DISSERTATION

# TOWARDS MODEL-BASED REGRESSION TEST SELECTION

Submitted by

Mohammed Al-Refai

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2019

Doctoral Committee:

Advisor: Sudipto Ghosh
Co-Advisor: Walter Cazzola

James M. Bieman
Indrakshi Ray
Leo Vijayasarathy

ABSTRACT

TOWARDS MODEL-BASED REGRESSION TEST SELECTION

Modern software development processes often use UML models to plan and manage the evolution of software systems. Regression testing is important to ensure that the evolution or adaptation did not break existing functionality. Regression testing can be expensive and is performed with limited resources and under time constraints. Regression test selection (RTS) approaches are used to reduce the cost. RTS is performed by analyzing the changes made to a system at the code or model level.

Existing model-based RTS approaches that use UML models have some limitations. They do not take into account the impact of changes to the inheritance hierarchy of the classes on test case selection. They use behavioral models to perform impact analysis and obtain traceability links between model elements and test cases. However, in practice, structural models such as class diagrams are most commonly used for designing and maintaining applications. Behavioral models are rarely used and even when they are used, they tend to be incomplete and lack fine-grained details needed to obtain the traceability links, which limits the applicability of the existing UML-based RTS approaches.

The goal of this dissertation is to address these limitations and improve the applicability of model-based RTS in practice. To achieve this goal, we proposed a new model-based RTS approach called FLiRTS 2. The development of FLiRTS 2 was driven by our experience accrued from two model-based RTS approaches. The first approach is called MaRTS, which we proposed to incorporate the information related to inheritance hierarchy changes for test case selection. MaRTS is based on UML class and activity diagrams that represent the fine-grained behaviors of a software system and its test cases. The second approach is called FLiRTS, which we proposed to investigate the use of fuzzy logic to enable RTS based on UML sequence and activity diagrams. The activity

diagrams lack fine-grained details needed to obtain the traceability links between models and test cases. MaRTS exploits reverse engineering tools to generate complete, fine-grained diagrams from source code. FLiRTS is based on refining a provided set of abstract activity diagrams to generate fine-grained activity diagrams.

We learned from our experience with MaRTS that performing static analysis on class diagrams enables the identification of test cases that are impacted by changes made to the inheritance hierarchy. Our experience with FLiRTS showed that fuzzy logic can be used to address the uncertainty introduced in the traceability links because of the use of refinements of abstract models. However, it became evident that the applicability of MaRTS and FLiRTS is limited because the process that generates complete behavioral diagrams is expensive, does not scale up to real world projects, and may not always be feasible due to the heterogeneity, complexity, and size of software applications. Therefore, we proposed FLiRTS 2, which extends FLiRTS by dropping the need for using behavioral diagrams and instead relying only on the presence of UML class diagrams. In the absence of behavioral diagrams, fuzzy logic addresses the uncertainty in determining which classes and relationships in the class diagram are actually exercised by the test cases. The generalization and realization relationships in the class diagram are used to identify test cases that are impacted by the changes made to the inheritance hierarchy.

We conducted a large evaluation of FLiRTS 2 and compared its safety, precision, reduction in test suite size, and the fault detection ability of the reduced test suites with that of two code-based RTS approaches that represent the state-of-art for dynamic and static RTS. The results of our empirical studies showed that FLiRTS 2 achieved high safety and reduction in test suite size. The fault detection ability of the reduced test suites was comparable to that achieved by the full test suites. FLiRTS 2 is applicable to a wide range of systems of varying domains and sizes.

# ACKNOWLEDGEMENTS

I express my profound thanks and gratitude to my advisers, Dr. Sudipto Ghosh and Dr. Walter Cazzola, for their invaluable guidance and support. I am very grateful to them for their input on how to improve my research skills and work. They patiently and continuously spent a lot of effort and countless hours to help me with identify and refine research ideas, review my drafts, and provide valuable feedback on my work.

I will forever be grateful to my previous adviser, Dr. Robert France, for his guidance and support, and for introducing me to the modeling community. My appreciation goes to my advisory committee members, Dr. James Bieman, Dr. Indrakshi Ray, and Dr. Leo Vijayasarathy, for taking the time to read my dissertation and evaluating my work. My appreciation also goes to my former committee member Dr. Dan Turk. I thank the entire Computer Science Department staff for always being supportive and for helping me in completing paperwork.

I am indebted to the principal investigators of the Repository for Model Driven Development (ReMoDD), Dr. James Bieman, Dr. Betty Cheng, Dr. Robert France, and Dr. Sudipto Ghosh, for supporting me and giving me the opportunity and honor to be involved in the ReMoDD project.

I would like to express my sincere thanks to my parents, Nayef and Khawla, and my wife, Dania, for their unlimited support and encouragement.

# DEDICATION

*To my family.*

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

# Chapter 1

# Introduction

Regression testing is the process of running the existing test cases on a new version of a system to ensure that the performed modifications do not introduce new faults to previously tested code [2, 3]. Regression testing is one of the most expensive activities performed during the lifecycle of a software system with some studies [4–6] estimating that it can take up to 80% of the testing budget and up to 50% of the software maintenance cost. Regression test selection (RTS) approaches are used to improve regression testing efficiency [3]. 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 [3, 7].

The RTS problem has been studied for over three decades [8, 9]. RTS can be based on the analysis of code-level or model-level changes of a software system. RTS approaches classify test cases as retestable, reusable, or obsolete. Retestable test cases exercise the modified parts of the system and need to be re-executed. Reusable test cases only exercise unmodified parts of the system and do not need to be re-executed. Obsolete test cases are invalid and cannot be executed on the modified version of the software because their input/output relation is no longer correct, or they no longer test what they were designed to test [4]. An RTS approach is *precise* if the test cases that are not affected by the changes are not selected, and *safe* if it guarantees the selection of all the test cases that are affected by the changes.

Traditional code-based RTS approaches take four inputs: the two versions (new and old) of a software system, the original test suite, and dependency information of the test cases on the old version. The outputs are the sets of the retestable and reusable test cases [10]. Code-based RTS assumes that obsolete test cases are already identified and eliminated from the test suite [9].

Existing model-based RTS approaches use behavioral diagrams [11–13] or a combination of structural and behavioral diagrams [7, 14–16]. They also take four inputs: two versions (new and

old) of the models that represent the software system, the original test cases (model-level or code-level test cases), and traceability links from test cases to the models representing the system under test. The outputs are the sets of the retestable and reusable test cases. Some model-based RTS approaches [7, 14] can also identify certain types of obsolete test cases.

## 1.1 Context

France and Rumpe [17] describe two broad types of models used in software development: development models and runtime models. Development models provide abstractions above the code level. Examples of development models are requirements, design, and implementation models [17]. Model-Driven Development [17, 18] is concerned with creating such models to describe a software system and systematically transforming these models to concrete implementations. These models are also used to ease software maintenance and evolution. Dzidek *et al.* [19] showed through empirical studies that using and updating UML documentation during the maintenance and evolution of a software system increased the functional correctness and design quality of the evolved system.

Several approaches exploit class diagrams during the maintenance phase to perform automatic refactorings. Jensen *et al.* [20] propose REMODEL, which uses genetic programming to automatically generate refactoring patterns and to apply them to the class diagram of the application under maintenance. Moghadam *et al.* [21] automatically refactor the source code after design-level changes. The changes in the class diagram are used to identify the structural differences in the application design that are mapped to code-level refactorings and applied to the source code. The systematic literature review of da Silva *et al.* [22] showed that the UML profile-based mechanism is often used to customize a class diagram to properly support the design of context-awareness and self-adaptiveness properties of self-adaptive systems [1, 23–30].

Runtime models present aspects of an executing system at a high level of abstraction, and are used to manage the complexity and to ease the planning process of runtime adaptation [17]. Most

existing model-based adaptation approaches are coarse-grained and focus on using models at runtime to support self-adaptation in autonomous systems [31–36]. Adaptations in these approaches are performed at the component level, and are limited to adding, removing, and reconnecting components of the software system. On the other hand, fine-grained model-based adaptation uses UML diagrams that provide fine-grained views of a system. The *Fine Grained Adaptation* (FiGA) framework [37, 38] uses structural and behavioral UML diagrams to support the planning of unanticipated and fine-grained adaptations on running Java software systems. Vathsavayi *et al.* [39] proposed an adaptation approach that supports changes made to UML class diagrams. The approach uses genetic algorithms to dynamically modify the class diagram in response to the changing environment; the design-level changes are then propagated to the code level using the Javaleon platform [40].

Model-based RTS can be used within the development approaches that already apply model-driven development, model-based evolution, and model-based runtime adaptations of software systems. In these contexts, the evolution or adaptation and test selection processes can be performed at the same level of abstraction using information regarding model-level changes. Yoo *et al.* [9] expect that the use of model-based RTS will grow and have crucial importance in the future because for large systems, model-based approaches can scale up better than code-based RTS approaches. 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 [7]. Regression test selection tools can be programming language-independent, and they can be based on a standard and widely used modeling notation such as UML [7].

## 1.2 Problem Description

The existing model-based RTS approaches suffer from the following limitations:

1. **Lack of support for inheritance hierarchy changes.** These approaches do not take into account the changes to inherited and overridden operations along the inheritance hierarchy,

which can result in missing affected test cases that must be selected. Even a simple change, such as deleting a generalization relationship between two classes, or adding an overriding operation, can impact many classes along the inheritance hierarchy because they affect the inherited and overridden operations.

2. **Incompleteness of behavioral models.** These approaches require their models to be detailed and complete, but in practice, behavioral diagrams tend to be incomplete [41, 42]. Lange *et al.* [42] analyzed UML models in three industrial projects. They found that more than 50% of objects in sequence diagrams are not named, more than 40% of the operations are not called in sequence diagrams, and more than 30% of the classes do not occur as objects in the sequence diagrams. In another study, Lange *et al.* [41] analyzed 14 industrial UML models of different sizes from various organizations found that between 40% and 78% of the operations represented in the class diagrams were not called in the sequence diagrams, and between 35% and 61% of the classes represented in the class diagrams did not occur as objects in the sequence diagrams. Moreover, models are generally created at a high level of abstraction and lack low-level details, such as use- and call-dependencies, which are required by the existing model-based RTS approaches to relate the existing test cases to the models representing the system under test. Briand et al. [7] assumes that the traceability links from test cases to model elements are provided, and requires that each use case is associated with a sequence diagram, and that the sequence diagrams refer to each other using interaction use (identified by the *ref* keyword). Ye *et al.* [11] requires each action node in the activity diagram to represent some function call in the corresponding program. In these approaches, the models simply provide a different representation of the code, and contain enough information to obtain the coverage of test cases at the model level, but obtaining such models is not always easy in practice [7]. This lack of traceability from requirements or design models to test cases is a known issue in model-based RTS, and is likely to limit its role [9].

3. **Unavailabity of behavioral models in practice.** The applicability of the approaches in practice is limited because they need behavioral diagrams but these are available less of-

**Table 1.1:** UML Diagram Type Usage

| UML diagram | Surveys and Projects | | | | | | |
|---|---|---|---|---|---|---|---|
| | **[43, 44]** | **[45]** | **[46]** | **[47]** | **[48]** | **[49]** | **[50]** |
| Class | 73% | 93% | 99% | 85% | 61% | 63% | 100% |
| Activity | 32% | 60% | 47% | 56% | 33% | 54% | 20% |
| Sequence | 50% | 89% | 94% | 35% | 41% | 54% | 20% |
| State Machine | 29% | 63% | 91% | 22% | 8% | 27% | 0% |

ten than class diagrams. Researchers have studied the use of UML diagrams in real-world software development through developer surveys [43–49] and by evaluating open source projects [50]. Table 1.1 summarizes the results of these studies. The values in the table were calculated as the ratio of the number of survey respondents (or projects) that use a specific diagram type to the total number of the respondents (or projects). The values in a column do not add up to 100% because a respondent (or a project) may use multiple diagram types. The behavioral diagrams are clearly used less often than class diagrams. The difference in the usage frequency between the class diagram and the three diagram types that are used in the existing model-based RTS approaches (activity, sequence, and state diagrams) is more than 20% in 5 studies. In the other 3 studies, the difference ranges between 4% and 52% depending on the diagram type.

There is a need to address these limitations and improve the applicability of model-based RTS to model-based software development projects. Because the behavioral diagrams are used less often than class diagrams, there is a need for an RTS approach that only uses class diagrams.

## 1.3 Approach and Contributions

In this dissertation, we propose a new model-based RTS approach called FLiRTS 2 that addresses the three limitations. The development of FLiRTS 2 is driven by our experience accrued from two model-based RTS approaches called MaRTS and FLiRTS.

MaRTS is a **M**odel-based **R**egression **T**est **S**election approach that addresses the first limitation regarding the inheritance hierarchy changes. MaRTS reverse engineers the source code to generate fine-grained activity diagrams. FLiRTS is a **F**uzzy **L**ogic-based **R**egression **T**est **S**election approach that refines the provided abstract activity diagrams to generate fine-grained activity diagrams. In both cases the fine-grained activity diagrams are used to obtain the traceability links from the test cases to the model elements. MaRTS uses static analysis on class diagrams to enable the identification of the test cases that are impacted by changes made to the inheritance hierarchy. FLiRTS uses fuzzy logic to address the uncertainty introduced in the correctness of the traceability links obtained using the generated refinements, which have varying probabilities of being correct.

However, it was evident that the applicability of model-based RTS is limited because the process that generates fine-grained behavioral diagrams is expensive, does not scale to real world projects, and may not always be feasible due to the heterogeneity, complexity, and size of software applications. Therefore, we proposed FLiRTS 2, which extends FLiRTS by dropping the need for behavioral diagrams instead and only using class diagrams, which is the most commonly provided diagram type in practice as shown in Table 1.1. The class diagram represents the design of the system under test and its test classes. Langer *et al.* [51] showed that class diagrams used in practice contain classes and interfaces, operation signatures and return types, generalization and realization relationships, and associations. FLiRTS 2 does not use the call usage dependency relationships between classes because this relationship type is not commonly provided. The lack of behavioral diagrams and call usage dependencies create uncertainty about which relationships and classes in the class diagram are actually traversed by test cases during the execution. FLiRTS 2 addresses this uncertainty by using fuzzy logic while classifying the test cases as reusable or retestable.

FLiRTS 2 addresses all the three limitations. It uses the generalization and realization relationships in the class diagram to identify test cases that are impacted by the changes to the inheritance hierarchy. It relies only on class diagrams to address the second and third limitations.

We developed a prototype tool for FLiRTS 2. The tool accepts UML class diagrams produced using IBM Rational Software Architect [52] (file type is EMX) and Eclipse Modeling Frame-

work [53] (file type is UML). The tool also supports class diagrams that are customized using the UML-profile mechanism for a particular domain. We conducted a large evaluation of FLiRTS 2 on 8060 revisions of 21 open source projects. We compared the safety and precision of FLiRTS 2 with that of two code-based RTS approaches, Ekstazi [10] and STARTS [54], that represent the state-of-art for dynamic and static RTS respectively. We also evaluated the reduction in the number of test cases selected by FLiRTS 2, and used mutation testing to evaluate the fault detection ability of the selected test cases.

The contributions of the dissertation are the FLiRTS 2 approach and its evaluation. To the best of our knowledge, this is the largest evaluation of model-based RTS in terms of the number of used subjects and their revisions, and the first evaluation that compares model-based and code-based RTS approaches.

The rest of the dissertation is organized as follows. Chapter 2 describes related work on code-based and model-based regression test selection approaches, and fuzzy logic-based test selection and prioritization approaches. Chapter 3 provides background information on the FiGA framework and fuzzy logic. Chapter 4 presents MaRTS and its evaluation. Chapter 5 describes FLiRTS and its pilot case study. Chapter 6 presents FLiRTS 2 and its evaluation. The conclusions and directions for future work are outlined in Chapter 7.

# Chapter 2

# Related Work

Regression test selection has been studied for over three decades [8, 9]. This section discusses related work for model-based and code-based RTS approaches including the approaches that use fuzzy logic to select test cases.

In general, RTS approaches classify test cases into three categories (obsolete, retestable, and reusable) defined by Leung and White [4]. Obsolete test cases are invalid and cannot be executed on the modified version of the software because their input/output relation is no longer correct, or they no longer test what they were designed to test [9]. Retestable test cases are still valid, and exercise the modified parts of the software. These tests need to be re-executed for regression testing to be safe. Reusable test cases only exercise unmodified parts of the program, and thus, while they are still valid, they do not need to be re-executed to ensure safe regression testing.

A safe RTS technique must select all *modification-traversing* test cases for regression testing [55]. A test case is considered to be modification-traversing for a program $P$ if it executes changed code in $P$, or if it formerly executed code that was deleted in $P$ [9]. A safe RTS approach is not considered to be safe from all possible faults because some program changes might cause side effects on other unmodified parts of the program. An RTS approach is considered safe in the sense that if there exists a test case that traverses modified code, then it will definitely be selected for regression testing [9].

Rothermel and Harrold [55] identified two criteria to evaluate regression test selection approaches: inclusiveness and precision. We define these criteria here because we will use them to evaluate our proposed RTS approach. Inclusiveness measures the extent to which an RTS approach selects test cases that traverse modified code for regression testing. Suppose a test suite contains $T$ test cases, such that $N$ test cases among $T$ are modification-traversing for $P$ and $P'$, and suppose that the RTS approach selects $M$ of these $N$ test cases for regression testing, then the inclusiveness

of the RTS approach with respect to $P$, $P'$, and $T$ is $M/N$ [55]. The RTS approach is considered to be safe if for all $P$, $P'$, and $T$, the inclusiveness of the RTS approach is 100%. For example, if $T$ is 50 and $N$ is 40, and the RTS approach selects 30 of the 40 test cases, then the inclusiveness is 0.75 (=30/40).

Precision measures the extent to which an RTS approach omits test cases that are not modification-traversing because they are also not fault-revealing test cases [55]. Suppose a test suite contains $T$ test cases, and $K$ test cases among $T$ are not modification-traversing for $P$ and $P'$. If the RTS approach omits $O$ of these $K$ test cases, then the precision of the RTS approach with respect to $P$, $P'$, and $T$ is $O/K$ [55]. For example, if $T$ is 50 and $K$ is 40, and the RTS approach omits 30 of the 40 test cases, then the precision is 0.75 (=30/40).

## 2.1 Model-based RTS approaches

Model-based RTS classifies test cases according to changes made to the models that represent the system under test [7]. Most of the existing approaches [7, 11–16] are based on UML models, and few approaches [56, 57] use other types of models. The UML model-based approaches use activity, class, sequence, state chart, and use case diagrams.

Chen *et al.* [12] use UML activity models for specification-based RTS. An activity model represents the requirements and specifications of a system. Changes to the specifications result in modifications to the activity model, and these specification-based changes at the model level drive the selection of test cases.

Briand *et al.* [7] present an RTS approach based on UML design models that include use case, class, and sequence diagrams. This approach identifies changes in the three types of models and the impact these changes have on the test cases. Functional testing is supported where each test case triggers operations belonging to interface classes. Such operations are represented as use cases in the use case model, and each use case is connected to sequence models that represent the interaction scenarios of that use case.

Korel *et al.* [13] use control and data dependencies in an extended finite state machine to identify the impact of model changes. Ural *et al.* [58], and Lity *et al.* [59] proposed model-based RTS approaches based on state machine diagrams.

Farooq *et al.* [14] use UML class and state machine models. This approach identifies changes in the class model and in the state diagrams, and uses the impacted and changed elements of the state diagrams to classify test cases.

Zech *et al.* [15, 16] present a generic model-based RTS platform, which is based on the model versioning tool MoVE. The approach consists of three phases: change identification, impact analysis, and test case selection, which are controlled by OCL queries. This tool was demonstrated on a small program as a proof of concept, and the results showed that the OCL queries for impact analysis are complex even for simple rules.

Ye *et al.* [11] select test cases based on changes performed to UML activity diagrams. This approach requires instrumenting the software system to record the execution traces of code-level test cases, and requires each action node in the activity diagram to represent some function call in the corresponding program. By building the correlations between the actions in the diagram and functions in the program, this approach can obtain the corresponding path of each test case in the activity diagram according to the execution trace of the test case.

Muccini *et al.* [56, 57] use component-based models to represent the architecture of a system and labeled transition systems (LTS) to represent the component behaviors. The LTS are combined with the structural information to build graphs that are used to perform RTS.

In general, the use of model-based RTS techniques is growing, and will have crucial importance in the future because (1) model-based approaches can be more efficient, and can scale up better than code-based approaches for large software systems [7, 9], and (2) it is easier to analyze the changes between different versions of models that represent a system at a high level of abstraction compared to the changes between different code versions [7]. Regression test selection tools can be programming language-independent and based on standard and widely used modeling notations

such as UML [7]. For the approaches that already use models to apply evolution and runtime adaptation, applying model-based RTS techniques is likely to be more convenient than applying code-based RTS techniques as both the adaptation and RTS processes can be performed at the same level of abstraction.

## 2.2 Code-based RTS approaches

Code-based RTS depends on (1) identifying changes made to the source code, (2) performing static or dynamic dependency analysis to compute dependencies from test cases to the system under test, and (3) selecting test cases that traverse modified code. Code changes can be identified at different levels of granularity such as class, method, or statement. Similarly, dependencies can be computed from test cases to classes, methods, or statements.

Engström *et al.* [8] categorize the code-based approaches into three major groups: firewall-based group, graph-walk-based group, and dependency-based group. The last group can be considered to be a subgroup of the firewall-based group because firewall-based RTS is based on dependencies between program entities such as classes.

Firewall approaches [9, 60, 61] are based on the concept of specifying a firewall around the entities of the system that need to be retested. A firewall is a conceptual boundary that contains the set of all modified and affected entities in a program. Kung *et al.* [60] apply RTS at the class level. This approach constructs an object relation diagram (ORD) that describes static relationships among classes. The approach reports the classes executed by each test case. The firewall of a class *C* is defined as a set that contains *C* and all the classes that are dependent on *C*. When *C* is modified, then all the test cases traversing any of the classes in its firewall are selected. Hsia *et al.* [62], and White and Abdullah [63] apply firewall-based RTS at the class level. Jang *et al.* [61] apply firewall-based RTS at the method level to C++ software. They identify firewalls around all the methods affected by a change and select all the test cases exercising these methods for regression testing.

Skoglund and Runeson [64] found in empirical studies that the class-level firewall RTS is not a precise technique, i.e., it can select test cases that are not modification-traversing. They proposed an improved approach over the class-level firewall approach by removing the class firewall and using a change-based RTS approach that only selects those test cases that exercise the changed classes instead of all the classes in the firewall.

Soetens *et al.* [65] proposed a firewall-based RTS approach based on the FAMIX model. Changes made to Java software are identified as change objects in the FAMIX model. Changes that can be identified in the model are additions, removals, and modifications of packages, classes, methods, attributes, and method invocation statements. Dependencies between software entities are defined in the model and these dependencies are exploited for test selection. Each identified change is mapped to its set of relevant test cases based on the change dependence hierarchy defined in the model. The model in this approach assumes that there is a one-to-one relationship between a method invocation statement and the callee method. This approach cannot classify test cases that traverse changed constructors.

Soetens *et al.* [66] extended their approach to support dynamic binding when performing test selection. They extended the FAMIX model to include dependencies from a method invocation to all the methods that this invocation can refer to. This technique to specify such dependencies is static and is based on method names. For example, if the program contains a method invocation $obj.m()$, then the model will have dependencies from this invocation to all the methods in the program that carry the same name $m$.

Many graph-walk approaches address the problem of RTS. Rothermel and Harrold [67] propose a safe approach for RTS for procedural programs. The algorithm uses control-flow graphs (CFG) to represent each procedure in a program *P* and its modified version *P'*. Each node in a CFG represents a simple or conditional statement, and each edge represents the flow of control between statements. Entities affected by modifications are selected by traversing in parallel the CFGs of *P* and *P'*, and when the target entities of like-labeled CFG edges in *P* and *P'* differ, the edge is added

to the set of affected entities. The graph-walk approaches are known to be safe when they apply RTS at the edge level (i.e., select test cases that traverse affected edges in the graph).

Rothermel and Harrold extended the CFG-based algorithm for C++ using Inter-procedural Control-Flow Graphs (ICFG) [68]. The analysis algorithm in this approach works only for complete programs. When classes interact with other classes, the called classes must be fully analyzed. The analysis algorithm cannot be applied to programs that call external libraries, unless the code of the libraries is available.

Harrold *et al.* [69] extended the CFG approach for Java software using the Java Inter-class Graph (JIG) as a representation that handles interprocedural interactions through calls to methods. Each method call statement is represented as a call node in the JIG. A call edge connects each call node to the entry node of the called method. If the call is virtual, the call node is connected to the entry node of each method that can be bound to the call. For example, each edge from a call node to the entry node of a method *C.m* is labeled with the type of the receiver that causes *C.m* to be bound to the call. The class hierarchy analysis technique [70] is used to resolve all possible virtual call bindings. This representation supports the identification of which method calls are affected by comparing the JIGs constructed from the original and modified programs.

Vokolos and Frankl [71] consider RTS based on text differencing using the Unix `diff` command. The approach compares the original code version with the modified version to identify the modified statements, and selects test cases that exercise code blocks containing these statements.

The current trend [10, 54, 72–75] is to focus on class-level RTS by (1) identifying changes at the class level and (2) computing depedencies from test cases to the classes under test. This is because class-level RTS can be more efficient than identifying changes and computing depedencies at the method or statement level. RTS approaches [10,54,73] were recently proposed to make RTS more cost-effective in modern software systems. In addition to supporting class-level RTS, these approaches consider a test class as a test case, and thus, select test classes instead of test methods. Gligoric *et al.* [10] proposed Ekstazi, an approach that tracks dynamic dependencies of test cases at

the class level and selects test cases that traverse modified classes. Ekstazi is a safe RTS approach, and its safety is based on the formally proven safety of the change-based RTS approach [64].

Legunsen *et al.* [54, 72] proposed STARTS, which is a static RTS approach that is based on the idea of the class-level firewall. STARTS builds a dependency graph of program types based on compile-time information, and selects test cases that can reach changed types in the transitive closure of the dependency graph.

Zhang [73] proposed HyRTS, which is a dynamic and hybrid approach that supports analyzing the adapted classes at multiple granularity levels (i.e., method and file levels) to improve the precision and selection time.

Yu *et al.* [74] stated that traditional RTS approaches that use fine-grained dependency analysis from test cases to statements, basic blocks, or methods, are difficult to apply in continuous integration (CI) systems because they require significant analysis time and cannot handle rapid changes in CI environments. The authors implemented two static RTS approaches at different granularity levels and applied them within CI environments on 37 Java projects, and compared their cost effectiveness to *ReTestAll*, i.e., executing all the test cases. The results showed that on 97.3% of the Java projects, the RTS approach that supports dependency analysis from test cases to methods performed much more poorly than ReTestAll, and the RTS approach that supports dependency analysis from test cases to classes saved more time than ReTestAll for more than half of the projects.

Gyori *et al.* [75] compared different variants of dynamic and static class-level RTS with project-level RTS in the Maven Central open source ecosystem. An ecosystem may contain a large number interconnected projects, where client projects transitively depend on library projects. Project-level RTS identifies changes at the project level and computes dependencies from test cases to projects. When a library changes, then all test cases in the library and all test cases in all the library's transitive clients are selected. Class-level RTS identifies changes at the class level and computes dependencies from test cases to the classes. The results showed that class-level RTS can be an order of magnitude less costly than project-level RTS in terms of test suite reduction.

Other recent RTS approaches [76, 77] were proposed to address the scenario when the regression testing budget is limited, and the test suite reduction is more important than safety and fault detection ability.

Romano *et al.* [76] proposed an RTS approach called SPIRITuS, which uses information retrieval to select test cases. The main design goals of SPIRITuS are to be easy to adapt to different programming languages, and to be tunable via a threshold. SPIRITuS represents all the methods of the classes under test as documents via the Vector Space Model (VSM). To decide if a method in an adapted program must be retested or not, the lexical similarity between the method versions in the original and adapted programs is computed using the cosine similarity measure. If the similarity is below a given threshold, then all the test cases exercising the method are selected. SPIRITuS was evaluated and compared with Ekstazi and a diff-based RTS technique in terms of reduction in test suite size and fault detection ability. The diff-based technique selects the test cases that cover changed methods, where differences are computed via the UNIX `diff` tool. The results showed that SPIRITuS selects significantly fewer test cases than Ekstazi and the diff-based technique but detects fewer faults than them.

Azizi *et al.* [77] eliminated the need for dynamic and static analysis. They proposed ReTEST, a language-independent RTS approach that facilitates a lightweight analysis by using information retrieval. ReTEST tokenizes the source code of the test cases to compute their diversity. ReTEST selects test cases based on test failure history, test case diversity, and code change history at the line level.

