# A Fuzzy Logic Based Approach for Model-based Regression Test Selection

Mohammed Al-Refai
Computer Science Department
Colorado State University
Fort Collins, CO, USA
Email: al-refai@cs.colostate.edu

Walter Cazzola
Department of Computer Science
Università degli Studi di Milano
Milan, Italy
Email: cazzola@di.unimi.it

Sudipto Ghosh
Computer Science Department
Colorado State University
Fort Collins, CO, USA
Email: ghosh@cs.colostate.edu

***Abstract*—Regression testing is performed to verify that previously developed functionality of a software system is not broken when changes are made to the system. Since executing all the existing test cases can be expensive, regression test selection (RTS) approaches are used to select a subset of them, thereby improving the efficiency of regression testing. Model-based RTS approaches select test cases on the basis of changes made to the models of a software system. While these approaches are useful in projects that already use model-driven development methodologies, a key obstacle is that the models are generally created at a high level of abstraction. They lack the information needed to build traceability links between the models and the coverage-related execution traces from the code-level test cases.**

**In this paper, we propose a fuzzy logic based approach named FLiRTS, for UML model-based RTS. FLiRTS automatically refines abstract UML models to generate multiple detailed UML models that permit the identification of the traceability links. The process introduces a degree of uncertainty, which is addressed by applying fuzzy logic based on the refinements to allow the classification of the test cases as retestable according to the probabilistic correctness associated with the used refinement. The potential of using FLiRTS is demonstrated on a simple case study. The results are promising and comparable to those obtained from a model-based approach (MaRTS) that requires detailed design models, and a code-based approach (DejaVu).**

***Index Terms*—fuzzy logic, model-based testing, regression test selection, UML models**

## I. Introduction

The purpose of regression testing is to test a new version of a system to ensure that the performed modifications do not introduce new faults to previously tested code [1]. Regression testing is one of the most expensive activities performed during the lifecycle of a software system. Regression test selection (RTS) approaches are used to improve regression testing efficiency [1]. RTS is defined as the activity of selecting a subset of test cases from an existing test set to verify that the affected functionality of a program is still correct [1], [2].

RTS is performed by analyzing the changes made to a system at the code or model level. Existing model-based RTS approaches use design models [2], [3], [4], [5]. The use of model-based RTS approaches is growing, and will have crucial importance in the future. For large systems, model-based approaches can scale up better than code-based approaches [6]. Maintaining traceability between the model and the test cases is more practical than maintaining traceability between the code and the test cases because dependencies can be specified at a higher level of abstraction [2]. The effort required for regression testing can be estimated at an early phase, i.e., at design time, and before propagating the changes to the code [2]. Test selection is performed at the model level using standard and widely used modeling notations (e.g., UML), which can be mapped to multiple programming languages making the RTS approach more reusable.

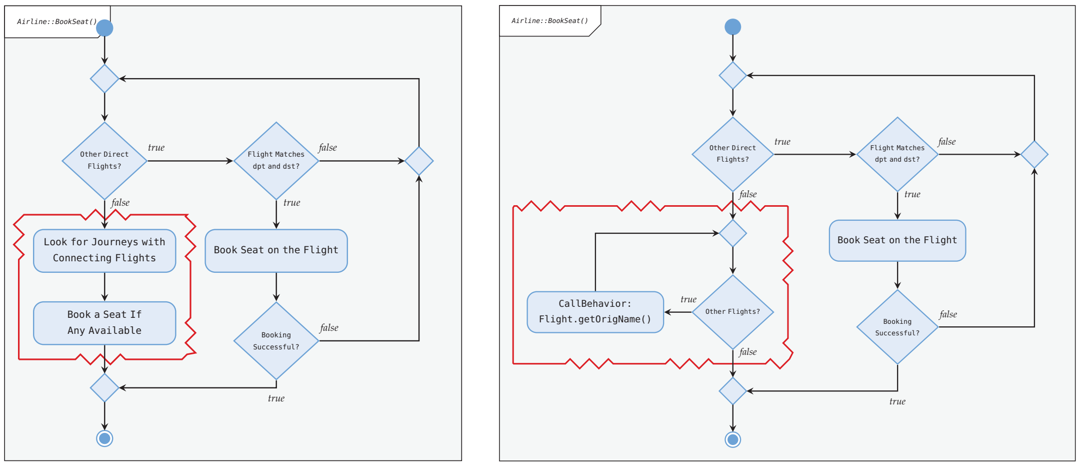
Despite the need for model-based RTS approaches, there is a major obstacle to the application of RTS at the model-level [2], [3], [4]. Models are generally created at a high level of abstraction and lack low-level details, such as use- and call-dependencies, which prevents relating the existing test cases to the models representing the system under test. This lack of traceability from requirements or design models to test cases is a known issue in model-based RTS, and is likely to severely limit its role [6]. This problem also makes it difficult to apply RTS in approaches that use design models to perform adaptation and validation at runtime, such as the DiVA project [7] that uses component-based models, which are specified at a high level of abstraction.

As a workaround, the approaches proposed by Briand et al. [2], Farooq et al. [3], and Al-Refai et al. [5] require their models to be detailed and complete with respect to the implementation of the software system, and to contain enough information to obtain the coverage of model elements when test cases are executed, which is not always a common practice [2]. In these cases, the models are simply used as a different representation of the code. Another possible solution would be to use incomplete coverage information of the test cases at the model-level, which will lead to inaccurate results.

