Model–based regression testing by OCL

Article *in* International Journal on Software Tools for Technology Transfer · February 2017

DOI: 10.1007/s10009-015-0408-8

CITATIONS
13

READS
29,452

5 authors, including:

Philipp Kalb
University of Innsbruck
8 PUBLICATIONS **54** CITATIONS
SEE PROFILE

Michael Felderer
University of Innsbruck
286 PUBLICATIONS **3,082** CITATIONS
SEE PROFILE

Colin Atkinson
Universität Mannheim
210 PUBLICATIONS **5,332** CITATIONS
SEE PROFILE

Ruth Breu
University of Innsbruck
199 PUBLICATIONS **2,667** CITATIONS
SEE PROFILE

Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Model–based regression testing by OCL

Philipp Zech[1], **Philipp Kalb**[1], **Michael Felderer**[1], **Colin Atkinson**[2], **Ruth Breu**[1]

[1] University of Innsbruck, Innsbruck, Austria
[2] University of Mannheim, Mannheim, Germany

The date of receipt and acceptance will be inserted by the editor

Abstract. Model–based testing has gained widespread acceptance over the last decade, not only in academia but also in industry. Despite its powerful features of abstraction and automation, most existing approaches and tools provide only limited support for regression testing. Yet regression testing, the repeated execution of selected test cases after system modification, is vital, because changes may introduce new bugs or unwanted side–effects that must be avoided at all costs. Model–based testing's potential for supporting regression testing has yet to be explored, even though syntactic and semantic abstractions within software models already allow identifying changes in software systems. This change information can easily be used for test case selection. In this article, we present a model–based regression testing method based on OCL. By means of a running example using the UML Testing Profile we show how our method supports regression testing on the basis of an existing model–based testing method.

Key words: Model–based testing; regression testing; model–based regression testing; model evolution; software testing; software evolution

1 Introduction

Regression testing, i.e., the selective retesting of software modules, attempts to ensure that changes to an existing software system do not alter its existing behavior in an unexpected way. Although this sounds quite simple, however, in recent years, regression testing has become a difficult task. One of the main reasons is the increased complexity of today's software systems. This complexity not only results from the systems themselves, but from their production environments, the daily changes they are subjected to, and also from reliance on third–party components. Consequently, regression testing, has become an expensive, time–consuming and very inefficient task.

In recent years, model–based testing (MBT) has emerged as an efficient alternative to existing code–based testing approaches [1,2]. In applying MBT, software models are used to either describe or derive test cases for a given system under test (SUT). However, existing MBT approaches either completely lack support for regression testing, or at best, only provide a tailored solution for their specific kind of models [1].

To address this lack of support for regression testing, in this article we present a model–based approach for regression testing using the Object Constraint Language (OCL). Based on the Model Versioning and Evolution (MoVE) framework [3], which enables change identification and management within and among models, our approach is capable of processing arbitrary system and test models. Using OCL our approach can be parameterized by different change identification, impact analysis and test selection strategies.

A key result of our method is its *genericity* regarding formal constraints like meta–model conformance. This is due to OCL and MoVE's flexible and generic support for managing arbitrary model resources. As we demonstrate in our running example, our method is applicable to arbitrary models by enabling efficient regression testing.

To avoid any confusion in how we use terms such as *test case*, *model* and the like, in the following we provide concise definitions of the most important terms.

- *Model.* A model is an abstract view of some real–world system. A model thus contains elements that describe some real–world entities of the system, e.g., a process or component.
- *Version.* A version of a model is a *defined* state of evolution of the modeled real–world system. Subse-

quent versions of a model describe the evolution and changes of the real–world system over time.
- *Test Case.* A test case is a behavioral model to evaluate some entity (or entities), such as a process or a component, of the real–world system. Its *coverage* is the set of evaluated entities.
- *Model–based Testing.* Model–based testing (MBT) is the application of models for testing and support for automated generation of testing artifacts like test cases.

Overview of the Proposed Method Our method involves four steps to calculate a new regression test set.

(1) *Delta Calculation.* First, our method calculates the *delta* between two succinct versions A and B of a model for a SUT. The resulting *delta* is the set containing the modified model elements, i.e., $\delta = B \setminus A$.
(2) *Delta Expansion.* Next, our method calculates the *expanded delta* according to a given test case selection strategy S (Section 6.1). The resulting *expanded delta* is the set containing the modified model elements conjuncted with the set of elements S', selected by the strategy S from model B, i.e., $\delta' = \delta \cup \gamma$, where $\gamma = \{e \mid e \in B \text{ and } e \in S'\}$.
(3) *Test Set Generation.* Third, after calculating δ', our method calculates the *new regression test set.* The resulting *regression test set* is the set containing all test cases tc[1] which refer to (i.e., contain) elements from δ', i.e., $\tau = \{tc \mid \pi(tc) \cap \delta' \neq \varnothing\}$, where π is an operator, calculating the coverage of a test case, i.e., which elements of the SUT are covered by test case tc.
(4) *Test Set Reduction.* As a last step, our method reduces the calculated regression test set, if possible. The resulting *reduced regression test set* is the set containing only *original* and *necessary* test cases (i.e., no redundant and only required functionality w.r.t. changed components). Thus, for *original* test cases, $\tau' = \{tc \mid \pi(tc) \cap \Pi(\tau \setminus \{tc\}) = \varnothing\}$, where π is defined as in setp (3), and Π the operator, applying π to all test cases contained in a test set τ (which in turn result in another set, that $\cap$ is applicable). For *necessary* test cases, $\tau' = \{tc \mid \pi(tc) \neq \varnothing\}$.

Results In our article, we show how model–based testing can be used to support regression testing. Through an illustrative example we show how a model versioning engine and OCL can be used to calculate meaningful regression test sets for a given model of a SUT. Further, we show to what extent different regression test selection, test case prioritization, and test set reduction techniques can be applied.

[1] Observe that a test case tc also can be regarded as a set that refers to model elements of the SUT, thus the following definition of τ is sound.

1.1 Contribution

This article extends a previous paper on a generic platform for model-based regression testing [4] and makes several contributions to MBT and regression testing:

(1) *Independent regression testing support for MBT.* As a main contribution, our work delivers a generic approach for model–based regression testing. We call our method *generic* as it is not restricted to any specific meta–models or similar formal constraints, but instead is generic by supporting any model, inasmuch as it provides a meta–model. So far, this special topic has mostly been neglected in the MBT research community. Yet, for MBT to be a complete approach, regression testing is essential.
(2) *Traceability between model artifacts.* Based on the change identification and management capabilities of the MoVE framework (see Section 4), our approach supports full traceability between different models and their various artifacts. This yields efficient fault detection capabilities, assures model consistency and validity and hence, also test suite validity.
(3) *A library of OCL queries.* As our approach makes use of OCL, one of our contributions is an OCL regression testing library. This library contains OCL queries, useful for performing regression testing in an MBT manner. Keeping genericty in mind, the queries are *typeable* for arbitrary kinds of models and their artifacts. The queries cover different change identification, impact analysis, regression test selection strategies and test set reduction strategies.

1.2 Article organization

The remainder of our article is structured as follows, Section 2 provides the necessary background as well as a discussion of related work in the area of model–based regression testing. Section 3 briefly introduces our running example. Following that, Section 4 introduces the MoVE framework. Section 5 contains the main contribution of our article, a discussion of our novel approach for model–based regression testing. Section 7 discusses results from a detailed application of our method on our running example. We conclude in Section 8 with an outlook on future work.

2 Background and related work

In this section we provide relevant background on regression testing techniques (Section 2.1), model repositories (Section 2.2), analysis (Section 2.3), and the UML Testing Profile (Section 2.4). Finally, we give an overview of related work in the area of model–based regression testing (Section 2.5).

2.1 Regression testing techniques

Regression testing is the selective retesting of a system or component to verify that modifications have not caused unintended side effects and that the system or component still complies with its specified requirements [5]. Rerunning every test after each modification is not feasible, thus a trade–off has to be found between the confidence gained from regression testing, and the resources used for it [6]. For this reason, several regression testing techniques summarized in the following paragraphs have been developed.

Regression Test Selection This technique is applied to select only those tests from a regression test set that are affected by changes. Rothermel and Harrold have published a detailed survey paper on regression test selection techniques [7]. In that survey, several techniques have been evaluated (e.g., dependency graphs or symbolic execution) according to their inclusiveness, precision, efficiency, and generality.

