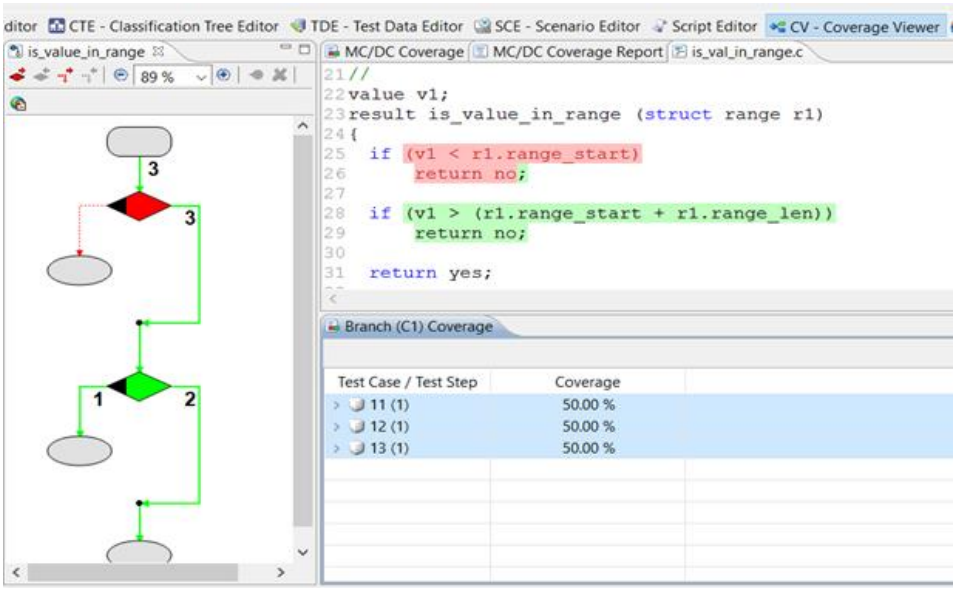

Figure 10. Coverage Viewer

### 4.2.8 Automatic Test Case Generator

This feature of TESSY is for automatically generating test cases from specified input value ranges. It generates test cases which involve all combinations of input values for multiple inputs.
As shown in figure 11 below, currentRawPressure variable is set to value range [2:8] which means it will be assigned 7 different values 2,3,4,5,6,7,8 and currentRawTemperature variable is assigned range [1:5] that means it will be assigned 5 different values 1,2,3,4, 5. Thus there are 35 test cases generated with all combinations of the two inputs. In figure 11 below shows 16 of those test cases.

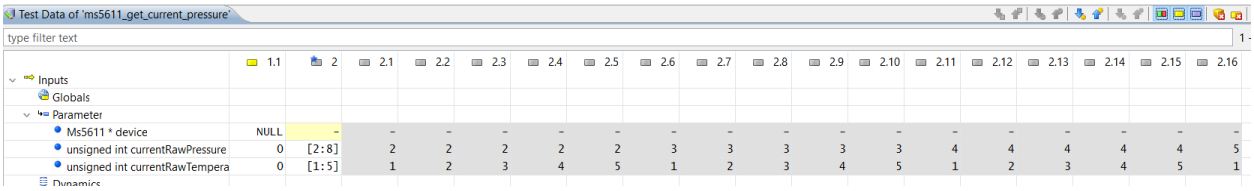


Figure 11. Test Generator Function

# 5. Tools and Resources

A number of commercial and open source tools are available to assist the experimenter in conducting the study. The main combinatorial testing tools from NIST and RazorCat are:

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

Additional resources that may be utilized to construct and support the automated testing architecture include:

1. Device under Test Software – VCU Smart Sensor SW code basis
2. Keil uVision debugger ST-Link debug software
3. Cigwyn32
4. ST STM32F405 – ARMM4 Cortex processor development board.
5. Ubuntu 16.04 LTS 64 Linux
6. GNU11 C programming language, the GNU GCC Compiler Version 7.3.
7. The VCU Software Requirements Specification Document (Github)
8. The VCU Software Design Document (GitHub)
9. Optional: MathWorks Simulink development environment
10. Optional: MathWorks Arduino Toolkit
11. Optional: MathWorks Microprocessor Toolkit
12. Host computers, servers and database software
13. USB-8451 I²C/SPI Interface Device (generated using an Arduino Mega 2560)

# 6. References

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

[2] 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.

[3] 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,” 2018.
[7] 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.
[8] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard, “An Evaluation of Exhaustive Testing for Data Structures,” p. 18, 2003.
[9] 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.
[10] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson, “A Practical Tutorial on Modified Condition/Decision Coverage,” *NASA*, vol. TM-2001-210876, 2001.
[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, “A density-based greedy algorithm for higher strength covering arrays,” *Softw Test Verif Reliab*, vol. 19, pp. 37–53, 2009.
[14] “Tessy: Hitex: Dynamic Module/Unit Test.” [Online]. Available: https://www.hitex.com/tools-components/test-tools/TESSY/. [Accessed: 02-Jun-2019].
[15] “Tessy V4.1 User Manual Revision 41.024.” Razorcat, Feb-2019.
[16] A. Memon, I. Banerjee, and A. Nagarajan, “What test oracle should I use for effective GUI testing?,” in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, Montreal, Que., Canada, 2003, pp. 164–173.