To overcome this obstacle, we propose a new approach called FLiRTS that uses fuzzy logic to perform model-based RTS. The approach uses UML activity diagrams that model the behaviors of the system's methods at a high level of abstraction, and UML sequence diagrams that model the system's use case scenarios. In the activity diagrams, a single node can represent multiple code-level statements and its label can rarely be used to trace back to such a piece of code. This level of abstraction prevents relating the existing test cases to the activity diagrams as we will show in Section II.

From the provided activity diagrams, FLiRTS automatically

978-1-5386-3492-9/17 $31.00 © 2017 IEEE
DOI 10.1109/MODELS.2017.17



(a) Original. (b) Refinement.

Fig. 1. Activity Diagram Representing `Airline.bookSeat()` and One Possible Refinement.

generates more detailed ones called *refinements*. A refinement is an activity diagram that contains more flows and nodes than the one that was refined. For example, consider an activity diagram that contains an action node representing multiple call statements. In a refinement, such an action node can be replaced with multiple call behavior nodes. FLiRTS generates the refinements according to the provided usage scenario in the sequence diagram to avoid the generation of completely inconsistent and unrelated activity diagrams. The refinements from each activity diagram are combined and used in the RTS algorithm. This combined set of refinements contains enough information to permit the identification of traceability links between the models and the test cases. However, the obtained traceability links may be correct or incorrect depending on how compliant the used refinement is to the corresponding source code. FLiRTS classifies test cases as retestable or reusable [8] by using fuzzy logic with a degree of confidence related to the compliance of the used refinement. The classification is performed with respect to all possible combinations of refinements. The most trustworthy combination of refinements is used to get the final result.

## II. Motivating Example

Using models that are at a high level of abstraction hinders the building of traceability links between the models of the system and each test case, which makes it impossible to select the test cases. We illustrate this problem with an example. The airline reservation system (ARS), used as a running example in this paper, is a class project implemented by undergraduate students in a software engineering course. The portion of ARS used here consists of eight classes, and supports only basic seat booking capabilities but not the ability to prioritize flights by prices, airlines, or other criteria. Fig. 1(a) shows an activity diagram of the `Airline.bookSeat()` method, which is at a high level of abstraction. It lacks low level details, such as call behavior nodes to other activity diagrams. Fig. 2 shows

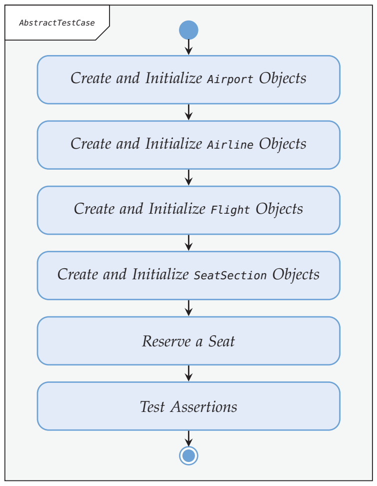

AbstractTestCase
Create and Initialize Airport Objects
Create and Initialize Airline Objects
Create and Initialize Flight Objects
Create and Initialize SeatSection Objects
Reserve a Seat
Test Assertions

Fig. 2. Activity Diagram Representing a Test Case.

the activity diagram of a test case that tests a scenario for booking a seat on a direct flight. The corresponding code view includes a direct call to `SystemManager.bookSeat()`, which calls `Airline.bookSeat()`.

The initial version of `Airline.bookSeat()` supports booking a seat on a direct flight, but does not consider trips involving connecting flights. In the next version, the activity diagram representing `Airline.bookSeat()` is adapted to include connecting flights. First, it searches for a direct flight that matches the inputs, and if such a flight is found, then a seat is booked on it. If no direct flights are found, then a trip is formed by finding flights that have the given airports as departure or destination airports, and combining these flights in a journey from the departure airport to the destination airport. This adaptation is performed by adding to the activity diagram in Fig. 1(a) two new action nodes (bordered in red) that describe the new functionality.

Regression test selection must be conducted because the behavior of `Airline.bookSeat()` was modified. The test case shown in Fig. 2 should be selected because it traverses the adapted `Airline.bookSeat()` method. The test case calls `SystemManager.bookSeat()`, which calls `Airline.bookSeat()`. However, building the traceability links between the activity diagram representing `Airline.bookSeat()` and the activity diagram of this test case is not possible, making it difficult to correctly classify the test case as retestable or reusable. The reason is that these activity diagrams (including the activity diagram of `SystemManager.bookSeat()` not shown here), are at a high level of abstraction, and lack information regarding the calls between them. Additionally, the labels of action nodes in these activity diagrams can refer to the same concept using different words/terminology, and therefore, we cannot relate these diagrams to each other based on these labels.

## III. Fuzzy Logic

Fuzzy logic uses a form of logic that is not binary (i.e., true or false), but relies on multiple truth values that can be between completely true and completely false [9]. We introduce the basic concepts by adapting an example taken from [10].

