2020 NDIA GROUND VEHICLE SYSTEMS ENGINEERING AND TECHNOLOGY SYMPOSIUM
MODELING SIMULATION AND SOFTWARE TECHNICAL SESSION
AUGUST 11-13, 2020 - NOVI, MICHIGAN

# USING A MODEL-BASED APPROACH FOR TEST AND EVALUATION

**Retonya Dunning[1], Whit Matteson[1], Richard Wise[1], Jennifer Sharpe[1]**

[1]Georgia Tech Research Institute, Atlanta, GA

## ABSTRACT

*The purpose of this paper is to describe a methodology for applying Model-Based Systems Engineering (MBSE) practices to Test and Evaluation (T&E) practices. The Georgia Tech Research Institute GTRI has developed a process which includes using MBSE tooling & modeling languages, automatic test case generation based on modeling, and requirements coverage thereof. This paper describes the developed process and the benefits that it brings to T&E practices.*

**Citation:** R. Dunning, W. Matteson, R. Wise, J. Sharpe, "Using a Model-based Approach for Test and Evaluation", In *Proceedings of the Ground Vehicle Systems Engineering and Technology Symposium* (GVSETS), NDIA, Novi, MI, Aug. 11-13, 2020.

## 1. INTRODUCTION

Test and Evaluation practices can be one of the most time-consuming steps in system development. Some test cases, such as software tests, can be automatically executed; however, engineers must still manually write test cases in order to verify requirements and test scripts that execute the test cases. The GTRI has conducted a pilot model-based approach leveraging the Systems Modeling Language (SysML) and the UML Testing Profile (UTP) to expedite the Test and Evaluation (T&E) craft by automatically generating test cases based upon a canonical system specification. Using a model-based approach creates a sustainable and maintainable process to produce test cases that can be run in an existing test infrastructure.

GTRI has developed a procedure to utilize Model Based Systems Engineering (MBSE) in the creation of a test suite to verify requirements. This process uses a SysML model of the expected system behavior to generate a test suite to run in an existing test environment. The major steps in this procedure involve importing requirements into a SysML modeling tool, creating a SysML state machine to model the expected behavior of the system-under-test, utilizing open source tools to traverse the model to generate a test suite containing abstract test cases, tying the abstract test cases to concrete, executable test cases, and exporting the test suite in a format digestible by the test environment. Each of these steps is detailed in subsequent sections.

## 2. BACKGROUND

Currently, test engineers primarily manually develop test cases based on system requirements, which may require the generation of hundreds or even thousands of test cases. This often leads to missed test cases, redundant test cases, and other inefficiencies that result in suboptimal T&E. Auto-generation of test cases from a comprehensive, unified model leads to greater test coverage and more efficient test execution.

