Improving Model-Based Regression Test Selection

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

***Abstract*—Existing model-based regression test selection approaches are based on analyzing changes performed at the model level. These approaches have three limitations. First, they cannot detect all types of changes from design models. Second, they do not identify the impact of changes to the inheritance hierarchy of the classes. Third, their applicability is limited due to the abstraction gap between the code-level regression test cases and the models that represent the software system at a high level of abstraction. This paper discusses two model-based RTS approaches to overcome these limitations, the evaluation plan, and the current status.**

***Index Terms*—regression testing, model-based regression test selection, UML activity diagram, UML class diagram, fuzzy logic**

I. Problem

Models can be used to perform evolution and runtime adaptation of a software system [1], [2]. Regression testing of the evolved and adapted models is needed to ensure that previously tested functionality is still correct. Regression testing is one of the most expensive activities performed during the lifecycle of a software system [3], [4]. Regression test selection (RTS) [5] improves regression testing efficiency by selecting a subset of test cases from the original test set for regression testing [5], [6]. Model-based RTS has several advantages over code-based RTS. The effort for testing can be estimated earlier [6], [7], [8], tools for regression testing can be programming language independent [6], [8], and managing traceability can be more practical at the model level [6], [8]. Model-based RTS can scale up better than code-based RTS for large software systems [9].

Existing model-based RTS approaches suffer from the following three problems. First, they cannot detect all types of changes from UML class, sequence, and state machine diagrams that are used in these approaches [6], [7], [8]. Briand et al. [6] provided an example for such a change, which is a modification to an operation implementation that does not affect the operation's signature and contract. Second, they do not support the identification of changes to inherited and overridden operations along the inheritance hierarchy [6], [7], [8]. As a result, existing model-based RTS approaches can miss relevant test cases that need to be reexecuted. The third problem is the lack of traceability links between code-level test cases and the models representing the software system. The reason is that models are generally created at a high level of abstraction and lack low-level details that are needed to obtain the coverage of test cases at the model level. This lack of traceability is a known issue in model-based RTS approaches, and can severely limit the applicability of these approaches [9].

II. Related Work

The RTS problem has been studied for over three decades [10], [9]. Most of the existing approaches are code-based [11], [12], [13], [14], [3], [4], and little work exists in the literature on model-based RTS. We summarizes the limitations of existing model-based RTS approaches. Farooq et al. [7] used 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. [6] presented an RTS approach based on UML use case models, class models, and sequence models. This approach can identify the addition and deletion of generalization relations between classes. Zech et al. [8] presented a generic model-based RTS platform, which is based on the model versioning tool, *MoVE*. Briand et al. and Zech et al. do not support the identification of inherited and overridden operations along the inheritance hierarchy. Korel et al. [15] used control and data dependencies in an extended finite state machine to identify the impact of model changes and perform RTS. This approach does not use UML class model. All of the mentioned approaches use design-time models, and require these models to contain enough information to obtain the coverage of test cases at the model level, which is not always a common practice [6].

III. Proposed Solution

In this work we propose two model-based RTS approaches. The first approach called MaRTS addresses the first two problems discussed in section I. The second approach called FLiRTS uses fuzzy logic to address the third problem discussed in section I.

A. *MaRTS*

MaRTS [16] is a model-based RTS approach that uses (1) a UML design class diagram to represent the static structure of a software system, and (2) UML activity diagrams to represent the fine-grained behaviors of a software system. The class and activity diagrams are reverse engineered from the original version of the software system. Each method of the software system and each test case is represented as an activity diagram.

The activity diagrams used in MaRTS are detailed and executable. MaRTS exploits the *Rational Software Architect* (RSA) simulation toolkit 9.0[1] to execute test cases at the model level. Each action node has an associated code snippet that contains code statements. When the execution flow reaches an action node in a model, the code snippet associated with the action node is executed. Each code-level method invocation statement is represented at the model level as a call to the corresponding activity diagram. When the model execution flow reaches such a call, the associated activity diagram is executed [17], [16]. MaRTS is based on the following steps.

