Journal of Software: Evolution and Process

JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2015; **27**:913–952
Published online 29 October 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.1752

Delta-based regression testing: a formal framework towards model-driven regression testing

Maryam Nooraei Abadeh[1] and Seyed-Hassan Mirian-Hosseinabadi[2,*,†]

[1]*Department of Computer Engineering, Science and Research Branch, Islamic Azad University, Tehran, Iran*
[2]*Department of Computer Science, Sharif University of Technology, Tehran, Iran*

ABSTRACT

The increase in complexity and rate of technological changes in modern software development has led to a demand for systematic methods that raise the abstraction level for system maintenance and regression testing. Model-driven development (MDD) has promised to reduce extra coding efforts in software maintenance activities using traceable change management. The research described in this paper presents a Z-notation-based framework, called delta-based regression testing (DbRT), for formal modeling of regression testing in the context of MDD. The framework proposes to propagate the changes from a software specification to testing artifacts in order to preserve consistency after system evolution. Also, an effective delta-based selection technique is provided for regression testing at the platform-independent level. The framework is further enriched by introducing a new category of coverage patterns for DbRT. Complex coverage patterns can be defined using a declarative query language syntax for examining the adequacy of regression testing. Finally, an implementation technique and an analysis plan are provided to assess the effectiveness of the proposed framework. The assessment process is expected to be beneficial to both the platform-independent and platform-specific level of DbRT by identifying the desired coverage according to available testing resources. Copyright © 2015 John Wiley & Sons, Ltd.

Received 18 November 2014; Revised 8 August 2015; Accepted 31 August 2015

KEY WORDS: consistency; coverage criteria; delta model; model-driven development; model transformation; regression testing; test case selection

1. INTRODUCTION

The advantages of model-driven architecture (MDA) [1], for example, abstraction modeling, automatic code generation, reusability, effort reduction, and efficient complexity management, can influence all phases of the software life cycle. Model transformation and traceability are two key concepts in model-driven software development, which can provide a traceable chain between life cycle artifacts. The systematic use of models by combining domain-specific modeling languages (DSMLs) and model transformers, analyzers, and generators provides enough flexibility and automation for model-driven development (MDD) [2]. On the other hand, using formal methods as a technique based on mathematics in the system specification, development, and verification ensures the completeness and correctness of systems. Using a combination of MDD abilities with the completeness and preciseness of formal methods, it is possible to increase productivity and quality of the model-driven analysis and design [2].

Software maintenance activities, for example, impact analysis and propagation of changes, regression test selection and prioritization, take up a high proportion of the software life cycle cost and time. In this paper, we propose to apply MDA capabilities in regression testing in a formal way to reduce the associated efforts. The purpose of regression testing is to verify new versions of a

*Correspondence to: Seyed-Hassan Mirian-Hosseinabadi, Department of Computer Science, Sharif University, Tehran, Islamic Republic of Iran.
†E-mail: hmirian@sharif.ac.ir

system to prevent functionality inconsistencies among different versions [3]. To avoid rerunning the whole test suite on the updated system, various regression test selection techniques have been proposed. The main activity is to choose test cases from the existing test suites that are necessary to exercise new functionalities according to a modified version [4]. Propagating design changes to the corresponding testing artifacts leads to a consistent regression test suite.

As a technical contribution of the current paper, we use the Z specification language [5] not only for its capability in software system modeling, development and verification but also for its adaptation in formalizing MDD concepts, for example, model refactoring, transformation rules (using pre-condition and post-condition), meta-model definition, and refinement theory to produce a concrete specification. Besides, verification tools such as CZT [6] and Z/EVES [7] have been well-developed for type-checking, proving, and analyzing the specifications written in the Z notation. Although Object Constraint Language (OCL) can be used to answer some analysis issues in MDD, it is only a specification language and the mentioned mechanisms for consistency checking are not supported by OCL.

The main contributions of our framework can be summarized as follows: (i) a formal modeling framework is proposed for platform-independent regression testing based on UML2 state machine semantics and the Z notation syntax; (ii) a concrete scenario is introduced for the delta-based regression test selection technique (DbRTST) in a seamless and systematic way, which is shown by a case study; (iii) a declarative query syntax is defined to specify test goals to be covered during regression testing; (iv) a new category of coverage criteria, as a collection of rules that define test requirements, is specified for regression testing called delta-based coverage criteria; and (v) a plan consisting of the characteristics and metrics is provided to analyze delta-based regression testing.

The rest of the paper is organized as follows: Section 2 reconsiders the related concepts. Section 3 explains the motivation of our approach by an example. Section 4 extends the formalism for platform-independent testing. Section 5 introduces delta-based regression testing framework. Section 6 discusses our approach for automatic refinement and concrete testing. The practical discussion and analysis of the framework are provided in Section 7. Section 8 reviews the related works and compares the similar approaches to our work. Finally, Section 9 concludes the paper and gives suggestions for future works.

2. BACKGROUND

In this section, we review some concepts that are prerequisites for our formal framework.

2.1. Regression testing

Regression testing as a testing activity during the system evolution and maintenance phase can prevent the unexpected effects of the changes at different levels of abstraction. Three main approaches have been studied in regression testing to keep/maximize the value of the accrued test suite: selection, minimization, and prioritization.

Regression test selection techniques (RTSTs) select a cost-effective subset of valid test cases from a previously validated version to verify change effects. An RTST essentially consists of two major activities: identifying affected parts of a system after the maintenance phase and selecting a subset of test cases from the initial test suite to effectively test the affected parts of the system. A well-known classification of regression test cases is suggested by Briand *et al.* [4], which classifies test suites into obsolete, reusable, and re-testable test cases. Obsolete test cases are invalid for the new version and should be removed from the original test pool, whereas the others are still valid to be rerun.

Minimization techniques aim to reduce the size of a test suite by eliminating redundant test cases. Effective minimization techniques keep the level of coverage of a reduced subset the same as the original test suite while reducing the maintenance costs and time. In comparison with test case selection techniques that also attempt to reduce the size of a test suite, the selection not only focuses on the current version of a system but most of the selection techniques are also change-aware [8].

Finally, test case prioritization techniques attempt to schedule test cases in such an order that meet desired properties, such as fault detection, at an earlier stage [9]. The important issues of regression

DOI: 10.1002/smr

testing at the platform-independent level will be solved by methodically analyzing system specifications and will be enhanced by following test strategies, for example, coverage criteria to adequately cover demanded features of the updated models. A main metric that is often used as the prioritization criterion in coverage-based prioritization is structural coverage. The intuition behind the idea is that early maximization of structural coverage will also increase the ability of early maximization of the fault detection [9].

2.2. *Model transformations and consistency*

Model transformations (MTs) use simple rules or complex graph-based patterns to manipulate models. Each MT rule is specified using a pre-conditioned pattern which determines the required condition to fire the rule and a post-condition pattern which describes the resulting model after applying the rule. In other words, pre-conditions express what may be true before an operational rule is executed, and post-conditions declare what will be true after the rule has been executed, as long as the pre-conditions were satisfied. Elements which only appear in the pre-condition of an MT are deleted, elements that only appear in the post-condition are created, the elements that are present on both sides are updated, and other model elements remain unchanged.

An important type of MTs in the MDA philosophy is a refinement rule, which transforms a model that exhibits a sufficient degree of independence, called a platform-independent model (PIM), to one or more models that combine the specifications of the PIM with the details of a special platform, called a platform-specific model (PSM). Model-driven engineering can be partially or completely automated using different types of model transformations. Model-to-model transformations are used in automated support of PIM-to-PSM mappings, which adds platform-specific knowledge to the PIM. Model-to-Text transformations are used as code generators to map PSMs to source code artifacts that will run on the target implementation platform [10].

The meta object facility (MOF) specification [11] is the foundation of meta-modeling in the MDA environment where models can be exchanged between applications and rendered into different formats, for example, XMI, and used to generate code artifacts. Transformation rules between models can be used at the meta-model MOF level, for transformations that need to map model languages. Currently, the MOF is used as a widely acceptable meta-modeling language to derive practical meta-modeling paradigms.

Model transformations and preserving consistency of models are potentially two interrelated concepts in MDA. When the consistency problem and model transformations are integrated, different interesting transformations can be defined, as mentioned by Hausmann *et al.* [12]. In our approach, bidirectional consistency-preserving transformations, described by Stevens [13], are desirable. Bidirectional and change-propagating model transformations are two important types of transformations which implicitly try to keep consistency and symmetry between source and target models. Using model transformations and traceability links, we can bridge the gap between MDD and model-driven testing (MDT). It can reduce maintenance and debugging problems by providing traceability links in a forward and backward direction simultaneously. Moreover, model transformation rules establish different granularities between source and target meta-models. Fine-grained traceability links will get a better handle to determine ‘something’ that has been changed or fault origins in ‘somewhere’ of a software model (or code). An important parameter to determine the level of granularity is the trade-off between precision and effort.

Two common languages for bidirectional and change propagating model transformations are Query/View/Transformation relations [10] and Triple Graph Grammars (TGGs) [14] respectively.

2.3. *Model-driven testing*

Model-driven testing can promote MDA advantages in software testing. MDT is started at a high level of abstraction to derive abstract test cases, and then transforms the abstract test cases to concrete test code using stepwise refinement. UML2 testing profile [15] is a standard for visual test specification and black-box testing in the four views, including architecture, behavior, data, and time. UML specifications of a behavioral model at the PIM level can be transformed into a test design model at the platform-independent test model (PIT). It enables early integration of software testing into the

DOI: 10.1002/smr

overall development process to promote failure prevention by early fault detection. When a PIM is defined in the design phase, it is possible to derive the corresponding PIT in the testing phase. Finally, codes and concrete test scripts can be generated from platform-specific design models and platform-specific test model (PST), respectively. Combining model-based testing and MDA capabilities, for example, traceability, automatic model transformation, and refinement tools provide a cost-effective way to verify system evolution especially in rapidly changing environments.

The general solution for model-driven testing, inspired by Dai [16], is shown in Figure 1. It shows the meta-model-based traceability/transformation between a system design model and its test requirement model. The source and target instance models conform to their corresponding meta-models. Refinement rules are used to automate the movement to a concrete environment step by step, for example, when enriching the PIT to the PST required test, specific properties must be added. When a test case detects any type of 'mistake' at different levels of abstraction, it is possible to follow traceability links to identify and resolve its origins at an early stage of the software development life cycle that results in enhancing the software quality and reducing the maintenance cost. In this paper, we extend the MDT philosophy to expose model-driven regression testing. Promoting the philosophy of MDA to the maintenance phase can reinforce worthy domains like 'Model Driven Software Maintenance', which leads to cost-effective configuration management.

2.4. Briefly on the Z notation

Z [5] is a formal specification language based upon mathematical logic and the Zermelo–Fraenkel set theory with an effective structuring mechanism for precisely modeling, specifying, and analyzing computing systems.

The symbols used in the Z notation are principally similar to common mathematical symbols, for example, it uses well-known symbols for number sets ($\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$), set operations ($\cup$, $\cap$, $\times$ and $\setminus$), quantifiers ($\forall$, $\exists$, $\nexists$ and $\exists_1$), and first-order logic ($\wedge$, $\vee$ and $\Rightarrow$). In the Z mathematical language, there are four ways of introducing a type: as a given set, as a free type, as a power set, or as a Cartesian product. The power set of A is shown by $\mathbb{P}$ A. A relation R over two sets X and Y is a subset of the Cartesian product, formally, $R \in \mathbb{P}(X \times Y)$. The domain and the range of R are denoted by **dom** R and **ran** R, respectively. The domain and range restriction of a relation R by a set P are the relation obtained by considering only the pairs of R where, respectively, the first elements and the second elements are members of P, formally $P \lhd R$ and $R \rhd P$.

A main feature of Z is providing a way of decomposing a specification into small pieces called schemas. The schemas can be used to explain structured details of a specification and to describe a transformation from one state of a system to another. Also, by constructing a sequence of specifications, each including more details than the last, a concrete specification that satisfies the abstract ones can be reached [5]. In more realistic projects, such a separation of concerns is essential

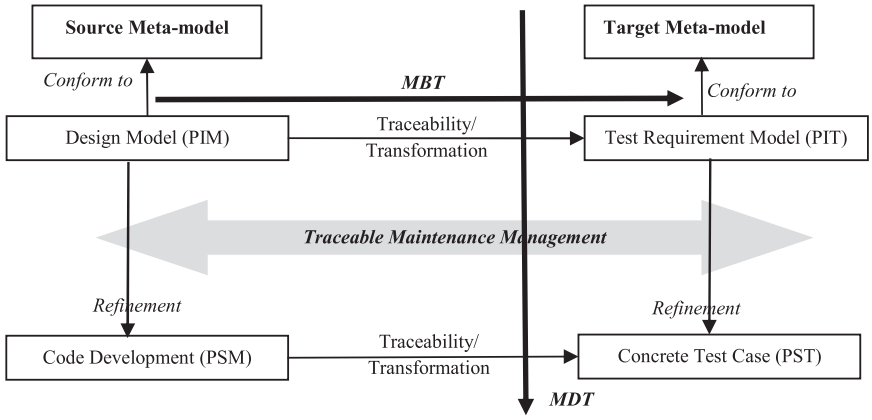


Figure 1. Model-driven testing philosophy.

DOI: 10.1002/smr

to decrease the complexity. Later, using the Z schema language, it is possible to describe different aspects of a system separately, then relate, and combine them.

An operation schema is defined as a transition from one subset of the state space (the set of states that describe a system behavior) to another with related input and output variables. State variables before applying the operation are termed pre-state variables and are shown undecorated. The corresponding post-state variables have the same identifiers with a prime (′) appended. The name of input and output variables should be ended by a question mark (?) and exclamation (!), respectively. In a Z operation schema, a state variable is implicitly equated to its primed post-state unless included in a Δ-list. A Δ-list is a schema including both before and after change variables. For an operation op, **pre** op is its pre-condition set. In fact, **pre** op is composed of the states which meet the pre-conditions of op. Furthermore, in the Z schema language, a schema can be used as a type, declaration, or (part of a) predicate [5]. The following is an example of a simple schema. The schema requires the input variable *in?* of type $\mathbb{Z}$ and produces the output variable *square!* as the same type of the input. The pre-conditions of the schema imply that $in? \in set$ and $set \neq \emptyset$. The post-conditions imply that in the modified state after the operation execution, $square! \geq 0$ and $square! = in? * in?$. That is, the square of an input number is positive and its square is the result of multiplying the number by itself.

