International Journal on Software and Systems Modeling manuscript No.
(will be inserted by the editor)

Supporting Inheritance Hierarchy Changes in Model-based Regression Test Selection

Mohammed Al-Refai · Sudipto Ghosh · Walter Cazzola

Received: date / Accepted: date

Abstract Models can be used to ease and manage the development, evolution, and runtime adaptation of a software system. When models are adapted, the resulting models must be rigorously tested. Apart from adding new test cases, it is also important to perform regression testing to ensure that the evolution or adaptation did not break existing functionality. Since regression testing is performed with limited resources and under time constraints, regression test selection (RTS) techniques are needed to reduce the cost of regression testing. Applying model-level RTS for model-based evolution and adaptation is more convenient than using code-level RTS because the test selection process happens at the same level of abstraction as that of evolution and adaptation.

In earlier work, we proposed a model-based RTS approach called MaRTS to be used with a fine-grained model-based adaptation framework that targets applications implemented in Java. MaRTS uses UML models consisting of class and activity diagrams. It classifies test cases as obsolete, reusable, or retestable based on changes made to UML class and activity diagrams of the system being adapted. However, MaRTS did not take into account the changes made to the inheritance hierarchy in the class diagram and the impact of these changes on the selection of test cases. This paper extends MaRTS to support such changes, and demonstrates that the extended approach performs as well as or better than code-based RTS approaches in safely selecting regression test cases. While MaRTS can generally be used during any model-driven development or model-based evolution activity, we have developed it in the context of runtime adaptation. We evaluated the extended MaRTS on a set of applications, and compared the results with code-based RTS

M. Al-Refai
Computer Science Department, Colorado State University, USA
E-mail: al-refai@cs.colostate.edu

S. Ghosh
Computer Science Department, Colorado State University, USA
E-mail: ghosh@cs.colostate.edu

W. Cazzola
Department of Computer Science, Università degli Studi di Milano, Milan, Italy
E-mail: cazzola@di.unimi.it

approaches that also support changes to the inheritance hierarchy. The results showed that the extended MaRTS selected all the test cases relevant to the inheritance hierarchy changes, and that the fault detection ability of the selected test cases was never lower than that of the baseline test cases. The extended MaRTS achieved comparable results to a graph-walk code-based RTS approach (DejaVu), and showed a higher reduction in the number of selected test cases when compared with a static analysis code-based RTS approach (ChEOPSJ).

Keywords executable UML models · inheritance hierarchy · model-based adaptation · model-based regression test selection · model validation · runtime adaptation · UML activity diagram · UML class diagram

1 Introduction

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 requirement, design, and implementation models [17]. Model-Driven Development 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. Empirical studies showed 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 [12].

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]. Examples of software systems that require runtime adaptation are transportation management systems and online gaming systems, which need to be adapted at runtime without stopping their execution for safety and business reasons.

Since changes made to a system for evolution or runtime adaptation purposes can also break existing functionality, regression testing must be performed to ensure that the relevant existing test cases still pass [20]. Regression testing is one of the most expensive activities performed during the lifecycle of a software system [21, 30]. During both evolution and runtime adaptation, there may be severe restrictions on the time and resources available to perform regression testing. Regression test selection (RTS) [20] is an approach to improve regression testing efficiency by selecting a subset of the original test set to verify that the affected functionality of a system is still correct [4, 20]. RTS classifies existing test cases as obsolete, retestable, and reusable. Obsolete test cases are invalid because they cannot be executed on the modified version of the system, or the test inputs do not conform to the modified functionality of the system. Obsolete test cases need to be modified or deleted. 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 they do not need to be re-executed [25].

We chose to work in the area of model-based RTS because it has several advantages over code-based RTS. First, researchers have reported that the use of model-based RTS approaches will have crucial importance in the future because

for large scale systems, model-based RTS can scale up better than code-based RTS [37]. Second, the effort required for regression testing can be estimated at an early phase before propagating the changes to the software system [4, 38]. Third, managing traceability at the design level can be more practical than doing it at the code level because it enables the specification of dependencies at a higher level of abstraction [4,38]. Fourth, Harrold [20] suggests that working closer to the architectural level may be more efficient than at the source code level.

Model-based RTS approaches can be more convenient for approaches that already apply model-driven development, model-based evolution, and model-based runtime adaptations of software systems. The reason is that both the evolution/adaptation and test selection processes can be performed at the same level of abstraction, i.e., based on model-level changes. However, existing model-based RTS approaches suffer from the following drawbacks. First, they cannot detect all types of changes from UML class, sequence, and statechart diagrams used in these approaches [4, 14, 38]. Briand et al. [4] provided an example for such unsupported changes, which is a modification to an operation implementation that does not affect the operation's signature and contract. Second, they do not support the identification of changes to inherited and overridden operations along the inheritance hierarchy [4, 14, 38], which can affect test cases.

As a consequence of these limitations, existing model-based RTS approaches are unsafe and less precise compared to code-based RTS approaches that support fine-grained changes and consider changes that impact the inheritance hierarchy. Fine-grained changes are those that can be made at a low level of abstraction, such as changes at the statement level inside a method implementation. A safe RTS approach must select all the test cases that exercise added, deleted, and modified code. A precise RTS approach only selects those test cases that are relevant to the code modifications.

In prior work [2], we proposed a model-based RTS approach called MaRTS to be used at runtime for regression testing of unanticipated fine-grained adaptations performed at the model level. MaRTS uses UML design class and activity diagrams to represent fine-grained behaviors of a software system and its test cases. MaRTS classifies the test cases as obsolete, reusable, or retestable based on fine-grained changes made to the UML class and activity diagrams while adapting the system. Unlike other existing model-based RTS approaches [4,9,14,38], MaRTS can identify fine-grained changes to an operation implementation at the block and statement levels.

However, MaRTS [2] does not take into account the changes made to the inheritance hierarchy in the class diagram. Even a simple change, such as deleting a generalization relation between two classes, or adding or removing an overriding method, can impact many classes along the inheritance hierarchy by affecting the inherited and overridden methods. Thus, test cases that traverse the affected classes and methods must be selected for safe regression testing. Due to this limitation, MaRTS incorrectly classified 120 test cases for a chess program that we used in a case study as obsolete, while these test cases are still retestable and should have been selected for regression testing, as explained later in the paper. Such a limitation is common to all the model-based approaches, such as [4, 9, 14, 38].