## 2.3 Fuzzy logic-based RTS approaches

There are model-based and code-based approaches that use fuzzy logic to select or prioritize test cases. Regression test prioritization concerns the identification of the ideal ordering of test cases that maximizes desirable properties, such as early fault detection [9]. Each approach treats test case selection or prioritization as a decision-making problem, and uses a fuzzy logic system to

select or prioritize test cases based on multiple inputs regarding the test cases and the system under test, such as test coverage, failure history, and system change history.

**Code-based Approches.** Xu *et al.* [78] propose a code-based RTS approach that is based on fuzzy expert systems to select test cases when source code and its change history are not available. The fuzzy expert systems select relevant test cases by using fuzzy logic to correlate knowledge represented by one or more of the following: customer profile, analysis of test case coverage and results, system failure rate, and change in system architecture. This approach assumes that this knowledge is available, and can be used to provide inputs to the fuzzy expert system. The output of the fuzzy expert systems aid the system testers in their decision-making process to select the most relevant test cases.

Malz *et al.* [79] propose a code-based approach that uses 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 cooperation between the software agents. This approach assumes that the test coverage is provided, and uses this information as one of the priority values to prioritize test cases.

**Model-based Approches.** To the best of our knowledge, there are no model-based approaches that use fuzzy logic to perform RTS. The existing model-based approaches use fuzzy logic for test case prioritization, and they only rely on behavioral diagrams.

Rapos *et al.* [80] propose a model-based approach that uses fuzzy logic to prioritize test cases based on information available from the symbolic execution tree for 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 numerical output called priority for each test case. The test cases are prioritized based on these outputs.

Rhmann *et al.* [81] propose an approach that prioritizes test cases using fuzzy logic based on changes performed to state machine diagram. A state machine diagram is used to represent the behavior of the system. When the system is modified, the diagram is changed to represent the

modified behavior of the system. Then, the state machine diagram is converted into a weighted extended finite state machine, where weights are assigned to nodes and edges based on change information and risk exposure analysis. Then, test paths are generated from the modified state diagram. Finally, risk exposure and change information of each generated test path are calculated and used as inputs to the fuzzy logic system to prioritize the test paths.

# Chapter 3

# Background

In this chapter, we provide background information on the FiGA framework and fuzzy logic. FiGA supports model-based adaptations at runtime and MaRTS was developed to work within the context of FiGA. Fuzzy logic was used in FLiRTS and FLiRTS 2 to address the uncertainty of test coverage at the model level when classifying test cases.

## 3.1 FiGA Framework

Cazzola et al. [37, 38] proposed a *Fine-Grained Adaptation* (FiGA) framework that supports model-based adaptation of Java programs at runtime. FiGA is based on the idea that models can be used to present abstractions of a software system that can be changed by a developer, and these abstractions ease system adaptation by minimizing the interventions performed directly on the code. FiGA uses class and activity models to describe the aspects of the runtime structure and behavior that can be modified by a developer at runtime. Changes to the models are propagated to changes in the running system.

FiGA uses (1) ReverseЯ [82], which is used to generate models from a running Java program, and (2) JavAdaptor [83], which can update a Java program during its execution without stopping it. ReverseЯ is a tool that is based on @Java annotations [84]. @Java extends the Java annotation framework with one that supports (1) custom types and runtime-bound values in annotations, and (2) a finer-grained annotation model that allows annotations on blocks and expressions. FiGA does not apply static analysis of the code to generate models. Instead, ReverseЯ defines a set of @Java annotations that can be used for generating models from running code. This allows ReverseЯ to represent runtime information in the models, i.e., sequence and activity models. Developers annotate a Java program with such annotations during development time. At runtime, these anno-

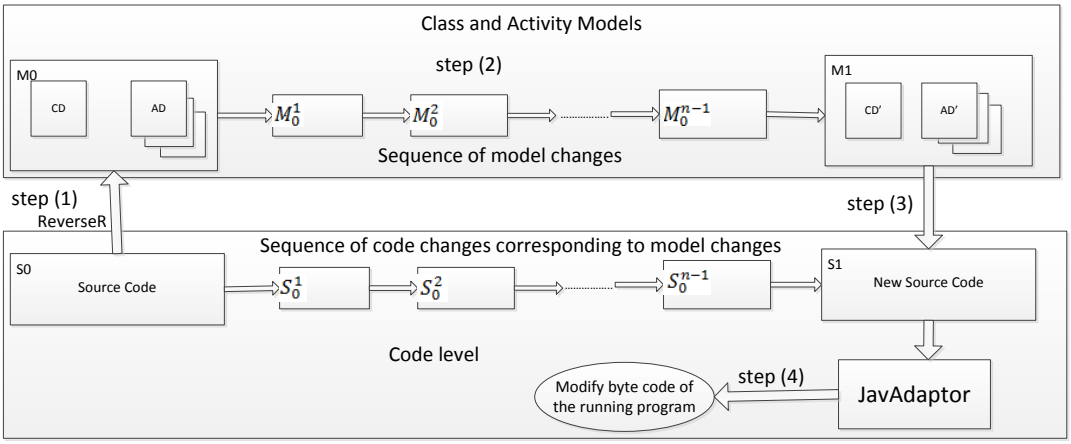



**Figure 3.1:** Overview of the FiGA Approach.

tations drive the model extraction from the running Java program. ReverseЯ takes as input a set of compiled classes and produces an IBM Rational Software Architect model file.

JavAdaptor works at a low level, requiring a compiled version of the class to be updated as an input and a connection to the Java virtual machine in which the program is executing. Changes to the Java source code drive the application update. The FiGA framework extends JavAdaptor by using UML models instead of the source code to drive the updating process. As shown in Fig. 3.1 the FiGA framework supports the adaptation of a running program through a repetitive loop composed of five steps that can be used whenever the application needs to be updated.

1. **Generating Models from the Source Code.** ReverseЯ [82] is used to generate the model $M_0$ consisting of class and activity models (CD and AD) from the running source code by reverse engineering annotated source code ($S_0$). Each individual method is represented as an activity model.

2. **Adapting the Models.** A developer changes the class and activity models to apply the needed adaptation. Model changes are expressed as a sequence of elementary operations, where applying an elementary operation to a model $M_0^{i-1}$ produces a modified model version

$M_0^i$. Fig. 3.1 shows the sequence of modified model versions, $M_0^1$, $M_0^2$, ... $M_1$. $M_1$ is the last modified version before changes are propagated from the model to the running application.

3. **Propagating changes to the source code.** The changes performed to the diagrams are mapped to corresponding code changes via the application of some predefined mapping functions. An operation applied to a model $M_0^{i-1}$ to produce $M_0^i$ is mapped to a corresponding code change that is applied to the corresponding source code $S_0^{i-1}$ to produce $S_0^i$. Fig. 3.1 shows the sequence of modified versions of the source code, $S_0^1$, $S_0^2$, ... $S_0^{n-1}$, to get from $S_0$ to $S_1$. Complete details of the adaptation semantics and the mapping functions are given in Cazzola *et al.* [37, 38]. Model changes are propagated to the code of the application only when all required model changes are performed. The sequence of model changes is determined by model differencing between $M_0$ and $M_1$, and mapped into calls to code change operators with the proper parameters.

4. **Updating the running application.** In this step, JavAdaptor deploys the changes to the running application without stopping it. The state of the updated running application is preserved. Details on how JavAdaptor preserves a running program state can be found in Pukall *et al.* [83].

The activity diagrams generated via ReverseЯ during step (1) provide fine-grained descriptions of method bodies, which makes them suitable for performing fine-grained adaptations of method behaviors. Each method of a class is represented as an activity diagram where: (1) each activity diagram has a single initial node and a single final node, and (2) there is no flow termination except for the final node. The activity diagram elements that are supported in FiGA are action nodes, call behavior nodes, decision and merge nodes, and initial and final nodes, and transition flows that are used to connect these nodes [37, 38].

The activity diagrams generated using ReverseЯ are detailed because they contain code statements associated with their action nodes and transition flows. If Java statements are annotated with the `@CallAction` annotation, then ReverseЯ creates an action node and adds the Java

statements to the code snippet of the action node. A Java conditional statement that is annotated with the `@Decision` annotation is mapped to a decision node with outgoing transition flows. The boolean expression of the conditional statement and the negation of the expression are represented as code snippets of the outgoing transition flows. If developers think that visualizing branches of a conditional statement in the model is not important, then they can annotate the statement with the `@CallAction`. In this case, the entire conditional statement and its body are copied to a code snippet of an action node. Additionally, ReverseЯ maps method call statements that are annotated with `@CallBehavior` to calls to activity diagrams representing the called methods, where the call statement to an activity diagram is added to a code snippet of an action node [85].

The activity diagrams generated by ReverseЯ are compliant with the Rational Software Architect (RSA) modeling tool [52]. These diagrams can be executed using the RSA simulation toolkit 9.0[1]. When the model execution flow reaches an action node, the code snippet associated with the node is executed. When the model execution flow reaches a decision node, its outgoing transition flows are evaluated and the flow that evaluates to `True` is executed.

## 3.2 Fuzzy Logic

Fuzzy logic is a many-valued logic in which the truth values are not limited to *true* and *false* but may range between *completely true* and *completely false* [86]. This type of logic provides reasoning mechanisms to deal with uncertainty. We demonstrate the application of fuzzy logic to address uncertainty using a restaurant tipping system that we adapted from [87]. This system is controlled by fuzzy logic to model how to choose a tip at a restaurant based on the food quality and provided service, rated between 0 and 10, where the tip value can be between 0% and 30% of the total billing amount.

Fuzzy logic is commonly associated with reasoning using natural languages. For example, the phrase "if the provided service is good then pay generous tip" contains uncertainty because the

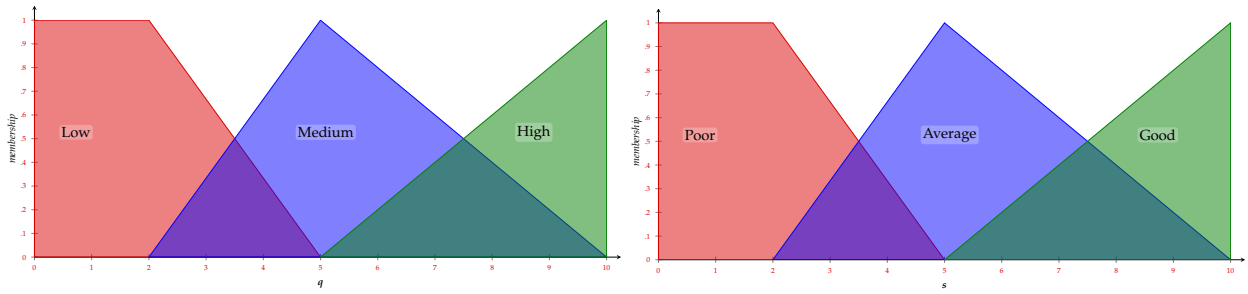
[1] http://www-03.ibm.com/software/products/en/ratisoftarchsimutool

meaning of "good service" and "generous tip" are unclear. Does "generous tip" mean a tip value between 20% and 30% of the total billing amount? Does "good service" refer to a specific rating value for service? These terms can be understood in different ways. Fuzzy logic is used to deal with this type of reasoning.

A fuzzy logic system is a control system that is based on fuzzy logic, and in concept, is much closer to reasoning using natural language than traditional control systems that are based on accurate mathematical models to define the system responses to its inputs [88]. Fuzzy logic systems can be used to address uncertainty in control systems that do not have accurate mathematical models. A fuzzy logic system is based on dividing the input domain of a controlled system into pre-defined categories in order to map inputs into each of these categories with a certain degree of truth, and constructing linguistic rules that are applied to reason about the inputs and make decisions towards an output [88].

The process for fuzzy logic systems consists of an input phase known as *fuzzification*, a processing phase known as *inference*, and an output phase known as *defuzzification* [88, 89]. We demonstrate the three phases of a fuzzy logic system using the restaurant tipping system example.

**Fuzzification phase.** The inputs to a fuzzy logic system are called *input crisp variable*s, which take discrete values called *input crisp value*s. For example, in the restaurant tipping system that is managed by fuzzy logic, an input crisp variable is the provided service, and an input crisp value can be a rating value between 0 and 10. An input crisp variable is fuzzy and its input crisp values can carry considerable amount of uncertainty. Therefore, the input domain of an input variable is divided into pre-defined categories known as input *fuzzy set*s. These fuzzy sets can overlap. An input value of the variable fits in each of these input fuzzy sets with a certain degree of truth, which needs to be modeled. Modeling requires assigning each fuzzy set a membership function that defines how each value within the input space of the fuzzy set is mapped to a membership value between 0 and 1 [88, 90]. The input crisp variables for the restaurant tipping system are **q** for food quality and **s** for the provided service. As shown in Figs. 3.2a and 3.2b, the input domains of **q** and **s** are divided into input fuzzy sets where the *Low* and *Poor* fuzzy sets have trapezoidal membership




**(a)** Input fuzzy sets for **q** **(b)** Input fuzzy sets for **s**

**Figure 3.2:** Input Fuzzy Sets.

functions and the other sets have triangular membership functions. One or more *output variable*s and their *output fuzzy set*s can also be defined for a fuzzy logic system. The output variable for the tipping system is **t** (for tip). Fig. 3.3 shows the output fuzzy sets for **t** and the triangular membership functions we defined for these sets. In general, the triangular and trapezoidal membership functions are the most commonly used function types for fuzzy sets because they produce good results for most of the domains [91, 92]. There are other types of membership functions such as the gaussian and sigmoidal functions.

The fuzzification step is needed to fuzzify the input crisp values of the input crisp variables. In particular, this step maps each input crisp value to a set of *input fuzzy value*s using the pre-defined input fuzzy sets. Each input fuzzy value represents the membership degree of the input crisp value in one of the input fuzzy sets [88, 89]. In the restaurant tipping example, suppose that the input crisp value for **q** is 7. The membership value of 7 is zero in the *Low* set, 0.6 in the *Medium* set, and 0.4 in the *High* set. Suppose that the input crisp value for **s** is 6. The membership value of 6 is zero in the *Poor* set, 0.8 in the *Average* set, and 0.2 in the *Good* set. Step (1) of Fig. 3.4 shows the fuzzification step for the input crisp values 7 and 6.

**Inference phase.** This phase determines the meaning of a set of input fuzzy values produced in the fuzzification phase based on a pre-defined set of inference rules. These rules are needed to model the relations between the inputs and outputs of a fuzzy logic system, enable reasoning about the fuzzy inputs, and make decisions towards what action to take. The inference rules for a fuzzy

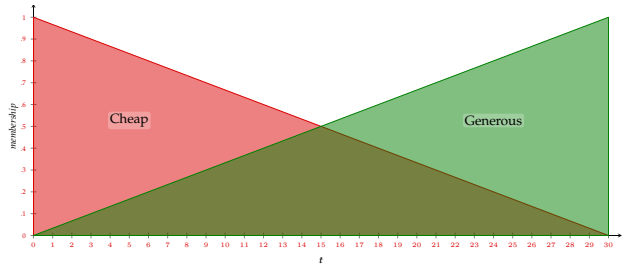



**Figure 3.3:** Output fuzzy sets for **t**.

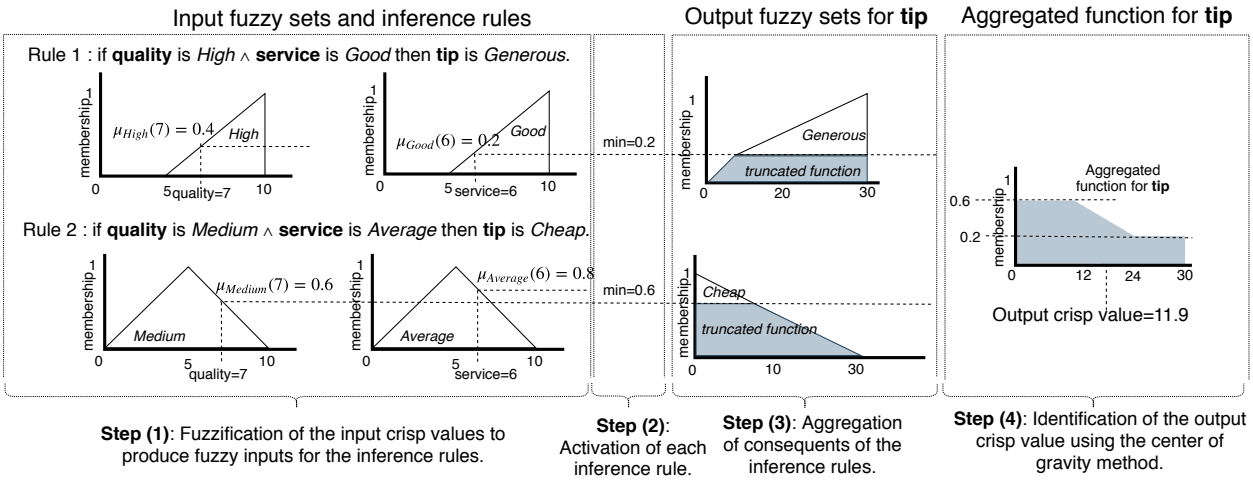



**Figure 3.4:** Fuzzification, Inference, and Defuzzification.

logic system are specified in natural language in the form *"if antecedent then consequent"*, where the *antecedent* can be any number of logical statements. The antecedent of a rule contains one or more input crisp variables and their input fuzzy sets.

There are several methods to evaluate the inference rules and aggregate their results, such as Mamdani [90, 93] and Sugeno [90]. In Mamdani, the output fuzzy sets with membership functions are defined for the output variable of the fuzzy logic system, and these sets are used in the consequents of the inference rules. The Sugeno method is similar to the Mamdani method. The fuzzification of the input crisp values and the activation of the inference rules (step (1) and step (2) of Fig. 3.4) are done in the same way. The primary difference is that in Sugeno there are no output fuzzy sets defined for the output variable. Instead, the consequent of each inference rule is a polynomial function that produces a single crisp output for the output variable.

Mamdani is the most commonly used method in practice and in the literature [90] because it is more intuitive, works better with human input, and is well-suited to handle fuzziness and data uncertainty by using membership functions in the consequents of the inference rules [94, 95]. The main criticisms against the Sugeno method is that the expressiveness and interpretability properties of the Mamdani method are lost because the consequents of the rules are not fuzzy sets [95]. The simplification of the consequents with polynomial functions can lead to loss of the linguistic meaning of membership [94]. Therefore, we use the Mamdani method in the restaurant tipping system. The output fuzzy sets *Cheap* and *Generous* are used in the consequents of the inference rules. An example of an inference rule is:

`if` **q** `is` *Low* $\wedge$ **s** `is` *Poor* `then` **t** `is` *Cheap*.

The antecedent of an inference rule can have multiple inputs in conjunctive canonical form (connected by the $\wedge$ operator) or in disjunctive canonical form (connected by the $\vee$ operator). Table 3.1 summarizes all the inference rules that we defined for the system.

Each inference rule is evaluated as follows. First, the rule is activated by calculating the minimum of the fuzzy inputs of the input crisp values used in the rule if the rule's antecedent is in conjunctive canonical form, or by calculating the maximum of the fuzzy inputs if the rule's antecedent is in disjunctive canonical form. Second, the calculated minimum (or maximum) value propagates to the output fuzzy set used in the rule by truncating its membership function [88, 90]. For example, in Fig. 3.4, the input crisp values used in *rule 1* are 7 and 6, and their fuzzy inputs are 0.4 ($\boldsymbol{\mu_{High}}$**(7)=0.4**) and 0.2 ($\boldsymbol{\mu_{Good}}$**(6)=0.2**), respectively. *Rule 1* is activated by applying the minimum operation between 0.4 and 0.2 and the result is 0.2 as shown in step (2) of Fig. 3.4. Table 3.2 shows the results of activating each of the 9 rules of Table 3.1 using the minimum operation. Thereafter, the minimum value calculated in each rule propagates to the output fuzzy set (i.e., *Cheap* or *Generous*) used in the rule by truncating its respective membership function. The truncated functions are shown in step (3) of Fig. 3.4 as the shaded areas of the membership functions of *Cheap* and *Generous*.

**Table 3.1:** Inference Rules

| s \ q | Low | Medium | High |
|---|---|---|---|
| **Poor** | Cheap | Cheap | Cheap |
| **Average** | Cheap | Cheap | Cheap |
| **Good** | Cheap | Generous | Generous |

**Table 3.2:** Activation of Inference Rules

| s \ q | $\mu_{Low}(7)$=0 | $\mu_{Medium}(7)$=0.6 | $\mu_{High}(7)$=0.4 |
|---|---|---|---|
| $\mu_{Poor}(6)$=0 | 0 | 0 | 0 |
| $\mu_{Average}(6)$=0.8 | 0 | 0.6 | 0.4 |
| $\mu_{Good}(6)$=0.2 | 0 | 0.2 | 0.2 |

After the evaluation of all the inference rules, their outputs need to be combined [88, 90]. The process of obtaining the overall consequent from the individual consequents of multiple rules is known as aggregation of rules [90]. The fuzzy union operator [90] (also called max operator) is applied to the truncated membership functions of the rules to produce an aggregated membership function. In the example, the union operator is applied to the truncated functions shown in step (3) of Fig. 3.4 to produce the aggregated membership function shown in Step (4) of Fig. 3.4. This aggregated function for **t** represents the final output of the inference phase.

**Defuzzification phase.** This phase is needed to reduce the aggregated membership function produced by the inference process into a single *output crisp value* to derive a final decision for the fuzzy logic system [88, 89]. There are multiple defuzzification methods such as the centroid, rightmost-max, and leftmost-max [90]. The centroid method [87,90] (also called the center of gravity method) is the most commonly used defuzzification method [90]. It finds the output crisp point $x$ where a vertical line from $x$ would slice the fuzzy set covered by the aggregated membership function into two equal masses. In the example, the centroid method is used with the aggregated membership function of **t** to find the output crisp value. The defuzzification process returns 11.9 as the output crisp value for the **t** variable as shown in Step (4) of Fig. 3.4. We also applied the other defuzzification methods, and the produced outputs were 12.0 for rightmost-max, and 0 for leftmost-max. These two methods find the left most and right most output crisp points that have the highest membership values in the aggregated membership function. Unlike the centroid method,

leftmost-max and rightmost-max do not consider the whole area of the fuzzy set covered by the aggregated membership function. Therefore, these two methods can output extreme values such as the value 0 produced by the leftmost-max in this example. The output values 11.9 and 12 are more reasonable than the output value 0 because the inputs for **q** and **s** are higher than 5 out of 10.

The output crisp value can be interpreted by mapping it through the membership functions of the *Cheap* and *Generous* fuzzy sets to probabilities for being cheap and generous. The membership value of 11.9 is 0.40 in the *Generous* set and 0.60 in the *Cheap* set. If a membership value exceeds a predefined threshold, then the output set that produced the value becomes the final decision of the fuzzy logic system. In the tipping example, if we define 50% as the threshold, then the final decision of the fuzzy logic system is that the tip is cheap because the membership value of 11.9 in the *Cheap* set exceeded the threshold.

# Chapter 4

# Model-based RTS Using UML Class and Activity Diagrams

This chapter presents MaRTS [96], a **M**odel-based **R**egression **T**est **S**election approach. We proposed MaRTS to address the first limitation described in Section 1.2 of Chapter 1. The existing model-based RTS approaches [7,11,14–16] do not support the identification of changes to inherited and overridden operations along the inheritance hierarchy, which can result in missing relevant test cases that must be selected for regression testing.

MaRTS uses class and activity diagrams to represent fine-grained behaviors of a software system and its test cases. Each activity diagram describes the details of an operation's implementation. MaRTS is based on static analysis of the class diagram to identify the changes to inherited and overridden operations along the inheritance hierarchy, and dynamic analysis of the test coverage information that is obtained by executing the models representing the test cases with the models representing the system under test. The test cases are classified as obsolete, reusable, or retestable. MaRTS can support both unit and functional test cases.

## 4.1 Proposed Approach

We developed MaRTS within the context of FiGA [37, 38] that is based on ReverseЯ [82] to generate UML models from a running Java software system. MaRTS exploits ReverseЯ to generate UML class and activity diagrams from the source code of the system under test and its test cases. Each method of a class is represented as an activity diagram. MaRTS uses the following five steps that are automated:

1. Extract the operations-table from the original class diagram (Section 4.1.1).

2. Calculate the traceability matrix (Section 4.1.2).

3. Identify model changes (Section 4.1.3).

4. Extract the operations-table from the adapted class diagram (Section 4.1.4).

5. Classify test cases (Section 4.1.5).


**Figure 4.1:** Chess Class Diagram Before Refactoring.

### 4.1.1 Extraction of the Operations-Table from the Original Class Diagram

An *operations-table* stores for each class the operations that can be invoked on an object of that class type. In this step, the operations-table is extracted from the original class diagram. This table will be used in a later step along with the operations-table generated from the new class

**Table 4.1:** Operations-table for the `Pawn` and `Knight` Classes Shown in Fig. 4.1

| Classes | Operations | | | |
|---|---|---|---|---|
| | **Declaring Class** | **Operation Name** | **Parameters types** | **Return Type** |
| `Pawn` | `Pawn` | `onePossibleMove` | `int,int` | `String` |
| | `Pawn` | `toString` | `None` | `String` |
| | `Pawn` | `getColor` | `None` | `Color` |
| | `Pawn` | `getColumn` | `None` | `int` |
| | `Pawn` | `getPosition` | `None` | `String` |
| | `Pawn` | `getRow` | `None` | `int` |
| | `Pawn` | `setColumn` | `int` | `void` |
| | `Pawn` | `setPosition` | `String` | `void` |
| | `Pawn` | `setRow` | `int` | `void` |
| | `Pawn` | `legalMoves` | `None` | `ArrayList<String>` |
| `Knight` | `Knight` | `onePossibleMove` | `int,int` | `String` |
| | `Knight` | `toString` | `None` | `String` |
| | `Knight` | `getColor` | `None` | `Color` |
| | `Knight` | `getColumn` | `None` | `int` |
| | `Knight` | `getPosition` | `None` | `String` |
| | `Knight` | `getRow` | `None` | `int` |
| | `Knight` | `setColumn` | `int` | `void` |
| | `Knight` | `setPosition` | `String` | `void` |
| | `Knight` | `setRow` | `int` | `void` |
| | `Knight` | `legalMoves` | `None` | `ArrayList<String>` |

diagram to determine which operations are inherited or overridden in each class. Fig. 4.1 shows a class diagram for a chess program, which is used as our running example in this chapter. Table 4.1 shows part of the operations-table with the entries for the `Pawn` and `Knight` classes from Fig. 4.1.

The first column of Table 4.1 shows the class names. For each operation that can be called on an instance of a class listed in column 1, columns 2, 3, 4, and 5 store the operation's declaring class, name, formal parameter types, and return type respectively. Columns 3, 4, and 5 constitute the signature of the operation. For each class in the first column of the table, the name of its superclass is also stored. This is not shown in Table 4.1 because the `Pawn` and `Knight` classes only implicitly inherit from `Object`.

MaRTS does not store interface operations when populating the operations-table. This information is not needed for RTS because a change to an interface operation is realized by a change to the class that implements it, and changes to class operations are captured in the operations-table. Interfaces can still be used as types in the operations-table, i.e., return types and parameter types of operations.

### 4.1.2 Traceability Matrix Calculation

As explained in Section 3.1 of Chapter 3, the activity diagrams generated in FiGA are detailed and can be executed using the RSA simulation toolkit 9.0. We implemented a component for RSA that can collect coverage information for test cases at the model level. This component is applied after the models are obtained from the program via ReverseЯ and before these models are adapted. The RSA model simulation tool generates an executable representation of the UML models obtained via ReverseЯ. The component is used to instrument this executable representation with statements that write the coverage of test cases at the model level to a file during model execution.

The activity diagrams representing the test cases are executed with the activity diagrams representing the program methods. During model execution, three types of information are collected for each test case: (1) which activity diagrams are executed by the test case, (2) what is the receiver object type for each executed activity diagram, and (3) which flows in each activity diagram are executed. This information is used to obtain the traceability matrix at the transition flow level, henceforth called the flow-level traceability matrix. This matrix can also be used to obtain the activity-level traceability matrix by omitting information about the traversed transition flows to only record which activity diagrams representing program methods are traversed by exercising test cases.