Obviously, the post-condition defines a logical predicate which relates pre-state variables to post-state ones. Z specifications are composed, checked, and validated using interactive tools, for example, CZT [6] and Z/EVES [7]. The essential notations of the Z specification language are mentioned in Appendix A.

SchemaExample

$in?, square!: \mathbb{Z}$
$set: \mathbb{P}\,\mathbb{Z}$ // *Declaration Part*

$in? \in set$
$set \neq \emptyset$
$square! \geq 0$ // *Predicate Part*
$square! = in? * in?$

3. MOTIVATION: AN EXAMPLE

The example in Figure 2 shows an UML2 state machine that models the behavior of a web service at the design level. Before applying changes, the model includes six system states and associated transitions. The web service behavior begins in the 'Idle' state and accepts incoming task requests and processes them in the 'InService' state according to the routine manner unless an error/interrupt occurs. A processing service can be transmitted to the 'Finish', 'Suspend', or 'Cancelled' states according to the shown guard conditions. If a service becomes inaccessible, it is transmitted to the 'OutService' state. After any transition, the result model is provided by performing effects (e.g., calling a function, changing a variable value, or changing to another state). Suppose during the maintenance phase, the developer decides to improve the system behavior, for example, he decides to log errors and interrupts to remove their essential causes.

After undergoing corrections or improvements, changed elements are distinguished by the distinct tags. The main challenge is to determine a way to propagate the design changes using a well-defined delta model to the original test suites and to provide a technique based on the delta model to select an efficient and consistent regression test suite. According to Figure 1, traceability links are

DOI: 10.1002/smr

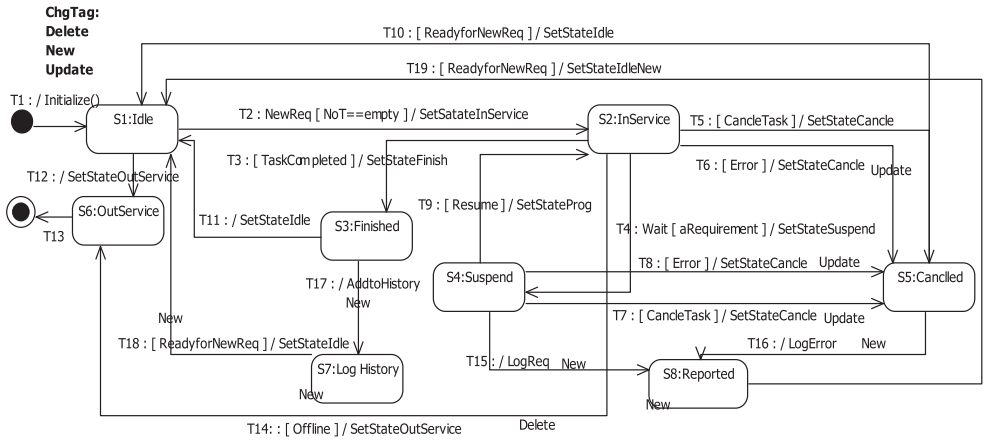


Figure 2. The UML state machine of a web service.

followed from the PIM to the PIT at the abstract level and refined to the PSM and PST, respectively. We derive test artifacts from abstract test models at the PIT level, so the PIT should be updated after evolving the corresponding PIM. The problem of change propagation to test artifacts from the updated model essentially involves two issues:

1. Using backward tracing, called debugging traceability links, to find out which elements could have caused an observed fault? Which other elements may depend upon erroneous elements?
2. Using forward tracing, called maintenance traceability links, to find which elements will be affected by a change? Will a particular element be affected by a change? Which elements could affect 'this' element?

Another strong motivation of this work is the lack of a query language to ask specific patterns of changes with their associated in modified models at both platform-independent and platform-specific levels. We propose a declarative query language to specify patterns of test requirements which should be covered in regression testing. Our query language defines ad hoc coverage criteria ranging from simple changed element coverage to complex predicate coverage with multiple condition coverage, for example, type of changed elements, number of changes, and type of change operations. It also promotes on-the-fly testing for regression test suites especially for ad hoc needs arising after system evolution. Then, the adequacy of the regression test suite is measured using a new category of coverage criteria called *delta-based coverage criteria*.

4. FORMAL OVERVIEW OF THE APPROACH

In this section, we propose the precise definitions of the behavioral model, the delta model, and model refactoring, which allow capturing the nature of the change unambiguously. Also, we define the dependencies between delta records in order to prepare an optimized delta model for regression testing. Finally, the abstract testing terminology based on the formalized behavioral model is described.

4.1. Test model

In the MOF meta-modeling environment, each model conforms to its meta-model. We integrate UML2 State Machines (SMs) meta-model and the theory of labeled and typed attributed graphs [17] to introduce the labeled state machine (LSM) as the behavioral model of our testing framework. An LSM is used to derive abstract test cases so it is called a *Test Model*. LSMs are more appropriate for our test space modeling rather than control flow graphs, because (i) LSMs naturally provide a

DOI: 10.1002/smr

transition system which is appropriate for change representation, (ii) LSMs can cover useful static parameters of a system beside its behavior (i.e., LSMs provide suitable means for describing class methods), so the impact analysis is performed precisely, (iii) LSM's meta-model provides a rich semantic to represent behavioral models, abstract test cases, coverage criteria, and their relations. However, our approach due to its formalizing style can be extended to other DSMLs.

Definition 1. (Test model). A test model is defined by a schema, named *TestModel*, where its elements are finite sets of the free-type *TestMetaModel*. This free type behaves as a meta-model for defining the elements of a test model. So, each test model is an instance model where its elements conform to the meta-model *TestMetaModel*. The test meta-model includes *STATE, TRANSITION, EVENT, GUARDCONDITION*, and *EFFECT* named by their *IDs.* A test model describes a behavior that includes visiting an active state and then triggering one event, satisfying a certain condition value assignment for guard conditions and traversing one of the transitions and acting an effect on the system. The granularity of the meta-model *TestMetaModel* is specified at a fine-grained level to provide precise change detection for regression test selecting and fault location discovering.
To take the advantages of the available theories about typed attributed graph modeling, a *Type* and some attributes and their *Values* are assigned to each element of a test model by the function *type* and *value*. At this level of abstraction, we introduce *IDs* and *Values* as given sets:

$$
\begin{aligned}
&[ID, Value]\\
&TestMetaModel ::= STATE\langle\langle ID\rangle\rangle\\
&\quad \mid TRANSITION\langle\langle ID\rangle\rangle\\
&\quad \mid EVENT\langle\langle ID\rangle\rangle\\
&\quad \mid GUARDCONDITION\langle\langle ID\rangle\rangle\\
&\quad \mid EFFECT\langle\langle ID\rangle\rangle
\end{aligned}
$$

We introduce two main binary relations: *FromSource* and *ToTarget*. The former shows the 'source' state from which a given 'transition' starts, and the latter shows the 'destination' (state) where a given transition ends. The sets *State*, *Transition*, *Event*, *GuardCon*, and *Effect* define distinct elements of *TestMetaModel*. The algebraic theory of graph transformation based on labeled graphs implies that changed elements should be labeled by a changed tag *ChgTag* to be distinguished in transformations shown by the relation *TagRelation* in the schema. Finally, the relation *labelE* is used to assign a suitable label including *Event*, *GuardCon*, and *Effect* to each transition.

The predicate part defines the conditions on the domain and range of the defined relations, for example, dom *FromSource* = *Trans*, *dom ToTarget* = *Trans* (all transitions should have a source and a destination state), ran *FromSource* $\subseteq$ *Stt* and ran *ToTarget* $\subseteq$ *Stt*. All elements of a test model, named *TotalElem,* are used for tracing and analyzing applied changes on the model in next steps.

$$
\begin{aligned}
&State == \text{ran } STATE\\
&Transition == \text{ran } TRANSITION\\
&Event == \text{ran } EVENT\\
&GuardCon == \text{ran } GUARDCONDITION\\
&Effect == \text{ran } EFFECT\\
&typeID == ID\\
&attrID == ID\\
&attinstanseset == attrID \twoheadrightarrow Value\\
&ChgTag ::= New \mid Update \mid Delete
\end{aligned}
$$

DOI: 10.1002/smr

TestModel

$TotalElem: \mathbb{P}\ TestMetaModel$
$Stt: \mathbb{F}\ State$
$Eve: \mathbb{F}\ Event$
$Guardcon: \mathbb{F}\ GuardCon$
$EffectT: \mathbb{F}\ Effect$
$Trans: \mathbb{F}\ Transition$
$FromSource: Transition \rightarrow State$
$ToTarget: Transition \rightarrow State$
$type: TestMetaModel \nrightarrow typeID$
$value: TestMetaModel \nrightarrow attinstanseset$
$labelE: Transition \leftrightarrow TestMetaModel$
$TagRelation: TestMetaModel \leftrightarrow ChgTag$

$\forall trans: Trans \bullet \exists s1, s2: Stt \bullet trans \mapsto s1 \in ToTarget \wedge trans \mapsto s2 \in FromSource$
$\text{ran}\ ToTarget \subseteq Stt; \text{ran}\ FromSource \subseteq Stt; \text{ran}\ labelE = EffectT \cup Eve \cup Guardcon;$
$\text{dom}\ ToTarget = Trans; \text{dom}\ FromSource = Trans; \text{dom}\ labelE \subseteq Trans$
$TotalElem = Stt \cup Trans \cup Eve \cup Guardcon \cup EffectT$

4.2. Model synchronization

Definition 2. (Model synchronization). Suppose SM and TM are source and target meta-models, respectively, and $\text{ConsR} \subseteq \text{SM} \times \text{TM}$ is the consistency relation to be established. We define a multi-levels model synchronizer on both sides, named *x-Directional Multi Level (xDML)* model synchronizer. Formally, the recursive free-type *ConsR* explains the model synchronizer *xDML*. The bidirectional form of *xDML*, named *BiDiML*, takes as input two source and target models satisfying the consistency relation *ConsR*, and their change models, $\mathbb{P}\ SM$ and $\mathbb{P}\ TM$, respectively. The outputs are two updated models where their modifications are synchronized. Forward and backward model synchronizers are defined by *FDML* and *BDML*. The former synchronizes a target model with the corresponding updated source model and the latter performs synchronization in the opposite direction. The end point of recursion is defined by *BasicML*.
Formally, the model synchronizer *ConsR* can be expressed as follows:

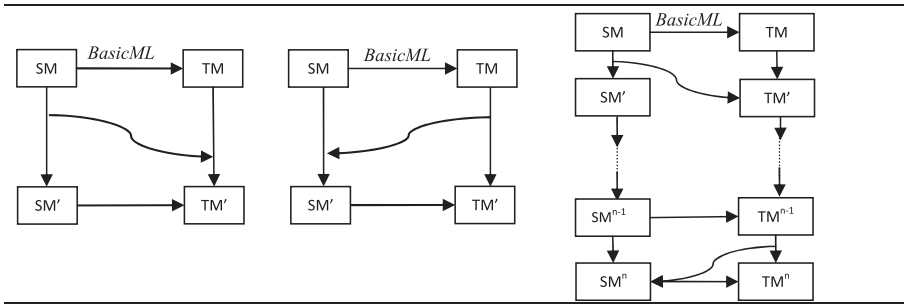
[SM, TM]
$ConsR ::= BasicML\ \langle\langle SM \times TM \rangle\rangle$
$\quad \mid FDML\ \langle\langle ConsR \times \mathbb{P}SM \rangle\rangle$
$\quad \mid BiDiML \langle\langle ConsR \times \mathbb{P}SM \times \mathbb{P}TM \rangle\rangle$
$\quad \mid BDML \langle\langle ConsR \times \mathbb{P}TM \rangle\rangle$

Different types of model synchronizer *ConsR* are shown in Figure 3. The directed arrows show the basic model transformation direction and the curved arrows show the change propagating ones.

It is possible to redefine *ConsR* to shift focus to consistency-preserving transformations between source and target models as follows:

Definition 3. (Consistency preserving). A model transformation is consistency-preserving if it transforms a source model in a way that its consistency with the target model is not violated. The free-type *ConsistentR* defines such transformations. It relates the elements of source model with their corresponding elements of target model and vice versa.

DOI: 10.1002/smr



(a) FDML (b) BDML (c) BiDiML

Figure 3. Model synchronizer *ConsR*.

$$ConsistentR ::= BaseMT\langle\langle ConsistentR\rangle\rangle$$
$$\mid UpdateTarget\langle\langle ConsistentR \times DirectDeltaTrans\rangle\rangle$$
$$\mid Sync\ \langle\langle ConsistentR \times DirectTransSource \times DirectTransTarget\rangle\rangle$$

where *DirectDeltaTrans* introduces a direct model transformation as a type of the endogenous model transformations [18]. It takes a (source or target) model and a delta model as inputs and returns an updated model in the same meta-model. *UpdateTarget* describes an update propagation transformation in a single direction while *Sync* emphasizes on bidirectional model synchronizers. The end point of the recursion is defined by *BaseMT*. Direct transformations of the source and target models are defined as follows:

$$DirectDeltaTrans :: = DirectTransSource\ \langle\langle TM \times \mathbb{P}\, TM\rangle\rangle$$
$$\mid DirectTransTarget\ \langle\langle SM \times \mathbb{P}\, SM\rangle\rangle$$

4.3. Delta model

Structurally, a model of changes can be presented by two main techniques: directed deltas and symmetric deltas [19] which are different in change representing. The direct delta organizes changes on a model as a sequence of delta operations, while the symmetric delta represents the consequent of the delta operation as the set difference between two compared versions. In this paper, the directed delta technique is used to express ongoing changes on a test model.

Definition 4. (Change). A change is considered as a transition from one subset of test model elements, named pre-state, to another subset in the same meta-model, named post-state. There are different parameters and dimensions to capture changes and system evolution in the related literature (e.g., [20]). In this paper, a customized set of these parameters including the subject of changes, type of changes, and order of changes are used. In our context, the changes arise from delta operations at a fine level of granularity.

