2016 IEEE International Conference on Software Testing, Verification and Validation

Model-based Regression Test Selection for Validating Runtime Adaptation of Software Systems

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

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

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

***Abstract*—An increasing number of modern software systems need to be adapted at runtime without stopping their execution. Runtime adaptations can introduce faults in existing functionality, and thus, regression testing must be conducted after an adaptation is performed but before the adaptation is deployed to the running system. Regression testing must be completed subject to time and resource constraints. Thus, test selection techniques are needed to reduce the cost of regression testing.**

The FiGA framework provides a complete loop from code to models and back that allows fine-grained model-based adaptation and validation of running Java systems without stopping their execution. In this paper we present a model-based test selection approach for regression testing during the validation activity to be used with the FiGA framework. The evaluation results show that our approach was able to reduce the number of selected test cases, and that the model-level fault detection ability of the selected test cases was never lower than that of the original test cases.

***Index Terms*—model-based regression test selection, model-based validation, model-based dynamic adaptation, executable UML models, JUnit tests, unanticipated adaptation.**

I. Introduction

The ability to perform runtime adaptations is becoming a requirement for many highly available software systems. These systems need to be adapted without stopping them. For example, intelligent transportation systems must provide continuous services to human and software clients for safety reasons.

Runtime adaptation is needed based on changes in a software system context and requirements. These changes can be foreseen at design time, and in this case, anticipated adaptations are prepared during development and included in the system design. When unforeseen changes discovered at runtime, unanticipated adaptations must be planned and deployed into the running system. In both cases, the runtime adaptation process is complex, and models can be used to manage this complexity by representing aspects of a running system at a higher abstraction level to ease the planning process [1].

Model-based approaches focus on using models at runtime to support self-adaptation in autonomous systems, and is dominated by work in the self-adaptation area [2], [3], [4], [5], [6], [7]. These approaches support anticipated coarse-grained adaptations that are controlled and automated by the MAPE feedback loop in autonomous systems [8]. Adaptation in these approaches is restricted to the addition, deletion, and reconnection of components. On the other hand, unanticipated adaptations require human intervention. These adaptations involve code changes, and can be fine-grained at the level of classes, methods, and statements.

Models that provide a fine-grained view of the running system and its implementation can be used by developers to plan fine-grained adaptations to support unforeseen changes. For example, the *fine grained adaptation* (FiGA) framework supports unanticipated and fine-grained adaptations of a running system at the model level [9], [10].

Runtime adaptations, both anticipated and unanticipated, can affect existing functionality and can introduce faults to the running system. Therefore, regression testing needs to be performed to ensure that the modified parts of the software behave as intended [11]. Regression testing needs to be performed before the deployment of the adaptation.

Regression testing is one of the most expensive activities performed during the lifecycle of a software system, and regression test selection (RTS) [11] is one strategy to make regression testing more efficient and effective. RTS is defined as the activity of selecting a subset of test cases from an existing test set to verify that the affected functionality of a program is still correct [11], [12]. For runtime adaptations, testing must be performed under tighter time constraints and resources than what is normally associated with pre-deployment testing. Thus, it is important to reduce the number of regression test cases that must be re-executed.

RTS can be based on the analysis of code or model level changes of a software system. Model-based RTS techniques can be more efficient and convenient for the approaches that already use models to apply adaptations at runtime. Additionally, the use of model-based RTS techniques is growing, and will have crucial importance in the future for several reasons: (1) model-based approaches can scale up better than code-based approaches for large software systems [13], (2) maintaining traceability at the model level can be more practical compared to maintaining traceability at the code level because dependencies are specified at a higher level of abstraction [12], and (3) it is easier to analyze the changes between different versions of models compared to the changes between different code versions [12]. However, to the best of our knowledge, none of the studied model-based approaches for runtime adaptation support RTS at the model level.

We propose a new model-based approach for regression test selection, which is used at runtime for regression testing of unanticipated fine-grained adaptations performed at the model level. The proposed approach uses (1) class models and activity models to represent fine-grained behaviors of a software system and its test cases, and (2) a fine-grained model comparison tool to identity model changes during the adaptation process. The proposed approach exploits model execution, which is used to record the coverage information for each test case at the activity model level. The approach can apply RTS at different levels of granularity. The proposed approach is applied within the *fine grained adaptation* (FiGA) framework [9], [10] that supports unanticipated and fine-grained adaptation of a running system through model adaptation.

In Sect. II we provide an overview of the FiGA framework. In Sect. III we describe the proposed regression test selection approach. Evaluation of the proposed approach is described in Sect. IV. Related work is presented in Sect. V, and we conclude with plans for further work in Sect. VI.

II. Background of the FiGA Framework

The *Fine-Grained Adaptation* (FiGA) framework [9], [10] allows a developer to adapt a program that is running on a standard JVM without stopping it by modifying UML models and propagating model changes to the source code. The program change 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 framework uses ReverseЯ [15] to generate UML models from the source code, and uses the JavAdaptor [16] tool to modify the running Java program without stopping it.

JavAdaptor works at a low level, requiring as an input the compiled version of the class to update and a connection to the Java virtual machine in which the program is executing. JavAdaptor is responsible for updating the running byte code while preserving the program state.

We extended the FiGA framework to support the validation of runtime adaptations at the model level [14]. The rest of Sect. II summarizes the extended FiGA framework. The FiGA framework that is integrated with the validation component is shown in Fig. 1. In this framework, the adaptation and validation of a running program is performed through a repetitive loop that is composed of five steps. This process can be repeated whenever the application needs to be updated. Each step (except step(3b) and step(5)) has a program code part indicated by (step(n)) and a test code part indicated by (step(n)'). Both parts of a step are performed in parallel.

Step (1): Model Generation from Program Code. ReverseЯ [15] is used to generate the UML models from the application source code.

Step (1)': Model Extraction from Test Suites. Assuming that the original application has a pre-deployment code level test suite to validate the application before of its adaptation, ReverseЯ extracts UML class and activity models from the test suite of the application. Each individual test case is represented as an activity model.

Step (2): Modification of Program Model. Developers change the models to deal with the needed adaptation. Each model change can be expressed as a sequence of elementary model changes (γ_i) that can be easily and automatically mapped to a code change (δ_i). The model changes are determined by model differencing [17], [18], and mapped to calls to the change operators with the proper parameters.