Table 4.2 shows a portion of a flow-level traceability matrix for `testGetColumn` test case shown in Listing 1. For the sake of simplicity, we omitted the information related to the receiver types. We show the traversed transition flows as a source-destination node pair. The

**Table 4.2:** Portion of a Flow-level Traceability Matrix

| Activity diagrams | Transition flows | testGetColumn |
|---|---|---|
| ChessBoard_initialize | (Start node → Receive args) | x |
| | (Receive args → Initialize chess pieces) | x |
| | (Initialize chess pieces → End node) | x |
| Pawn_getColumn | (Start node → Receive args) | x |
| | (Receive args → Return column field) | x |
| | (Return column field → End node) | x |

`testGetColumn` test case traverses the flows of the `ChessBoard_initialize` and `Pawn-etColumn` activity diagrams; in the third column of the table, an `'x'` denotes that the transition flow in that row was traversed by the test case.

```
class Test {
   public void testGetColumn() {
        ChessBoard board = new ChessBoard();
        board.initialize();
        ChessPiece piece=board.getPiece("d7");
        assertEquals(piece.getColumn(), 3);
   }

}
```

**Listing 1:** Chess Program Test Case.

### 4.1.3 Model Change Identification

The UML diagrams generated by ReverseЯ are compliant with the RSA modeling tool, which MaRTS uses to identify the model changes that occur when developers modify the class and activity diagrams. This tool identifies the changed model elements and how they are changed. The list of differences is saved in a text file and used as input to classify the test cases. The class diagram changes that can be identified by the RSA model comparison feature are:

- Addition, deletion, and modification of class attributes and operations.
- Addition, deletion, and modification of classes and relations between classes.

The activity diagram changes that can be identified by the RSA model comparison tool are:

- Addition and deletion of action nodes, call behavior action nodes, decision and merge nodes, and start and end nodes.
- Modification of action nodes based on changes to code stored in a code snippet associated with an action node.
- Addition and deletion of transition flows.
- Modification of a boolean expression associated with a transition flow.
- Addition, deletion, and modification of attributes of an activity diagram.
- Addition, deletion, and modification of an activity diagram.

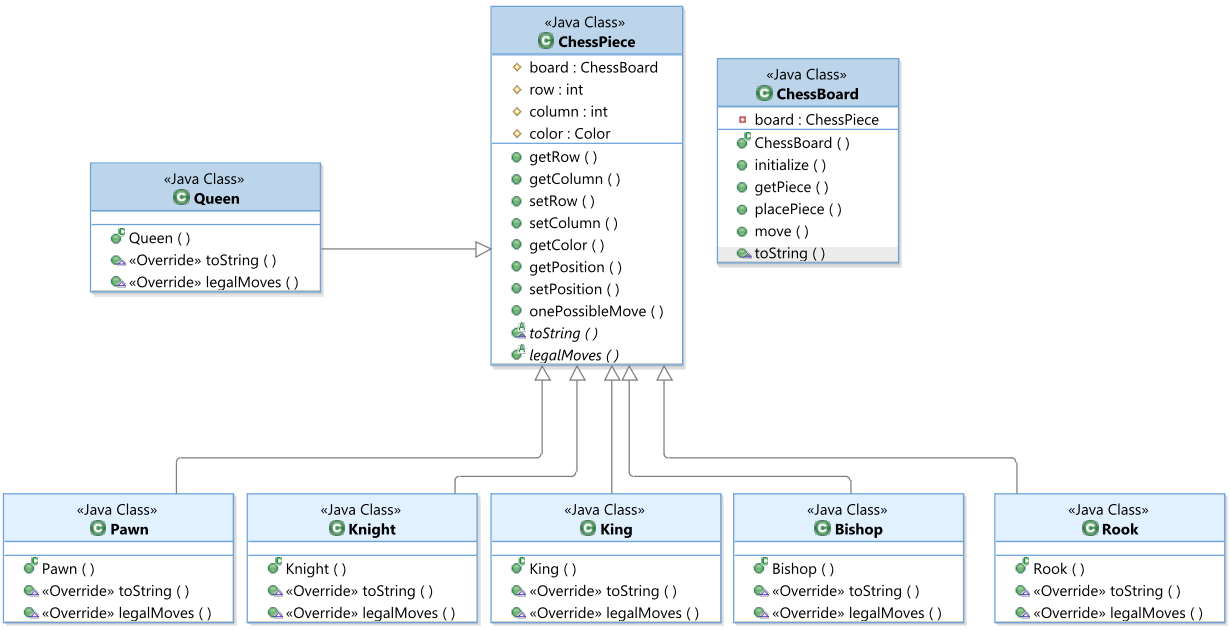



**Figure 4.2:** Class Diagram After Refactoring.

### 4.1.4 Extraction of the Operations-Table from the Adapted Class Diagram

During the adaptation process of the class diagram, the operations declared and inherited in each class might change, which results in a change in the operations-table. Thus, once the devel-

**Table 4.3:** Operations-table for the Modified `Pawn` and `Knight` Classes

| Classes | Operations | | | |
|---|---|---|---|---|
| | **Declaring Class** | **Operation Name** | **Parameters types** | **Return Type** |
| `Pawn` extends `ChessPiece` | `ChessPiece` | `onePossibleMove` | `int,int` | `String` |
| | `Pawn` | `toString` | `None` | `String` |
| | `ChessPiece` | `getColor` | `None` | `Color` |
| | `ChessPiece` | `getColumn` | `None` | `int` |
| | `ChessPiece` | `getPosition` | `None` | `String` |
| | `ChessPiece` | `getRow` | `None` | `int` |
| | `ChessPiece` | `setColumn` | `int` | `void` |
| | `ChessPiece` | `setPosition` | `String` | `void` |
| | `ChessPiece` | `setRow` | `int` | `void` |
| | `Pawn` | `legalMoves` | `None` | `ArrayList<String>` |
| `Knight` extends `ChessPiece` | `ChessPiece` | `onePossibleMove` | `int,int` | `String` |
| | `Knight` | `toString` | `None` | `String` |
| | `ChessPiece` | `getColor` | `None` | `Color` |
| | `ChessPiece` | `getColumn` | `None` | `int` |
| | `ChessPiece` | `getPosition` | `None` | `String` |
| | `ChessPiece` | `getRow` | `None` | `int` |
| | `ChessPiece` | `setColumn` | `int` | `void` |
| | `ChessPiece` | `setPosition` | `String` | `void` |
| | `ChessPiece` | `setRow` | `int` | `void` |
| | `Knight` | `legalMoves` | `None` | `ArrayList<String>` |

opers have fully adapted the class and the activity diagrams, the operations-table is again extracted from the adapted class diagram.

In the original class diagram shown in Fig. 4.1, each of the methods `getRow()`, `getColumn()`, `setRow()`, `setColumn()`, `getColor()`, `getPosition()`, `setPosition()`, and `onePossibleMove()` that are declared in the `ChessPiece` interface have a copy of the same implementation in all the implementing classes. Suppose that the chess program is refactored to reduce the code duplication. The interface is converted to an abstract class `ChessPiece`. The classes `Pawn`, `Knight`, `King`, `Queen`, `Rook`, and `Bishop` now *extend* `ChessPiece` instead of *implementing* it. According to Fowler's refactoring catalog [97], the redundant methods are *pulled up* from the subclasses to the `ChessPiece` class. The *realization* relation is replaced with a *generalization* relation. We adapted the class and activity diagrams to apply this refactoring. Six

`generalization` relations were added from the subclasses to the superclass `ChessPiece`, and six `realization` relations were deleted. The existing 48 operations were deleted from the subclasses, and eight operations were added to the `ChessPiece` class. The newly added eight operations were inherited by all of the subclasses. Fig. 4.2 shows a portion of the refactored classes for the chess program.

Table 4.3 shows the operations-table for the `Pawn` and `Knight` classes after the chess program is refactored. Now the `Pawn` and `Knight` classes `extend` the `ChessPiece` class.

### 4.1.5 Test Case Classification

Algorithm 1 adopts a greedy approach to accomplish the task. At first, all the test cases are considered to be reusable, i.e., they belong to the set of reusable test cases. The algorithm iterates through the test cases affected by the model changes. Then it compares the operations-tables extracted from the class model before and after the change to determine which operations have been changed—including if they have been moved along the inheritance hierarchy. The activity-level traceability matrix is used to determine which test case is affected by those changes. The following rules are applied by Algorithm 1 (lines 3-12):

1. When an operation *op*
   (a) initially inherited by a class *C* is now overridden by *C* or one of its ancestors, or
   (b) initially declared by a class *C* is now moved to one of the ancestors of *C*, or
   (c) initially declared or inherited by a class *C* is now neither declared nor inherited by *C*

   *Action:* move any test case that traverses *op* on a receiver of type *C* from the set of reusable test cases to the set of retestable ones. Algorithm 1 calls Algorithm 2 (`findRetestableTests`) in line 6 to perform this action.
2. When an operation *op* initially declared or inherited by a class *C* is now neither declared nor inherited by *C*

   *Action:* move any test case that directly calls *op* on a receiver of type *C* is moved from the set of reusable test cases to the set of obsolete ones.

**Algorithm 1:** classifyTestCases($T$, $OT$, $OT'$, $TM_f$, $TM_a$, $MD$, $AM$)

**Input:**

$T$: Set of initial test cases.
$OT$: Operations-table extracted from the original class diagram.
$OT'$: Operations-table extracted from the adapted class diagram.
$TM_f$: Flow-level traceability matrix.
$TM_a$: Activity-level traceability matrix.
$MD$: Model differences generated by RSA.
$AM$: Set of all activity diagrams representing the original system.

**Output:**

$T_r$: Set of retestable test cases.
$T_o$: Set of obsolete test cases.
$T_u$: Set of reusable test cases.

```
1  T_r=T_o=∅
2  T_u=T
3  for each operation op ∈ OT do
       /* Each operation op has the following attributes in the
          operation table: <c, dc, sig, rt>, where c is a class in OT
          that has op (i.e., op is inherited or declared in c), dc is
          the declaring class of the operation (dc can be c or any
          ancestor of c), sig is the signature, and rt is the return
          type of the operation.                                    */
4      if OT' contains an operation op' where op'.c = op.c AND op'.sig = op.sig AND op'.rt = op.rt
        then
5          if op'.dc ≠ op.dc then
6              findRetestableTests(TM_a, T_r, T_o, T_u, c, op);
7          end
8      else
9          findObsoleteTests(TM_a, T_r, T_o, T_u, c, op);
10         findRetestableTests(TM_a, T_r, T_o, T_u, c, op);
11     end
12 end
13 for each change ch in MD do
14     if ch involves deletion/modification of a transition flow tflow in an activity diagram act then
15         findRetestableTestsTrans(TM_f, T_r, T_o, T_u, tflow, act);
16     end
17     if ch involves deletion/modification of a node n in an activity diagram act then
18         findRetestableTestsNodes(TM_f, T_r, T_o, T_u, n, act);
19     end
20     if ch involves deletion/modification of a constructor c in a class then
21         findRetestableTestsConstructors(TM_a, T_r, T_o, T_u, c);
22     end
23     if ch involves deletion/modification of a field f in a class then
24         findRetestableTestsFields(TM_a, T_r, T_o, T_u, AM, f);
25     end
26 end
27 return T_r, T_o, T_u;
```

**Algorithm 2:** findRetestableTests($TM_a$, $T_r$, $T_o$, $T_u$, C, OP)

**Input:**

$TM_a$: Activity-level traceability matrix.
$T_r$: Set of retestable test cases.
$T_o$: Set of obsolete test cases.
$T_u$: Set of reusable test cases.
*C*: A class name.
*OP*: An operation.

**Output:**

$T_r$: Set of retestable test cases.
$T_o$: Set of obsolete test cases.
$T_u$: Set of reusable test cases.

```
1  for each test case tc ∈ TM_a do
       /* The traceability matrix TM_a provides information about (1)
          the set T_c that contains the activity diagrams traversed by tc
          along with their receiver types, and (2) the set T_d that
          contains the activity diagrams that are directly called by tc
          along with their receiver types                                */
2      if tc ∈ T_u then
3          if tc.T_c.contains(OP) such that OP is called on a receiver of type rc then
4              if rc = C then
5                  remove tc from T_u;
6                  add tc to T_r;
7              end
8          end
9      end
10 end
```

Once Algorithm 1 completes iterating over all the modified operations, the test cases that are still in the set of reusable test cases are classified by using model differencing. The activity- and flow-level traceability matrices are used to detect which of these test cases are traversing (i) a deleted or modified transition flow, (ii) a transition flow ending in a deleted or modified node, (iii) a deleted or modified constructor and (iv) a usage of a field that has been deleted or modified. They are all moved to the set of the retestable test cases (lines 14-27 in Algorithm 1 deal with these cases).

Let us show how the `testGetColumn()` test case shown in Listing 1 is classified by Algorithm 1 after the chess system is adapted. Tables 4.1 and 4.3 show the operations-table before and after the adaptation respectively. In Table 4.1, the `Pawn` class contains `Pawn.getColumn()` that was moved to the `ChessPiece` class during the adaptation process. This is evident from Table 4.3 where the entry for `getColumn()` in `Pawn` reports `ChessPiece` as the declaring class. According to the second case of the first rule reported above, the `testGetColumn()` test case is classified as retestable.

## 4.2 Evaluation

The goals of the evaluation were to (1) demonstrate that MaRTS is at least as inclusive and precise as some code-based RTS approaches that support changes to the inheritance hierarchy, and (2) evaluate the fault detection ability of the reduced test set with the fault detection ability of the full test set. The Inclusiveness and precision are defined in Chapter 2. The empirical study is driven by the following Research Questions (RQ):

RQ1: Does MaRTS have lower inclusiveness and precision compared to the inclusiveness and precision of the code-based RTS approaches that consider changes to the inheritance hierarchy?

RQ2: What is the difference between the fault detection ability of the reduced test set achieved by MaRTS and the fault detection ability of the full test set?

We compared the results from running MaRTS and two code-based RTS approaches DejaVu[2] and ChEOPSJ[3] on four subject programs. DejaVu is an implementation of the approach proposed by Harrold et al. [69] while ChEOPSJ is an implementation of the approach proposed by Soetens et al. [66]. Both tools support RTS for Java programs. We selected DejaVu because it is safe and precise, and because it detects fine-grained changes at the statement level. We selected ChEOPSJ because it detects fine-grained changes to method invocations. Both detect changes to the inheritance hierarchy. We did not compare MaRTS with Ekstazi [10] and STARTS [54] because they detect changes at the class level and perform class-level dependency analysis.

We did not compare MaRTS with the existing model-based RTS approaches because they lack tool support (or tools are unavailable), and they do not consider the impact of changes to the inheritance hierarchy at the model level. Therefore, it is more relevant to demonstrate that MaRTS provides results comparable to those provided by the code-based approaches.

### 4.2.1 Experimental Setup

MaRTS does not support the following features of Java due to limitations in the underlying tools used to reverse engineer and execute the models: multithreaded programming, generic types, and new features introduced in Java versions 8-11. Therefore, we did not evaluate MaRTS using subject programs that use these features. We used four subject programs: (1) graph package of the Java Universal Network/Graph Framework (JUNG)[4], (2) Siena[5], (3) chess program used as a motivating example in this paper, and (4) XML-security[6]. These programs were implemented using Java 6 and 7. They use classes, interfaces, `extends` relations between classes, `extends` relations between interfaces, and `implements` relations between classes and interfaces. These subjects do not use

[2]http://sofya.unl.edu/doc/manual/user/apps-dejavu.html

[3]http://win.ua.ac.be/~qsoeten/other/cheopsj/

[4]http://jung.sourceforge.net/download.html

[5]http://sir.unl.edu/portal/bios/siena.php

[6]http://sir.unl.edu/portal/bios/xml-security.php

**Table 4.4:** Original Programs

| Software System | Version | Num. classes | Num. interfaces | Num. extends relations | Num. implements relations | Num. methods | LOC |
|---|---|---|---|---|---|---|---|
| JUNG | 1.3.0 | 13 | 12 | 12 | 11 | 146 | 3655 |
| Siena | 1.8 | 9 | 0 | 0 | 0 | 95 | 1605 |
| Chess | 0 | 7 | 1 | 0 | 6 | 65 | 1074 |
| XML-security | 2 | 173 | 6 | 131 | 30 | 1172 | 16800 |

generic types, and do not use multithreaded programming. We used EvoSuite [98] in its default setting to generate JUnit test cases for each subject program. The reason for using EvoSuite is that it targets both the coverage and mutation score, i.e., generating test cases with high coverage and mutation score.

Multiple versions of the JUNG graph package are available. We selected versions 1.3.0 and 1.4.0 because the adaptation from version 1.3.0 to 1.4.0 involves changes to the inheritance hierarchy. Table 4.4 summarizes the data about version 1.3.0. We used EvoSuite to generate JUnit test cases for each class in version 1.3.0. A total of 188 test cases were generated in 60 seconds that exercised 81% of the statements in version 1.3.0. We extracted class and activity diagrams from version 1.3.0 and its generated test cases.

Siena is an Internet-scale event notification middleware implemented in Java. Siena is logically divided into a set of six components consisting of nine classes. We obtained the source code for versions 1.8, 1.12, and 1.14, where each version consists of nine classes. Table 4.4 summarizes the data about version 1.8. We used EvoSuite to generate JUnit test cases for each class of version 1.8. The tool generated 107 JUnit test cases that exercise 89% of the statements. We extracted class and activity diagrams from version 1.8 and its generated test cases.

The chess program was presented in Section 4.1.1. We used EvoSuite to generate JUnit test cases for each class of the program, and 130 test cases that exercise 96% of the statements were generated. We created class and activity diagrams for the original version of the chess program and its test cases.

XML-security is a component library implementing XML signature and encryption standards. We selected versions v2 and v3 because the adaptation from version v2 to v3 involves a large set

of changes to classes, interfaces, realization and generalization relations, and operations. Table 4.4 summarizes the data about version v2. XML-security v2 comes with a JUnit test suite that consists of 94 test cases, which exercise 31% of the statements in v2. We used EvoSuite to generate JUnit test cases for all the classes in v2. The generated test cases did not improve the coverage of the existing test suite. Therefore, we excluded the generated test cases from this study, and only considered the existing test cases that come with the application. We did not manually create new test cases to improve the coverage because we are not domain experts in XML-security. We extracted class and activity diagrams from version v2 and its test cases.

#### 4.2.1.1 Adaptations at the Model Level

For each subject program, we manually adapted the class and activity diagrams from one version to the following version in a systematic way. First, the code differences between the two versions were identified. Second, these code differences were applied at the model level. If an `extends` relation is added/deleted between two classes at the code level, then a `generalization` relation is added/deleted between the two classes the model level. If an `implements` relation is added/deleted between a class and an interface at the code level, then a `realization` relation is added/deleted between the class and the interface the model level. If a class/interface is added/deleted at the code level, then a corresponding class/interface is added/deleted at the model level. If a class attribute is added/deleted at the code level, then the attribute is added/deleted in the corresponding class at the model level. If a method is added/deleted at the code level, then a corresponding operation is added/deleted at the model level. If new statements are added to a method implementation at the code level, then the activity diagram representing the method is modified by adding these new statements to the code snippet of the corresponding action node in the activity diagram. The deletion of statements from a method is treated in the same way.

**JUNG**. Table 4.5 summarizes the data about the adaptation from version 1.3.0 to 1.4.0. The adaptation involved changes to the inheritance hierarchy. Four generalization relations were modified in version 1.4.0. For example, class `SparseVertex` that extended `AbstractSparseVer-`

**Table 4.5:** Adaptations Performed on Models

| Software system | Evolution | Changes | | | |
|---|---|---|---|---|---|
| | | classes & interfaces | generalizations | realizations | operations |
| JUNG | 1.3.0 → 1.4.0 | 5 | 7 | 2 | 79 |
| Siena | 1.8 → 1.12 | 0 | 0 | 0 | 9 |
| Siena | 1.8 → 1.14 | 0 | 0 | 0 | 11 |
| Chess | 0 → 1 | 1 | 6 | 6 | 56 |
| XML-security | 2 → 3 | 52 | 37 | 2 | 311 |

tex in version 1.3.0 was modified to extend SimpleSparseVertex in version 1.4.0. Similarly, class UndirectedSparseGraph that extended AbstractSparseGraph was modified to extend SparseGraph in version 1.4.0. Three new generalization relations were added during the adaptation process. For example, a new class SparseGraph was added. A new generalization relation was added from this new class to the existing class AbstractSparseGraph. Additionally, operations were moved between classes along the inheritance hierarchy, i.e., operations deleted from a subclass and added to its super class. For example, 19 operations were moved from SparseVertex class to its superclass SimpleSparseVertex, and 7 operations were moved from AbstractSparseGraph class to its super class. These 19 and 7 operations are still inherited by the SparseVertex and AbstractSparseGraph classes, respectively. All these operations were counted among the 79 modified operations shown in Table 4.5.

**Siena**. We adapted the models of Siena from version 1.8 to 1.12, and from version 1.8 to 1.14. Version 1.8 is the first version for both the adaptations. The reason for considering these adaptations is that moving to any other version does not result in reducing the number of selected test cases. Table 4.5 summarizes the data for the adaptations. These two adaptations do not involve changes to the inheritance hierarchy, such as adding or deleting generalization relations. They only involve method-level changes. The added/deleted/modified operations in these adaptations are not inherited/overridden along the inheritance hierarchy.

**Chess**. Table 4.5 summarizes the data about the adaptation. Forty eight operations were deleted from the subclasses that extend the ChessPiece class, and 8 operations were added

**Table 4.6:** Number of Test Cases Selected by the RTS Approaches

| Software system | Evolution | Num. Test Cases | Retestable Test Cases | | |
|---|---|---|---|---|---|
| | | | DejaVu | ChEOPSJ | MaRTS |
| JUNG | 1.3.0 → 1.4.0 | 188 | 188 | 178 | 188 |
| Siena | 1.8 → 1.12 | 107 | 26 | 54 | 26 |
| Siena | 1.8 → 1.14 | 107 | 36 | 59 | 36 |
| Chess | 0 → 1 | 130 | 130 | 126 | 130 |
| XML-security | 2 → 3 | 94 | 94 | N/A | 84 |

to the ChessPiece class. This adaptation involves changes to the inheritance hierarchy. Six generalization relations were added from the subclasses to the superclass ChessPiece, and six realization relations were deleted. The 8 newly added operations are inherited by all of the subclasses.

**XML-security**. We adapted the class and activity diagrams of XML-security from version 2 to 3. This adaptation involved these changes: 44 classes deleted, 2 classes added, 2 interfaces deleted, 5 interfaces added, 35 generalization relations deleted, 2 generalization relations added, 2 realization relations added, and 287 operations deleted, and 24 operations added.

After the model-level adaptation process was completed, we applied MaRTS to classify test cases. We also applied DejaVu and ChEOPSJ at the code level for each subject.

### 4.2.2 RQ1: Inclusiveness and Precision

Table 4.6 shows the results of running the three RTS approaches.

**JUNG**. MaRTS classified all the 188 test cases as retestable. The reason is that most of these test cases traverse the 19 operations that were moved along the inheritance hierarchy. The rest of these test cases traverse modified constructors and/or the modified activity diagrams representing the operations AbstractSparceGraph.addVertex() and AbstractSparceGraph.addEdge(). MaRTS did not classify any test case as obsolete because the adaptation did not involve deleting operations that are directly called from test cases. DejaVu also classified all the

test cases as retestable for the same reason. ChEOPSJ selected 178 test cases out of 188. It missed ten test cases that traverse modified code. The reason for missing these test cases is that below the method level, ChEOPSJ only records changes on method invocations, but not on local variables and other types of statements. ChEOPSJ does not support identifying changes to constructors [66]. MaRTS was able to select these ten test cases as retestable because it can identify any change to a method implementation in the activity diagram representing the method. For example, if any statement inside an action node is modified, then the model differencing tool identifies that action node as modified. MaRTS can also identify changes performed to a constructor through model differencing when a constructor is modified in the class model.

**Siena**. MaRTS selected 26 out of 107 test cases when moving from version 1.8 to 1.12, and selected 36 out of 107 test cases when moving from version 1.8 to 1.14, i.e., by considering all performed changes to move from version 1.8 to 1.12 then to 1.14. The reason for this reduction in the number of selected test cases is that these adaptations are inside the method implementations and do not include changes to the inheritance hierarchy. The modified methods are traversed by few test cases in the available test set. DejaVu achieved similar results as MaRTS, i.e., both DejaVu and MaRTS classified the same set of test cases as retestable.

ChEOPSJ showed different results. For the adaptation from version 1.8 to 1.12, ChEOPSJ classified 54 test cases out of 107 as retestable. ChEOPSJ missed two modification-traversing test cases that were selected by MaRTS and DejaVu. ChEOPSJ also classified more test cases as retestable compared to MaRTS and DejaVu even though these extra test cases were not traversing modified code. The reason is that ChEOPSJ is based on static analysis of dependencies between modified code and test cases. For each change, ChEOPSJ first identifies the method where the change occurred. Second, it identifies all the methods that directly or indirectly invoke the method where the change occurred by following the chain of invocation dependencies between methods. It selects every test case that contains an invocation to any of the identified methods because of the potential for the test case to execute the modified methods.

**Table 4.7:** Number of False Positives (FP) and False Negatives (FN) for the Studied RTS Approaches

| Software system | Evolution | DejaVu | | ChEOPSJ | | MaRTS | |
|---|---|---|---|---|---|---|---|
| | | Num.FP | Num.FN | Num.FP | Num.FN | Num.FP | Num.FN |
| JUNG | 1.3.0 → 1.4.0 | 0 | 0 | 0 | 10 | 0 | 0 |
| Siena | 1.8 → 1.12 | 0 | 0 | 30 | 2 | 0 | 0 |
| Siena | 1.8 → 1.14 | 0 | 0 | 28 | 4 | 0 | 0 |
| Chess | 0 → 1 | 0 | 0 | 0 | 4 | 0 | 0 |
| XML-security | 2 → 3 | 0 | 0 | N/A | N/A | 0 | 0 |

For the adaptation from version 1.8 to 1.14 (i.e., considering all changes from 1.8 to 1.14), ChEOPSJ selected 59 out of 107 test cases. It missed five modification-traversing test cases that were selected by MaRTS and DejaVu. These five test cases traverse the method `AttributeValue.booleanValue()`, for which statements were modified inside the method body. One possible reason is that ChEOPSJ does not identify all types of changes that can be performed inside a method body.

**Chess**. For the chess program, MaRTS classified all of the 130 test cases as retestable because every test case traverses at least one operation that was moved between classes or an activity diagram/constructor that accesses modified fields. MaRTS did not classify any test case as obsolete although there were 48 deleted operations from the subclasses of the `ChessPiece` class, and many test cases contain direct calls to the deleted operations. The reason is that these subclasses inherit operations from the `ChessPiece` class with the same signatures as the deleted ones, and thus, the test cases that contain direct calls to the deleted operations will call the inherited ones in the adapted version of the models.

DejaVu classified all of the 130 test cases as retestable while ChEOPSJ classified 126 out of 130 as retestable. ChEOPSJ missed four test cases because they traverse operations that access modified instance fields. ChEOPSJ cannot identify such changes. In MaRTS, changes to instance fields can be identified through model differences, and activity diagrams accessing these fields are identified by parsing the file containing the models.

**XML-security**. MaRTS classified 84 out of 94 test cases as retestable, and 10 test cases as obsolete. The reason is that all of the 94 test cases traverse deleted operations and/or modified bodies of operations. The 10 obsolete test cases contain direct calls to deleted methods. The test classes containing these 10 test cases are `KeyWrapTest.java` and `BlockEncryptionTest.java`. These test classes do not compile with the XML-security v3. These test classes only contain the 10 test cases classified as obsolete by MaRTS, where these obsolete test cases either need to be modified or deleted. DejaVu selected all of the 94 test cases for regression testing, where all the test cases traverse modified and/or deleted code. DejaVu does not address identifying obsolete test cases.

We did not get results for ChEOPSJ when we ran it on the XML-security subject. It did not detect code changes that it is supposed to detect. Other researchers [76] encountered the same issue with the subjects used in their study. Table 4.6 and Table 4.7 do not show results for ChEOPSJ with respect to the XML-security subject.

**Inclusiveness results.** DejaVu is a safe tool and classifies all modification-traversing test cases as retestable, and therefore, its inclusiveness was 100% for all the subject programs. Because MaRTS selected the same set of test cases that were selected by DejaVu for all the subject programs, its inclusiveness was also 100%. ChEOPSJ missed modification-traversing test cases, and its inclusiveness was 94% for JUNG, 96% for Chess, 92% for Siena version 1.12, and 88% for version 1.14.

**Precision results.** The precision was 100% for MaRTS and DejaVu because both approaches did not classify any test case that is non modification-traversing as retestable for all the subject programs. The precision of ChEOPSJ was 100% for JUNG and Chess, 62% for Siena version 1.12, and 60% for version 1.14. Table 4.7 shows the number of false positives and false negatives for each of the studied RTS approaches.

Thus, the answer to *RQ1* is that the inclusiveness and precision for MaRTS were never lower than the inclusiveness and precision of DejaVu and ChEOPSJ.