Definition 5. (Delta operation). A delta operation is a direct model transformation for refactoring models in the same abstract syntax. Delta operations are generally divided into two main types: 'Add' and 'Delete'. Another type of delta operations, the operation 'Update', is considered as sequences of operations 'Add' and 'Delete' on the same element. Each delta operation has pre-condition and post-conditions to verify a model structure before and after its execution. In our framework, post-state elements of each delta operation have the same identifiers with a prime (') appended. Using pattern matching, pre-condition and post-condition patterns are recognized for each delta operation. Delta operations may be executed in batch, incremental, or change-driven manner [21]. In this paper, batch and incremental modes are desirable. The former performs all

DOI: 10.1002/smr

Table I. Customized delta operations for an LSM and related schemas.

TestMetaModel DeltaOp	State	Transition	Label
New	*AddStt*	*AddTransition*	*AddLabel*
Delete	*DelStt*	*DelTransition*	*DelLabel*
Update	*UpdateStt*	*DelTransition* ⨟ *AddTransition*	*DelLabel AddLabel*

LSM, labeled state machine.

delta operations where their pre-conditions are satisfied; the latter executes delta operations one by one in an incremental mode. In the two manners, transformations update a test model without the costly re-evaluation of unchanged elements. In a batch implementation, the pre-condition is the union of delta operation pre-conditions and the result model keeps the *union* of conflict-free post-conditions. It means that to apply a delta model to a test model, all pre-conditions of its delta operations should be satisfied, and after the execution of delta operations, all post-conditions should be established in the model. Formally, for delta operations $\mathrm{MTOp_i}$ that $i \geq 1$, $\mathrm{DeltaPreCon} = \cup_{i=1}^{n} \mathrm{Pre\text{-}condition}\ (\mathrm{MTOp_i})$ and $\mathrm{DeltaPostCon} = \cup_{i=1}^{n} \mathrm{Post\text{-}condition}\ (\mathrm{MTOp_i})$.

Also, *unsafe* delta operations are introduced as operations which their pre-conditions are not satisfied in a model, so they cannot execute in any order. A delta operation consists of a pre-condition, a post-condition, and a list of variables. According to the refinement calculus in Z, semantically, a delta operation can be defined by *DeltaOperation*. It describes an operation that begins in a state satisfying *pre-condition* and ends in a state satisfying *post-condition*. Only those variables listed in *chgSubjectModelElements* are affected. Two steps for a refinement of an abstract test case are discussed in Section 6.

DeltaOperation = chgSubjectModelElements: [pre-conditionElem, post-conditionElem]

Definition 5.1. (Additional delta operation). An additional delta operation is an operator AddMT: $TestModel \rightarrow TestModel'$ which computes for a test model $TestModel < Stt, Eve, Guardcon, EffectT, Trans>$, an updated $TestModel'$ which $Stt \subset Stt'$, $Eve \subset Eve'$, $Guardcon \subset Guardcon'$, $EffectT \subset EffectT'$ and $Trans \subset Trans'$, or in shortly $TotalElem \subset TotalElem'$. This operation indicates a graph pattern with new matches in the post-state. In order to provide a generic solution to query-specific information in an updated test model, a tag '*New*' is appended to each new element using the relation *TagRelation*. *TagRelation* is extendable to subgraph matching when a complex pattern is identified.

Definition 5.2. (Removal delta operation). A removal delta operation is defined in the form DelMT: $TestModel \rightarrow TestModel'$ which computes for a test model $TestModel < Stt, Eve, Guardcon, EffectT, Trans>$, an updated $TestModel'$ which $Stt' \subset Stt$, $Eve' \subset Eve$, $Guardcon' \subset Guardcon$, $EffectT' \subset EffectT$ and $Trans' \subset Trans$, or in shortly $TotalElem' \subset TotalElem$. A tag '*Delete*' is attached to removable elements using the relation *TagRelation*.

Finally, each updated element is specified by a tag '*Update*'. According to our test meta-model, different delta operations for test model refactoring can be defined. Table I shows various fine-grained delta operations and their corresponding schemas. It is possible to combine different schemas to produce a complex pattern of changes.

For example, consider two delta operations 'AddMT' and 'DelMT' on the state set of the test model *TestModel*. To add an input state s? to the set of states of the test model, the schema *AddStt* is used. Its pre-conditions force that the new state s? should not exist in *TestModel* and the post-condition adds the state s? to the set of states. The state-removal *DelStt* deletes an existing state s? from the set of states and restricts the domain of the functions *value* and *type*. A complete list of delta operation schemas is provided in Appendix B.

DOI: 10.1002/smr

AddStt

$\Delta TestModel$
$\Delta DeltaTestModel$
$s?: State$
$v?: Value$
$attribute?: attrID$

$s? \notin Stt$
$Stt' = Stt \cup \{s?\}$
$value' = value \cup \{s? \mapsto \{attribute? \mapsto v?\}\}$
$TagRelation' = TagRelation \cup \{s? \mapsto New\}$

DelStt

$\Delta TestModel$
$s?: State$
$type?: typeID$
$attribute?: attrID$

$s? \in Stt$
$Stt' = Stt \setminus \{s?\}$
$FromSource' = FromSource \rhd \{s?\}$
$ToTarget' = ToTarget \rhd \{s?\}$
$value' = \{s?\} \lhd value$
$type' = \{s?\} \lhd type$
$TagRelation' = TagRelation \cup \{s? \mapsto Delete\}$

An updating model transformation can be directly defined by a schema like *UpdateStt* or by composing of two removal and additional operation schemas. The schema *UpdateStt* implies that for an updated state *s?*, its type and attribute set should be updated and a tag *Update* should be attached to distinguish the undergoing change on the element.

UpdateStt

$\Delta TestModel$
$s?: State$
$type?: typeID$
$v?: Value$
$attribute?: attrID$

$s? \in Stt$
$value' = value \oplus \{s? \mapsto \{attribute? \mapsto v?\}\}$
$type' = type \oplus \{s? \mapsto type?\}$
$TagRelation' = TagRelation \oplus \{s? \mapsto Update\}$

DOI: 10.1002/smr

A schema disjunction allows us to describe different behaviors of a system by merging the declaration parts of the corresponding schemas and disjoining their predicate parts. So, we are now ready to define a collection of delta operations as a model refactoring system for our framework. All manipulations on the state and transition set of a test model can be summarized as *ChangeState* and *ChangeTransition*, respectively. The changes on a transition include direct manipulations on the transition properties or on its label parameters:

$$ChangeState \triangleq AddStt \lor DelStt \lor UpdateStt$$

and,

$$\begin{aligned} &ChangeTransition \triangleq \\ &\quad AddTransition \\ &\quad \lor DelTransition \\ &\quad \lor UpdateTransition \\ &\quad \lor AddLabel \\ &\quad \lor UpdateLabel \\ &\quad \lor DelLabel \end{aligned}$$

So, test model refactoring in a batch manner and its pre-conditions are defined by *TestModelRefac* and *DeltaPreCon*, respectively. *DeltaPreCon* forces that to manipulate a model using a set of delta operations, the pre-condition of each delta operation should be satisfied.

$$TestModelRefac \triangleq ChangeTransition \lor ChangeState$$

$$DeltaPreCon \triangleq \text{pre } ChangeState \lor \text{pre } ChangeTransition$$

Or, to summarize:

$$DeltaPreCon \triangleq \text{pre } TestModelRefac$$

Definition 6. (Delta signature). Each delta signature is a four-tuple: a changed model element, a delta operation and before and after values of the changed element. It is clear that before and after values for added and deleted elements are respectively empty. The delta signatures for our case study are defined in Section 7.3. Formally,

$$DeltaSIG == TestMetaModel \times DeltaOp \times Value \times Value$$

In order to simplify, all delta operations are categorized by the free-type *DeltaOp* as distinct sets of direct model transformations that are semantically described in Definition 5.

$$DeltaOp ::= AddMT \mid DelMT \mid UpdateMT$$

4.4. Delta signature dependencies

The applicability of a delta operation as an endogenous transformation is denoted by its pre-condition satisfaction. Pre-conditions implicitly specify the dependency between delta operations. For example, when a delta operation 'AddMT' adds an event as a label of a transition of an LSM, its pre-condition implies that the corresponding transition has been added before, or a delta operation 'DelMT' can only be applied to remove a model element *e*, if that *e* already exists in the model. Another type of dependency is forced by system characteristic and developer decisions. We categorize dependencies between delta signatures into internal and external dependencies depending on the reason why the dependency holds.

Definition 7. (Internal dependency). Two delta signatures are internally dependent on each other if the pre-conditions of their delta operations are involved. We define two kinds of internal dependencies: *direct* and *indirect* dependencies. The *direct* dependency, called also *definition/use* dependency, implies that a delta signature sig_1 accesses the same element which is added by a delta signature sig_2. In other words, a relation *definition/use* holds pre-condition ($sig_1.op$) $\subset$ post-condition ($sig_2.op$). It is obvious that delta operations which add an element to a model are the definition part of this type of dependency, and delta operations which delete or update an element are the usage part.

DOI: 10.1002/smr

A delta signature sig_1 *indirectly* depends on delta signature sig_2, if the subject element of sig_1 is in a 'related to', for example, 'belongs to' or 'contained/container' relation with the subject element of sig_2. For instance, if sig_1 adds an argument to an event method that was added by sig_2, then sig_1 is said to be *indirectly* depended upon sig_2.

Definition 8. (External dependency). Some dependencies are external to a system delta model. Although this type of dependency is not a part of system changes, it impacts the order of them. Some ordering parameters like time, developer decisions, or arbitrary optimization methods can denote an order for executing the delta operations which do not have prior internal dependencies. This type of dependency is called the *external dependency*.

Finally, *independent* delta signatures are delta signatures which no restriction is applied on their execution order. An independent delta signature can be performed in parallel while the result of the parallel execution of modifications is equivalent to a serial execution.

Definition 9. (Delta model – abstract syntax). Syntactically, a delta model is defined by a set of three-tuple ($DeltaSIG \times dependency \times DeltaSIG$) where *DeltaSIG* is a delta signature and *dependency* is an established dependency between two delta signatures. According to Definitions 7 and 8, the relations between delta signatures are defined by the free-type *dependency* with four kinds of dependencies:

$$dependency ::= Direct \mid Indirect \mid Indep \mid Conflict$$

It is worth noting that (as it is expressed in *DeltaTestModel* predicate) *dependency* relations between *DeltaSIGs* are irreflexive, asymmetric, and transitive. In other words, for delta signatures x, y, and z and dependency relations a: (i) *not* $x\ a\ x$; (ii) if $x\ a\ y$ implies *not* $y\ a\ x$; and (iii) if $x\ a\ y$ and $y\ a\ z$ implies $x\ a\ z$.

Based on the mentioned descriptions, a delta model is defined by the schema *DeltaTestModel*. The schema predicate also implies that deleted and updated model elements should be the members of model elements *TotalElem* while added model elements should not be the members of *TotalElem*. All delta model elements are accessible from the set *TotalDeltaSig*.

DeltaTestModel

$$
\begin{aligned}
&TestModel \\
&TotalDeltaSig: \mathbb{P}\ DeltaSIG \\
&SIGSet: \mathbb{P}\ (DeltaSIG \times dependency \times DeltaSIG) \\
&\hline
&\forall x, y, z: DeltaSIG;\ a: dependency \mid (x, a, y) \in SIGSet \bullet \\
&\quad x\ .\ 2 \in \{DelMT\ ,\ UpdateMT\} \Rightarrow x\ .\ 1 \in TotalElem \\
&\quad \land\ x\ .\ 2 = AddMT \Rightarrow x\ .\ 1 \notin TotalElem \\
&\quad \land\ (x, a, x) \notin SIGSet \\
&\quad \land\ (y, a, x) \notin SIGSet \\
&\quad \land (y, a, z) \in SIGSet \Rightarrow (x, a, z) \in SIGSet \\
&TotalDeltaSig = \{\ x: DeltaSIG \mid \exists y: SIGSet \bullet x = y\ .\ 1 \lor x = y\ .\ 3\ \}
\end{aligned}
$$

Definition 10. (Delta model – semantic). Semantically, a delta model consists of three kinds of information: delta operations, changed elements, and their dependencies. Changed elements are divided into two categories: additional elements *AdditionalDeltaElement* and removable elements *RemovableDeltaElement*. Updated elements are interpreted as removed and added elements. Additional elements are a set of elements which are disjoint with the pre-state model elements.

DOI: 10.1002/smr

AdditionalDeltaElement

$$\Delta TestModel$$
$$addTotalElem: \mathbb{P}\ TestMetaModel$$
$$addTotalElem = \{\ e: TestMetaModel \mid e \in TotalElem' \wedge e \notin TotalElem\ \}$$

Any deleted model element should be a member of model elements before applying a removal change as follows:

RemovableDeltaElement

$$\Delta TestModel$$
$$remTotalElem: \mathbb{P}\ TestMetaModel$$
$$remTotalElem = \{\ e: TestMetaModel \mid e \notin TotalElem' \wedge e \in TotalElem\ \}$$

Finally, elements that do not appear in an *optimized* delta model are unchanged elements. An *optimized* delta model is a minimized form of a delta model which its redundancies and conflicts are removed. More details are provided in Section 4.5.

4.5. Delta model optimization

We propose to optimize a delta model before propagating it to the testing phase. It leads to improving the efficiency of RTST because only test cases which traverse the optimized delta model are investigated. For example, if an element is added by a delta operation and deleted by another, no test requirement will be added to cover these changes.

Different behaviors for direct delta-dependency management are summarized in Table II. To manage indirect dependencies, all changes on a container should be performed earlier than changes on the corresponding contained. For example, the occurrence of a couple of delta operations AddMT with a direct dependency is reported as a conflict for the second additional operation, while for indirect dependency, delta signatures is managed by applying the container at first. Another conflict arises when the tuple (UpdateMT, AddMT) is applied to a specific model element. To manage such conflicts, we should remove the unsafe operations (the operations with an unsafe access) from a delta model, for example, the operation AddMT in this example.

Briefly, delta model optimization management keeps the order of safe delta operations, removes unsafe delta operations, performs the changes on a container earlier, and does not combine changes (in order to keep precision in RTSTs).

4.6. A formal terminology for platform-independent testing

In this section, the definitions of relevant concepts for platform-independent testing are introduced.