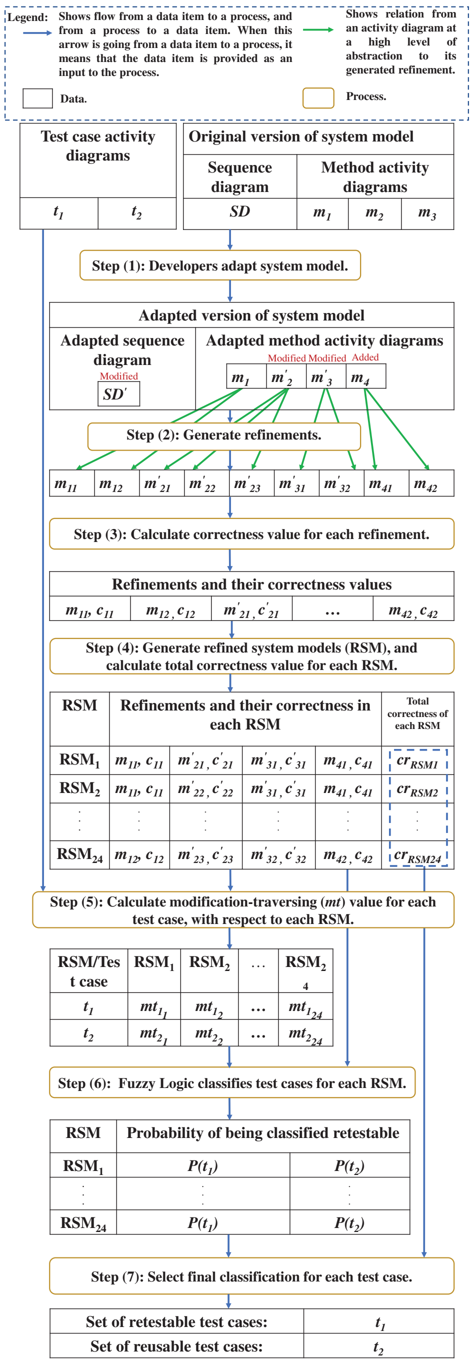
A fuzzy logic approach involves the steps of (1) fuzzification, (2) inference, and (3) defuzzification [9]. The approach uses input variables that take discrete values called input crisp values. For example, in a temperature control system managed by fuzzy logic, an input variable is the temperature, and an input crisp value can be 32°F or 40°F.

The fuzzification step maps the input crisp values to input fuzzy values by using input fuzzy sets. A fuzzy set is one that allows its members to have different values of membership in the interval [0,1] based on a membership function that defines for each fuzzy set how a value within the input space of a fuzzy set is mapped to a membership value between 0 and 1. For example, input fuzzy sets for the temperature variable can be *too cold*, *cold*, and *warm*. An input crisp value of 32°F fits in each of these sets with a specific membership value, called input fuzzy value. For example, if the input crisp value 32°F fits in the *cold* set with a membership value 0.6, then the input fuzzy value for 32°F is 0.6 with respect to the *cold* set.

Inferencing evaluates predefined inference rules using the input fuzzy values obtained in the fuzzification step. An inference rule can be "*if the temperature is too cold, then turn on the heater*". Evaluating all the inference rules produces a set of fuzzy output values. Defuzzification combines the fuzzy output values and produces a final output crisp value.

## IV. FLiRTS: Fuzzy Logic Based RTS Approach

FLiRTS automatically generates refinements from the provided activity diagrams subject to some constraints, uses the refinements to calculate the input values to be provided to a fuzzy logic-based classifier, and uses the classifier to attach probabilities to test cases to classify them as retestable or reusable. FLiRTS does not care how the original tests were created. Fig. 3 shows the main steps of FLiRTS.



Legend:
Shows flow from a data item to a process, and from a process to a data item. When this arrow is going from a data item to a process, it means that the data item is provided as an input to the process.
Shows relation from an activity diagram at a high level of abstraction to its generated refinement.
Data.
Process.
Test case activity diagrams
$t_1$ $t_2$
Original version of system model
Sequence diagram
Method activity diagrams
$SD$ $m_1$ $m_2$ $m_3$
Step (1): Developers adapt system model.
Adapted version of system model
Adapted sequence diagram
Modified
$SD'$
Adapted method activity diagrams
Modified Modified Added
$m_1$ $m'_2$ $m'_3$ $m_4$
Step (2): Generate refinements.
$m_{11}$ $m_{12}$ $m'_{21}$ $m'_{22}$ $m'_{23}$ $m'_{31}$ $m'_{32}$ $m_{41}$ $m_{42}$
Step (3): Calculate correctness value for each refinement.
Refinements and their correctness values
$m_{11}, c_{11}$ $m_{12}, c_{12}$ $m'_{21}, c'_{21}$ … $m_{42}, c_{42}$
Step (4): Generate refined system models (RSM), and calculate total correctness value for each RSM.
RSM
Refinements and their correctness in each RSM
Total correctness of each RSM
$RSM_1$ $m_{11}, c_{11}$ $m'_{21}, c'_{21}$ $m'_{31}, c'_{31}$ $m_{41}, c_{41}$ $cr_{RSM1}$
$RSM_2$ $m_{11}, c_{11}$ $m'_{22}, c'_{22}$ $m'_{31}, c'_{31}$ $m_{41}, c_{41}$ $cr_{RSM2}$
$RSM_{24}$ $m_{12}, c_{12}$ $m'_{23}, c'_{23}$ $m'_{32}, c'_{32}$ $m_{42}, c_{42}$ $cr_{RSM24}$
Step (5): Calculate modification-traversing ($mt$) value for each test case, with respect to each RSM.
RSM/Test case $RSM_1$ $RSM_2$ … $RSM_{24}$
$t_1$ $mt_{1_1}$ $mt_{1_2}$ … $mt_{1_{24}}$
$t_2$ $mt_{2_1}$ $mt_{2_2}$ … $mt_{2_{24}}$
Step (6): Fuzzy Logic classifies test cases for each RSM.
RSM Probability of being classified retestable
$RSM_1$ $P(t_1)$ $P(t_2)$
$RSM_{24}$ $P(t_1)$ $P(t_2)$
Step (7): Select final classification for each test case.
Set of retestable test cases: $t_1$
Set of reusable test cases: $t_2$