### 4.2.3 RQ2: Fault Detection Ability

The MaRTS results showed a reduction in the number of selected test cases only for the Siena program for the adaptation from version 1.8 to 1.12, and from 1.8 to 1.14. Therefore, we evaluated the fault detection ability of the reduced test sets obtained by MaRTS for these two adaptations. We used mutation testing in this experiment to compare the fault detection ability of the reduced test sets with the fault detection ability of the full test sets. There are no tools (to the best of our knowledge) that support systematic generation of mutations at the model level. Therefore, we used a code-level mutation testing tool on the two versions of the Siena program (1.12 and 1.14).

The experiment consists of three steps. In the first step, all the 107 test cases in the baseline test suite were executed on the code for version 1.8; all the test cases passed. This check is needed to ensure that the baseline test cases do not expose faults in the original version.

In the second step, `PIT`[7] was used to apply first-order method-level mutation operators to versions 1.12 and 1.14. The applied mutation operators[8] were (1) Conditionals Boundary Mutator, (2) Increments Mutator, (3) Invert Negatives Mutator, (4) Math Mutator, (5) Negate Conditionals Mutator, and (6) Void Method Calls Mutator. We configured `PIT` to only mutate the modified methods to adapt from version 1.8 to 1.12 and to 1.14.

In the third step, for each version, we ran `PIT` with both the full and reduced test sets. `PIT` generates a mutation report that shows (1) information about all the applied mutations, such as the location of a mutated statement and the change made to that statement, and (2) which mutations survived or were killed by the full and reduced test sets.

Table 4.8 shows the mutation testing results. Both the full and reduced test sets killed exactly the same set of mutants in both the versions (40 out of 134 mutants in version 1.12 and 42 out of 136 mutants in version 1.14). The fault detection ability of the reduced test set was never lower than that of the full test set. The reason is that any test case that traverses an adapted method was

[7] http://pitest.org

[8] http://pitest.org/quickstart/mutators/

**Table 4.8:** Mutation results for the Siena program

| Program | Mutants | Full Test Set | | Reduced Test Set | |
|---|---|---|---|---|---|
| | | size | score | size | score |
| `Siena 1.12` | 134 | 107 | 29.8% | 26 | 29.8% |
| `Siena 1.14` | 136 | 107 | 30.9% | 36 | 30.9% |

in the reduced test set, i.e., classified as retestable, and we used `PIT` to mutate only the adapted methods. The remaining mutants were not killed by either the full or the reduced test set.

Thus, the answer to *RQ2* is that the fault detection ability of the reduced test set achieved by MaRTS was equal to the fault detection ability of the full test set.

### 4.2.4 Discussion

MaRTS achieved results comparable to DejaVu, but it outperformed ChEOPSJ in terms of inclusiveness and precision. The inclusiveness of MaRTS was 100%. Inclusiveness is important for ensuring the correctness of the changes performed to a system because the modification-traversing test cases can reveal faults in the system. Moreover, MaRTS can also identify one type of obsolete test cases as in the case of XML-security study. The code-based RTS approaches compared in this study do not address the identification of any type of obsolete test cases.

The fault detection ability experiment showed that the reduced test sets obtained by MaRTS had the same fault detection ability as that of the full test sets. The reason is that MaRTS classified all modification-traversing test cases as retestable. We only mutated the adapted methods, assuming that developers insert new faults only to these methods during the adaptation process.

### 4.2.5 Threats to Validity

We identify several threats to validity of the results of our case study.

**External validity.** It is difficult to generalize from a study of only four subject programs. However, we selected program versions that incorporate various types of modifications, such as changes to classes, methods, inheritance hierarchy, and class attributes.

**Internal validity.** The unknown factors that might affect the outcome of the analyses are possible errors in our algorithm implementation, and that the test cases were generated only using one test case generation tool. To control the first factor, we tested the implementation of MaRTS on different change scenarios. We also compared the results achieved by MaRTS for the case studies with those of DejaVu, which is known to be safe and precise.

We used EvoSuite to generate JUnit test cases for the subject programs. The results could potentially change if other test generation tools were used or test sets with different coverage numbers were used (i.e., to what extent do the test cases exercise modified code). We plan to evaluate the proposed approach on additional test suites generated by other test case generation tools.

A major threat is that the same person selected the subject programs, generated the test cases, reverse engineered the models, performed the model-level adaptations, and executed the RTS tools. There is a potential for getting different results if different people worked on these steps. The test generation process and RTS approaches were automated, and thus, having other people perform those steps would not make a difference if they used the same tool configurations. The adaptations are, however manual, which can lead to different modifications. However, since we started from a particular version of code and finished at a well-defined version of code, the differences are not likely to be significant.

**Construct validity.** We used the reduction in the number of selected test cases, safety, and precision in our study. However, there are other metrics that can be used to evaluate an RTS approach, such as its efficiency in terms of reducing regression testing time. We plan to evaluate the efficiency of MaRTS in the future.

**Conclusion validity.** The main threat to conclusion validity is the sample size used in the evaluation. We only used 4 subjects with one test suite for each of them. Using additional subjects could affect the conclusions regarding the inclusiveness, precision, and fault detection ability.

### 4.2.6 Time Complexity for MaRTS

We did not empirically compare the running times of MaRTS, DejaVu, and ChEOPSJ because they are prototype tools that were built by different people with different skills and were not optimized for efficiency. However, we performed a theoretical analysis of MaRTS time complexity. Algorithm 1 has two main loops. The first loop (lines 3-12) iterates through all the operations in the original operations-table. Let us suppose that $N$ is the number of classes in the program. In the worst case, they are all part of a single inheritance hierarchy, i.e., there is a linear chain of $N$ classes. Let $c$ be a constant representing the number of operations in the topmost class in the hierarchy; in the worst case this is also the largest number of methods in any class of the application. Suppose that each class adds $c$ new methods to those inherited from its parent. In the worst case, the number of methods available for invoking is $c$ for the top-most class, $(c + c)$ for the second class, $(c+c+c)$ for the third class, and so on. The total number of operations, and thus, the number of entries in the operations-table is $(N * (N + 1)/2)c$, which is $O(N^2)$.

The original and adapted operation tables are implemented as HashMaps, so the cost of retrieving elements is a constant that does not affect the worst case time complexity.

For each test case in the traceability matrix, the traversed activity diagrams and edges by the test case are stored as HashSets, and the cost of retrieving from a HashSet is constant $k$. Therefore, the cost of a single call to the `findRetestableTests()` algorithm is $T * k$, where $T$ is the total number of baseline test cases, and $k$ is a constant representing the cost to retrieve an item from a HashSet. Thus, the worst case time complexity to call `findRetestableTests()` for all the operations in the operations-table is $O(T * N^2)$ because the number of entries in the operations-table is $O(N^2)$. The same worst case time complexity applies to the `findRetestableTests()` algorithm.

The second loop of Algorithm 1 (lines 13-25) iterates through the class and activity diagram differences. In the worst case, (1) every statement is represented as a separate action node, in the activity diagram, (2) all original nodes are modified/deleted, (3) all transition flows between all

nodes are modified/deleted, (4) all constructors are modified/deleted, and (5) all fields in the class diagram are modified/deleted.

Let $n_{anodes}$ be the number of action nodes in the original program, $n_{flows}$ the number of transition flows between the action nodes, $n_{cons}$ the number of constructors in the class diagram, and $n_{fields}$ the number of fields in the class diagram.

The cost to call the other algorithms from Algorithm 1 is as follows:

| **Algorithm name** | **Complexity** |
|---|---|
| `findRetestableTestsTrans()` | $O(T * n_{flows})$ |
| `findRetestableTestsNodes()` | $O(T * n_{anodes})$ |
| `findRetestableTestsConstructors()` | $O(T * n_{cons})$ |
| `findRetestableTestsFields()` | $O(T * n_{fields} * n_{anodes})$ |

In the algorithm `findRetestableTestsTrans()`, the loop that iterates through all the test cases in the traceability matrix will execute $T$ times for each entry relevant to the transition flow (of which there are $O(n_{flows})$). For each test case, the traversed transition flows are stored in a HashSet. The algorithm that determines whether the HashSet contains the transition flow or not takes constant time. The complexity of the other algorithms can be explained in a similar manner.

The total worst case time complexity for Algorithm 1 is thus, $O(T * N^2))$ + $(T * (n_{flows}$ + $n_{anodes}$ + $n_{cons}$ + $(n_{fields} * n_{anodes}))))$, which can be approximated to $O(T * N^2$ + $T * (n_{fields} * n_{anodes}))$ because the number of flows, action nodes, and constructors is likely to be much smaller than the product $(n_{fields} * n_{anodes})$, and therefore neglectable.

Note that since we use Rational Software Architect (RSA) to adapt the models, the model differencing task is done by the tool. This cost is not being considered in our analysis.

Harrold et al.'s [69] DejaVu approach uses the algorithm proposed by Rothermel and Harrold [67]. This algorithm iterates through the control flow graphs of the original and modified versions of a program, and selects test cases that traverse modified edges. The worst case time

complexity for this algorithm is $O(2 * n + T * n^2)$ [67], where $T$ is the number of baseline test cases and $n$ is the total number of program statements.

The largest term in the time complexity of our algorithm is $T * n_{fields} * n_{anodes}$, and in the worst case, $n_{anodes}$ is the same as the number of program statements, $n$. Thus, this term becomes comparable to the largest term in the time complexity of Rothermel and Harrold's algorithm ($T*n^2$) when the total number of fields, $n_{fields}$ in a program is equal to the total number of statements, $n$, in the program. However, in practice, $n_{fields}$ is much smaller than $n$.

## 4.3 Limitations

MaRTS only supports the Java programming language because of the underlying toolset. RSA supports executing UML models with Java, and ReverseЯ supports reverse engineering models from annotated Java code. MaRTS can be extended to other object-oriented programming languages if the underlying tools are extended to support them, or replaced with analogous tools for the desired programming language.

The current version of MaRTS does not support multiple inheritance between classes because it was designed to analyze the inheritance hierarchy in the class models generated from Java. We can overcome this limitation by extending the operations-table to store information for multiple inheritance, and modify the test classification algorithm accordingly.

Due to limitations in the underlying reverse engineering and model execution tools, MaRTS does not support multithreaded programming, generic types, and new features introduced in Java 8-11. To integrate the unsupported features such as functional interfaces and default methods in Java interfaces in MaRTS, we need to extend the operations-table. We also need to extend the classification algorithm to identify the impact of changes to these features and classify test cases accordingly. The integration of such Java features in future versions of MaRTS is feasible assuming that the underlying tools can support the reverse engineering of these features and execute them at the model level.

MaRTS does not use associations and compositions in the UML class diagram due to a limitation in RSA. This tool always transforms an association with multiplicity more than one to a vector, while the actual data type can be different, e.g., `ArrayList`. Instead of using associations and compositions, MaRTS uses class attributes, where the constraints for compositions are specified in the constructors.

An activity diagram extracted by ReverseЯ contains one final node because ReverseЯ requires a method body to only have one return statement. To make the code compatible with ReverseЯ, we transform the implementation of each method that has multiple return statements to only have one return statement. Moreover, ReverseЯ requires annotated Java code. However, annotating the code takes extra effort and time from developers, and they do not tend to provide that in practice.

MaRTS requires UML activity diagrams that are complete and fine-grained, and it relies on reverse engineering to extract them from source code. Reverse engineering can be based on static or dynamic analysis of the source code [99]. Both approaches are challenging because of the heterogeneity, complexity, and size of software applications [99–101]. The challenges to reverse engineering of models from source code are the scalability to large and complex systems, the lack of metamodels that support reverse engineering of different programming languages, and the inability to map all aspects of a certain programming language such as Java to elements in models [100–102].

The above challenges to reverse engineering create obstacles to the application of MaRTS in practice. Therefore, we investigated the use of activity diagrams developed during design-time. Unfortunately, as described in Section 1.2 of Chapter 1, behavioral diagrams are used less often than class diagrams, and even when behavioral diagrams are used, they tend to be incomplete and lack low level details that are needed to obtain the traceability links from test cases to model elements.

# Chapter 5

# Fuzzy Logic-based RTS Using Behavioral Diagrams

This chapter presents FLiRTS [103], a **F**uzzy **L**ogic-based **R**egression **T**est **S**election approach. We proposed FLiRTS to address the second limitation described in Section 1.2 of Chapter 1. The behavioral diagrams used in practice tend to be incomplete and represent aspects of a software system at a high level of abstraction, which also limits the applicability of MaRTS because it requires complete and fine-grained UML activity diagrams.

FLiRTS uses UML activity diagrams that model the behaviors of the system's operations at a high level of abstraction, and a UML sequence diagram that models 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. These activity diagrams are not executable. This level of abstraction prevents relating the existing test cases to the activity diagrams. FLiRTS is based on generating more detailed refinements from each provided activity diagram according to the provided sequence diagram. The refinements contain enough information to permit the identification of traceability links between the activity diagrams and the test cases. However, the obtained traceability links may be correct or incorrect depending on the generated refinements. FLiRTS uses fuzzy logic to address this uncertainty and classifies test cases as retestable or reusable.

## 5.1 Using Fuzzy Logic in RTS

The RTS process can be considered as a decision-making system [78]. The decision concerns the classification of the test cases as retestable or reusable. In such a system, we need to model the inputs, outputs, and their relations. If the RTS process that is applied within the system is based on abstract models that represent aspects of a software system at a high level of abstraction, then

the environment under which the RTS process is operating is uncertain and has the following three characteristics:

**Incompleteness of information.** The used models lack information regarding call/usage dependencies between model elements, which prohibits obtaining the traceability links from test cases to model elements. This incompleteness of the traceability information need to be addressed by the RTS system.

**Fuzziness of inputs**. To overcome the lack of the traceability information that is normally provided as an input to a model-based RTS approach, we need to derive from the used models other types of input variables to be provided to the RTS system to classify test cases. However, these variables and their values can carry considerable fuzziness because they do not determine with full certainty whether or not a test case traverses changed model elements. We explain through an example. Suppose that the input variable called *distance* represents, with respect to a test case, the minimum number of model elements that need to be traversed by the test case to reach a changed model element. Suppose that the distance is calculated according to static analysis of the information available in the abstract models. When the distance is low, then there is a high likelihood that the test case traverses changed elements, and when the distance is high, then the likelihood is low. However, the input domain of the distance variable is fuzzy because we are uncertain about what range of values is low or high. Is a low value 0-5 or 0-10? Can the ranges for low and high overlap? The uncertainty of an input variable and its input domain need to be addressed to enable correct classification of the test cases.

**Fuzziness of relations between inputs and outputs.** Because the input variables derived from the abstract models do not determine with full certainty whether or not a test case traverses changed model elements, the relations between these input variables and the retestable and reusable categories can be fuzzy/imprecise. Let us consider the distance input variable defined previously. If the distance of a test case exists within the range of values defined for low, does this indicate that the test case is reusable? If the ranges for high and low distances overlap, which values from the overlapping area indicate that the test case is reusable and which values indicate that the test

case is retestable? This fuzziness in the relations between the input and output variables need to be addressed.

We chose fuzzy logic to address these characteristics and enable RTS based on abstract models because of the following reasons. First, fuzzy logic can be used to address decision making under uncertainty and incomplete information. As stated by Zadeh [104], "fuzzy logic is a way to formalize the capability to reason and make decisions in an environment of uncertainty, incompleteness of information, and partiality of truth". Second, the fuzziness of an input variable and its input domain can be addressed in fuzzy logic by partitioning the input domain into pre-defined input fuzzy sets. An input value fits in each input fuzzy set with a certain degree of truth based on a membership function defined for the set.

Third, the fuzziness of the relations between the inputs and outputs is addressed in fuzzy logic using linguistic inference rules that model these relations in the form of if-then rules. These rules provide capabilities for reasoning and making decisions according to the provided fuzzy inputs.

Finally, a traditional RTS approach supports binary classification of test cases as retestable or reusable. However, the binary classification cannot be directly applied in the absence of information regarding traceability links of test cases. Fuzzy logic addresses this issue because it concerns computing the degree of truth that the output belongs to a certain category. If fuzzy logic is used in the RTS process, then each test case will be assigned truth values (e.g., output probabilities) for being reusable and retestable. The final classification of the test case is based on these values.

To sum up, a fuzzy logic system for model-based RTS could be applied by (1) deriving input variables from the models that represent the system under test and its test cases, (2) defining input fuzzy sets and membership functions for the input domain of each variable, (3) defining an output variable (i.e., test case category), and (3) defining a set of inference rules to model the relations between the inputs and outputs, and to enable reasoning and making decisions towards an output, i.e., classifying test cases as retestable or reusable.

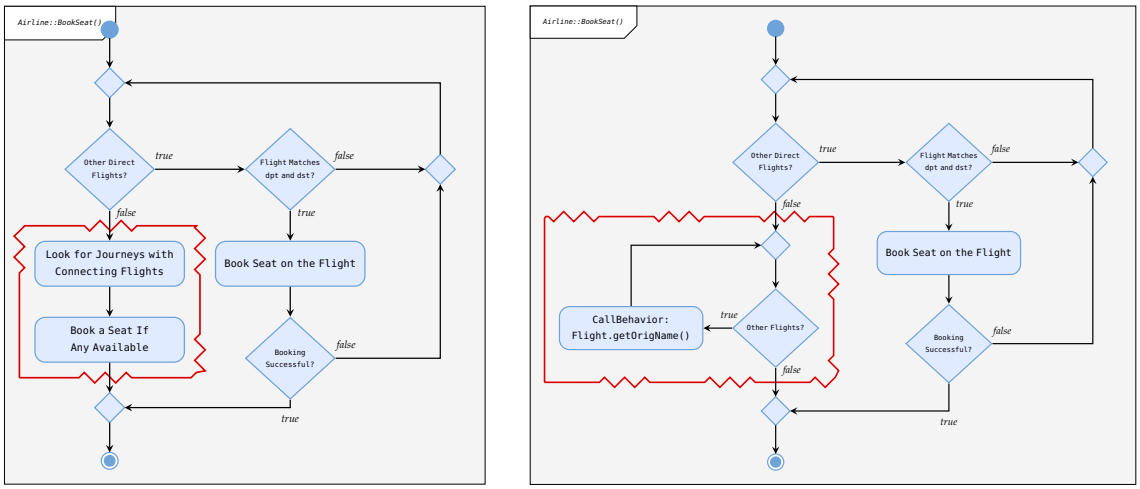
## 5.2 Proposed Approach

FLiRTS automatically generates more detailed activity diagrams called *refinements* from the provided activity diagrams that represent the system's methods at a high level of abstraction. 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 [4] 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.

The following subsections present a motivating example then describe the process of FLiRTS.

### 5.2.1 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 chapter, is a class project implemented by undergraduate students in a software engineering course. The portion of ARS used here supports only basic seat booking capabilities but not the ability to prioritize flights by prices, airlines, or other criteria. Fig. 5.1a shows an activity diagram of the `Airline.bookSeat()` method, which is at a high level of




**(a)** Original. **(b)** Refinement.

**Figure 5.1:** Activity Diagram Representing `Airline.bookSeat()` and One Possible Refinement.

abstraction. It lacks low level details, such as call behavior nodes to other activity diagrams. Fig. 5.2 shows 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. 5.1a 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. 5.2 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 dia-

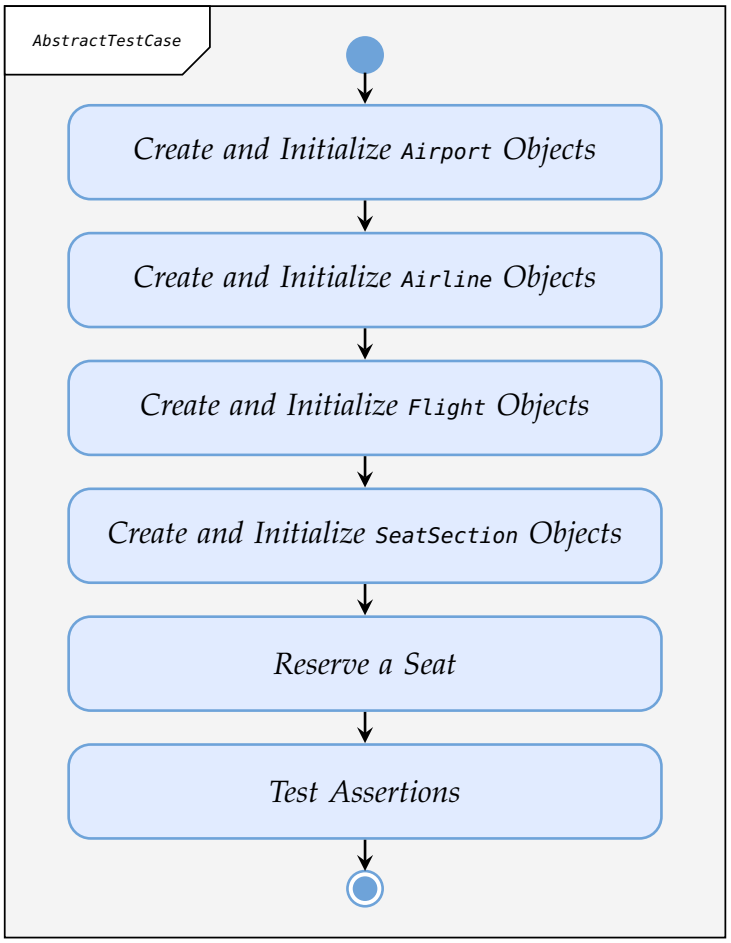



**Figure 5.2:** Activity Diagram Representing a Test Case.

gram 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.

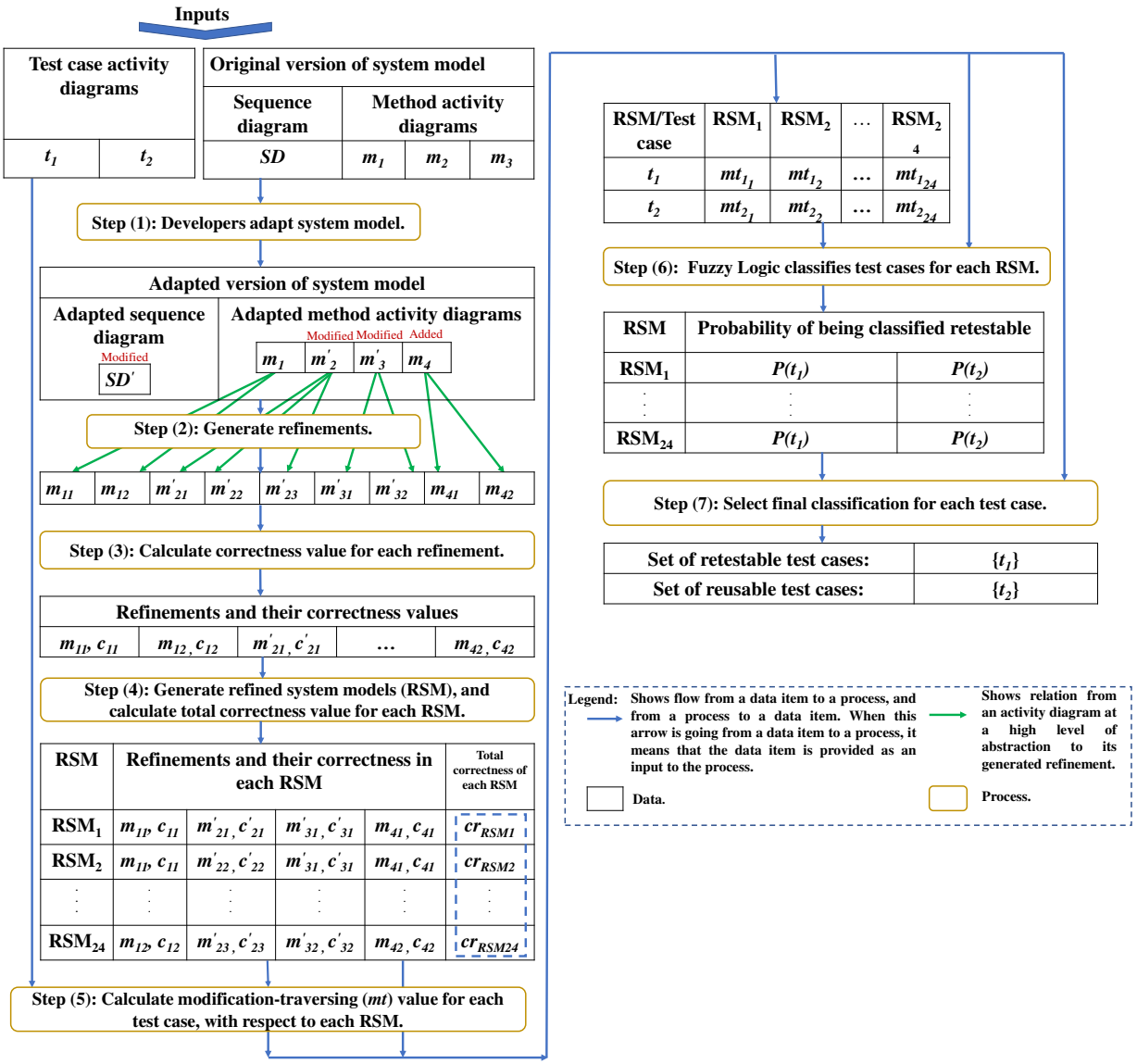



**Figure 5.3:** FLiRTS Process.

### 5.2.2 FLiRTS: Fuzzy Logic-based Regression Test Selection

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. 5.3 shows the main steps of FLiRTS.

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 in the correctness of the traceability links obtained from the refinements used in each refined system model.

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.

### 5.2.3 Generate Refinements of Activity Diagrams

A naive approach randomly generates 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.

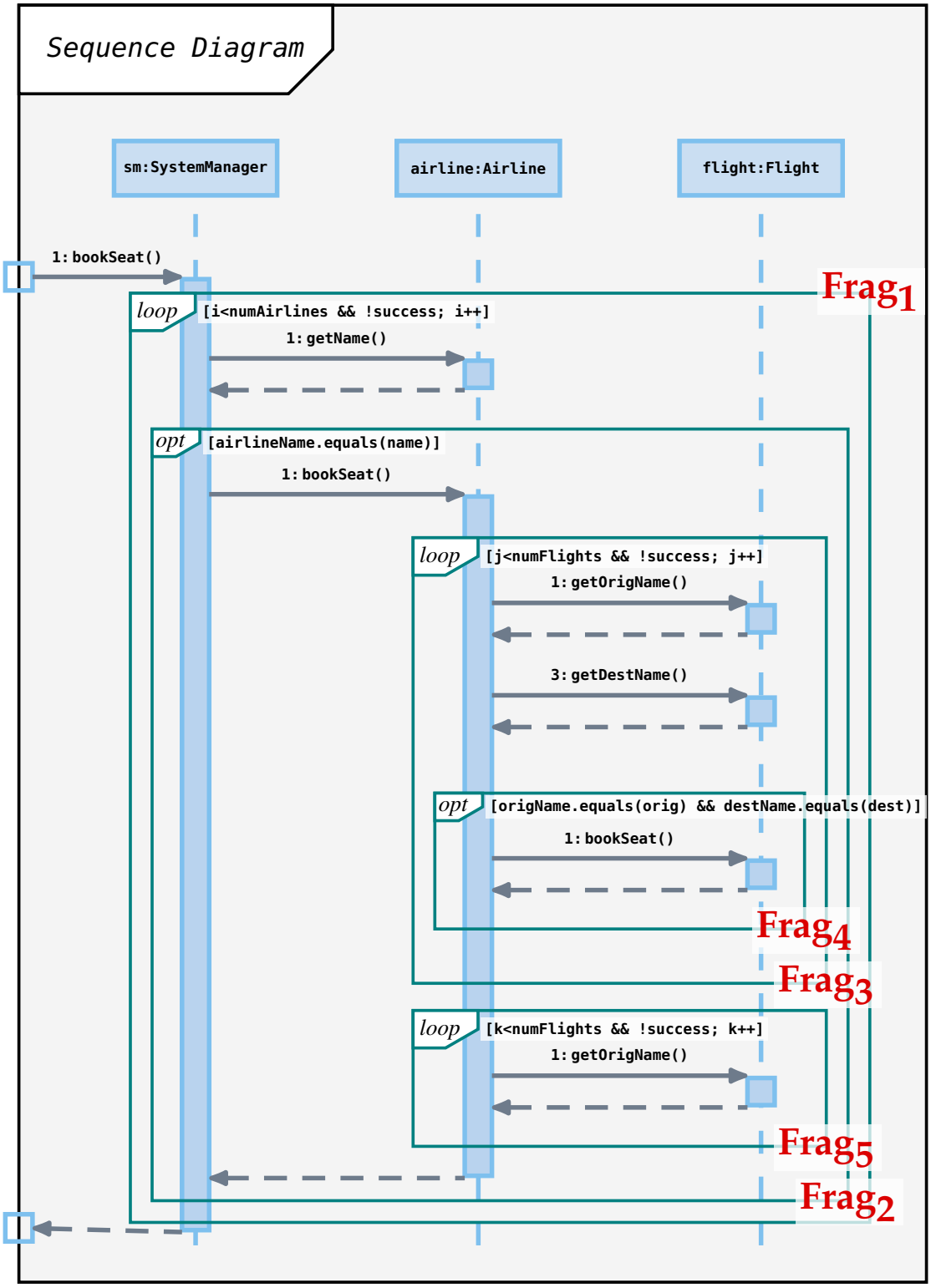



**Figure 5.4:** Partial Sequence Diagram.

To continue our running example, the sequence diagram shown in Fig. 5.4 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 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. 5.1a that represents `Airline.bookSeat()`, the partial sequence diagram presented in Fig. 5.4, 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 3:** `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.

```
1  R_e=∅
2  Set D_s = ∅;
3  S_e = getElementsFromSequenceDiagram(msg, sd);
4  A_e = getElementsFromActivityDiagram(ad);
5  ActivityDiagram ref = ad;
6  for each combined fragment c ∈ S_e do
7      for each decision node d ∈ A_e do
8          if d.nl = c.nl then
               /* nl refers to the element nesting level                    */
9              D_s.add((c, d));
10         end
11     end
12 end
13 R_e = createRefinements(ref, D_s);
14 for each refinement rf ∈ R_e do
15     for each message m ∈ S_e where m.nl=0 do
16         addCallBehaviorNode(rf, 0, m, NULL);
17     end
18 end
19 return R_e;
```

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. 5.1b), 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.