Step (2)': Modification of Test Suite Model. Developers adapt the models of the test cases to specify new test configurations and assertions in order to validate the adapted models of the application.

Step (3): Adaptation Process for Program. The sequence of elementary model operations and their mappings to program code changes are defined formally as follows: Let S_0 represent source code for a running Java program and M_0 its UML model. M_1 represents the model obtained after adapting M_0, and S_1 represents the program source code obtained after propagating model changes to S_0. Let $\boxplus$ be the *change sequencing operator*: $M_1 = M_0 \boxplus \Gamma$, where Γ is a composition of change operations expressed with model operators γ_i, each representing an elementary change such that $\Gamma = \gamma_1 \boxplus \gamma_2 \boxplus \cdots \boxplus \gamma_i \boxplus \cdots \boxplus \gamma_n$.

We define Δ as those changes necessary to adapt the source code to the system modeled by M_1 such that $S_1 = S_0 \boxplus \Delta$ where Δ is obtained by composing the elementary changes (δ_i) on the code: $\Delta = \delta_1 \boxplus \delta_2 \boxplus \cdots \boxplus \delta_i \boxplus \cdots \boxplus \delta_n$.

Each model change operator has a mapping to a corresponding elementary code change. Therefore, the composition of model change operators (Γ) expresses the changes made at the model level, and the composition of code change operators (Δ) expresses the corresponding set of code level changes. The σ function maps the set of model changes to code changes, such that $\Delta = \sigma(\Gamma)$. Details of the adaptation semantics are given in Section 2 of our previous work [10].

Step (3)': Adaptation Process for Test Suites. The definitions in step (3) also apply to the sequence of elementary model operations and their mappings to test suite code changes.

Step (3b): Validation of Adapted Program Model. In the FiGA framework, the models representing the program and the test suite are executable [14]. The models representing the test suite are executed with the models representing the program. If the execution of a test model fails, the developer needs to modify the models representing the program to fix the faults (i.e., goes back to step 2 and 2'). The validation process is completed when all the models for the test cases pass, after which the adaptation can be deployed.

Step (4): Propagating Changes to the Program. Model changes are propagated to the running program only when all required model changes are performed. The sequence of model changes is determined by model differencing between M_0 and M_1, and mapped into calls to code change operators with the proper parameters.

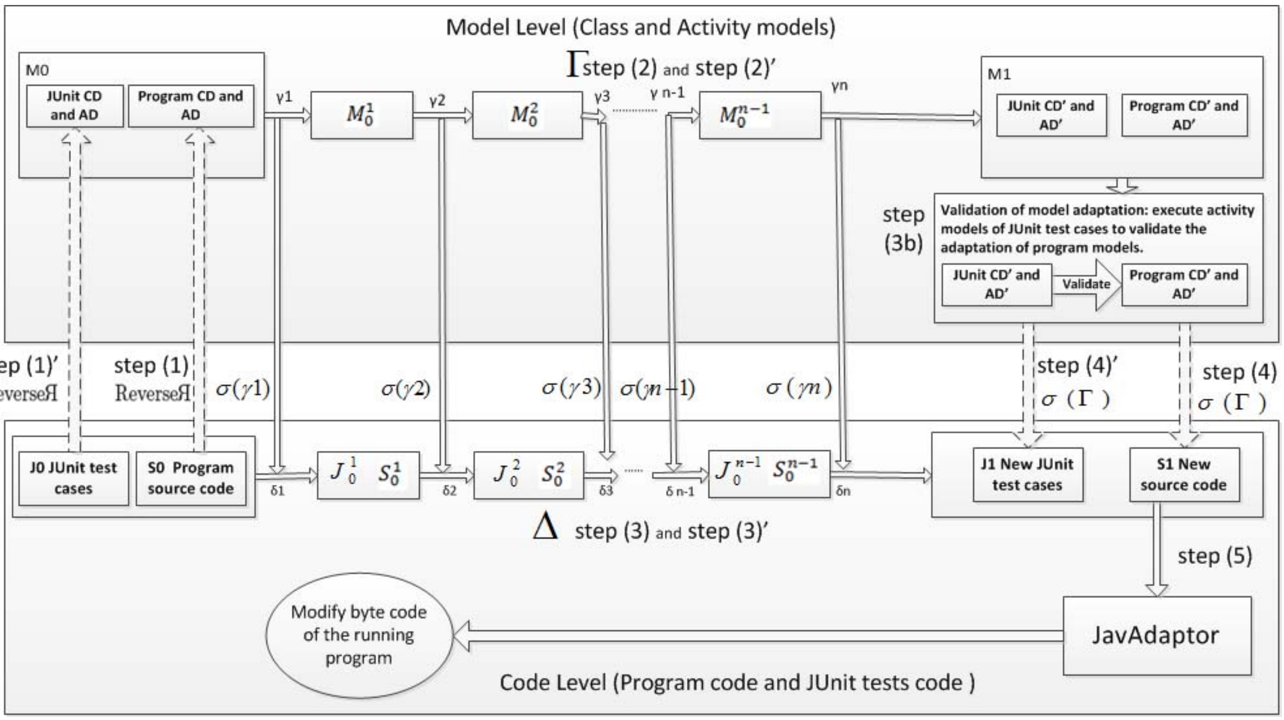

Fig. 1. Overview of the FiGA Approach [14].

Step (4)': Propagating Changes to the Test Suites. Similar to step (4), the test suite models are also kept synchronized with the test suite code.

Step (5): Updating the Running Application using JavAdaptor. The modified Java classes are selected as input to the JavAdaptor tool, which is triggered to deploy program changes to the running application without stopping it while preserving its state [16]. At this point, future adaptations occur by starting again from step 1 but with the new source code and test suite instead.