Fig. 3. FLiRTS process.

The original version of the system model contains (1) a UML sequence diagram that describes the usage scenarios of the application, and (2) activity diagrams that model the behavior of the system's methods. The sequence diagram only uses objects and methods that are specified in the UML class diagram of the whole system.

Developers adapt the sequence and activity diagrams of the software system in step 1. In step 2, refinements are automatically generated from each activity diagram that exists in the adapted version of the system model. The generation process is constrained by the adapted UML sequence diagram.

A refinement generated from an activity diagram, $A$, can be more or less correct depending on how much it is compliant with the expected method implementation represented by $A$. A correct refinement is one where (1) each element in the refinement (i.e., decision, loop, and call behavior) has a corresponding element in the expected method implementation, and (2) the order of the elements in the refinement matches the order of their corresponding elements in the expected method implementation. A non-compliant refinement can (1) have extra elements that do not exist in the expected method implementation, (2) miss some elements that do exist in the expected method implementation, and (3) have a mismatch between the order of some/all of the elements in the refinement and that of the corresponding elements in the expected method implementation. The measure of compliance is based on counting the differences with an optional weighting mechanism to distinguish between different kinds of mismatches, and then normalizing them on the interval [0, 10]. A value of 10 means that there are no differences. In step 3, the correctness value is calculated for each generated refinement.

Each test case is modeled by an activity diagram that includes call behavior nodes, each of which directly links to an activity diagram of a system method. The link between activity diagrams is by name and it holds even when the activity diagrams are refined since each refinement maintains the name of the activity diagram it is refining. Each activity diagram in the system model leads to several possible refinements. A *refined system model* is one where each activity diagram is replaced by one of its refinements. Several combinations of the refinements are possible, which leads to the creation of several refined system models (step 4). Depending on the refinements used in a system model, a test case may or may not traverse it. The reliability of the traversing information is directly dependent of the correctness of the used refinements. Fuzzy logic is used to address the uncertainty introduced by the process employed to find out the traceability link.

To apply fuzzy logic, we define two input variables, $cr$ and $mt$. The crisp value of $cr$ is defined in terms of the extent to which a test case traverses correct refinements in a refined system model (step 4). The crisp value of $mt$ is defined in terms of the extent to which a test case traverses modified activity diagrams in a refined system model (step 5). Fuzzy sets are defined for both the input variables.

We define an output variable corresponding to the test case classification and define fuzzy sets for this output variable. Step 6 applies the fuzzy logic classifier. The final results of FLiRTS for each test case $T$ is a set of refined system models, and the probabilities for *Retestable* and *Reusable* associated with $T$. In step 7, the final classification for $T$ is selected based on the probabilities associated with the most trustworthy refined system model. If such a system model cannot be determined, then the probabilities from all refined system models that are above a threshold are used.

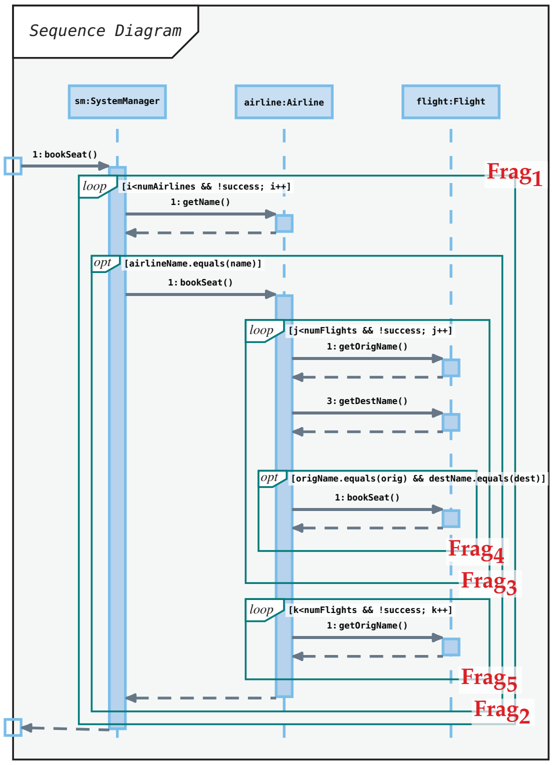


Fig. 4. Partial Sequence Diagram.

