2023 NDIA MICHIGAN CHAPTER
GROUND VEHICLE SYSTEMS ENGINEERING
AND TECHNOLOGY SYMPOSIUM
DIGITAL ENGINEERING SYSTEMS ENGINEERING (DE/SE) TECHNICAL SESSION
AUGUST 15-17, 2023 - NOVI, MICHIGAN

AUTOMATION OF TEST CASE GENERATION AND SOFTWARE SYSTEM MODELLING

Michael Lingg, PhD[1], **Howard Paul**[1], **Sachin Kushwaha**[1], **Jaiden Ortiz**[2]

[1]Array of Engineers, Grand Rapids, MI, [2] Grand Valley State University, Allendale, MI

ABSTRACT

System and software requirements provide a definition of what the system implementation is required to do, and are a necessary component to independent requirement based testing for safety critical systems. However as vital as these requirements are, the requirements often are not analyzed until a safety assessment is performed, or the system fails during testing. Automating the system analysis and testing can be used to help to shift left the software life cycle, particularly when the automation augments, rather than replaces, human test developers. This paper presents a method to convert textual requirements into a logical model of the system. This logical model can be used for various automated system analysis procedures, as well as automated test generation. We show this automation can provide significant insight into possible issues in the system, as well as significantly accelerating the time required for test development.

Citation: M. Lingg, H. Paul, S. Kushwaha, J Ortiz, "Automation of Test Case Generation and Software System Modelling," In *Proceedings of the Ground Vehicle Systems Engineering and Technology Symposium* (GVSETS), NDIA, Novi, MI, Aug. 15-17, 2023.

1. INTRODUCTION

The use of software is ever expanding, and continuously evolving. In the past, changing the behavior of equipment and vehicles would require physical servicing to replace physical components and hardware. The use of software to control behavior means mission parameters can be completely redefined in the field in seconds, making software invaluable in an ever changing environment.

However the benefits of software are not without cost. Poor software quality has been estimated to cost the US around $2.84 trillion dollars [1] in 2018 alone, with losses due to software failures exceeding a third of this cost. Costs due to bugs can climb upwards

of billions of dollars as evidenced in the Soviet Gas Pipeline Explosion in 1982, reach to the hundreds of billions in cost to repair the Y2K bug [2], or impact the safety and lives of numerous people, as in the Boeing MCAS software failure that lead to the loss of two Boeing 737 MAX aircraft, and 346 lives [3].

Conversely, as technology advances, the cost, and complexity, of software systems continues to grow [4]. These costs are made worse with safety critical systems, where the system must be proven to perform the expected, and only the expected, functionality. This has led to concerns related to not just the increase of cost, but questions of are costs sustainable at the current rate of growth in safety critical systems [5]. The growth of costs is further enhanced by the level of detail required for higher levels of system criticality [6].

For decades the solution for safety critical systems was the waterfall method [7]. This method primarily exists to satisfy independent Requirements Based Tests (RBT) [8], where tests are developed purely from the requirements, independent of software, to try to ensure the software implementation performs all desired operations as specified by the requirements, and only these operations. Agile methods are one approach intended to attempt to shift left [9] the software development, or perform testing earlier in the software life cycle, when changes cost less. This approach leads to other difficulties, however [10]. When the system as a whole is not defined prior to implementation, unexpected interactions can occur between components of the system.

While the use of agile has produced successful results in shifting the software life cycle to the left, projects with longer life cycles may not be able to shift left as effectively [11]. Aviation software projects have a very long life cycle, with significant updates being performed decades into the life of the project. These latter updates have to be carefully identified and integrated into a very large existing system, at very high cost.

In addition to the cost of failing to discover software bugs, or discovering software bugs later in the software life cycle, the cost of sufficient testing to identify software bugs can be very high. Development of a single test case by hand can take one or more hours for simple behavior, and can easily be in the tens of hours for more complex behavior. In systems with thousands of requirements, manual test development often reaches into engineer years. Any use of automation can provide significant cost benefits, along with the consistency provided by automating the process.

Any method which can be used to analyze or test the system earlier in order to shift left the software life cycle can provide a significant reduction of the costs describe above, and thus is of great interest to the field of software development. In this paper, automated methods to model a system, based on requirement text, are explored. The paper then looks into how this model can be used to analyze and test the system as it is being developed. Additionally, the paper looks at how automation of test generation can accelerate the testing of a safety critical system.

2. BACKGROUND

System analysis is an often overlooked part of the software development life cycle. Many projects, at worse, either simply implement the system design and move onto testing, or at best, perform a safety assessment on the system after the system is completed. Gaps in the system, dead code or masked signals, may not be identified until the testing of the software. In addition, software testing or a safety assessment can often overlook gaps in the system as requirements may not be fully exercised in isolation, but conflict with each other, so unit testing will miss bugs that integration testing is needed to identify.

Take for example the following requirements:

```
Requirement 1:
FMU_software shall set OVERSPEED_FAULT
to True if the following logic
evaluates to True:
```

```
(
  (BRAKE_SENSOR_STATUS is equal to
   BRAKES_APPLIED)
  AND
  (BRAKE_SENSOR_STATUS_VALID is equal
   to True)
  AND
  (
    (RECORDED_SPEED is greater than
     IN_FLIGHT_TH) AND
    (RECORDED_SPEED_VALID is True)\n
  ) is equal to True unchanged for
    greater than OVERSPEED_FAULT_DELAY
  AND
  (FAULT_DETECT_ENABLED is equal to
   True)
  AND
  (ON_GROUND is equal to True)
  AND
  (ON_GROUND_VALID is equal to True)
)

Otherwise, set OVERSPEED_FAULT to
False.

Requirement 2:
FMU_software shall set the
FAULT_DETECT_ENABLED to True if the
following logic evaluates to True:
(
  (
    (PBIT_STATUS is equal to True)
    AND
    (PBIT_STATUS_VALID is equal to
     True)
  )
  AND
  (
    (RECORDED_SPEED is greater than
     IN_FLIGHT_TH)
    AND
    (RECORDED_SPEED_VALID is equal
     to True)
  ) is True for greater than
    STABLE_FLIGHT_DELAY
  AND
  (
    (ON_GROUND is equal to False)
    AND
    (ON_GROUND_VALID is equal to
     True)
  )
)

Otherwise, set FAULT_DETECT_ENABLED
to False.
```

Taken separately these requirements are perfectly valid and unit tests will pass just fine. Providing 100% unit test coverage of the code. However full integration tests will identify that Requirement 1 needs ON_GROUND set to True, while Requirement 2, which is an input to Requirement 1, needs ON_GROUND set to False. The methods section 3.4, will show how to identify these types of logic errors in requirements earlier in the software life cycle, though automated analysis of the logic defined by the requirements.

