Supporting Concern-Based Regression Testing and Prioritization in a Model-Driven Environment

Roberto S. Silva Filho, Christof J. Budnik, William M. Hasling, Monica McKenna, Rajesh Subramanyan
Siemens Corporate Research
Software Engineering Department
755 College Road East
Princeton, NJ USA 08540
{Roberto.Silva-Filho.ext, Christof.Budnik, Bill.Hasling, Monica.McKenna, Rajesh.Subramanyan}@siemens.com

***Abstract*—Traditional regression testing and prioritization approaches are bottom-up (or white-box). They rely on the analysis of the impact of changes in source code artifacts, identifying corresponding parts of software to retest. While effective in minimizing the amount of testing required to validate code changes, they do not leverage on specification-level design and requirements concerns that motivated these changes. Model-based testing approaches support a top-down (or black box) testing approach, where design and requirements models are used in support of test generation. They augment code-based approaches with the ability to test from a higher-level design and requirements perspective. In this paper, we present a model-based regression testing and prioritization approach that efficiently selects test cases for regression testing based on different concerns. It relies on traceability links between models, test cases and code artifacts, together with user-defined properties associated to model elements. In particular we describe how to support concern-based regression testing and prioritization using TDE/UML, an extensible model-based testing environment.**

Keywords- model-driven testing; test development; regression testing; test prioritization.

I. INTRODUCTION

In current incremental and interactive development processes [1], tests occur in every stage of software development process. Every time a program is changed due to the addition of new features or fixing of existing issues, tests are run to ensure the quality of the change, and that other features, not directly related to the change, are still working as required. The goal is to find, as early as possible, any defects introduced in the software due to either corrective or evolutive maintenance activities.

In large software projects, tests account for a great amount of effort with respect to both their development and execution. Software quality assurance is many times supported by exhaustive software testing especially before major releases.

The goal of regression testing is to minimize the amount of test cases that need to be executed when a software change occurs, without jeopardizing the detection of faults that may have been introduced. The main idea is to prevent the execution of tests that exercise parts of the code that are not affected by the software change, thus significantly reducing the testing effort needed to validate new software versions, improving the overall productivity of the software development process.

Two important techniques: change impact analysis and prioritization are usually combined in the production of optimal regression test suites [2]. Change impact analysis approaches apply different strategies in the selection of test cases to validate the software after an evolution step. They strive to balance attributes such as inclusiveness, precision, efficiency and generality, while minimizing the number of tests to execute [3]. Likewise, prioritization strategies' goal is to reorder tests based on different criteria such as their fault revealing potential [4]. This information can be used to schedule test execution in order to more effectively reveal faults in the program.

Code-based (or white-box) prioritization approaches have focused on ranking tests based on their ability to reveal errors in the code. A common approach has been to apply code-level metrics based on test coverage, as criteria for prioritization [2], [5]. While very effective in selecting a sub-set of tests that cover specific code changes, these approaches are agnostic to requirements, organizational and architectural-level concerns such as: features, non-functional requirements, risks, and client-base priorities, to cite a few.

Recently specification-based (black-box) approaches as model-driven engineering (or MDE) [6] have been applied in the development and testing of complex software systems. MDE facilitate software development by focusing on the use of models rather than source code as its primary artifact. By relying on abstractions that are closer to the problem domain requirements, MDE helps to bridge the gap between problem and software implementation domains. MDE achieves this goal through the automation of the process of transforming high-level software models into lower-level artifacts, including tests and reports.

In this context, model-based testing approaches, e.g., [7], [8], have been developed to simplify the process of test development and execution. In these approaches, models are used to describe the system's expected behavior, while tools automate the process of test generation and execution. Models have also been applied in the process of test prioritization [9] and regression testing [10]. In particular, model-based integrated development environments such as

TDE/UML [8] provide an extensible platform where these approaches can be implemented.