The goal of this paper is to extend MaRTS to support the impact of changes made to the inheritance hierarchy. This will enable MaRTS to get results comparable to those obtained by using code-based RTS approaches that are fine-grained

and support these types of changes (e.g., Harrold et al. [21]). In order to achieve this goal, MaRTS must be extended to identify (1) changes performed to UML classes and their relations, (2) fine-grained changes performed to method implementations represented as UML activity diagrams, and (3) the impact of both types of changes on the inheritance hierarchy, such as identifying which operations are inherited and overridden in each class.

The extended MaRTS is based on (1) static analysis of the UML class diagram to identify the changes in the inheritance hierarchy, (2) dynamic analysis of the test case execution at the model level to determine the coverage information for each test case, and (3) fine-grained model comparison to identify changes performed to UML class and activity diagrams during the adaptation process. We exploit the *Rational Software Architect* (RSA) simulation toolkit 9.0[1] to execute test cases at the model level.

MaRTS can be used within the contexts of model-based evolution and runtime adaptation even though it was originally developed in the context of runtime adaptation. Most existing model-based adaptation approaches are coarse-grained and focus on using models at runtime to support self-adaptation in autonomous systems [10, 15, 18, 19, 26, 34]. Adaptations in these approaches are performed at the component level, and are limited to adding/removing/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 to support unanticipated adaptations. For example, the *Fine Grained Adaptation* (FiGA) framework [6,7] uses diagrams to support the planning of unanticipated and fine-grained adaptations on running Java software systems. These adaptations involve humans making model changes that can be fine-grained, such as those at the level of classes and methods. We developed MaRTS within the context of FiGA.

The extended MaRTS was evaluated on four applications, and compared with two code-based RTS approaches that support changes to the inheritance hierarchy. MaRTS achieved results comparable to the results achieved by DejaVu, and outperformed ChEOPSJ.

2 Background

In this section, we provide background information on the FiGA framework. Then, we summarize MaRTS and explain its limitation related to the impact of the inheritance hierarchy changes that we address in this paper.

2.1 FiGA Framework

The *Fine-Grained Adaptation* (FiGA) framework [6,7] allows a developer to adapt a running program on a standard JVM without stopping it. The developer modifies UML models and propagates the model changes to the source code. The program updating process is kept separate from the running program instance until the changes are ready to be compiled and loaded into the Java virtual machine, so as not to compromise the service provided by the program. The FiGA

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

framework uses ReverseЯ [5] to extract UML class and activity diagrams from source code and JavAdaptor [27, 28] to update a running Java program without stopping it. ReverseЯ extracts UML activity diagrams from an annotated running Java program. ReverseЯ uses a set of @Java annotations [8] designed to describe the diagrams and introduced during the code development. When the program is executed, these annotations drive the process of extracting activity diagrams from the running Java program.

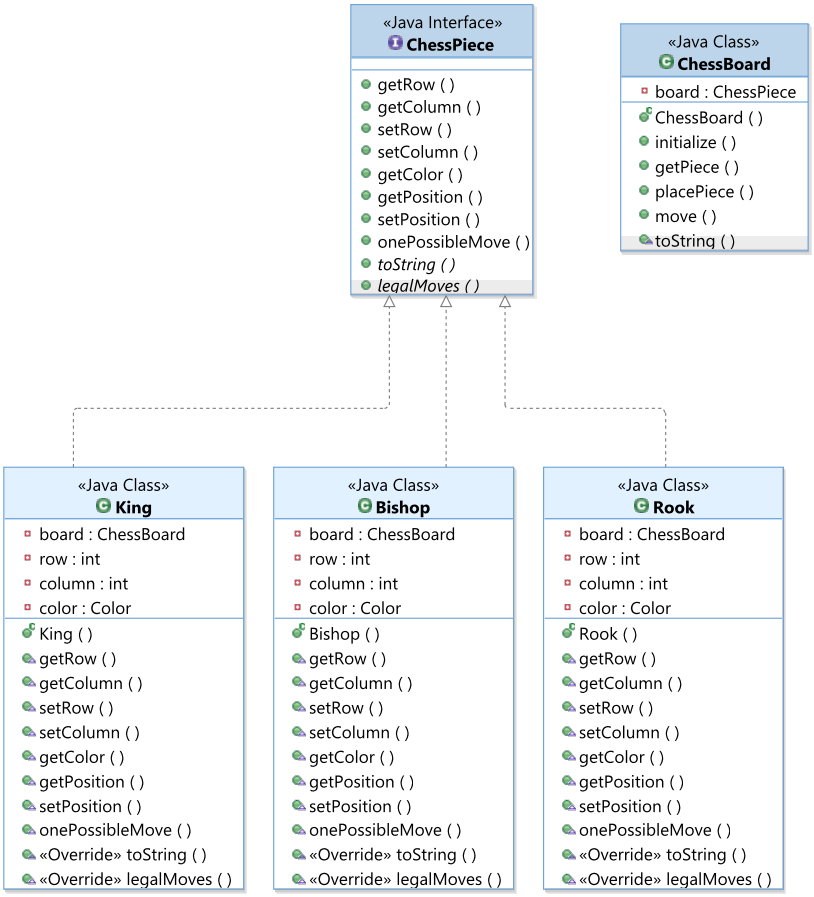

Fig. 1: Partial Chess Class Diagram Before Refactoring.

FiGA supports UML class and activity diagrams. The supported elements in the UML class diagram are classes, interfaces, and operations and attributes declared in the classes and interfaces, generalization and realization relations, and associations between classes. Fig. 1 shows a partial class diagram for a chess program, which is used as a running example in Section 2.3. Even though ReverseЯ supports the generation of associations, MaRTS does not use them. Instead it uses the attributes in the classes.

