Using Fuzzy Logic & Symbolic Execution to Prioritize UML-RT Test Cases

Eric J. Rapos and Juergen Dingel
School of Computing
Queen's University
Kingston, Ontario, Canada
Email: {eric,dingel}@cs.queensu.ca

***Abstract*—The relative ease of test case generation associated with model-based testing can lead to an increased number of test cases being identified for any given system; this is problematic as it is becoming near impossible to run (or even generate) all of the possible tests in available time frames. Test case prioritization is a method of ranking the tests in order of importance, or priority based on criteria specific to a domain or implementation, and selecting some subset of tests to generate and run. Some approaches require the generation of all tests, and simply prioritize the ones to be run, however we propose an approach that prevents unnecessary generation of tests through the use of symbolic execution trees to determine which tests provide the most benefit to coverage of execution. Our approach makes use of fuzzy logic, specifically fuzzy control systems, to prioritize test cases that are generated from these execution trees; the prioritization is based on natural language rules about testing priority. Within this paper we present our motivation, some background research, our methodology and implementation, results, and conclusions.**

I. Introduction

When it is not possible to run all tests for a system for any number of reasons, it becomes necessary to determine which subset of tests can be run to achieve maximum coverage. This paper looks at our approach to prioritizing UML-RT test cases using fuzzy logic. The basis for the work is our previous work on the incremental generation of UML-RT test cases using symbolic execution as medium for generating tests[1][2].

A. Motivation

Model-based testing (MBT) makes use of models of systems (abstractions) as primary artifacts in software testing. In traditional testing, unit tests are based solely on lines of code, whereas in MBT they may be based on any number of different models. This combined with a number of other factors has made it easier to automatically generate complete and thorough test suites for a system. The downside is that these test suites are often very large, and can be redundant in some cases. The problem here is that not only does it take a significant amount of time to generate these tests, it also takes a long time to actually run the full suite tests (in the order of days in some cases).

B. Contributions

We propose using fuzzy logic to determine a subset of all tests for a system that will perform a reasonably complete test of the system, given some set of constraints on testing (resources, time, etc.). The choice of fuzzy logic for this task comes from the flexibility it provides, and the ability to describe rules in a natural language that test engineers can understand and adjust as necessary. Our choice to continue the use of symbolic execution as a means of test case generation allows us to obtain information about the nature of the execution of the systems, as well as insight into the behaviour of each individual test prior to their actual generation. This advanced knowledge provides additional savings in computation time. Thus, our work makes the following contributions:

- a method of selecting a *reasonably complete* subset of tests based on natural language rules and information available from symbolic execution trees, which provides improved coverage over random selection
- the ability to use only information available from symbolic execution in the prioritization process so as to avoid regeneration of tests that will not be run.
- a tool capable of implementing this selection and presenting results to testers

II. Background and Related Work

For this work, there are two main areas of related work that need to be discussed to ensure a basic understanding of the proposed work. The first area is Model-Based Testing, in which we present the relevant information from the related MSc thesis[1][2], and provide necessary background. The second section presents the relevant aspects of fuzzy logic, and the basics required for this project. The third section will deal specifically with comparisons to other methods for test case prioritization both in general and using fuzzy logic.

A. Model-Based Testing

Model-based testing (MBT) is the process of testing a system for which the primary artifacts are models of the system. Recent MBT work has trended towards the automation of this process, where tests are automatically generated from a model of the system, whether it is a behavioural model, a structural model, or some other type of model; the most common of course being behavioural as these types of models provide the most information about what the system is intended to do, therefore the most information about what needs to be tested.

In previous work[1][2], we studied the evolution of UML-RT models, and the impact of evolution on test case generation. Specifically, we looked at how the tests changed given a

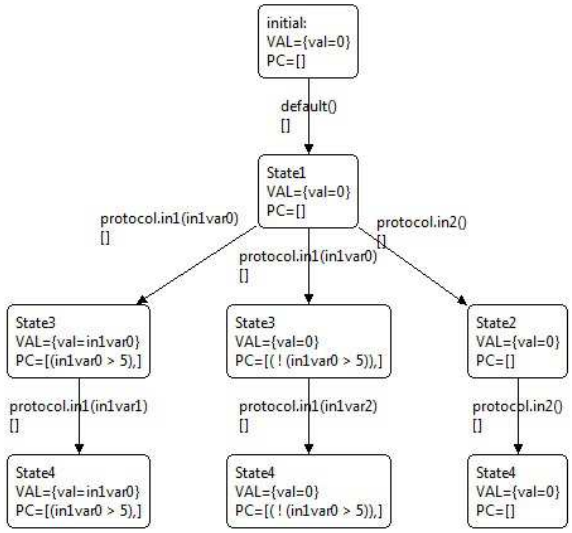


Fig. 1: Example Symbolic Execution Tree

catalog of mutations to the source models. The end goal was to develop a tool which was able to detect the changes to a model version and update the test suite with minimal computational effort, such that the new test suite was reflective of the new model.

We were successfully able to do this using symbolic execution trees as a medium for test case generation, an extension of the work of Zurowska and Dingel[3]; by symbolically executing the model, we were able to obtain an execution tree, from which we could systematically generate a test suite. In the case of UML-RT models, a test suite was simply a series of ordered inputs (as function calls), and the expected outputs. We were able to reduce the computation time by only performing symbolic execution when necessary on sub-graphs of the model, and the results were promising. However the work left one large area for improvement, in that it still would generate large test suites, which take long periods of time to run, and we wanted to come up with a way to identify a strictly necessary subset, which is where this work picks up.

For the purpose of this prioritization work, we will not be looking at the incremental test case generation aspect, but simply the prioritization of tests given information available from the symbolic execution tree for a model.

For context, in the existing work, once the symbolic execution tree is obtained, the tests are generated in a depth-first fashion, with each leaf node of the tree indicating the end of a test run; therefore the number of leafs is the number of tests required and the longest path is the largest test case. For example, the symbolic execution tree seen in Figure 1 would generate 3 tests, each of length 3 as seen in Figure 2. Given the results of this work, we can avoid the full generation and select only the necessary cases for generation and execution.