Next, consider the impact of changes to the system. In most legacy projects the change impact analysis will look at requirement redlines, like the following example.

```
FMU_software shall set the
FUEL_LEVEL_VALID to True and set
FUEL_LEVEL to the value of
IN_FUEL_LEVEL if the following logic
evaluates to True:

(
  ( IN_FUEL_LEVEL_ACTIVE is equal to
    True)
   AND
   (
    ( IN_FUEL_LEVEL is greater than
      or equal to
      IN_FUEL_LEVEL_ICD_MIN)
    OR
    ( IN_FUEL_LEVEL is less than or
```

```
      equal to IN_FUEL_LEVEL_ICD_MAX)
  )
)

Otherwise, set the FUEL_LEVEL_VALID to
False and set FUEL_LEVEL to
IN_FUEL_LEVEL_ICD_MIN.
```

The provided change is relatively simple, a single condition is added. This condition is must be added to the implementation, and be added to the existing tests. Additionally the change will result in a new logic path from the inputs to define the behavior of IN_FUEL_LEVEL_ACTIVE, and all requirements this requirement traces to will be impacted. The cost of this change can be estimated based on the average cost to update a requirement, but this is a fairly coarse estimate. Since this is a fairly simple example, the impacts are rather easy to estimate. In an active, mature project, the changes may involve hundreds of modified requirements, with interrelated impacts to the logic. The cost to estimate, and update the system continues to grow exponentially with the number of requirements changed. The end of the methods section 3.2 will look at how automation can significantly save on costs to identify and estimate impact, and an accurate model of the system can more precisely model the impact of changes.

Lastly, consider the impact of requirements on testing. Looking at the previous redline requirement as a whole new requirement, there are is a logical AND as well as an OR, which according to the rule of tests cases being equal to the number of MCDC logic gates + 1, there will be three test cases. A test developer will need to create a test procedure outline for each of the three tests cases, define input conditions and expected output conditions. While this is a relatively trivial example, some tests may have many more cases to implement, each case with a very similar structure to create, as well as the software engineer having to determine the MCDC conditions. If we consider only the redline change, a single test case will have to be added. Engineers typically copy/paste a previous test case to save on time, which can be error prone, particularly when done with dozens, or hundreds, of changed requirements. In the methods section 3.2 we will look at how automation can eliminate the tedious, repetitive, creation of test sections, and save time on determining what the MCDC cases are.

3. METHODS

3.1. Building the Model of a Single Requirement

This algorithm builds a logical model of the system by identifying the inputs, conditions, and outputs of blocks of logic. Inputs are signals, or variables, with defined types and ranges, or constants. Conditions are broken down into mathematical operators, comparisons, logical operators and timers. Outputs are signals, that are set to input signals, or constants, and may be set using a combination of mathematical operations.

To help visualize how a logic graph is constructed, an example is provided of a generic customer requirement, with a simplified data dictionary. The data dictionary exists to define the types, and ranges, of each variable, allowing the algorithm to determine a key set of values for system analysis. For example, Boolean variables are fully exercised just by being set to both True and False. Alternately, numerical values may have a nominal, minimum and maximum value that define the boundary of the equivalence classes, values necessary to properly exercise comparisons, and may be tested for data type minimum and maximum for robustness.

```
Requirement:
(condition1) The software shall set
   OUTPUT to True and set OUTPUT_VALID
   to True if the following is True:
(
   (INPUT is greater than COMPARISON)
   AND
   (INPUT_VALID is equal to True)
```

```
)

(condition2) otherwise, set OUTPUT
   to False and OUTPUT_VALID to True
   if the following logic evaluates
   to True:
(
   (INPUT is less than or equal to
    COMPARISON) AND
   (INPUT_VALID is equal to True)
)

(condition3) otherwise, set maintain
   the current value of OUTPUT and
   set OUTPUT_VALID to False.

Data Dictionary:
INPUT {"Type": "Integer",
       "Range": "0..10"}
INPUT_VALID {"Type": "Boolean"}
OUTPUT {"Type": "Boolean"}
OUTPUT_VALID {"Type": "Boolean"}
COMPARISON {"Type": "Constant",
            "Value": "5"}
Boolean {"Range": "True, False"}
```

The red text in the requirement are labels added to identify each output condition of the requirement, and will be referenced below.

First, each condition is identified and isolated. In the requirement above, there are three conditions, each with corresponding outputs. Two of the three conditions have input logic associated with the conditions, while the third condition is always true if reached. These conditions can be visually represented as shown in figure 1, where condition1 is is the priority condition, and if a condition evaluates to false the logic will fall through to the next condition, until the always true condition3 is reached.

Figure 1: Visualization of requirement conditions.

Next the logical conditions for each condition are added to the graph, as shown in figure 2. Note the otherwise condition has an input condition of true, this indicates the case will always be true if the other conditions all fall through.

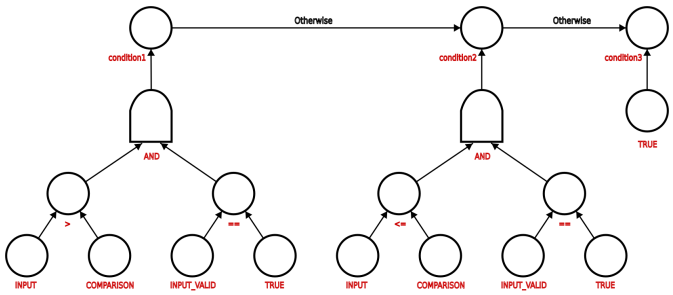

Figure 2: Visualization of requirement logical conditions.

Last the output values are added to the visualization. All three conditions output the same signals, but set different values. So the connection indicates both the signal being output, and the value being set, as shown in figure 3.