Coverage identification This technique is applied to identify those parts of the system that need additional tests due to the change. Changes in the system can introduce new parts, which are not exercised by existing tests. Simple approaches can use code coverage analysis tools [8] to uncover changed portions not touched by existing tests. More advanced approaches typically use some sophisticated data structure (e.g., program dependency graphs [9]) to capture also data and control dependencies in the source code.

Test prioritization This technique is applied to optimize the order of the test execution according to some criteria such as the risks mitigated by specific tests. Test prioritization techniques can have several goals. One can optimize the order of the test set to increase the rate of fault detection, code coverage, or risk mitigation. Elbaum et al. [10] analyzed nine test prioritization techniques (e.g., random ordering or total statement coverage prioritization) with the conclusion that even simple approaches can improve the rate of fault detection. However, the performance overhead of more sophisticated approaches is relatively high.

Test set reduction This technique is applied to optimize a calculated test set by removing redundant test cases, i.e., calculating a subset of test cases with the same coverage as the original test set [11,12,13,14,15]. However, although it results in obvious cost reductions in terms of execution times, and hence, testing costs, test set reduction should be done carefully, as it can severely compromise the fault detection capabilities of a test set [16, 17].

Test set execution This technique is applied to automatically execute the tests in an efficient way. These techniques have found their way into the practice, as most of the current testing tools have this functionality.

2.2 Model repositories

One of the most popular model versioning systems is CDO [18] which is part of the Eclipse Modeling Framework. CDO supports model element versioning, Multi User Access, Transactional Access, Scalability, Thread Safety and Collaboration. CDO does not support instance specifications, state–machines or component diagrams and there is no support for workflows. The most comparable non–commercial system is Unicase [19]. Unicase is a CASE–tool that supports modelling artefacts of a software engineering project, such as components and tasks. Its focus is on software development processes and linkage of artefacts. Its main disadvantages are the fixed metamodel and the limited possibility to integrate client–side tools. IBM Rational Doors [20] is a commercial requirements management tool. Doors manages requirements in a centralized repository. State-based transition systems can model changes and their consequences to related requirements. Within Doors one cannot automate state changes or propagate changes automatically throughout the data.

Another system by IBM is the Jazz–platform [21]. This platform is not a single product but a platform for team–based software development for Rational products such as IBM Rational Team Concert. Its current strengths are the linkage between artefacts using the Linked Lifecycle Data standard.

The third commercial tool is in–Step [22] of microTOOL. In–Step is a tool for process-driven project management in the area of system and software development. All activities defined within the project have a status attribute. Similar to the functionality in Doors, one can trace changes of elements and also ascertain possible impacts on other elements. in–Step allows state machines to be defined to guide the changes to elements. Change propagation and evolving the underlying metamodel is not possible.

In summary, the strengths of MoVE, compared to the other available tools, are the support of a generic method for model versioning, the support of several tool–independent meta–models and the availability of generic change handling mechanism. MoVE also supports state machines for model elements to provide means for workflows and lifecycles.

2.3 Model analysis

Current model analysis frameworks for software or respectively UML models can be subdivided into several groups.

The first group uses the UML profiling mechanism to identify a domain specific set of properties of a model.

Test Architecture	Test Behavior	Test Data	Time
SUT	Test objective	Wildcards	Timer
Test context	Test case	Data pool	Time zone
Test component	Defaults	Data partition	
Test configuration	Verdicts	Data selection	
Test control	Validation action	Coding rules	
Arbiter	Test log		
Scheduler			

Table 1. UTP testing concepts

An example is the MARTE UML profile that adds capabilities to UML to understand Real Time and Embedded Systems properties [23].

More generic Frameworks are the second group of Analysis Frameworks. These Frameworks use query languages to gather information from models. The most important language is the Object Constraint Language (OCL) [24]. specified by the ObjectManagement Group (OMG). OCL is tailored for UML models and the de facto standard to describe system constraints and queries. OCL can also be used to define metrics [25]. EMF–IncQuery [26] is an alternative framework for defining declarative queries which makes use of graph patterns to optimize the performance.

The third group is formed by approaches that are derived from logic programming. Examples are the usage of Prolog [27,28] to query models or Answer Set programming [29,30] to reason on UML Diagrams. Prolog is a declarative programming language based on the logic programming paradigm. In Answer Set Programming, search problems are reduced to computing stable models, and solvers for these stable models are generated, to finally perform searches.

In general there is a couple of approaches to analyse static models. The innovation in our approach is that we incorporate the knowledge about the continuously changing models and also the workflows on model elements to support a more holistic analyses also including an analysis of the model evolution.

2.4 UML Testing Profile

The UML Testing Profile (UTP) [31] provides concepts to develop test specifications and test models for black-box testing. UTP has been standardized by the OMG and mapping rules to the executable test definition languages TTCN-3 [32] and JUnit [33] have been defined. The profile introduces four concept groups for *Test Architecture*, *Test Behavior*, *Test Data* and *Time*. Table 1 gives an overview of the most important concepts of the UTP.

An overview of the meaning of the four concept groups is presented in the following paragraphs.

Test Time This concept is related to time constraints and observations within a test specification. These concepts, in addition to the existing UML 2.0 time concepts, are needed to provide a complete test specification. A timer controls the test execution and reacts to start and stop requests as well as timeout events. The graphical syntax for timer actions is adopted from the Message Sequence Charts used by TTCN-3.

Test Data The data a test is based on is supplied via so-called data pools. These either have the form of data partitions (equivalence classes) or explicit values. The test data is used in the stimuli and observations of a test. A Stimulus represents the test data sent to the SUT in order to assess its reaction.

Test Behavior A test specifies the interaction of the SUT with test components in order to realize the test objective. The test is specified in terms of sequences, alternatives, loops, stimuli, and observations from the SUT. During execution a test verdict is returned to the arbiter. The arbiter assesses the correctness of the SUT and finally sets the verdict of the whole test.

Test Architecture The concepts of the test architecture are related to the structure and the configuration of tests, each consisting of test components and a test context. Test components interact with each other and the SUT to realize the test behavior. The test context encapsulates the SUT and a set of tests as well as an arbiter interface that evaluates and generates the verdict, and a scheduler interface that controls the execution of the test cases. The composite structure of the test context is referred to as the test configuration.

More details on the informal semantics of the concepts can be found in [31]. UTP does not offer the possibility to directly execute the described tests. To execute the tests a transformation or mapping to another language needs to be employed. Such language mappings exist for JUnit and TTCN-3.

2.5 Related work on model–based regression testing

Model-based regression testing (MBRT), which relies on explicit models for regression testing purposes, has several advantages over regression testing on the code level [34]. The effort for testing can be estimated earlier, tools for regression testing can be largely technology independent, the management of traceability and test automation at the model level is more practical, no complex static and dynamic code analysis is required, and models are smaller compared to the size of modifiable elements because they are more abstract. Model–based regression testing approaches determine impacts of system