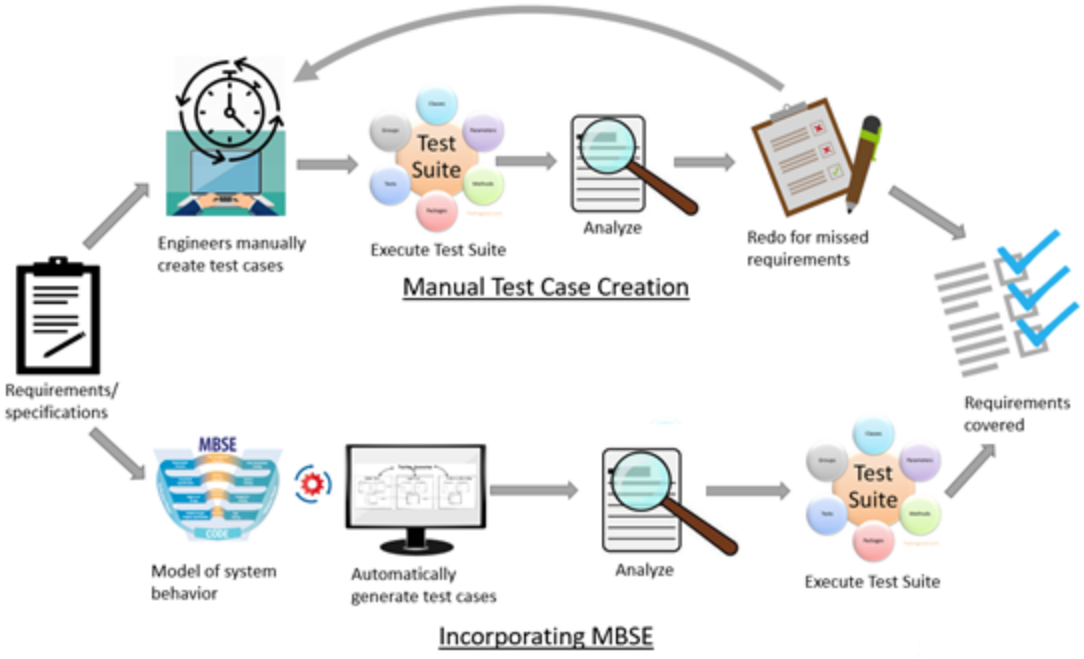


**Figure 1**. Manual Test Case Creation vs. Incorporating MBSE in Test Case Creation

In general, there is a need to further bridge the gap between the system architecting and T&E communities. Presently, system requirements exist separately from the selection and design of T&E methods that will be applied to the System-Under-Test. There is no formal, model-based linkage between these concepts that can help identify and characterize the coverage and gaps of the T&E environment. Since T&E activities are required to be independent of system design to a degree, this effort expounds on a formal method created that applies MBSE directly to the design and selection of T&E analyses. Figure 1 above depicts the two test case generation processes.

Many engineering activities, including T&E, use models; but they are disjointed. There is no underlying model providing the "glue" needed to have a coherent, synchronized set of views for T&E stakeholders. An MBSE approach seeks to establish a comprehensive description of the interested system, including but not limited to a SysML model. A well-formed, consistent, and repeatable model developed using MBSE provides the basis for developing and refining custom validation rules similar to Unit Testing in software architecting and design. Unit Tests are typically automated tests written and run by software developers to ensure separate sections of software code behave as expected and meet the intended design. In a similar fashion, this approach can be applied to SysML behavior diagrams such as state-machine and activity diagrams, which models the sequence of events for a single object. Ultimately, this will facilitate the auto-generation of test cases with the MBSE environment, providing traceability back to system requirements.

Auto-generating test suites is effective both cost- and time-wise, as the engineers who would have originally had to manually create these now have time to focus on ensuring fidelity, accuracy, and overall adherence to the original requirements. Success can be measured quantitatively by

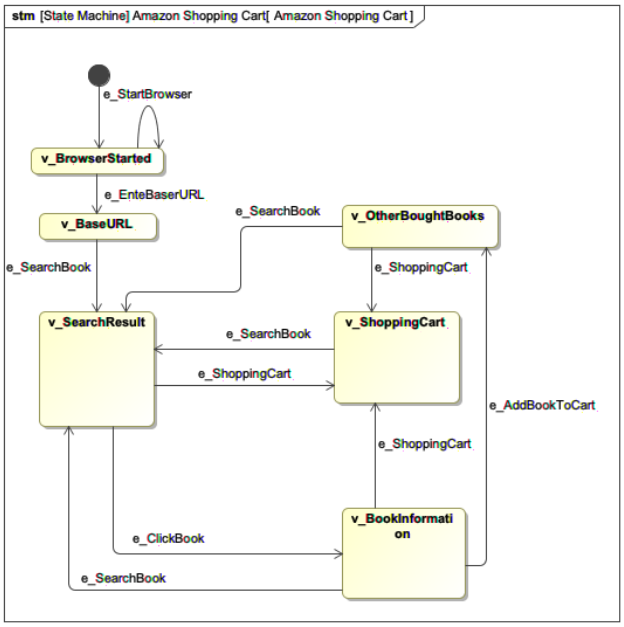


**Figure 2.** SysML Model of Expected Behavior of System

examining percentage of test coverage via auto generated cases and the percentage of invalid auto-generated test cases. This metric, among others, indicates how well T&E can be adequately captured in an MBSE environment in support of a digital platform ecosystem.

Additionally, no open-source tool currently exists which efficiently facilitates all steps of this proposed process. Before developing this approach, the researchers took time to explore open-source and public solutions for auto-generating test cases. Some temporary solutions were found, but each one had a roadblock that caused the researchers to look into creating a holistic approach.

Ultimately, model-based testing – particularly auto generation of test cases – facilitates more complete testing of systems. This minimizes risks of system failures, creates an efficient framework for regression testing and functional testing of new features as the systems evolve, and allows for earlier detection and resolution of discrepancies between system design and requirements.

## 3. CREATING A MODEL FOR T&E

The first step in this model-based process is to create a model by importing requirements into Cameo Systems Modeler. The requirements utilized in this investigation began in a CSV/Excel format. From there, two methods were identified for importing requirements in a CSV/Excel format into Cameo. The first method involves importing the requirements from a CSV/Excel file into a requirements table, while the second involves using the Import CSV plugin within Cameo Systems Modeler.