B. Fuzzy Logic

Fuzzy logic is a form of many-valued logic, meaning that it is not binary and allows infinitely many degrees of truth

```
Test Case 1:
        default();
        protocol.in1(in1var0);
        protocol.in1(in1var1);
Test Case 2:
        default();
        protocol.in1(in1var0);
        protocol.in1(in1var2);
Test Case 3:
        default();
        protocol.in2();
        protocol.in2();
```

Fig. 2: Resulting Test Suite for SET in Figure 1

and falsehood. The concept was first introduced by Zadeh in 1965[4]. This type of logic is very commonly associated with reasoning using natural language, and compared to computing with words. For example, the phrase "if you feel hot then take off your sweater" deals with the complexity of understanding what is meant by 'hot' in this context. Is 'hot' an absolute temperature value? Are there properties associated with it? Or is it simply meant in a way that is vague, but easily understood by humans? It is more than likely the last of these, and fuzzy logic looks at how to use this type of reasoning more formally, such that it can be computed.

Work in fuzzy logic has become more widespread, and the notion of a fuzzy logic control system has received a lot of attention: Lee provides some context on the fuzzy logic controller[5], and follow up work has been done with genetic algorithms[6] and neural networks[7]. While not necessarily required, a control system for test case prioritization could be applied using the same type of process, in that there are some number inputs that are used to determine an output (priority).

The accepted process for most fuzzy control systems involves 4 main steps[8]:

1) fuzzification
2) inference
3) composition
4) defuzzification

Fuzzification is the act of taking some crisp input value (an observable input) and determining its membership in a previously defined input fuzzy set (membership values between 0 and 1 for how much the observed value 'fits' into a specific set); for example a temperature of $10\,^{\circ}\mathrm{C}$ may be considered "cool", but not as cool as $6\,^{\circ}\mathrm{C}$, so they may receive values of 0.6 and 0.8 respectively in the fuzzy set "cool". **Inference** deals with determining what a set of fuzzy inputs actually means based on a specified rule (of which there are usually many); for example "if temperature is cool and you feel cool then add another layer" tells you what kind of action should be taken if the input values for the sets 'temperature' and 'you feel' match those in the rule. During inference a t-norm and s-norm are chosen for use in calculation of values. T-norms are used to define fuzzy *AND* operators, and s-norms are used to define fuzzy *OR* operators. **Composition** is simply the process of combining the results of all rules for your system in

some manner (there are many different ways presented in the literature); for example if you have 10 rules, 4 of which say you should add a layer, three of which say you shouldn't change the number of layers, and 3 of which say you should remove a layer, you can see that the system is suggesting (with not very strong confidence) to add a layer. Finally, **defuzzification** is the process of returning back to a crisp output based on the composed results, such that your system has an actionable result. It is not always the case that this is necessary, the fuzzy result may be more meaningful than the crisp, but there are many cases where a crisp result is needed; for example in our case, a specific priority is needed rather than a membership in a fuzzy set.

C. Related Work

While this is the first work on prioritizing UML-RT test cases using fuzzy logic, the concept of test case prioritization in general using fuzzy logic is not a new concept and there are a number of works on the subject.

Work by Chaudhary et. al.[9] focuses on prioritizing test cases for GUI based software using fuzzy logic. Focus was on the inputs (event type, event interaction, and count based criteria), which fit into five levels (very low, low, medium, high, and very high), thus producing 125 rules; each rule takes into account all inputs, much like our own implementation. However, it is worth noting that this approach only uses information about what the tests are aimed at testing as part of the factors for prioritization, whereas our methodology examines the potential effectiveness of a given test through coverage of symbolic states.

Similarly Malz et. al.[10] present their work on prioritizing test cases using software agents and fuzzy logic. With this work, the focus is on the use of software agents, which introduces a collaborative element to prioritization, with the agents working on different priority values, and determining final priority in cooperation with the other agents.

Alakeel presents work[11] on using fuzzy logic to prioritize regression testing of programs with assertions, which is another specific application of fuzzy test case prioritization. The focus on regression testing provides prior knowledge which is used in the prioritization, such as whether or not the assertions have been affected, partially affected, or not affected. However one downside to this approach is that prioritization focuses on the rate at which tests violate assertions in programs, which is taken to show that a programming fault exists, however does not necessarily take into account testing the desired behaviour of the system to ensure it acts as expected.

From these examples it is evident there is merit to using fuzzy logic to prioritize test cases, with each of these works demonstrating success. However an added benefit of our approach is the white-box style of prioritization that comes from the use of symbolic execution and the information it provides. Namely the ability to measure state coverage of a system allows for an accurate metric to measure success.

More generally, test case optimization on source code is another area where researchers have focused. A family of empirical studies by Elbaum et. al.[12] looks at a number of different methods for prioritizing test cases, comparing their methodology with random ordering of tests, as well as an optimal ordering (since defects are known). They look at approaches such as statement coverage, and fault detection probability, as well as combinations of these approaches which utilize prior testing results to add additional information. One difference with the type of prioritization presented here and our work is the prior existence of test cases; our method of prioritizing tests does not presuppose the existence of generated tests, and merely relies on the symbolic execution tree of the system (from which tests can be generated) and a knowledge of how tests are generated systematically. This adds the benefit of being able to prioritize which tests are created, as well as which tests are run.

Another empirical study by Rothermel et. al.[13] looks at prioritization of tests in a similar manner, comparing to random ordering and optimal ordering, however they also introduce unordered as a method of prioritization. In terms of the methods of prioritization to compare to these three baselines, the authors propose branch coverage, statement coverage, and fault exposing potential (FEP). One area where our approach is more robust is in determining coverage. Since our definition of coverage stems from symbolic state coverage, we have the advantage of being able to compare coverage of all possible execution paths through a system.

A third study of techniques similar to our work was conducted by Di Nardo et. al.[14] in which they look specifically at coverage based test case prioritization. They focus on four types of coverage criteria: total coverage, additional coverage, total coverage of modified code, and additional coverage of modified code. While the idea of prioritizing based on coverage is similar, this work still focuses on types of code level coverage such as statements, function calls, branches, etc.. Their results show that prioritizing tests with high coverage provides better fault detection, however we propose that the use of symbolic execution will provide additional information that is useful for coverage based test case prioritization.