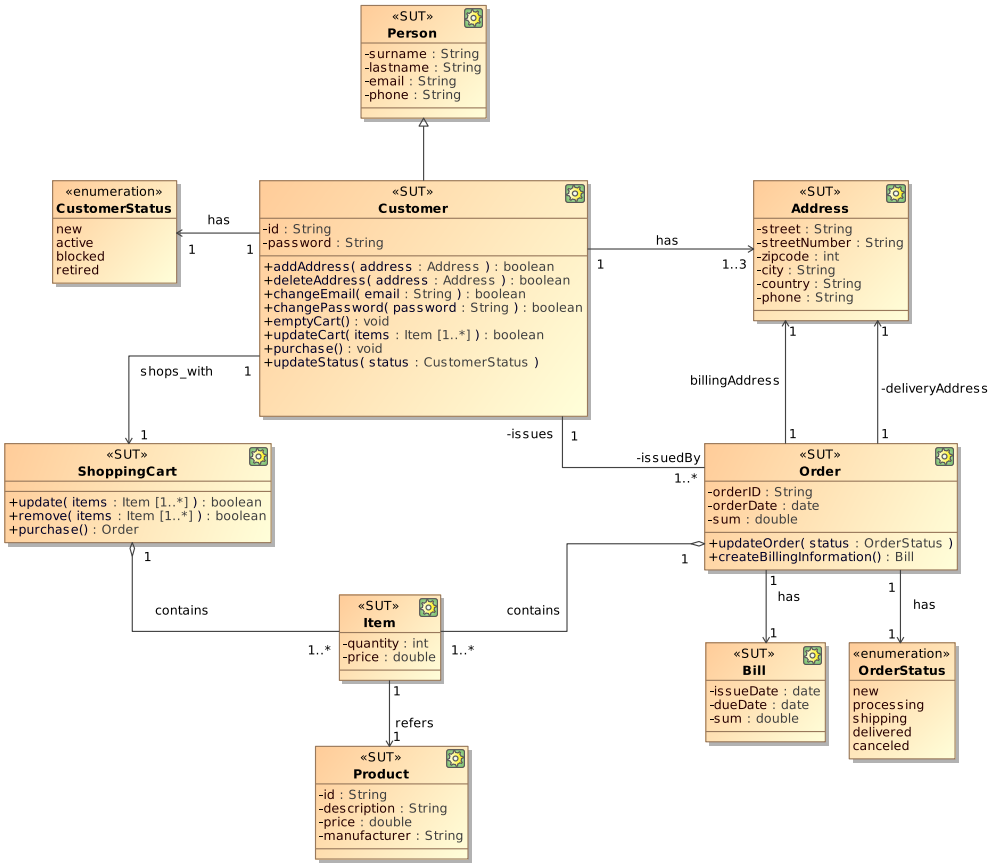

Fig. 1. Class diagram of running online shop example

model changes on the baseline test suite. UML–based system models typically consider class models and a specific type of behavior models, such as state machines [35], sequence diagrams [34] or activity diagrams [36].

In the approach of Chen et al. [37], regression test suites are generated from extended finite state machines. A dependency analysis searches for the effects of changes expressed as elementary modifications, and creates test cases for the changed parts of the system.

The method presented by Korel et al. [38] works similarly on extended finite state machines, and its focus is to reduce an existing regression test suite.

Fraser et al. [39] define an approach to regression testing and test suite updating with model checkers. Windmüller et al. [40] present active continuous quality control (ACQC) which applies incremental active automata learning technology periodically in order to infer evolving behavioral automata of complex applications. ACQC allows to reveal bugs by inspecting difference views of (tailored) models and enables not only to check for designated regressions and bugs but to validate whether the system behaves as expected across versions. Finally, Felderer et al. [41,42] define a test evolution management method. In this approach state machines are attached to all model elements. The evolution process is then initiated by adding, modifying or deleting model elements which trigger change events and fire transitions in state machines. The model is then changed manually or automatically until it is consistent and executable. Based on a test requirement considering the actual state of model elements, a test suite is selected and executed.

3 Running example

In the course of our article we will refer to the running example—a small system realizing an online shop. It allows the main concepts of our method to be presented throughout our article. We use UTP to model relevant test cases.

Figure 1 shows the class diagram for the example. The shop allows registered customers to purchase various products. Each *Customer* is associated with at least one *Address* (or at most three) and a *CustomerStatus* class. For shopping, each customer has a *ShoppingCart* to store selected items. Each *Item* refers to a *Product*. Once finished shopping, a *Customer* may purchase the items in the shopping cart, which results in an *Order*. Finally, an order is associated with an *OrderStatus* indicating its processing. Also, each order is associated with

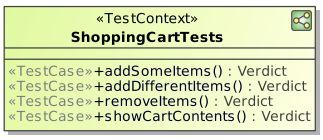

Fig. 2. UTP test cases for the *ShoppingCart* class from the online shop example (see Fig. 1)

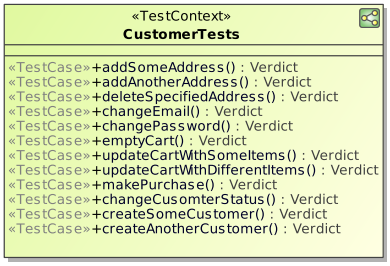

Fig. 3. UTP test cases for the *Customer* class from the online shop example (see Fig. 1)

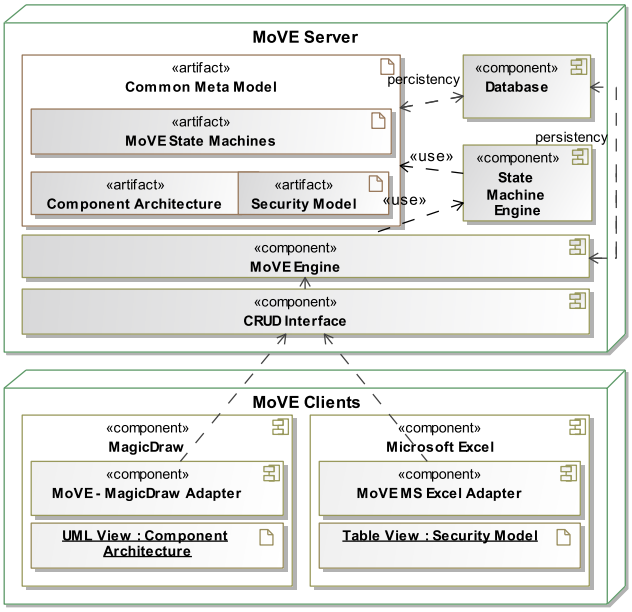

Fig. 5. The MoVE architecture

a *Bill*. For UTP conformance, SUT classes apply the *SUT* stereotype.

Modeling test cases for the *ShoppingCart* class from Fig. 1 results in the class diagram from Fig. 2. It defines a *Test Context ShoppingCartTests*, which contains four *test cases*, *addSomeItems*, *addDifferentItems*, *removeItems*, and *showCartContents*.

Figure 3 shows another test context for the *Customer* class from our online shop example, while Fig. 4 shows the test behavior for the *addSomeAddress* test case. First, the *CustomerTestComponent* creates the *CustomerTests* test context, then invokes the *addSomeAddress* test case, which then invokes *addAddress* on the *Customer* class (after creating it). After *addAddress* returns, the test contexts checks the outcome and returns the result to the *CustomerTestsComponent*. Additionally, we have defined a timing constraint to assure this all happens within 100 milliseconds.

4 MoVE—Model Versioning and Evolution

The MoVE (*Mo*del *V*ersioning and *E*volution) framework is a model repository supporting versioning of arbitrary models together with the provision of a method to support collaborative workflows on data artifacts. MoVE was developed as an infrastructure to manage Living Models [43,44], i.e., it was developed to be a repository that allows to store and receive models providing support for model evolution.

In the course of several case studies with industry partners we identified several major problems that arise with modern IT systems with a focus on modeling:

- *Heterogeneous Environment:* In general different types of models and modeling languages are used within a single project.
- *Linkage of Models:* Usually projects to not consist of isolated models but inter–linked models for example to obtain traceability between artifacts.
- *Models are part of workflows:* Models are a living entity that are constrained and managed with workflows.
- *Models are not static:* Models change during the life of a system. Change–driven engineering becomes important.

The main challenges for MoVE are therefore to allow heterogeneous models, analyze changes on models and to support workflows on models and model elements. In this section we present the MoVE evolution engine as our approach to tackle the described challenges and introduce it also as an infrastructure for our Model–based Regression Testing approach.

4.1 Conceptional architecture of MoVE

Figure 5 shows the core concepts of the MoVE architecture. The MoVE server provides persistence for instances of the *Common Meta Model*. The *Common Meta Model* serves as an integrating data model, yet does not necessarily comprise the union of data from all connected MoVE Clients (requirements engineering tools, enterprise architecture management tools and the like). Rather, the Common Meta Model integrates the data which is necessary for data traceability and stakeholder collaboration across system boundaries. The Common

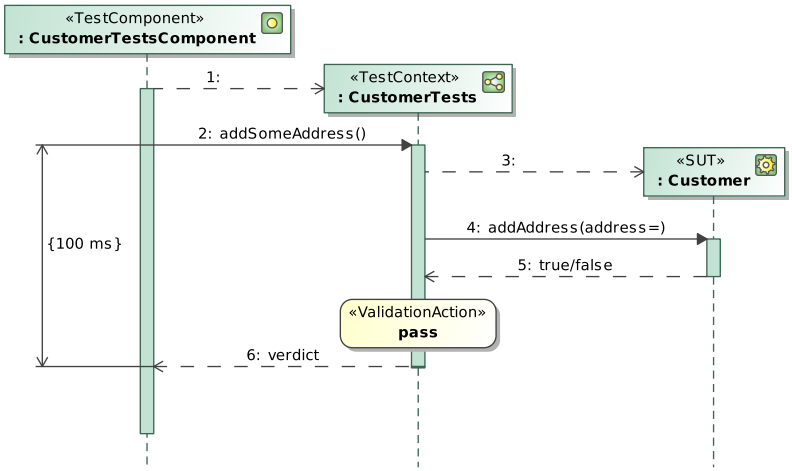

Fig. 4. Test behavior for test case *addSomeAddress*

Meta Model may be structured in a hierarchical way, e.g. defining model layers.