### 5.2.4 Prepare Inputs for Fuzzy Logic classifier

The first 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.

**Table 5.1:** Fuzzy logic inputs and outputs for test case $T_1$

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

The second 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.

In our running example, suppose that the detailed activity diagram of a test case $T_1$ contains a call behavior node that calls 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 $A_2$.

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. 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 $A_2$). Table 5.1 shows the input crisp values assigned to **cr** and **mt** for each refined system model that is traversed by $T_1$.

### 5.2.5 Apply the Fuzzy Logic Classifier

We first define the input fuzzy sets for the input variables **cr** and **mt**. The sets are *High*, *Medium*, and *Low*. Fig. 5.5 shows the input fuzzy sets defined for **cr**. Fig. 5.6 shows the input fuzzy sets

defined for **mt**. Each fuzzy set of **cr** represents a degree of the correctness of the refinements traversed by a test case. An input crisp value assigned to **cr** or **mt** can fit in each of the input fuzzy sets with a specific membership value. We also define an output variable called **testClassification** that has two output fuzzy sets called *Retestable* and *Reusable*.

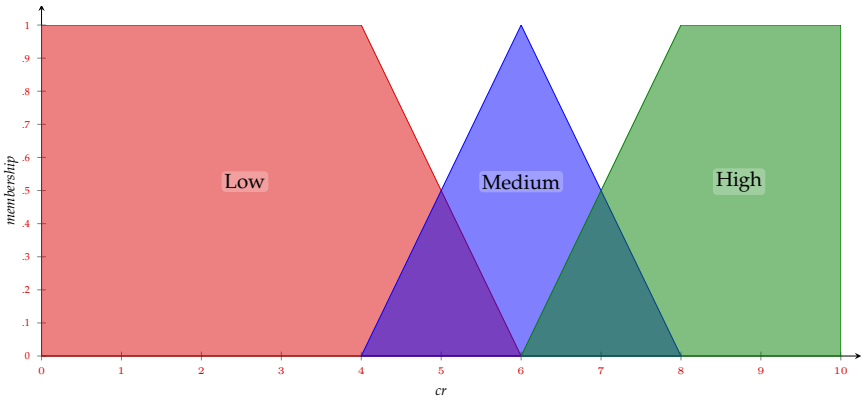



**Figure 5.5:** Fuzzy Sets for the Variable **cr**.

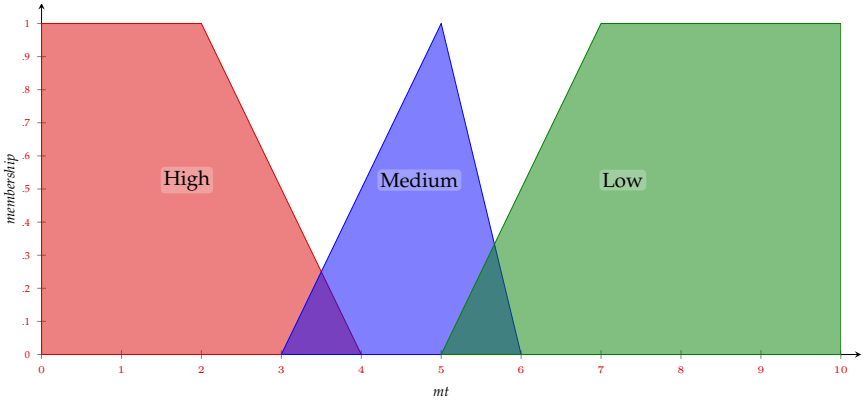



**Figure 5.6:** Fuzzy Sets for the Variable **mt**.

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 **cr** and **mt** 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 5.1), and its membership value is 0.7 for the *High* set and 0.3 for the *Medium* set (from Fig. 5.5).

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* $\wedge$ **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 5.1. 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 5.1, 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.

## 5.3 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: DejaVu [69] and MaRTS [105]. We used nine test cases in this study. The initial version of ARS does not support forming trips that involve connecting flights. We adapted the initial version at the model level and code level to support connecting flights. 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.

**Table 5.2:** 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% |

**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, 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%.

For a retestable test case, Table 5.2 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.

**Generalizability and Thresholds.** We cannot generalize the results from one study that used a small system, only nine test sets, and simple scenarios. We do not have a specific threshold for the probability value that can be used to choose between reusability and retestability.

**Safety of FLiRTS.** As mentioned in Section 5.2.3, 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.

## 5.4 Limitations

FLiRTS only uses behavioral diagrams. In practice, behavioral diagrams are used less often than structural diagrams as shown in Table 1.1, which limits the applicability of FLiRTS.

The process of FLiRTS does not scale up for real-world projects because it is based on generating large number of refined system models. If a system under test has $M$ methods represented as activity diagrams, and FLiRTS generated $N$ refinements from each of the $M$ activity diagrams, then the number of the refined system models will be $N^M$. This scalability issue limits the applicability of FLiRTS in practice.

# Chapter 6

# Fuzzy Logic-based RTS Using Class Diagrams

This chapter presents FLiRTS 2 that extends FLiRTS to address all the limitations described in Section 1.2 of Chapter 1. The behavioral diagrams are used less often than the class diagram as shown in Table 1.1, and even when both types of diagrams are used, the behavioral diagrams may be inconsistent with the structural diagrams or incomplete [41, 42]. Therefore, it is desirable to develop an RTS approach that *only* uses class diagrams given that this is the most commonly used diagram type. FLiRTS 2 extends FLiRTS by dropping the need for behavioral diagrams and focusing solely on UML class diagrams. FLiRTS 2 uses the generalization and realization relationships to address the identification of test cases that are affected by the changes to the inheritance hierarchy.

FLiRTS 2 requires as inputs a class diagram showing the design of the software system and its test classes, and the names of changed classes between the original and the adapted versions of the class diagram. Langer *et al.* [51] showed that class diagrams used in practice contain classes and interfaces, operation signatures and return types, generalization and realization relationships, and associations. FLiRTS 2 classifies test cases as retestable or reusable.

## 6.1 Proposed Approach

This section provides an overview of FLiRTS 2 (Sect. 6.1.1), and a description of its main steps (Sect. 6.1.2) and the process used to tune the fuzzy logic system and FLiRTS 2 parameters (Sect. 6.1.3).

### 6.1.1 FLiRTS 2 Overview

FLiRTS 2 follows the current trend in recent code-based RTS research [10,54,73] that considers a test class to be a test case. FLiRTS 2 classifies test classes as retestable or reusable according to the changes made to the UML class diagram representing the system under test.

FLiRTS 2 depends on the knowledge of which classes under test are directly invoked by each test class. These invocations are modeled in the class diagram as *call usage dependency* relationships [106]. The class diagram must contain the test classes, their *call usage dependencies*, and the classes under test. No other *usage dependency* relationships in the class diagram, such as the *create* and *send* dependencies, or usage dependencies with customized stereotypes are required. However, if provided, they are treated as *call usage dependencies*. Both unit and system test cases are supported.

FLiRTS 2 can be used with class diagrams that are customized using the UML-profile mechanism for a particular domain. A profile is defined using stereotypes, their tagged values, and constraints, which are applied to specific model elements, such as classes and operations. As described later in Sect. 6.1.2.1 and Sect. 6.1.2.2, FLiRTS 2 supports stereotypes, tagged values, and constraints, but they are optional.

FLiRTS 2 does not use any behavioral diagram. It also does not need static/dynamic call dependency information except for the one from the test classes. FLiRTS 2 constructs from the UML class diagram a graph, called *class relationships graph* (CRG), connecting the nodes representing the classes through edges representing the various relationships between them (e.g., associations and generalizations). It identifies from this CRG all the paths from the test classes to the adapted classes. These paths are used as a substitute for the traceability information used in the canonical RTS approaches to classify test classes. The class diagram cannot provide complete information about the real usage of the diagram elements and whether or not the identified paths are actually exercised. The class diagram provides a static view of the application and the paths are identi-

fied with varying degrees of confidence that the relationships are actually exercised. FLiRTS 2 addresses this uncertainty by adopting a probabilistic approach based on fuzzy logic.

The input crisp variables are based on the types of the relationships between the classes. The variables model (1) the probability that the execution of a test class traverses some adapted classes, and (2) the minimum distance from a test class to an adapted class. The input crisp values are calculated for each test class and used by the fuzzy logic system to calculate the probabilities of a test class belonging to the reusable and retestable output fuzzy sets. If the probability of a test class for being retestable is above a given threshold, then the test class is classified as retestable. Otherwise, the test class is classified as reusable.

The results of a fuzzy logic system depend on several factors that need to be carefully configured to get acceptable results. However, there are no general rules or guidelines for configuring these factors that are appropriate for every domain [91, 92]. We fine-tuned (Sect. 6.1.3) the fuzzy logic system through a controlled experiment that selected the best configuration by comparing the test classification results obtained by systematically varying the factors through several possible values.

### 6.1.2 FLiRTS 2 Process

The process consists of five steps that are all automated:

1. Building the CRG from the class diagram.
2. Marking adapted classes in the CRG.
3. Calculating the paths to the adapted classes.
4. Calculating the input crisp values.
5. Classifying the test cases.

We demonstrate these steps on a small example taken from Kapitsaki and Venieris [1] for a context-aware lecture reservation service provided in a university campus. The service is available to every faculty and visiting professor who wants to book a lecture room. The context-aware

service shows room availability only for those campus buildings close to the user's current location. The list of available rooms is based on weather conditions because the campus provides a number of open air facilities for lectures.

In the original design, the weather—implemented by the class `Weather`—could only assume the states *sunny*, *rain*, or *snow* without any relationship to the temperature. In the adapted design [1] shown in Fig. 6.1, the temperature concept is added for use with the *good weather* condition. The added elements were the classes `Temperature` and `DigitalThermometer`, an aggregation relationship from `Weather` to `Temperature`, a usage dependency relationship from `Temperature` to `DigitalThermometer`, and the necessary stereotypes and tag values.

The class diagram did not include test classes. We added two test classes `LectureReservationAppTest` and `LocationProviderAppTest` and these are shown in Fig. 6.1.

#### 6.1.2.1 Building the CRG from the Class Diagram

The CRG is a weighted directed graph extracted from the adapted class diagram. We need to capture the directions of the element types because a directed edge from node `A` to node `B` indicates a likelihood that the element represented by `A` calls operations in the element represented by `B`.

Each class/interface is represented as a node in the CRG. Weighted directed edges are added between the nodes in the CRG when the corresponding classes/interfaces in the class diagram are connected by one of the UML element types: association, generalization, realization, formal parameter and return types of operations, stereotype tagged values, and usage dependency relationships. Fig. 6.2 shows how these element types are mapped into directed edges in the CRG. OCL expressions may be used in the class diagram. However, no extra information is obtained regarding the relationships because the navigation between classes in the OCL expressions use the associations and operation parameters that we already use to add edges to the CRG.

FLiRTS 2 can identify the test classes that are impacted by the changes to the inheritance hierarchy because it uses the generalization and realization relationships in the CRG. If there is a generalization (or a realization) relationship from a class (or an interface) `B` to a class (or an inter-

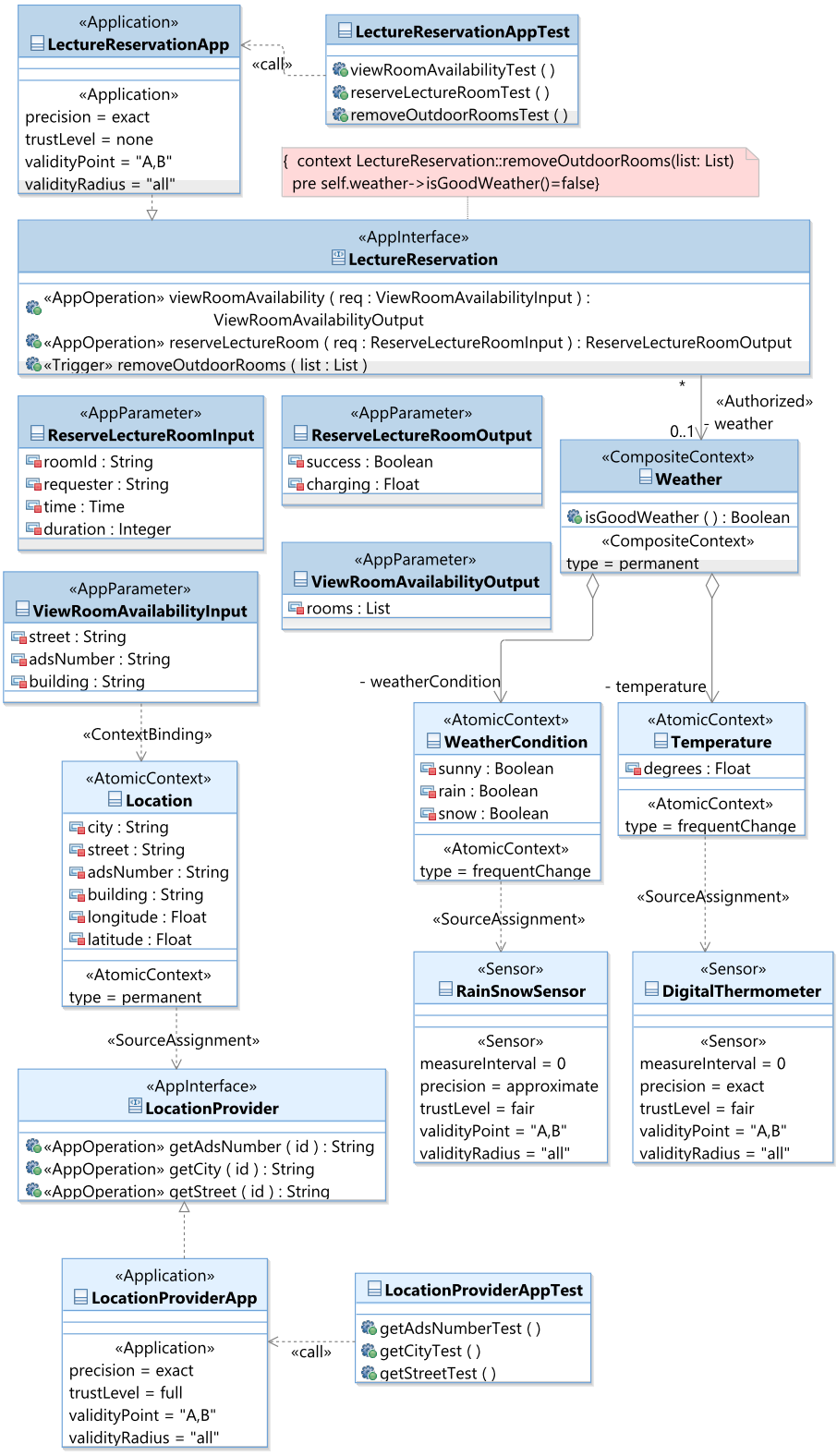



**Figure 6.1:** Partial Class Diagram after Adaptation [1].

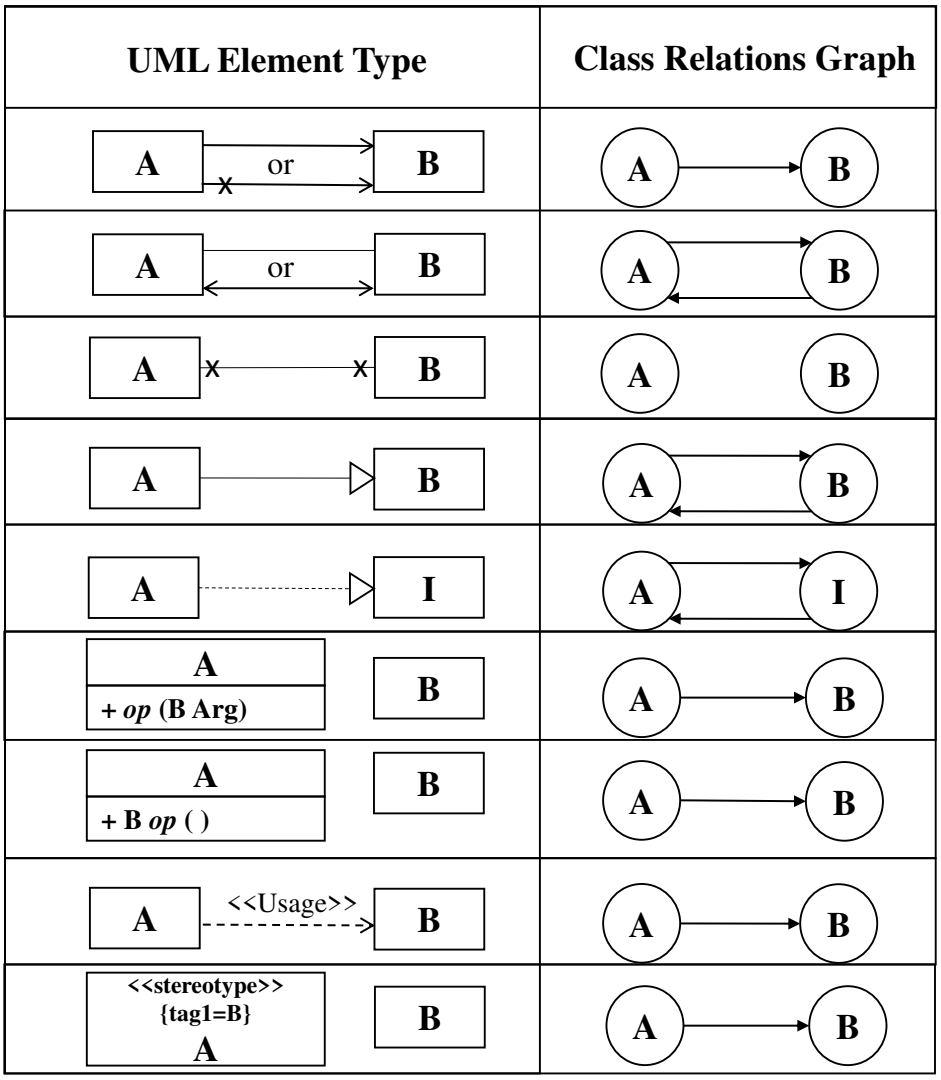



**Figure 6.2:** Mapping Rules from UML Element Types to CRG.

face) `A`, then two edges are added in the CRG from node `A` to node `B` and vice versa. Let us suppose that $N$ is the number of classes that are part of an inheritance hierarchy in the class diagram. A class `C` that belongs to this inheritance hierarchy will have a path to each of the other $N - 1$ classes that belong to the same inheritance hierarchy. If a test class `T` has a path to `C`, then `T` will also have a path to each of the other $N - 1$ classes along the inheritance hierarchy. If any of the $N$ classes is adapted, then `T` will have a path to the adapted class, and this path can be used to classify `T` as retestable.

Each element type implicitly has a different likelihood to be exercised by a test execution and therefore to drive the execution of some adapted class. Such a likelihood depends on several factors that are unpredictable. Each edge in the CRG has a weight that represents the likelihood. Each weight is a power of 2 from 0 to 6. For the associations shown in the first three rows of Fig. 6.2, we use the same weight but the association type determines the number and direction of the edges that are added to the CRG. Similarly, all types of *usage dependency* relationships are assigned an equal weight. The actual weight assigned to each element type is determined by the tuning process described in Sect. 6.1.3.

Only one directed edge *e* can be added from a node `A` to a node `B` in the CRG. The weight of the edge is:

$$\omega_e = \sum_{i=1}^{N} \omega_i$$

where $N$ is the number of the UML element types introducing a relationship from class `A` to class `B`, and $\omega_i$ is the weight assigned to the UML element type. For example, suppose that there are 3 associations and 1 generalization from class `A` to class `B` and that the weights 8 and 16 are assigned to the associations and generalizations respectively. Then, the weights of the edges from `A` to `B` and `B` to `A` are 40 (=8*3 + 16*1) and 16 (=16*1) respectively.

#### 6.1.2.2 Marking Adapted Classes in the CRG

This step requires the names of the adapted classes for labeling the corresponding CRG nodes as adapted. For example, in Fig. 6.3, the nodes `Weather`, `Temperature`, and `DigitalThermometer` are marked adapted because we added new relationships/elements to these classes.

The names of the adapted classes can be obtained in multiple ways. For example, model comparison techniques [107–109] can use the original and the adapted class diagrams to identify the adapted model elements (operation signatures, relationships, stereotypes and tagged values, and OCL constraints) and the classes and the interfaces containing these elements. The FiGA framework [38] automatically records the changes made to a model while it is being adapted. These changes include the names of the adapted classes. Briand et al. [7] suggest another approach

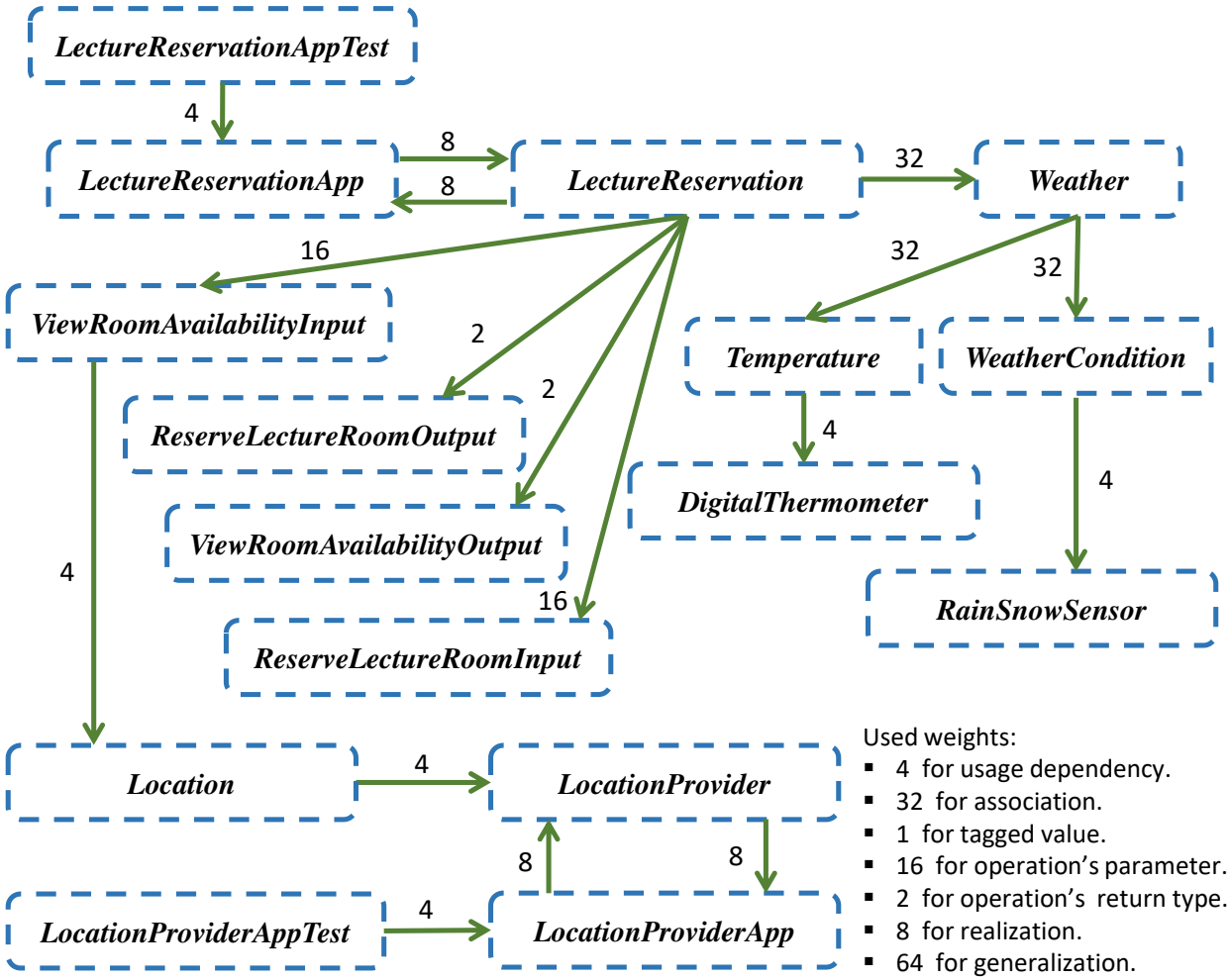



**Figure 6.3:** Extracted CRG.

where developers use stereotypes to indicate all the classes that they expect to be adapted. This approach addresses the case when fine-grained code modifications inside a method body are not visible in the class diagram. In this case, the developers add UML stereotypes to classes to indicate that operation implementations in these classes will be modified. FLiRTS 2 reads the stereotypes to get the names of these classes and flag them in the CRG as adapted.

#### 6.1.2.3 Calculating Paths to the Adapted Classes

The input crisp values in FLiRTS 2 are computed w.r.t. each test class. Therefore, for each test class $t$, we find in the CRG a set of paths $P_t$ that is used to compute the input values.

We define $P_t$ of a test class $t$ as the set of all simple paths (i.e., paths that do not contain cycles), such that each path starts from $t$ and ends in an adapted node without passing through any other adapted nodes. The reason for stopping at the first adapted node in each calculated path is that we use these paths (1) to calculate the probability that the execution of the test class reaches some adapted class and (2) to find the shortest path from the test class to an adapted class. In both cases, it is sufficient to reach the first adapted node in a path.

#### 6.1.2.4 Calculating Input Crisp Values

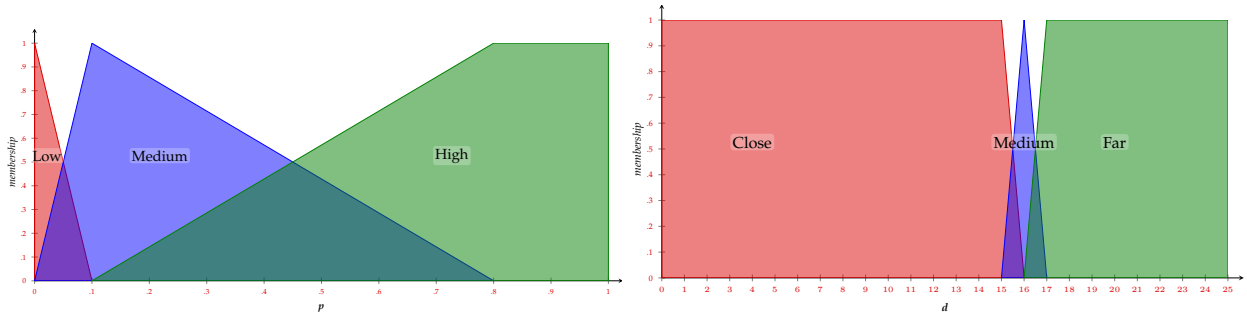
Two input crisp variables **p** and **d** are used. The probability that the execution of a test class traverses some adapted classes is represented by **p**, which considers the weights assigned to the UML element types in the construction of the CRG. The shortest distance from a test class to an adapted class is represented by **d**, where the shortest distance is the minimum number of edges in the CRG that must be exercised by a test class to reach an adapted class.

Given a test class $t$, and the set $P_t$ defined in Sect. 6.1.2.3, the value of **p** w.r.t. $t$ is zero when $P_t$ is empty. Otherwise,

$$\mathbf{p} = \sum_{\rho \in P_t} \prod_{i=1}^{|\rho|} \left(\frac{\omega_i}{\Omega}\right)$$

where $|\rho|$ is the path length, $\omega_i$ is the weight associated with the edge $i$, and $\Omega$ is the sum of the weights of all the edges with the same source node as $i$. The higher the value of **p** for $t$, the higher the probability that the execution of $t$ traverses some adapted classes. In the CRG of Fig. 6.3, there is only one simple path starting from the test class `LectureReservationAppTest` that ends in an adapted class (`Weather`). The value of **p** for `LectureReservationAppTest` is 0.42 (=(4/4)*(8/8)*(32/76)).