Once the requirements are imported into Cameo Systems Modeler, the next step in the process is to create a state machine of the expected behavior of the system-under-test, based on requirements and SME knowledge of the system. In the state machine, the states represent the various conditions of the system, and the transitions represent directed relationships between states. Figure 2 shows a simple example of this.

It is important to model the system-under-test accurately, as the validity of the generated test cases is dependent on the accuracy of the state machine. There are several useful elements to aid

in this, namely the use of actions and guards. Actions are called behaviors associated with triggers or states to be executed, and can be utilized to set, increment, and/or decrement variables. Guards are conditions that must be met before a transition can occur [1]. Once an action is utilized to set, increment, or decrement a variable, the guard can then be utilized to make sure the variable meets a specific condition before a transition can be triggered toward a new state. This state machine of the expected behavior of the system-under-test facilitates the automatic generation of test cases in a subsequent step, by using opaque behaviors within a framework developed in SysML and UTP to generate a JSON file representing the state machine for the open-source tools.

## 4. AUTOMATIC GENERATION OF TEST CASES

### *4.1. Abstract Test Case Generation*

Once an accurate state machine has been constructed, an extensible and executable framework operates upon it in order to generate test cases by iterating through different paths through the state machine, where each path represents a potential test case. This framework was developed in SysML and UTP, and centers on the integration of the No Magic software tools Cameo Systems Modeler and the Cameo Simulation Toolkit with the open-source tools AltWalker [2] and GraphWalker [3].

AltWalker is a model-based testing framework developed in Python that automates test execution, and relies on GraphWalker to generate a path through a directed graph based on a finite state machine. GraphWalker comes with several different path generators and stop conditions such as "random path generation" and "stop at 100% requirements coverage" to produce a path through the state machine, which represents a single test case. Essentially, AltWalker is leveraged as a low-level of effort entry point to interact with GraphWalker. This is mainly due to the fact that in order to directly interact with GraphWalker, a build automation tool, such as Maven, needs to be used. AltWalker does not require this and can be interacted with through its API or through basic command line scripting.

The framework used for this model-based T&E process includes opaque behaviors, which read in a state machine reflecting the expected behavior of the system-under-test in order to produce an AltWalker schema conformant JSON file. The command-line-interface utility function provided by the Cameo Simulation Toolkit is used to invoke AltWalker, which reads the JSON file, executes GraphWalker, and reports a path through the state machine. This is then wrapped in an iterator to run the path generation n-number of times, as specified by the user, to produce the test cases which form a test suite.

### *4.2. Test Case Coverage of Requirements*

After the paths are generated and the test suite is formed, additional opaque behaviors read the path generation results, ignore duplicates, and import the unique test cases back into the T&E framework for requirements coverage analysis to find the minimal test suite for the maximum requirement coverage. In order to facilitate this, the system requirements are tied to states within the state machine which satisfy each requirement. In this way, the states within the test cases imply requirement coverage, in regards to which requirements will be tested when the test case is executed. It is important to note that one of the stop conditions for path generation is requirement coverage percentage, which the user has the ability to dictate. Depending on the complexity of the model, it may be feasible to set the stop condition to 100% requirement coverage.

AltWalker can be run in either online or offline mode. When run in online mode, the steps are executed as they are generated and a report providing requirement coverage along with other model statistics can be provided. However, when

run in offline mode, there is currently no option to provide a report on the model requirement coverage statistics. A ticket has been opened for AltWalker to add this option in a future update. In either case, a requirement coverage matrix is generated next, utilizing the states that are covered by the test cases that satisfy the requirements. The figure below illustrates the trace of the generated path elements to the requirements satisfied by those elements.