In this paper, we discuss our approach to model-based regression testing and selection. Instead of relying on the analysis of structural model changes alone, our approach incorporates different user-defined concerns in the process of selection and prioritization of test cases. In particular, user-defined concerns, such as last change date, requirements, risk, and features, are represented as properties in the model. Moreover, through traceability links between requirements, model, test cases and code artifacts, these concerns are used to automatically select and prioritize test procedures, before they are used for code generation and execution. Finally, our approach relies on the online monitoring of changes in the model, identifying model changes without the need for model differencing that uses a lot of computational power. We illustrate our approach by showing how TDE/UML [8], a model-based testing environment for UML, is extended to support concern-based regression testing and prioritization. The approach demonstrates how the combined use of: traceability links, test-driven environments, incremental change tracking, and extensible architectures can be applied in support of regression test generation.

This paper is organized as follows: Section II introduces the model-based testing strategy supported by TDE/UML. This approach is extended by concern-based regression and prioritization strategy as described in Section III. Related work is discussed in Section IV followed by the conclusion and discussion of future work in Section V.

II. MODEL-BASED TESTING WITH TDE/UML

TDE/UML [8] is a tool suite for model-based test generation based on UML. Its overall goal is to generate functional test cases based on use cases, represented as activity diagrams. TDE/UML was developed at SIEMENS Corporate Research (SCR) to automate and formalize the testing process as much as possible, delivering a more systematic and efficient system and integration level tests. The tester annotates the diagrams with additional test data such as coverage requirements, constraints, and preconditions. TDE/UML provides an integrated environment supporting model creating and verification, test suite and test code generation.

A. TDE/UML Characteristics

A distinctive characteristic of TDE/UML is the use of category partition method [11] for input data generation. This approach reduces the space of test cases, without jeopardizing its generality, while keeping the traceability between UML models and test cases. In TDE/UML, the category partition method is integrated with UML diagrams through the use of annotations specified in a language similar to OCL (Object Constraint Language). These annotations define constraints connecting the model description to user defined data categories and choices, prescribed in the category partition method. An additional advantage of TDE/UML is its computational power and efficiency in generating test cases. TDE/UML also provides a plug-in oriented architecture, supporting different extensions, including our concern-based regression testing and prioritization approach.

The benefit is an earliest possible testing in the software lifecycle, which reduces test cycles and improves product quality. TDE/UML has been used within SIEMENS on numerous projects from different domains.

B. TDE/UML Model-Based Approach

The TDE/UML model-based testing approach is summarized in Figure 1. TDE/UML supports both the creation of UML models, and the generation of tests and reports based on these models. TDE/UML is also highly customizable, supporting plug-ins in different parts of the test design and generation pipeline. The main components of the system are summarized as follows.

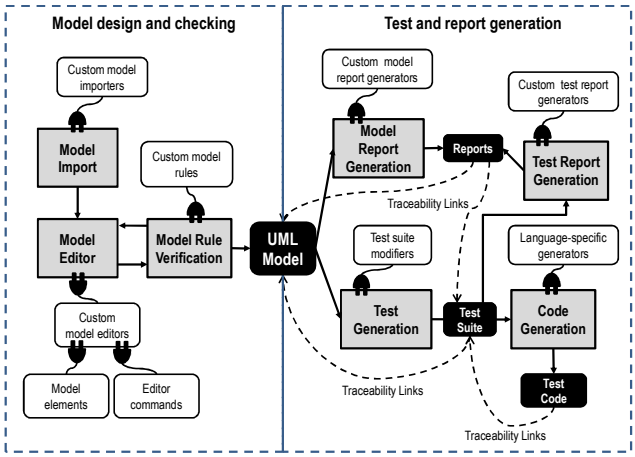

Figure 1. TDE/UML Model-Based Architecture Overview

UML Model: TDE/UML currently supports UML activity and sequence diagrams, as well as class diagrams representing choices and categories in the category partition method.

Model Importer: The use of UML diagrams allows TDE/UML to interoperate with existing modeling tools. This integration is implemented by custom model importers.

Model Editor: Different UML diagrams can be supported. For such, custom model editors, supporting specific UML diagrams and their respective editing commands (e.g. create activity, create note, add guard, etc.) can be defined.

Model Rule Verification: During its development, models can be checked for different consistency and style rules. In particular, TDE/UML supports syntax and semantic checking of OCL data and control constraints, defined within notes in the model, that specify data-driven guards and data input constraints for activity diagram elements.