## A. Generate Refinements of Activity Diagrams

A naive approach randomly generate refinements. For example, existing action nodes in the activity diagram can be refined by or replaced with call behavior nodes for each operation in the class diagram. This can result in a large number of refinements, many of which will have a low level of compliance, adversely affect the reliability of the test classification results. Therefore, the refinement generation process must be constrained to favor the generation of refinements with a higher degree of compliance. FLiRTS uses the adapted sequence diagram of use case scenarios for this purpose.

To continue our running example, the sequence diagram shown in Fig. 4 represents two scenarios to reserve a seat on a direct flight or on connecting flights. Due to space limitations, this diagram is partial and does not show all the lifelines, fragments, and messages for these scenarios. We named the combined fragments in this diagram to make it easy to refer to them in the text. Combined fragment "*Frag3*" is responsible for finding a direct flight and reserving a seat on it.

To refine each high level activity diagram, we start from its corresponding message in the sequence diagram and navigate through the elements of the message to refine the elements of the activity diagram. Algorithm 1 shows how FLiRTS generates the refinements. It takes as inputs (1) an activity diagram representing a method behavior, (2) a sequence diagram representing the use case scenarios of the system, and (3) a message in this sequence diagram to be used to refine the

```
Algorithm 1: refineActivityDiagram($ad$, $msg$, $sd$)
  Input  :
    $ad$: Activity diagram.
    $msg$: Message in a sequence diagram.
    $sd$: Sequence diagram containing the message $msg$.
  Output:
    $R_e$: Set of refined activity diagrams.
  $R_e$=$\emptyset$
  Set $D_s = \emptyset$;
  $S_e$ = getElementsFromSequenceDiagram($msg$, $sd$);
  $A_e$ = getElementsFromActivityDiagram($ad$);
  ActivityDiagram $ref = ad$;
  for each combined fragment $c \in S_e$ do
      for each decision node $d \in A_e$ do
          if d.nl = c.nl then
              /* $nl$ refers to the element nesting level */
              $D_s.add((c, d))$;
          end
      end
  end
  $R_e$ = createRefinements($ref$, $D_s$);
  for each refinement $rf \in R_e$ do
      for each message $m \in S_e$ where m.nl=0 do
          addCallBehaviorNode($rf$, 0, $m$, NULL);
      end
  end
  return $R_e$;
```

corresponding activity diagram. The algorithm returns a set of refinements generated from the activity diagram. We illustrate the algorithm using the activity diagram in Fig. 1(a) that represents `Airline.bookSeat()`, the partial sequence diagram presented in Fig. 4, and the `Airline.bookSeat()` message that is sent from the `SystemManager` lifeline to the `Airline` lifeline.

Algorithm 1 extracts information about the elements that start from the execution specification of the input message and the elements in the input activity diagram. The information extracted from each element contains its nesting level and type. The nesting level of an element is defined as the number of combined fragments that surround the element, where the outermost combined fragment starts from the execution specification of the input message, $msg$. For example, the nesting level of "*Frag3*" is zero because it is not surrounded by any fragment that starts from the execution specification of `Airline.bookSeat()`. The nesting level of an element in the activity diagram is defined with respect to how deep it is located inside nested decision-merge structures. For example, the nesting level of the decision node labeled "*Flight Matches dpt and dst?*" is one because it is contained inside a decision-merge that forms a loop structure whose decision node is labeled "*Other Direct Flights?*".

Algorithm 1 navigates through the combined fragments of the message and the decision nodes of the activity diagram, and checks for matches between them based on their nesting levels. If a combined fragment in the sequence diagram matches a decision node, $D$, in the activity diagram, then new nodes and transition flows are created to form a new structure corresponding to the combined fragment (e.g., loop structure for loop fragment, and decision-merge structure for alt fragment). If the combined fragment contains messages, then for each of these messages, a new call behavior node is created inside the new structure. Three new refinements are created by adding the new structure (1) before decision node $D$ in the first refinement, (2) after $D$ in the second refinement, and (3) by replacing $D$ in the third refinement.

In our example, combined fragment "*Frag5*" matches the decision node labeled "*Other Direct Flights?*" because both of them are at the same nesting level. Thus, three new refinements are created. In one refinement (Fig. 1(b)), a new decision structure is created and added to the transition flow labeled "false" that is outgoing from the decision node labeled "*Other Direct Flights?*". The new decision construct replaces the two action nodes on that transition flow.

Finally, Algorithm 1 iterates through all the messages that were sent by the lifeline as a result of receiving the message that was provided as an argument to Algorithm 1. These messages are at nesting level zero. The algorithm adds call behavior nodes for these messages on the main transition flow in each of the refinements.

We defined a set of operators to refine an activity diagram by adding nodes and structures (*AddActionNode*, *AddCallBehaviorNode*, *AddDecisionStructure*, and *AddLoopStructure*). The set also contains the corresponding deletion operators. The set of operators is minimal and covers all the possible unitary changes that can be performed on an activity diagram (e.g., adding or removing a call action, replacing an action node by a new decision structure, adding a new loop, and adding call nodes inside the loop).