The value of **d** w.r.t. $t$ is the number of edges in the shortest path in $P_t$. When $P_t$ is empty, **d** is infinity. From Fig. 6.3, the values of **d** are 3 and infinity for `LectureReservationAppTest` and `LocationProviderAppTest` respectively. A smaller value of **d** indicates that the test class must traverse fewer class relationships to reach some adapted class, and thus, the test class has a higher likelihood of being retestable.




**(a)** Input fuzzy sets for **p** **(b)** Input fuzzy sets for **d**

**Figure 6.4:** Input Fuzzy Sets.

#### 6.1.2.5 Classifying the test cases

The classification process involves the fuzzification, inference, and defuzzification [89] phases of fuzzy-logic systems using the input crisp variables **p** and **d** and their input crisp values.

**Input fuzzy sets.** For variable **p**, the sets are *Low*, *Medium*, and *High*, as shown in Fig. 6.4a. For variable **d**, the sets are *Close*, *Medium*, and *Far*, as shown in Fig. 6.4b. The membership functions of the input fuzzy sets of **p** and **d** were chosen according to the tuning process described in Sect. 6.1.3. The values assigned to **p** and **d** fit in these sets with a specific membership value between zero and one.

**Output fuzzy sets.** An output crisp variable, **tc**, is used for test classification. Its output fuzzy sets are *Retestable* and *Reusable* as shown in Fig. 6.5. These sets were defined to be of equal size and their membership functions are trapezoidal. The functions map the output crisp value to probabilities for the test class being retestable and reusable. The boundaries for the *Reusable* set are $(0, 1)$, $(25, 1)$ and $(50, 0)$; those for the *Retestable* set are $(25, 0)$, $(50, 1)$ and $(75, 1)$.

**Fuzzification phase.** This phase calculates the membership values in the input fuzzy sets for input crisp values assigned to **p** and **d**. The input crisp value of **p** w.r.t. `LectureReservationAppTest` is 0.42. Using the fuzzy sets of **p** in Fig. 6.4a, the membership of 0.42 is zero in the *Low* set, 0.54 in the *Medium* set, and 0.46 in the *High* set. The input crisp value of **d** w.r.t. `LectureReservationAppTest` is 3. Using the fuzzy sets of **d** in Fig. 6.4b, the mem-

**Table 6.1:** Inference Rules

| p \ d | Close | Medium | Far |
|---|---|---|---|
| **Low** | Retestable | Reusable | Reusable |
| **Medium** | Retestable | Retestable | Retestable |
| **High** | Retestable | Retestable | Retestable |

**Table 6.2:** Activating Inference Rules

| p \ d | $\mu_{Close}$**(3)=1** | $\mu_{Medium}$**(3)=0** | $\mu_{Far}$**(3)=0** |
|---|---|---|---|
| $\mu_{Low}$**(0.42)=0** | 0 | 0 | 0 |
| $\mu_{Medium}$**(0.42)=0.54** | 0.54 | 0 | 0 |
| $\mu_{High}$**(0.42)=0.46** | 0.46 | 0 | 0 |

bership of 3 is 1 in the *Close* set, zero in the *Medium* set, and zero in the *Far* set. Step (1) of Fig. 6.5 shows the fuzzification step for the input crisp values 0.42 and 3.

**Inference phase.** We apply Mamdani inference method described in Chapter 3. FLiRTS 2 uses nine inference rules to consider all the combinations for the input fuzzy sets of **p** and **d**. The antecedents of the rules contain the input fuzzy sets of **p** and **d**, and the consequents contain the output fuzzy sets of **tc**. Table 6.1 summarizes the inference rules. The rules are in conjunctive canonical form where the conditions for the input crisp variables are connected by the AND ($\wedge$) operator. Fig. 6.5 shows the application of 2 of the 9 rules.

The inference rules are evaluated as follows. First, each inference rule is activated by calculating the minimum of the fuzzy inputs of the input crisp values used in the rule. For example, in Fig. 6.5, the input crisp values used in *rule 1* are 3 and 0.42, and their *fuzzy input*s are 1 ($\mu_{Close}$**(3)=1**) and 0.54 ($\mu_{Medium}$**(0.42)=0.54**), respectively. *Rule 1* is activated by applying the minimum operation between 1 and 0.54 and the result of *rule 1* is 0.54 as shown in step 2 of Fig. 6.5. Table 6.2 shows the results of activating each of the 9 rules. Second, the minimum value calculated in each rule propagates to the output fuzzy set (i.e., *Retestable* or *Reusable*) used in the rule by truncating its respective membership function. The truncated function of the *Retestable* set is shown as the shaded area in Fig. 6.5.

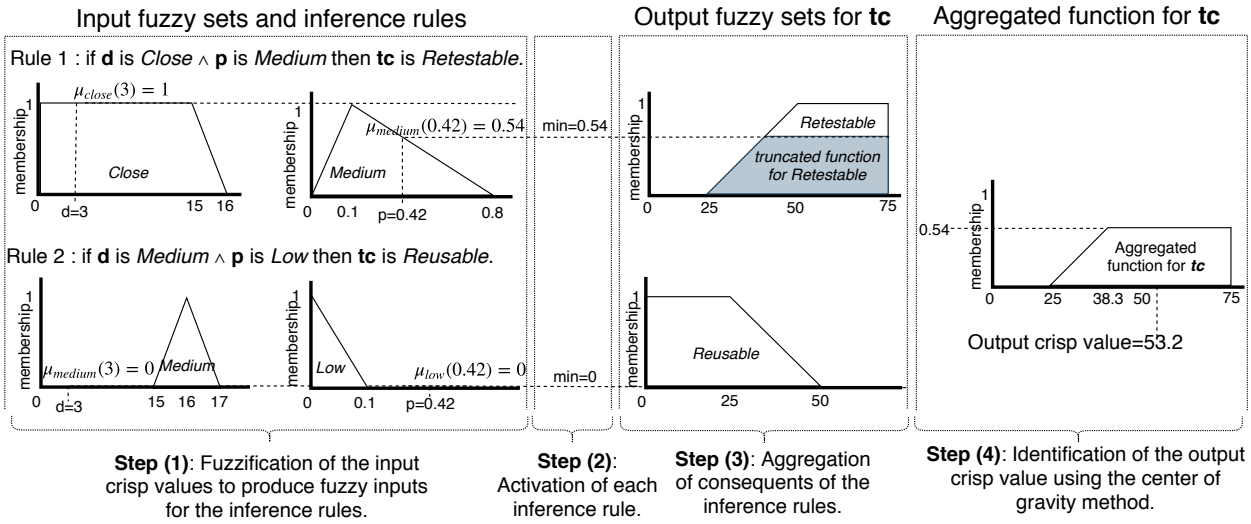



**Figure 6.5:** Fuzzification, Inference, and Defuzzification Phases.

After the evaluation of all the inference rules, their outputs need to be combined. The fuzzy union operator [90] is applied to the truncated membership functions to produce an aggregated membership function. The truncated functions shown in step (3) of Fig. 6.5 are aggregated to produce the aggregated membership function for the variable **tc**, shown in Step (4) of Fig. 3.4.

**Defuzzification phase.** The final *output crisp value* is found using the center of gravity of the aggregated membership function for **tc**. In our example, the defuzzification process returns 53.2 as the output crisp value as shown in Step (4) of Fig. 6.5. Finally, the output fuzzy sets, *Retestable* and *Reusable*, are used to map the *output crisp value* 53.2 to probabilities for *Retestable* and *Reusable*. The membership value of 53.2 in the *Retestable* set is 1, which means that the probability of `LectureReservationAppTest` being *retestable* is 100%.

The final classification of the test class is based on the probability of being *retestable* and a user-defined threshold. If the probability is above the threshold, the test class is classified as retestable; otherwise it is reusable. For example, if the threshold was 70%, then `LectureReservation-AppTest` would be classified as retestable.

### 6.1.3 Tuning FLiRTS 2

As should be evident from Sect. 6.1.2, a *reliable* test case classification result relies on the use of a proper configuration of the fuzzy system. A configuration is a triplet of (1) a function that assigns a weight to each UML relationship type, (2) the input fuzzy sets for the variables **p** and **d**, and (3) the selection threshold to classify a test class as retestable.

The result provided by FLiRTS 2 can be considered *reliable* when the resulting test classification shows the lowest safety and precision violations, and highest test suite reduction [72, 110] w.r.t. those achievable with other model-based or code-based RTS approaches on the same applications. In this respect, our term of comparison is Ekstazi [10]. Ekstazi is a code-based RTS approach known to be safe in terms of selecting all the modification-traversing test classes, widely evaluated on a large number of revisions, and being adopted by several popular open source projects; as such it can be considered the state-of-the-art for RTS tools.

The tuning process consists of finding the configuration $\bar{\mathbf{c}} \in \mathbf{C}$—the set of all possible configurations, or *configuration space*—that minimizes the safety and precision violations w.r.t. Ekstazi and maximizes the test suite reduction for a particular set of subjects **S** that we use as a *sample set*. Given $\mathbf{s} \in \mathbf{S}$, let $\mathbf{R_s}$ be the set of all revisions for **s**. Given $\mathbf{r} \in \mathbf{R_s}$, let $\mathbf{E^{sr}}$ and $\mathbf{F_c^{sr}}$ be the set of test cases of the revision **r** selected by Ekstazi and FLiRTS 2 with a configuration $\mathbf{c} \in \mathbf{C}$ respectively. Safety ($\mathbf{SV_c^{sr}}$) and precision ($\mathbf{PV_c^{sr}}$) violations and test suite reduction ($\mathbf{TR_c^{sr}}$) for FLiRTS 2 with the configuration $\mathbf{c} \in \mathbf{C}$ w.r.t. Ekstazi on the subject $\mathbf{s} \in \mathbf{S}$ and revision $\mathbf{r} \in \mathbf{R_s}$ are calculated as:

$$\mathbf{SV_c^{sr}} = \frac{|\mathbf{E^{sr}} \setminus \mathbf{F_c^{sr}}|}{|\mathbf{E^{sr}} \cup \mathbf{F_c^{sr}}|} \quad \mathbf{PV_c^{sr}} = \frac{|\mathbf{F_c^{sr}} \setminus \mathbf{E^{sr}}|}{|\mathbf{E^{sr}} \cup \mathbf{F_c^{sr}}|} \quad \mathbf{TR_c^{sr}} = \frac{|\mathbf{T^{sr}}| - |\mathbf{F_c^{sr}}|}{|\mathbf{T^{sr}}|}$$

where $\mathbf{T^{sr}}$ is the test suite for the revision **r** of the subject **s** before RTS is performed. The values of $\mathbf{SV_c^{sr}}$, $\mathbf{PV_c^{sr}}$, and $\mathbf{TR_c^{sr}}$ are multiplied by 100 to make them percentages. Lower percentages for $\mathbf{SV_c^{sr}}$ and $\mathbf{PV_c^{sr}}$, and higher percentages for $\mathbf{TR_c^{sr}}$ are better. $\mathbf{SV_c^{sr}}$, $\mathbf{PV_c^{sr}}$ and $\mathbf{TR_c^{sr}}$ can be used to define, for each configuration **c**, the set of their average values over the revisions in $\mathbf{r} \in \mathbf{R_S}$ for each

$\mathbf{s} \in \mathbf{S}$ as

$$\mathbf{V_c} = \{\langle \mathbf{A\text{-}SV_c^s}, \mathbf{A\text{-}PV_c^s}, \mathbf{A\text{-}TR_c^s} \rangle | \forall \mathbf{s} \in \mathbf{S}\}$$

where

$$\mathbf{A\text{-}SV_c^s} = \underset{\mathbf{r} \in \mathbf{R_S}}{\texttt{average}}\, \mathbf{SV_c^{sr}}$$

$$\mathbf{A\text{-}PV_c^s} = \underset{\mathbf{r} \in \mathbf{R_S}}{\texttt{average}}\, \mathbf{PV_c^{sr}}$$

$$\mathbf{A\text{-}TR_c^s} = \underset{\mathbf{r} \in \mathbf{R_S}}{\texttt{average}}\, \mathbf{TR_c^{sr}}.$$

They are also used to define for each subject $\mathbf{s} \in \mathbf{S}$, their optimum value over all the configurations in $\mathbf{C}$ as

$$\mathbf{opt} = \{\langle \underset{\mathbf{c} \in \mathbf{C}}{\texttt{min}}\, \mathbf{A\text{-}SV_c^s}, \underset{\mathbf{c} \in \mathbf{C}}{\texttt{min}}\, \mathbf{A\text{-}PV_c^s}, \underset{\mathbf{c} \in \mathbf{C}}{\texttt{max}}\, \mathbf{A\text{-}TR_c^s} \rangle | \forall \mathbf{s} \in \mathbf{S}\}.$$

The best configuration $\overline{\mathbf{c}}$ is the one that shows the minimal distance from the optimum values for safety and precision violations and test suite reduction.

$$\exists \overline{\mathbf{c}} \in \mathbf{C} \text{ s.t. } \underset{\mathbf{c} \in \mathbf{C}}{\texttt{min}}\, \texttt{Distance}(\mathbf{opt}, \mathbf{V_c})$$

To ensure that is $\overline{\mathbf{c}}$ the best configuration, we repeated the tuning process using the median instead of the average computation. In both cases, we computed the distance using *Manhattan*, *Chebyshev*, *Euclidean*, and *Canberra* distances [111].

Sect. 6.1.3.1 presents the subject applications used in the tuning process. Sect. 6.1.3.2 describes the process of calculating the configuration space. We discuss issues about the tuning process and threats to validity reported in Sect. 6.1.3.3.

#### 6.1.3.1 Subject Applications for Tuning FLiRTS 2

Even though FLiRTS 2 is a model-based approach, we are forced to tune it using code-based subjects because of the lack of large open-source model-based subjects and other model-based

RTS tools. We used the 8 subjects listed in Table 6.3. These are selected from the 21 open-source Java projects in Table 6.4, which are known to be compatible with Ekstazi since they were used in its evaluation [10, 72].

We downloaded the revisions of every subject using the methodology in Legunsen *et al.* [72]. From the earliest revision to the latest $\overline{\textbf{SHA}}$ listed in Table 6.3, we chose all those revisions that (1) correctly compiled, and (2) for which the tests and Ekstazi compiled and ran successfully.

**From source code to models.** The UML class diagrams for selected subjects were automatically extracted from each revision by using the Java to UML transformation plugin for the Rational Software Architect (RSA) [52]. The extracted models contain classes, interfaces, operations with input parameters and return types, associations, generalizations, and realizations. They do not contain usage dependency relationships because these are not supported by the RSA transformation plugin. Thus, the diagrams do not contain the information regarding the direct invocations from the test classes to the classes under test. We obtained this information from the corresponding code-level revision using `Apache Commons BCEL` [112], which supports static analysis of binary Java class files, and stored it in a text file. The text files were provided along with the corresponding class diagrams as inputs to FLiRTS 2. FLiRTS 2 supports reading the information regarding the direct invocations from the class diagram as well as from a separate text file when the information is not represented in the class diagram.

**Marking adapted classes.** As explained in Sect. 6.1.2, FLiRTS 2 needs the names of the adapted classes. However, EMF compare [109] and any other model comparison approaches do not work with the extracted models because each extracted model uses a different set of element *id*s and every comparison concludes that every model element has changed. Thus, we were forced to use the `git diff` command on the source code. The names of the modified classes/interfaces produced by `git diff` were used as an input to FLiRTS 2 to mark the adapted classes in the class diagram that was generated from the same source code.

**Selecting revisions.** Only revisions that differed from the previously selected one by more than the 3% of its Java classes were selected. The aim is to increase the chances of having multiple

**Table 6.3:** Selected Projects for the Tuning Process

| ID | $\overline{\textbf{SHA}}$ | $\overline{\textbf{A}_{\textbf{cl}}}$ | $\overline{\textbf{A}_{\textbf{ad}}}$ | $\overline{\textbf{Revs}}$ |
|---|---|---|---|---|
| $p_1$ | 4e5a699 | 222 | 66 | 223 |
| $p_2$ | 0dad342 | 510 | 58 | 41 |
| $p_3$ | c873192 | 375 | 58 | 85 |
| $p_4$ | a3b01f4 | 471 | 43 | 163 |
| $p_{13}$ | 792da67 | 158 | 10 | 407 |
| $p_{15}$ | db51a1c | 85 | 7 | 318 |
| $p_{16}$ | c66efa9 | 53 | 6 | 118 |
| $p_{20}$ | 14d7643 | 38 | 4 | 122 |

code changes between the selected revisions and to reduce the cases where the changes do not involve code modifications (e.g., only changes to comments).

We selected 8 subjects, 4 with the smallest and 4 with the largest average number of changed classes shown with a gray background in Table 6.3. This choice permits us to tune FLiRTS 2 to work with both highly variable subjects (e.g., systems that introduce new functionality) and mostly stable subjects (e.g., systems whose changes are due to bug fixing). Table 6.3 shows for each subject, the latest revision used ($\overline{\textbf{SHA}}$), the number of used revisions ($\overline{\textbf{Revs}}$), and among all the used revisions, the average number of classes ($\overline{\textbf{A}_{\textbf{cl}}}$), and the average number of adapted classes ($\overline{\textbf{A}_{\textbf{ad}}}$).

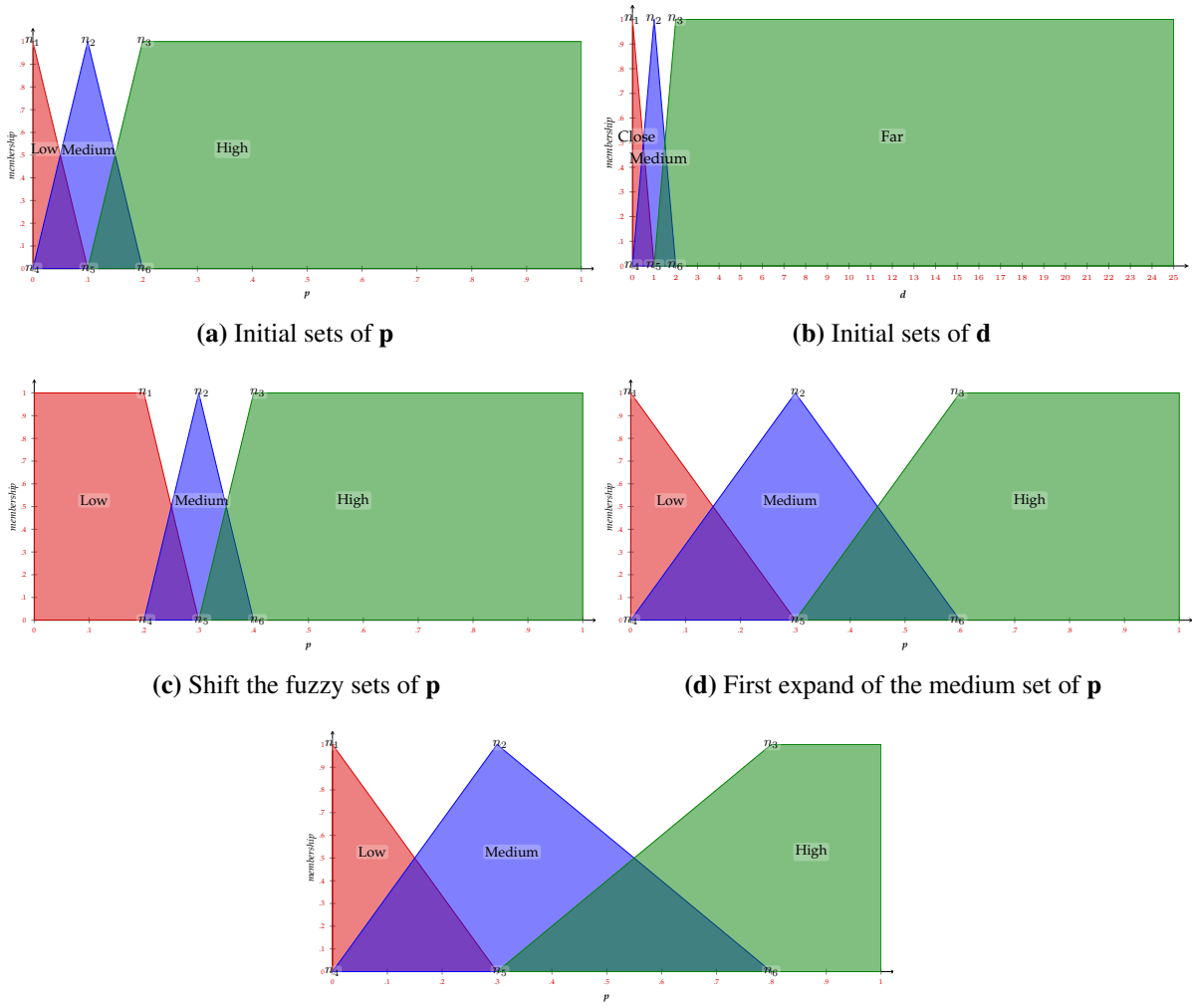
#### 6.1.3.2 Selecting the Best Configuration

The *configuration space* is the set of the combinations of values that can be assumed by the 3 elements in the triplet

$$\langle \textit{weight assignment}, \textit{input fuzzy sets}, \textit{selection threshold} \rangle.$$

We now describe how these combinations were calculated.

**Weight Assignment.** This function assigns a weight among 1, 2, 4, 8, 16, and 32 to the considered UML element types: association, generalization, realization, formal parameters of operations, return types of operations, and usage dependency relationships. We did not consider




**(a)** Initial sets of **p**

**(b)** Initial sets of **d**

**(c)** Shift the fuzzy sets of **p**

**(d)** First expand of the medium set of **p**

**(e)** Second expand of the medium set of **p**

**Figure 6.6:** Initial and Generated Input Fuzzy Sets.

tagged values because the class diagrams generated from the code-level revisions do not use UML profiles and do not contain tagged values. Because there are 6 weights for 6 element types, there are 720 (=6!) weight assignment functions.

**Input fuzzy sets.** The input fuzzy sets for the two input crisp variables **p** and **d** were generated by applying *shift* and *expand* operations to the initial sets. All generated fuzzy sets have triangular/trapezoidal shapes because these are proven to produce good results for most of the domains [91,92]. Fig. 6.6a and Fig. 6.6b show the initial sets for **p** and **d**, and the points for the fuzzy sets from $n_1$ through $n_6$. We apply the *shift* and *expand* operations to these points.

The input values range between 0 and 1 for **p** and 0 and 25 for **d**. We explored the input values of **d** of all the test classes of all the revisions of the eight sample subjects, and the largest value of **d** was never higher than 25. Therefore, we applied the *shift* and *expand* operations on the input fuzzy sets of **d** until we reached the value 25. The values above 25 will always remain in the *Far* set in all the generated fuzzy set configurations. We applied *shift* and *expand* by a given amount $\varepsilon$, where $\varepsilon$ was chosen to be one-fifth of the range of **p** (0-1) and **d** (0-25). Thus, the value of $\varepsilon$ was 0.2 for **p** and 5 for **d**.

The generation process starts from the initial values and repeatedly applies the shift operation using the amount $\varepsilon$ to the right on all the sets. To shift by $\varepsilon$ means to add $\varepsilon$ units to each point from $n_1$ to $n_6$ in each set. We stop shifting when the *Medium* set reaches the right boundary. This happens when the points on the right, $n_3$ and $n_6$, are clipped at the right boundary. Fig. 6.6c shows how the initial fuzzy sets of **p** were shifted.

Each application of the *shift* operation generates a possible group of fuzzy sets. Each group is further refined by repeatedly applying an *expand* operation to the *Medium* set. Specifically, the point $n_4$ of the *Medium* set is moved to the left by $\varepsilon$, and the point $n_6$ of the *Medium* set is moved to the right by $\varepsilon$. The other sets are modified similarly. We repeatedly apply the expand operation until the left points $n_1$ and $n_4$ reach the left boundary and the right points $n_3$ and $n_6$ reach the right boundary. Fig. 6.6d and Fig. 6.6e show the result of two applications of the *expand* operation to the *Medium* set of Fig. 6.6c. In the first expand (Fig. 6.6d), we moved the four points $n_1$, $n_4$, $n_3$, and $n_6$ . In the second expand (Fig. 6.6e), we only moved $n_3$ and $n_6$ but not $n_1$ and $n_4$ because they already reached the left boundary in the first expand.

We generated 21 fuzzy set groups each for **p** and **d**. The fuzzy logic system uses one fuzzy set group for **p** and one for **d** together in a function block, which is a primitive object that contains the input and output variables and their fuzzy sets, and the inference rules [87]. Therefore, the total number of generated combinations was 21*21=441.

**Selection threshold.** We considered 5 possible values for the selection threshold: 50%, 60%, 70%, 80%, and 90%.

**Configuration space.** The total number of the configurations used to tune FLiRTS 2 was 720*441*5=1,587,600.

**Best configuration.** We ran Ekstazi on the selected subjects and FLiRTS 2 on the corresponding model inputs with each configuration from the calculated configuration space. The total number of FLiRTS 2 runs was 2,344,885,200 (=1,587,600 configurations * 1,477 revisions of the sample subjects). For every configuration, subject, and revision of a subject, we calculated the safety and precision violations w.r.t. Ekstazi, and test suite reduction by using the formulas described in Sect. 6.1.3. Since we use median and average along with four distance measures, we obtained 8 different candidates for the best configuration.

The best configuration used Manhattan distance with the average-based tuning process. This configuration uses the weights 2 for association, 1 for realization, 32 for generalization, 4 for return type, 16 for input parameter, and 8 for usage dependency. The input fuzzy sets for this configuration are shown in Fig. 6.4, and the selection threshold is 50%. This configuration resulted in the lowest safety violation among all the 8 configurations. We use safety violation as our final selection criterion because we consider safety to be more important than precision and test suite reduction. Using this configuration for each of the selected subjects, the median safety violation was zero and the average safety violation was less than 5. Using the other 7 configurations with the sample subjects produced a median safety violation that was higher than zero for 4 sample subjects and an average safety violation higher than 5 for 7 subjects.

#### 6.1.3.3 Discussion

FLiRTS 2 must be tuned only once for a new context and objective for which it is used. For example, we gave a higher priority to safety than to test suite reduction. Practitioners who have different objectives, such as higher test suite reduction when regression testing time is limited as in continuous integration environments with frequent commits, will need to re-tune FLiRTS 2 to achieve higher test suite reduction. Re-tuning is also required if the approach is used for models

where UML profiles and tagged values are used. In our tuning approach we did not use models that have tagged values, and thus, we did not use weights for them.

Below we present threats to validity that can affect the outcome of the tuning process.

**External validity.** We used one criterion to select the subjects to train FLiRTS 2 based on the largest/smallest number of changed classes. These subjects may not be representative, so we cannot generalize the tuning result. Moreover, selecting the subjects based on other criteria such as the subject size, number of test cases, and application domain could impact the tuning outcome. Different application domains can involve other types of model-level changes, such as changes to OCL expressions, stereotypes, and tagged values, which are used to identify adapted classes in the class diagram. However, this threat is reduced because the selected subjects vary in size, number of revisions, and number of test classes.

**Internal validity.** We reverse engineered the class diagrams from code-level revisions. The generated associations were all directed, i.e., the class diagrams did not include the other types of associations shown in Fig. 6.2. Moreover, the generated diagrams did not include some design information such as tagged values of stereotypes and OCL expressions. Although providing this information is optional for FLiRTS 2, having it could impact the outcome.

We used the `git diff` command between the code-level revisions to identify the adapted classes. Applying model comparison between the class diagrams could identify fewer adapted classes because some code-level changes may not be detectable at the model level. We extracted the direct invocations from test classes to classes under test from the code-level revisions. These deviations from FLiRTS 2, which should only use model-level information, could introduce errors and change the results.

The best configuration we found is a local optimum with respect to the configuration settings used in the tuning process. Using different weights, membership functions for the fuzzy sets, selection thresholds, and distance measures could impact the outcome.

Other factors that can affect the outcome are errors in the FLiRTS 2 implementation. To reduce this threat, we built our approach on mature tools (i.e., JGraphT [113] and jFuzzyLogic [87]) and tested it.

**Construct validity.** We chose Ekstazi as the ground truth against which to tune FLiRTS 2. However, Ekstazi may not represent the ground truth for all RTS, and tuning FLiRTS 2 with respect to other RTS approaches could impact the tuning outcome. However, Ekstazi is safe, and is similar to FLiRTS 2 in terms of supporting class-level RTS.