Definition 11. (Atomic testing pattern). For a given system behavior in the form of LSM, an atomic testing pattern is a five-tuple ($State \times Event \times GuardCon \times Effect \times Transition$) where its elements are LSM meta-model elements, and its behavior includes visiting the *State*, activating one *Event,* satisfying a *GuardCon* value, effecting on the system by an *Effect*, and traversing one of the outgoing *Transition*.

Table II. Optimization management of direct dependencies.

op1/op2	AddMT	UpdateMT	DelMT
AddMT	Conflict for op2	Sequential (def/use)	None
UpdateMT	Conflict for op1	Sequential	DelMT
DelMT	AddMT	Conflict for op2	Conflict for op2

DOI: 10.1002/smr

The formal definition of an atomic testing pattern is described in the following axiomatic definition. To keep the connectivity between the elements of an atomic testing pattern *tc* with the transition element *tc.5*, its state should be defined as the source state of the transition *tc.5* and its event, guard condition, and effect as the labels of *tc.5*.

$$
\begin{array}{|l}
AtomicTestCasePattern: \mathbb{P}\,(State \times Event \times GuardCon \times Effect \times Transition) \\
\hline
\forall tc: AtomicTestCasePattern \\
\quad \bullet\ \exists m: TestModel \\
\qquad \bullet\ tc\,.\,1 = m\,.\,FromSource\ tc\,.\,5 \\
\qquad\ \ \wedge\ tc\,.\,2 = m\,.\,labelE\ tc\,.\,5 \\
\qquad\ \ \wedge\ tc\,.\,3 = m\,.\,labelE\ tc\,.\,5 \\
\qquad\ \ \wedge\ tc\,.\,4 = m\,.\,labelE\ tc\,.\,5
\end{array}
$$

Definition 12. (Abstract test case pattern). For a given LSM, an abstract test case (ATC) pattern is a sequence of atomic testing patterns that made a path pattern. An ATC denotes a test requirement that should be traversed according to a selected coverage criterion.

$$ATC == \text{seq}\ AtomicTestCasePattern$$

According to an ATC definition, a concrete test case (CTC) is a path pattern which proper values are assigned to its variables. The valid input space for a concrete test case is defined based on the input spaces of its variables; in our context, a valid input space for an ATC is a subset of the input variables which satisfy the corresponding guard conditions. It can be derived directly from the formal specification of an ATC.

Definition 13. (Transition trace). The transition trace *traceTCTran* of an ATC *t* is defined recursively as the sequence of traversed transitions in its path, and the transition trace of a set of ATCs is the union of their transition traces. When the transition trace of a set of ATCs covers all transitions in a test model, then *all-transition coverage criterion* has been satisfied. We called such set of test requirements the *transition surjective test suite.* The minimum set of surjective test suite provides an optimized suite for related coverage criteria. To examine the effect of a change on a test suite, we define the inverse trace *inverseTrace* of a transition *t* as all ATCs which traverse the transition *t*.

$$
\begin{array}{|l}
\textit{TraceTCTransition} \\
\hline
TestModel \\
traceTCTran: \mathbb{N} \times ATC \times \mathbb{N} \rightarrow\!\!\!\!\!\!\rightarrow \text{seq}\ Transition \\
inverseTrace: Transition \rightarrow\!\!\!\!\!\!\rightarrow \mathbb{P}\ ATC \\
\hline
\text{dom}\ inverseTrace \subseteq Trans \\
\forall t: ATC \mid t \neq \langle\rangle \\
\quad \bullet\ \exists i, j: 1\ ..\ \#t \mid i < j \\
\qquad \bullet\ traceTCTran\ (1, t, 1) = \langle (t\ 1)\,.\,5 \rangle \\
\qquad\ \ \wedge\ traceTCTran\ (i, t, j) = traceTCTran\ (i, t, (j - 1)) \frown \langle (t\ j)\,.\,5 \rangle \\
\qquad\ \ \wedge\ (\forall trans: Trans \\
\qquad\qquad \bullet\ inverseTrace\ trans = \{\ t: ATC \mid \exists i: \mathbb{N} \bullet (t\ i)\,.\,5 = trans\ \})
\end{array}
$$

DOI: 10.1002/smr

Definition 14. (State trace). The state trace *traceTCState* of an ATC is the sequence of traversed states in its path, and the state trace of a set of ATCs is the union of their state traces. When the state trace of a set of ATCs covers all states in a test model then *all-state coverage criterion* has been satisfied. All issues of transition trace are valid for the state trace. The formal definition of *traceTCState* is obtained by a simple substitution in *TraceTCTransition.*

Definition 15. (Test case semantic). The semantic of a test case is defined based on a set of inputs and outputs as a couple <Inputs, Outputs> [22]. Inputs include *pre-conditions* and *steps.* The former assures that the test case can be executed in the current state of the system; in our approach, it corresponds to the guard conditions. The latter defines a sequence of actions that should be executed by either testers or automatic testing procedure, which in our context is equivalent to the sequence of events in a path pattern.

The output parameters include *expected results* and *post-conditions*, which are dependent on concrete test case execution. The former specifies expected results of performing a test case and the latter specifies the post-state of a model after performing test case steps. Post-conditions in our formal framework can be modeled by transition effects.

5. REGRESSION TESTING BASED ON THE DELTA MODEL

To verify the correctness of a modified design model, the corresponding delta model should be propagated from the system specification to testing framework and a consistent test suite should be selected. In Figure 4, a microscopic view of delta-based regression testing is shown. Integrating the PIM and the PIT in the approach, offers efficient traceability to derive suitable regression test suite and to keep consistency between software design and testing phase. It provides a typical infrastructure solution for expressing regression testing in the model-driven fashion. SM and SM′ denote that other behavioral models can be used in the approach if they are converted to a suitable model for software testing.

Four considerable issues are induced from Figure 4 including debugging traceability links, maintenance traceability links, change propagating rules, and distinct categories of regression test suites, which are defined as follows:

Definition 16. (Debugging traceability link). A debugging traceability link is created between a test model and test requirements at the platform-independent level, in such a way that each test model element can be a target for a test requirement in a coverage criterion. Debugging links encourage the process of locating faults using backward trace links. This type of link aims at solving debugging problems in a more precise way, for example, finding elements that could have caused an observed fault and finding model elements that are dependent upon an erroneous element. Formally, a traceability link *TraceLink* is defined as a mapping between a set of predicates *TestRequirementPred* and the test meta-model *TestMetaModel* as follows:

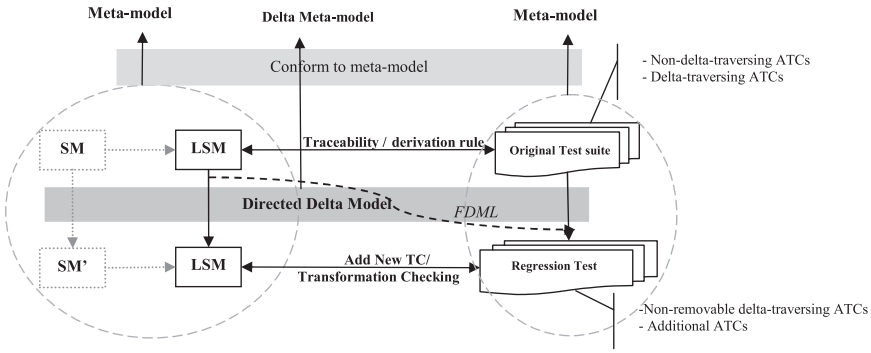


Figure 4. A microscopic view of the framework.

DOI: 10.1002/smr

[*TestRequirementPred*]

|*TraceLink: TestMetaModel* ↔ *TestRequirementPred*

Definition 17. (Maintenance traceability link). Maintenance traceability links consist of forward traceable links that re-established in the maintenance phase to support *traceable maintenance management*; a subset of these links includes a new set of traceability links that are created for newly added elements. Using maintenance traceability links, it is possible to determine the test requirements which need to be rerun, removed, and added from/to the predicate set *TestRequirementPred*. Maintenance traceability links improve efficiency in terms of required time for change impact analysis and regression test case selection. These links can be queried to derive applicable information for regression testing, for example, which model elements will be affected by a change? Will a particular model element be affected by a change? Which model elements could affect 'this' model element?

Definition 18. (Change propagating rule). Change propagating rules are used to propagate the delta model from the PIM to the PIT in order to preserve consistency in DbRT. Each rule takes a delta model and two consistent models including a behavioral model and a test requirement model in a uniform format (e.g., XMI format) and manipulates the latter such that consistency is held between them. For example, when an element is removed from a test model by a delta operation DelMT, a rule should remove all ATCs that traverse the element. So after the rule execution, ATCs which traverse the deleted element do not exist in the test case pool (to keep safe access). In Listing 1, an informal specification of the change propagating rules is defined. The readers may refer to Appendix C for the formal definition of change propagating rules in both incremental and batch modes.

In all rules, the test model and test requirements are known as *TestFrameworkModels*. A rule of type *RuleToRemoveATC* in its pre-condition distinguishes ATCs that traverse a removed element and deletes them from the test pool as execution result, a rule of type *RuleToAddATC* adds new test requirements for newly created model elements (it is clear that these model elements are not traversed by any ATC), and finally a rule of type *RuleToUpdateATC* detects ATCs that traverse an updated element to rerun them.

RuleToRemoveATC *(Delta Model, TestFrameworkModels):*
{
  Pre-condition:
    Existence of test cases which traverse deleted delta elements
  Post-condition:
    Non-existence of the test cases in the test suite
}
RuleToAddATC *(Delta Model, TestFrameworkModels):*
{
  Pre-condition:
    Non-existence of test cases which traverse added delta elements
  Post-condition:
    Existence of the test cases in the test suite
}
RuleToUpdateATC *(Delta Model, TestFrameworkModels):*
{
  Pre-condition:
    Existence of test cases which traverse updated delta elements
  Post-condition:
    Updated test cases
}

Listing 1. Change propagating rules

DOI: 10.1002/smr

5.1. Classification of delta-based regression test suites

To provide the DbRTST that covers all modified parts of a model, known as the safe technique, we classify initial ATCs into two categories: delta-traversing and non-delta-traversing ATCs. The latter, called also *applicable* ATCs, do not traverse any delta model elements in their trace paths. This category is reusable for the new version of a model, but it is unnecessary to be rerun on the updated model; however, they are valid and reusable for regression testing of the future versions. The former visit at least one of the delta model elements in their trace paths and are divided into two distinct subsets of ATCs: *removable* and *updatable* ATCs.

To categorize regression test cases, we need two parameters: *totalTrace* and *AllTest*. The former denotes all model elements that are traced by each ATC, that is, the states, events, guard conditions, effects, and transitions. The latter defines the original test suite for a behavioral model. It is clear that new ATCs are not a subset of *AllTest*.

$$
\begin{array}{l}
TestModel \\
AllTest: \mathbb{P}\ ATC \\
totalTrace: ATC \rightarrow \mathbb{P}\ TestMetaModel \\
\hline
\forall tc: ATC \mid tc \neq \langle\rangle \\
\quad \bullet\ \exists i: 1 \mathinner{..} \#tc \\
\quad\quad \bullet\ totalTrace\ tc \\
\quad\quad\quad = \{\ e: TestMetaModel \mid (tc\ i).1 = e \lor (tc\ i).2 = e \lor (tc\ i).3 = e \lor (tc\ i).4 = e \lor (tc\ i).5 = e\ \} \\
\quad\quad \land\ AllTest \\
\quad\quad\quad = \{\ s: ATC \mid (s\ i).1 \in Stt \land (s\ i).2 \in Event \land (s\ i).3 \in Guardcon \land (s\ i).4 \in EffectT \land (s\ i). \\
\quad\quad\quad\quad 5 \in Trans\ \}
\end{array}
$$

According to these definitions, delta-traversing and non-delta-traversing test cases can be defined by the following schemas. The predicate parts of the schemas enforce that the elements of a delta-traversing ATC appear in delta signatures whose *DeltaOp* is *DelMT* or *UpdateMT* while the intersection of the elements of non-delta-traversing ATCs and delta elements is empty.

$$
\begin{array}{l}
DeltaTraversalATC \\
\hline
DeltaTestModel \\
deltats: \mathbb{P}\ TestMetaModel \\
traverse: \mathbb{P}\ ATC \\
\hline
deltats \\
\quad = \{\ e: TestMetaModel \\
\quad\quad \mid \exists s: TotalDeltaSig \bullet s.1 = e \land s.2 \in \{DelMT, UpdateMT\}\ \} \\
\forall s: ATC \mid s \in traverse \bullet s \in AllTest \land totalTrace\ s \cap deltats \neq \varnothing \\
\hline
\end{array}
$$

$$
\begin{array}{l}
NonDeltaTraversalATC \\
\hline
DeltaTestModel \\
deltats: \mathbb{P}\ TestMetaModel \\
nontraverse: \mathbb{P}\ ATC \\
\hline
deltats \\
\quad = \{\ e: TestMetaModel \\
\quad\quad \mid \exists s: TotalDeltaSig \bullet s.1 = e\ \} \\
\forall s: ATC \mid s \in nontraverse \bullet s \in AllTest \land totalTrace\ s \cap deltats = \varnothing \\
\hline
\end{array}
$$

DOI: 10.1002/smr

Definition 19. (Updatable ATC). An *updatable* ATC is a delta-traversing test case that traverses at least one delta signature which its delta operation is 'Update'. *Updatable* ATCs should be re-executed in order to verify the correctness of an updated test model. Formally

UpdatableATC

$$
\begin{array}{l}
DeltaTestModel \\
deltats: \mathbb{P}\ TestMetaModel \\
updatable: \mathbb{P}\ ATC \\
\hline
updatable \subseteq AllTest \\
deltats \\
\quad = \{\ e: TestMetaModel \mid \exists s: TotalDeltaSig \bullet s.1 = e \land s.2 = UpdateMT\ \} \\
\forall s: ATC \mid s \in updatable \bullet s \in AllTest \land totalTrace\ s \cap deltats \neq \emptyset
\end{array}
$$

Definition 20. (Removable ATCs). A *removable* ATC is a delta-traversing test case that traverses at least one delta signature which its delta operation is 'Delete'. These ATCs are no longer valid to be rerun on the updated test model. Formally

RemovableATC

$$
\begin{array}{l}
DeltaTestModel \\
deltats: \mathbb{P}\ TestMetaModel \\
removable: \mathbb{P}\ ATC \\
\hline
deltats = \{\ e: TestMetaModel \mid \exists s: TotalDeltaSig \bullet s.1 = e \land s.2 = DelMT\ \} \\
\forall s: ATC \mid s \in removable \bullet s \in AllTest \land totalTrace\ s \cap deltats \neq \emptyset
\end{array}
$$

According to the defined categories, delta-traversing test cases can be redefined as

$$DeltaTraversalATC \triangleq UpdatableATC \lor RemovableATC$$

The remaining class consists of test cases that should be generated for testing the new functionalities of a model. This category is defined by the following schema:

NewATC

$$
\begin{array}{l}
DeltaTestModel \\
deltats: \mathbb{P}\ TestMetaModel \\
newtest: \mathbb{P}\ ATC \\
\hline
deltats = \{\ e: TestMetaModel \mid \exists s: TotalDeltaSig \bullet s.1 = e \land s.2 = AddMT\ \} \\
newtest = \{\ s: ATC \mid s \notin AllTest \land totalTrace\ s \cap deltats \neq \emptyset\ \}
\end{array}
$$

Finally, a safe regression test suite consists of updatable and new test cases:

$$RegressionTestCases \triangleq UpdatableATC \lor NewATC$$

5.2. Delta-based coverage criteria

Software testing criteria have important roles in the testing process [23]. We define a new category of coverage criteria, called *delta-based coverage criteria*, that are used during the maintenance phase of software development. These criteria are used for the following purposes:

DOI: 10.1002/smr

- Defining rules to determine whether adequate regression testing has been performed and can it be terminated and,
- Measuring coverage degree when a specific criterion is associated with each test suite.