Srikanth et. al.[15] provide another look at test case prioritization, in which they prioritize tests based on requirements, specifically the factors of customer assigned priority, requirements volatility, implementation complexity, and fault proneness. Prioritization based on requirements reveals less about the system than approaches such as ours, which utilizes the available information from symbolic execution to prioritize tests, much like a white-box method of prioritization.

Perhaps the work most closely related to ours is by Korel et. al[16] on test prioritization using system models for early fault detection. This method, using executable finite state machines (EFSMs) as models, would be equivalent to testing based solely on the UML-RT state machine models that are part of our work - leaving out the step of symbolically executing the system to provide more detailed information. Another major difference is that this method looks at merely assigning a test to be either high priority or low priority, then randomly ordering the high then low priority tests. Our method allows for a much finer granularity of priority, which is essentially infinite. Finally, their work requires the added step of generating or creating the EFSM, whereas the symbolic execution tree portion of our prioritization is already a necessary step in test case generation, requiring no additional steps in order to prioritize the tests.

Collectively, there are a number of works on both test case prioritization in general, as well as using fuzzy logic, however our approach proposes advantages over existing techniques, which involve the use of symbolic execution. Through obtaining the symbolic execution tree, we are able to discern information about the execution of the program to use in the prioritization of tests, mainly symbolic state coverage, but also the relative importance of a given test compared to other tests, the overall complexity of the whole system, and the amount of interaction with other systems (measured by amount of output signals generated by a particular test). Add to these factors the ability to know all of this information about the test suite without having to have first generated all of the tests (or regenerated if the case may be), since we leverage the symbolic execution tree to automatically generate tests and it becomes evident the contributions our work makes over existing test-case prioritization work.

As such we have chosen to continue to explore the concept of test case prioritization using fuzzy logic with our work in testing UML-RT, specifically highlighting the use of symbolic execution as the medium for test case generation, in an attempt to gather as much information about the resulting test suite prior to its generation and make use of execution information in the priority assignment decisions.

III. Methodology

In this section we present the process taken to implement test case prioritization for UML-RT models. There were three main areas of work for this project (input/output identification, rule selection, and presentation of results) which all feed into the tool implementation at the end of the work. The implementation subsection will deal with the mapping of our work to the four steps of the fuzzy control systems presented earlier in this paper, and how specifically they were implemented.

A. Input & Output Set Identification

The first step in programming any fuzzy control system is to determine what the inputs and outputs are going to be, and to present them as fuzzy sets which will be used throughout the process.

Recall, one of the main goals of this work is to utilize only information available from the symbolic execution tree, such that the full test suite need not be generated in order to prioritize, thus saving a step along the way. As such we took a look at the information available within the tree itself, as well as information that can be inferred from the tree given knowledge of the test generation process. The result was a set of four inputs which have an impact on how important it is to run each test case. The output of our system is naturally a priority level, however we also needed to determine how to represent this as a fuzzy set.

This step in the work maps to the **fuzzification** step for the inputs and the **defuzzification** step for the output. Each of the following sections will present one attribute, and how it was represented as a fuzzy set. It is important to note that the choices for the mappings of these fuzzy sets are based largely on the sample data set we have been working with, however they can be easily scaled and adjusted to suit any set of models, and fine-tuned to encompass multiple sets.

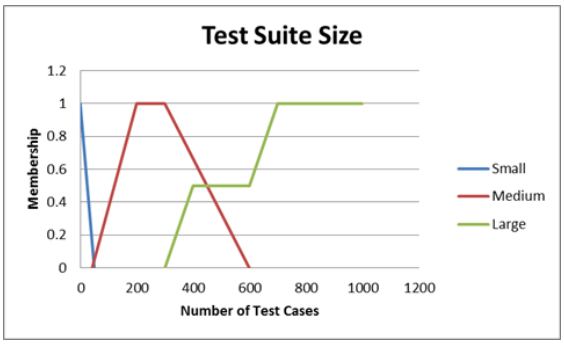


Fig. 3: Test Suite Size - Fuzzy Input Set

1) Test Suite Size - Input: The first input variable that we chose to examine was the size of the test suite in terms of the absolute number of test cases. This was considered to be an important factor on the priority of a given test for a number of reasons. First, if the overall size of the test suite is small, then it is less important to weed out test cases as run-time becomes less of an issue; each test in a small test suite can have a higher priority. On the other hand, as the test suite increases in size there becomes a need to lower the priority of some of the test cases.

For this input, it was necessary to come up with fuzzy sets to represent size; as such we developed three fuzzy sets for test suite size: small, medium and large. These sets are shown in Figure 3. In this instance a test suite is small if it has up to 50 test cases, medium from 40 to 600, and large if greater than 300, each with differing levels of membership.

The crisp inputs for this input are taken directly from the symbolic execution tree; each leaf node in the tree is a final state in a path, meaning that the number of leaf nodes is the number of tests that will be generated from the given tree. For example the SET in Figure 1 would be of size 3 (as demonstrated in Figure 2).

2) Symbolic Execution Tree Size - Input: The second aspect we chose to examine was the overall size of the symbolic execution tree, which is measured in the total number of symbolic states contained in the tree. The larger the tree (overall, not just breadth or depth), the more complex the system is, generally. This particular input allows us to add weight to test cases which come from a large symbolic execution trees as the more complex a system is, the more important each individual test case is.

Just as with the previous input, we chose to have three fuzzy sets representing symbolic execution tree size: small (0-700), medium (200-1000), and large ($>$ 700), each with differing levels of membership. These sets are shown in Figure 4.

The crisp input values for this input are calculated by counting the total number of symbolic states in the entire execution tree. While this seems like it would be an expensive calculation to perform, especially since it is similar in order to the generation of a full test suite, this is actually done at time of generation of the SET, and stored as an attribute, requiring no additional computation to use in the prioritization process. For example the SET in Figure 1 has a total size of 8.