In FiGA framework, UML class and activity models are used to represent, adapt, and execute test cases. The current FiGA framework considers JUnit test cases. Each individual test case is represented as an activity model. The FiGA framework exploits the *Rational software architect* (RSA) simulation toolkit 9.0[1] to execute the activity models. We added the Java and JUnit libraries to the tool and enabled model execution using Java. JUnit assertions are evaluated at the model level. The FiGA framework supports the following mappings from the code level to the model level in order to integrate with the RSA model simulation tool:

1) Mapping Java statements to code snippets of action nodes and transition flows [14]. When the model execution flow reaches an action node, the code snippet associated with the node is executed. Similarly, when the model execution flow reaches a decision node, its outgoing transition flows are evaluated and the transition that evaluates to `True` is executed.
2) Mapping method calls to calls between activity models representing these methods [14]. Formal parameters of an operation are represented as local attributes in the activity model that implements it. These attributes are used to pass actual parameters between calls to activity models during their execution. We implemented a technique to pass caller objects and actual parameters to the calls between activity models. If an activity model representing the `A::m` method has a loop that repeatedly calls an activity model representing the `B::n` method, then different instance of the called activity model is uniquely identified for each call and the technique to pass parameters passes actual parameters to it.

The executable models in our approach provide abstraction over code and have the benefits of ease of understanding and visualizing, but are complete and precise enough to be executable like code. For example, each action node of an activity model is associated with an executable code snippet that contains Java statements, and each transition flow that is outgoing from a decision node has a boolean statement associated with it.

The validation process includes regression testing and testing new behaviors introduced during adaptation. Our prior work [14] addresses the validation of new behaviors and adaptation of test cases at the model level. The focus of this paper is on regression testing. In this work we present

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

a new model-based regression test selection approach that can be used within the FiGA framework. The steps of the proposed RTS approach start after `step(1)` and `step(1)'`, and end before `step(3b)`. The process of selecting test cases for regression testing needs to be performed before the execution of regression test cases in `step(3b)`. The new RTS approach is described in the following section.

III. Model-Based Regression Test Selection

The proposed model-based regression test selection approach consists of three steps:

1) Calculate a traceability matrix that relates activity models representing the test cases to activity models representing methods of the program.
2) Identify the changes made to the original class and activity models to obtain the adapted ones.
3) Classify test cases represented as activity models into the three categories: obsolete, retestable, and reusable, as defined by White [19]. Obsolete test cases are invalid and cannot be executed on the modified version of the software. Retestable test cases are still valid, and exercise the modified parts of the software. Retestable test cases need to be re-executed for regression testing to be safe. Reusable test cases only exercise unmodified parts of the program, and thus, while they are still valid, they do not need to be re-executed to ensure safe regression testing. A safe regression test selection technique must select all `modification-traversing` test cases for regression testing [20]. A test case is considered to be `modification-traversing` for a program `P` if it executes changed code in `P`, or if it formerly executed code that had been deleted in `P` [13].
   A safe RTS approach is not defined to be safe from all possible faults (e.g., some program changes might cause side effects on other unmodified parts of the program). A safe RTS means that, if there exists a `modification-traversing` test case, then it will be selected for regression testing [13].

A. Calculate the Traceability Matrix

This step is performed after the class and activity models are generated from the original software system and its JUnit test suite, and before adapting the models, i.e., after `step (1)` and `step (1)'` in Fig. 1. The activity models representing the JUnit test cases are executed with the activity models representing methods of the program. During execution, two types of information are collected: (1) what activity models are exercised by each test case, and (2) what flows in each activity model are exercised by each test case.

The first type of information is used to calculate the traceability matrix at a coarser granularity level. We refer to it as activity-level traceability matrix. The second type of information is used to calculate the traceability matrix at a finer granularity level. We refer to it as flow-level traceability matrix. The activity-level traceability matrix is used to classify test cases according to whether they traverse modified/unmodified activity models. The flow-level traceability matrix is used to classify test cases according to whether they traverse modified/unmodified transition flows in activity models.

```
Diff1. Add (a2<Opaque Action>)(b2<Opaque Action>)<Control Flow>
       to Activity1<Activity>: Edge
Diff2. Add arg1<Property> to Activity1<Activity>.ownedAttribute: Property
Diff3. Add m1<Call Behavior Action> to Activity1<Activity>: Node
Diff4. Modify a2<Opaque Action>.body: from "varArray[i]=var;"
       to "varArray[++i]=var;"
```

Fig. 2. An example for RSA model diff report.

B. Identify the Model Changes

The models generated by ReverseЯ are compliant with the Rational Software Architect (RSA) modeling tool. The proposed RTS approach uses the RSA model comparison tool to identify model changes. When developers make changes to the models, this tool identifies changed model elements and types of changes made to them. The class model changes that can be identified by the RSA model comparison tool are:

1) Addition, deletion, and modification of class attributes and operations.
2) Addition, deletion, and modification of classes and relations between classes.

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

1) Addition and deletion of action nodes, call behavior action nodes, decision and merge nodes, and start and end nodes.
2) Modification of action nodes based on changes to code stored in a code snippet associated with an action node.
3) Addition and deletion of transition flows.
4) Modification of a transition flow, and modification of a boolean expression associated with a transition flow.
5) Addition, deletion, and modification of attributes of an activity model.
6) Addition, deletion, and modification of an activity model.

The changes to class and activity models identified by the RSA model comparison tool are saved in a *model diff report*. Fig. 2 shows an example of a *model diff report* where `Diff1` states that a transition flow is added from action node a_2 to action node b_2 in `Activity1` activity model. `Diff2` states that the attribute arg_1 is added to the activity model `Activity1`. `Diff3` states that a call behavior action node labeled with m_1 is added to the activity model `Activity1`. `Diff4` states that the code statement `varArray[i]=var` of the action node a_2 is modified to `varArray[++i]=var`.