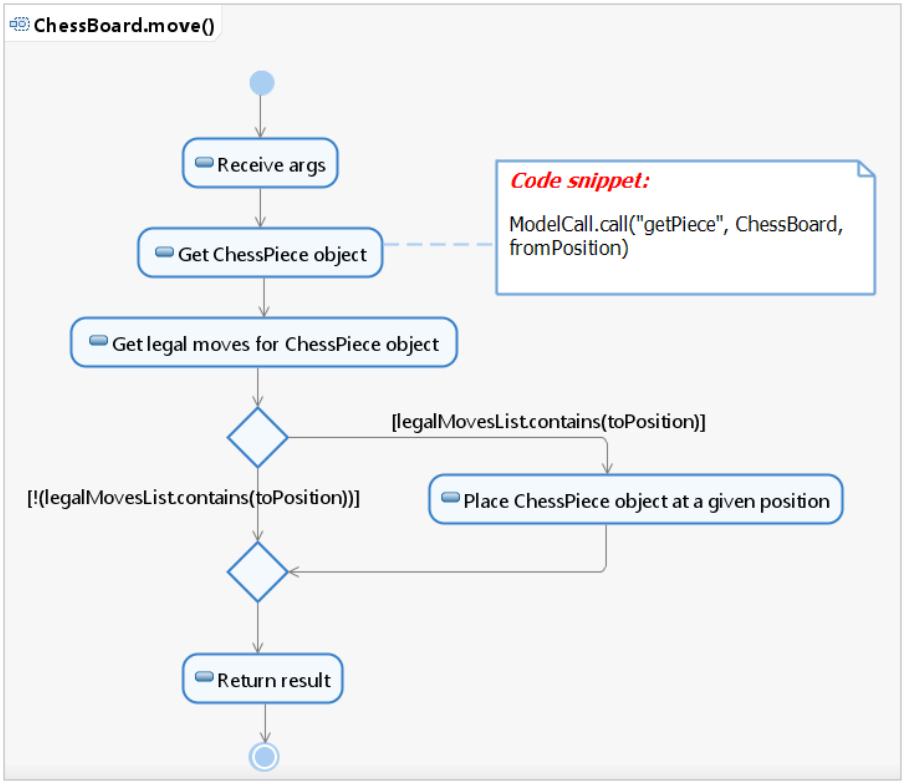

Fig. 2: UML Activity Diagram for the `ChessBoard.move()` Method.

```
public boolean move(String fromPosition, String toPosition) {
        ChessPiece cp = getPiece(fromPosition);
        ArrayList<String> legalMovesList = cp.legalMoves();
        boolean result = false;
        if (legalMovesList.contains(toPosition)) {
                result = placePiece(cp, toPosition);
        }
        return result;
  }
```

Listing 1: The `ChessBoard.move()` Method.

In FiGA, 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 UML activity diagram elements that are supported in FiGA are action nodes, call behavior nodes, decision and merge nodes, and initial and final nodes. An activity diagram generated using FiGA is detailed, where each action node in the activity diagram has a code snippet associated with it, and Java statements are contained inside the code snippet. Such activity diagrams are executable, i.e., when the model execu-

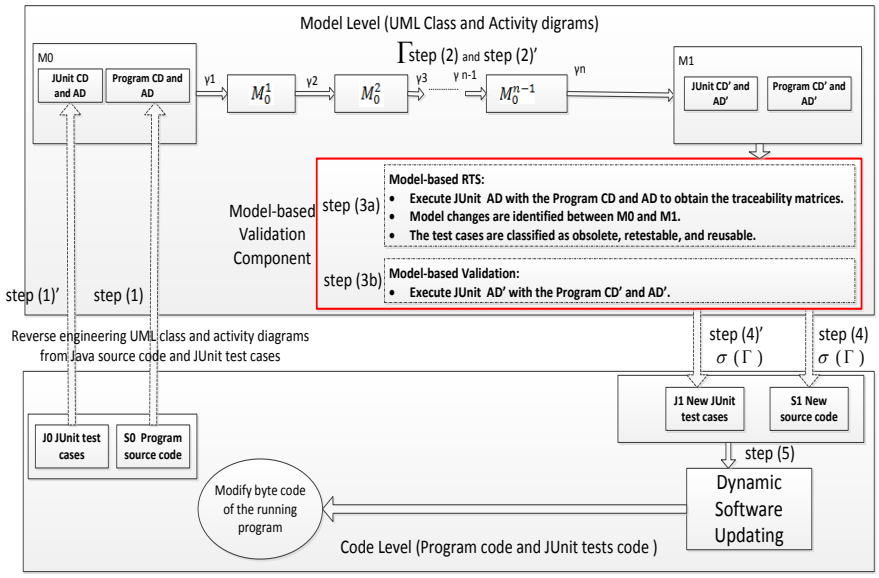

Fig. 3: Overview of the FiGA Framework with the Model-based Validation Component.

tion flow reaches an action node, then the code snippet associated with the action node is executed. Fig. 2 shows an example for a UML activity diagram representing `ChessBoard.move(String, String)` method, and shows an example for a code snippet associated with the action node labeled as *Get ChessPiece object*. Listing 1 shows the implementation of `ChessBoard.move(String, String)`. FiGA maps a code-level method call statement to a call statement to the activity diagram representing the called method, where the call statement to the activity diagram is added to a code snippet of an action node. For example, the code snippet shown in Fig. 2 contains a call statement to the activity diagram representing the `ChessBoard.getPiece(String)` method, where `ModelCall` is a class that we implemented to pass parameters between activity diagrams. When the model execution flow reaches that statement in the code snippet, then the activity diagram representing the `ChessBoard.getPiece(String)` method is called and the input parameter `fromPosition` is passed to it.

Fig. 3 shows the steps of the FiGA framework. The model-based validation component (bordered in red in Fig. 3) is not part of the FiGA framework. This component is proposed in our works [1, 2] and used within FiGA. The process to adapt a running program within FiGA is performed through a repetitive loop of four steps 1, 2, 4, and 5 (i.e., without taking into account step (3a) and step (3b)). Each step (except step 5) has a program code part indicated by (step (n)) and a test code part indicated by (step (n)'). The process starts by generating UML class and activity diagrams from the source code and JUnit test suites through ReverseЯ [5] (step (1) and step (1)'). Then, developers manually modify the models to support the needed adaptation (step (2) and step (2)'). Finally, a code patch is generated for

the code and the test suites from the modified models (step (4) and step (4)'), and the changes are deployed to the running application through JavAdaptor [27, 28] (step (5)). Details about the whole process can be found in [6, 7].