MoVE supports *partial models* which are models that only contain a specific subset of instances of the complete system model. Each partial model is a view of the complete system model concentrating on a specific aspect. In the example of Fig. 5 the first partial model shows the components of the current system (*Component Architecture*) described in a UML diagram. The second partial model represents the security aspects of the system (*Security Model*) which can be described for instance in Excel spreadsheets or tables. The security model might for example contain security requirements for components of the system. The Common Meta Model is used to link the security requirements with the system components.

On the client side, Fig. 5 shows that we provide a sophisticated tool implementation that covers all functionality of MoVE, we integrate adapters into already established products. The main idea of this architecture is to allow stakeholders to use tools which are customized for the stakeholders task and view of the system, instead of providing one generic tool. To achieve this we provide *MoVE Clients* which are connected to the MoVE repository through adapters and integrate into the tools by using the tools plugin interface. The adapters are responsible for supporting the check out of an existing model, the commit of changed models or model elements and support of conflict resolution in case of conflicting commits. In addition, an adapter's main responsibility is to translate a tool's proprietary model representation into the UML representation within MoVE. In order to demonstrate the bandwidth of possible MoVE Clients we currently have implemented an adapter to a modeling tool (noMagic MagicDraw [45]) and to (MS Excel) spreadsheets.

Technical details: MoVE is based on the Eclipse Rich Client Platform (RCP) and uses Eclipse Modeling Framework (EMF) for general modeling tasks. As already explained, MoVE supports model versioning and model element versioning which means that each single instance of a model can be separately versioned in MoVE. This is achieved by using a database as the underlying persistency framework. To store model elements in the database, MoVE converts the XMI/UML representation of models into a java–class representation of the metamodel. Each class defined in the UML representation of the metamodel is converted to a java class by using Eclipse Xpand to generate *class* files. While java classes represent the metamodel, objects of the java classes represent single instances. The java classes are enriched with hibernate annotations which are used to store the java objects in the database. In short, the persistency layer of MoVE transforms metamodels into java classes and uses hibernate to map the java classes of the metamodel to database schemas and instances of the metamodel to java objects that are stored in the database.

4.2 The change–driven process

One of the major capabilities of MoVE is supporting changes. Software and their respective models are under continuous change which must be considered by a model repository because of its high impact on the underlying data. MoVE uses a change–driven process to manage and control changes. In our previous work [3] we have already shown how the use of state machines can be used to help to support workflows on models and model elements. In this section we will explain the components used for the change–driven process without giving further details on our state machine implementation.

Figure 6 shows an activity diagram containing the most important actions and artifacts in a change process which is started on a commit of a model or model element. After an initial commit MoVE always stores

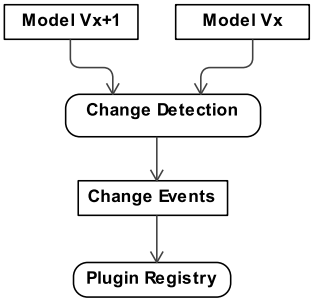

Fig. 6. MoVE change–driven process

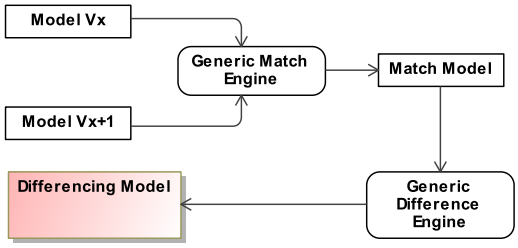

Fig. 7. Delta calculation with EMF Compare

a version of the model V_x which can be checked out and updated by MoVE Clients. The model V_{x+1} depicts an updated version of the same model. The first action executed by the change–driven process is the *change detection* which is a component to calculate the differences between V_x and V_{x+1}, respectively the Δ of both model versions.

The Δ contains information about the model elements that have been added, deleted or updated. If V_{x+1} is a complete model, EMF Compare is used to calculate the Δ. Otherwise, if V_{x+1} only comprises a single model element, an algorithm implemented in Java is applied to compare the model element to its previous version. The algorithm is a simplified version of the EMF Compare tool's algorithm.

Figure 7 shows a conceptual view of the algorithm applied by EMF Compare to calculate the Δ which corresponds to the *Differencing Model* of EMF Compare. In a first phase EMF Compare calculates a *Match Model* which contains a mapping between elements of V_x and V_{x+1}. The mapping is based on a similarity metric that defines how similar two elements are. Alternatively, EMF Compare can use the XMI–IDs to identify similar elements. The *Match Model* is used as input for the *Generic Difference Engine*. The difference calculation results in a *Differencing Model* which is an EMF model that contains a more detailed description of changes of elements. One can find an extensive explanation of the EMF Compare algorithm and models in the EMF Compare Developers Guide [46].

Using the Δ the *change detection*–component derives *change–events*. A change–event contains information about the changed element and the type of change. Since model changes can typically cause a chain of further implicit changes, such as the change of references for elements connected by an association, it is possible to limit the change–event generation by applying OCL rules. Before generating the event, MoVE executes the OCL constraint(s) for the current model element of the Δ and only sends an event if the constraint evaluates to true. Otherwise the event is not distributed and has no further consequences. The OCL constraints are also used by our regression testing methodology as described in Section 5.1.

The most important change–events in MoVE are:

- *MODEL_CHANGE* is a general event that denotes that a certain partial model has changed containing only the name of the model.
- *INSTANCE_ADDED* is generated when a new instance has been added containing the name of the instance and a reference.
- *INSTANCE_DELETED* is generated when an instance has been deleted, containing the same information as INSTANCE_ADDED.
- *INSTANCE_UPDATED* is generated when an instance has been updated.

After an event is created it is forwarded to the plugin registry as shown in Fig. 6. MoVE allows custom–tailored plugins to be created that listen to an event. Such a plugin can be used to implement additional functionality for MoVE. The plugin registry can be configured using a graphical interface, which allows MoVE to configure a different set of plugins for each model. If an event occurs the plugin registry forwards it to the respective plugins. In this article we implemented a plugin that listens to MODEL_CHANGE events.

In summary, we presented the model repository framework MoVE which supports the integration of models from heterogeneous sources and collaborative workflows in model artifacts. Due to its flexible API MoVE can be integrated into various platforms and supports not only XMI based clients but also comes with an API that makes it possible to directly work on objects without the bridge of XMI files. The only limitation is that the client applications provide an API to integrate plugins and access its data.

5 A model-based regression testing approach

In the following, we introduce our new method for generic, model–based regression testing. First, we introduce the core concepts of our method.

We designed the method to support a broad range of MBT approaches using their custom models. We achieve such generic support thanks to MoVE's flexible and generic support for managing arbitrary model resources (see Section 4).