C. Classify the Test Cases

The process to classify test cases takes four inputs: (1) the flow-level traceability matrix, (2) the activity-level traceability matrix, (3) the *model diff report* that is generated by the RSA model comparison tool, and (4) a set of all activity models representing the program methods; let us call this set as the *activityModelsSet*. Based on these inputs, the activity models representing the test cases are classified as obsolete, retestable, and reusable as follows:

1) Obsolete: In our approach, an activity model of a test case is classified as obsolete if (1) it contains a direct call to a deleted activity model, or (2) it contains a direct call to an activity model with modified attributes (because the test inputs do not match the type and/or number of the model attributes).
   The test cases are classified as follows: for each change in the *model diff report*, if the change involves the deletion of an activity model or the deletion/addition/modification of an owned attribute of an activity model, then the activity-level traceability matrix is used to find all test cases that traverse that activity model. If any of these test cases has a direct call to that activity model, then the test case is classified as obsolete.
   Test cases corresponding to activity models that are classified as obsolete need to be modified or deleted from the test suite. Our approach does not identify all types of obsolete test cases, such as when the original test oracle (corresponding to JUnit assertions) does not conform to the new software specification and needs to be changed. Supporting the identification of such cases by our approach is planned for future work.
2) Retestable: Classification of test cases as retestable can be performed at the activity-level and at the flow-level. In the first case, the activity-level traceability matrix is used, and the test cases are classified as follows:
   - For each change in the *model diff report* that involves the deletion or modification of an activity model, the traceability matrix is used to find all test cases that traverse the deleted/modified activity model, and these test cases are classified as retestable (if these test cases do not have direct calls to the deleted activity model).
   - For each change in the *model diff report* that involves the deletion or modification of a class attribute, the *activityModelsSet* is used to find all activity models that access the attribute. Next, the traceability matrix is used to find all test cases that traverse any of these activity models, and these tests are classified as retestable.

   In the second case, the flow-level traceability matrix is used. A transition flow in the *model diff report* or in the flow-level traceability matrix is labeled by its source and destination nodes. The test cases are classified as follows:
   - For each change in the *model diff report* that involves the deletion or modification of a transition flow, the traceability matrix is used to find all test cases that traverse this transition flow, and these tests are classified as retestable.
   - For each change in the *model diff report* that involves the addition of a transition flow, if the source node of the transition flow is not identified as added in the *model diff report* (the source node exists in the original models before the adaptation), then the traceability matrix is used to find all test cases that traverse any transition flow ending in this source node, and these tests are classified as retestable.
   - For each change in the *model diff report* that involves the deletion or modification of a node, the traceability matrix is used to find all test cases that traverse any transition flow ending in the node, and these tests are classified as retestable.
   - For each change in the *model diff report* that involves the deletion or modification of a class attribute, the *activityModelsSet* is used to find all action nodes that access the attribute. Next, the traceability matrix is used to find all test cases that traverse any transition flow ending in any of these action nodes that access the attribute, and these tests are classified as retestable.
3) Reusable: Test cases that are not classified as obsolete or retestable are classified as reusable. These tests do not traverse modified or deleted elements of activity models, so they are unnecessary for safe regression testing.

D. Tool Implementation

We implemented a prototype tool for our approach. The tool consists of two components. The first component is a processor that is used to collect coverage information for test cases at the model level. This component is applied after the models are obtained from the program via ReverseЯ and before these models are adapted. The RSA model simulation tool generates an executable representation of the models that are obtained via ReverseЯ. In particular, the RSA tool generates Java code that represents the UML models, and this code contains links to the UML model files (e.g., EMX files of the UML models), and contains calls to the API of the RSA model simulation tool. The processor component is used to instrument the executable representation of the UML models with statements that write to a file, during model execution, the names of the executed activity models, flows, and nodes. Let us call this file as the `execution history file`.

Executing activity models representing test cases with the activity models representing methods of the program (as explained in Sect. III-A) produces an `execution history file` that contains for each test case the names of all activity models and their elements that were traversed by the test case.

The second component implements the test classification technique described in Sect. III-C. The second component reads the `execution history file` and prepares the flow-level and activity-level traceability matrices. Then, it reads the other inputs (the *model diff report* and *activityModelsSet*) and classify the test cases as explained in Sect. III-C. This component is applied after the models of the program are adapted and before the validation process is started (after `step (2)` and before `step (3b)`).

IV. Evaluation and Discussion

The goals of this study were to (1) evaluate the reduction in the number of test cases that can be achieved using the proposed model-based RTS (Sect. IV-B), (2) evaluate the time

TABLE I
ORIGINAL ARS AND NANOXML 2.0, AND THEIR MODELS

Software System	Number of classes	Number of methods	LOC	Number of baseline tests	Number of activity models of methods	Number of activity models of test cases
ARS	9	58	721	61	43	61
NanoXML	19	147	5827	97	59	97

TABLE II
ADAPTED MODELS OF ARS AND NANOXML

Software System	Change type	Classes	Activity models of methods
ARS	*Added*	0	5
	Deleted	0	0
	Modified	4	14
NanoXML	*Added*	2	16
	Deleted	0	0
	Modified	6	13

saved by executing the reduced test suite (taking into account the execution time of the RTS approach) compared to when the full test suite is executed (Sect. IV-C), and (3) evaluate the fault detection ability of the reduced test suite at the model level (Sect. IV-D).

A. Description of the Subject Programs

We evaluated the proposed model-based RTS approach on two subjects: an airline reservation system (ARS) and the NanoXML[2] parser. The ARS is a prototype system that was implemented by undergraduate students, and it is an example of a highly-available system that cannot be stopped at runtime. The ARS implements features related to flight reservations. A flight is characterized by the airline, the departure and destination airports, and its schedule. Each customer can look for a flight and reserve a seat on it. In our design, a `SystemManager` class provides access to all the ARS functionality; including the access to airport, airline, and flight instances. The baseline JUnit test suite of the original ARS contains 61 test cases that provide 100% branch coverage. Table I summarizes the data about the ARS and the extracted activity models from the ARS program and its baseline test cases.

The first version of our ARS implementation has a limitation in its *flight reservation* functionality by supporting reservations only on direct flights. The class and activity models of the ARS software were adapted to support reservation on flights with stops.

We chose NanoXML because (1) it was implemented by a third party, (2) its size is larger than the size of ARS, and (3) to evaluate how the model-based RTS approach would work when code-based adaptations to support new functionality are applied at the model level (multiple versions of NanoXML are available[3] and each adding new functionality to the previous version). NanoXML is an XML parser written in Java. Version 2.1 supports parsing XML namespaces that is not supported in version 2.0. The NanoXML package provides XML files to be used as inputs for self testing. We created 97 code level test cases for NanoXML 2.0 such that each test case contains statements to read and parse an input XML file. Each file contains XML data with different properties than XML data in other files (i.e., different number of XML entities, children, and attributes), and therefore, each test case contains different assertion statements to check the properties of the results. The test suite provides 81% statement coverage in NanoXML version 2.0. Table I summarizes the data about the NanoXML version 2.0.