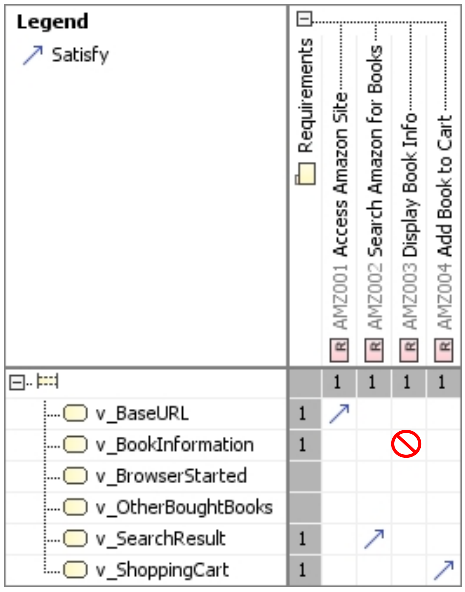


**Figure 3.** Test Case Requirement Coverage Matrix

The rows on the left contain the model elements and the columns across the top contain the requirements the elements satisfy. An arrow at an intersection of the row and column indicates the model element in the row satisfies the requirement in the column. In Cameo, the default for a requirement that is not satisfied is a column of blank spaces; this indicates that none of the states covered by the test case(s) satisfy the requirement. For illustrative purposes, the ⃠ symbol was added to identify the requirement that is not satisfied because the state was not a part of the test case(s). This requirement coverage matrix allows for a quick assessment of which requirements are not covered, which may indicate gaps, inaccuracies, or necessary modifications to the model.

## 5. TEST EXECUTION AND VERIFICATION / VALIDATION

Once sufficient requirements coverage by the test suite has been established, it is then run in the test environment and the test results are collected. After converting these results to a CSV file, Cameo Systems Modeler can import the test results, and represent them as matrix which traces the successfully verified requirements to the test cases that passed, and the unverified requirements to test case failures. The Requirements Report Wizard within Cameo Systems Modeler can then output the requirements and various matrices into a report.

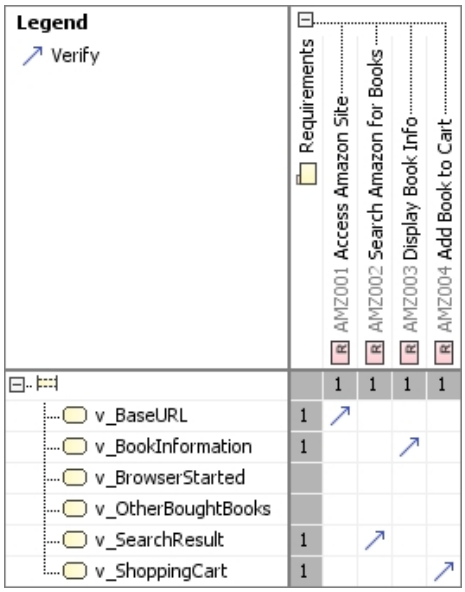


**Figure 4**. Test Suite Run Results

In this manufactured example, Figure 4 illustrates the results of executing a set of concrete test cases, and ultimately indicates which requirements have been met by the system – exposed by executing the automatically-generated test suite. The rows on the left contain the model elements and the columns across the top contain the requirements the elements verify. An arrow at an intersection of the row and column indicates the model element in the row verifies the requirement in the column.

This is a primary advantage of a model-based T&E environment, where all requirements are

captured, the system is described, and the test cases are generated in one cohesive space. There is a single source of truth for each of these elements within the environment, where the results of test execution and the traces back to requirements are all captured and displayed together in an organized, consistent manner.

## 6. FUTURE WORK

### *6.1. Concrete Test Generation*

Concrete test cases are test cases containing details to be tested in an actual test environment. The idea here is to use the abstract test suite generated by the AltWalker and Graphwalker tools as a scaffolding from which to extend and embed or link to the information necessary for the test cases to be executed in a test environment. Each abstract test case including data values and behavior will need to be mapped to concrete data values or actions such as API calls held by the real system under test [4]. A tool or adaptor will need to be developed to assist in the translation from high-level abstract test cases to lower level system under test actions. We want to use the power of object orientation granted by UML and SysML to develop this approach in a flexible manner so that different test frameworks and test environments can be explored without specifying this information in the abstract test cases themselves.

### *6.2. Automated Test Execution*

Two test automation frameworks are currently being investigated both of which are primarily used for software testing: AltWalker and the Robot Framework. AltWalker provides the capability to tie executable C# or Python code to the states or vertices in the state machine graph. The online mode command tells AltWalker to generate a path through the state machine according to the type of path generator chosen and stop conditions specified and execute code corresponding to each state along the way. Robot Framework is an open source automation framework with a rich eco-system of libraries and tools and uses human-readable keyword syntax [5]. Additional Python or Java based libraries can be developed to further extend the framework's utility. Robot Framework has a layered architecture making it promising for use as the framework to use for automated testing of hardware and software systems.

## 7. CONCLUSION

### *7.1. Summary and Impact*

Model-based testing has proven to be extremely useful, particularly through auto generation of test cases. Auto generation of test cases facilitates more complete testing of systems, and increases efficiency of generating, executing, and managing test cases. This minimizes risks of system failures, creates an efficient framework for regression testing and functional testing of new features as the systems evolve, and allows for earlier detection and resolution of discrepancies between system design and requirements. The usage of a framework developed in SysML allows for a federated and repeatable architecture, which ultimately leads to ensuring correct and efficient test-case generation and traceability to system requirements.