Model Report Generation: Supports of the process of exporting UML models to different formats, and the generation of model documents. For example: HTML and word processing documents reports, or formats compatible to other UML tools.

Test Generation: During the test generation, the annotated UML model is used to produce a *Test Suite.* This

process is configurable and supports different data and path modifiers, which implement coverage algorithms, including the *happy path* (user-defined critical path), data coverage, path coverage, path-data coverage and others.

Test Suite: Is a data structure representing a set of test procedures derived from UML models. Test procedures are the basic product of test generation. They describe a set of test steps, operating over specific data bindings, as well as generic template code to be used in code generation.

Traceability Links: are defined between individual test steps, artifacts and the model. Optionally, traceability links from generated code to their originating test steps are also generated. These links help in the process of regression testing as described in III.E.

Code Generation: The code generation is based on the test procedures described in *Test Suites*, and on the traceability links to the model. Based on that information, generators (each specific to a programming language) are used to produce executable test procedures.

Report Generation: *Test Suites* can also be used as a basis for generating more detailed test reports, for example, summarizing coverage information.

Test Suite Modifiers: can be defined to further refine the generated test procedures and their steps. Modifiers are also used to filter and reorganize the generated test procedures within a *Test Suite*.

In our approach, the Test Suite Modifiers are used to prioritize test procedures based on different concerns, and to filter out test procedures that were not affected by changes in the model. In the next section, we describe our model-based regression testing and prioritization approach.

III. CONCERN-BASED REGRESSION TESTING AND PRIORITIZATION

Concern-based regression testing and prioritization supports users in selecting and reordering sub-sets of test cases based on different criteria. These criteria include not only changes in the model but also specification-level concerns. It is divided in a set of successive steps illustrated in Figure 2, and summarized as follows:

1. During edit time, the model is monitored for changes as the users modify, add and remove existing elements in the UML diagrams. The model is also annotated with different concerns, represented as element properties.
2. During test generation, and using the timestamps collected during edit time, both structural and semantic changes in the model are identified. This information is used to classify test procedures as re-testable (either new or impacted by changes) and reusable (not affected by changes). Obsolete tests are NOT identified during code generation, but are shown in step 2 of Figure 2 for illustration purposes.
3. During the filtering step, procedures are selected according to different attributes. For example, re-testable procedures are selected for generation based on timestamps of model elements that originate that procedure.
4. During prioritization, tests previously selected for regression testing are reordered based on different attributes such as: risk, change impact, and other user-defined properties associated to model elements.
5. Finally, code is generated and executed. Obsolete tests are deleted, and reusable code is optionally executed.

The key insight of our approach is the use of user-defined properties to represent design and requirements concerns, the monitoring of changes as the test model is edited which produces timestamps, the change impact analysis based on these timestamps, and the use of traceability links between different artifacts generated by the model-based environment. These links are kept consistent as the model is successively transformed from high-level elements into intermediate test procedures, and ultimately into code and report artifacts. By tapping into this process, we can efficiently streamline the regression testing and prioritization process in an efficient way, and can possibly apply this strategy to existing MDE tools.