The activity diagram constructs that the operators support are action, decision, merge, call behavior, start, and end nodes. Each operator takes input parameters. For example, the *AddActionNode* operator takes as input (1) a new action node that will be added to an activity diagram, (2) the target activity diagram, (3) the existing flow to which the new action node will be added, and (4) the existing node after which the new action node will be added. The *AddDecisionStructure* operator takes extra inputs, such as decision and merge nodes, and flows between the nodes.

## B. Prepare Inputs for Fuzzy Logic classifier

The first input variable $mt$ takes crisp values representing the extent to which a test case traverses modified activity diagrams in a refined system model. This value is defined as the minimum number of call behavior nodes that need to be traversed by the test case to reach a refinement generated from a modified activity diagram.

The second variable $cr$ takes crisp values representing the extent to which the test case traverses correct refinements in a refined system model. This value is calculated by averaging the compliance values for the refinements in the model, which, in turn, are calculated as described earlier.

In our running example, suppose that the detailed activity diagram of a test case $T_1$ contains a call behavior node that calls

TABLE I
FUZZY LOGIC INPUTS AND OUTPUTS FOR TEST CASE $T_1$

| Refined System Model | Input Crisp Values | | Output Crisp Values | |
|---|---|---|---|---|
| | *mt* | *cr* | *reusable* | *retestable* |
| $\{S_1, A_1\}$ | 2 | 8.5 | 0 | 100% |
| $\{S_1, A_2\}$ | 2 | 5.7 | 35% | 65% |
| $\{S_2, A_1\}$ | 2 | 7.5 | 0 | 100% |
| $\{S_2, A_2\}$ | 2 | 5.4 | 35% | 65% |

the activity diagram representing `SystemManager.bookSeat()`. Suppose that $S_1$ and $S_2$ are two refinements generated from this activity diagram, and $A_1$ and $A_2$ are two refinements generated from the activity diagram representing `Airline.bookSeat()`. The Cartesian product $\{S_1, S_2\} \times \{A_1, A_2\}$ represents all the possible refined system models when `SystemManager.bookSeat()` and `Airline.bookSeat()` are the only activity diagrams in the system model. Both $S_1$ and $S_2$ contain a call behavior node that calls $A_1$ and $A_2$.

The input crisp value of *mt* for $T_1$ with respect to the refined system model $\{S_1, A_2\}$ is 2 because two call behavior nodes need to be traversed by $T_1$ to reach $A_2$ (i.e., a call from $T_1$ to $S_1$ followed by a call from $S_1$ to $A_2$). To calculate the input crisp value of *cr* for $T_1$, assume that the correctness value of $S_1$ is 8 and $A_2$ is 7. The input crisp value of *cr* for $T_1$ with respect to the refined system model $\{S_1, A_2\}$ is (8+7)/2=7.5. Table I shows the input crisp values assigned to $mt$ and $cr$ for each refined system model that is traversed by $T_1$.

## C. Apply the Fuzzy Logic Classifier

We first define the input fuzzy sets for the input variables $mt$ and $cr$. The sets are *High*, *Medium*, and *Low*. Fig. 5 shows the input fuzzy sets defined for $cr$. Each fuzzy set represents a degree of the correctness of the refinements traversed by a test case. An input crisp value assigned to *cr* can fit in each of these sets with a specific membership value. We also define an output variable called *testClassification* that has two output fuzzy sets called *Retestable* and *Reusable*.

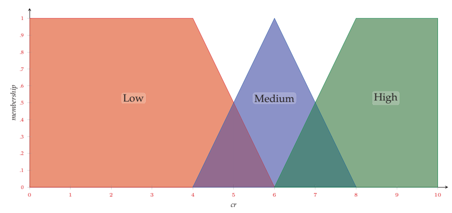


Fig. 5. Fuzzy Sets for the Variable *cr*

The fuzzy logic process used to classify the test cases involves three steps. In step 1, the input crisp values that are assigned to the input variables *mt* and *cr* are fuzzified based on the input fuzzy sets defined for each of these input variables. For example, the input crisp value assigned to *cr* for $T_1$ with respect to the refined system model $\{S_2, A_1\}$ is 7.5 (from Table I), and its membership value is 0.7 for the *High* set and 0.3 for the *Medium* set (from Fig. 5).

In step 2, inference rules are applied to the fuzzy inputs obtained in the fuzzification step to produce a set of fuzzy outputs. We defined a set of inference rules based on the fuzzy inputs. Here is an example of an inference rule:

```
if cr is High and mt is Medium
then testClassification is Retestable
```

In step 3, the defuzzification process combines the fuzzy output values to produce an output crisp value for the output variable *testClassification*. The output fuzzy sets, *Retestable* and *Reusable*, map the output crisp value to the probabilities of the test case for being *Retestable* and *Reusable*. These probabilities are shown in columns 4 and 5 in Table I. For each test case, the output of the fuzzy logic system is a set of refined system models, and the probability values for *Retestable* and *Reusable* that are associated with each refined system model.

We classify each test case by considering the probability values associated with the refined system model that has the highest correctness value. For example, in Table I, the refined system model with the highest correctness value is $\{S_1, A_1\}$. Test case $T_1$ is classified as retestable because its probability for being *Retestable* is 100% with respect to this refined system model. If such a refined system model cannot be determined, then we use the probability values from all refined system models that are above a specific threshold.