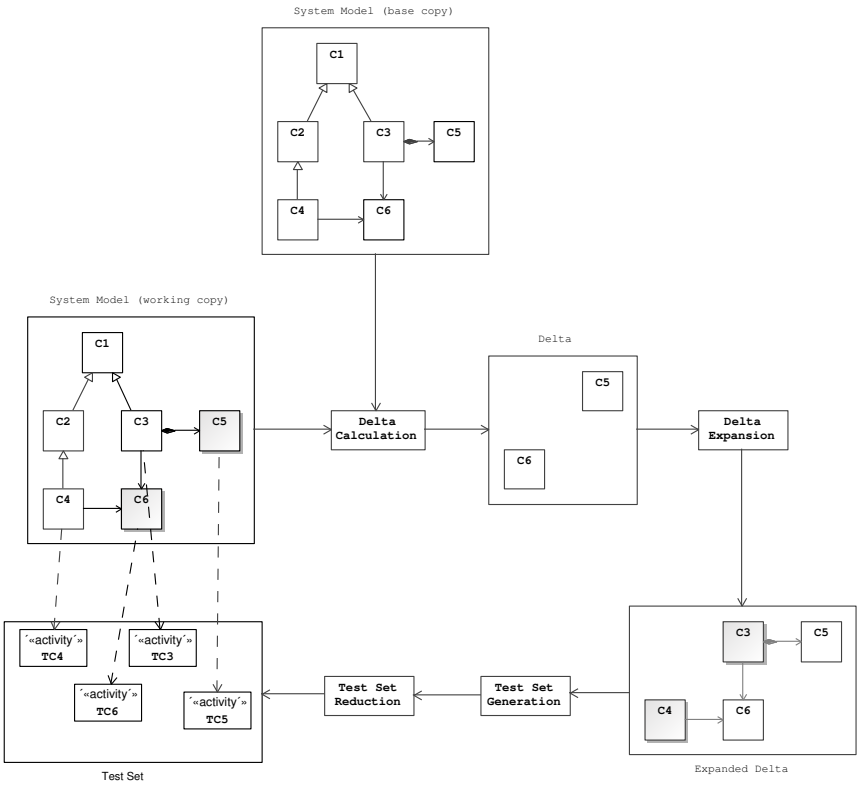

Fig. 8. Model-based regression testing by OCL

As shown in Fig. 8, as a first step, two different versions of the same model are compared, i.e. the *Base Model* (initial development model) and the *Working Model* (current development model) to calculate a *delta* from those two models. The resulting *change set* (delta) then contains the differences between the two versions of the model. Next, we expand this initial delta using some selected regression testing strategy by adding further elements from the SUT model. Based on this *expanded delta* a new test set is generated by selecting test cases which cover model elements contained in the (expanded) delta. As a last step, we reduce the test set by applying some test set reduction strategy. We perform test set reduction primarily to avoid redundant test cases (i.e., to ensure that the final test set only contains original test cases). This results in a more efficient testing process, as the necessary amount of time for (re)testing is reduced.

Our approach is *completely unaware* of any (meta)model. However, by allowing the calculations in each step to be customized and constrained by means of OCL queries, we successfully circumvent the need for meta information and enable a broad range of existing models to be supported.

In the following, the four tasks, *delta calculation*, *delta expansion*, *test set generation* and *test set reduction* are discussed in more detail.

5.1 Delta calculation

Calculating the *delta* (change set) is the first step of our method. Depending on whether one uses a combined SUT/Test model, prior to calculating the delta, the scope of the delta calculation needs to be defined, i.e., which model elements of the SUT need to be considered during delta calculation. To define the scope, we use OCL queries. If one only wants to consider potentially changed *Class* elements of the SUT model, for example an OCL query as in Listing 1 suffices. The query from Listing 1 assumes that each SUT specific model element from the SUT *Base Model* has a stereotype *SUT* applied. In our example the stereotype *SUT* is defined in a profile named *Test*. As already stated, the usage of such *scoping* queries is optional. However, if using a combined SUT/Test model, they are vital to assure that the resulting delta only contains elements from the SUT rather than test model related elements.

```
context InstanceSpecification:
  allInstances()->select(e |
    e.getAppliedStereotype('Test::SUT') <> null);
```

Listing 1. OCL query to define the scope of the delta calculation

With the scope defined, the delta can be calculated. The underlying idea hereby is to use the notion of a *left* (Base Model) and a *right* (Working Model) model. Comparing these two models and their elements results in the delta.

More specifically, elements from the right model which are different to their counterparts from the left model are added to the delta. The matching of respective model elements between the left and the right model is achieved using the IDs of the model elements. If an ID changes, we use some similarity metric–based back up strategy to identify respective counterparts. In case of newly added model elements (i.e., elements, only appearing in the right model) they are just added to the delta without any further consideration (provided they are not excluded due to some scoping query). In case of deleted model elements in the right model (i.e., elements, only appearing in the left model) the delta remains unchanged. Applying delta calculation to the models from Fig. 8 (by respecting the scoping query from Listing 1) results in a delta containing the two model elements *C5* and *C6*. More formally, $\delta = Base\,Model \setminus Working\,Model = \{C5, C6\}$.

Applying delta calculation by applying the OCL query from Listing 2 to two successive versions of our online shop model where the latter contains a modified *ShoppingCart* class, results in a delta containing this very class, i.e., $\delta = \{ShoppingCart\}$. The query from Listing 2 is similar to the one from Listing 1, yet the profile has been changed from *Test* to *UTP*.

```
context InstanceSpecification:
  allInstances()->select(e |
    e.getAppliedStereotype('UTP::SUT') <> null;
```

Listing 2. OCL query to define the scope of the delta calculation regarding UTP

5.2 Delta expansion

Expanding the delta is where a concrete regression test selection strategy comes into play for the first time. It should be obvious that the initially calculated delta already represents a regression test selection strategy, viz., the minimal change set only considering the modified elements, but nothing else. Yet, at the most, such a simple strategy does not suffice since it is necessary to evaluate whether the modified parts (i.e., elements) in the SUT also affect unchanged parts. At this stage we again employ OCL queries to define the concrete expansion strategy. For example, the OCL query from Listing 3 states, that the concrete strategy shall expand the initial delta by either adding all classes which reference classes contained in the initial change set or classes which are referenced by the very classes contained in the initial delta.

```
context InstanceSpecification:
 self->asBag()->
  union(self.relationships->
   iterate(rel:Relationship;
    result: Set(InstanceSpecification) = Set{} |
     result->union(rel.relatedElement->
       asType(InstanceSpefication))));
```

Listing 3. OCL query used to expand the delta according to a selected regression testing strategy, e.g., incorporating referenced classes

In case of the models from Fig. 8, the strategy from Listing 3 adds the classes *C3* and *C4* to the initial delta, resulting in the expanded delta. More formally, $\delta' = \delta \cup \gamma$, where $\gamma = \{C3, C4\}$ under the strategy from Listing 3. Thus, $\delta' = \{C3, C4, C5, C6\}$.

Applying this kind of delta expansion to our online shop example from Section 3 with a delta from the previous section, i.e., $\delta = \{ShoppingCart\}$, and an OCL query as in Listing 4 results in an expanded delta containing two classes, viz. *ShoppingCart* and *Customer*, i.e., $\delta' = \{ShoppingCart, Customer\}$. The term *DELTA* in Listing 4 is replaced by the name of elements in δ'. This step is executed by our engine to parametrize the OCL Queries to achieve more general OCL queries. We also apply this replacement in the following queries.

```
context InstanceSpecification:
 allInstances()->select(e | e.relationships->
  exists(rel |
   rel->contains(element:Element |
   element.classifier.name  in (DELTA)
   and element.getAppliedStereotype('UTP::SUT') <>
    null)));
```

Listing 4. OCL query used to expand the delta, already containing the *ShoppingCart* class, to also include classes using the *ShoppingCart* class

The query from Listing 4 selects all classes which use (i.e., have a directed relationship to) the *ShoppingCart*. Further, it also restricts the scope to only selecting classes from the SUT. In case of UTP, such a scoping restriction is necessary to not select a test context which also is defined using the UML class notion and has direct relationships to the SUT classes for which it defines a test context.

5.3 Test set generation

With the expanded delta, we next generate a new test suite. As the format of potentially available test cases may differ (e.g., activity diagrams, sequence diagrams, a requirement definition or also a trace) we again employ OCL queries to define the scope, i.e., the concrete test case selection strategy, for the resulting test suite. In other words, we define what kind of test cases to include in the newly generated test suite. For example, the OCL query from Listing 5 restricts the scope of a newly generated test suite to only contain test cases, defined using UML activity diagrams.

```
context Activity:
 allInstances()->select(e | e.
  getAppliedStereotype('Test::Testcase') <> null;
```

Listing 5. OCL query used to restrict the scope of the newly generated test suite, e.g., only to use test cases, defined by activity diagrams

Given the set of potential test case candidates based on a scoping/selection strategy, we then select the respective test cases from this set. This is done by evaluating associations either from a test case candidate to the elements

of the expanded delta, or, the other way round, by evaluating associations from elements of the expanded delta to potential test case candidates. If there exists some association either way, a test case is selected and added to the newly generated test suite. In the case of the models from Fig. 8 the newly generated test suite would contain the test cases *TC3*, *TC4*, *TC5* and *TC6*. More formally, $\tau = TC \cap \delta'$, where $\cap$ stresses connectedness between potential test cases and elements from the expanded delta, TC denotes the set of available test cases and τ denotes a new test suite.