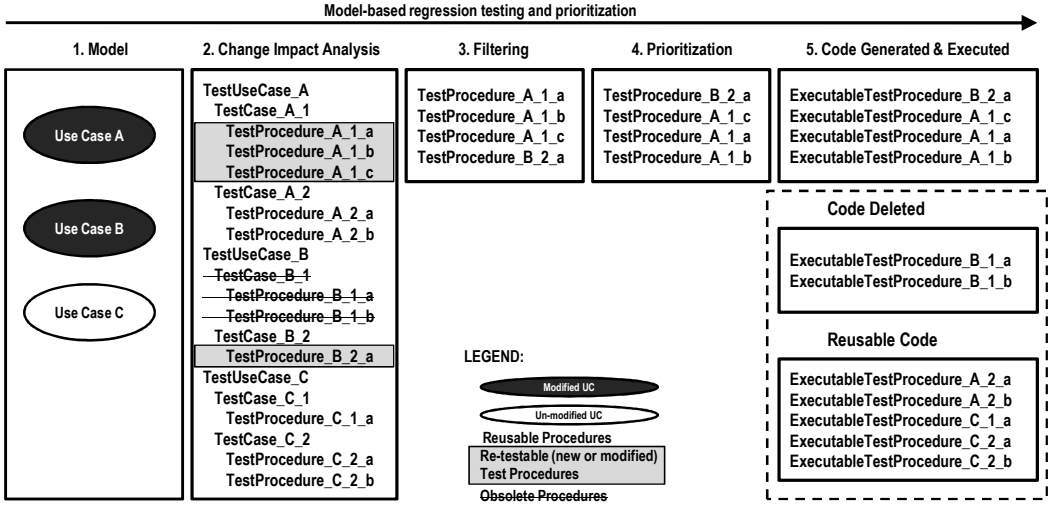

Figure 2. General example showing the main steps of the approach.

In order to validate our approach, we extended TDE/UML as illustrated in Figure 4. In the following sections, we further describe these extensions.

A. *Integrating Concerns in TDE/UML Model Editor*

For every UML element in the model, properties can be defined to represent different concerns such as risk, features, requirements, ownership, and so on. A model element can have different properties, allowing these concerns to overlap in different ways. Once defined by users, properties can be modified and viewed, at model edit time, by clicking on the elements of the model, and using the Properties panel, as shown in Figure 3.

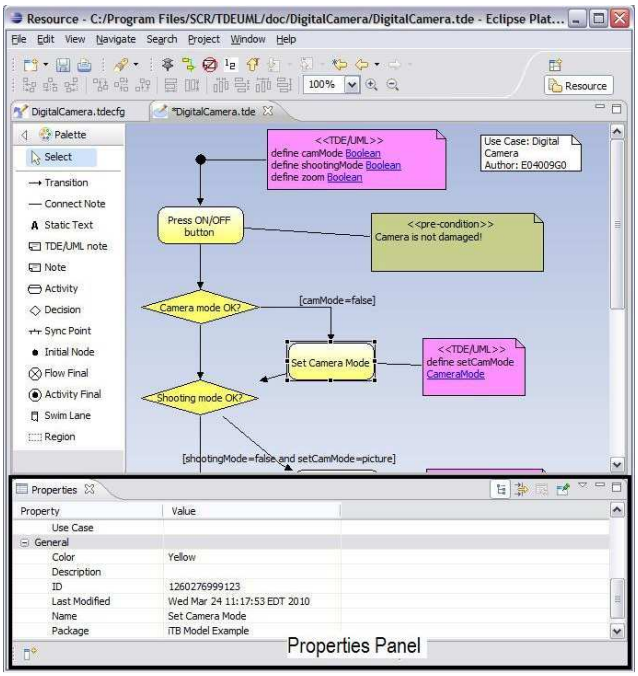

Figure 3. Setting up and Viewing Properties in TDE/UML

B. *Tracking Changes in the Model*

Timestamps are regular element properties and can be inspected at edit time. For example, Figure 3 shows the "Last Modified" property of the "Set Camera Mode" activity. This property represents the last time this activity was modified. Internally, timestamps are strings representing year (YY), month (MM), day (DD), hour (HH), minute (MM) and second (SS) according to GMT time zone. It is stored in the format: YYYYMMDDHHMMSS. Our approach assumes the local computer clock is regularly updated through an SNTP server (this feature is standard in modern operating systems such as UNIX/Linux and Windows). In particular, we adopted the following time stamping criteria for tracking changes in the model:

Semantic Element Updates: Updates in existing diagram elements include modifications of: activity names, decision nodes expressions, note expressions marked with the <<TDE/UML>> stereotype, category names and choices, as well as decision nodes and transition guards. We also consider changes in any user-defined properties. These modifications are all considered semantic changes, and result in the update of their respective elements timestamps.

Structural changes: Upon creation, new activity and decision nodes, sync points, transitions, as well as initial and final nodes all have their timestamps updated. The removal of single nodes in activity diagrams usually result in the deletion of two transitions, and the creation of a new transition between adjacent nodes. This new transition is tagged as changed, as well as the adjacent nodes that it connects. The special cases of deletion of initial or final nodes in a diagram results in the deletion of a transition, and the time stamping of predecessor or successor nodes in the activity diagram. For example, the deletion of a note results in the update of its associated diagram element. Non-semantic changes as the laying out of activities and decision nodes in the diagram are not considered.