## V. Pilot Study and Discussion

We conducted a pilot study using the ARS system to compare the test classification results obtained using FLiRTS and two other RTS approaches. One is a code-based approach called DejaVu [11] for Java programs. The other is model-based [5]; we call the approach MaRTS in this paper. MaRTS classifies test cases based on changes performed to UML class and executable activity diagrams. Each activity diagram models a method of the software system. In contrast to FLiRTS, the activity diagrams used in MaRTS are at a low level of abstraction. Each action node is associated with the corresponding code snippet from the program. When model execution flow reaches an action node, the code snippet associated with the action node is executed.

We used nine test cases in this study. We adapted ARS at both the code and model levels as described in Section II. We applied FLiRTS to the models and DejaVu to the code. We applied MaRTS to a different version of the activity diagrams of the ARS system; these diagrams are executable and we had previously used them to evaluate MaRTS [12].

*a) Test classification results:* DejaVu and MaRTS classified the same 8 test cases out of 9 as retestable and the remaining one as reusable. With FLiRTS, there were no ties involved with the correctness values of the refined system models. Thus, we considered the probabilities associated with the refined system model with the highest correctness value. Each of the 8 test cases classified as retestable by the other approaches was also classified by FLiRTS as retestable with a probability higher than 95%. The test case that was classified as reusable by the other approaches was also classified as reusable by FLiRTS with probability value equal to 80%.

TABLE II
FUZZY LOGIC OUTPUTS FOR A TEST CASE CLASSIFIED AS RETESTABLE

| **Normalized correctness values** | **Probability** | |
|---|---|---|
| | *Reusable* | *Retestable* |
| 9.2 | 2% | 98% |
| 8.9 | 4% | 96% |
| 8.4 | 6% | 94% |
| 7.8 | 10% | 90% |
| 6.1 | 24% | 76% |

For a retestable test case, Table II shows examples of the correctness values of five refined system models, and the *Retestable* and *Reusable* probabilities obtained for the test case from these models. In this example, the highest correctness value is 9.2 out of 10. From this system model, we get a 98% probability for the test case to be retestable.

*b) Generalizability:* We cannot generalize the results from one study that used a small system, only nine test sets, and simple scenarios.

*c) Thresholds:* Currently, we do not have a specific threshold for the probability value that can be used to choose between reusability and retestability. In this study we considered a probability value that is at least 80% to be good enough to classify a test case as reusable. However, we cannot generalize this probability value to other subjects and test cases. We plan to evaluate our approach on additional subjects to define a generalizable threshold as well as a criterion for tuning our input fuzzy sets.

*d) Safety of FLiRTS:* As mentioned in Section IV-A, the refinements are generated by applying a minimal set of operators that cover all possible unitary changes. Any possible refinement can be expressed as a combination of these operators. Since we do not know the expected implementation, the operators are applied randomly but constrained by the provided sequence diagram as explained. While this can generate refinements with a low degree of trustworthiness, the minimality property ensures that the refinement close to the expected implementation will also be generated. The fuzzy logic system selects that one to calculate the RTS result.

## VI. RELATED WORK

We summarize related work on (1) model-based RTS, and (2) fuzzy logic-based RTS and test prioritization.

*Model-based RTS approaches.* Briand et al. [2] select test cases based on changes performed to UML use case diagrams, class diagrams, and sequence diagrams. Farooq et al. [3] identify changes made to UML class and state machine diagrams, and use the impacted and changed elements of the state diagrams to select test cases. Zech et al. [4] presented a generic model-based RTS platform controlled by OCL queries. The approach is based on identifying changes made to the models, and selecting the models representing the test cases based on the changes. Korel et al. [13], Ural et al. [14], and Lity et al. [15] proposed model-based RTS approaches that are based on state machine diagrams.

Al-Refai et al. [5] proposed a model-based RTS approach (MaRTS) to classify test cases based on changes performed to UML class and executable activity diagrams.

None of these approaches use fuzzy logic. They are based on using models that are at a low level of abstraction and contain enough information that permits building the traceability links between the system models and the test cases.

*Fuzzy logic-based approaches for RTS and test prioritization.* Xu et al. [16] used fuzzy expert systems to select test cases when the source code and its change history are not available. The fuzzy expert systems select relevant test cases by correlating the knowledge represented by one or more items, such as customer profile, analysis of test case coverage and results, system failure rate, and change in system architecture. This approach assumes that the knowledge is available and can be used to provide inputs to the fuzzy expert system. Malz et al. [17] use software agents and fuzzy logic for prioritizing test cases to increase the test effectiveness and fault detection rate. The software agents perform collaborative work on different priority values, where the final priority is determined based on the cooperation between the software agents. Both the approaches are code-based, and assume that the test coverage information is available.

Rapos et al. [10] proposed a fuzzy-logic and model-based approach to prioritize test cases using information available from the symbolic execution tree obtained from a model. The inputs to the fuzzy-logic system are test suite size, symbolic execution tree size, and relative test case size. The fuzzy-logic system produces a single crisp output called priority for each test case. The test cases are prioritized based on the crisp outputs. This approach assumes that the coverage information from test cases at the model-level is available, and does not target models that are at a high level of abstraction.