2.2 MaRTS

We proposed a model-based validation approach [1] based on executing the models representing the test cases with the models representing the program methods to validate the adaptation at the model level. This approach is used within the FiGA framework to support the validation of runtime adaptations performed at the model level. In particular, we added and used step (3b) shown in Fig. 3 within FiGA. In step (3b), The adapted models are validated through model execution [1]. The RSA simulation toolkit 9.0 was used to execute the UML activity diagrams in FiGA.

We addressed the problem of regression test selection at the model level on top of the FiGA framework [2]. We extended the model-based validation component [1] shown in Fig. 3 to include step (3a) that classifies regression test cases at the model level. The process to classify test cases at the model level, that we refer to as MaRTS, consists of the following steps [2]:

1. For each UML activity diagram representing a test case, *tc*, two traceability matrices are calculated: activity-level and flow-level. The former records which activity diagrams representing program methods are traversed by exercising *tc*; the latter records which transition-flows of the activity diagrams are traversed by exercising *tc*. A transition flow is an edge that starts an activity node after the previous one is finished.
2. Model changes are identified during the adaptation process using the RSA model comparison tool[2]. The class diagram changes that can be identified by the RSA model comparison tool 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.
3. The test cases are classified as obsolete, retestable, and reusable, according to whether or not they traverse modified activity diagrams or transition-flows of activity diagrams. Our RTS approach can identify only one type of obsolete test cases, those that contain a direct call to a deleted method.

A safe RTS technique must select all *modification-traversing* test cases for regression testing [29]. A test case is considered to be modification-traversing for a

[2] https://www.ibm.com/developerworks/rational/library/05/712_comp2/index.html

program P if it executes changed code in P, or if it formerly executed code that had been deleted in P [37]. 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 modification-traversing test case, then it will definitely be selected for regression testing [37]. Rothermel and Harrold [29] identified inclusiveness to evaluate the safety of regression test selection approaches. Inclusiveness measures the extent to which an RTS approach selects *modification-traversing* test cases for regression testing. The precision of an RTS approach measures to what extent an RTS approach omits *non-modification-traversing* test cases [29]. A non-modification-traversing test case is also a non-fault-revealing test case [29]. A precise RTS technique only selects those test cases that are relevant to a modification.

2.3 Lack of Support for the Inheritance Hierarchy Changes

MaRTS does not consider the impact of changes to the inheritance hierarchy when classifying the test cases. We illustrate this limitation using a chess program as an example. This program is a classroom project that only supports the basic functionality to create a chessboard and move chess pieces, and it does not use artificial intelligence to play with humans. Fig. 1 shows a partial class diagram for the chess program. For example, the classes `Bishop`, `King`, and `Rook` implement the `ChessPiece` interface.

```
class Test {
   public void testMovePawn(){
       ChessBoard board = new ChessBoard();
         board.initialize();
         board.move("c2", "c4");
       ....
   }
   public void testGetColumn() {
        ChessBoard board = new ChessBoard();
        board.initialize();
        ChessPiece piece=board.getPiece("d7");
        assertEquals(piece.getColumn(), 3);
   }

}
```

Listing 2: Chess Program Test Cases.

Listing 2 shows a test class that contains the test cases `testMovePawn` and `testGetColumn`. The `testMovePawn` test method creates a `ChessBoard` object and initializes the board with instances of `Pawn`, `Knight`, `King`, `Queen`, `Rook`, and `Bishop`. The test case then calls `board.move("c2", "c4")` to move an object on the board if the move is legal. As shown in Listing 1, `ChessBoard.move(String, String)` calls the `getPiece(fromPosition)` method that returns the chesspiece at the board position defined by the argument. `ChessPiece` is the static type of `cp`, but its dynamic type can be any class that implements the `ChessPiece` interface.

Currently, 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. Let us consider refactoring the chess program so that the code duplication is minimized. 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 [16], 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 along with the 48 activity diagrams representing them 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 shows a portion of the refactored classes for the chess program.

Fig. 4: Partial Class Diagram After Refactoring.

MaRTS misses the fact that `Pawn` class inherits `ChessPiece.getColumn()`, and that the `testGetColumn` test case calls `ChessPiece.getColumn()` instead of `Pawn.getColumn()`. The approach classifies the `testGetColumn` test case as obsolete because it contains a direct call to the deleted `Pawn.getColumn()` method. However, the

`testGetColumn` test case should be classified as retestable because in the modified program version, it calls the inherited `ChessPiece.getColumn()` method instead of the deleted method. We used the EvoSuite test generation tool[3] to generate JUnit test cases for each class of the chess program. A total of 130 test cases were generated. Based on the changes made to the class and activity diagrams, MaRTS classified 10 test cases as retestable and 120 test cases as obsolete. The 120 test cases were classified as obsolete because they contain direct calls to the deleted activity diagrams representing the deleted operations. However, these test cases are actually retestable because the subclasses inherit the eight new operations added to the `ChessPiece` class, and the test cases call the inherited operations in the modified program.

3 MaRTS with Support to Changes to the Inheritance Hierarchy

The extended MaRTS consists of five steps:

1. Extract the operations-table from the original class diagram (Section 3.1).
2. Calculate the traceability matrix (Section 3.2).
3. Identify model changes (Section 3.3).
4. Extract the operations-table from the adapted class diagram (Section 3.4).
5. Classify test cases (Section 3.5).

Steps 1 and 4 were specifically introduced to support the analysis of the inheritance hierarchy. Steps 2 and 5 modify the corresponding steps from the previous MaRTS approach [2]. In step 2, the activity-level traceability matrix now also stores the receiver type for each activity diagram call. Step 5 now takes into account the impact of class diagram changes on the inheritance hierarchy while classifying test cases. Step 3 is unchanged. All these steps are automated.