Extract information from the UML class diagrams. An *operations-table* is extracted from the original class diagram. This table stores for each class the operations that are declared and inherited by the class, and the name of its superclass. When developers adapt the class diagram, the declared and inherited operations in each class might change. Therefore, an *operations-table* is also extracted from the adapted class diagram.

Calculate traceability matrix. This step is performed before adapting the models. The test cases are executed at the model level, and coverage information is collected for each test case: (1) what activity diagrams are executed, and (2) what flows in each activity diagram are executed. This information is used to obtain the traceability matrix that relates each test case to the activity diagrams and the flows that were traversed by the test case.

Identify model changes. MaRTS uses RSA model comparison[2] to identify fine-grained model changes after developers adapt the class and activity diagrams. This tool can identify fine-grained changes in the activity diagrams at the level of flows, nodes, and code statements associated with action nodes.

Classify test cases. Our algorithm classifies the test cases as obsolete, retestable, or reusable. Initially, all the test cases are classified as reusable. Next, the algorithm compares the *operations-tables* to identify which operations were changed along the inheritance hierarchy. The traceability matrix is used to determine each test case that is affected by those changes, and the affected test cases are classified as retestable or obsolete. The remaining test cases are classified based on the identified model differences.

B. FLiRTS

FLiRTS is a fuzzy logic-based RTS approach that performs RTS using design models that are at a high level of abstraction. FLiRTS uses a UML design class diagram to represent the static structure of the system and UML activity diagrams to represent the behaviors of the system's methods at a high level of abstraction. In contrast to MaRTS, action nodes in these activity diagrams are not executable, and do not contain code statements. Each test case is modeled as an activity diagram that includes call behavior nodes, each of which directly links to an operation in the class diagram. This level of abstraction prevents obtaining the coverage of test cases at the model level. We propose two solutions for this problem.

Activity diagram-based solution. This solution is based on automatically generating detailed activity diagrams called *refinements* from the abstract activity diagrams [18]. A refinement is an activity diagram that contains more flows and nodes compared to the one that it is refining it. FLiRTS assumes that a UML sequence diagram that represents all the use case scenarios of the system is available, and is used to control the refinement generation process to avoid the generation of completely inconsistent and unrelated activity diagrams. Each activity diagram in the system model has several possible refinements. A *refined system model* is the system model where each activity diagram is replaced by one of its refinements. Several combinations of the refinements are possible, which leads to the creation of several refined system models. A test case may or may not traverse activity diagrams in a refined system model depending on which refinements are used, and their correctness.

Fuzzy logic is used to address this uncertainty. We define two input variables and set their crisp values in terms of (1) the extent to which a test case traverses modified activity diagrams in a refined system model, and (2) the extent to which a test case traverses correct refinements in a refined system model. These values are provided as inputs to the fuzzy logic-based classifier. The final results of the classifier for each test case are the probabilities for *Retestable* and *Reusable* associated with each refined system model. We use the most correct refined system model to obtain the final classification for the test case.

Class diagram-based solution. First, FLiRTS reads the design class diagram and extracts relations between the classifiers. We consider the association, composition, generalization, realization, and usage relations. The usage relation type that we consider is defined as the one from a class/interface C to a class/interface D, where C has an operation with a return type and/or a parameter type of D.

Second, FLiRTS uses the extracted relations to build a *relations graph*. In this graph, the classifiers are nodes, and the extracted relations are directed edges between the nodes. Third, FLiRTS uses the activity diagrams representing the test cases along with the paths between the classifiers in the *relations graph* to estimate the coverage of each test case. Based on the *relations graph*, there can be multiple paths that start from a test case and end in adapted/evolved classes. However, we are uncertain about which one of these paths is traversed by the test case. We use fuzzy logic to address this uncertainty for each test case by considering the number of such paths, their length, and types of their edges.