Definition 21. (Delta-based coverage criteria). A delta-based coverage criterion *CovCriteria* is defined as a mapping between a test model and a set of predicates *TestRequirementPred* on the test model. Coverage criteria satisfaction is defined by a total function from *CovCriteria* to a subset of abstract test cases. It ensures that there is at least one abstract test case to satisfy each predicate of *TestRequirementPred*. Formally, it is defined as follows:

$$
\begin{array}{|l}
CovCriteria: TestModel \rightarrow \mathbb{P}\ TestRequirementPred \\
CovCriteriaSatisfication: CovCriteria \twoheadrightarrow \mathbb{P}\ ATC \\
\hline
\forall pred: TestRequirementPred \\
\quad \bullet\ \{pred\} \in \text{ran}\ CovCriteria \\
\quad\quad \Leftrightarrow (\exists\ t: ATC \bullet\ \{t\} \in \text{ran}\ CovCriteriaSatisfication)
\end{array}
$$

Two classes of the delta-based coverage criteria are defined in our testing framework including *strategy-based* and ad hoc. The former is inspired by the subsumption hierarchy for transition-based coverage criteria [24] that are customized for delta-based regression testing. The latter is realized using a query language to directly examine the coverage of special parts of a changed model in regression testing. It results in deriving significant abstract test cases in response to regression testing queries.

$CovCriteriaType ::= StrategyBasedCC \mid AdHocCC$

5.2.1. Strategy-based coverage criteria. Some of strategy-based coverage criteria are defined by distinct *Rules* in this section. The formal definitions of all rules are not described in the paper due to the similarity of the patterns. In all *Rules*, *CCSet* denotes a subset of ATCs that satisfy the corresponding coverage criterion.

Rule 1. Delta-state coverage. A regression test suite satisfies delta-state coverage criterion if it visits all added and updated states in a delta model. The schema *AllStateCoverage* formally describes this rule.

AllStateCoverage

$$
\begin{array}{|l}
DeltaTestModel \\
\hline
\forall e: State \mid \exists s: TotalDeltaSig \bullet\ s.1 = e \wedge s.2 \in \{AddMT, UpdateMT\} \\
\quad \bullet\ \exists t: AtomicTestCasePattern \bullet\ t.1 = e \wedge \langle t \rangle \in CCSet \\
\hline
\end{array}
$$

Rule 2. Delta-transition coverage. A regression test suite satisfies the delta-transition coverage criterion if it visits all (non-removed) modified transitions sequences of length zero and one in a delta model. Because a state is considered as a transition of length zero, delta-transition coverage rule subsumes delta-states coverage rule. It means that the coverage of all delta-transitions of length zero is equivalent to the coverage of all delta-states and is formalized by a substitution in the schema *AllStateCoverage*. The delta-transition coverage rule can be extended by changing the transition sequence lengths. For example, a delta-*k*-transition coverage criterion is satisfied if all modified transitions sequences up to length *k* in a delta model are traversed. In Figure 5, the hierarchical relation between different delta-based coverage criteria is shown. Every coverage rule in an upper level satisfies the lower one

DOI: 10.1002/smr

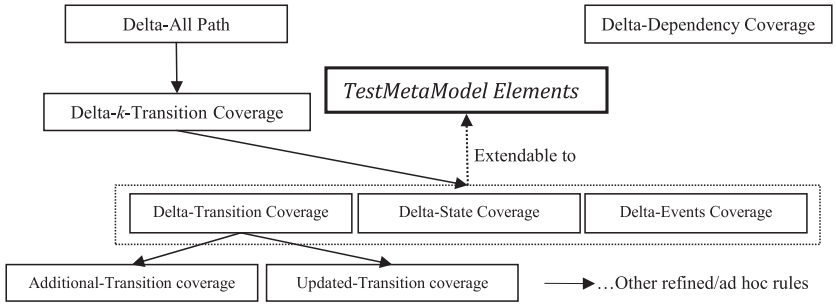


Figure 5. Simplified delta-base coverage criteria subsumption.

(delta-transition coverage rules subsume all rules in the form of '*delta-TestMetaModel elements*', but it is shown in a same level to focus on the position of the TestMetaModel elements in the delta-based coverage rules).

AllTransitionCoverage

DeltaTestModel

$$\forall e: Transition \mid \exists s: TotalDeltaSig \bullet s.1 = e \land s.2 \in \{AddMT, UpdateMT\} \bullet \exists t: AtomicTestCasePattern \bullet t.1 = e \land \langle t \rangle \in CCSet$$

It is possible to extend the delta-based coverage criteria for different TestMetaModel elements. Another strategy-based coverage criterion is defined based on the dependencies between delta signatures.

Rule 3. Delta-dependency coverage. A regression test suite satisfies the delta-dependency coverage criterion of a type *x: dependency* if it visits all changed elements that dependencies between their delta signatures are of the type *x*. It is clear that direct and indirect dependencies are the most important type for this coverage rule. For example, a rule can define as coverage all delta signatures which have direct dependencies. This rule provides a direct facility to follow and analyze impacts of a change to ensure the adequacy of regression testing.

When a delta model includes only one type of delta operations or a specific type of delta operations is required, refined coverage criteria can be used. We introduce coverage criterion refinement as a technique to transform a coverage criterion to a more detailed rule. It provides a fast handle to examine test requirement adequacy based on the refined rules. For example, when a delta model includes only additional delta operations, just new elements should be targeted for generating new test cases to keep coverage criteria satisfaction valid. To do so, each rule can be divided into two sub-criteria for additional and updated model elements. For instance, the reduced delta-state coverage rules are defined by the schemas *AllAddStateCoverage* and *AllUpdateStateCoverage*. Other refined rules can be defined in a similar pattern.

Refined Rule 1-1. Additional-state coverage criterion. A regression test suite satisfies the additional-state coverage criterion if it visits all newly added states in *TestModel′*. It introduces a reduced set of the delta-state coverage criteria.

AllAddStateCoverage

DeltaTestModel

$$\forall e: State \mid \exists s: TotalDeltaSig \bullet s.1 = e \land s.2 = AddMT \bullet \exists t: AtomicTestCasePattern \bullet t.1 = e \land \langle t \rangle \in CCSet$$

DOI: 10.1002/smr

Refined Rule 1-2. Updated-state coverage criterion. A regression test suite satisfies the updated-state coverage criterion if it visits all updated states in *TestModel′*.

AllUpdateStateCoverage

$$DeltaTestModel$$

$$\forall e: Stt \mid \exists s: TotalDeltaSig \bullet s.1 = e \land s.2 = UpdateMT \bullet \exists t: AtomicTestCasePattern \bullet t.1 = e \land \langle t \rangle \in CCSet$$

5.3. Ad hoc coverage criteria

Ideally, a query language allows users to formulate their queries in a semantically unambiguous way. In addition to the strategy-based coverage criteria, we define ad hoc coverage criteria that their semantics are specified by a precise formal syntax. They are used to retrieve information that matches a detailed set of criteria to cover specific changed/unchanged elements or delta dependencies in a test model (e.g., coverage of states which updated in a new model or coverage of more than three changed events). The use of ad hoc coverage criteria can improve the quality of regression testing by providing meaningful test requirements in response to flexible queries. Complex queries can provide a narrow view aimed at covering specific parts in an updated model and extracting effective ATCs.

Definition 22. (Delta-based query pattern). A delta-based query pattern (DQP) is built on the meta-model *TestMetaModel* and the change tag *ChgTag* to investigate coverage of different elements that are changed by delta operations. An DQP is described by the recursive free-type *DQP*. The *DQP* language enables testers to define significant queries to acquire adequate coverage of distinct test suites. Using *ChgTag* in the definition of DQP gives us a great opportunity to leverage synergies between testing and model transformation tools that work based on the labeled graph morphism. The DQP is a declarative query language because users do not need to define how the query is processed but they only focus on properties that a result should have.

$$DQP ::= Traverse \langle\langle \mathbb{N} \times State \times Event \times GuardCon \times Transition \times ChgTag \rangle\rangle \mid atomQuery \langle\langle EXP \times DQP \rangle\rangle \mid compQuery \langle\langle DQP \times EXP \times DQP \rangle\rangle \mid QueryDep \langle\langle DQP \times dependency \times DQP \rangle\rangle$$

$$EXP ::= AND \langle\langle DQP \times DQP \rangle\rangle \mid UNION \langle\langle DQP \times DQP \rangle\rangle \mid NOT \langle\langle DQP \rangle\rangle \mid DQP\text{^}+$$

The end point of recursion is defined by the function *Traverse* to pass a tagged model element $\mathbb{N} \geq 1$ times. *atomQuery* defines negative queries and transitive closures DQP^+. A compound query *compQuery* enables developer to specify more complex queries by combining atomic queries using logical operation *AND* and *UNION*. An DQP can be extended to derive more meaningful test requirements in critical system testing which is beyond the scope of this paper. In complex queries a subgraph or a combined pattern in a delta model can be investigated. To compare the coverage of different ad hoc coverage rules, we introduce the metric *coverability* in Section 7.3.1.

5.4. Delta-based prioritization techniques

Test case prioritization techniques arrange test case execution in an order which specific performance goals, for example, rate of fault detection, are met earlier [9]. Formally, a prioritization function *PrFunc,* suitable for regression testing, weights and schedules different sets of ATC according to the performance parameters related to system modifications, for example, prioritizing ATCs which traverse a longer path or more changes.

$$PrFunc: \mathbb{P}\, ATC \rightarrow \mathbb{R}$$

$$\exists\ t: \mathrm{dom}\ PrFunc \bullet \forall s: \mathrm{dom}\ PrFunc \bullet PrFunc\ t \geqslant PrFunc\ s$$

DOI: 10.1002/smr

In delta-based regression testing, one of the most important priority functions is the frequency of occurrence of delta elements in an ATC. It means that among regression test suites, ATCs that meet more delta elements in their traces need to be executed with a higher priority. It is advisable that each prioritization technique should be better than random prioritization and no-prioritization techniques.

5.5. *Fault modeling*

It is difficult to evolve systems in a fault-free manner, but using fault models which encompass more possibility faults in a model, it is possible to reduce the complexity of generating tests for all possible faults. A fault model groups different types of descriptions, for example, behavioral, functional, and structural concepts that are being used in a model. A more concrete specification for a PIT according to a fault model [*FM*], as a set of fault patterns, can be defined as follows:

$$| \; ConceretTesting: TestModel \times TestRequirementPred \times FM \rightarrow \mathbb{P}\, ATC$$

It declares that using a fault model leads to identify the subsets of ATCs which probably detect more faults in a system model. We derive four fault patterns for model-driven regression testing, which can be prevented by automatic pattern matching tools during the model evolution process.

Pattern 1. Unsafe access faults: this fault pattern targets the definition and usage of model elements in order to prevent access to model elements before their definition. Additional and removal delta operations in their pre-conditions check the existence of a required model element. Also in our incremental regression testing, the dependency relations between different delta signature in Definitions 9 and 10 recheck such faults in a wide range of changes.

Pattern 2. Redundancy faults: this type of fault is caused by manipulating models under incorrect conditions which leads to add unnecessary/redundant model elements. We establish the model maintenance phase based on well-defined delta operations which enforce strict pre-condition and post-condition to keep the consistency of a model structure. Each additional delta operation in its pre-condition checks the existence of a model element, so it prevents redundancies.

Pattern 3. Structural faults: a structural fault means that incorrect types of elements are added to a model. Enforcing the formal meta-model syntax restricts the domain of model elements in a test model.

Pattern 4. Transformation rule faults: some faults may arise from the model transformations also, for example, missing elements from pattern matching. A complete list of these faults is defined by Darabos *et al.* [25].

Fault coverage ability can be computed for a test pattern to reflect its effectiveness and to measure the test quality. Our approach to identify faulty elements and measure test quality in a behavioral model is based on test spectra inspired from code coverage. It is possible to produce a ranking of potential failure-inducing elements based on the suspiciousness score [26] as a technique that is widely used in comparative studies of fault localization techniques. The research question is how is it possible to use delta-based regression testing in ranking potential failure-inducing affecting edits at the design level?

Given two versions of a test model, TM and TM′, and a regression test suite *RegressionTestCases*, the DbRTST uses a set of atomic delta signatures and selects affected test cases *afTC*. For each affected test, we identify a set of affecting changes as a subset of changes that may cause test failures. Then for each atomic change *d* in a delta model, we denote a set of affected test cases *afTC d* as a subset of all test cases that exercise *d*. Formally,

$$afTC: DeltaOp \nrightarrow \mathbb{P}\, ATC$$

$$\forall\, d{:}DeltaOp \bullet afTC\, d = \{t : RegressionTestCases \mid \exists\, s{:}TotalDeltaSig \bullet s.2 = d \wedge totalTrace\, t \cap \{s.1\} \neq \emptyset\}$$

DOI: 10.1002/smr

The spectrum-based fault localization techniques to rank affecting changes, assign higher suspiciousness scores to model elements executed by failing tests than elements executed by passed tests. Obviously, fault localization using the ranking technique reduces the analyst efforts in manually finding failure causes in a system.

