MaRTS: A Model-Based Regression Test Selection Approach

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

***Abstract*—Models can be used to plan the evolution and runtime adaptation of a software system. Regression testing of the evolved and adapted models is important to ensure that the previously tested functionality is not broken. Regression testing is performed with limited time and resource constraints. Thus, regression test selection (RTS) techniques are needed to reduce the cost of regression testing. Existing model-based RTS approaches cannot detect all types of fine-grained changes that can be made at a low level of abstraction, and they do not consider the impact of inheritance hierarchy changes on the selection of test cases.**

We propose a model-based RTS approach called MaRTS that classifies test cases based on changes performed to UML class and activity diagrams. It supports both fine-grained and inheritance hierarchy changes. We compared MaRTS with two code-based RTS approaches using four applications. MaRTS achieved results comparable to a dynamic code-based RTS approach (DejaVu), and outperformed a static code-based RTS approach (ChEOPSJ). The fault detection ability of the selected test cases was equal to that of the baseline test cases.

***Index Terms*—inheritance hierarchy, model-based adaptation, model-based regression test selection, UML activity diagram, UML class diagram**

I. Introduction

Regression testing is one of the most expensive activities performed during the lifecycle of a software system [1], [2]. Regression test selection (RTS) [3] is an approach that improves regression testing efficiency and reduces regression testing time by selecting a subset of the original test set for regression testing [3], [4].

RTS approaches can be based on the analysis of code or model-level changes of a software system. Model-based RTS has some advantages over code-based RTS. First, it enables early estimation of the effort required for regression testing [4]. Second, it can scale up better than code-based RTS approaches for large scale software systems [5]. Third, model-based RTS techniques can be more convenient for approaches that already apply evolution/adaptation at the model level because both the evolution/adaptation and test selection processes can be performed at the same level of abstraction [6].

Existing model-based RTS approaches suffer from the following limitations. First, they cannot detect all types of fine-grained changes from UML class, sequence, and state machine diagrams used in these approaches [4], [7], [8]. An example of such a change is a modification to an operation implementation that does not affect the operation's signature and contract [4]. Fine-grained changes are those that can be made at a low level of abstraction, such as changes to a statement inside a method implementation. Second, they do not support the identification of changes to inherited and overridden operations along the inheritance hierarchy [4], [7], [8], which leads to situations where relevant test cases that traverse such inherited and overridden methods are not selected for regression testing.

We propose a model-based RTS approach called MaRTS to be used for regression testing of unanticipated fine-grained adaptations performed at the model level. MaRTS uses UML design class and activity diagrams to represent behaviors of a software system and its test cases. MaRTS is based on (1) static analysis of the UML class diagram to identify the changes in the inheritance hierarchy, (2) fine-grained model comparison to identify changes performed to UML class and activity diagrams, and (3) dynamic analysis of the test case execution at the model level to determine the coverage for each test case.

We evaluated MaRTS on four applications, and compared it with two code-based RTS approaches. We also evaluated the fault detection ability of the reduced test sets achieved by MaRTS.

II. Approach

MaRTS classifies the test cases as obsolete, retestable or reusable. Obsolete test cases are invalid and cannot be executed on the modified version of the software system. Retestable test cases exercise the modified parts of the software system, and need to be selected for regression testing. Reusable test cases only exercise unmodified parts of the system, and they do not need to be re-executed for safe regression testing [9]. A safe RTS technique must select all *modification-traversing* test cases for regression testing [10]. 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 [5].

In a prior work [6], we applied MaRTS within the context of a *Fine Grained Adaptation* (FiGA) framework [11], [12] that uses UML diagrams to support unanticipated and fine-grained adaptations on running Java software systems. FiGA uses ReverseЯ [13] to extract UML class and activity diagrams from Java source code, and JavAdaptor [14], [15] to update a running Java program without stopping it. In FiGA, each individual method is represented as an activity diagram. The UML activity diagram elements that are supported are initial and final nodes, action nodes, call behavior nodes, and decision and merge nodes. An activity diagram generated using ReverseЯ is executable, where each action node in the activity diagram has a code snippet associated with it, and Java statements are contained inside the code snippet. When the model execution flow reaches an action node, then the code snippet associated with the action node is executed. Additionally, ReverseЯ maps a code-level method invocation statement to a call to the correspoding activity diagram. When the model execution flow reaches such a call, the called activity diagram is executed [6], [16].

In MaRTS, each method of the software system is represented as a UML activity diagram. The same thing applies to each test case. These activity diagrams are executable. We exploit the *Rational software architect* (RSA) simulation toolkit 9.0[1] to execute test cases at the model level.

MaRTS consists of the following five steps:

1) Extract operations-table from the original class diagram.
2) Calculate the traceability matrix.
3) Identify model changes.
4) Extract operations-table from the adapted class diagram.
5) Classify test cases.