Diagram updates. Whenever elements are removed or added to an activity diagram, the diagram itself has its timestamp updated. This approach captures changes such as the deletion of whole sub-diagrams or individual transitions, that otherwise would be undetected by our time stamping approach.

The change tracking feature was implemented by modifying the UML model elements to support timestamp properties, and by modifying existing commands in the activity diagram editor to record changes as the model is modified.

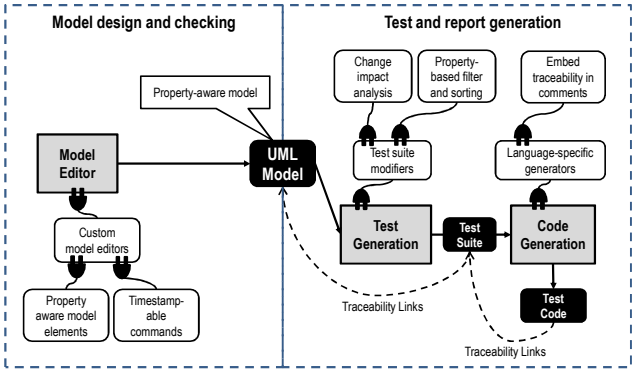

Figure 4. Extending TDE/UML for Regression Testing and Concern-Based Prioritization

C. *Change Impact Analysis*

The change impact analysis used in TDE/UML relies on the traceability links from generated test procedures and the model, and the change tracking approach previously discussed. Using these links, the model elements can be inspected for their respective properties and timestamps.

Before generating a test suite based on the model, developers are asked to define different parameters as shown in Figure 5 for example, the data and path coverage algorithms. Optionally, they can also specify a time frame (time range start, time range end) within which changes in the model are considered for regression testing and/or prioritization.

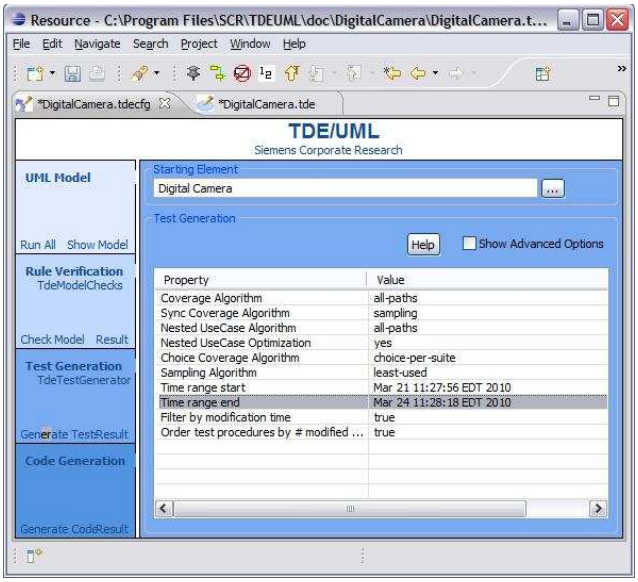

Figure 5. TDE/UML Test Generator UI

As previously described, the regression test procedure happens in three steps (steps 2, 3 and 4 of Figure 2). First, a full *Test Suite* is generated, according to the selected coverage algorithm parameters. In a second step, the resulting *Test Suite* is filtered. Test procedures that have any step whose traceability link points to a modified element within the provided time interval, are selected. Test procedures not originated from modified model elements are discarded for the time being. Third, the test procedures are prioritized according to user-defined criteria. The result is then presented to the end user as shown in Figure 6.

D. Concern-based Prioritization

Prioritization consists on selecting and reordering test procedures based on a priority function. This function is based on the values of one or more model or test procedure properties. For example, a prioritization approach can be defined to reorder test procedures based on the number of re-testable steps they have. Another prioritization schema may consider the average risk of all the steps in each test procedure. Prioritization can also be performed independently from change impact analysis, and may involve different properties at a time.

Test prioritization is implemented by test suite modifiers, installed in the test generation pipeline (see Figure 4). These modifiers reorganize test procedures according to different criteria. For example, test procedures can be sorted based on the number of steps originated in modified elements in the model. This heuristic allows test procedures that cover the highest number of changed model elements (and therefore may have the highest fault reveling potential), to be executed first.