Using the expanded delta for our running example from the previous Section, i.e., $\delta' = \{ShoppingCart, Customer\}$ results in a test set containing the test cases from the test contexts as shown in Fig. 2 and 3, i.e., $\tau = \{addSomeItems, addDifferentItems, \cdots\}$. The respective OCL query is given in Listing 6. The query from Listing 6 is again more or less the same as the query from Listing 5, but the profile has been changed from *Test* to *UTP* and the context to *Operation* to meet the UTP.

```
context Operation:
 allInstances()->select(e | e.
  getAppliedStereotype('UTP::Testcase') <> null;
```

Listing 6. OCL query used to restrict the scope of the newly generated test suite, e.g., only to use test cases, defined by operations in the context of UTP

This initial set of test cases requires further reduction, as it also contains tests not related to our changed *ShoppingCart* class. This is discussed in Section 5.4.

At the time of writing, the definition of the various OCL queries is done manually. However, we are about to create a library of useful OCL queries for regression testing. In defining a query for regression testing, a tester need not follow any requirements posed by our approach,. It is only necessary that the application of the respective model–based testing approach be valid. Hence, our approach is also completely language independent, as it can deal with any kind of model and hence, any kind of target language, used to generate test cases into.

As our approach evolved from the area of model versioning and not primarily software testing, our terminology differs from a testers', defined e.g. in [47]. Delta calculation corresponds to *change identification* in [47], delta expansion to *impact analysis*, and test set generation to *regression test selection*.

5.4 Test set reduction

Regarding test set reduction, we have identified two situations

1. to remove test cases with the same coverage, and
2. to remove uninteresting test cases (i.e., not covering the changed model elements).

Both of these situations are discussed in the following.

When removing test cases with the same coverage, our strategy is as follows. First, for each test case contained in a newly generated test suite, we calculate its coverage by the elements of the SUT model the test includes. Next, we create all two–element subsets from the test suite τ. We then calculate a similarity score for each of those subsets. Finally, based on a user defined threshold we consider the similarity score of each two–element subset and, given that some similarity score exceeds the defined threshold, one of the two test cases from the subset in question is removed from the test suite τ. More formally, $\tau' \subseteq \tau$ where $\forall(a,b) \mid a,b \in \tau', \pi(a) \cap \pi(b) = \varnothing$ where π denotes the operator calculating the coverage of a test case a or b. Such a reduction is done without the use of OCL queries.

When applying reduction to remove uninteresting test cases from the test set, we again rely on OCL queries. At this point, we skip showing some arbitrary query for our exemplary class diagram from Fig. 8 (due to no further specified activities). We instead provide a concrete query for our running example, which removes test cases from the test set from the previous Section, not related to the *ShoppingCart* class (i.e., *addSomeAddress*).

```
context Interaction:
 allInstances()->select(e |
  e.lifelines->exists(line |
   line.represents->contains(
    element:ConnectableElement |
     element.classifier.name in (DELTA) and
      element.getAppliedStereotype('UTP::SUT') <>
      null)));
```

Listing 7. OCL query to reduce the newly generated test suite, e.g., only to use test cases, which use the *ShoppingCart* class

Listing 7 shows the respective OCL query. Running this query on the test set τ from the previous Section removes all test cases not related to the *ShoppingCart* class, i.e., which do not make use of it. Thus, the resulting reduced test set τ' would, besides containing all test cases from the *ShoppingCartTests* context, further contain *updateCartWithSomeItems*, *updateCartWithDifferentItems* and *emptyCart* from the *CustomerTests* test context, which were implicitly added during delta expansion.

Performing test set reduction results in an obvious cost benefit. However, it should be kept in mind, that in some cases this may result in undetected errors in the SUT. Thus, test set reduction, and also the respective threshold, should be applied and defined carefully.

6 OCL-based regression testing techniques

In the following we show how our method can be applied on various test selection, test set minimization and,

test prioritization strategies commonly used in regression testing. We will not give an overview of the various techniques, as this would go beyond the scope of this article. However, we provide further references, where necessary.

6.1 Test case selection

Test case selection addresses the problem of selecting *necessary* test cases for a modified release of a SUT for uncovering potential regressions. For this, various strategies exist in regression testing [48]. As our approach is model-based, obviously not all of those selection techniques (e.g., symbolic execution or program slicing) can be realized by our method. Yet, certain techniques can be realized quite easily (e.g., modification based selection or test case selection using a graph walk approach) as we show in the following.

6.1.1 Modification

The underlying assumption of modification–based test case selection is to identify modifications in design models of the SUT. Our proposed method already realizes this selection strategy at no cost by the delta calculation (see Section 5.1). Thus, we do not require an extra OCL query for modification based test case selection, but instead build upon the change identification capabilities of MoVE (see Section 4.2).

Assuming we have two different, consecutive versions of our web shop model (see Fig. 1), in which the *ShoppingCart* class differs, modification based selection would identify this very class and subsequently select test cases attached to this class.

6.1.2 Graph walk

Rothermel and Harrold introduced graph walk as a test case selection technique based on Control Dependence Graphs (CDGs) [49]. By a depth-first traversal of the CDGs of different versions of a program, they identify traces that reach modified, i.e., mismatching, nodes. Subsequently, they select all test cases that execute the nodes prior to the mismatching ones.

For model-based regression testing, we search the structure of the model (i.e., the class hierarchy) of the SUT to identify the propagation of modifications. In the following, we present two graph walk approaches, *link–traversal* and *link–inheritance*, which extend an initially calculated delta (see Section 5.1).

Link–Traversal Our first graph walk approach is called *link–traversal*, i.e., we search the graph of the model by following usages inside the model (e.g., references between the classes). The OCL query from Listing 4 in Section 5.2 already shows an application of link–traversal, i.e., we select all those classes which refer to classes from the δ. In this way, we select those classes affected by changes of classes selected during delta calculation (see Section 5.1).

Assuming that $\delta = \{ShoppingCart\}$, using the query from Listing 4 for link–usage we get $\delta' = \{ShoppingCart, Customer\}$. If we again consider Fig. 1 we can see that *Customer* uses *ShoppingCart*, thus, changes in the *ShoppingCart* class propagate into *Customer*.

Link–Inheritance Our second graph walk approach is called *link–inheritance*, i.e., we search the graph of the model by following inheritance relationships top–down. For this, we assume $\delta = \{Person\}$. As shown in Fig. 1, *Customer* extends *Person*. Thus, we expect δ' to also contain *Customer*, as changes from *Person* will propagate to *Customer*. Hence, for *link–inheritance* we define an OCL query as in Listing 8 to select all those classes which inherit from *Person* that is, classes where changes from *Person* propagate to.

```
context InstanceSpecification:
 allInstances()->select(inst |
  inst->sourceDirectedRelationships->exists(rel |
   rel.targets->contains( target |
    target.classifier.name in (DELTA)  and
    rel.source.getAppliedStereotype('UTP::SUT') <>
     null)))
```

Listing 8. OCL query realizing Link–Inheritance for Person

Executing the query from Listing 8 results in $\delta' = \{Person, Customer\}$.

6.2 Test set reduction

In regression testing, test set reduction is the act of reducing the set of selected test cases by removing unnecessary/unchanged tests. Due to its NP-complete nature, efficient reduction strategies rely on heuristics. However, there exist a few other approaches [48].

In the context of our approach, test set reduction is performed on models. In the following, we present two methods, first, one that removes unchanged tests and second, one that removes unnecessary tests whose coverage lies outside δ'.

6.2.1 Reduction by removing unchanged tests

Reduction by removing unchanged tests is achieved without the use of any OCL queries. For this, we resort to MoVE and its change identification capabilities (see Section 4.2). By formulating the set of existing tests and of selected tests τ as partial models, MoVE identifies tests from τ, which have been changed to their previous versions. Thus, it supplies the set of changed tests.

6.2.2 Reduction by removing tests not covering δ

By applying reduction in removing tests not covered in δ, we effectively remove *unnecessary* tests. Section 5.4

already shows an application of this reduction strategy. Consider again δ, which initially only contained *ShoppingCart*. After expanding by link–traversal, $\delta' = \{ShoppingCart, Customer\}$. Based on this δ' a new test set τ would contain sixteen test cases, among them nine unnecessary from *CustomerTests* (see Fig. 3). Thus, to reduce the test set to the relevant test cases, i.e., test cases which examine *ShoppingCart*, we defined the OCL query from Listing 7. Applying this query on the initial test set τ, which contains the sixteen test cases from *ShoppingCartTests* (see Fig. 2) and *CustomerTests* (see Fig. 3), results in a reduced test set τ', which contains only seven test cases anymore (see Section 5.4).