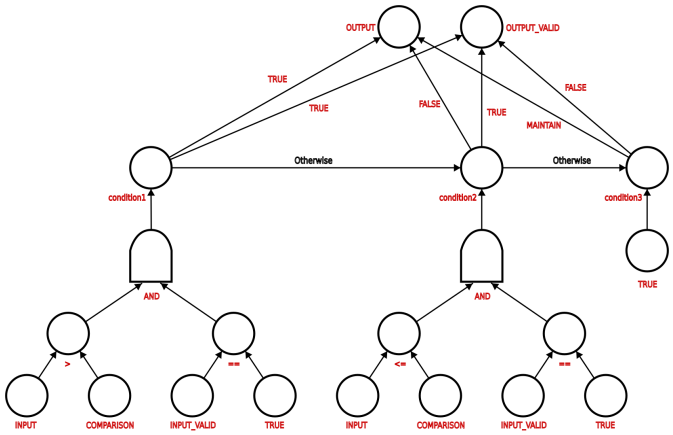

Figure 3: Visualization of full requirement.

This method of converting requirement text to a model is done via two methods. The first method, used when the requirements are very structured, is using a set of regular expressions can be used to parse the requirement text and produce the model logic graph. This method proved extremely viable, albeit restrictive, while developing the prototype of this tool. The second method, being developed with the assistance of Dr Rafiq [12] at Grand Valley State University's (GVSU) Applied Computing Institute (ACI), is the use of knowledge graphs and machine learning to provided a layered approach to read vague and generically written textual requirements, and produce very structured requirements. These structured requirements can be read by both the regular expressions and NLP to provide a higher confidence result. This approach is outlined in figure 4, where the text of a requirement is converted into a knowledge graph with relationships identified, then the relationships are parsed by an NLP engine to produce a postfix equation, and ultimately a logical graph of the system. Throughout the process, a human in the loop is used to check that the automation is correctly parsing the requirement. This allows the tool to be used in a DO-178B/C environment without having to qualify the tool, as there is independent human oversight.

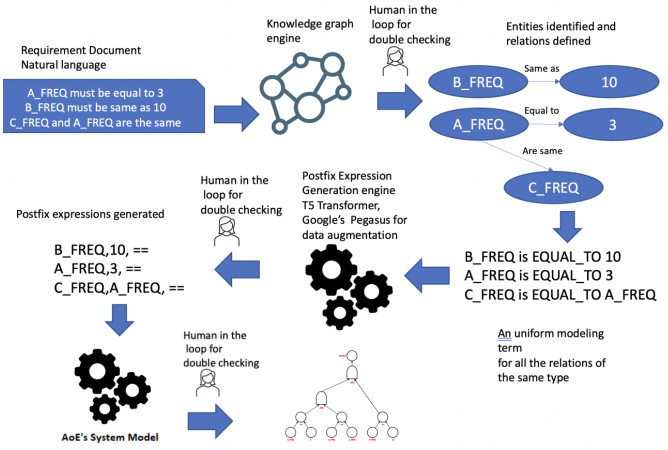

Figure 4: Machine Learning Example Pipeline.

3.2. Analysis of a Single Requirement

System analysis provided by this algorithm for a single requirement falls into three categories. First is identifying MCDC coverage for test automation. Second is determining what outputs will be produced for a given set of inputs. Third is determining what sets of inputs can produce a desired output.

For MCDC coverage, each node type in the logic graph has different computations. Constant nodes simply provide a constant value, and signal nodes provide the datatype and data dictionary min/max values initially. Mathematical nodes look at all combination of child node values to identify what outputs can be produced. Comparator nodes look at what values are available from their children to identify cases that can set the comparison result to true and false. Logical nodes identify what child values are available to produce the MCDC condition for the current logical operator. If the children do not have values needed to set a desired condition, the parent will ask the child to output those values, and this process is repeated to the requirement inputs to determine all values necessary to set the desired values.

To follow this process through the example tree, the focus starts on one of the comparators, shown in figure 5. The comparator outputs true if the signal INPUT is greater than the constant COMPARISON. INPUT has a data dictionary minimum of 0, and maximum of 10, while COMPARISON has a value of 5. All combinations of INPUT and COMPARISON are compared with the greater than operator, with the results shown in figure 5, of INPUT = 0 greater than COMPARISON = 5 is False, and INPUT = 10 greater than COMPARISON = 5 is False.

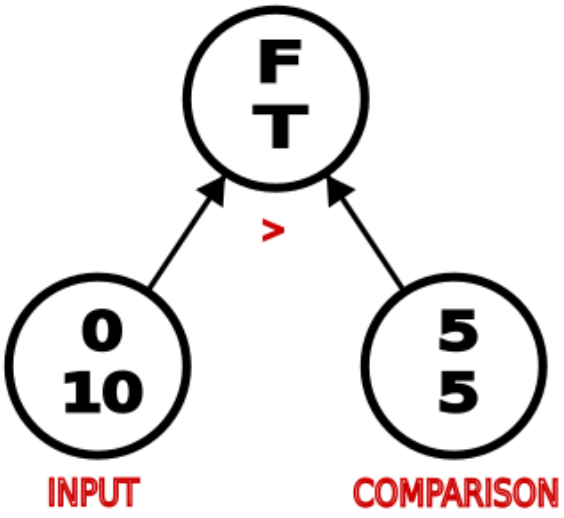

Figure 5: Visualization of MCDC Comparison.

Both the true and false result of the comparison are produced, so the output of comparison is fully exercised, satisfying MCDC, and no other values, such as a nominal value, are necessary. If this did not produce all necessary outputs, say if the comparison operator was is equal to, the following process would be used. The is equal to operator would find neither 0 or 10 is equal to 5, so all outputs are false. Since the constant cannot change, the INPUT signal would have to be set to a value that makes the is equal operator true. This would be a value of 5. So the equal to operator would tell the INPUT child node to add a value of 5 to its list of values, since this is within the INPUT data dictionary range, the value would be successfully added.

Next is a quick dive into how MCDC is determined for a logical operator. In the example we only have AND operators, which is satisfied by the input cases True/True, True/False and False/True, as shown in 6. From looking at the greater than comparison above, we know the first case was a false result, and the second was a true result. So to satisfy the logical AND operator, we would use the greater than operator's second case twice, and first case once. Analyzing the children of the is equal operator would reveal what cases of the is equal operator would satisfy the inputs needed by the AND. For more details on producing and automating MCDC tests, see our white paper on automation of MCDC testing [13].