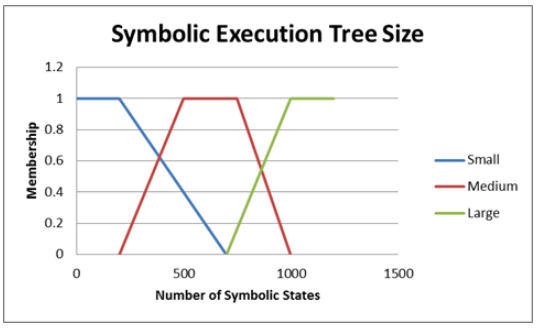


Fig. 4: Symbolic Execution Tree Size - Fuzzy Input Set

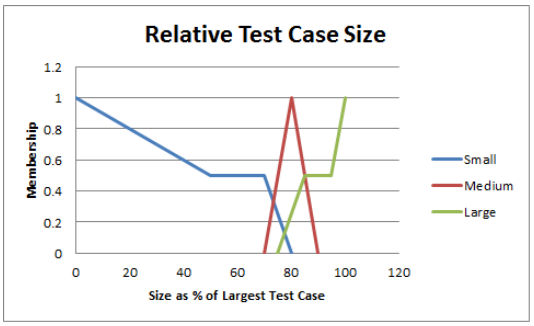


Fig. 5: Relative Test Case Size - Fuzzy Input Set

3) Relative Test Case Size - Input: The previous two inputs deal specifically with attributes of the entire test suite, which if taken alone would assign the same priority to every test case in a test suite, which defeats the purpose of this work; it is necessary to look at individual tests as well. This is why the next attribute is the relative size of the individual test cases compared to the whole test suite. This is an important input, as it is a good measure of the relative importance of the given test case; a larger test case will test more of the system and is therefore more important overall.

Again, just as with the previous two inputs, we were able to come up with three fuzzy input sets to represent the relative test case size: small (0-80%), medium (70-90%), and large (75-100%). These fuzzy sets are shown in Figure 5.

The crisp input values for this input are calculated by comparing the length of the test case to the longest test case in the test suite, and representing it as a percentage; therefore the largest test case receives a crisp input value of 100%, and one half that size is assigned a value of 50%. This helps differentiate between tests that are shorter (for example tests which terminate immediately based on a planned error) and those that run for a longer period (for example tests which demonstrate desired performance for a long period). Test case lengths are also stored as an attribute in the SET leaves during generation. In our running example from Figure 1, each test here would receive a priority of 100% since they are all of length 4, which is the longest test case; this is not always the case.

4) Output Significance - Input: The fourth and final input that we used dealt with the importance of a test case based on its output significance. In a UML-RT state machine, not only

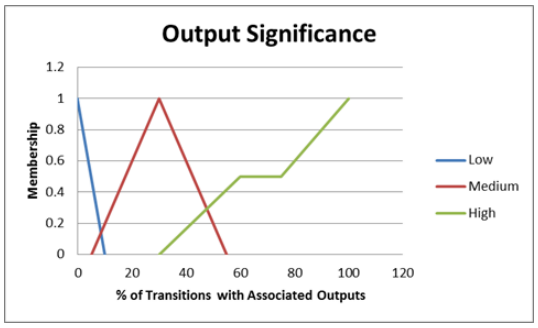


Fig. 6: Output Significance - Fuzzy Input Set

are inputs received from an outside source, output signals may also be sent to other capsules or state machines, and outputs can be logged. It was deemed that the more output generated by a specific test case, the more important it was; thus a test case with high output significance would have a higher priority, in general.

Again, this input divided nicely into three fuzzy sets, although they are slightly different as they don't necessarily deal with size, but significance. Therefore the three sets are: low (0-10%), medium (5-55%), and high (30-100%). These sets are shown in Figure 6.

The crisp input values for this set are obtained as a percentage of all of the inputs which have an associated output. For example if every input transition has an associated output, then the output significance would have a value of 100%. Just as with the other input values, this information is calculated at the time of generation and stored in the leaves of the SET. Our example SET in Figure 1 does not produce outputs due to its simplicity, so each of the tests would receive 0% as its calculated output significance, which provides full membership in the low class.

5) Test Case Priority - Output: As stated, it was obvious all along that the output would be a priority for a given test case, the question then became how to represent this, as well as the fuzzy sets.

The last item that was necessary was to determine how to assign values to the fuzzy outputs sets, and this choice was fairly arbitrary as the output need only be something numerical than can be ordered. For this, we chose an output value in the range from 0 to 1 - these will be the defuzzified results of the process which are used to determine the priority of each test case.

We decided that there should be four fuzzy sets that deal with priority: low (0-0.5), medium(0.4-0.6), high (0.45-0.95), and very high (0.9-1.0). These sets are shown in Figure 7. The reason for the change to four sets, while all of the inputs have three, is that there is a need for the "very high" set in order to assign a significantly higher priority value to specific test cases which merit a higher priority given only one of the inputs.

B. Rule Selection

Once we had a starting point (fuzzy input sets) and an end point (fuzzy output set) it was necessary to find a way to get

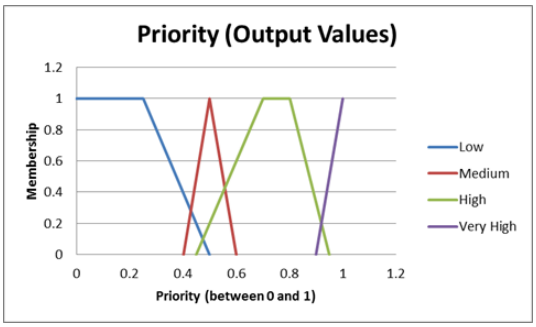


Fig. 7: Test Case Priority - Fuzzy Output Set

from one to the other, and the first step in this process is the selection of rules to infer priority, which is a mapping of the inference step of a fuzzy control system.