IV. Plan for Evaluation and Validation

We plan to empirically evaluate MaRTS and FLiRTS using subjects as follows:

1) Compare the inclusiveness and precision of MaRTS and FLiRTS with that of two code-based RTS approaches.

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

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

2) Evaluate the fault detection ability of the reduced test sets.
3) Identify generalizable thresholds for the fuzzy sets/rules used in FLiRTS.

V. Expected Contributions

This work will contribute to the modeling field by showing that the proposed RTS techniques are feasible and produce results that are comparable to that of code-based RTS. We expect that MaRTS will improve the safety and precision of model-based RTS, and FLiRTS will improve the applicability, safety, and precision of model-based RTS when applied with models that are at a high level of abstraction and lack traceability with the test cases.

MaRTS and FliRTS can be used within the contexts of model-based evolution and runtime adaptation. For example, the approaches that use models to perform runtime adaptation [19], [20], [21], [22] can utilize MaRTS and FLiRTS to classify regression test cases based on model-level changes.

VI. Current Status

We implemented a prototype tool that automates the process of MaRTS. We compared MaRTS with two code-based RTS approaches, DejaVu [4] and ChEOPSJ [14], in terms of their inclusiveness, precision, the fault detection ability of the retestable test set. We used four subject programs: JUNG, Siena, XML-security, and chess, which is a classroom project.

We extracted class and activity diagrams from the original version of each subject program and its test cases. We adapted the diagrams from one version to the following version. Then, we applied MaRTS, DejaVu, and ChEOPSJ to classify the test cases. We used `PIT`[3] to apply method-level mutation operators to the adapted versions at the code level. We ran `PIT` with both the original and retestable test sets on each version, and reported the killed mutants by each test set. The inclusiveness of DejaVu and MaRTS was 100% for all the programs. The inclusiveness of ChEOPSJ was 94% for JUNG, 96% for Chess, and 92% for Siena. The precision was 100% for MaRTS and DejaVu. The precision of ChEOPSJ was 100% for JUNG and Chess, and 62% for Siena. ChEOPSJ did not produce results for XML-security because of a bug in this tool. The retestable test sets obtained by MaRTS achieved the same fault detection ability that was achieved by the full test sets.

We conducted a preliminary evaluation of FLiRTS. We obtained comparable results on inclusiveness and precision with DejaVu and MaRTS [18].

VII. Proposed Timeline for the Remaining Tasks

August - October 2017:

- Collect subjects for the experiments of FLiRTS.
- Complete the implementation of FLiRTS.
- Perform the experiments using FLiRTS and DejaVu.

October - December 2017:

- Analyze and report the results of the experiments.
- Write the dissertation.

[3]http://pitest.org

Acknowledgment

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

References

[1] W. J. Dzidek, E. Arisholm, and L. C. Briand, "A realistic empirical evaluation of the costs and benefits of uml in software maintenance," *IEEE Transactions on software engineering*, vol. 34, no. 3, pp. 407–432, 2008.

[2] G. S. Blair, N. Bencomo, and R. B. France, "Models@run.time," *IEEE Computer*, vol. 42, no. 10, pp. 22–27, Oct. 2009.

[3] 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.

[4] 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.

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

[6] 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.

[7] Q. 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 *17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, ECBS 2010, Oxford, England, UK, 22-26 March 2010*, 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] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] M. Al-refai, W. Cazzola, and S. Ghosh, “A Fuzzy Logic Based Approach for Model-based Regression Test Selection,” in *Proceedings of the ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MoDELS’17)*, Austin, TX, USA, 17th of Sep.-22th of Sep. 2017.

[19] 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.

[20] 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.

[21] 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.

[22] 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. Cheng, R. B. France, and U. Aßmann, Eds. Springer, Aug. 2014, pp. 237–258.