3.1 Extraction of the Operations-Table from the Original Class Diagram

This step is performed before adapting the models. An *operations-table* is extracted from the class diagram. This table stores for each class the operations that can be invoked on an object of that class type. For example, Table 1 shows part of the operations-table with the entries for the `Pawn` and `Knight` classes from Fig. 1.

The first column of Table 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 1 because the `Pawn` and `Knight` classes only implicitly inherit from `Object`.

The information stored in the operations-table can be used to determine which operations are inherited or overridden in each class. This information is used along with other inputs to classify test cases as explained later in Section 3.5.

[3] `http://www.evosuite.org/`

Table 1: Operations-table for the `Pawn` and `Knight` Classes Shown in Fig. 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>

3.2 Traceability Matrix Calculation

This step is performed before adapting the models. The activity diagrams representing the test cases are executed with the activity diagrams representing the program methods to record the coverage of test cases at the model level. The RSA simulation toolkit 9.0 is used to execute the models.

During model execution, four types of information are collected for each test case: (1) which activity diagrams are executed by the test case, (2) which activity diagrams are directly called, (3) what is the receiver object type for each executed activity diagram, and (4) 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.

Table 2 shows a portion of a flow-level traceability matrix. For the sake of simplicity, we omitted the information related to the receiver types and which activity diagrams are directly called by each test case. We show the traversed transition flows as a source-destination node pair. The `testGetColumn` test case traverses the flows of the `ChessBoard_initialize` and `Pawn_getColumn` activity diagrams, where in the third column of the table, `x` denotes that the transition flow in that row was traversed by the test case.

Table 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

Table 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>

3.3 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.

3.4 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 developers have fully adapted the class and the activity diagrams, an operations-table is extracted from the adapted class diagram.

Table 3 shows the operations-table for the `Pawn` and `Knight` classes after the chess program is refactored as explained in Section 2. Now the `Pawn` and `Knight` classes `extend` the `ChessPiece` class.