There are other techniques that can be used to tune fuzzy logic systems, such as genetic algorithms, which we did not consider in this work. We plan to investigate such techniques in the future.

**Conclusion validity.** We only used 8 subjects to tune FLiRTS 2. The use of additional subjects could impact the outcome of the tuning process.

## 6.2 Evaluation

The evaluation goals were to compare FLiRTS 2 with other RTS tools in terms of (1) safety violation, (2) precision violation, (3) test suite reduction, and (4) fault detection ability of the obtained reduced test suites. The terms safety violation, precision violation, and test suite reduction are defined in Sect. 6.1.3. The fault detection ability was compared by using mutation testing on the subjects using the full test suites and the reduced test suites obtained by the RTS tools.

The empirical study is driven by the following Research Questions (RQ):

RQ1: What is the safety violation of FLiRTS 2 with respect to *state-of-the-art* dynamic and static RTS approaches?

RQ2: What is the precision violation of FLiRTS 2 with respect to *state-of-the-art* dynamic and static RTS approaches?

RQ3: What is the reduction in test suite size achieved by FLiRTS 2?

RQ4: What is the difference between the fault detection ability of the reduced test sets achieved by FLiRTS 2 and the fault detection ability of the full test sets and the reduced test sets achieved by *state-of-the-art* dynamic and static RTS approaches?

It was not possible to compare FLiRTS 2 with other model-based RTS approaches because neither their tool implementations (e.g., [7, 14]), nor the models used in the reported studies are available. Moreover, some RTS approaches use behavioral diagrams (e.g., sequence and activity diagrams) and it is difficult to extract them from the thousands of source code revisions used in our study. As an example, the state of the art model-based RTS approach [7] supports diagram types represented using old version of UML (i.e., UML 2.0), and assumes that each use case is associated with a sequence diagram and that the sequence diagrams refer to each other using interaction use (identified by the *ref* keyword). It is not possible to create these detailed diagrams from the code-level revisions. To the best of our knowledge, there is no freely available repository that contains subjects with test cases and design models for several of their revisions.

Therefore, we use two code-based RTS tools, Ekstazi [10] and STARTS [54]. They are both *state-of-the-art* and have been widely evaluated on a large number of revisions of real world projects [72]. Ekstazi, STARTS, and FLiRTS 2 use class-level RTS, i.e., they all identify changes at the class level and select every test class that traverses or depends on any changed class. Ekstazi uses dynamic analysis and STARTS uses static analysis of compiled Java code.

### 6.2.1 Experimental Setup

We evaluated FLiRTS 2 using the 21 subjects listed in Table 6.4. The revisions were selected using the same method described in Sect. 6.1.3.1 with an additional criterion that the subject should also run with STARTS on the chosen revisions. Moreover, we relaxed the constraint of the number of changes between a revision and its successor from greater than 3% of the classes to at least one class. The total number of revisions selected from the 21 subjects was 8,060. Table 6.4 shows the number of the selected revisions for each subject. We automatically generated the UML class diagrams from the 8,060 revisions and identified the adapted classes using the method described in

**Table 6.4:** Subjects

| ID | Subject Name | SHA | Revs | $A_{ad}$ | $A_{tc}$ |
|---|---|---|---|---|---|
| $p_1$ | commons-net | $4e5a699$ | 905 | 2.63 | 38.95 |
| $p_2$ | commons-collections | $0dad342$ | 235 | 10.36 | 241.56 |
| $p_3$ | commons-imaging | $d2ec76b$ | 217 | 5.63 | 85.51 |
| $p_4$ | asterisk-java | $e36c655$ | 478 | 7.93 | 39.52 |
| $p_5$ | commons-jxpath | $eff47ab$ | 42 | 7.64 | 49.14 |
| $p_6$ | commons-configuration | $20fed44$ | 429 | 3.56 | 163.63 |
| $p_7$ | commons-lang | $809e2be$ | 540 | 2.05 | 152.83 |
| $p_8$ | commons-io | $58b0f79$ | 629 | 4.59 | 95.33 |
| $p_9$ | commons-cli | $18f8576$ | 216 | 2.81 | 29.0 |
| $p_{10}$ | commons-validator | $8d0b6a1$ | 329 | 2.76 | 68.57 |
| $p_{11}$ | commons-text | $5ee9331$ | 336 | 3.7 | 35.92 |
| $p_{12}$ | commons-dbcp | $5d46f24$ | 468 | 3.61 | 44.72 |
| $p_{13}$ | commons-compress | $5fd497f$ | 1387 | 2.82 | 73.73 |
| $p_{14}$ | commons-pool | $d4e0e88$ | 495 | 2.52 | 21.12 |
| $p_{15}$ | commons-codec | $db51a1c$ | 679 | 2.13 | 41.48 |
| $p_{16}$ | commons-dbutils | $c66efa9$ | 178 | 3.07 | 23.74 |
| $p_{17}$ | HikariCP | $d1fbf7e$ | 44 | 3.02 | 2.27 |
| $p_{18}$ | stream-lib | $6e0edb5$ | 40 | 2.0 | 25.18 |
| $p_{19}$ | commons-fileupload | $b1498c9$ | 293 | 1.82 | 11.09 |
| $p_{20}$ | commons-email | $14d7643$ | 117 | 2.26 | 16.01 |
| $p_{21}$ | invokebinder | $ce6bfeb$ | 3 | 1.0 | 3.0 |

**SHA** is the latest available revision we used in the experiment.
**Revs** is the number of revisions per each subject.
$\mathbf{A_{ad}}$ ($\mathbf{A_{tc}}$) is the average number of adapted classes (test classes) among all classes (test classes) of all the revisions per each subject respectively.

Sect. 6.1.3.1. We ran Ekstazi and STARTS on the 8,060 code-level revisions, and FLiRTS 2 on the corresponding models using the best configuration obtained during the tuning process. Table 6.5 shows the RTS results in terms of safety violation, precision violation, and test suite reduction. Table 6.6 shows the fault detection ability results.

### 6.2.2 RQ1: Safety Violation

As shown in Table 6.5, the median safety violation is zero (or close to zero) for all the subjects. Fig. 6.7 shows that the average safety violation with respect to Ekstazi and STARTS was below 20% for 19 out of 21 subjects. Among all the 8,060 revisions, the average safety violation of

**Table 6.5:** RTS Results

| ID | A-SV$_e$ % | A-SV$_s$ % | M-SV$_e$ % | M-SV$_s$ % | A-PV$_e$ % | A-PV$_s$ % | M-PV$_e$ % | M-PV$_s$ % | A-R$_e$ % | A-R$_s$ % | A-R$_f$ % |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $p_1$ | 10.87 | 9.5 | 0.0 | 0.0 | 54.8 | 54.26 | 88.46 | 85.71 | 92.24 | 91.88 | 67.0 |
| $p_2$ | 6.01 | 10.94 | 0.0 | 0.0 | 73.66 | 67.63 | 99.3 | 99.2 | 95.75 | 91.97 | 62.91 |
| $p_3$ | 8.91 | 11.28 | 0.0 | 0.0 | 55.93 | 19.22 | 46.91 | 0.0 | 75.72 | 37.08 | 32.2 |
| $p_4$ | 8.93 | 3.34 | 3.7 | 0.0 | 74.04 | 70.65 | 83.66 | 80.0 | 87.6 | 85.15 | 51.53 |
| $p_5$ | 14.41 | 14.55 | 0.0 | 0.0 | 50.59 | 44.95 | 37.14 | 31.42 | 71.02 | 65.41 | 40.22 |
| $p_6$ | 29.69 | 16.87 | 1.4 | 0.79 | 47.63 | 44.66 | 51.26 | 37.01 | 80.06 | 73.93 | 45.72 |
| $p_7$ | 11.91 | 16.99 | 0.0 | 0.0 | 31.86 | 28.24 | 0.0 | 0.0 | 96.73 | 89.28 | 96.26 |
| $p_8$ | 9.97 | 11.82 | 0.0 | 0.0 | 40.99 | 36.71 | 22.22 | 5.26 | 92.49 | 89.2 | 84.61 |
| $p_9$ | 10.65 | 12.81 | 0.0 | 0.0 | 42.62 | 37.0 | 14.28 | 4.16 | 72.79 | 65.32 | 52.45 |
| $p_{10}$ | 21.14 | 22.71 | 0.0 | 0.0 | 34.08 | 30.33 | 0.0 | 0.0 | 93.33 | 92.32 | 90.58 |
| $p_{11}$ | 11.72 | 11.79 | 0.0 | 0.0 | 47.73 | 46.4 | 47.73 | 40.0 | 91.12 | 90.6 | 77.76 |
| $p_{12}$ | 13.1 | 6.58 | 0.0 | 0.0 | 53.1 | 53.74 | 57.69 | 62.5 | 85.91 | 83.73 | 67.38 |
| $p_{13}$ | 8.94 | 14.63 | 0.0 | 0.0 | 50.16 | 31.41 | 56.89 | 11.11 | 87.17 | 70.4 | 64.85 |
| $p_{14}$ | 9.02 | 9.72 | 0.0 | 0.0 | 61.96 | 46.65 | 66.66 | 36.84 | 84.66 | 72.43 | 49.89 |
| $p_{15}$ | 5.8 | 6.68 | 0.0 | 0.0 | 63.56 | 63.04 | 92.0 | 89.65 | 94.26 | 93.57 | 61.57 |
| $p_{16}$ | 6.46 | 6.24 | 0.0 | 0.0 | 61.33 | 57.44 | 89.44 | 89.44 | 86.6 | 83.0 | 60.69 |
| $p_{17}$ | 17.05 | 15.91 | 0.0 | 0.0 | 21.21 | 4.55 | 0.0 | 0.0 | 37.12 | 18.18 | 28.79 |
| $p_{18}$ | 4.83 | 4.83 | 0.0 | 0.0 | 50.66 | 45.17 | 60.0 | 52.27 | 92.56 | 90.47 | 78.96 |
| $p_{19}$ | 10.71 | 14.75 | 0.0 | 0.0 | 48.7 | 43.63 | 28.57 | 0.0 | 85.18 | 74.07 | 66.18 |
| $p_{20}$ | 5.93 | 6.81 | 0.0 | 0.0 | 43.32 | 42.19 | 30.76 | 11.11 | 73.43 | 71.38 | 55.03 |
| $p_{21}$ | 0.0 | 0.0 | 0.0 | 0.0 | 22.22 | 0.0 | 0.0 | 0.0 | 44.44 | 22.22 | 22.22 |

**A-** or **M-SV$_i$** is the average/median (per subject) safety violation of FLiRTS 2 with respect to tool **i**, i.e., Ekstazi and STARTS.
**A-** or **M-PV$_i$** is the average/median (per subject) precision violation of FLiRTS 2 with respect to tool **i**, i.e., Ekstazi and STARTS.
**A-R$_e$**, **A-R$_s$**, and **A-R$_f$** is the average reduction (per subject) in test suite size achieved by Ekstazi, STARTS, and FLiRTS 2, respectively.

FLiRTS 2 was 11.11% with respect to Ekstazi and 11.63% with respect to STARTS. This indicates that FLiRTS 2 missed on average about 12% of the test classes that were selected by the other tools. This is because FLiRTS 2 is designed to work with UML class diagrams, and this diagram type cannot provide certain types of information used by the other tools such as (1) dynamic dependencies of test classes with the classes under test, (2) Java reflection, (3) exceptions, (4) dependencies from test classes to third party libraries and input configuration files, and (5) the code contained inside the method body (e.g., local variables referencing classes and method invocations on classes). The lack of this information resulted in missing relevant test classes that must be classified as retestable.

### 6.2.3 RQ2: Precision Violation

Fig. 6.8 shows that the average precision violation of FLiRTS 2 with respect to Ekstazi was higher than 50% for 11 of the subjects; with respect to STARTS it was higher than 50% for 6 of the subjects. Among all the 8,060 revisions, the average precision violation of FLiRTS 2 was

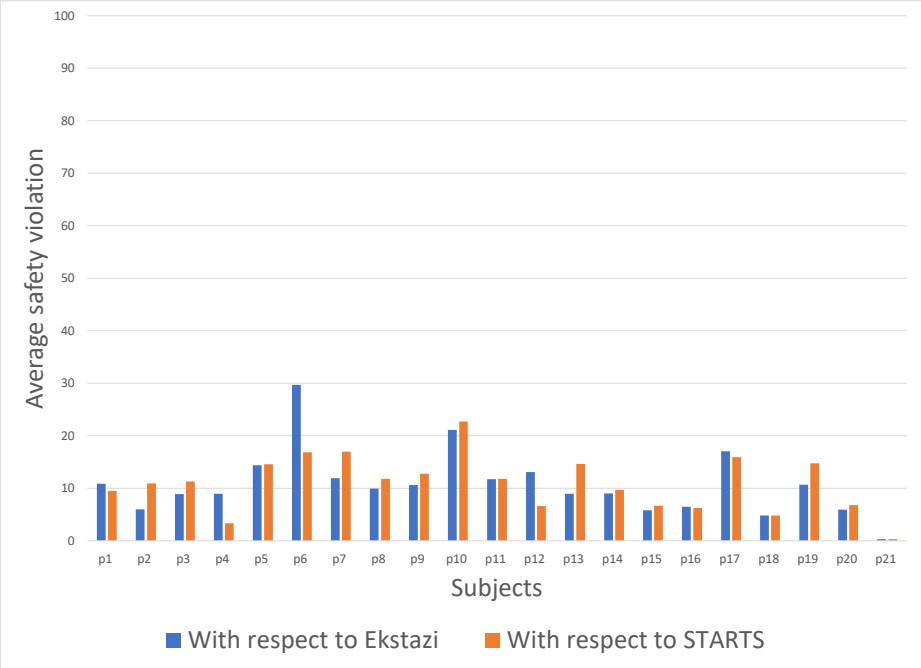



**Figure 6.7:** Averages of Safety Violation of FLiRTS 2

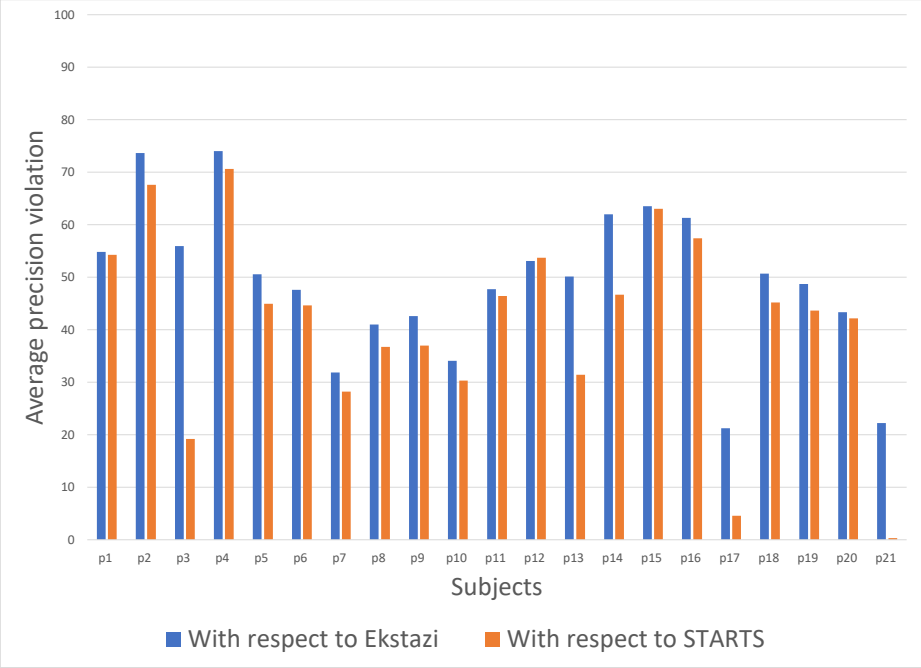



**Figure 6.8:** Averages of Precision Violation of FLiRTS 2

51.86% with respect to Ekstazi and 44.72% with respect to STARTS. This is because FLiRTS 2 is based on a probabilistic model, while Ekstazi is based on collecting the dynamic dependencies of

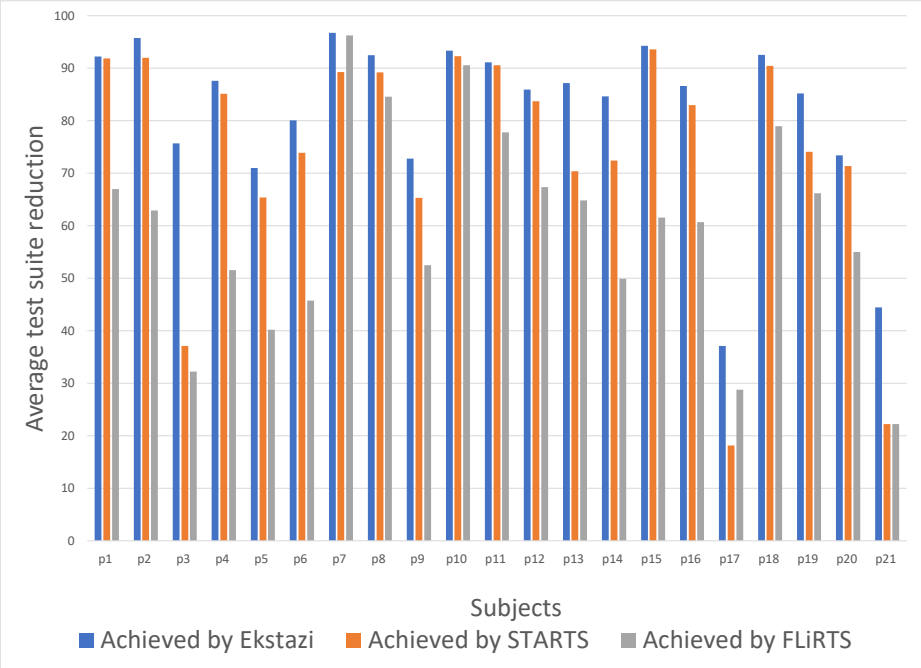



**Figure 6.9:** Test Suite Reduction of FLiRTS 2

the test classes, which can more precisely exclude test classes that do not traverse adapted classes. STARTS is a static approach and therefore less precise than Ekstazi. Thus, FLiRTS 2 achieved a lower average precision violation with respect to STARTS than with respect to Ekstazi.

### 6.2.4 RQ3: Reduction in Test Suite Size

Although FLiRTS 2 was less precise than Ekstazi and STARTS, it showed a reduction in the test suite size. Fig. 6.9 shows that FLiRTS 2 achieved a reduction for all the subjects, and that the average reduction was more than 50% for 15 out of 21 subjects. The average reduction achieved by FLiRTS 2 over all the subjects was 65.74%. STARTS achieved a higher average reduction than FLiRTS 2 for 18 subjects. Because Ekstazi is a dynamic approach, it achieved a higher average reduction than both FLiRTS 2 and STARTS for all the 21 subjects.

**Table 6.6:** Fault Detection Ability Results

| ID | Revs | A-MC$_{full}$ | A-MC$_{f}$ | A-MC$_{e}$ | A-MC$_{s}$ | e>f | s>f | f>e | f>s |
|---|---|---|---|---|---|---|---|---|---|
| p$_{2}$ | 173 | 50.90 | 49.59 | 26.64 | 26.64 | 7 | 7 | 72 | 71 |
| p$_{3}$ | 14 | 48.42 | 37.92 | 40.57 | 40.57 | 9 | 9 | 2 | 2 |
| p$_{4}$ | 462 | 14.27 | 14.04 | 13.02 | 13.03 | 7 | 7 | 28 | 28 |
| p$_{6}$ | 301 | 85.67 | 85.14 | 76.87 | 76.20 | 18 | 15 | 53 | 50 |
| p$_{7}$ | 283 | 83.13 | 81.77 | 48.73 | 49.31 | 13 | 8 | 141 | 127 |
| p$_{8}$ | 230 | 74.43 | 72.39 | 44.88 | 44.94 | 19 | 24 | 95 | 100 |
| p$_{9}$ | 85 | 85.12 | 80.69 | 43.96 | 45.61 | 6 | 6 | 45 | 42 |
| p$_{10}$ | 111 | 78.83 | 77.94 | 56.80 | 56.48 | 32 | 31 | 33 | 33 |
| p$_{11}$ | 240 | 79.54 | 79.48 | 47.62 | 47.64 | 7 | 8 | 110 | 108 |
| p$_{12}$ | 242 | 44.66 | 38.78 | 21.92 | 19.31 | 27 | 22 | 116 | 119 |
| p$_{14}$ | 38 | 63.28 | 50.50 | 30.68 | 30.76 | 2 | 2 | 14 | 13 |
| p$_{15}$ | 300 | 76.28 | 75.66 | 33.14 | 32.69 | 7 | 5 | 167 | 168 |
| p$_{16}$ | 102 | 56.35 | 55.60 | 24.73 | 24.80 | 4 | 4 | 61 | 54 |
| p$_{17}$ | 36 | 25.33 | 22.63 | 23.97 | 23.97 | 1 | 1 | 2 | 2 |
| p$_{18}$ | 19 | 65.42 | 65.26 | 30.42 | 33.52 | 1 | 1 | 9 | 9 |
| p$_{20}$ | 34 | 62.94 | 62.94 | 24.29 | 24.29 | 1 | 1 | 21 | 21 |
| p$_{21}$ | 3 | 57.33 | 57.33 | 57.33 | 57.33 | 0 | 0 | 0 | 0 |
| all | 2673 | 61.32 | 59.72 | 37.89 | 37.66 | 161 | 151 | 969 | 947 |

**Revs** is the number of revisions for which PIT worked.
**A-MC$_{full}$**, **A-MC$_{f}$**, **A-FD$_{e}$**, and **A-FD$_{s}$** are the average mutation scores achieved by the full test classes, and the test classes selected by FLiRTS, Ekstazi, and STARTS, respectively.
Columns **e>f** and **s>f** show the number of revisions in which the test classes selected by Ekstazi and STARTS, respectively, achieved higher mutation scores than the test classes selected by FLiRTS.
Columns **f>e** and **f>s** show the number of revisions in which the test classes selected by FLiRTS achieved higher mutation scores than the test classes selected by Ekstazi and STARTS, respectively.

### 6.2.5 RQ4: Fault Detection Ability

In the absence of real faults, mutation operators are often used to seed faults in testing experiments [114]. We use a mutation-based approach with the PIT [115] tool to apply first-order method-level mutation operators to the versions of the 21 subjects. We applied all the 13 PIT mutation operators [115], and configured PIT to mutate only the adapted classes of each version.

For each version, we ran PIT with the full test suites and those selected by FLiRTS 2, Ekstazi, and STARTS. Table 6.6 shows the mutation testing results. Due to JUnit version compatibility issues, PIT did not work for any of the revisions of p$_{1}$, p$_{5}$, p$_{13}$, and p$_{19}$. It also did not work for some of the revisions of the other subjects. The calculations used to create Table 6.6 did not include these revisions and subjects.

Table 6.6 shows that FLiRTS 2 achieved higher average mutation scores per subject than Ekstazi and STARTS for 15 out of 17 subjects. When identifying adapted classes, Ekstazi and STARTS ignore changes that are (1) undetectable by the smart checksums used to compare `.class` files, (2) related to compile time annotations and debug information, and (3) are present in the source code but not visible at the bytecode level. On the other hand, some of these changes were visible to the `diff` command used to generate the list of adapted classes provided as input to FLiRTS 2.

FLiRTS 2 showed lower mutation scores than Ekstazi and STARTS for 6% of the revisions used in the mutation testing experiment. The last column of Table 6.6 shows that FLiRTS 2 obtained lower mutation scores than Ekstazi and STARTS for 161 and 151 revisions, respectively. When only considering the 161 revisions, the average mutation score was 53.1% for FLiRTS 2 and 61% for Ekstazi. When only considering the 151 revisions, the average mutation score was 54.5% for FLiRTS 2 and 62.3% for STARTS. The reason for the loss in the fault detection ability was that FLiRTS 2 missed 12% of the test classes that were selected by the other tools, and this 12% included modification-traversing test classes that killed mutants in the adapted classes.

Considering all the revisions used in the mutation testing study, the average mutation scores achieved by the test classes selected by FLiRTS 2 and the full test suites were 59.72% and 61.32%, respectively. This indicates that even after reducing the test suites for all the subjects, the loss in FLiRTS 2's fault detection ability was low (1.6% on average). Note that this is not a result of higher precision violation of FLiRTS 2, which indicates that it selected more test cases than necessary. The extra test cases traverse unmodified classes, but mutation faults were seeded only in the modified classes. Only the modification-traversing test cases could have detected the faults, which shows that FLiRTS 2 did select the important test cases.

### 6.2.6 Threats to Validity

The external and internal threats to validity to the tuning process also apply here. We mitigated them in the same way as before. Moreover, we used 8,060 revisions from 21 open-source projects varying in size, application domain, and number of test classes.

**Construct validity.** We could have used other metrics (e.g., test coverage) to evaluate the effectiveness of FLiRTS 2. However, we used the most common metrics in the research literature: safety violation, precision violation, reduction in test suite size, and fault detection ability.

We did not perform a comparison of efficiency in terms of reducing regression testing time between FLiRTS 2 and the other RTS tools. Applying the end-to-end process with FLiRTS 2 would require models that were evolved and comparable. However, our reverse engineered models could not be compared using model comparison tools.

**Conclusion Validity.** We only used 21 subjects to evaluate FLiRTS 2. The use of additional subjects could affect the conclusions of the evaluation.

# Chapter 7

# Conclusions and Future Work

Regression testing is the process of running the existing test cases on a modified version of a system to ensure that the performed modifications do not break the existing functionality. Regression testing is an expensive process, and RTS approaches are used to improve regression testing efficiency by selecting a subset of test cases from an existing test suite to verify that the affected functionality of a program is still correct.

Existing model-based RTS approaches classify test cases by analyzing changes performed at the model level. These approaches suffer from the following limitations. First, they do not take into account the impact of inheritance hierarchy changes on the classification of test cases. Second, they require complete and detailed behavioral diagrams, but in practice, behavioral models tend to be incomplete and lack details needed to obtain the traceability links from test cases to model elements. Third, the applicability of model-based RTS in practice is limited because the behavioral diagrams are used less often than class diagrams.

In this dissertation, we proposed a static model-based RTS approach called FLiRTS 2 that addresses these three limitations. The development of FLiRTS 2 was driven by our experience accrued from two model-based RTS approaches called MaRTS and FLiRTS.

The key advantage of using FLiRTS 2 is that it relies only on class diagrams containing information that is commonly provided by software developers. It needs far less information than other model-based RTS approaches. No behavioral models, traceability links from the test cases to model elements, or coverage data are needed. In the absence of this information, FLiRTS 2 uses a probabilistic approach based on fuzzy logic to address the uncertainty in determining which classes and relationships in the class diagram are actually exercised by the test cases.

Using 21 subjects we compared the safety violation, precision violation, test suite reduction, and fault detection ability of FLiRTS 2 with those of two code-based RTS approaches, Ekstazi

and STARTS. The average safety violation of FLiRTS 2 was 11.11% with respect to Ekstazi and 11.63% with respect to STARTS. The average precision violation of FLiRTS 2 was 51.86% with respect to Ekstazi and 44.72% with respect to STARTS. The average test suite reduction using FLiRTS 2 was 65.74%. The average fault detection ability of the full test suites was 61.32%; for the reduced test suites it was 59.72% for FLiRTS 2, 37.89% for Ekstazi, and 37.66% for STARTS. In spite of the limited information available to FLiRTS 2, it was able to achieve test suite reduction with a small loss (i.e., less than 2% on average) of the fault detection effectiveness.

The subjects used in the evaluation of FLiRTS 2 are not representative of all domains. We plan to perform further empirical studies using additional subjects from different domains. We will investigate if using search-based software engineering techniques such as genetic algorithms [91] to tune the fuzzy sets and inference rules of the fuzzy logic system improves the safety and precision of FLiRTS 2. Moreover, we plan to investigate and improve the applicability of FLiRTS 2 in different domains. Some domains use class diagrams and customized UML profiles for the design and management of context-aware systems. Others use component-based models to support runtime adaptations. Thus, instead of the Class Relationships Graph, we will need to develop a new graph that represents the relationships between the components.

The future work directions identified below investigate the use of non-traditional RTS techniques that are based on fuzzy logic and machine learning to improve the applicability of model-based RTS in practice.

**Using Fuzzy Logic with Other Types of Models.** We plan to investigate the applicability of fuzzy logic to classify test cases in the approaches that use component-based models at runtime to support self-adaptation in autonomous systems [31–36]. These approaches use the Monitor-Analyze-Plan-Execute over shared Knowledge feedback control loop (MAPE-K) to plan and perform runtime adaptations. The monitoring phase extracts the information from the managed system and its environment. The analyzing phase analyzes system information and models the adaptation plan. The planning phase determines a set of actions to adapt the managed system, and the execute phase carries out the actions. Fuzzy logic can be used after the analysis phase, in which the adap-

tation plan is modeled but not yet executed. Possible inputs to the fuzzy logic system can include information extracted from the component-based models such as relationships between the components, the adaptation plan, and the monitored information regarding the context and environment of the managed system. We will investigate using fuzzy logic to classify test cases based on this information.