## VII. CONCLUSIONS AND FUTURE WORK

We proposed a model-based RTS approach called FLiRTS, which uses fuzzy logic to classify existing test cases as retestable and reusable based on changes performed to UML activity diagrams that represent behaviors of a software system at a high level of abstraction. The initial results from FLiRTS were comparable to DejaVu and MaRTS.

We will perform large scale empirical studies to compare the reduction, fault detection ability, precision, and safety of FLiRTS with other RTS approaches. A formal argument on the safety property will be developed. We will improve the algorithm used in FLiRTS to minimize the number of incorrect refinements that are generated by taking into account additional characteristics when performing structural matching, such as the order of the combined fragments that are within the same nesting level in the sequence diagram. We will define appropriate threshold values. We will extend the approach to support test suite minimization and prioritization, which will require changes to input variables and the fuzzy logic system.

## ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation under Grant No. CNS 1305381.

## REFERENCES

[1] M. J. Harrold, “Testing Evolving Software,” *Journal of Systems and Software*, vol. 47, no. 2-3, pp. 173–181, Jul. 1999.

[2] L. C. Briand, Y. Labiche, and S. He, “Automating Regression Test Selection Based on UML Designs,” *Journal on Information and Software Technology*, vol. 51, no. 1, pp. 16–30, Jan. 2009.

[3] Q.-u.-a. Farooq, M. Z. Z. Iqbal, Z. I Malik, and M. Riebisch, “A Model-Based Regression Testing Approach for Evolving Software Systems with Flexible Tool Support,” in *Proceedings of the 17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems (ECBS’10)*. Oxford, UK: IEEE, Mar. 2010, pp. 41–49.

[4] P. Zech, M. Felderer, P. Kalb, and R. Breu, “A Generic Platform for Model-Based Regression Testing,” in *Proceedings of the 5th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA’12)*, ser. Lecture Notes in Computer Science 7609, T. Margaria and B. Steffen, Eds. Heraclion, Crete: Springer, Oct. 2012, pp. 112–126.

[5] M. Al-Refai, S. Ghosh, and W. Cazzola, “Model-based Regression Test Selection for Validating Runtime Adaptation of Software Systems,” in *Proceedings of the 9th IEEE International Conference on Software Testing, Verification and Validation (ICST’16)*, L. Briand and S. Khurshid, Eds. Chicago, IL, USA: IEEE, 10th-15th of Apr. 2016, pp. 288–298.

[6] S. Yoo and M. Harman, “Regression Testing Minimization, Selection and Prioritization: A Survey,” *Journal of Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, Mar. 2012.

[7] F. Fleurey and A. Solberg, “A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems,” in *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS’09)*, A. Schürr and B. Selic, Eds. Denver, CO, USA: ACM, Oct. 2009, pp. 606–621.

[8] H. K. N. Leung and L. J. White, “Insights into Regression Testing,” in *Proceedings of Conference on Software Maintenance*. Miami, FL, USA: IEEE, Oct. 1989, pp. 60–69.

[9] M. Bergmann, *An Introduction to Many-Valued and Fuzzy Logic: Semantics, Algebras, and Derivation Systems*. Cambridge University Press, 2008.

[10] E. J. Rapos and J. Dingel, “Using Fuzzy Logic and Symbolic Execution to Prioritize UML-RT Test Cases,” in *Proceedings of the 8th International Conference on Software Testing, Verification and Validation (ICST’15)*, G. Fraser and D. Marinov, Eds. Graz, Austria: IEEE, Apr. 2015.

[11] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi, “Regression Test Selection for Java Software,” in *Proceedings of the 16th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA’01)*, J. Vlissides, Ed. Tampa, FL, USa: ACM, Oct. 2001, pp. 312–326.

[12] M. Al-Refai, W. Cazzola, S. Ghosh, and R. France, “Using Models to Validate Unanticipated, Fine-Grained Adaptations at Runtime,” in *Proceedings of the 17th IEEE International Symposium on High Assurance Systems Engineering (HASE’16)*, H. Waeselynck and R. Babiceanu, Eds. Orlando, FL, USA: IEEE, 7th-9th of Jan. 2016, pp. 23–30.

[13] B. Korel, L. H. Tahat, and B. Vaysburg, “Model based regression test reduction using dependence analysis,” in *Software Maintenance, 2002. Proceedings. International Conference on*. IEEE, 2002, pp. 214–223.

[14] H. Ural and H. Yenigün, “Regression test suite selection using dependence analysis,” *Journal of Software: Evolution and Process*, vol. 25, no. 7, pp. 681–709, 2013.

[15] S. Lity, T. Morbach, T. Thüm, and I. Schaefer, “Applying incremental model slicing to product-line regression testing,” in *International Conference on Software Reuse*. Springer, 2016, pp. 3–19.

[16] Z. Xu, K. Gao, T. M. Khoshgoftaar, and N. Seliya, “System Regression Test Planning with a Fuzzy Expert System,” *Information Sciences*, vol. 259, pp. 532–543, 2014.

[17] C. Malz, N. Jazdi, and P. Göhner, “Prioritization of Test Cases Using Software Agents and Fuzzy Logic,” in *Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST’12)*, A. Bertolino and Y. Labiche, Eds. Montreal, BC, Canada: IEEE, Apr. 2012, pp. 483–486.