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
```

3.5 Test Case Classification

The goal of the whole process is to classify the test cases as obsolete, retestable or reusable. 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 skims the set of those 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 skimming 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*

(a) 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.

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 from Fig. 1 is classified by Algorithm 1 after the system is adapted as described in Sect. 2. Tables 1 and 3 show the operations-table before and after the adaptation respectively. In Table 1, the `Pawn` class contains `Pawn.getColumn()` that was moved to the `ChessPiece` class during the adaptation process. This is evident from Table 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 Experimental Setup

In this section we describe the goals and setup of the experiment whose results are illustrated and discussed in Section 5 to evaluate MaRTS. The goals of the evaluation were to (1) demonstrate that MaRTS[4] 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.

Inclusiveness measures the extent to which a regression test selection technique selects modification-traversing test cases 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', where P' is a modified version of a program 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 [29]. If the inclusiveness of an RTS approach is 100% for all programs, then it is considered to be safe.

Precision measures the extent to which a regression test selection approach omits test cases that are non-modification-traversing from the retestable set. Suppose a test suite contains T test cases, and N test cases among T are non-modification-traversing for P and P'. If the RTS approach omits M of these N test cases, then the precision of the RTS approach with respect to P, P', and T is M/N [29].

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

[4] From this point on, the name MaRTS indicates the version that supports changes to the inheritance hierarchy.

Table 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
Chess	0	7	1	0	6	65	1074
Siena	1.8	9	0	0	0	95	1605
XML-security	2	173	6	131	30	1172	16800

RQ1: Does MaRTS have less 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[5] and ChEOPSJ[6] on four subject programs. DejaVu is an implementation of the approach proposed by Harrold et al. [21] while ChEOPSJ is an implementation of the approach proposed by Soetens et al. [33]. Both tools support RTS for Java programs. We selected DejaVu because it is considered to be the state-of-the-art approach for code-based RTS in terms of safety and precision, 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 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.

4.1 Subject Programs

We used four subject programs: (1) graph package of the Java Universal Network/-Graph Framework (JUNG)[7], (2) Siena[8], (3) XML-security[9], and (4) chess program used as a motivating example in this paper. 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 generic types, and do not use multithreaded programming. We used EvoSuite [3] 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 summarizes the data about version

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

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

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

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

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

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 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.

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 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.

The chess program was presented in Section 2.3. We used EvoSuite to generate JUnit test cases for each class of the chess program. The tool generated 130 JUnit test cases that exercise 96% of the statements. We created class and activity diagrams for the original version of the chess program and its test cases.

4.2 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.

Table 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

JUNG. Table 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 `AbstractSparseVertex` 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 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 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 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 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.

Table 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

5 Evaluation and Discussion

In this section we answer the two previously listed research questions, discuss our findings, and identify threats to validity. We also provide a theoretical analysis of the time complexity of MaRTS.

5.1 Evaluation of Inclusiveness and Precision

Table 6 shows the results of running the three RTS approaches. We have prepared a link[10] for a zip archive that contains the code for the current implementation of MaRTS, code subjects used in the study, test cases generated by EvoSuite, models of the subjects and the test cases, the model difference files generated by RSA, the traceability matrix for each subject, and the fault detection ability data.

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 [33]. 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

[10] `http://www.cs.colostate.edu/~malref82/RTSExperiments.zip`

Table 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

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.

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. We found that 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`. We found that 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 run it on the XML-security subject because of a bug in this tool. It did not detect code changes that it is supposed to detect, and did not produce results. Table 6 and Table 7 do not show results for ChEOPSJ with respect to the XML-security subject.

Checking functionality after adaptation. For each adapted subject program, we executed the retestable test cases and they passed.

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 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 less than the inclusiveness and precision of DejaVu and ChEOPSJ.

5.2 Evaluation of the 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.

Table 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 second step, PIT[11] was used to apply first-order method-level mutation operators to versions 1.12 and 1.14. The applied mutation operators[12] 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 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 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.

5.3 Discussion

MaRTS achieved results comparable to those achieved by 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 paper 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 considered mutating the adapted methods, assuming that developers insert new faults only to these methods during the adaptation process.

[11] http://pitest.org

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

In general, MaRTS can support both functional and unit test cases. In our work, three subject programs (JUNG, Siena, and Chess) use unit test cases. The XML-security program uses functional test cases.

The key ideas of MaRTS are applicable to object-oriented programming languages in general even though currently MaRTS only supports Java 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 because it was designed to analyze the inheritance hierarchy in Java. We can overcome this limitation by extending the operations-table to store information for multiple inheritance, and modify the test classification algorithm accordingly.

MaRTS does not support the following features of Java due to limitations in the underlying tools: multithreaded programming, generic types, and new features introduced in Java 8 and 9, such as default methods in interfaces, functional interfaces, and lambda expressions. The lack of support for these features can result in missing modification-traversing test cases when MaRTS is applied to Java projects that use these features. 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 support the reverse engineering of the features and executing them at the model level.

MaRTS may produce incorrect results when Java reflection and multithreading are applied. These limitations also apply to all the existing code-based RTS approaches because dynamically collected dependencies for test cases may not cover all possible paths that can be traversed by the test cases [21, 29].

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-tables. Interfaces can still be used as types in the operations-table, i.e., return types and parameter types of operations.

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.

MaRTS supports exception handling. RSA can execute activity diagrams that contain action nodes for exception handling code. If a throw statement that is associated with an action node is executed, then the execution flow is transferred to an action node that contains the corresponding exception handler, i.e., catch block, and the exception handler is executed. The activity diagram flows that are incoming to the action nodes containing the throw statement and the exception handler are recorded in the traceability matrix.

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.