6.3 Test case prioritization

Test case priorization seeks to find the ideal ordering of test cases so that the tester obtains maximum benefit [48]. In their survey, Yoo and Harman list various techniques, among them coverage- or interaction-based techniques. For our method, in the following we examine two techniques related to test coverage.

6.3.1 Priorization based on δ

Our first prioritization strategy prioritizes tests based on their selection order, i.e., whether they are selected by virtue of referring to any element of δ or δ'. More formally, our priorization strategy is defined as $\Pi(\tau) \cap \delta > \Pi(\tau) \cap (\delta' \setminus \delta)$, expressing that the execution order of a test, whose coverage coincides with δ is higher then the order of a test whose coverage only coincides with δ'. Recall, that Π is the operator, that by applying π on a test set calculates the coverage of a test case, i.e., which elements (e.g., classes) of the SUT are covered by a test case *tc*.

Assuming that $\delta = \{ShoppingCart\}$, $\delta' = \{ShoppingCart, Customer\}$ and τ' contains all tests from *ShoppingCartTests* (see Fig. 2) and *updateCartWithSomeItems*, *updateCartWithDifferentItems* and *emptyCart* from the *CustomerTests* (see Fig. 3), this priorization strategy would assign test cases related to the *ShoppingCart* a higher execution order than test cases related to *CustomerTests*.

6.3.2 Priorization based on δ'

The second test case priorization strategy examined in this article is based on δ'. Basically, tests whose coverage coincide with δ' get assigned a higher execution order than ones whose coverage only coincides with δ. More formally, $\Pi(\tau) \cap \delta' > \Pi(\tau) \cap \delta$. This prioritization strategy is essentially the inverse of prioritizing tests based on δ, as discussed in the previous section. We thus skip a further discussion of this priorization strategy.

Observe that prioritization based on δ puts a stronger focus on coverage in a sense that changed classes will be tested first. However, our second strategy which is based on δ' puts a stronger focus on interactions between changed classes and the reminder of the system.

	Graph Walk	
Modification	Link–Traversal	Link–Inheritance
4	16	12

Table 2. Number of test cases selected by different selection techniques

7 Discussion

In this article we showed how regression testing can be supported and enhanced by model–based testing. Building upon a model versioning engine, i.e., MoVE, and additional OCL statements we implemented different test case selection, test set reduction and, finally, test case prioritization strategies. For this, OCL occupies a key role as the specification language for the various mentioned strategies, i.e., test case selection, test set reduction, and test case prioritization. The task of MoVE is to provide the necessary infrastructure and execution environment for our method. Using an illustrative example, i.e., our Webshop, we gradually applied the different steps of our proposed model–based regression testing method and showed its feasibility.

Table 2 shows our results in applying different test case selection strategies on our Webshop example. Obviously, for *modification* we get the smallest possible set of test cases, i.e., those test cases evaluating any model element of our initial delta (δ) which is calculated between the base model and the working model. By applying *link–traversal*, δ is expanded with model elements, which link to (i.e., *use*) elements from δ. The resulting extended delta (δ') delivers a larger test set, as it includes further test cases besides those of the initially changed model elements of the SUT. Finally, also for our third evaluated test case selection strategy, i.e., *link–inheritance*, we get a larger test set than with *modification*, since it also considers further model elements, but this time those which inherit from the initially changed model elements. Thus, applying different test case selection strategies, implemented by OCL, we obtain different test sets.

Table 3 shows the results of applying our proposed test set reduction strategies. Applying our first strategy, i.e., to remove only unchanged tests, results in no change in the actual size of the test set. This is simply due to the assumption that the tests have been modified. However, in applying our second strategy, i.e., removing tests not coinciding with δ, the size of the test set is reduced significantly. The reason is that with this strategy we remove those tests which were added due to expanding δ. However, this expansion yields a test set which contains tests not even associated with model elements from δ.

Removing Unchanged Tests	Removing Tests not Coinciding with δ
16	7

Table 3. Size of reduced test set by Different reduction strategies (for the test set calculated by link–traversal)

Prioritization based on δ	Prioritization based on δ'
addSomeItems	emptyCart
addDifferentItems	updateCartWithSomeItems
removeItems	updateCartWithDifferentItems
showCartContents	addSomeItems
emptyCart	addDifferentItems
updateCartWithSomeItems	removeItems
updateCartWithDifferentItems	showCartContents

Table 4. Execution order of test cases for different prioritization strategies (for the test set calculated by Link–Traversal and reduced by removing tests not coinciding with δ)

Thus, our second reduction strategy is valuable for removing such *unnecessary* tests from a test set.

Table 4 illustrates the results of applying our proposed test case prioritization strategies. First, we applied prioritization which assigns test cases that were selected by virtue of association with any element of δ a higher priority (see first column). Consequently, test cases associated with the *ShoppingCart* class get assigned a higher execution order then those associated with the *Customer* class. In contrast, our second strategy, i.e., to prioritize test cases based on δ', does the inverse of prioriziting test cases based on δ. Thus, the resulting execution order of test cases is flipped.

Although we have successfully applied our proposed method, it does have some limitations. First of all, currently, our method only works with class diagrams, i.e., we do not consider any changes in the underlying behavior of the application described by sequence or activity diagrams for example. As a result of this, not every test case selection strategy is feasible with our approach, i.e., integer programming or symbolic execution [48]. Finally, the necessary OCL queries which we use to implement the various strategies (test case selection, test set reduction and test set prioritization), may become quite complex, and thus, error prone, potentially invalidating the results of some strategy. However, since the queries are usually very similar and additionally, we already provide a set of template queries (i.e., the illustrated queries of this article), it suffices to apply minor adaptions to those queries, which in turn reduced the risk of introducing errors in the queries.

8 Summary and future work

In this article we presented a method for model–based regression testing, which allows existing model–based testing approaches to be equipped with features for regression testing. The approach is based on the model versioning engine MoVE and additional OCL statements to implement different test case selection, reduction, and prioritization strategies. Using an illustrative example, we have shown that our method is feasible and produces meaningful results. Thanks to MoVE's support for heterogeneous models, which at least have a valid metamodel, our approach is very flexible in that it can deal with many different models. Further, the application of OCL provides additional flexibility by means of configuring test case selection, test set minimization and test case prioritization strategies.

Regarding future work, we have identified five issues. First of all, we want to extend the current method to support additional UML diagrams besides class diagrams. The biggest challenge thereby is to support complex diagram types which are not directly supported by the standard EMF meta model ECore. For these diagram types EMF Compare is too generic and therefore the calculated deltas have a lack of semantics. We currently experiment with the Epsilon Compare Language (ECL) [50] which allows to compare homogeneous or heterogeneous models by a rule based language. With ECL we are able to provide more specific comparison results and therefore better support more complex diagram types. The presented method in this article would not change except that EMF Compare will be enhanced by a set of ECL rules. This will mainly result in a more powerful approach, as we can then support further test case selection, test set minimization and test case prioritization strategies. Second, we also plan to support further model–based testing techniques, not UML but different modeling approaches, e.g., domain specific language modeling or TestSheets [51]. Third, since we plan to support further UML models as well as further modeling approaches, we want to extend the set of template OCL queries. Fourth, we plan to also consider regression testing of non–functional properties like security [52, 53]. Finally, we plan to perform empirical studies, i.e., controlled experiments and industrial case studies to investigate the impact of the presented approach on regression testing.

Acknowledgments

This research was partially funded by the research projects MBOSTECO (FWF P 26194-N15) and QE LaB—Living Models for Open Systems (FFG 822740).