Rules for fuzzy control systems are of the form:

$if\ \langle antecedent \rangle\ then\ \langle consequent \rangle$

where $\langle antecedent \rangle$ can be any number of logical statements. For this implementation, we chose to have every rule consider all four inputs, and assign a value for the output, specifically, they are of the form:

```
if
        TestSuiteSize is W and
        SETSize is X and
        RelativeSize is Y and
        Output is Z
then
        Priority is A
```

where *W,X,Y,Z* are fuzzy input sets and *A* is a fuzzy output set.

Our first approach to generating the rules was to use a systematic approach, listing all possible combinations of the inputs, which provided $3^4 = 81$ rules to begin with. For each of the 81 rules we then chose an output value based on the specific input values based on logical reasoning using information about the testing process and goals. In an effort to reduce the redundancy of applying 81 rules, we began to search for a way to combine some of the rules in an appropriate manner. This was done by finding the rules where overlap exists in the output fuzzy set, and the input fuzzy sets differed by only one or two. For example, two rules which have small as the input set for the first three inputs, and the fourth inputs are low for the first and medium for the second, and both rules assign an output of high priority, can be combined into a single rule where the low and medium output significance is identified as low OR medium. This particular example can be seen as our first rule. Through this process, we were able to consolidate down to 39 rules, without any loss of information.

The resulting 39 rules can be found in Table I where each of the input columns map to *W,X,Y,Z* respectively, and the priority column represents *A*. Instances where a value could be one of two options are represented with a logical or (||), whereas instances where all three sets are accepted are represented with "- - -" (and the attribute is simply excluded from the rule during implementation).

It is important to note that since the rules are designed in such a way that a value for priority is calculated by each applicable rule, and not all rules will apply to all tests, there is no way that two contradictory rules can be applied equally, thus canceling each other out. However it is possible that two rules may assign membership in differing priority classes; since these values are aggregated in later stages of prioritization, this is not an issue, and the conflicting assignments contribute to a more granular priority value being assigned, since a test may have qualities of two (or more) different priorities, but usually one more than the others.

C. Presentation of Results

The last item of consideration before the actual implementation was to determine how results were to be presented. While it is excellent to have a priority for each test case, we had to determine what this meant to a tester of UML-RT models, and how best to present the results.

We came up with two methods for selecting the appropriate tests from the test suite such that it would automatically present only the tests that need to be generated and run. The first method was presenting only the tests that were above a certain priority threshold, meaning only tests that were of a set priority would be tested. The second selection method was to choose a certain percentage of the top priority test cases; whereas the previous method may only provide a small number of test cases, this method provides a consistent number of test cases, albeit they may have a lower priority overall.

The values for these are something that can (and should) be fairly fluid, meaning that the selection tool will prompt the user for a minimum priority value or a percentage of tests to select.

Additionally, we felt that it may also be desirable to simply output all of the tests, and their priorities, sorted from highest to lowest, to allow the tester the opportunity to select a number of test cases manually from the list.

Thus for this tool we came up with three display options presented to the user at the beginning of prioritization:

1) All Test Cases with Priority $\geq$ Some Threshold
2) The Top Percentage of Test Cases
3) All Test Cases (sorted by descending Priority)

D. Implementation

Just as was the case for the implementation of the test case generation[2], this project is implemented as a plugin for IBM's Rational Software Architect Real-Time Edition (RSA-RTE)[17], which is an IDE for UML-RT development. The plugin code is developed in Java, making use of the associated libraries for UML-RT.

In this section, we present the implementation in terms of the four steps involved in fuzzy logic control systems.

1) Fuzzification: To perform fuzzification of the input sets, it is first necessary to observe crisp values for the input described in Section III-A. To first do this, the selected UML-RT Capsule is symbolically executed, and the symbolic execution tree is stored in memory for analysis.

TABLE I: Fuzzy Rules

Test Suite Size	SET Size	Relative Size	Output Significance	Priority
SMALL	SMALL	SMALL	LOW || MEDIUM	HIGH
SMALL	SMALL	SMALL	HIGH	VERY HIGH
SMALL	SMALL	MEDIUM || LARGE	LOW || MEDIUM	HIGH
SMALL	SMALL	MEDIUM || LARGE	HIGH	VERY HIGH
SMALL	SMALL || LARGE	MEDIUM	LOW	HIGH
SMALL	SMALL || LARGE	MEDIUM	MEDIUM || HIGH	VERY HIGH
SMALL	MEDIUM	SMALL	LOW || MEDIUM	MEDIUM
SMALL	MEDIUM	SMALL	HIGH	HIGH
SMALL	MEDIUM	MEDIUM	LOW || MEDIUM	HIGH
SMALL	MEDIUM	LARGE	LOW || MEDIUM	HIGH
SMALL	LARGE	SMALL	- - -	HIGH
SMALL	- - -	LARGE	- - -	VERY HIGH
SMALL || LARGE	MEDIUM	MEDIUM	HIGH	VERY HIGH
MEDIUM	SMALL	MEDIUM	LOW	LOW
MEDIUM	SMALL	MEDIUM	MEDIUM	MEDIUM
MEDIUM	SMALL	LARGE	LOW || MEDIUM	MEDIUM
MEDIUM	SMALL	LARGE	HIGH	HIGH
MEDIUM	MEDIUM	SMALL	LOW || MEDIUM	LOW
MEDIUM	MEDIUM	SMALL	HIGH	MEDIUM
MEDIUM	MEDIUM	MEDIUM	LOW || MEDIUM	MEDIUM
MEDIUM	MEDIUM	MEDIUM	HIGH	HIGH
MEDIUM	MEDIUM || LARGE	LARGE	LOW	MEDIUM
MEDIUM	MEDIUM || LARGE	LARGE	MEDIUM || HIGH	HIGH
MEDIUM	LARGE	MEDIUM	LOW	MEDIUM
MEDIUM	LARGE	MEDIUM	MEDIUM	HIGH
MEDIUM || LARGE	SMALL	SMALL	- - -	LOW
MEDIUM || LARGE	SMALL	MEDIUM	HIGH	HIGH
MEDIUM || LARGE	LARGE	SMALL	LOW || MEDIUM	LOW
MEDIUM || LARGE	LARGE	SMALL	HIGH	MEDIUM
MEDIUM || LARGE	LARGE	MEDIUM	HIGH	VERY HIGH
LARGE	SMALL	MEDIUM	LOW || MEDIUM	MEDIUM
LARGE	SMALL	LARGE	LOW	HIGH
LARGE	SMALL	LARGE	MEDIUM || HIGH	VERY HIGH
LARGE	MEDIUM	SMALL	- - -	MEDIUM
LARGE	MEDIUM	MEDIUM	LOW	MEDIUM
LARGE	MEDIUM	MEDIUM	MEDIUM	HIGH
LARGE	MEDIUM || LARGE	LARGE	LOW || MEDIUM	MEDIUM
LARGE	MEDIUM || LARGE	LARGE	HIGH	HIGH
LARGE	LARGE	MEDIUM	LOW || MEDIUM	HIGH