We also support prioritization by other properties. For example, risk. Users can define individual risks for each activity, or may program the system to calculate these risks. Test procedures with steps originated on activities with high risks are ranked higher than those testing lower risk activities.

Hence, the approach allows the combination of different prioritization and regression testing approaches, generating different test suites. The key to this feature is the support for test suite modifiers in TDE/UML, and the ability of the UI in supporting the customization of these policies.

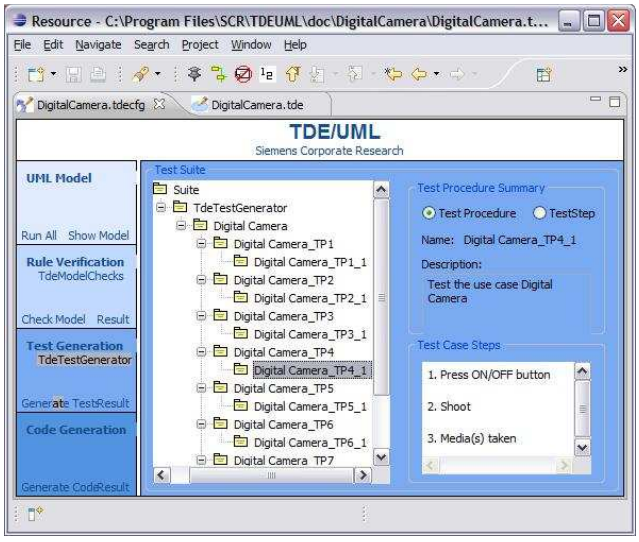

Figure 6. TDE/UML Test Suite Browser

E. Code Generation

After classifying, selecting and prioritizing test procedures, executable tests (code) and reports can be generated. In particular, we assume two different scenarios supporting the generation of executable tests. In the first scenario, only the filtered and prioritized test procedures are used. This allows the fast generation of executable tests for new and modified features, introduced within a time period. This strategy can also be used for generation of executable tests for specific concerns. In a second scenario, during a major software release, for example, a more complete test generation is performed. In that case, re-testable and reusable tests are both generated and executed. Test procedures are generated into individual executable test files as shown in the example of Figure 2.

In both scenarios, obsolete tests are identified by comparing the reusable and re-testable executable tests with the existing executable test code base. This process is automated by the use of test signatures, stored as comments in each executable test source file.

Test signatures are strings derived by composing the full test procedure path. They combine the test use case, test case, test procedure and individual test steps names, including their corresponding data bindings. For example, the signature of *ExecutableTestProcedureA_2_a* of Figure 2 will be the string:

```
TestUseCaseA/TestCaseA_2/TestProcedureA_2_
a/Step1[data1],Step2[data2],…,StepN[dataN]
```

This signature name uniquely identifies executable test cases. By comparing these signatures against generated test

procedures, obsolete tests can be efficiently identified and removed from the code base.

IV. Related work

In both industry and the research literature, there is an increasing interest in model-based regression testing and prioritization. This section discusses current work in the area, comparing them to our approach.

An analysis of existing code-based regression testing and prioritization approaches is presented at [13] and [14]. In all these approaches, code is the main artifact being analyzed. Code-based regression testing is time consuming. It usually requires testers to access and understand the code, or when automation is used, requires the parsing of the whole program code base. An approach for regression test selection where requirements are represented as comments in the code is proposed by [15]. This approach, however, lacks adequate automation to manage requirements changes.

Different model-based prioritization approaches have been proposed in the literature [9], [4] including risk-based approaches such as [16] and [17]. Our work builds upon existing approaches by supporting the combination of prioritization and regression testing based on different user-defined concerns.

Recent developments in model-based regression testing include: model-based test prioritization heuristics [10],[7] that focus on model-based change impact analysis, and the use of traceability information [12] in support of automatic test generation based on UML sequence diagrams. In particular, the work of [12] and [7] perform change impact analysis based on the differentiating of model diagrams. This approach is very costly and time consuming since it requires the compilation of two or more models in a single step. A big advantage of our approach is the minimization of these costs through the tracking of model changes at edit-time, recording change timestamps, as the model evolves, and the ability to combine specification-based concerns with model changes.