MaRTS can scale up to large programs because all of its steps are automated. MaRTS requires the UML models used with it to be detailed and executable in order to obtain the coverage of test cases at the model level. Therefore, MaRTS is not applicable to model-driven development approaches that use models at a high level of abstraction and lack traceability links between the code-level test cases and the models representing the software system.

A. Extraction of the Operations-Table from the Original Class Diagram

This step is performed before developers adapt the models. An *operations-table* is extracted from the class diagram. This table stores for each class the operations that are declared and inherited by the class. For each operation, the *operations-table* stores the operation's declaring class, name, formal parameter types, and return type. For each class in the table, the name of its superclass is also stored.

B. Traceability Matrix Calculation

This step is performed before developers adapt the models. The activity diagrams representing the test cases are executed with the activity diagrams representing the program methods in order to obtain the coverage of test cases at the model level.

During model execution, four types of coverage information are collected for each test case: (1) what activity diagrams are executed by the test case, (2) what activity diagrams are directly called by the test case, (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 activity-level and flow-level traceability matrices that relate each test case to the activity diagrams and their flows that were traversed by the test case.

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

C. Model Change Identification

MaRTS uses RSA model comparison to identify the model changes after developers adapt the class and activity diagrams. The class diagram changes that can be identified are addition/deletion/modification of interfaces, classes, class attributes, operations, and generalization and realization relations. The activity diagram changes that can be identified are addition/deletion/modification of nodes, transition flows, code stored in a code snippet associated with an action node, and the boolean expression associated with a transition flow.

D. Extraction of the Operations-Table from the Adapted Class Diagram

When developers adapt the class diagram, the declared and inherited operations in each class might change. Therefore, an *operations-table* is extracted from the adapted class diagram. The information stored in the *operations-tables* that are extracted from the original and adapted class diagrams are used to determine changes to inherited or overridden operations in each class.

E. Test Case Classification

We proposed a classification algorithm that takes the following inputs: (1) the *operations-tables* extracted from the original and adapted class diagrams, (2) the identified model differences, (3) the flow-level and activity-level traceability matrices, (4) the set of UML activity diagrams representing the methods of the software system, and (5) the set of activity diagrams representing the baseline test cases. The algorithm classifies the test cases as obsolete, retestable, or reusable.

Initially, all the test cases are assumed to be reusable. The algorithm compares the *operations-tables* to identify which operations were changed along the inheritance hierarchy. The activity-level traceability matrix is used to determine each test case that is affected by those changes. The following rules are applied:

1) If an operation *op* is initially declared or inherited by a class *C*, and is now neither declared nor inherited by *C*, then, find each test case that traverses *op* on a receiver of type *C*. If a found test case directly calls *op* on a receiver of type *C*, then flag the test case as obsolete. Otherwise, flag the test case as retestable.
2) If an operation *op* is
   a) initially inherited by a class *C* from an ancestor class *B*, and is now overridden by *C*, or is inherited by *C* from one of its ancestors other than *B*, or
   b) initially declared by a class *C*, and is now inherited by *C* from one of its ancestors.

   then, flag any reusable test case that traverses *op* on a receiver of type *C* as retestable.

Once the algorithm completes iterating over all entries of the *operations-tables*, the test cases that are still flagged as reusable are classified based on the identified model differences. If such a test case traverses deleted or modified transition flows and/or nodes, then the test case is flagged as retestable.

III. Case Study

The goals of the evaluation were to (1) compare the inclusiveness and precision of MaRTS with that of two code-based RTS approaches that support changes to the inheritance hierarchy, and (2) evaluate the fault detection ability of the retestable test set with that of the original test set. Inclusiveness measures the extent to which a regression test selection technique selects modification-traversing test cases for regression testing, and precision measures the extent to which a regression test selection approach excludes test cases that are non-modification-traversing [10].

We compared MaRTS with DejaVu [2] and ChEOPSJ [17]. DejaVu detects fine-grained changes at the statement level, and ChEOPSJ detects fine-grained changes to method invocations. Both tools support the identification of changes to the inheritance hierarchy, and support RTS for Java software systems. We did not compare MaRTS with the existing model-based RTS approaches because they lack tool support (or tools are unavailable).

A. Subject Programs and their Adaptations

We used four subject programs: (1) graph package of the Java Universal Network/Graph Framework (JUNG)[2], (2) Siena[3], (3) XML-security[4], and (4) chess program, which is a classroom project that only supports the functionality to create a chessboard and move chess pieces. These programs were implemented using Java 6 and 7. They do not use generic types and multithreaded programming. Table I summarizes the data for the original versions of each subject.