**Using fuzzy logic for test case prioritization.** Regression test prioritization is concerned with the identification of the ideal ordering of test cases that maximizes desirable properties, such as early fault detection [9]. We plan to investigate the use of fuzzy logic in model-based regression test prioritization based on information available in the class diagram. Such information includes diversity of test classes, which can be computed based on the dissimilarity between the relationships that start from test classes to classes under test.

**Machine learning techniques.** Supervised and unsupervised machine learning techniques are gaining traction in software engineering research. We will investigate the use of such techniques instead of fuzzy logic to support RTS based on information available in the UML class diagrams.

# Bibliography


[1] Georgia M. Kapitsaki and Iakovos S. Venieris. PCP: Privacy-Aware Context Profile Towards Context-Aware Application Development. In David Taniar and Eric Pardede, editors, *Proceedings of the 10th International Conference on Information Integration and Web-Based Applications & Services (iiWAS'08)*, pages 104–110, Linz, Austria, November 2008. ACM.

[2] Antonia Bertolino. Software Testing Research: Achievements, Challenges, Dreams. In *2007 Future of Software Engineering*, pages 85–103. IEEE Computer Society, 2007.

[3] Mary Jean Harrold. Testing Evolving Software. *Journal of Systems and Software*, 47(2-3):173–181, July 1999.

[4] Hareton K. N. Leung and Lee J. White. Insights into Regression Testing. In *Proceedings of Conference on Software Maintenance*, pages 60–69, Miami, FL, USA, October 1989. IEEE.

[5] Pavan Kumar Chittimalli and Mary Jean Harrold. Recomputing Coverage Information to Assist Regression Testing. *IEEE Transactions on Software Engineering*, 35(4):452–469, 2009.

[6] Emelie Engström and Per Runeson. A Qualitative Survey of Regression Testing Practices. In *International Conference on Product Focused Software Process Improvement*, pages 3–16. Springer, 2010.

[7] Lionel C. Briand, Yvan Labiche, and Siyaun He. Automating Regression Test Selection Based on UML Designs. *Journal on Information and Software Technology*, 51(1):16–30, January 2009.

[8] Emelie Engström, Per Runeson, and Mats Skoglund. A Systematic Review on Regression Test Selection Techniques. *Information and Software Technology*, 52(1):14–30, January 2010.

[9] Shin Yoo and Mark Harman. Regression Testing Minimization, Selection and Prioritization: A Survey. *Journal of Software Testing, Verification and Reliability*, 22(2):67–120, March 2012.

[10] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical Regression Test Selection with Dynamic File Dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA'15)*, pages 211–222, Baltimore, MD, USA, July 2015. ACM.

[11] Nan Ye, Xin Chen, Peng Jiang, Wenxu Ding, and Xuandong Li. Automatic Regression Test Selection Based on Activity Diagrams. In *Proceedings of the 5th International Conference on Secure Software Integration & Reliability Improvement Companion (SSIRI-C'11)*, pages 166–171, Jeju Island, South Korea, June 2011. IEEE.

[12] Yanping Chen, Robert L. Probert, and D. Paul Sims. Specification-Based Regression Test Selection with Risk Analysis. In Darlen A. Stewart and J. Howard Johnson, editors, *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'02)*, pages 1–14. IBM Press, September 2002.

[13] Bogdan Korel, Luay Ho Tahat, and Boris Vaysburg. Model Based Regression Test Reduction Using Dependence Analysis. In Giuliano Antoniol and Ira D. Baxter, editors, *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, pages 214–223, Montréal, Quebec, Canada, October 2002. IEEE.

[14] Qurat-ul-ann Farooq, Muhammad Zohaib Z. Iqbal, Zafar I. Malik, and Matthias 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)*, pages 41–49, Oxford, UK, March 2010. IEEE.

[15] Philipp Zech, Michael Felderer, Philipp Kalb, and Ruth Breu. A Generic Platform for Model-Based Regression Testing. In Tiziana Margaria and Bernhard Steffen, editors, *Proceedings of the 5th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'12)*, Lecture Notes in Computer Science 7609, pages 112–126, Heraclion, Crete, October 2012. Springer.

[16] Philipp Zech, Philipp Kalb, Michael Felderer, Colin Atkinson, and Ruth Breu. Model-Based Regression Testing by OCL. *International Journal on Software Tools for Technology Transfer*, 19(1):115–131, February 2017.

[17] Robert B. France and Bernhard Rumpe. Model-Driven Development of Complex Software: A Research Roadmap. In Lionel C. Briand and Alexander L. Wolf, editors, *Proceedings of Future of Software Engineering (FoSE'07)*, pages 37–54, Minneapolis, MN, USA, May 2007. IEEE Computer Society.

[18] Colin Atkinson and Thomas Kuhne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, 2003.

[19] Wojciech James Dzidek, Erik Arisholm, and Lionel C. Briand. A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance. *IEEE Transactions on Software Engineering*, 34(3):407–432, May/June 2008.

[20] Adam C. Jensen and Betty H. C. Cheng. On the Use of Genetic Programming for Automated Refactoring and the Introduction of Design Patterns. In Jürgen Branke, editor, *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation (GECCO'10)*, pages 1341–1348, Portland, OR, USA, July 2010. ACM.

[21] Iman Hemati Moghadam and Mel Ó. Cinneéide. Automated Refactoring Using Design Differencing. In Tom Mens and Anthony Cleve, editors, *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR'12)*, pages 43–52, Szeged, Hungary, March 2012. IEEE.

[22] João Pablo S. da Silva, Miguel Ecar, Marcelo S. Pimenta, Gilleanes T. A. Guedes, Luiz Paulo Franz, and Luciano Marchezan. A Systematic Literature Review of UML-Based Domain-Specific Modeling Languages for Self-Adaptive Systems. In Danny Weyns, editor, *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'18)*, pages 87–93, Gothenburg, Sweden, May 2018. ACM.

[23] Quan Z. Sheng and Boualem Benatallah. ContextUML: A UML-Based Modeling Language for Model-Driven Development of Context-Aware Web Services. In Wayne Brookes, Elaine Lawrence, Robert Steele, and Elizabeth Chang, editors, *Proceedings of the International Conference on Mobile Business (ICMB'05)*, pages 206–212, Sydney, Australia, July 2005. IEEE.

[24] Christof Simons and Guido Wirtz. Modeling Context in Mobile Distributed Systems with the UML. *Journal of Visual Languages and Computing*, 18:420–439, 2007.

[25] I-Ching Hsu. Extending UML to Model Web 2.0-Based Context-Aware Applications. *Software—Practice and Experience*, 42(10):1211–1227, October 2012.

[26] Mohamed Salah Benselim and Hassina Seridi-Bouchelaghem. Extended UML for the Development of Context-Aware Applications. In Rachid Benlamri, editor, *Proceedings of the 4th International Conference on Networked Digital Technologies (NDT'12)*, Communications in Computer and Information Science 293, pages 33–43, Dubai, United Arab Emirates, April 2012. Springer.

[27] Rossano P. Pinto, Eleri Cardozo, Paulo R. S. L. Coelho, and Eliane G. Guimarães. A Domain-Independent Middleware Framework for Context-Aware Applications. In *Proceedings of the 6th International Workshop on Adaptive and Reflective Middleware (ARM'07)*, pages 5:1–5:6, Newport Beach, CA, USA, November 2007. ACM.

[28] Ahmed Al-Alshuhai and François Siewe. An Extension of Class Diagram to Model the Structure of Context-Aware Systems. In *Proceedings of the 6th International Joint Con-*

*ference on Advances in Engineering and Technology (AET 2015)*, Cochin, India, December 2015.

[29] Haider Boudjemline, Mohamed Touahria, Abdelhak Boubetra, and Hamza Kaabeche. Heavyweight Extension to the UML Class Diagram Metamodel for Modeling Context Aware Systems in Ubiquitous Computing. *Journal of Pervasive Computing and Communications*, 13(3):238–251, June 2017.

[30] João Pablo S. da Silva, Miguel Ecar, Marcelo S. Pimenta, Gilleanes T. A. Guedes, and Elder M. Rodrigues. Towards a Domain-Specific Modeling Language for Self-Adaptive Systems Conceptual Modeling. In Uirá Kulesza, editor, *Proceedings of the XXXII Brazilian Symposium on Software Engineering (SBES'18)*, pages 208–213, Sao Carlos, Brazil, September 2018. ACM.

[31] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjørven. Beyond Design Time: Using Architecture Models for Runtime Adaptability. *IEEE Software*, 23(2):62–70, March 2006.

[32] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure. *IEEE Computer*, 37(10):46–54, October 2004.

[33] John C. Georgas, André van der Hoek, and Richard N. Taylor. Using Architectural Models to Manage and Visualize Runtime Adaptation. *IEEE Computer*, 42(10):52–60, October 2009.

[34] Brice Morin, Olivier Barais, Jean-Marc Jézéquel, Franck Fleurey, and Arnor Solberg. Models@Run.time to Support Dynamic Adaptation. *IEEE Computer*, 42(10):44–51, October 2009.

[35] Thomas Vogel and Holger Giese. Adaptation and Abstract Runtime Models. In *Proceedings of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'10)*, pages 39–48, Cape Town, South Africa, May 2010. ACM.

[36] Thomas Cottenier, Aswin van den Berg, and Tzilla Elrad. Motorola WEAVR: Aspect Orientation and Model-Driven Engineering. *Journal of Object Technology*, 6(7):51–88, August 2007.

[37] Walter Cazzola, Nicole Alicia Rossini, Phillipa Bennett, Sai Pradeep Mandalaparty, and Robert B. France. Fine-Grained Semi-Automated Runtime Evolution. In Nelly Bencomo, Betty H. C. Cheng, Robert B. France, and Uwe Aßmann, editors, *MoDELS@Run-Time*, Lecture Notes in Computer Science 8378, pages 237–258. Springer, August 2014.

[38] Walter Cazzola, Nicole Alicia Rossini, Mohammed Al-Refai, and Robert B. France. Fine-Grained Software Evolution using UML Activity and Class Models. In Ana Moreira and Bernhard Schätz, editors, *Proceedings of the 16$^{th}$ International Conference on Model Driven Engineering Languages and Systems (MoDELS'13)*, Lecture Notes in Computer Science 8107, pages 271–286, Miami, FL, USA, 29th of September-4th of October 2013. Springer.

[39] Hdaytullah, Sriharsha Vathsavayi, Outi Räihä, Kai Koskimies, and Allan Raundahl Gregersen. Applying Genetic Self-Architecting for Distributed Systems. In *Proceedings of the 4th World Congress on Nature and Biologically Inspired Computing (NaBIC'12)*, pages 44–52, Mexico City, Mexico, November 2012. IEEE.

[40] Allan Raundahl Gregersen and Bo Nørregaard Jørgensen. Dynamic Update of Java Applications—Balancing Change Flexibility vs Programming Transparency. *Journal of Software Maintenance and Evolution: Research and Practice*, 21(2):81–112, March/April 2009.

[41] Christian F. J. Lange, Michel R. V. Chaudron, and Johan Muskens. In Practice: UML Software Architecture and Design Description. *IEEE Software*, 23(2):40–46, March/April 2006.

[42] Christian F. J. Lange and Michel R. V. Chaudron. An Empirical Assessment of Completeness in UML Designs. In *Proceedings of the 8th International Conference on Empirical Assessment in Software Engineering (EASE '04)*, pages 111–121. IET, 2004.

[43] Brian Dobing and Jeffrey Parsons. How UML is Used. *Communications of the ACM*, 49(5):109–113, May 2006.

[44] Brian Dobing and Jeffrey Parsons. Current Practices in the Use of UML. In Jacky Akoka, Stephen W. Liddle, Il-Yeol Song, Michela Bertolotto, and Isabelle Comyn-Wattiau, editors, *Proceedings of the 24th International Conference on Perspectives in Conceptual Modeling (ER'05)*, pages 2–11, Klagenfurt, Austria, October 2005.

[45] Martin Grossman, Jay E. Aronson, and Richard V. McCarthy. Does UML Make the Grade? Insights from the Software Development Community. *Information and Software Technology*, 47(6):383–397, April 2005.

[46] Luciane Telinski Wiedermann Agner, Inali Wisniewski Soares, Paulo César Stadzisz, and Jean Marcelo Simão. A Brazilian Survey on UML and Model-Driven Practices for Embedded Software Development. *Journal of Systems and Software*, 86(4):997–1005, April 2013.

[47] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical Assessment of MDE in Industry. In Harald Gall and Nenad Medvidović, editors, *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 471–480, Honolulu, HI, USA, May 2011. IEEE.

[48] Ana M. Fernández-Sáez, Danilo Caivano, Marcela Genero, and Michel R. V. Chaudron. On the Use of UML Documentation in Software Maintenance: Results from a Survey in

Industry. In Jordi Cabot, editor, *Proceedings of the 18th International Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 292–301, Ottawa, ON, Canada, September 2015. IEEE.

[49] Marian Petre. UML in Practice. In Betty H. C. Cheng and Klaus Pohl, editors, *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*, pages 722–731, San Francisco, CA, USA, May 2013. IEEE.

[50] Mohd Hafeez Osman and Michel R. V. Chaudron. UML Usage in Open Source Software Development: A Field Study. In *Proceedings of 3rd International Workshop on Experiences and Empirical Studies in Software Modeling (EESSMod'13)*, pages 23–32, Miami, FL, USA, October 2013.

[51] Philip Langer, Tanja Mayerhofer, Manuel Wimmer, and Gerti Kappel. On the Usage of UML: Initial Results of Analyzing Open UML Models. In Hans-Georf Fill, Dimitris Karagiannis, and Ulrich Reimer, editors, *Proceedings of Modellierung (Modellierung'14)*, Lecture Notes in Informatics 225, pages 289–304, Wien, Austria, March 2014. Springer.

[52] Daniel Leroux, Martin Nally, and Kenneth Hussey. Rational Software Architect: A Tool for Domain-Specific Modeling. *IBM Systems Journal*, 45(3):555–568, 2006.

[53] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[54] Owolabi Legunsen, August Shi, and Darko Marinov. STARTS: STAtic Regression Test Selection. In Massimiliano Di Penta and Tien N. Nguyen, editors, *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*, pages 949–954, Urbana-Champaign, IL, USA, October 2017. IEEE Press.

[55] Gregg Rothermel and Mary Jean Harrold. Analyzing Regression Test Selection Techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, August 1996.

[56] Henry Muccini, Marcio Dias, and Debra J. Richardson. Reasoning about Software Architecture-Based Regression Testing Through a Case Study. In *29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, volume 2, pages 189–195. IEEE, 2005.

[57] Henry Muccini, Marcio S. Dias, and Debra J. Richardson. Towards Software Architecture-Based Regression Testing. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–7. ACM, 2005.

[58] Hasan Ural and Hüsnü Yenigün. Regression Test Suite Selection Using Dependence Analysis. *Journal of Software: Evolution and Process*, 25:681–709, 2013.

[59] Sascha Lity, Thomas Morbach, Thomas Thüm, and Ina Schaefer. Applying Incremental Model Slicing to Product-Line Regression Testing. In Georgia M. Kapitsaki and Eduardo Santana de Almeida, editors, *Proceedings of the 15th International Conference on Software Reuse (ICSR'16)*, Lecture Notes in Computer Science 9679, pages 3–19, Limassol, Cyprus, June 2016. Springer.

[60] David C. Kung, Jerry Gao, Pei Hsia, Yasufumi Toyoshima, and Cris Chen. On Regression Testing of Object-Oriented Programs. *Journal of Systems and Software*, 32(1):21–40, January 1996.

[61] Yoon Kyu Jang, Malcolm Munro, and Yong Rae Kwon. An Improved Method of Selecting Regression Tests for C++ Programs. *Journal of Software Maintenance and Evolution*, 13(5):331–350, September/October 2011.

[62] Pei Hsia, Xiaolin Li, David Chen-Ho Kung, Chih-Tung Hsu, Liang Li, Yasufumi Toyoshima, and Criss Chen. A Technique for the Selective Revalidation of OO Software. *Journal of Software: Evolution and Process*, 9(4):217–233, July 1997.

[63] Lee J. White and Khalil Abdullah. A Firewall Approach for Regression Testing of Object-Oriented Software. In *Proceedings of the 10th International Software Quality Week (QW'97)*, San Francisco, CA, USA, May 1997.

[64] Mats Skoglund and Per Runeson. Improving Class Firewall Regression Test Selection by Removing the Class Firewall. *International Journal of Software Engineering and Knowledge Engineering*, 17(3):359–378, June 2007.

[65] Quinten David Soetens, Serge Demeyer, and Andy Zaidman. Change-Based Test Selection in the Presence of Developer Tests. In Anthony Cleve and Filippo Ricca, editors, *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR'13)*, pages 101–110, Genoa, Italy, March 2013. IEEE.

[66] Quinten David Soetens, Serge Demeyer, Andy Zaidman, and Javier Pérez. Change-Based Test Selection: An Empirical Evaluation. *Empirical Software Engineering*, pages 1–43, November 2015.

[67] Gregg Rothermel and Mary Jean Harrold. A Safe, Efficient Regression Test Selection Technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.

[68] Gregg Rothermel, Mary Jean Harrold, and Jeinay Dedhia. Regression Test Selection for C++ Software. *Software Testing, Verification and Reliability*, 10(2):77–109, June 2000.

[69] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. Regression Test Selection for Java Software. In John Vlissides, editor, *Proceedings of the 16th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, pages 312–326, Tampa, FL, USA, October 2001. ACM.

[70] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In Walter G. Olthoff, editor, *Proceedings of*

*the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, LNCS 952, pages 77–101, Åarhus, Denmark, August 1995. Springer.

[71] Filippos Vokolos and Phyllis G. Frankl. Empirical Evaluation of the Textual Differencing Regression Testing Technique. In *Proceedings of the International Conference on Software Maintenance (SM'98)*, pages 44–53, Bethesda, MD, USA, November 1998.

[72] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. An Extensive Study of Static Regression Test Selection in Modern Software Evolution. In Jane Cleland-Huang and Zhendong Su, editors, *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*, pages 583–594, Seattle, WA, USA, November 2016. ACM.

[73] Lingming Zhang. Hybrid Regression Test Selection. In Marsha Chechik and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*, pages 199–209, Gotheburg, Sweden, May/June 2018. IEEE.

[74] Tingting Yu and Ting Wang. A Study of Regression Test Selection in Continuous Integration Environments. In Sudipto Ghosh and Roberto Natella, editors, *Proceedings of the 29th International Symposium on Software Reliability Engineering (ISSRE'18)*, pages 135–143, Memphis, TN, USA, October 2018. IEEE.

[75] Alex Gyori, Owolabi Legunsen, Farah Hariri, and Darko Marinov. Evaluating Regression Test Selection Opportunities in a Very Large Open-Source Ecosystem. In Sudipto Ghosh and Roberto Natella, editors, *Proceedings of the 29th International Symposium on Software Reliability Engineering (ISSRE'18)*, pages 112–122, Memphis, TN, USA, October 2018. IEEE.

[76] Simone Romano, Giuseppe Scanniello, Giuliano Antoniol, and Alessandro Marchetto. SPIRITuS: a SimPle Information Retrieval RegressIon Test Selection Approach. *Information and Software Technology*, 99:62–80, July 2018.

[77] Maral Azizi and Hyunsook Do. ReTEST: A Cost Effective Test Case Selection Technique for Modern Software Development. In Sudipto Ghosh and Roberto Natella, editors, *Proceedings of the 29th International Symposium on Software Reliability Engineering (ISSRE'18)*, pages 144–154, Memphis, TN, USA, October 2018. IEEE.

[78] Zhiwei Xu, Kehan Gao, Taghi M. Khoshgoftaar, and Naeem Seliya. System Regression Test Planning with a Fuzzy Expert System. *Information Sciences*, 259:532–543, 2014.

[79] Christoph Malz, Nasser Jazdi, and Peter Göhner. Prioritization of Test Cases Using Software Agents and Fuzzy Logic. In Antonia Bertolino and Yvan Labiche, editors, *Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST'12)*, pages 483–486, Montréal, BC, Canada, April 2012. IEEE.

[80] Eric J. Rapos and Jürgen Dingel. Using Fuzzy Logic and Symbolic Execution to Prioritize UML-RT Test Cases. In Gordon Fraser and Darko Marinov, editors, *Proceedings of the 8th International Conference on Software Testing, Verification and Validation (ICST'15)*, Graz, Austria, April 2015. IEEE.

[81] Wasiur Rhmann and Vipin Saxena. Fuzzy Expert System Based Test Cases Prioritization from UML State Machine Diagram using Risk Information. *International Journal on Mathematical Sciences and Computing*, 3(1):17–27, January 2017.

[82] Walter Cazzola, Sonia Pini, Ahmed Ghoneim, and Gunter Saake. Co-Evolving Application Code and Design Models by Exploiting Meta-Data. In *Proceedings of the 22$^{nd}$ Annual ACM Symposium on Applied Computing (SAC'07)*, pages 1275–1279, Seoul, South Korea, on 11th-15th of March 2007. ACM Press.

[83] Mario Pukall, Christian Kästner, Walter Cazzola, Sebastian Götz, Alexander Grebhahn, Reimar Schöter, and Gunter Saake. JavAdaptor — Flexible Runtime Updates of Java Applications. *Software—Practice and Experience*, 43(2):153–185, February 2013.

[84] Walter Cazzola and Edoardo Vacchi. @Java: Bringing a Richer Annotation Model to Java. *Computer Languages, Systems & Structures*, 40(1):2–18, April 2014.

[85] Mohammed Al-Refai, Walter Cazzola, Sudipto Ghosh, and Robert France. Using Models to Validate Unanticipated, Fine-Grained Adaptations at Runtime. In Helene Waeselynck and Radu Babiceanu, editors, *Proceedings of the 17th IEEE International Symposium on High Assurance Systems Engineering (HASE'16)*, pages 23–30, Orlando, FL, USA, 7th-9th of January 2016. IEEE.

[86] Lotfi A. Zadeh. Fuzzy Sets. *Information and Control*, 8(3):338–353, June 1965.

[87] Pablo Cingolani and Jesús Alcalaá-Fdez. jFuzzyLogic: A Java Library to Design Fuzzy Logic Controllers According to the Standard for Fuzzy Control Programming. *International Journal of Computational Intelligence Systems*, 6(1):61–75, 2013.

[88] Chuen-Chien Lee. Fuzzy Logic in Control Systems: Fuzzy Logic Controller. *IEEE Transactions on systems, man, and cybernetics*, 20(2):419–435, 1990.

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

[90] Timothy J. Ross. *Fuzzy Logic with Engineering Applications*. John Wiley & Sons, Ltd, third edition, December 2011.

[91] Oscar Cordón, Fernando Gomide, Francisco Herrera, Frank Hoffmann, and Luis Magdalena. Ten Years of Genetic Fuzzy Systems: Current Framework and New Trends. *Fuzzy Sets and Systems*, 141(1):5–31, January 2004.

[92] Marcos Evandro Cintra, Heloisa A. Camargo, and Maria Carolina Monard. A Study on Techniques for the Automatic Generation of Membership Functions for Pattern Recognition. In *Proceedings of the 3rd Conference of Trinational Academy of Sciences*, pages 1–10, Foz de Iguaçu, Brazil, 2008.

[93] Ebrahim H. Mamdani and Sedrak Assilian. An Experiment in Linguistic Synthesis with a Fuzzy Logic Controller. *International Journal of Man-Machine Studies*, 7(1):1–13, January 1975.

[94] Conrad Power, Alvin Simms, and Roger White. Hierarchical Fuzzy Pattern Matching for the Regional Comparison of Land Use Maps. *International Journal of Geographical Information Science*, 15(1):77–100, 2001.

[95] Abdelwahab Hamam and Nicolas D. Georganas. A Comparison of Mamdani and Sugeno Fuzzy Inference Systems for Evaluating the Quality of Experience of Hapto-Audio-Visual Applications. In *2008 IEEE International Workshop on Haptic Audio visual Environments and Games*, pages 87–92. IEEE, 2008.

[96] Mohammed Al-Refai, Sudipto Ghosh, and Walter Cazzola. Supporting Inheritance Hierarchy Changes in Model-Based Regression Test Selection. *Software and Systems Modeling*, 16(62):1–22, December 2017. Special Issue on Model-Based Testing.

[97] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading, Massachusetts, June 1999.

[98] Andrea Arcuri, José Campos, and Gordon Fraser. Unit Test Generation During Software Development: EvoSuite Plugins for Maven, IntelliJ and Jenkins. In Lionel Briand and Sarfraz Khurshid, editors, *Proceedings of the 9th IEEE International Conference on Software Testing, Verification and Validation (ICST'16)*, pages 401–408, Chicago, IL, USA, April 2016. IEEE.

[99] Gerardo Canfora, Massimiliano Di Penta, and Luigi Cerulo. Achievements and Challenges in Software Reverse Engineering. *Commun. ACM*, 54(4):142–151, 2011.

[100] Hugo Bruneliere, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. Modisco: A Model Driven Reverse Engineering Framework. *Information and Software Technology*, 56(8):1012–1032, 2014.

[101] Alexander Bergmayr, Hugo Bruneliere, Jordi Cabot, Jokin Garcia, Tanja Mayerhofer, and Manuel Wimmer. fREX: fUML-Based Reverse Engineering of Executable Behavior for Software Dynamic Analysis. In *Proceedings of the 8th International Workshop on Modeling in Software Engineering*, pages 20–26. ACM, 2016.

[102] Claudia Raibulet, Francesca Arcelli Fontana, and Marco Zanoni. Model-Driven Reverse Engineering Approaches: A Systematic Literature Review. *IEEE Access*, 5:14516–14542, 2017.

[103] Mohammed Al-Refai, Walter Cazzola, and Sudipto Ghosh. A Fuzzy Logic Based Approach for Model-Based Regression Test Selection. In Jeff Gray and Vinay Kulkarni, editors, *Proceedings of the 20$^{th}$ International Conference on Model Driven Engineering Languages and Systems (MoDELS'17)*, pages 55–62, Austin, TX, USA, 17th-22nd of September 2017. IEEE.

[104] Lotfi A. Zadeh. Is There a Need for Fuzzy Logic? *Information sciences*, 178(13):2751–2779, 2008.

[105] Mohammed Al-Refai, Sudipto Ghosh, and Walter Cazzola. Model-based Regression Test Selection for Validating Runtime Adaptation of Software Systems. In Lionel Briand and Sarfraz Khurshid, editors, *Proceedings of the 9th IEEE International Conference on Software Testing, Verification and Validation (ICST'16)*, pages 288–298, Chicago, IL, USA, 10th-15th of April 2016. IEEE.

[106] Conrad Bock, Steve Cook, Pete Rivett, Tom Rutt, Ed Seidewitz, Bran Selic, and Doug Tolbert. About the Unified Modeling Language Specification Version 2.5.1. Available at https://www.omg.org/spec/UML/2.5.1/, December 2017.

[107] Zhenchang Xing and Eleni Stroulia. Differencing Logical UML Models. *Automated Software Engineering*, 14(2):215–259, June 2007.

[108] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In Andreas Zeller, editor, *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE'11)*, pages 179–189, Szeged, Hungary, September 2011. ACM.

[109] Cédric Brun and Alfonso Pierantonio. Model Differences in the Eclipse Modeling Framework. *UPGRADE, The European Journal for the Informatics Professional*, 9(2):29–34, April 2008.

[110] August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. Balancing Trade-offs in Test-suite Reduction. In Alessandro Orso and Margaret-Anne Storey, editors, *Proceedings of the 22nd International Symposium on Foundations of Software Engineering (FSE'14)*, pages 246–256, Hong Kong, China, November 2014. ACM.

[111] Michel Marie Deza and Elena Deza. *Encyclopedia of Distances*. Springer, 2009.

[112] Vikram Goyal. Analyze Your Classes. *O'Reilly*, October 2003. Available at http://www.onjava.com/pub/a/onjava/2003/10/22/bcel.html?page=1.

[113] Barak Naveh and J. V. Sichi. JGraphT a Free Java Graph Library, 2011.

[114] James H. Andrews, Lionel C. Briand, and Yvan Labiche. Is Mutation an Appropriate Tool for Testing Experiments? In William Griswold and Bashar Nuseibeh, editors, *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pages 402–411, Saint Louis, MO, USA, May 2005. IEEE.

[115] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. PIT: A Practical Mutation Testing Tool for Java. In Abhik Roychoudhury, editor, *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)*, pages 449–452, Saarbrücken, Germany, July 2016. ACM.