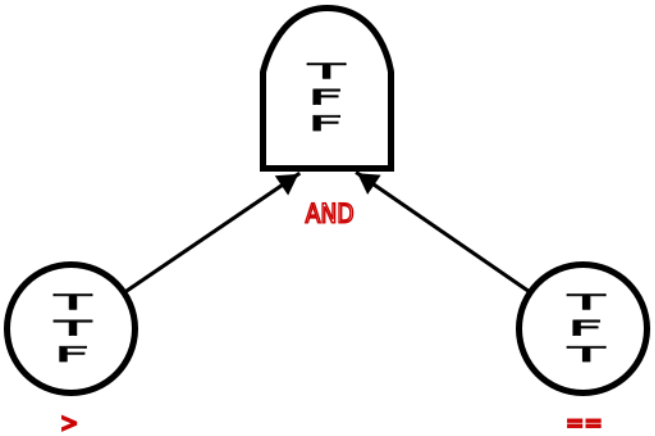

Figure 6: Visualization of MCDC Logical Operator.

A full run of this requirement through this tool produces a JSON test procedure, that satisfies all MCDC cases, and can be converted into a ReqIF, or other format for importing test procedures to a formal requirement tool like DOORS or Cameo.

Next we will look at system analysis, which is performed on the model of the system produced from parsing the requirement text. The basics are identifying what outputs are set from known inputs and what inputs are set from known outputs. This is not that dissimilar to analysis provided by similar tools, such as SCADE/Simulink. The difference is this tool tool starts from text based requirements, rather than requiring requirement developers to start with a model of the system. We can also provide an analysis of how a signal, or set of signals, would impact outputs if allowed to be set to any value, while other signals are locked to specified values. This tool then provides advanced manual and automated system analysis to locate potential conflicts within the requirements. First we will look at the basics of how outputs are identified from input values, and what inputs are necessary to set specified outputs.

Identifying what outputs are set from a set of locked down inputs starts with looking at the inputs being produced by the requirement. If a user wants to see what outputs are set when INPUT is set to 6, and INPUT_VALID is set to true, the system will find which requirements input these signals, and would find a match in the example requirement. For analysis of the outputs of this requirement, the logic graph will be followed from the outputs down to the inputs. If the input is one of the locked down inputs, the desired value will be passed to the parent node to be evaluated. Evaluations continue up the logic graph until outputs are identified, as shown in figure 7. For the example requirement and the inputs specified above, the value of INPUT will be greater than COMPARISON, and INPUT_VALID is true, so the AND of condition1 will evaluate to true. condition1 sets OUTPUT to true and OUTPUT_VALID to true, so the outputs have been identified from the specified inputs. The algorithm could analyze the other conditions, but once a condition evaluates to true, the otherwise cases do not need to be evaluated.

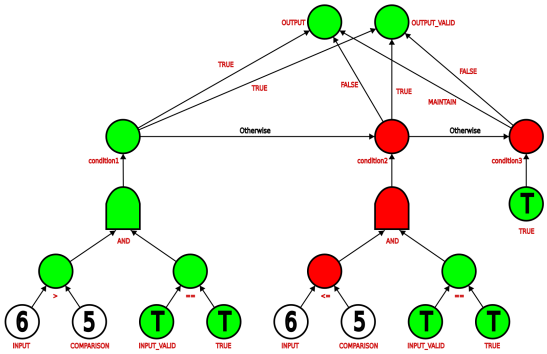

Figure 7: Visualization of Output Analysis.

Identifying what inputs are needed to set known outputs starts with looking at the outputs being produced by the requirement. If a user wants to see what inputs can set OUTPUT to True and OUTPUT_VALID to true, the system will first find which requirement produces these signals, and would find a match in the example requirement. Next the system checks the conditions in order, to see if any condition sets the outputs to the desired values. In this case, condition1 sets the outputs as desired. Then the logic is followed down from condition1. For the AND to be set true, both the greater than and equal to comparisons must be true. The equal to comparison can only be set true if INPUT_VALID is true. Following the logic graph down from the greater than finds any value of INPUT greater than COMPARISON (5) will set that comparator true. So the system analysis will report OUTPUT=true and OUTPUT_VALID=true is satisfied by any value of INPUT greater than 5, and INPUT_VALID set to true, as shown in figure 8.

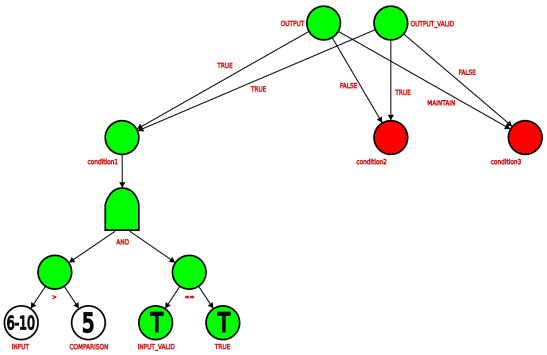

Figure 8: Visualization of Input Analysis.

The system analysis goes one step further in that if not all signals are locked down, any signal not locked down is evaluated at nominal, data dictionary min and max, and datatype min and max. An example is if we want to know the outputs if only INPUT is locked down to 6. Because INPUT_VALID can be either true or false, both condition1 and condition3 can evaluate to true in this case. So the outputs are either OUTPUT=true and OUTPUT_VALID=true as shown in figure 7, or OUTPUT=maintain and OUTPUT_VALID=false as shown in figure 9.

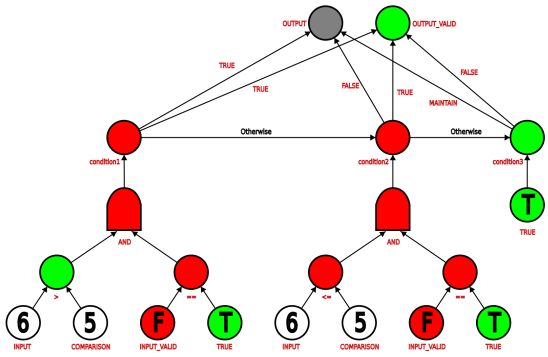

Figure 9: Visualization of Output Analysis, Validity False.