6. AUTOMATIC REFINEMENT AND CONCRETE TESTING

Model refinement is an essential technique in MDD to refine abstract models to more concrete ones in order to automatically generate complete and precise software systems. This is comparable with the compilation of a high-level program into a lower level one. From this viewpoint, our framework involves two issues: explaining abstract models and applying the theory of stepwise refinement to automatically transform abstract specifications to concrete ones. Z specification patterns and existing validation and proof tools provide the best support for these aspects. The patterns written in Z can be used to automatically derive concrete test scripts in a specific language.

According to [27], two directions are defined for refinement: forward and backward simulations in which the latter is out of the scope of this paper. In forward simulation, a refinement rule *RR* converts an abstract state $State_A$ with an abstract operation Op_A to a concrete state $State_C$ with a concrete operation Op_C whose corresponding abstract satisfies the pre-condition of Op_A. The theory of refinement implies that the development steps using stepwise refinement rules (*RR*, *RR′* and …) should keep the pre-condition of abstract operations and also concrete operations produce results that are consistent with the abstract one. Formally,

$$\forall State_A; State_C \bullet \text{pre } Op_A \wedge RR \Rightarrow \text{pre } Op_C$$

$$\forall State_A; State_C; State_C' \bullet \text{pre } Op_A \wedge Op_C \wedge RR \Rightarrow \exists State_A' \bullet Op_A \wedge RR'$$

In our study, a concrete specification for an ATC based on a valid input space can be defined as an ATC whose domains of its variables are assigned to a range of values. The following axiomatic definitions show the concrete test cases in two steps of refinement. *CTC* uses the same range of values for ATC elements while *CTC1*, in a more concrete specification, determines the distinct input valid spaces include *valEv*, *valGC*, and *valEf* for different types of elements, for example, Events, Guard Conditions, and Effect, respectively. Also, the concrete definition detects faults at the design level by comparing the expected result with the actual one. *The rep* introduces a suitable type of report.

$$Report ::= FaultDetected \mid NoFaultDetected$$

$$CTC : \mathbb{P}(State \times Event \times GuardCon \times Effect \times Transition)$$
$$val : \mathbb{P} attinstanseset$$
$$\forall ctc : CTC \bullet \langle ctc \rangle \in ATC \wedge value\ ctc\ .\ 2 \in val \wedge value\ ctc\ .\ 3 \in val \wedge value\ ctc\ .\ 4 \in val$$

$$CTC1 : \mathbb{P}(State \times Event \times GuardCon \times Effect \times Transition)$$
$$Computed: CTC1 \nrightarrow Value$$
$$Expected? : Value$$
$$valEv, valGC, valEf : \mathbb{P} attinstanseset$$
$$rep! : Report$$
$$\forall ctc : CTC1 \bullet \langle ctc \rangle \in ATC \wedge value\ ctc\ .\ 2 \in valEv \wedge value\ ctc\ .\ 3 \in valGC \wedge value\ ctc\ .\ 4 \in valEf \wedge Computed\ ctc \neq Expected? \Rightarrow rep! = FaultDetected$$

A textual refined rule for change propagation based on traceability links in the DbRTST is provided in Section 7.2. At the PSM level, more refinement rules are provided for concrete model-driven

DOI: 10.1002/smr

regression testing. It is possible and would be useful to utilize refinement automation tools, for example, Fastest [28] for producing concrete test cases automatically.

7. PRACTICAL DISCUSSION AND EVALUATION

In this section, we demonstrate the practical issues and the application of model-driven regression testing by the elaboration of the motivating example described in Section 3 and three other case studies. We also demonstrate an example of change propagating rules using the VIATRA2 transformation language [29]. The analysis of the potential advantages of DbRT, the limitations, and our approaches to reduce them are discussed at the end of this section.

7.1. Implementation architecture

In modern modeling environments, for example, the Eclipse Modeling Framework, model changes can be propagated using incremental transformation mechanisms. In these mechanisms, changes are propagated from a source to the relative target model on-the-fly, without the costly re-transformation of unchanged parts of the target model [21]. Also, a live model transformation [21], as a novel incremental mechanism, propagates changes continuously to the target domains, when run-time change notifications are received. Model refactoring frameworks preserve the change history of models as an external document which are named 'delta model' in this work.

When we make any change in system behavioral models, for example, adding or removing an element, the change should be automatically applied to the test requirement space to keep consistency. At the same time, the test model is also updated to display the post-condition of the system. According to the framework properties, the implementation architecture of delta-based regression testing is shown in Figure 6.

The architecture includes three major components: a test model, a transformer engine that is dependent on an optimized delta model and transformation rules, and the original consistent test suites according to a selected coverage criterion. Obviously, a new consistent regression test suite is obtained by propagating the delta model to the original test suite. In the architecture, the transformation engine using consistent rules can implicitly realize a regression test selector component.

7.2. Implementing change propagating rules

As previously mentioned, there are different languages for bidirectional model synchronization. We now turn our attention to the application of VIATRA2 as a strong incremental transformation engine in the Eclipse Modeling Framework. Its transformation engine implements the runtime change management feature of the framework by focusing on 'synchronization'.

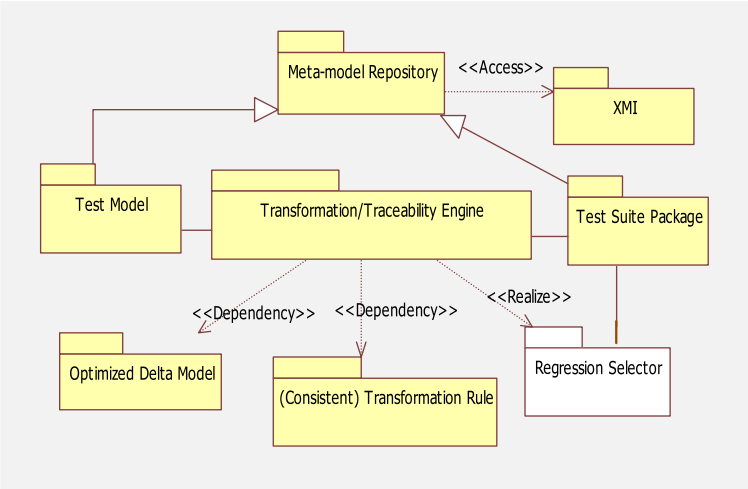


Figure 6. Implementation architecture of the framework.

DOI: 10.1002/smr

The VIATRA2 plug-in realizes the transformation between different DSMLs with incremental pattern matching. It consists of three main extensions that together form a model transformation environment to develop model-to-model and model-to-text transformations including (i) graph patterns to define pre-state/post-state of a model, (ii) graph transformation rules to transform models based on relation links between their meta-models, and (iii) abstract state machine rules can be used for the description of control structures. Using such automatic implemented synchronizers, potentially, improves performance by recording change operations and propagating them incrementally.

From a practice-oriented viewpoint, we need (i) source and target meta-model definitions which are imported in a transformation rule, (ii) the traceability links between source and target model elements based on the meta-models, and (iii) an incremental rule to propagate changes according to the pre-state and post-states pattern matching. Source and target meta-models are conformed to the formalism provided in Section 4. Listings 2 and 3 show a graph transformation rule that detects newly removed model states in a test model and the traceability relations between the design model and test requirement template in the concrete environment, respectively.

As a common technique in graph-based transformation approaches to indicate the conditions that should not appear in a post-condition, VIATRA2 uses negative application conditions as sub-patterns enclosed in a 'neg' instruction. In VIATRA2, only variable names are capitalized. As the name of the type starts with a capital letter, it has to be surrounded in single apostrophes to mark it as the name of a meta-model element.

The transformation rule *delState* is activated when a developer deletes a state of the source model in the modeling environment. This rule detects the deleted state from the model and propagates the changes to the related target ATC. We define the pre-condition pattern of *delState* consisting of a state element and a trace link to a target ATC which traverses the state, called TestReqModel.ATC.s; the post-condition pattern signifying the removed trace link between the source and target elements; and finally, an action sequence is responsible for deleting the target ATC and counting the number of removable ATCs.

```
import StateMachine.metamodel;
import TestCaseMM.metamodel;
machine allStateCC
{ rule main() =  seq
{
 call delState(SM, State, TestReqModel);
}
gtrule delState(in SM, inout State, inout TestReqModel) =
{
    precondition pattern SMState(SM,State)=   {
      smelement(SM);
      state(State);
      smelement.states(X2, SM, State);
      find stateMapping(State, s);   }

   postcondition neg find stateMapping(State, s)

   action
 {
   delete(s);
   update numberRemATC = numberRemATC + 1;
   println("Trigger  Action   is  activated  by  deleted  state  "+
   fqn‡(State) );
   println("The UML State " + State + " has been deleted from the
   model  " + SM);
  }
 }
  pattern stateMapping(State, s) =    {
      state(State);
     ' TestReqModel.ATC.s'(s);
      traces.stateTrace(Trace, State, s); }
}
```

Listing 2. Simplified rule to handle a deleted state

‡The global unique identifier of elements in VIATRA2

DOI: 10.1002/smr

```
import StatemMchine.metamodel;
import TestCaseMM.metamodel;

// TRACE META-MODEL
relation(netTrace, 'StateMachine', 'TestFramework');
relation(transitionTrace, transition, 'ATC.t');
relation(stateTrace, state, 'ATC.s');
relation(gcTrace, guardcondition, 'ATC.gc');
relation(effectTrace, effect, 'ATC.eff');
relation(eventTrace, event, 'ATC.e');
```

Listing 3. Traceability model

7.3. Experimental results and analysis

In this section, we conduct some experiments to evaluate the effectiveness of our proposed approach on three different case studies according to the pattern shown on the motivation example. Also, a metric is provided to analysis the quality of delta-based coverage criteria in terms of the ability of covering changes.

To follow the change history, we need to keep the delta model between different versions of test models and perform optimization management. An implementation can potentially achieve by recording delta operations and then propagating updates in the incremental or batch manner.

We follow the state machine which describes the behavior of a web service in Figure 2. The optimized directed delta model is shown in Listing 4 (a) and a minimal test suite according to all-state and all-transition coverage criteria are derived in Listing 4 (b), which are marked by U, N, and R in to show Updatable, New, and Removable test requirements and other requirements are applicable (in order to simplify, each ATC pattern is shown by its dominant elements, including states and/or transitions). According to the formal definitions, each delta signature is defined by a set of four-tuple $(TestMetaModel \times DeltaOp \times Value \times Value)$. After propagating the delta model, the retest-all strategy selects all valid ATCs to rerun while the DbRTS strategy selects valid ATCs which traverse delta model elements. It shows a reasonable reduction in the regression test suite.

We apply our methodology on three industrial case studies including the behavior of a public proxy server inspired by [30], Personal Investment Management System (PIMS) [31], and data transfer interface of computing device [32], which underwent a major design change. Public proxy servers as a free and independent proxy checking system helps users to protect the identity and bypass surfing restrictions based on quality parameters of proxy servers, for example, access time, lifetime, and functionality. PIMS aims a person who has investments in banks and stock market for bookkeeping and computations concerning the investments using software assistance. A data transfer interface of computing device describes the behavior of device in data transfer rate, power consuming and management, full duplex data communications, and interrupt-driven functionality management.

1 (S7: *State*, ***AddMT***, empty, NewVal)
2 (S8: *State*, ***AddMT***, empty, NewVal)
3 (T15: *Transition*, ***AddMT***, empty, NewVal)
4 (T16: *Transition*, ***AddMT***, empty, NewVal)
5 (T17: *Transition*, ***AddMT***, empty, NewVal)
6 (T18: *Transition*, ***AddMT***, empty, NewVal)
7 (T19: *Transition*, ***AddMT***, empty, NewVal)
8 (T14: *Transition*, ***DelMT***, OldVal, empty)
9 (T6: *Transition,* ***UpdateMT(event)***, OldValue, NewVal)
10 (T7: *Transition,* ***UpdateMT***, OldValue, NewVal)
11 (T8: *Transition,* ***UpdateMT***, OldValue, NewVal)

(a). The *optimized* delta model of Fig. 2

CC: All states	**CC**: All transitions
<S0,S1,S2,S3,S1,S6>	<T1,T2,T5,T10,T12,T13>
<S0,S1,S2,S4,S5,S1,S6>	<T1,T2,T4,T9,T5,T10,T12,T13>
<S0,S1,S2,S5,S8,S1,S6> N	<T1,T2,T3,T11,T12,T13>
<S0,S1,S2,S3,S7,S1,S6> N	<T1,T2,T14,T13>R
	<T1,T2,T6,T10,T12,T13>U
	<T1,T2,T4,T7,T10,T12,T13>U
	<T1,T2,T4,T8,T10,T12,T13>U
	<T1,T2,T3,T17,T18,T12,T13>N
	<T1,T2,T6,T16,T19,T12,T13>N
	<T1,T2,T4,T15,T19,T12,T13>N

(b). The consistent test requirements after propagating the delta model

Listing 4. Test suite categorization for the web service behavioral model

In our experimental studies, we investigate the number of test cases after changes in the retest-all, DbRTS, and optimized DbRTS techniques after modifying the system model. The optimized

DOI: 10.1002/smr

DbRTS method removed redundant modification-traversing test case from consistent test suites provided by the DbRTST. It can use the prioritization techniques (e.g., frequency of occurrence of changes or a longer path) to remove redundant ATC among different test cases which traverse a specific delta element.

Tables III and IV present the results of the regression test selection using these techniques and the number and type of delta operations that performed on the case study design models. As shown in the comparison result of Figure 7, using the optimized DbRTST leads to a reasonable reduction in re-executing of all test cases.