References

1. Dias Neto, A.C., Subramanyan, R., Vieira, M., Travassos, G.H.: A Survey on Model–based Testing Approaches: A Systematic Review. (2007) 31–36
2. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2007)

3. Breu, M., Breu, R., Low, S.: Living on the MoVE: towards an architecture for a living models infrastructure. In: 2010 Fifth International Conference on Software Engineering Advances (ICSEA). (August 2010) 290 –295
4. Zech, P., Felderer, M., Kalb, P., Breu, R.: A generic platform for model-based regression testing. In: Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change. Springer (2012) 112–126
5. IEEE: Standard Glossary of Software Engineering Terminology. IEEE (1990)
6. Bourque, P., Dupuis, R., eds.: Software Engineering Body of Knowledge (SWEBOK). IEEE Computer Society, EUA (2004)
7. Rothermel, G., Harrold, M.J.: Analyzing Regression Test Selection Techniques. IEEE Transactions on Software Engineering **22** (1996)
8. Yang, Q., Li, J.J., Weiss, D.: A survey of coverage based testing tools. In: Proceedings of the 2006 international workshop on Automation of software test. AST '06, New York, NY, USA, ACM (2006) 99–103
9. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9** (July 1987) 319–349
10. Elbaum, S., Malishevsky, A.G., Rothermel, G.: Prioritizing test cases for regression testing. SIGSOFT Softw. Eng. Notes **25** (August 2000) 102–112
11. Harrold, M.J., Gupta, R., Soffa, M.L.: A methodology for controlling the size of a test suite. ACM Trans. Softw. Eng. Methodol. **2**(3) (July 1993) 270–285
12. Jones, J., Harrold, M.: Test-suite reduction and prioritization for modified condition/decision coverage. Software Engineering, IEEE Transactions on **29**(3) (2003) 195–209
13. Chen, T., Lau, M.: A new heuristic for test suite reduction. Information and Software Technology **40**(5) (1998) 347–354
14. Jeffrey, D., Gupta, R.: Test suite reduction with selective redundancy. In: Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on. (2005) 549–558
15. Jeffrey, D., Gupta, N.: Test suite reduction with selective redundancy. Software Maintenance, IEEE International Conference on **0** (2005) 549–558
16. Rothermel, G., Harrold, M.J., von Ronne, J., Hong, C.: Empirical studies of test–suite reduction. Software Testing, Verification and Reliability **12**(4) (2002) 219–249
17. Heimdahl, M.P.E., George, D.: Test-Suite Reduction for Model Based Tests: Effects on Test Quality and Implications for Testing. In: Proceedings of the 19th IEEE international conference on Automated software engineering. ASE '04, IEEE Computer Society (2004) 176–185
18. Foundation, E.: Eclipse CDO (2014) http://wiki.eclipse.org/CDO [accessed: March 31, 2015].
19. Brügge, B., Creighton, O., Helming, J., Kogel, M.: Unicase – An Ecosystem for Unified Software Engineering Research Tools. In: Third IEEE International Conference on Global Software Engineering, ICGSE. (2008)
20. IBM: Rational DOORS. (2014) http://www-01.ibm.com/software/awdtools/doors/ [accessed: March 31, 2015].
21. Barteit, C., Molter, G., Schumann, T.: A model repository for collaborative modeling with the jazz development platform. In: 42nd Hawaii International Conference on System Sciences, 2009. HICSS '09. (January 2009) 1 –10
22. microTOOL: in-step die projektmanagement-software. http://www.microtool.de/instep/de/index.asp (2014)
23. Thapa, V., Song, E., Kim, H.: An approach to verifying security and timing properties in UML models. In: 2010 15th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS). (March 2010) 193 –202
24. OMG: Object Constraint Language, V2.3.1. (2012)
25. Marchesi, M.: OOA metrics for the unified modeling language. In: Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering, 1998. (March 1998) 67 –73
26. Bergmann, G., Horváth, Á., Ráth, I., Varró, D.: Incremental evaluation of model queries over emf models: A tutorial on emf-incquery. In: Modelling Foundations and Applications. Springer (2011) 389–390
27. Stoerrle, H.: A PROLOG-based Approach to Representing and Querying Software Engineering Models. (2007)
28. Chimiak-Opoka, J., Felderer, M., Lenz, C., Lange, C.: Querying UML models using OCL and prolog: A performance study. In: IEEE International Conference on Software Testing Verification and Validation Workshop, 2008. ICSTW '08. (April 2008) 81 –88
29. Deng, W., Liang, Y.: Reason on UML diagrams with answer set programming. In: 2008 International Conference on Computer Science and Software Engineering. Volume 1. (December 2008) 205 –209
30. Brain, M., Cliffe, O., Vos, M.D.: A pragmatic programmer's guide to answer set programming. In: Software Engineering for Answer Set Programming (SEA09). (September 2009) 49–63
31. OMG: UML Testing Profile, Version 1.0. (2005) available at `http://www.omg.org/spec/UTP/1.0/PDF` [accessed: March 31, 2015].
32. ETSI'S : TESTING AND TEST CONTROL NOTATION VERSION 3 (2015) `http://www.ttcn-3.org/` [accessed: August 24, 2015].
33. Massol, V., Husted, T.: JUnit in Action. Manning Publications Co., Greenwich, CT, USA (2003)
34. Briand, L.C., Labiche, Y., He, S.: Automating Regression Test Selection based on UML Designs. Inf. Softw. Technol. **51**(1) (2009)
35. Farooq, Q., Iqbal, M.Z.Z., Malik, Z.I., Nadeem, A.: An approach for selective state machine based regression testing. In: A-MOST 07. (2007)
36. Chen, Y., Probert, R.L., Sims, D.P.: Specification–based Regression Test Selection with Risk Analysis. In: CASCON '02. (2002)
37. Chen, Y., Probert, R.L., Ural, H.: Model-based regression test suite generation using dependence analysis. In: Proceedings of the 3rd international workshop on Advances in model-based testing. A-MOST '07, New York, NY, USA, ACM (2007) 54–62
38. Korel, B., Tahat, L., Vaysburg, B.: Model based regression test reduction using dependence analysis. In: Proceedings of the International Conference on Software Maintenance (ICSM'02), Washington, DC, USA, IEEE Computer Society (2002) 214–

39. Fraser, G., Aichernig, B.K., Wotawa, F.: Handling Model Changes: Regression Testing and Test-Suite Update with Model-Checkers. Electronic Notes in Theoretical Computer Science **190**(2) (2007) 33 – 46 Proceedings of the Third Workshop on Model Based Testing (MBT 2007).
40. Windmüller, S., Neubauer, J., Steffen, B., Howar, F., Bauer, O.: Active continuous quality control. In: Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering, ACM (2013) 111–120
41. Felderer, M., Agreiter, B., Breu, R.: Managing evolution of service centric systems by test models. In: IASTED 2011. (2011)
42. Felderer, M., Agreiter, B., Breu, R.: Evolution of Security Requirements Tests for Service–centric Systems. In: ESSOS 2011. (2011)
43. Breu, R.: Ten Principles for Living Models - A Manifesto of Change-Driven Software Engineering. In: CISIS, IEEE Computer Society (2010) 1–8
44. Breu, R., Agreiter, B., Farwick, M., Felderer, M., Hafner, M., Innerhofer-Oberperfler, F.: Living models-ten principles for change-driven software engineering. Int. J. Software and Informatics **5**(1-2) (2011) 267–290
45. No Magic, Inc.: MagicDraw (2014) `http://www.magicdraw.com/` [accessed: March 31, 2015].
46. Eclipse Foundation: EMF Compare Developersguide (2014) `http://www.eclipse.org/emf/compare/documentation/latest/developer/developer-guide.html#Diff` [accessed: March 31, 2015].
47. Farooq, Q.u.a., Iqbal, M.Z., Malik, Z., Riebisch, M.: A Model-Based Regression Testing Approach for Evolving Software Systems with Flexible Tool Support. (2010) 41–49
48. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Software Testing, Verification and Reliability **22**(2) (2012) 67–120
49. Rothermel, G., Harrold, M.J.: A safe, efficient algorithm for regression test selection. In: Software Maintenance, 1993. CSM-93, Proceedings., Conference on, IEEE (1993) 358–367
50. Eclipse Foundation: Epsilon Comparison Language (2015) `http://www.eclipse.org/epsilon/doc/ecl/` [accessed: August 24, 2015].
51. Atkinson, C., Barth, F., Brenner, D.: Software Testing using Test Sheets. In: Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on, IEEE (2010) 454–459
52. Felderer, M., Fourneret, E.: A systematic classification of security regression testing approaches. International Journal on Software Tools for Technology Transfer **17**(3) (2015) 305–319
53. Felderer, M., Zech, P., Breu, R., Büchler, M., Pretschner, A.: Model-based security testing: a taxonomy and systematic classification. Software Testing, Verification and Reliability (2015) doi:10.1002/stvr.1580.