NanoXML version 2.0 was adapted at the model level to version 2.1. First, the code changes between both versions were extracted by differencing the code for version 2.0 with the one for version 2.1. Next, the class and activity models were extracted from version 2.0 and its baseline test cases. An activity model was extracted for each of the 97 test cases. Table I shows the data about the extracted activity models. Thereafter, the code changes between version 2.0 and 2.1 were applied on the class and activity models representing version 2.0 to get version 2.1 at the model level.

B. Reduction in the Number of Test Cases

We applied the proposed model-based RTS approach within the FiGA framework on the adapted models of ARS and NanoXML to evaluate the reduction in the number of test cases compared to the number of test cases in the complete test suite for each system.

ARS study: Table II summarizes data about the adapted models of the ARS to support reservation on flights with stops. Five new activity models were added to implement new methods, and 14 of the existing activity models were modified. The modifications involved adding/deleting nodes and transition flows between them, and modifying Java statements in code snippets of action nodes. Formal input parameters for the `Airline::bookSeat()` operation were modified, and the corresponding attributes in its activity model were also modified.

The RTS approach was applied at the activity-level and at the flow-level. The results are shown in Table III. Both the activity-level RTS and flow-level RTS selected the same set of obsolete test cases, which are 4 test cases that contain direct calls to the activity model of the method `Airline::bookSeat()`, where the input formal parameters to this method were modified. The activity-level RTS selected 23 out of 61 test cases as retestable, and the flow-level RTS selected 18 test cases out of 61 test cases as retestable.

NanoXML study: Table II summarizes the data about the adapted models of NanoXML 2.0 to get to version 2.1 at the model level. The RTS approach was applied at the activity-level and at the flow-level. The results are shown in Table III. Both the activity-level RTS and flow-level RTS classified the same set of test cases (75 out of 97) as retestable. The reason

[2]http://nanoxml.sourceforge.net/orig/
[3]http://sir.unl.edu/portal/bios/nanoxml.php

TABLE III
RESULTS OF THE MODEL-BASED RTS APPROACH

Software System	Level of RTS	Obsolete	Retestable	Reusable
ARS	Activity-level RTS	4	23	34
	Flow-level RTS	4	18	39
NanoXML	Activity-level RTS	0	75	22
	Flow-level RTS	0	75	22

for getting similar results at both levels is that for most of the modified activity models, specifically the activity models representing methods of the *XMLElement* and *StdXMLParser* classes, all test cases traversing a modified activity model are also traversing modified elements (i.e., nodes and transitions flows) in that model.

The RTS approach did not classify any test case as obsolete because the 97 test cases do not contain direct calls to deleted activity models or to activity models with modified attributes.

Evaluation of the safety of the RTS approach: In the ARS and NanoXML studies, if the set of test cases classified as retestable by the model-based RTS approach contains all *modification-traversing* test cases, then the RTS approach is considered to be safe for these studies. In our model-based RTS approach, a test case is considered to be a *modification-traversing* if it executes changed elements in the adapted models, and/or it formerly executed elements that had been deleted in the adapted models. We investigated the execution of the 61 test cases of the ARS on the original and adapted models, and recorded all the test cases that executed changed/added/deleted elements. This set of 18 test cases is called the *modification-traversing* set for the ARS. Similarly, we applied the same process on the 97 test cases of the NanoXML and extracted the *modification-traversing* set which consists of 75 test cases.

In the ARS study, the activity-level RTS selected 23 test cases as retestable, which included all the 18 test cases in the *modification-traversing* set. However, it selected 5 more test cases that traverse modified activity models, but they do not traverse the modified elements (modified edges and nodes) in these activity models. The flow-level RTS selected 18 test cases out of 61 test cases as retestable, which were exactly the same test cases in the *modification-traversing* set.

For NanoXML, both the flow-level and activity-level RTS, selected the same test cases (75 tests) as retestable, which were exactly the same test cases in the *modification-traversing* set. The remaining test cases do not traverse modified activity models.

C. Reduction in Time

The reduction in time is measured as the difference between the (1) time taken to execute all the original test cases and (2) the time it takes to run the test classification algorithm (Sect. III-C) and execute the selected test cases. We excluded from the calculation the time required to obtain the traceability matrix (Sect. III-A) and to identify model changes (Sect. III-B). The reason is that these two steps are performed during the adaptation process and they do not consume any time during the testing phase. Additionally, all model differencing is already performed in the FiGA framework to generate new source code and update the running system, so it is not an extra step that is only required by the RTS functionality.

We measured the reduction in time for the ARS and NanoXML studies when the flow-level RTS approach was applied on them (see in Sect. IV-B). The reduction in time was 1.756 seconds for the ARS study, and 1.1179 seconds for the NanoXML study. The time to classify test cases was 0.1240 seconds for ARS and 0.1740 seconds for NanoXML when the flow-level traceability matrix was used to classify the test cases.

D. The Fault Detection Ability of the Reduced Test Set

To evaluate the fault detection ability of the reduced test set, we compared the number of mutants killed by the reduced test set with the number of mutants killed by running the full test set.

We performed two mutation testing experiments. The adapted version of the NanoXML at the model level (the adapted models representing NanoXML version 2.1 as explained in Sect. IV-B) is the subject of the two experiments. The goal of the first experiment was to evaluate the fault detection ability of the reduced test set that is calculated based on changes made to all adapted models of the NanoXML 2.1.

In the second experiment, the goal was to evaluate the fault detection ability of a reduced test set that is calculated based on changes made to the models of a specific class. That is, for each adapted class, the model-based RTS approach was used to calculate a reduced test set by only considering changes made to the activity models representing methods for that class. Then, the fault detection ability of the reduced test set for each class was compared with the fault detection ability of the full test set.

Mutation experiment when RTS was applied on the adapted model as a whole: The experiment consists of four steps. In the first step, the code level NanoXML version 2.1 is available[4], and the code level baseline test suite (the 97 test cases) was executed on it and all tests passed. Similarly, the corresponding model level test cases (the activity models representing the 97 test cases) were executed on the adapted models representing NanoXML version 2.1 and all tests passed. A set of activity models representing test cases can be executed in a single run as follows. Calls to all activity models representing test cases are added to an activity model, and when this activity model is executed by the RSA model simulation tool, all of the called activity models inside it are automatically executed.