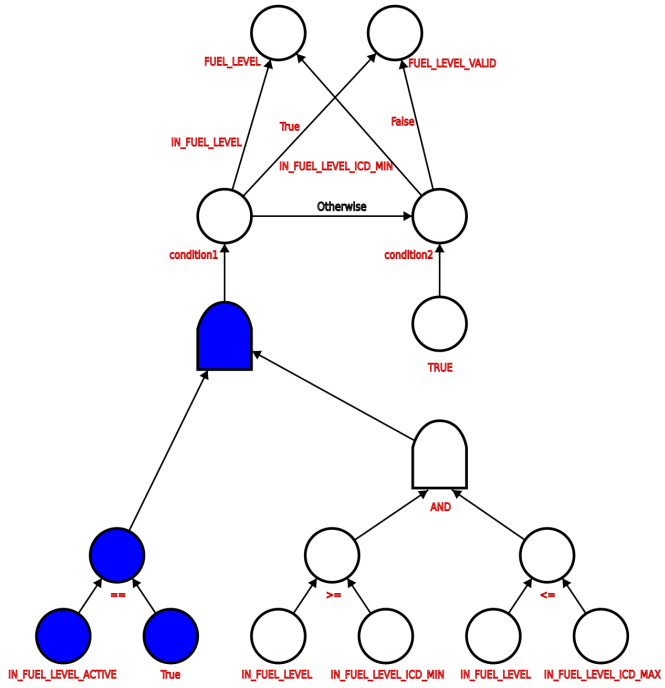

Figure 10: Visualization of requirement changes.

In addition to the basic logic analysis, a time based analysis can be performed. Identifying how signals propagate through the system can include time delays, as well as loop backs. As well as changing inputs over time and seeing how the set of outputs changes over time.

A final capability is analyzing requirement changes. Figure 10 shows a visualization of a requirement difference report, where blue shows new logic, and red would show removed logic, if there was any. Because the differences are determined on which logical conditions are different, estimations of complexity changes can be made much more fine grained. This is something we have only started exploring, so this is as much as we have to report in this paper.

3.3. Building a Model of Multiple Requirements

The analysis of a single requirement is simply a building block in the system. When one or more requirements are read in, the requirements are linked together by their signals. The data dictionary is used to differentiate between signals and constants within the system, while an Interface Control Document (ICD) is used to identify system inputs and outputs, and provide the details of these system signals. Figure 11 shows an example of multiple logic blocks, with their internal logic collapsed down to show just the requirement relationships. In this case, there are two system inputs passed in, these are converted to internal signals with validities, and a system output is produced from the internal signals. When each requirement's logic is brought into a logic graph, its inputs and outputs are mapped to the requirement. Then when the user wants to set certain signals, or find what inputs set certain signals, which requirements input or produce these signals can be

quickly looked up.

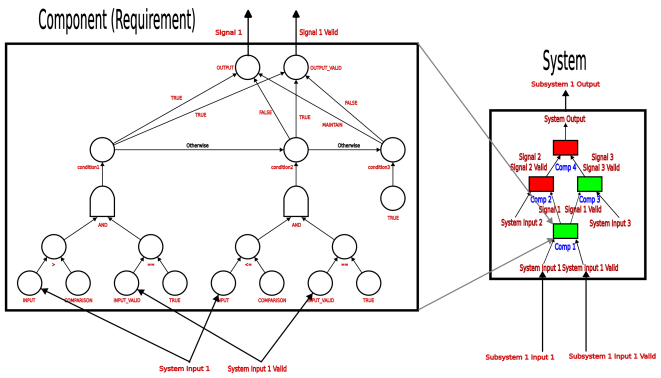

Figure 11: Visualization of Multiple Logic Block Relationships.

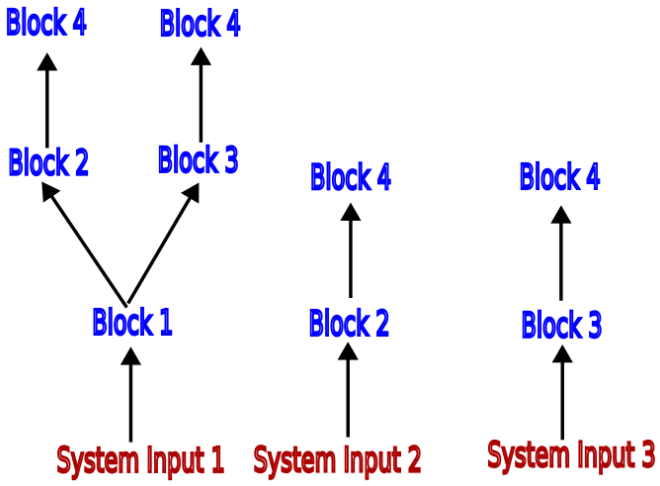

Figure 12: Visualization of Signal Path Trees.

3.4. System Analysis with Multiple Requirements

The core element of analyzing dataflow across multiple requirements is identifying the paths from a given signal to all inputs that can affect the signal, or all outputs affected by the signal. This analysis is performed by starting with a signal and a direction of toward the inputs or outputs. The signal mapping is used to find all requirements that input or output the desired signal. Then, depending on the direction, all input or output signals of the requirement are identified and added to a path tree, along with the identity of the requirement. This process is repeated for all signals found along the way, until system inputs or outputs are reached. This provides a complete analysis of the requirement path from the signal to system inputs or outputs, and all internal signals impacted along the way. Traversing the logic shown in figure 11 for each system input, will produce the traversals shown in 12. This is used for further system analysis described below, but can also be used to quickly provide an analysis of the impact of changing a given signal.

Identifying outputs set by locking down input signals, or what inputs are needed to set certain output values is extended to the multiple requirement scope. Using the signal path trees defined in figure 12, the trees are searched in a breath first traversal. Any requirements that show up more than once will only be in the analysis path once, at the last point found in the breadth first search. This ensures that each requirement is not processed until its dependencies have been processed. For example, parsing the path trees in figure 12 will produce an analysis list of Block 1, followed by Block 2 or Block 3 in any order, and finally Block 4, which is dependent on Blocks 2 and 3 last. Then this list of requirements is processed, starting with the defined input or outputs, and the inputs and outputs of each requirement along the way is combined into sets of valid signals along the analysis path. For the example, if we analyze the output with all system inputs set to known values, System Input 1 and System Input 1 Valid would be set to a known value, and Signal 1 and Signal 1 Valid would be identified. All of these would be passed to Comp 2, with System Input 2. This would be repeated for Comp 3 and 4, to find System Output.

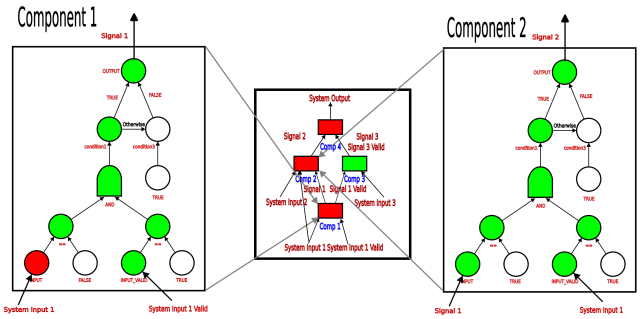

Figure 13: Visualization of an Output That Cannot be Set.