7.3.1. Change coverage analysis. In order to evaluate the effectiveness of the delta-based coverage criteria, two alternative measures are introduced: firstly, the coverability of a delta-based criterion; and secondly, the safety percentage of a delta-based criterion. Each updated or new element in a delta operation may be covered by a regression test requirement or not, which is denoted by the binary decision variable $p_i \in \{0,1\}$. The parameter d_i denotes a non-removed delta element i in a specific coverage criterion (e.g., in delta-state coverage rule, d_i denotes a changed state i), and |R| determines the number of non-removed changed elements in a test model. We define the coverability of a delta-based criterion (*Cov*) as follows:

$$Cov = \% \frac{\sum_{i=0}^{n} p_i d_i}{|\mathrm{R}|} \tag{1}$$

Table III. Impact analysis results using different techniques.

	Before Chg #ATCs	After Chg			Retest-all #ATCs	DbRTST #ATCs	Optimized DbRTST #ATCs
		#RemATCs	#UpATCs	#NewATCs			
CS1	18	3	9	7	22	16	16
CS2	30	6	15	11	35	26	21
CS3	44	10	21	17	51	38	30

DbRTST, delta-based regression test selection technique.

Table IV. Delta operations on the case studies.

	AddS	AddT	DelS	DelT	UpdateS	UpdateT
CS 1	4	2	1	2	0	2
CS 2	4	7	0	1	2	6
CS 3	6	9	3	5	3	7

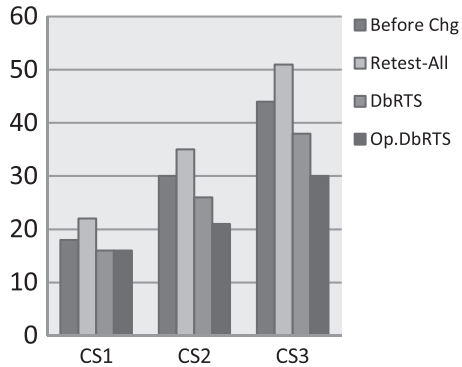


Figure 7. Comparison of regression test selection techniques (RTSTs) with delta-based regression test selection technique (DbRTST).

DOI: 10.1002/smr

Let |S| and |S′| represent the total number of delta-traversing ATCs and selected test cases by a delta-based coverage criterion, respectively. To compare two different coverage rules, the *Safety Percentage (SP)* is calculated as follows:

$$SP = \%\frac{|S'|}{|S|} \tag{2}$$

If the value of *Cov* or *SP* for a delta-based coverage criterion equals to '1', then the selected regression test suite is 100% safe.

Using these metrics, testers can compare different coverage criteria. Obviously, equivalent coverage criteria in terms of the coverability should have the same value. Because ad hoc coverage criteria (as customized reduction techniques) select a representative subset from the original test pool, it is vital to evaluate the adequacy of the resulting subset using the coverability metric and to compare its value by proper assigned values. For example, we calculate the coverability of some coverage rules on the case study as follows (the transition set of the changed model after removing T14, according to Listing 4 (a), is denoted by the vector <T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, ~~T14~~, T15, T16, T17, T18, T19>):

As shown in Table V, the analysis of reduction percentage for two rules with the same *Cov* can provide a strong indicator for testers to keep the size of test suites according to the restricted testing resources.

7.4. Qualitative analyzing of the framework attributes

Metrics can provide purposive approaches for assessing the quality of a framework. We propose two classes of metrics to analyze our framework: platform-independent testing metrics (PITMs) and platform-specific testing metrics (PSTMs). PITMs depend on abstract characteristics and quality attributes and are defined at the platform-independent level of MDT context, for example, safety, generality, and granularity effects. A qualitative analysis of our framework is provided in this section. PSTMs include all metrics that depend on practical platform characteristics of experiment, for example, test execution or test selection time. The PSTM analysis is postponed to the platform-specific testing level for future works.

Characteristic 1. (Safety). The DbRTST always selects *all* delta-traversing test cases from existing test requirements, so it is safe and the selected ATCs are named the *safe test suite*. According to the test suite categorization, a non-minimal safe regression test suite can be formulated as

$$| \, T_{\text{safe-}reg} == T_{\text{AllTest}} \setminus T_{\text{ndt}}$$

where T_{AllTest} and T_{ndt} determine the set of original test suite and non-delta-traversing ATCs, respectively.

Table V. The coverability analysis for some coverage rules

Coverage Rules	Binary Coverage Vector	*Cov* (%)
Rule 1. Delta-transition coverage	<0,0,0,0,0,1,1,1,0,0,0,0,0,0,-,1,1,1,1,1>	100
Rule 2. Additional-transition coverage	<0,0,0,0,0,0,0,0,0,0,0,0,0,0,-,1,1,1,1,1>	62.5
Rule 3. Updated-transition coverage	<0,0,0,0,0,1,1,1,0,0,0,0,0,0,-,0,0,0,0,0>	37.5
Rule 4. Coverage of all updated transitions which their events are changed	<0,0,0,0,0,1,0,0,0,0,0,0,0,0,-,0,0,0,0,0>	12.5
Rule 5. Coverage of at least four additional transitions	<0,0,0,0,0,0,0,0,0,0,0,0,0,0,-,1,1,1,1,1>	62.5
	<0,0,0,0,0,0,0,0,0,0,0,0,0,0,-,0,1,1,1,1>	50
	<0,0,0,0,0,0,0,0,0,0,0,0,0,0,-,1,0,1,1,1>	50
	<0,0,0,0,0,0,0,0,0,0,0,0,0,0,-,1,1,0,1,1>	50
	<0,0,0,0,0,0,0,0,0,0,0,0,0,0,-,1,1,1,0,1>	50
	<0,0,0,0,0,0,0,0,0,0,0,0,0,0,-,1,1,1,1,0>	50
	Maximum coverage	62.5

DOI: 10.1002/smr

Characteristic 2. (Precision). A precise RTS strategy omits test cases that do not produce different program outputs in a changed version [4]. As discussed by Rothermel and Harrold [33], there is no deterministic algorithm to determine the precision of a RTS method. We therefore need to rely on the functionalities provided in our framework.

Because the DbRTST is a safe technique, increases in precision are typically obtained by increases in analysis of the safe test suite. Based on the provided capabilities in our framework, discussion of precision is twofold: (i) an indirect analysis of our specification style that can potentially increase the precision in terms of the level of granularity and (ii) a direct analysis of DQPs that can potentially increase the precision in terms of determining affected elements.

In delta-based regression testing, traceability links aim to follow the paths that are more likely than others to produce different outputs. Fine-grained traceability relations can increase the precision of a selection strategy [34]. To analyze the precision, we consider post-condition variables of delta operations that semantically can produce different outputs in a modified version. These operations are refactoring delta operations that focus on *Event* and *GuardCon* among other *TestMetaModel* elements. So, updatable test cases which traverse post-condition variables of such delta operations are more likely than others to produce different outputs. Formally, $\forall tc: ATC \mid tc \in AllTest \bullet \exists i: 0..\#tc \bullet (tc\ i).2 \in Eve' \vee (tc\ i).3 \in Guardcon'$. Thus, the DbRTST performs graph traversing over fine-grained model elements and follows traceable links to omit ATCs that do not traverse post-state variables. According to [33], graph walk techniques that use dependency information increase the precision in comparison with the methods that choose all test cases. In the DbRTST, a precise regression test suite is a subset of the safe regression test suite which its members traverse post-state variables. Formally, $T_{precise\text{-}reg} \subseteq T_{safe\text{-}reg}$.

Furthermore, by direct analysis of test requirements using DQPs, testers can extract abstract test cases for precise regression testing. For example, a DQP can identify probable elements to discover faults as $Traverse \langle\langle \mathbb{N} \times Event' \times UpdateMT \rangle\rangle$.

Characteristic 3. (Efficiency). There are different parameters to discuss about the efficiency of a RTS approach in terms of its time and/or space requirements [33]. Traceability links in our approach provide enough efficiency in terms of required time for analyzing and classifying regression test cases. Our selective retest method decreases the retesting effort by identifying and bounding testing to only those parts of a test model which are affected by delta operations. The DbRTST uses a detailed delta model to follow changed elements and their dependencies through different versions of a test model. It implicitly promotes accurate propagation of changes to the corresponding test pool and results in a rapid and low-cost change management. The ability to automatically generate new test cases using traceability links and delta-based query patterns is another efficiency parameter for delta-based regression testing, which reduces the time and cost of delta-based regression testing.

Characteristic 4. (Generality). The particular emphasis on integrating meta-modeling and formal approaches in our formalizing style provides a general specification language for platform-independent testing in a wide and practical range of domains. Formal definition of meta-modeling concepts, for example, test meta-model, make it possible to adapt the framework to other behavioral models described in any DSMLs. It potentially provides a 'meta-model independent software testing' which is one of the goals for our future research. For prototyping various DSMLs, our formal framework offers an interesting balance between precision and ease of generation of the formal specifications, and validation and reusability of meta-modeling semantics.

Characteristic 5. (Consistency). Traceability relations between different phases of software development ensure that system analysis, design, and testing are compatible. Our approach propagates traceability links to the maintenance phase and regression testing. We would approve that a consistent regression test suite is a test suite which holds safe accesses. Using traceability links confirm that no access occurs to omitted elements, and proper predicates are defined to cover test requirements upon the current state of a system. In addition, delta-based regression testing relies on an optimized delta model, so minimal and conflict-free records of delta signatures are associated for selecting a consistent regression test suite.

DOI: 10.1002/smr

Characteristic 6. (Strong coverage). The flexible specification of the coverage criteria (strategy-based and ad hoc) provides an interactive handle for defining accurate regression testing requirements. In order to obtain adequate coverage in regression testing, testers can use queries (e.g., coverage of all updated transitions which their events are changed) to specify effective coverage patterns at the model level and to generate significant test suites incrementally. Figure 8 shows the relations between different retest strategies and their corresponding elements. The traceability links indicate that ad hoc and refined coverage rules *AdhocTraceLink* can cover subsets of delta model elements, the DbRTST covers all delta model elements according to *DeltaTraceLink* definition, and the retest-all strategy covers all changed and unchanged model elements as shown by *TraceLink*. The coverage of different rules can be evaluated using Equation (1) in Section 7.3.1. Customized coverage goals for model-driven regression testing improve the productivity by providing better control of the costs in the systems involving large-scale changes.

Characteristic 7. (Trace-based error localization). Delta-based regression testing promotes early fault detection to platform-independent software maintenance. It can discover the faults of model refactoring considering the pre-conditions of delta operations at an abstract level of software maintenance, and can facilitate maintenance-error localizing using backward traceability links among different versions of a system specification.

Characteristic 8. (On-the-fly testing). On-the-fly testing reduces complexity of the testing phase by combining test derivation and test execution. Incremental pattern matching using transformations without the costly re-evaluation of unchanged parts of the evolving model promotes on-the-fly regression testing. It leads to reduce the cost and effort in propagating changes to testing artifacts in model-driven regression testing. This regression testing strategy, instead of deriving a complete test suite for a system, only derives the afterwards test requirement based on specification changes. Also, the automatic mapping between an abstract test template and its executable version provides using refinement theory supported in some existing testing tools.

Characteristic 9. (Query-driven regression testing). Query-driven regression testing provided by the extendable language syntax DQP in Definition 22 will reduce the inherent complexity and the cost of evaluating complex design changes. Dealing with complex models, however, is almost exclusively an issue of the back end, using filter functions to select desired parts, for example, according to the fault diversity of the updated model or coverage-measures will decrease the complexity of regression testing.

7.5. External validity

Because model-based regression testing results on industrial systems are scarce in the literature, especially when integrated with the MDA capabilities, the analysis reported in this paper still can provide an important contribution despite its limitations. Our results rely on three industrial case studies complemented by the sets of delta operations, and simulated test suites provided us with a useful analytical base to assess the claims of the proposed framework. As shown in the comparison result of Figure 7, using the optimized DbRTST leads to a reasonable reduction in re-executing of all test cases. However, replicating our studies in various domains with different characteristics

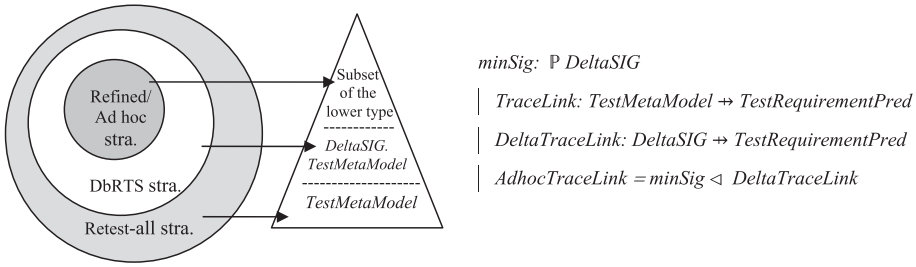


Figure 8. Traceability and the levels of coverage.

DOI: 10.1002/smr

(e.g., different sizes of models and number of change operations) acquires higher confidence in our results and better understanding of their limitations. We left as future work the evaluation of concrete DbRT when applied to more industrial systems.

Two main dimensions in coverage-based testing are elements which should be covered and methods which maximized the coverage. Therefore, we first need to determine the best coverage-based technique in the DbRTST. We are therefore interested in finding a heuristic that helps testers choose an optimal size of a selected test suite for DbRT. The proposed query-based regression testing as meta-heuristic method in regression testing can optimize coverage-based regression testing when it is automated as a testing task. Furthermore, in order to obtain a better result, we need to optimize DbRT fault revealing patterns within resource and time constraints.

However, incremental updating techniques have been widely used in different fields of computer science as a high-performance approach; their performance has been evaluated using some dedicated benchmark examples. To improve the performance adequacy of DbRT which used incremental updating, we plan to provide support for ad hoc pattern matching to allow testers to define efficient granularity of changes in the regression testing process.

8. RELATED WORK

Traditionally, there are two methods for specification-based regression testing, namely, informal methods (e.g., UML-based methods) and formal methods. Although integrated strategies by combining the advantages of UML and formal methods have attracted increasing attention, there is still a lack of integrated approaches in regression testing. An overview of related approaches is given as follows:

8.1. Informal methods

Unified modified language as a modeling language is designed to provide a standard way to visualize the design of a system. Various UML-based regression testing techniques are reported in the different substantial research areas of regression testing, for example, [4, 33, 35–37] and [38] in UML-based RTSs and [39–41] and [42] in UML-based regression test prioritization and minimization techniques.

A survey of RTSs provided by Rothermel and Harrold [33] and a recent systematic review are presented by [35] that classified model-based and code-based RTSs. Briand *et al*. [4] proposed an RTST based on analysis of UML sequence and class diagrams. Their approach assumes full traceability between the design model(s), the code, and the test cases. They also present a prototype tool to support the proposed impact analysis strategy.