The crisp values for the two test suite level inputs (Test Suite Size and Symbolic Execution Tree Size) are then measured and stored, and for each test case, the two test case level inputs (Relative Size and Output Significance) are measured and stored.

With these crisp values, we then obtain the membership in each of the three associated fuzzy sets for each input, using specific helper methods hard coded with the values from each of the graphs in Section III-A. Since the membership functions are all linear, this is a simple calculation to make. The result is 12 membership values (4 inputs, 3 sets each) for each test case.

With these twelve values, the inputs are considered to be fuzzified.

For example, our SET from Figure 1 would have the following crisp inputs:

```
Test Size: 3
SET Size: 8
Rel. Size: [100% 100% 100%]
Output Sig: [0% 0% 0%]
```

which would lead to the memberships in the fuzzy sets seen in the following Table (only first test case used as example):

TABLE II: Resulting memberships in fuzzy sets for first test case resulting from the SET from Figure 1

Attribute	Small/Low	Medium	Large/High
Test Suite Size	0.94	0	0
SET Size	1	0	0
Relative Size	0	0	1
Output Significance	1	0	0

2) Inference: The next step is the inference step, which involves utilizing the rules presented in Section III-B.

The internal representation of this step is a two-dimensional array for the result; the first dimension is the four output sets, and the second dimension is the rule number. Each rule will assign a membership value to one of the four outputs sets, thus the resulting two dimensional array will be of size 4-by-39, and each rule will contain only one non-zero entry.

The values are calculated using the rules from Section III-B; the t-norm and s-norm used for this work are *min*

and *max* respectively. Therefore, the calculation for the first rule would be as follows (capital letters represent indexing constants):

```
result[OUT.HI][0] =
    Math.min(membership[IN.TS][TS.SM],
    Math.min(membership[IN.SE][SE.SM],
    Math.min(membership[IN.RS][RS.SM],
    Math.max(membership[IN.OU][OU.LO],
         membership[IN.OU][OU.ME])));
```

The resulting 4-by-39 two-dimensional array provides us with inferred membership values for the four output fuzzy sets, which are then used in later steps.

In our running example, recall that the first test case from the SET in Figure 1 has membership: SMALL, SMALL, LARGE, LOW, which is only activated by the third rule - therefore this test case would be assigned a value for its membership in the *high priority*, and following the calculation style explained above, its membership in *high priority* would be calculated as:

```
min(0.94,1,(max(0,1)),(max(1,0)) = 0.94
```

giving this test case a membership of 0.94 in the High Priority fuzzy set.

3) Composition: The composition step deals with determining values for each of the four fuzzy sets based on the 39 rules. This was done by simply choosing the maximum value obtained from the rules for each of the four classes, thus composing the outputs into four values, one for each of the sets: low, medium, high, and very high. Recall the internal representation for these values being a 4-by-39 two-dimensional array, where each column represents the membership in a certain class, this is easily calculated by determining the maximum value in each column.

In our simple running example, since the membership values are discrete and do not overlap, only the one rule is used, therefore there is only non-zero membership in one fuzzy set, which is the result of 0.94 membership in High Priority.

4) Defuzzification: The final step taken in order to determine the priority of a given test case is to defuzzify the composed results into a crisp number. To do this, we chose to use the mean of maximums (MoM) method of defuzzifying. What this means is that given the four composed values, the tool simply takes the highest membership value for a set (or multiple sets if there is a tie), and determines the priority values associated with those membership values, and takes the mean of all values on that/those interval(s). The result is a single, crisp, value between 0 and 1 that represents the priority for that test case. This process is repeated for every test case, and the results are presented as described in Section III-C.

For our running example, a membership of 0.94 in high, and no membership in any of the other sets, is defuzzified to a crisp value of 0.747.

IV. Validation & Results

In order to validate our tool, we wanted to measure the tool's ability to select test cases which provide the best coverage of a system, which is measured by number of symbolic states within a state machine covered; the more states reached by a selected test suite, the more of the system that is covered. As such, symbolic state coverage was our metric of validation.

To test the implementation, we obtained 40 results for each of five test models. The five test models used are the five models presented in the appendices of prior work[2], and used throughout. They are five very different models, differing in size, complexity, output significance, and test suite size.

The 40 results for each model come from running at incremental 5% thresholds from 5-100, for both a fuzzy prioritization, and a random selection of tests, in order to compare the performance of our tool over random selection. For the random values however, each test was run 3 times, and the results were averaged, bringing the total number of runs on each model to 80. For example, Model 3 produces a symbolic execution tree with 689 states, by selecting 35% of all tests, our tool covers 306 states (44.4% coverage), where random selection of tests covers only 236 states (34.3% coverage) showing improved performance. Partial results (selection thresholds that are multiples of 10) can be seen in Table III and graphs showing the individual performance of each model can be seen in Figure 8.