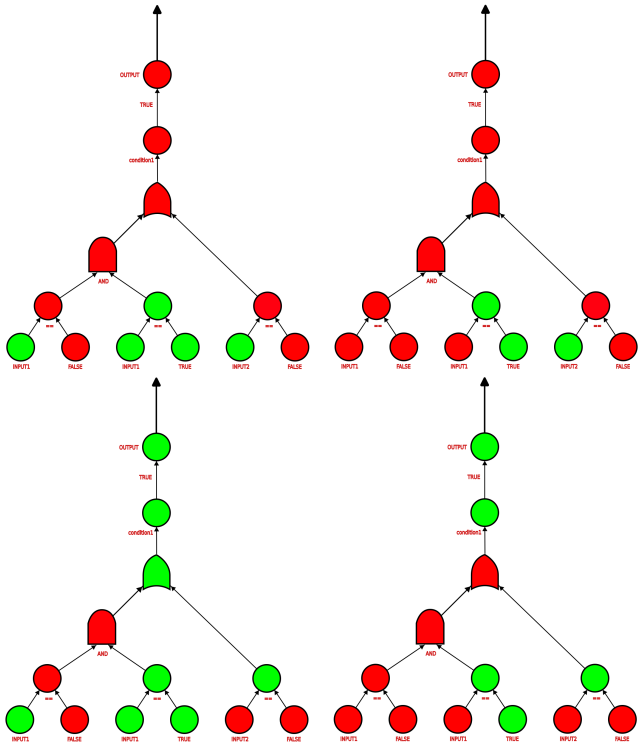

Figure 14: Visualization of an Input That is Masked Out.

Figure 13 uses a simplified version of the conflicting requirements in the background, to show what conflict analysis would look like. The conflict would be detected when looking at what sets Signal 2, from Component 2, to True and False. Setting Signal 2 to False works fine as there are a number of possible paths, but setting it True is different. This analysis would identify all inputs of Component 2 that set Signal 2 True, just like described in the previous section. Then these values would be used to set the output of Component 1, and identify all necessary inputs. The result is System Input 1 cannot be set to both True (required by Component 2) and False (required by Component 1) at the same time, no set of inputs will be identified that can set Signal 2 to True.

Figure 14 shows a simple example of a masked out input. Input 1 and 2 are tested with all possible values. The results show that due to input 1 being masked by the AND gate causing it to have to be set true and false at the same time. We can see this because the output is toggled by input 2, but never by input 1.

3.5. *Application to System or System of Systems*

A future capability is to be able to perform analysis on a system of systems. All of the above algorithms analyze the requirements for a single software application. However this same method can extend to how multiple components of a system, or multiple systems, interact together. In the same way inputs and outputs of requirements are linked together, inputs and outputs of components of a system, or systems of systems, can be connected together to perform dataflow analysis on larger systems.

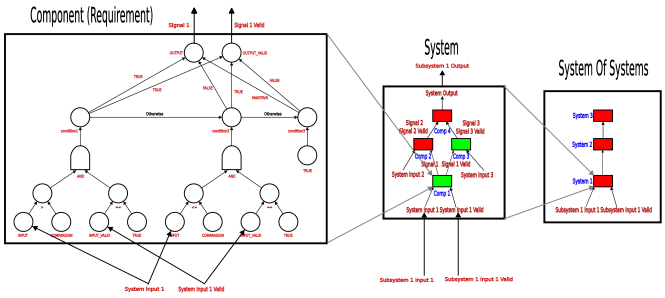

Figure 15: Visualization of a System of Systems.

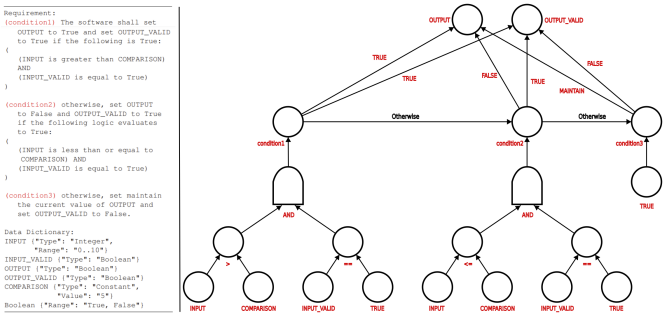

Figure 16: Example requirement text and model side-by-side.

3.6. Interoperability with other tools.

Interoperability is an important feature of this tool, not only having the ability to bring requirements into the system analysis tool, but the ability to integrate with other tools, and extend the digital engineering environment. Built into the tool is the capability to import and export different formats. JSON and XML are both supported, as well as interfaces to import and export data with the IBM DOORS Requirement Management Tool [14]. Beyond this, is a number of additional capabilities that are planned for future development. ReqIF and OLSC integration is planned to be added for integration with tools, such as JAMA and Cameo. To better integrate with existing text based requirement environments, a future expansion of this tool is to use Machine Learning to not only parse the requirements, but also use what it learned while parsing the requirement to maintain the requirement text when the model changes, in the same format the requirement was originally written in. Figure 16 shows a possible screen example of what having requirement text and model side by side might look like, so each is in synch with changes to the other

4. RESULTS

During manual development of tests we found that developing one fairly basic test procedure required 120 minutes to develop the test procedure, and 30 minutes to debug initial failures in the procedure, for a total time of 150 minutes to complete the test. The automated test case generation tool required 30 minutes to generate the test procedure, including the test developer's time to complete the test, and only 5 minutes to debug the test, for a total time of 35 minutes to complete the test. This is a little over 4x improvement in overall time to complete the test procedure. While the tool generates the test, the test developer is free to do other tasks, reducing the actual engineer-hours spent per test even further.

Though we do not have a comprehensive analysis of time spent on manually vs auto-generated test procedures, the improvements observed in this simple example would indicate that similar, or even greater, improvements could be made when testing more complex requirements. As requirements get more complicated, most time during manual test development is spent identifying the system inputs necessary to set the test inputs. The system modelling provided can generate sets of system inputs in a fraction the time it takes a test developer to identify all of the possible paths. Further, simply identifying the requirement paths back to system inputs can help test developers reduce the time necessary to develop tests, and help in identifying

gaps that need to be tested in the requirements. Even if increasingly complex tests do not show a linear reduction in time to develop the test, a reduction of nearly 2 hours to develop each test would represent a reduction in a man year for each 1000 requirements. This is very significant in systems with thousands of requirements.