In the second step, the `PIT` tool[5] was used to apply first-order method-level mutation operators to the code level NanoXML 2.1. There are no tools (to the best of our knowledge) that support generating mutations at the model level. The

[4]http://sir.unl.edu/portal/bios/nanoxml.php
[5]http://pitest.org

TABLE IV
MUTATION RESULTS WHEN RTS WAS APPLIED ON THE WHOLE ADAPTED MODELS

Mutations Introduced/Killed	**Number**	**Mutation Score**
Mutants in M	466	NA
Mutants killed by MT	310	67%
Mutants killed by RT	306	66%

TABLE V
MUTATION RESULTS PER CLASS, WHEN RTS WAS APPLIED ON EACH ADAPTED CLASS

Class name (c_i)	**Number of tests in** T_{c_i}	**Number of mutants in** M_{c_i}	**Mutation score of** T_{c_i}	**Mutation score of** MT
XMLElement	75	66	64%	64%
StdXMLParser	65	108	70%	70%
StdXMLBuilder	65	22	73%	73%
StdXMLReader	53	39	49%	59%
NonValidator	63	64	69%	69%
XMLWriter	52	61	48%	48%

`PIT` tool modifies the bytecode in memory (bytecode mutator), and generates a mutation report that shows information about all applied mutations, such as the location (i.e., class name, method name, and line number) of a mutated code and the change made to that code. The applied mutation operators were[6]: (1) Conditionals Boundary Mutator, (2) Increments Mutator, (3) Invert Negatives Mutator, (4) Math Mutator, (5) Negate Conditionals Mutator, and (6) Void Method Calls Mutator.

In the third step, we repeated each mutation on a copy of the class and activity models representing the NanoXML 2.1. For each mutation in the report generated by the `PIT` tool, we found the code line for that mutation in the NanoXML 2.1 program. Then, we found the corresponding code statement of that code line at the model level, and applied the same mutation to it. The code statement at the model level can be associated with an action node or a transition flow. For example, based on the report, we found that in the `XMLElement::addChild()` method of the NanoXML 2.1 program, the conditional expression `(child == null)` of an IF statement at line 350 was changed by the Negate Conditionals Mutator to `(child != null)`. We detected the decision node representing the IF statement in the activity model representing the `XMLElement::addChild()` method, and negated the conditional expressions that are associated with the outgoing transition flows of the decision node. Each mutation was applied on a copy of the class and activity models representing the NanoXML 2.1. Let us call the set of all mutated copies as *M*.

The set of activity models representing the baseline test cases for the NanoXML (the 97 test cases) is called *MT*. Because no changes were made to the test cases in *MT* and no new test cases were added to improve the test coverage, the test cases were not able to kill all model level mutants in *M*. The reduced set of activity models representing the test cases that were classified as retestable by our model-based RTS approach (the 75 test cases as shown in Table III) is called *RT*.

In the fourth step, the test cases of *MT* were executed on all mutants in *M*, and the killed mutants were reported. Similarly, the test cases of *RT* were executed on all mutants in *M*, and the killed mutants were reported. The results are shown in Table IV.

The test cases in *RT* achieved a mutation score that is close to the mutation score achieved by all test cases in *MT*, which indicates that the fault detection ability for the reduced test cases was comparable to the fault detection ability of the all test cases. The test cases of *MT* killed 4 more mutants in *M* than the test cases of *RT*. These 4 mutants were added in non-adapted activity models. We found that the test cases that killed these 4 mutants were not selected as retestable because they do not traverse adapted models.

Most of the live mutants (466-310) were not killed because they are not covered by the 97 test cases. These test cases were created based on the provided XML files, and they do not provide 100% coverage in the NanoXML code. The live mutants could be killed by improving the coverage of the test suite and checking additional internal states that are not checked by the existing test cases.

Mutation experiment when RTS was applied on each adapted class: In this experiment, we set up the test selection process to select a reduced test set for each adapted class as follows. The flow-level RTS process calculated a reduced test set for each adapted class c_i of NanoXML 2.1 by only considering changes made to the activity models for c_i. The reduced test set for c_i is called T_{c_i}. Next, for each adapted class c_i, the `PIT` tool was used to apply first-order method-level mutation operators only to c_i methods, and each applied mutation was repeated on a copy the class and activity models representing the NanoXML 2.1. Let us call the set of these mutated copies as M_{c_i}. We applied the six mutation operators listed previously.

Finally, for each c_i, the test cases of T_{c_i} were executed on all mutants in M_{c_i}, and the killed mutants were reported. Similarly, the set of all test cases (the set *MT* of the first mutation experiment) were executed on all mutants in M_{c_i}, and the killed mutants were reported. The results are shown in Table V.

We applied this study only on the 6 classes that were adapted to get to NanoXML 2.1 from version 2.0. Applying the model-based RTS approach for each class showed reduction in the number of retestable test cases per class. For example, the reduced test set for the *StdXMLReader* class consists of 53 out of 97 test cases. For five of these classes, the test cases of T_{c_i} and *MT* achieved equal mutation scores. The reduced test set for the *StdXMLReader* class achieved a smaller mutation score compared to *MT*. We found that the mutants that were killed

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

by *MT* but not by the reduced test set of the *StdXMLReader* class are the same 4 mutants that were not killed by the test cases in *RT* in the previous mutation experiment.

The two mutation experiments showed that the reduced test sets, which were calculated based on the adaptation made to the whole NanoXML models and based on the adapted models of each individual class, achieved mutation scores that were equal or close to the mutation scores achieved by all test cases. These results are promising but cannot be generalized to other software systems. We plan to apply the proposed RTS approach on additional systems and try different fault types.

E. Threats to Validity

We identify factors that might affect the outcome of the RTS approach.

Internal validity: In the FiGA framework, models are generated from annotated code. Developers can annotate the code at different granularity levels. For example, a method implementation can be annotated to represent each single statement as an action node in the activity model, or can be annotated to represent a couple of statements as an action node. If a program is annotated at a finer level of granularity such that each single statement is represented as a node in an activity model, then the size of the models and the flow-level traceability matrix will increase, and affect the result for reduction in time. Annotating a program at a coarser level of granularity might reduce the precision of the RTS approach, in which the RTS approach classifies test cases that are not `modification-traversing` as retestable.