MaRTS can scale up to large programs because all of its steps are automated, and can be utilized by model-driven development approaches and models@run.time approaches that perform evolution and adaptation at the model level.

5.4 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.

5.5 Time Complexity for the Extended MaRTS

Even though we did not empirically compare the running times using our approach, we performed a theoretical analysis of its 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 [21] DejaVu approach uses the algorithm proposed by Rothermel and Harrold [30]. 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 $O(2 * n + T * n^2)$ [30], 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.

6 Related Work

The regression test selection problem has been studied for over three decades. Engström et al. [13] presented a systematic literature review of existing code-based RTS approaches, focusing on the approaches that have been empirically evaluated. They categorize the 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 relationships and dependencies between software parts, such as classes.

Below we summarize the existing code-based approaches that consider the impact of inheritance hierarchy changes on RTS. We also summarize the existing model-based RTS approaches and compare them with MaRTS. To the best of our knowledge, existing model-based RTS approaches do not consider the impact of inheritance hierarchy changes.

6.1 Code-based Approaches

Firewall approaches [22, 24, 35] are based on the concept of defining the entities of the system that need to be retested, i.e., drawing a conceptual firewall around these entities. These approaches select all the test cases that exercise at least one entity from the firewall. Kung et al. [24] and Hsia et al. [22] applied RTS at the class level for C++ programs. Their approach is based on the idea that in addition to retesting the changed classes, it is also necessary to retest classes that are directly or indirectly dependent on the changed classes. Their approach constructs an object relation diagram (ORD) that describes static relationships between classes, which are association, aggregation, and inheritance. The approach also instruments the original program to record the classes exercised by each test case. When a class, C, is modified, the ORD is used to find a set of all classes that have relationships with C. All the test cases traversing any of these classes are selected for regression testing. White and Abdullah [35] proposed a similar

approach that selects all the test cases exercising at least one class from a firewall of a modified class.

These firewall approaches can be imprecise because they may select many test cases that do not need to be re-executed on the modified program version. If some classes along the inheritance hierarchy are modified, then it is not necessarily the case that all the other classes along the same inheritance hierarchy are impacted. Test cases traversing these classes do not always execute modified or affected parts of the code. MaRTS is more precise than these firewall approaches because it can identify which operations along the inheritance hierarchy are affected by class model changes, and only those test cases traversing the affected operations called on specific receiver types are selected.

Skoglund and Runeson [31] found in empirical studies that the class-level firewall RTS is not a precise technique, i.e., it can select test cases that are non modification-traversing. They proposed an improved approach over the class-level firewall approach by removing the class firewall and using a change-based RTS technique that selects only those test cases that exercise the changed classes. The change-based RTS is a subset of the class-level firewall approach, where constructing the ORD and calculating the class firewall are excluded. The selection is, thus, limited to those test cases that traverse changed classes. In contrast, the selection process in MaRTS is limited to those test cases that traverse modified and impacted methods.

Soetens et al. [32] proposed a change-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. In contrast to MaRTS, this approach does not support the impact of changes along the inheritance hierarchy.

Soetens et al. [33] extended their approach to support dynamic binding when applying test selection. They extended their FAMIX model so that a method invocation has relationships with all the methods that this invocation can refer to based on the method name. This technique to specify such relationships is static and is based on method names. For example, if a change is made to a method named m, then all the methods in the program that carry the same name m are identified. Then the model is used to map each of the identified methods to its set of relevant test cases. This approach has another limitation in that it cannot classify test cases that traverse changed constructors [33]. ChEOPSJ is an implementation for the RTS approach proposed by Soetens et al. [33]. In contrast, MaRTS takes into account the receiver type of a method invocation and the method signature (not only the method name as in [33]) to classify test cases, which makes MaRTS more precise than ChEOPSJ.

Rothermel and Harrold [30] proposed a safe graph-walk 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 identically labeled CFG edges in P and P' differ, then those edges are added to the set of affected entities.

Harrold et al. [21] 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 [11] 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. DejaVu is an implementation for the RTS approach proposed by Harrold et al. [21] for Java programming language. The difference between MaRTS and DejaVu is that instead of using CFGs with call edges labeled with receiver types, MaRTS uses a different technique based on static analysis to the UML class diagrams to identify for each class the operations that can be invoked on an object of the class type. Our evaluation showed that DejaVu and MaRTS achieved comparable results.

6.2 Model-based Approaches

Chen et al. [9] use UML activity diagrams for specification-based RTS. In their work, an activity diagram represents the requirements and specifications of a system. Their approach is used to apply black-box RTS. In contrast, MaRTS uses activity diagrams to represent detailed program behaviors. MaRTS also uses class diagrams to represent the static structure of software, and applies impact analysis to identify the activity diagrams impacted by class changes.

Korel et al. [23] use control and data dependencies in an extended finite state machine to identify the impact of model changes and perform RTS. This approach does not use UML class diagrams.

Farooq et al. [14] use UML class and state machine models for RTS. This approach identifies changes in the class and the state diagrams, and uses the impacted and changed elements of the state diagrams to apply RTS. This approach does not support the identification of the addition and deletion of the generalization relations, and does not support the identification of operations that are overridden and inherited along the inheritance hierarchy.

Briand et al. [4] presented an RTS approach based on UML use case models, class models, and sequence models. Their approach is applied at the design level, in which test cases are selected according to design change information. This approach can identify the addition and deletion of generalization relations between classes. However, it does not identify which operations are inherited and overridden along the inheritance hierarchy. For example, suppose that in the original class diagram, a class C inherits an operation *op*, and in the adapted class diagram, C overrides *op*. Then, any test case traversing *op* on a receiver of type C is affected and needs to be selected because the changes to inherited and overridden operations

can indirectly affect test cases, which then need to be selected for safe regression testing. Briand et al.'s approach can miss such test cases because it does not detect that *op* is no longer inherited. The extended MaRTS approach can identify the changes to the inherited and overridden operations along the inheritance hierarchy, and therefore, identifies the affected test cases and select them.

Zech et al. [38] present a generic model-based RTS platform, which is based on the model versioning tool, *MoVE*. The approach consists of the three phases: change identification, impact analysis, and test case selection. These phases are controlled by OCL queries. Similar to Briand et al. [4], this approach does support the identification of inherited and overridden operations along the inheritance hierarchy.

Yenigün et at. [36] considered the existing definitions of data dependencies between transitions in an extended finite state machine (EFSM), which represents the requirements of a system under test. They proposed new definitions and rules for identifying data dependencies in EFSM; these rules consider the generation and removal of data dependencies caused by multiple modifications. The new rules can be utilized for impact analysis in model-based RTS approaches that use EFSM.

To the best of our knowledge, MaRTS is the first RTS approach that provides a framework for using UML class and activity diagrams, where the models are executable. MaRTS improves over the other model-based approaches by considering the impact of inheritance hierarchy changes on the test classification process.

7 Conclusions and Future Work