Beyond automated test generation is analyzing and identifying conflicts within the system model. AoE developed a set of requirements [15] that are not directly tied to any project to use for testing this system. Figure 17 shows a visualization of this example system.

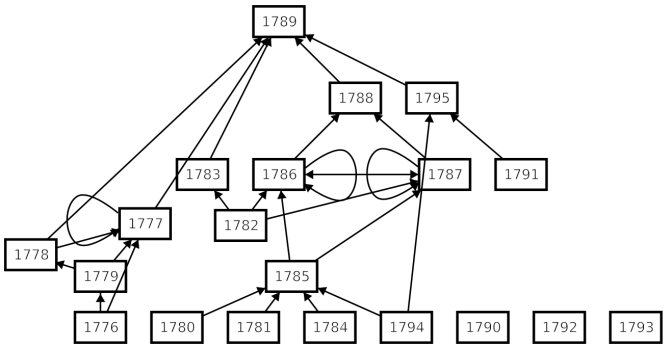

Figure 17: Visualization of Our Example System.

In this section, we will look at the results of the following types of analysis:

- Tracing a signal to all outputs.
- Tracing a signal to all inputs.
- Finding output values from setting inputs.
- Finding input values from setting outputs.
- Find outputs that cannot be set.
- Find inputs that cannot set any outputs.

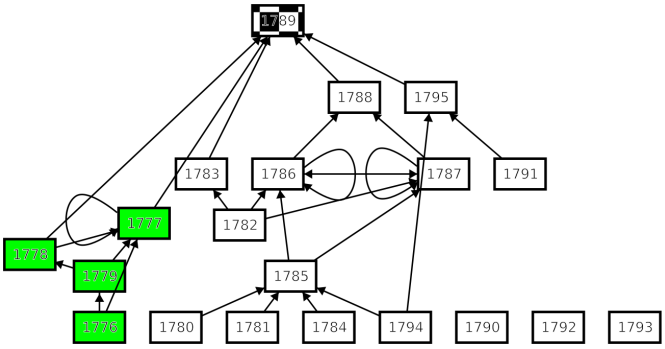

Figure 18: Visualization of Tracing a Signal to All Outputs.

The first analysis is tracing from a given signal to all outputs, or inputs, this signal impacts. Both analysis methods are fairly straightforward as described in the Methods 3.4 section. Figure 18 shows a signal (IN_FUEL_TANK_TEMP, which is an input to FMUREQ-1776) traced to all its outputs and 19 shows a signal (FUEL_LEVEL_LEAK, which is an input to FMUREQ-1788) traced to all its inputs. Each diagram shows a checkerboard pattern for the 'destination' requirements. In addition, information is provided as to what constants and system in/outputs are impacted by this signal, and all internal signal names that connect between requirements.

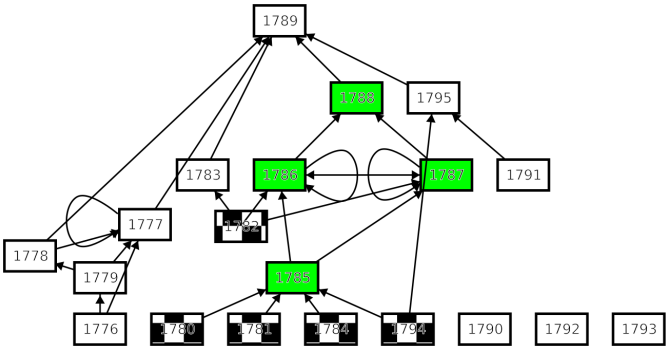

Figure 19: Test run of tracing a signal to all inputs.

Next we look at a test of the system to identify outputs set from known inputs. We set the output we want to identify as FAULT_DETECT_ENABLED, and the inputs as :

- IN_RECORDED_SPEED = 150

- IN_RECORDED_SPEED_ACTIVE = True
- IN_PBIT_STATUS = True
- IN_PBIT_STATUS_ACTIVE = True
- IN_ON_GROUND_ACTIVE = True
- IN_ON_GROUND = False
- IN_ON_GROUND_ACTIVE = True

Ultimately these inputs directly set the inputs of FMUREQ-1794, as shown in figure 20, leading to setting FAULT_DETECT_ENABLED to True.

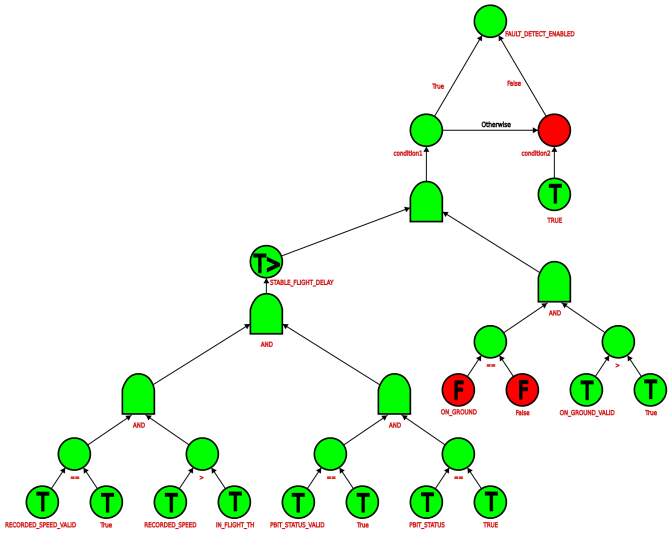

Figure 20: Test run of finding all outputs set by inputs.

We further double check this by more runs with IN_ON_GROUND = True on one run, IN_RECORDED_SPEED = IN_FILGHT_TH on another, and IN_PBIT_STATUS = False on a final test. In each case, FAULT_DETECT_ENABLE is set False, showing proper evaluation.

The process of identifying outputs is then reversed to identify what input combinations can set a known output. The same requirement will be tested due to its familiarity. Identifying what inputs set FAULT_DETECT_ENABLE to True is fairly simple as it is basically only the case shown in figure 20, with a few variations allowed for testing numeric boundary conditions of RECORDED_SPEED and the corresponding signal of IN_RECORDED_SPEED. Identifying all possible cases that set FAULT_DETECT_ENABLE to False is much more extensive as any combination of inputs different from that in figure 20, where RECORDED_SPEED is <= IN_FLIGHT_TH will set the output to False. All possible cases of this result number in the thousands. This large number of combinations is still identified in less than a minute, while humans just trying to write down all combinations will take longer.