V. Conclusions and future work

In this paper, we described an approach for model-based regression testing and prioritization that leverages user-defined properties and traceability links. We discussed our approach showing its integration with TDE/UML.

Currently, we support change-based regression testing based on timestamps, and property-based prioritization. The implementation, however, is batch-based. I.e. the prioritization and filtering is performed as part of the test generation process, using test suite modifiers as shown in Figure 4. We are currently working on a user interface to better support test developers in defining and analyzing alterative prioritization and regression testing scenarios before generating code.

Future work includes the refinement of the change impact algorithms in order to minimize the set of test procedures to be regenerated. The current change impact analysis algorithm employs a best-effort strategy that while guarantees coverage of all changes, is not optimal. We also plan on optimizing the use of traceability links in code generation. Finally, we plan on validating our approach by applying it in different business units at SIEMENS, thus refining our design to meet individual project needs, and by comparing it with existing approaches.

References

[1] C. Larman and V. R. Basili, "Iterative and Incremental Development: A Brief History," in *IEEE Computer*. vol. 36, 2003, pp. 47-56.

[2] G. Rothermel, R. H. Untch, C. Chengyun, and M. J. Harrold, "Prioritizing Test Cases for Regression Testing," in *IEEE TSE*, vol. 27, pp. 929-948, 2001.

[3] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *IEEE Transactions on Software Engineering*, vol. 22, pp. 529-551, 1996.

[4] A. Srivastava and J. Thiagarajan, "Effectively Prioritizing Tests in Development Environment," in *Intl. Symposium on Software Testing and Analysis* Roma, Italy: 2002.

[5] S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky, "Selecting a Cost-Effective Test Case Prioritization Technique " *Software Quality Journal*, vol. 12, pp. 185-210, September 2004.

[6] R. France and B. Rumpe, "Model-driven Development of Complex Software: A Research Roadmap," in *Future of Software Engineering*: IEEE Computer Society, 2007.

[7] L. C. Briand, Y. Labiche, and S. He, "Automating regression test selection based on UML designs," *Inf. Softw. Technol.*, vol. 51, pp. 16-30, 2009.

[8] B. Hasling, H. Goetz, and K. Beetz, "Model Based Testing of System Requirements using UML Use Case Models," in *Intl. Conf. on Software Testing, Verification, and Validation*, 2008.

[9] B. Korel, L. H. Tahat, and M. Harman, "Test Prioritization Using System Models," in *21st IEEE Intl. Conference on Software Maintenance.* 2005.

[10] O. Pilskalns, G. Uyan, and A. Andrews, "Regression Testing UML Designs," in *22nd IEEE International Conference on Software Maintenance*: IEEE Computer Society, 2006.

[11] T. J. Ostrand and M. J. Balcer, "The Category-partition Method for Specifying and Generating Fuctional Tests," *Commun. ACM*, vol. 31, pp. 676-686, 1988.

[12] L. Naslavsky, H. Ziv, and D. J. Richardson, "A Model-based Regression Test Selection Technique," in *IEEE International Conference on Software Maintenance*, 2009, pp. 515-518.

[13] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Prioritizing test cases for regression testing," in *ACM SIGSOFT International Symposium on Software testing and analysis* Portland, Oregon, United States: ACM, 2000.

[14] G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu, "On Test Suite Composition and Cost-effective Regression Testing," *ACM Trans. Software Engineering Methodology*, vol. 13, pp. 277-331, 2004.

[15] P. K. Chittimalli and M. J. Harrold, "Regression test selection on system requirements," in *1st India Software Engineering Conference* Hyderabad, India: ACM, 2008.

[16] R. Subramanyan and C. J. Budnik, "Test Selection Prioritization Strategy," in *33rd IEEE International Computer Software and Applications Conference - Vol 02*. 2009.

[17] Y. Chen, R. L. Probert, and D. P. Sims, "Specification-based Regression Test Selection with Risk Analysis," in *2002 Conference of the Centre for Advanced Studies on Collaborative Research* Toronto, Ontario, Canada: IBM Press, 2002.