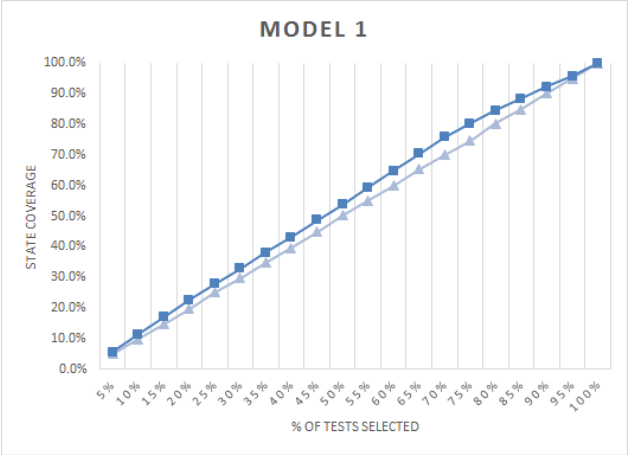
In terms of results, the tool was successfully able to generate an independent priority for each test case in a test suite which follows the logic of the fuzzy rules. In the majority of instances, the tool performed better than random selection with a few exceptions, where the results were comparable, showing that our method of fuzzy prioritization will generally show improved coverage for the same amount of testing. It is worth noting that in all of these examples, the resulting symbolic execution trees are fairly balanced, meaning that most of the tests are of the same or similar length with little variance. Because of this fact, there is minimal opportunity for improvement over random selection in terms of state coverage, and the results presented here show a substantial gain given the circumstances. However, in real-world examples, the execution trees will likely be less balanced (closest example of this is Model 3), and the gain over random selection will be much greater.

V. Future Work

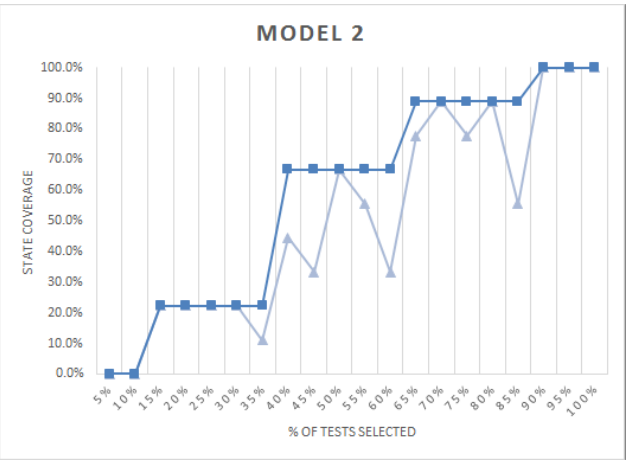
There are several areas where this work could be improved in future work, the most obvious being the fine tuning of the fuzzy rules. While changes were made during development, it is certainly possible to refine this set of rules even more. This type of fine tuning can include adjusting the membership in fuzzy sets as well, which may yield drastic changes in the results.

Another area that could provide some further improvement, and would warrant future work is the choice of other t-norms and s-norms other than min and max; this change may have a significant impact on the resulting test suites - a comparison with other common t-norms and s-norms would be interesting. Our suggestion for this expanded list would be the list presented in the work Gupta and Qi[18].

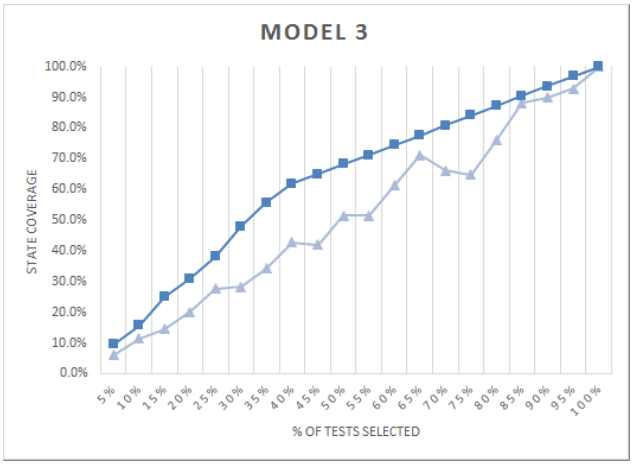
Another area where further work could be conducted would be a study of the performance gain, in terms of computation time, of this type of prioritization. While our work is successfully able to identify a set of models that should be



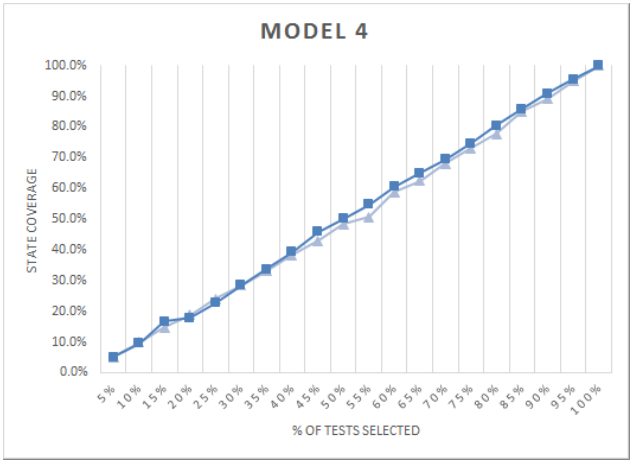
(a) Model 1



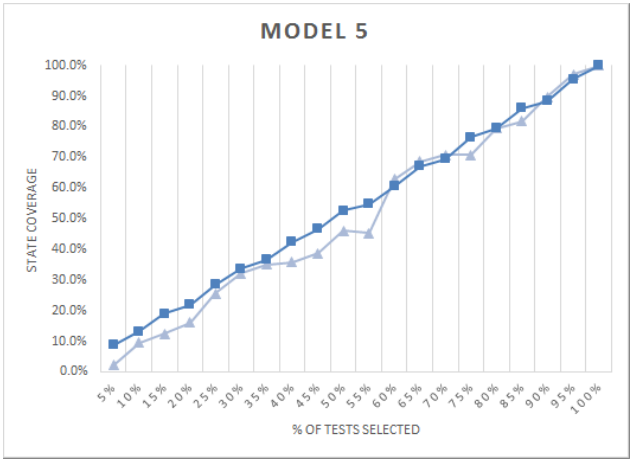
(b) Model 2



(c) Model 3



(d) Model 4



(e) Model 5

Fig. 8: Performance results of individual models - fuzzy prioritization (dark/square) vs. random selection (light/triangle)

TABLE III: Fuzzy Prioritization vs. Random Selection (number of symbolic states covered by selected tests)