Identifying outputs that cannot be set is performed by looping through all boundary values of an output signal, and verifying at least one case can successfully set the output. A simple example was provided in the methods 3.4 section, and for our results we look at a the complex real world case. The signal OVERSPEED_FAULT will be our real world example signal. This signal is set in FMUREQ-1795, via a number of ANDed signals. Both manual analysis, and tool automation, show a number of inputs that can set OVERSPEED_FAULT to false, but things get more interesting when we look at setting OVERSPEED_FAULT to true. All gates of FMUREQ-1795 must be set true, resulting in one possible path to set OVERSPEED_FAULT true, which includes FAULT_DETECT_ENABLED and ON_GROUND both set to true. FMUREQ-1785 must set FAULT_DETECT_ENABLED to true to satisfy our conditions up to this point. However, to set this condition, ON_GROUND must be set to true, leading to a conflict that prevents OVERSPEED_FAULT from ever being true. The analyzer will detect this by finding no set of input combinations can ever set OVERSPEED_FAULT to anything but false.

Going the other way and identifying inputs that do not set an output are done by identifying outputs which a given input is connected to, then setting those outputs to their equivalence class values, and finally searching back to find what input sets can set the outputs. If the inputs in question have

no impact on the outputs, they are masked out. Take 20 for an example. If this requirement is updated to require IN_PBIT_STATUS always be False, this will conflict with PBIT_STATUS, masking it out. Running the analyzer on this condition will show that neither IN_PBIT_STATUS set to True or False will change FAULT_DETECT_ENABLED, so IN_PBIT_STATUS is masked out.

Ultimately the goal of this algorithm is to save engineering time. Three experienced engineers performed the above analyses and timed their results. The total manual analysis time averaged between the three engineers was 45 minutes, while the automated analysis took 1 minute to compute all possible input values, the remaining computations were effectively instant.

5. CONCLUSION

In this paper we have presented an algorithm for parsing structured human written requirements into a system model using logic graph. The system model is then analyzed by a number of methods to determine if there are any gaps in the requirements.

We have previously shown that time to develop a simple test can be reduced by 75%, or two hours. Further we have discussed how this tool can reduce test developer time by automatically generating a test procedure skeleton that would need to be created manually, as well as identifying paths to system inputs. This framework for parsing requirements also works well as a template for writing non-ambiguous, testable, requirements that can be parsed with no possibility of error.

The system analysis identifies gaps in the requirements that is very time consuming for humans to identify. In some cases, the humans may not be able to identify the issues. The performance of this analysis was shown to be significantly faster than manual analysis, while providing a very high level of detail in the results.

We look to expand this tool in the future with better machine learning capabilities to be able to parse natural human language, and identify inconsistencies or gaps in the requirement text. Further system analysis methods are planned, including prompts to a user to define requirements for equivalence classes, or signals, not fully defined in the model.

6. ACKNOWLEDGEMENTS

The authors would like to acknowledge the work of Hattie Pimentel and Dominic Smith in developing the original prototype of this system, William VanSolkema and Jessica Ricksgers for testing the system, and the entire ATCG Ingenuity Friday team for review and suggestions for this paper.

7 References

[1] H. Krasner, "The cost of poor quality software in the us: A 2018 report," *Consortium for IT Software Quality, Tech. Rep*, vol. 10, 2018.

[2] "11 of the most costly software errors in history," Raygun Blog, March 2022 [Online]. [Online]. Available: https://raygun.com/blog/costly-software-errors-history/

[3] P. Johnston and R. Harris, "The boeing 737 max saga: lessons for software organizations," *Software Quality Professional*, vol. 21, no. 3, pp. 4–12, 2019.

[4] "Science direct software complexity topics," sciencedirect.com, Feb 2023 [Online]. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/software-complexity

[5] "Affordability wall in safety critical systems," trust-in-soft.com, Feb 2023 [Online]. [Online]. Available: https://trust-in-soft.com/blog/2022/09/29/why-safety-critical-software-is-hitting-the-affordability-wall-and-how-system-developers-can-cut-its-cost-dramatically/

[6] P. Koopman, "Cost of highly safety critical software," betterembsw.blogspot.com, Feb 2023 [Online]. [Online]. Available: https://betterembsw.blogspot.com/2018/10/cost-of-highly-safety-critical-software.html

[7] R. CLUTTON, "A brief history of agile, part 1: The rise of waterfall," tanzu.vmware.com, Feb 2023 [Online]. [Online]. Available: https://tanzu.vmware.com/content/blog/a-brief-history-of-agile-part-1-the-rise-of-waterfall

[8] V. T. P, "Applying safety critical requirement based test practices for non-safety critical systems," ishitvtech.in, Feb 2023 [Online]. [Online]. Available: https://ishitvtech.in/pdf/7_1_Applying.pdf

[9] A. Hicken, "The shift-left approach to software testing," stickyminds.com, Feb 2023 [Online]. [Online]. Available: https://www.stickyminds.com/article/shift-left-approach-software-testing

[10] "Agile methodology: Advantages and disadvantages," ccaps.umn.edu, Feb 2023 [Online]. [Online]. Available: https://ccaps.umn.edu/story/agile-methodology-advantages-and-disadvantages

[11] R. Kasauli, E. Knauss, B. Kanagwa, A. Nilsson, and G. Calikli, "Safety-critical systems and agile development: A mapping study," in *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2018, pp. 470–477.

[12] "Rahat ibn rafiq publications," scholar.google.com, Feb 2023 [Online]. [Online]. Available: https://scholar.google.com/citations?user=nkjWthMAAAAJ&hl=en&oi=ao

[13] M. Lingg, "Automation of test case generation and software system modeling," arrayofengineers.com, Feb 2023 [Online]. [Online]. Available: https://www.arrayofengineers.com/post/white-paper-automation-of-test-case-generation-and-software-system-modeling

[14] "Ibm doors," ibm.com, Feb 2023 [Online]. [Online]. Available: https://www.ibm.com/docs/en/ermd/9.7.0?topic=overview-doors

[15] "Atcg example requirements," github.com, June 2023 [Online]. [Online]. Available: https://github.com/ArrayofEngineers/atcg-example-reqs