In this work, we presented an extension to MaRTS—model-based approach for regression test selection—that adds support for changes relevant to the inheritance hierarchy and its impact on the selection of test cases.

MaRTS was evaluated on four applications and compared with two code-based RTS approaches, DejaVu and ChEOPSJ, that consider the impact of changes on the inheritance hierarchy for Java software. DejaVu is a safe graph-walk based RTS approach and ChEOPSJ is based on static analysis. For each of the four subject programs, MaRTS was able to (1) identify which operations are inherited and overloaded in each class based on the changes made to the class diagram, and (2) accordingly select all the relevant test cases. MaRTS and DejaVu selected all the modification-traversing test cases. MaRTS identified obsolete test cases for one of the subject programs. DejaVu and ChEOPSJ did not address the identification of obsolete test cases. MaRTS and DejaVu outperformed ChEOPSJ in terms of inclusiveness and precision, e.g., ChEOPSJ omitted some modification-traversing test cases. Thus, MaRTS should be useful to developers using model-based techniques for evolving or adapting software.

We also demonstrated through a mutation testing experiment that the reduced test sets obtained by MaRTS achieved the same fault detection ability that was achieved by the full test sets.

We plan to evaluate the inclusiveness and precision of MaRTS on additional subject programs. We will also evaluate the fault detection ability of the reduced test sets obtained by the approach, and the efficiency of the approach in terms of the reduction in regression testing time. We will develop a formal proof for safety and precision of MaRTS in the future.

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

We would like to acknowledge the support of Quinten Soetens who helped us install and run ChEOPSJ.

References

1. Al-Refai, M., Cazzola, W., Ghosh, S., France, R.: Using Models to Validate Unanticipated, Fine-Grained Adaptations at Runtime. In: H. Waeselynck, R. Babiceanu (eds.) Proceedings of the 17th IEEE International Symposium on High Assurance Systems Engineering (HASE'16), pp. 23–30. IEEE, Orlando, FL, USA (2016)
2. Al-Refai, M., Ghosh, S., Cazzola, W.: Model-based Regression Test Selection for Validating Runtime Adaptation of Software Systems. In: L. Briand, S. Khurshid (eds.) Proceedings of the 9th IEEE International Conference on Software Testing, Verification and Validation (ICST'16), pp. 288–298. IEEE, Chicago, IL, USA (2016)
3. Arcuri, A., Campos, J., Fraser, G.: Unit Test Generation During Software Development: EvoSuite Plugins for Maven, IntelliJ and Jenkins. In: L. Briand, S. Khurshid (eds.) Proceedings of the 9th IEEE International Conference on Software Testing, Verification and Validation (ICST'16), pp. 401–408. IEEE, Chicago, IL, USA (2016)
4. Briand, L.C., Labiche, Y., He, S.: Automating Regression Test Selection Based on UML Designs. Journal on Information and Software Technology **51**(1), 16–30 (2009)
5. Cazzola, W., Pini, S., Ghoneim, A., Saake, G.: Co-Evolving Application Code and Design Models by Exploiting Meta-Data. In: Proceedings of the 22$^{\text{nd}}$ Annual ACM Symposium on Applied Computing (SAC'07), pp. 1275–1279. ACM Press, Seoul, South Korea (2007)
6. Cazzola, W., Rossini, N.A., Al-Refai, M., France, R.B.: Fine-Grained Software Evolution using UML Activity and Class Models. In: A. Moreira, B. Schätz (eds.) Proceedings of the 16$^{\text{th}}$ International Conference on Model Driven Engineering Languages and Systems (MoDELS'13), Lecture Notes in Computer Science 8107, pp. 271–286. Springer, Miami, FL, USA (2013)
7. Cazzola, W., Rossini, N.A., Bennett, P., Pradeep Mandalaparty, S., France, R.B.: Fine-Grained Semi-Automated Runtime Evolution. In: N. Bencomo, B. Chang, R.B. France, U. Aßmann (eds.) MoDELS@Run-Time, Lecture Notes in Computer Science 8378, pp. 237–258. Springer (2014)
8. Cazzola, W., Vacchi, E.: @Java: Bringing a Richer Annotation Model to Java. Computer Languages, Systems & Structures **40**(1), 2–18 (2014). DOI 10.1016/j.cl.2014.02.002
9. Chen, Y., Probert, R.L., Sims, D.P.: Specification-Based Regression Test Selection with Risk Analysis. In: D.A. Stewart, J.H. Johnson (eds.) Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'02), pp. 1–14. IBM Press (2002)
10. Cottenier, T., van den Berg, A., Elrad, T.: Motorola WEAVR: Aspect Orientation and Model-Driven Engineering. Journal of Object Technology **6**(7), 51–88 (2007)
11. Dean, J., Grove, D., Chambers, C.: Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In: W.G. Olthoff (ed.) Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95), LNCS 952, pp. 77–101. Springer, Åarhus, Denmark (1995)
12. Dzidek, W.J., Arisholm, E., Briand, L.C.: A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance. IEEE Transactions on Software Engineering **34**(3), 407–432 (2008)
13. Engström, E., Runeson, P., Skoglund, M.: A Systematic Review on Regression Test Selection Techniques. Information and Software Technology **52**(1), 14–30 (2010)
14. Farooq, Q.u.a., Iqbal, M.Z.Z., I Malik, Z., Riebisch, M.: 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), pp. 41–49. IEEE, Oxford, UK (2010)
15. Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., Gjørven, E.: Beyond Design Time: Using Architecture Models for Runtime Adaptability. IEEE Software **23**(2), 62–70 (2006)
16. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading, Massachusetts (1999)

17. France, R.B., Rumpe, B.: Model-Driven Development of Complex Software: A Research Roadmap. In: L.C. Briand, A.L. Wolf (eds.) Proceedings of Future of Software Engineering (FoSE'07), pp. 37–54. IEEE Computer Society, Minneapolis, MN, USA (2007)
18. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure. IEEE Computer **37**(10), 46–54 (2004)
19. Georgas, J.C., van der Hoek, A., Taylor, R.N.: Using Architectural Models to Manage and Visualize Runtime Adaptation. IEEE Computer **42**(10), 52–60 (2009)
20. Harrold, M.J.: Testing Evolving Software. Journal of Systems and Software **47**(2-3), 173–181 (1999)
21. Harrold, M.J., Jones, J.A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S.A., Gujarathi, A.: Regression Test Selection for Java Software. In: J. Vlissides (ed.) Proceedings of the 16th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01), pp. 312–326. ACM, Tampa, FL, USa (2001)
22. Hsia, P., Li, X., Kung, D.C.H., Hsu, C.T., Li, L., Toyoshima, Y., Chen, C.: A Technique for the Selective Revalidation of OO Software. Journal of Software: Evolution and Process **9**(4), 217–233 (1997)
23. Korel, B., Tahat, L.H., Vaysburg, B.: Model Based Regression Test Reduction Using Dependence Analysis. In: G. Antoniol, I.D. Baxter (eds.) Proceedings of the International Conference on Software Maintenance (ICSM'02), pp. 214–223. IEEE, Montréal, Quebec, Canada (2002)
24. Kung, D.C., Gao, J., Hsia, P., Toyoshima, Y., Chen, C.: On Regression Testing of Object-Oriented Programs. Journal of Systems and Software **32**(1), 21–40 (1996)
25. Leung, H.K.N., White, L.J.: Insights into Regression Testing. In: Proceedings of Conference on Software Maintenance, pp. 60–69. IEEE, Miami, FL, USA (1989)
26. Morin, B., Barais, O., Jézéquel, J.M., Fleurey, F., Solberg, A.: Models@Run.time to Support Dynamic Adaptation. IEEE Computer **42**(10), 44–51 (2009)
27. Pukall, M., Grebhahn, A., Schröter, R., Kästner, C., Cazzola, W., Götz, S.: JavAdaptor: Unrestricted Dynamic Software Updates for Java. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE'11), pp. 989–991. IEEE, Waikiki, Honolulu, Hawaii (2011)
28. Pukall, M., Kästner, C., Cazzola, W., Götz, S., Grebhahn, A., Schöter, R., Saake, G.: JavAdaptor — Flexible Runtime Updates of Java Applications. Software—Practice and Experience **43**(2), 153–185 (2013). DOI 10.1002/spe.2107
29. Rothermel, G., Harrold, M.J.: Analyzing Regression Test Selection Techniques. IEEE Transactions on Software Engineering **22**(8), 529–551 (1996)
30. Rothermel, G., Harrold, M.J.: A Safe, Efficient Regression Test Selection Technique. ACM Transactions on Software Engineering and Methodology **6**(2), 173–210 (1997)
31. Skoglund, M., Runeson, P.: Improving Class Firewall Regression Test Selection by Removing the Class Firewall. International Journal of Software Engineering and Knowledge Engineering **17**(3), 359–378 (2007)
32. Soetens, Q.D., Demeyer, S., Zaidman, A.: Change-Based Test Selection in the Presence of Developer Tests. In: A. Cleve, F. Ricca (eds.) Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR'13), pp. 101–110. IEEE, Genoa, Italy (2013)
33. Soetens, Q.D., Demeyer, S., Zaidman, A., Pérez, J.: Change-Based Test Selection: An Empirical Evaluation. Empirical Software Engineering pp. 1–43 (2015)
34. Vogel, T., Giese, H.: Adaptation and Abstract Runtime Models. In: Proceedings of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'10), pp. 39–48. ACM, Cape Town, South Africa (2010)
35. White, L.J., Abdullah, K.: 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 (1997)
36. Yenigün, H.: Identifying the Effects of Modifications as Data Dependencies. Software Quality Journal **22**(4), 701–716 (2014)
37. Yoo, S., Harman, M.: Regression Testing Minimization, Selection and Prioritization: A Survey. Journal of Software Testing, Verification and Reliability **22**(2), 67–120 (2012)
38. Zech, P., Felderer, M., Kalb, P., Breu, R.: A Generic Platform for Model-Based Regression Testing. In: T. Margaria, B. Steffen (eds.) Proceedings of the 5th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'12), Lecture Notes in Computer Science 7609, pp. 112–126. Springer, Heraclion, Crete (2012)