	Model 1		Model 2		Model 3		Model 4		Model 5	
% Tests	**Fuzzy**	**Random**	**Fuzzy**	**Random**	**Fuzzy**	**Random**	**Fuzzy**	**Random**	**Fuzzy**	**Random**
10%	1435	1244	0	0	108	79	38	40	18	13
20%	2837	2493	2	2	213	139	72	76	30	22
30%	4140	3753	2	2	330	195	115	115	46	44
40%	5436	4996	6	4	426	294	158	154	58	49
50%	6757	6315	6	6	470	354	202	195	72	63
60%	8157	7533	6	3	513	422	244	237	83	86
70%	9530	8808	8	8	557	456	280	274	95	97
80%	10623	10084	8	8	601	524	324	313	109	109
90%	11611	11329	9	9	645	619	366	359	121	123
100%	12576	12576	9	9	689	689	403	403	137	137

tested based on the supplied fuzzy rules, as well as showing an improvement in coverage over random selection in most cases, it is unknown if the chosen models would reduce the amount of time required for testing. Another interesting metric to look at would be bug detection ability of selected tests.

VI. Conclusions

In this paper, we present our work on prioritizing UML-RT test cases using fuzzy logic. The approach follows the pattern of fuzzy logic control systems, making use of four inputs (test suite size, symbolic execution tree size, relative test case size, and output significance) to produce a single output (priority) for each test case in a UML-RT Test Suite. The system makes use of 39 fuzzy rules to infer priority (using min and max as the t-norm and s-norms respectively), and composes these rules by taking the maximum membership for each class. The result is defuzzified using the mean of maximums method to obtain the single crisp output for priority.

The tool implements the above methodology as a plugin to IBM RSA-RTE, an IDE for developing UML-RT Models. The results of running the tool on five example models show an improvement over random selection, in terms of symbolic state coverage, in almost every instance. Thus, the use of fuzzy logic, using only information available from symbolic execution, can help improve the symbolic state coverage of a system when only a subset of tests can be run.

References

[1] E. J. Rapos and J. Dingel, "Incremental test case generation for UML-RT models using symbolic execution," in *Proceedings of the Fifth International Conference on Software Testing, Verification and Validation*, ser. ICST '12, Montreal, Canada, Apr. 2012, pp. 962 –963.

[2] E. J. Rapos, "Understanding the effects of model evolution through incremental test case generation for UML-RT models," Masters Thesis, Queen's University, Kingston, ON, Sep. 2012. [Online]. Available: http://130.15.126.37/handle/1974/7534

[3] K. Zurowska and J. Dingel, "Symbolic execution of uml-rt state machines," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, ser. SAC '12. New York, NY, USA: ACM, 2012, pp. 1292–1299. [Online]. Available: http://doi.acm.org/10.1145/2245276.2231981

[4] L. Zadeh, "Fuzzy sets," *Information Control*, vol. 8, pp. 338–353, 1965.

[5] C. Lee, "Fuzzy logic in control systems: fuzzy logic controller. ii," *Systems, Man and Cybernetics, IEEE Transactions on*, vol. 20, no. 2, pp. 419–435, Mar 1990.

[6] F. Herrera, M. Lozano, and J. Verdegay, "Tuning fuzzy logic controllers by genetic algorithms," *International Journal of Approximate Reasoning*, vol. 12, no. 34, pp. 299 – 315, 1995. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0888613X9400033Y

[7] C.-T. Lin and C. Lee, "Neural-network-based fuzzy logic control and decision system," *Computers, IEEE Transactions on*, vol. 40, no. 12, pp. 1320–1336, Dec 1991.

[8] D. Kaur and K. Kaur, "Fuzzy expert systems based on membership functions and fuzzy rules," in *Artificial Intelligence and Computational Intelligence, 2009. AICI '09. International Conference on*, vol. 3, Nov 2009, pp. 513–517.

[9] N. Chaudhary, O. P. Sangwan, and Y. Singh, "Test case prioritization using fuzzy logic for gui based software," *(IJACSA) International Journal of Advanced Computer Science and Applications*, vol. 3, no. 12, pp. 222–227, 2012.

[10] C. Malz, N. Jazdi, and P. Gohner, "Prioritization of test cases using software agents and fuzzy logic," in *Proceedings of the Fifth International Conference on Software Testing, Verification and Validation*, ser. ICST '12, Montreal, Canada, Apr. 2012, pp. 483–486.

[11] A. M. Alakeel, "A fuzzy test cases prioritization technique for regression testing programs with assertions," in *Proceedings of the Sixth International Conference on Advanced Engineering Computing and Applications in Sciences*, 2012, pp. 78–82.

[12] S. Elbaum, A. Malishevsky, and G. Rothermel, "Test case prioritization: a family of empirical studies," *Software Engineering, IEEE Transactions on*, vol. 28, no. 2, pp. 159–182, Feb 2002.

[13] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Test case prioritization: an empirical study," in *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, 1999, pp. 179–188.

[14] D. Di Nardo, N. Alshahwan, L. Briand, and Y. Labiche, "Coverage-based test case prioritisation: An industrial case study," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, March 2013, pp. 302–311.

[15] H. Srikanth, L. Williams, and J. Osborne, "System test case prioritization of new and regression test cases," in *Empirical Software Engineering, 2005. 2005 International Symposium on*, Nov 2005, pp. 10 pp.–.

[16] B. Korel, L. Tahat, and M. Harman, "Test prioritization using system models," in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, Sept 2005, pp. 559–568.

[17] "Rational Software Architect: Realtime Edition (RSA-RTE)," 2014, [Online; accessed 14-October-2014]. [Online]. Available: $http://www-947.ibm.com/support/entry/portal/product/rational/rational_software_architect_realtime_edition?productContext=443691142$

[18] M. Gupta and J. Qi, "Theory of t-norms and fuzzy inference methods," *Fuzzy Sets and Systems*, vol. 40, no. 3, pp. 431 – 450, 1991, fuzzy Logic and Uncertainty Modelling. [Online]. Available: http://www.sciencedirect.com/science/article/pii/016501149190171L