### *7.2. Challenges and Advantages of MBSE Approach to T&E*

**Challenges**

Creating the initial model requires abstract thinking and knowledge of the system. The goal is in fact not to model the actual system, but to model the expected behavior of the system-under-test. It is easy for an inexperienced modeler to get lost in the details, which is time consuming and leads to an excessively complex model of the system. It is also a challenge to accurately model the behavior of a system without having any knowledge of the system. As such, collaboration between an experienced modeler and a SME of the system is necessary to avoid a bloated and/or inaccurate model of the system behavior.

This investigation used relatively simple examples in order to demonstrate proof-of-concept of this process, and did not explore the impact on feasibility or efficiency for more-complex systems. This could be done in future work. However, it is reasonable to expect that as programs collectively move toward a more MBSE-centric method for designing and procuring systems, the model content generated for complex systems as part of the general course of applying MBSE will aid in the creation of a state machine which reflects the expected behavior of a system-under-test.

**Advantages**

There are several benefits of creating a model of the expected behavior of the system to generate test cases. First, it allows for more efficient testing by reducing redundant test cases, as well as creating test cases to simultaneously cover several requirements, rather than creating requirements and test cases that are one to one. Also, construction of a SysML model reflective of the system can expose discrepancies between the design and the requirements. Finding these discrepancies earlier facilitates more efficient resolution.

Second, tying requirements to the state machine provides automated traceability. After importing the requirements and building the state machine from them, a requirements matrix can be created as a source of truth of what parts of the state machines verify the requirements. A new matrix produced after generating the test cases provides insight to requirement coverage. Also, after running the test cases and importing the results back into the model, creating a new matrix assists in identifying the verified requirements. Finally, a final report is produced, which enumerates which requirements were tested, which requirements still need testing, which requirements were tested and passed, and which requirements were tested but failed.

Third, creating an initial model of the expected behavior of the system sees its benefit during the continuous enhancement of the system. As systems evolve, there will be modifications to the requirements that drive modifications to the design of the system and system behaviors, which drive modifications to test cases. Making adjustments to the model of the behavior of the system is generally significantly easier and more efficient than manually locating and adjusting test cases, as well as verifying that they do not interfere or conflict with other test cases.

### *7.3. Similar Previous Efforts*

During the research behind developing the approach described in this paper, the research team encountered two similar previous research efforts.

The first effort [6] outlines a process for test generation and execution which entails creating a model, traversing the model, generating abstract test cases, mapping the abstract test cases to concrete test cases, and then executing those test cases on the system under test.

The second effort [7] focuses more on utilizing model-based verification and validation earlier in the system engineering cycle. This approach is to develop and verify a model of the system, and then build and validate the system to realize the model. The portion of the effort for developing the model focuses on SysML and tying various modeling aspects to simulation implementation, which is necessary for the execution of this process, particularly creating a complete and accurate state machine for the system-under-test.

## 8. REFERENCES

[1]Altom Consulting, *Use Actions and Guards*, 2019. Accessed on: Jun. 1 2020. [Online]. Available:

[2]Altom Consulting, *Overview*, 2019. Accessed on: Jun. 1 2020. [Online]. Available: https://altom.gitlab.io/AltWalker/AltWalker/overview.html

[3] Hermann, K. GraphWalker: Model-based testing. 2020. v4.2.0. https://github.com/GraphWalker

[4] Utting, M., & Legeard, B. (2007). *Practical model-based testing : a tools approach*(pp. 1 online resource (xix, 433 pages)). Retrieved from https://ebookcentral.proquest.com/lib/mit/detail.action?docID=282069 MIT Access Only

[5] Robot Framework Foundation, *Introduction*, 2020. Accessed on: Jun. 2 2020. [Online]. Available: https://robotframework.org

[6]D. Ganesan, M. Lindvall, C. Song and C. Schulze, *Model-based testing of NASA systems*, [Online]. Available: https://www.nasa.gov/sites/default/files/03-04_model-based_testing_of_nasa_systems.pdf.

[7]M. O. Khan, G. F. Dubos, J. Tirona and S. Standley, *Model-based verification and validation of the SMAP uplink processes*, *2013 IEEE Aerospace Conference*, Big Sky, MT, 2013, pp. 1-9. https://ieeexplore.ieee.org/document/6496913