The current approach does not support all changes that can be made to the class model, such as changes to the inheritance hierarchy, which might lead to some retestable test cases being omitted by the RTS approach.

Construct validity: The RTS approach is based on flow-level and activity-level coverage criteria. There are other coverage criteria (e.g., def-use criteria) that can be used to select test cases for regression testing, which were not explored in this work.

External validity: The experiment was performed using two subject programs, and therefore, our results cannot be generalized to other programs.

V. Related Work

This section discusses related work for code-based and model-based RTS approaches.

Code-based approaches: Many graph-walk approaches address the problem of RTS. Rothermel and Harrold [21] proposed a safe approach for RTS for procedural software. 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 flow of control between statements. Affected entities by modifications are selected by traversing in parallel the CFGs of *P* and *P'*, and when the target entities of like-labeled CFG edges in *P* and *P'* differ, then the edge is added to the set of affected entities.

Rothermel and Harrold extended the CFG-based algorithm for C++ using the Inter-procedural Control-Flow Graph (ICFG) [22]. The analysis algorithm in this approach works only for complete programs. When classes interact with other classes, the called classes must be fully analyzed. The analysis algorithm cannot be applied to programs that call external libraries, unless the code of the libraries is available. Harrold, Jones, Li, and Liang adopted a similar approach for Java software using the Java Inter-class Graph as representation [23], which handles incomplete programs and does not require complete analysis of the external libraries that are used.

These graph-walk approaches are known to be safe and precise when they apply RTS at the edge level (i.e., select test cases that traverse affected edges in the graph). Our approach is similar to these approaches in terms of its ability to apply RTS at the flow-level. For example, our approach selects test cases that traverse edges that are connected to modified action nodes, and test cases that formerly traversed deleted edges. However, our approach uses activity models instead of CFGs, where each of our action nodes can represent multiple code statements. This can make our approach less precise than graph-walk approaches that represent each statement as a separate node.

Firewall approaches [13], [24], [25] are based on the concept of drawing a firewall around the entities of the system that need to be retested. Kung et al. [24] applied RTS at the class level. This approach constructs an object relation diagram (ORD) that describes static relationships among classes, such as inheritance and association. The approach reports the classes executed by each test cases. The firewall of a class *C* is defined as a set of all classes that are dependent on *C*. When *C* is modified, then all test cases traversing any of the classes in its firewall are selected. Our approach can work at a finer level of granularity (i.e., flow-level).

Jang et al. [25] applied RTS at the level of method to C++ software. They identify firewalls around all affected methods by a change and select all test cases exercising these methods for regression testing. Our approach can select test cases at a finer level of granularity than the method level.

Vokolos and Frankl [26] considered RTS based on text differencing using Unix `diff`. The approach compares the original code version with the modified version to identify the modified statements, and selects test cases that exercises code blocks containing these statements.

Soetens et al. [27] proposed an RTS approach that is based on a change model that identifies fine-grained changes made to Java software as change objects according to FAMIX model. The FAMIX model represents software entities such as packages, classes, methods, and attributes. Dependencies between software entities are defined in the change model, and these dependencies are exploited for test selection. The change model can detect additions, removals, and modifications of packages, classes, methods, attributes, and method invocation statements. Each detected change is mapped to its set of relevant test cases based on change dependence hierarchy defined in the change model. This work applied mutation testing to evaluate the fault detection ability of the reduced test suite, and the evaluation showed that the reduced test suite and the complete test suite achieved comparable results.

Model-based approaches: Chen et al. [28] used UML activity models for specification-based RTS. In their work, an activity model represents requirements and specifications of a system. Changes to the specifications result in modifications to the activity model, and these modifications at the model level drive the selection of test cases. Their approach is used to apply black-box (specification-based) RTS. Our proposed approach is different than this approach because we use activity models to represent program behaviors.

Briand et al. [12] presented a technique for RTS based on UML use case models, and class and sequence models. Their approach is applied at the design level, in which test cases are selected according to design change information. Their approach identifies changes in the three types of models and the impact these changes have on the test cases. This approach supports functional testing, in which each test case triggers operations belonging to interface classes. Such operations are represented as use cases in the use case model, and each use case is connected to sequence models that represent the interaction scenarios of that use case. Korel et al. [29] used control and data dependencies in an extended finite state machine to identify the impact of model changes and perform RTS.

Farooq et al. [30] used UML class and state machine models for RTS. This approach identifies changes in the class model and in the state diagrams, and uses the impacted and changed elements of the state diagrams to apply RTS.

Zech et al. [31] presented 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, which are controlled by OCL queries. This tool was demonstrated on a small program as a proof of concept, and the results showed that the OCL queries for impact analysis are complex even for simple rules. Some of the presented model-based RTS approaches implemented prototype tools, such as START [30] and RTSTool [12]. These tools do not support UML activity models.

The proposed RTS approach improves over the presented model-based RTS approaches in terms of safety. In these model-based approaches, certain changes to method implementations may not be visible and detectable at the model level, such as a change to an operation implementation that does not modify its contract and signature [12]. Therefore, existing model-based RTS approaches are not safe with respect to such changes [12]. In our approach, such changes can be performed in activity models and can be identified and used for RTS, i.e., our approach identifies changes made to code statements associated with action nodes.

To the best of our knowledge, our proposed approach is the first approach that uses UML class and activity models to perform RTS, in which the models are executable, and RTS can be applied at different granularity levels.

VI. Conclusions and Future Work

In this work, we presented a new model-based approach for regression test selection. The proposed approach can be applied at runtime to select regression test cases to validate the adapted models of a running software system. The approach exploits model execution to obtain a traceability matrix between models of test cases and models of the program, and uses a model comparison tool to identify model changes. The approach can be used to apply regression test selection at different levels of granularity in activity models.

The proposed approach was applied within the FiGA framework, and was evaluated using two subject programs. The results showed that the proposed approach resulted in a reduction in the number of test cases that are selected for regression testing, and a reduction in time. The mutation testing results showed that the fault detection ability of the reduced test set was close to the fault detection ability of the full test set.