TABLE I
Original Programs

Subject	Version	Num. classes	Num. interfaces	Num. methods	LOC
`JUNG`	1.3.0	13	12	146	3655
`Chess`	0	7	1	65	1074
`Siena`	1.8	9	0	95	1605
`XML-security`	2	173	6	1172	16800

We used EvoSuite [18] to generate JUnit test cases for each of these versions. For JUNG, 188 test cases that achieve 81% statement coverage were generated. For Siena, 107 test cases that achieve 89% statement coverage were generated. For chess, 130 test cases that achieve 96% statement coverage were generated. The XML-security package has JUnit test suite that comes with it and achieves 31% statement coverage. The generated test cases for XML-security did not improve the coverage of the existing test suite. Therefore, we excluded the generated test cases for XML-security from this study, and only considered the existing test cases that come with the application.

[2]http://jung.sourceforge.net/download.html
[3]http://sir.unl.edu/portal/bios/siena.php
[4]http://sir.unl.edu/portal/bios/xml-security.php

TABLE II
Adaptations Performed on Models

Subject	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

We extracted class and activity diagrams from the original version of each subject program and its test cases. Then, we adapted the class and activity diagrams from one version to the following version in a systematic way. First, we identified the code-level differences between the two versions. Second, we manually applied these differences at the model level. The changes at the model level involved additions and deletions of classes, interfaces, operations, generalization and realization relations, and modifications to method implementations by modifying the activity diagrams representing these methods. Table II summarizes the changes performed on models.

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

B. Inclusiveness and Precision Results

Table III shows the results of running the three RTS approaches. For example, MaRTS and DejaVu classified all the 188 test cases of JUNG as retestable, and ChEOPSJ classified 178 of out of the 188 test cases as retestable. For the XML-security subject, MaRTS classified 10 out of 94 test cases as obsolete, and classified the remaining 84 test cases as retestable. We found that the 10 obsolete test cases contain calls to deleted operations. DejaVu and ChEOPSJ do not address the identification of obsolete test cases. DejaVu classified all the 94 test cases as retestable. Therefore, we excluded the 10 obsolete test cases from the calculations of the inclusiveness, precision, false positives, and false negatives for the three RTS tools.

We did not get RTS results for ChEOPSJ when we ran it on the XML-security subject because of a bug in ChEOPSJ. It did not detect code changes that it is supposed to detect, and did not produce results. Table III and Table IV do not show results for ChEOPSJ with respect to the XML-security subject.

Table IV shows the number of false positives and false negatives for each of the studied RTS approaches. 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. The same set of test cases that was classified as retestable by DejaVu was also classified as retestable by MaRTS for all the subject programs (excluding

TABLE III
TEST CASE CLASSIFICATION RESULTS

Subject	Evolution	Number of 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

the 10 obsolete test cases for XML-security). Therefore, the inclusiveness of MaRTS was also 100%. ChEOPSJ missed some 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. The reason is that ChEOPSJ only records changes to method invocations, but not to other types of statements in method bodies.

TABLE IV
NUMBER OF FALSE POSITIVES (FP) AND FALSE NEGATIVES (FN)

Subject	Evolution	DejaVu		ChEOPSJ		MaRTS	
		FP	FN	FP	FN	FP	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

The precision was 100% for MaRTS and DejaVu because neither classified any non modification-traversing test case as retestable for each subject program. The precision of ChEOPSJ was 100% for JUNG and Chess, 62% for Siena version 1.12, and 60% for version 1.14. The reason is that ChEOPSJ is based on static analysis of dependencies between modified code and test cases, which leads to classifying non modification-traversing test cases as retestable.

C. Fault Detection Ability Results

The results for MaRTS showed a reduction in the number of selected test cases only for the Siena subject for the adaptation from version 1.8 to 1.12, and from 1.8 to 1.14. We used mutation testing to evaluate the fault detection ability of these reduced test sets. We excluded the XML-security subject from the fault detection ability evaluation because all of its test cases were selected by MaRTS (excluding the 10 test cases that were classified as obsolete by MaRTS).

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. In particular, we used `PIT`[5] to apply first-order method-level mutation operators to the code-level versions 1.12 and 1.14. The applied mutation operators[6] 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 adapted methods. We ran `PIT` with both the original and retestable test sets on both the versions.

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

TABLE V
MUTATION TESTING RESULTS

Subject	Mutants	Full Test Set		Retestable 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%

Table V shows the mutation testing results. Both the original and retestable test sets killed exactly the same set of mutants in both the versions. The fault detection ability of the retestable test set was equal to that of the original test set.

D. 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.

We used EvoSuite to generate JUnit test cases for the subject programs. The results could change if other test generation tools were used or test sets with different coverage numbers were used. Additionally, the test cases generated for the Siena subject achieved low mutation scores. The fault detection ability results could change if other test sets that achieve different mutation scores were used. We plan to evaluate the proposed approach on additional test suites generated by other test case generation tools.