We plan to evaluate the safety and efficiency of the proposed approach on additional subject programs. We will support the identification of other types of changes (e.g., modifications to the inheritance hierarchy) at the model level and other coverage criteria (e.g., def-use criteria). We will evaluate the overhead required when the RTS approach is applied within the FiGA framework in terms of the effort spent on annotating code, reverse engineering, and applying changes at the activity model level.

Acknowledgment

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

References

[1] R. B. France and B. Rumpe, “Model-Driven Development of Complex Software: A Research Roadmap,” in *Proceedings of Future of Software Engineering (FoSE’07)*, L. C. Briand and A. L. Wolf, Eds. Minneapolis, MN, USA: IEEE Computer Society, May 2007, pp. 37–54.

[2] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjørven, “Beyond Design Time: Using Architecture Models for Runtime Adaptability,” *IEEE Software*, vol. 23, no. 2, pp. 62–70, Mar. 2006.

[3] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, “Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure,” *IEEE Computer*, vol. 37, no. 10, pp. 46–54, Oct. 2004.

[4] J. C. Georgas, A. van der Hoek, and R. N. Taylor, “Using Architectural Models to Manage and Visualize Runtime Adaptation,” *IEEE Computer*, vol. 42, no. 10, pp. 52–60, Oct. 2009.

[5] B. Morin, O. Barais, J.-M. Jézéquel, F. Fleurey, and A. Solberg, “Models@Run.time to Support Dynamic Adaptation,” *IEEE Computer*, vol. 42, no. 10, pp. 44–51, Oct. 2009.

[6] T. Vogel and H. Giese, “Adaptation and Abstract Runtime Models,” in *Proceedings of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS’10)*. Cape Town, South Africa: ACM, May 2010, pp. 39–48.

[7] T. Cottenier, A. van den Berg, and T. Elrad, “Motorola WEAVR: Aspect Orientation and Model-Driven Engineering,” *Journal of Object Technology*, vol. 6, no. 7, pp. 51–88, Aug. 2007.

[8] J. O. Kephart and D. M. Chess, “The Vision of Autonomic Computing,” *IEEE Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[9] W. Cazzola, N. A. Rossini, P. Bennett, S. Pradeep Mandalaparty, and R. B. France, “Fine-Grained Semi-Automated Runtime Evolution,” in *MoDELS@Run-Time*, ser. Lecture Notes in Computer Science 8378, N. Bencomo, B. Chang, R. B. France, and U. Aßmann, Eds. Springer, Aug. 2014, pp. 237–258.

[10] W. Cazzola, N. A. Rossini, M. Al-Refai, and R. B. France, "Fine-Grained Software Evolution using UML Activity and Class Models," in *Proceedings of the 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS'13)*, ser. Lecture Notes in Computer Science 8107, A. Moreira and B. Schätz, Eds. Miami, FL, USA: Springer, 29th of Sep.-4th of Oct. 2013, pp. 271–286.

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

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

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

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

[15] W. Cazzola, S. Pini, A. Ghoneim, and G. Saake, "Co-Evolving Application Code and Design Models by Exploiting Meta-Data," in *Proceedings of the 12th Annual ACM Symposium on Applied Computing (SAC'07)*. Seoul, South Korea: ACM Press, on 11th-15th of Mar. 2007, pp. 1275–1279.

[16] M. Pukall, C. Kästner, W. Cazzola, S. Götz, A. Grebhahn, R. Schöter, and G. Saake, "JavAdaptor — Flexible Runtime Updates of Java Applications," *Software—Practice and Experience*, vol. 43, no. 2, pp. 153–185, Feb. 2013.

[17] Z. Xing and E. Stroulia, "Differencing Logical UML Models," *Automated Software Engineering*, vol. 14, no. 2, pp. 215–259, Jun. 2007.

[18] S. Maoz, J. O. Ringert, and B. Rumpe, "ADDiff: Semantic Differencing for Activity Diagrams," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE'11)*, A. Zeller, Ed. Szeged, Hungary: ACM, Sep. 2011, pp. 179–189.

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

[20] G. Rothermel and M. J. Harrold, "Analyzing Regression Test Selection Techniques," *IEEE Transactions on Software Engineering*, vol. 22, no. 8, pp. 529–551, Aug. 1996.

[21] G. Rothermel, M. J. Harrold, "A Safe, Efficient Regression Test Selection Technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 2, pp. 173–210, Apr. 1997.

[22] G. Rothermel, M. J. Harrold, and J. Dedhia, "Regression Test Selection for C++ Software," *Software Testing, Verification and Reliability*, vol. 10, no. 2, pp. 77–109, Jun. 2000.

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

[24] D. C. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen, "On Regression Testing of Object-Oriented Programs," *Journal of Systems and Software*, vol. 32, no. 1, pp. 21–40, Jan. 1996.

[25] Y. K. Jang, M. Munro, and Y. R. Kwon, "An Improved Method of Selecting Regression Tests for C++ Programs," *Journal of Software Maintenance and Evolution*, vol. 13, no. 5, pp. 331–350, Sep./Oct. 2011.

[26] F. Vokolos and P. G. Frankl, "Empirical Evaluation of the Textual Differencing Regression Testing Technique," in *Proceedings of the International Conference on Software Maintenance (SM'98)*, Bethesda, MD, USA, Nov. 1998, pp. 44–53.

[27] Q. D. Soetens, S. Demeyer, and A. Zaidman, "Change-Based Test Selection in the Presence of Developer Tests," in *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR'13)*, A. Cleve and F. Ricca, Eds. Genoa, Italy: IEEE, Mar. 2013, pp. 101–110.

[28] Y. Chen, R. L. Probert, and D. P. Sims, "Specification-Based Regression Test Selection with Risk Analysis," in *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'02)*, D. A. Stewart and J. H. Johnson, Eds. IBM Press, Sep. 2002, pp. 1–14.

[29] B. Korel, L. H. Tahat, and B. Vaysburg, "Model Based Regression Test Reduction Using Dependence Analysis," in *Proceedings of the International Conference on Software Maintenance (SM'02)*. Montreal, Quebec, Canada: IEEE, Oct. 2002, pp. 214–233.

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

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