Another 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 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 inclusiveness and precision to evaluate MaRTS. 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.

IV. Related Work

The RTS problem has been studied for over three decades [5], [19]. Most of the existing approaches are code-based [1], [2], [17], [20], [21], [22], and little work exists in the literature on model-based RTS. We summarize the existing model-based RTS approaches and compare them with MaRTS.

Chen et al. [23] use UML activity diagrams to perform specification-based black-box RTS. In their approach, an activity diagram represents the requirements of a system. In contrast, MaRTS uses activity diagrams to represent fine-grained behaviors of a software system. Korel et al. [24] 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 support changes to the inheritance hierarchy because it does not use UML class diagram.

Farooq et al. [7] use UML class and state machine models for RTS. This approach does not support the identification of (1) the addition and deletion of the generalization relations, and (2) the overridden and inherited operations along the inheritance hierarchy.

Briand et al. [4] present an RTS approach based on UML use case models, class models, and sequence models. Zech et al. [8] present a generic model-based RTS platform, which is based on the model versioning tool, *MoVE*. The approach consists of the three phases that are controlled by OCL queries, namely, change identification, impact analysis, and test case selection. The approaches of Briand et al. and Zech et al. can identify the addition and deletion of generalization relations between classes. However, they do not identify the impact of such changes to the inherited and overridden operations along the inheritance hierarchy, which can result in missing some retestable test cases.

In contrast to the above mentioned model-based RTS approaches, MaRTS can identify changes along the inheritance hierarchy and classify test cases accordingly.

V. Conclusions and Future Work

In this work, we presented a model-based RTS approach that supports fine-grained changes in method implementation and changes to the inheritance hierarchy, and takes into account the impact of such changes on the selection of test cases. MaRTS was evaluated on four subjects and compared with two code-based RTS approaches, DejaVu and ChEOPSJ, which consider changes to the inheritance hierarchy and support Java software. MaRTS outperformed ChEOPSJ and achieved comparable results to DejaVu in terms of inclusiveness and precision. MaRTS was able to identify a certain type of obsolete test cases. DejaVu and ChEOPSJ do not address the identification of obsolete test cases. The retestable test sets obtained by MaRTS achieved the same fault detection ability that was achieved by the full test sets.

We will evaluate the inclusiveness and precision of MaRTS on additional subject programs, and evaluate its efficiency in terms of reducing regression testing time.

Acknowledgment

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

References

[1] G. Rothermel and 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.

[2] 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.

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

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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, Sep. 2013, pp. 271–286.

[12] 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.

[13] W. Cazzola, S. Pini, A. Ghoneim, and G. Saake, "Co-Evolving Application Code and Design Models by Exploiting Meta-Data," in *Proceedings of the 22nd Annual ACM Symposium on Applied Computing (SAC'07)*. Seoul, South Korea: ACM Press, Mar. 2007, pp. 1275–1279.

[14] M. Pukall, A. Grebhahn, R. Schröter, C. Kästner, W. Cazzola, and S. Götz, "JavAdaptor: Unrestricted Dynamic Software Updates for Java," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*. Waikiki, Honolulu, Hawaii: IEEE, on 21st-28th of May 2011, pp. 989–991.

[15] 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.

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

[17] Q. D. Soetens, S. Demeyer, A. Zaidman, and J. Pérez, “Change-Based Test Selection: An Empirical Evaluation,” *Empirical Software Engineering*, pp. 1–43, Nov. 2015.

[18] A. Arcuri, J. Campos, and G. Fraser, “Unit Test Generation During Software Development: EvoSuite Plugins for Maven, IntelliJ and Jenkins,” in *Proceedings of the 9th IEEE International Conference on Software Testing, Verification and Validation (ICST’16)*, L. Briand and S. Khurshid, Eds. Chicago, IL, USA: IEEE, Apr. 2016, pp. 401–408.

[19] E. Engström, P. Runeson, and M. Skoglund, “A Systematic Review on Regression Test Selection Techniques,” *Information and Software Technology*, vol. 52, no. 1, pp. 14–30, Jan. 2010.

[20] L. J. White and K. Abdullah, “A Firewall Approach for Regression Testing of Object-Oriented Software,” in *Proceedings of the 10th International Software Quality Week (QW’97)*, San Francisco, CA, USA, May 1997.

[21] 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.

[22] M. Skoglund and P. Runeson, “Improving Class Firewall Regression Test Selection by Removing the Class Firewall,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 17, no. 3, pp. 359–378, Jun. 2007.

[23] 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.

[24] 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 (ICSM’02)*, G. Antoniol and I. D. Baxter, Eds. Montréal, Quebec, Canada: IEEE, Oct. 2002, pp. 214–223.