Farooq [36] presented an RTS approach based on identified changes in both the state and class diagrams of UML used for model-based regression testing. They utilized Briand *et al.* [4] classification to divide test suites into obsolete, reusable, and re-testable. Also, an Eclipse-based tool for model-based regression testing compliant with UML2 is proposed in their research.

Chen *et al.* [38] proposed a specification-based RTS technique based on UML activity diagrams for modeling the potentially affected requirements and system behavior. They also classified the regression test cases that are to be selected into target and safety test cases based on the change analysis. Wu and Offutt [37] presented a UML-based technique that attempts to help resolve difficulties introduced by the implementation transparent characteristics of component-based software systems. In corrective maintenance activities, the technique started with UML diagrams that represent changes to a component and used them to support regression testing. Also, a framework to evaluate the similarities of the old and new components and corresponding retesting strategies provided in this paper.

Korel *et al.* [40] presented several model-based test prioritization heuristics. They also compared model-based test prioritization heuristics using an experimental study to show their effectiveness in early fault detection. Hemmati *et al.* [39, 41] examined different diversity-based algorithms which operate on the similarity between test cases' paths in the state model. They gave high priority to test

DOI: 10.1002/smr

cases whose paths are most dissimilar to compare the effectiveness of the best similarity-based selection technique with other frequent selection techniques in the literature.

Tahat *et al.* [42] presented and evaluated two model-based selective methods and a dependence-based method of test prioritization utilizing the state model of the system under test. These methods considered the modifications both on the code and model of a system. The existing test suite is executed on the system model and its execution information is used to prioritize tests.

However, a few of UML-based approaches cover MDA aspects. Naslavsky *et al.* [34] presented an idea for regression testing using class diagrams and sequence diagrams using MDA concepts. They make use of traceability for regression testing in the context of UML sequence and class diagrams. Farooq [43] discussed a model-driven methodology for test generation and regression test selection using BPMN 2.0 [44] and UML2 Testing Profile [15] for test specification while a trace model is used to express relation between source and target elements.

Pilskalns *et al.* [45] discussed another modeling approach for regression testing the design models instead of testing the concrete code. In some studies, similar DSMLs are used for regression testing, for example, Yuan *et al.* [46] utilized the business process diagram and transformed it into an abstract test model to cover structural aspects of test specification.

8.2. Formal methods

Formal methods provide an unambiguous complement for validating and verifying software artifacts at an appropriate level of abstraction. Software testing based on formal specifications can improve the software quality through early detection of specification errors. Different approaches are carried out for test case generation using formal methods, especially Z, for example [47–49] and [28]. But there are scarcely relevant works in the field of regression testing that support formal specifications.

Chen *et al.* [50] presented an approach for regression testing using object-Z specification [51]. They proposed a hierarchical diagram to generate regression tests, from which regression test scenarios are generated. Liang [52] presented an approach for regression testing based on TCOZ [53] specification. TCOZ specifications use an integration of object-Z and timed communicating sequential processes (CSP). To generate regression tests from TCOZ specifications, a graph named as test chart is built from the specification.

To the best knowledge of the authors, there is not an integrated method of MDA and formal notations for regression testing. The proposed formal framework not only can extend to support code generation in MDD using refinement technique but also provides the executable semantics to the modeling notations, which is a shortage in UML2 + OCL notations.

We define a formal concrete scenario for regression testing as a precise and platform-independent specification. It integrates model-based testing and MDA using the capabilities like delta models, traceability, and change propagation towards introducing model-driven regression testing. Table VI compares our work with various approaches showing similarity to our framework for regression testing. The comparison criteria include the support of: (i) precision; (ii) safety; (iii) efficiency; (iv) generality or platform independency; (v) query-driven testing; (vi) on-the-fly or incremental testing; (vii) trace-based error localization; (viii) customized coverability; (xi) automatic refinement; and (x) fault modeling.

Our approach for model-driven regression test selection is safe, efficient, and potentially more precise than the other similar approaches. DbRT by means of exploring fine-grained traceability links, leveraging on-the-fly testing, and supporting the direct analysis of DQPs to determine affected elements is capable of achieving better precision than the comparable model-based approaches. The approaches provided in [4, 34, 36] and [43] support model-based regression test selection rely on traceability relationships among artifacts. These selective approaches are safe, efficient, and precise. But, the majority of the approaches do not support on-the-fly testing, automatic refinement, and fault modeling in model-based regression testing. Query-driven regression testing and the ability of covering customized evolving parts in a model, as the new capabilities in regression testing, are proposed in this paper. Finally, our approach due to its formalizing style can be extended to other DSMLs, while the domain of the other approaches is limited to a specific modeling language.

DOI: 10.1002/smr

Table VI. Comparison of the proposed framework to the other similar studies.

	Precision	Safety	Efficiency	Generality	Query-driven testing	On-the-fly testing	Trace-based error localization	Customized coverability	Automatic refinement	Fault modeling
Briand *et al.*(2009)	✓	High	✓				✓			
Farooq *et al.* (2007)	✓	High	✓				✓			
Naslavsky *et al.* (2007)	✓	High	✓				✓		✓	
Farooq (2010)	✓	High	✓				✓		✓	
Wu and Offutt (2003)		Low								
Chen *et al.* (2002)		Medium								
DbRT	✓	High	✓	✓	✓	✓	✓	✓	✓	✓

DbRTST, delta-based regression test selection technique.

DOI: 10.1002/smr

9. CONCLUSION AND FUTURE WORK

In complex and rapidly evolving systems, mature engineering disciplines model systems as an abstract state and a sequence of operations on this state. Operations on the systems will continuously evolve and improve the design models using refinement steps or model refactoring transformations. Specifying these systems using the Z formal language can reduce complexity and significantly enhance quality, by supporting refinement based on mathematically relating the abstract and concrete states and by supporting separation of concerns in describing and analyzing different aspects of them in isolation. Because modifications of critical components of a product can impact the reliability, a precise change impact analysis during regression testing is also needed to identify any undesired impact on the product quality. For regression testing to be effective in such systems, it needs to be seen as a comprehensive testing methodology that considers abstract level testability and traces levels of abstraction towards an implementation to verify the functionality after ongoing changes. The proposed formal framework aims to keep pace with model-driven regression testing in a formal way to develop an effective testing approach for the system evolution process.

Delta-based regression testing can solve a number of restricted outlooks in model-driven regression testing, including well-defined delta signature definition, regression test suite identification, and some limiting factors of incremental maintenance and retesting. In this paper, key concepts of delta-based regression testing includes system refactoring, model transformation, traceability, delta-based coverage criteria that are formalized by Z notation that can provide the input of Z-based tools for various kinds of dynamic and static analysis, and functional verification.

We consider two distinct phases in delta-based regression testing: the trace link initialing phase and the update phase. In the first phase, traceability links using derivation rules between system behavior model and abstract test cases are created. After system evolution, test requirements will be updated to keep consistency between the PIM and the PIT. At this point, the critical phase of regression testing, that is, selecting regression test suite based on delta models begins. It aims at minimizing the huge costs of retesting all test cases using change impact analysis and tackling the RTS problem at the platform-independent level. The DbRTS approach uses an optimized delta model to propagate PIM changes to the PIT that results in minimum redundant and conflict-free regression test suites.

We divide abstract test suites into distinct categories according to the delta operations which they traverse. Test case generation for new functionalities of a system is performed by applying derivation rules on recently added elements and targeting them in new coverage rules. Delta-based regression testing is enriched by prioritization and refinement techniques that are also reusable in platform-specific regression testing. In addition, a precise formalism to define model-level coverage criteria and adequate test requirements is proposed. The result of applying DbRT on three practical case studies, with different sizes and change scenarios and from different application domains, shows a reasonable reduction in the number of regression test cases. By analyzing the framework, we have also identified some key quality attributes such as safety, precision, efficiency, effective granularities, strong coverage, and generality for delta-based regression testing. Finally, the safety of the reduced test suites is evaluated by new metrics.

In future works, we will continue to refine the approach from the platform-independent (abstract) level to the platform-specific (concrete) level using the automatic theory of behavior refinement and evaluate the platform-specific testing using appropriate PSTMs. At the platform-specific level, we provide an automatic tool to generate regression test cases for the delta-based coverage criteria. It can be integrated with any modeling environments, for example, VIATRA2, a parser, for example, XMLWare, and a testing tool, for example, JUnit. This combination enables testers to define expressive queries to determine the uncovered elements in regression testing and cover a recently changed element adequately.

Moreover, we provide more complex DQPs integrated by a tool to promote ‘query driven regression testing’ as a cost-effective regression testing approach. A query-driven test case generator provides precise regression test cases which should be supplemented with an engine that generates abstract test cases in response to queries for regression testing. Finally, a primary focus for future work is to investigate how model-driven transformations can be implemented on other existing model transformation used for bidirectional model synchronization languages like Triple Graph Grammars

DOI: 10.1002/smr

or Query/View/Transformation relations and ATL. Concrete model-driven regression testing can be enriched by a tool for ranking potential failure-inducing changes at the design level in order to reduce developers' effort in manually examining all affecting changes. This tool should be compared with different spectrum-based fault localization techniques in change ranking.

APPENDIX A: ESSENTIAL NOTATIONS OF Z SPECIFICATION LANGUAGE

Essential notations of Z specification language are described as follows:

Notation	Name	Definition
(e1,e2)	Pair	
e1 ↦ e2	Maplet	(e1,e2)
first e (e.1)		first(e1,e2) = e1
second e (e.2)		second(e1,e2) = e2
e1 × e2		{i1 : e1; i2 : e2 • (i1, i2)}
(e1,e2, ...,en)	Tuple extension	(e1,(e2, ...,en)) where n ≥ 2
e1 × e2 × ... × en	Cartesian product	e1 × (e2 × ... × en) where n ≥ 2
∀ x : A | q(x) • p(x)	Universal quantification	∀ x • x ∈ A ∧ q(x) ⇒ p(x)
∃ x : A | q(x) • p(x)	Existential quantification	∃ x • x ∈ A ∧ q(x) ∧ p(x)
∃ 1 x : A | q(x) • p(x)	Unique existential quantification	
e ↑ n	Iterated product	e × ... × e where there are n ≥ 2 occurrences of e
ℙ e	Set of all subsets	{i : 𝕎 | i ⊆ e}
𝔽 e	Set of all finite subsets	{i1 : 𝕎 | ∀ i2 : ℙ ℙ e | ∅ ∈ i2 ∧ (∀ i3 : i2 • ∀ i4 : e • i3∪{i4} ∈ i2) • i1 ∈ i2}
dom e	Domain	{i : e • first i}
ran e	Range	{i : e • second i}
e1◁ e2	Domain restriction	{i : e2 | first i ∈ e1}
e1▷ e2	Range restriction	{i : e1 | second i ∈ e2}
e1⩤ e2	Domain subtraction	{i : e2 | first i ∉ e1}
e1⩥ e2	Range subtraction	{i : e1 | second i ∉ e1}
e1⨾ e2	Relational composition	{i1 : e1; i2 : e2 | second i1 = first i2 • first i1 ↦ second i2}
e1 ⊕ e2	Relational overriding	((dom e2) ⩤ e1) ∪ e
e1 ⇸ e2	Partial functions	{i1 : P (e1 × e2) | ∀ i2, i3 : i1 | first i2 = first i3 • second i2 = second i3}
e1 → e2	Total functions	{i : e1 ⇸ e2 | dom i = e1}
⟨e1, ...,en⟩	Sequence	{1↦ e1, ..., n ↦ en}
θS	Bindings	
ΔS	Delta	Shortcut for S ∧ S′
ΞS	Xi	Shortcut for S ∧ S′ ∧ θS = θS′
S ⨟ T	Schema composition	∃ State″ • (∃ State′ • [S; State″ | θState′ = θState″]) ∧ (∃ State • [T; State″ | θState = θState″])

DOI: 10.1002/smr

APPENDIX B: TEST MODEL REFACTORING

The schemas for updating a test model using direct model transformation using Z notations are presented as follows:

AddTransition

$\Delta TestModel$
$trans?: Transition$
$s1?, s2?: State$
$type?: typeID$
$attribute?: attrID$
$v?: Value$

$trans? \notin Trans; s1? \in Stt; s2? \in Stt$
$Trans' = Trans \cup \{trans?\}$
$value' = value \cup \{trans? \mapsto \{attribute? \mapsto v?\}\}$
$type' = type \cup \{trans? \mapsto type?\}$
$FromSource' = FromSource \cup \{trans? \mapsto s1?\}$
$ToTarget' = ToTarget \cup \{trans? \mapsto s2?\}$
$TagRelation' = TagRelation \cup \{trans? \mapsto New\}$

AddLabel

$\Delta TestModel$
$e?: TestMetaModel$
$trans?: Transition$

$e? \notin TotalElem$
$e? \in Event \Rightarrow Eve' = Eve \cup \{e?\}$
$e? \in GuardCon \Rightarrow Guardcon' = Guardcon \cup \{e?\}$
$e? \in Effect \Rightarrow EffectT' = EffectT \cup \{e?\}$
$labelE' = labelE \cup \{trans? \mapsto e?\}$
$TagRelation' = TagRelation \cup \{e? \mapsto New\}$

DelTransition

$\Delta TestModel$
$trans?: Transition$
$type?: typeID$
$attribute?: attrID$
$v?: Value$

$trans? \in Trans$
$Trans' = Trans \setminus \{trans?\}$
$FromSource' = \{trans?\} \lhd FromSource$
$ToTarget' = \{trans?\} \lhd ToTarget$
$value' = \{trans?\} \lhd value$
$type' = \{trans?\} \lhd type$
$TagRelation' = TagRelation \cup \{trans? \mapsto Delete\}$

DelLabel

$\Delta TestModel$
$e?: TestMetaModel$
$trans?: Transition$

$e? \in TotalElem$
$e? \in Event \Rightarrow Eve' = Eve \setminus \{e?\}$
$e? \in GuardCon \Rightarrow Guardcon' = Guardcon \setminus \{e?\}$
$e? \in Effect \Rightarrow EffectT' = EffectT \setminus \{e?\}$
$labelE' = labelE \setminus \{trans? \mapsto e?\}$
$TagRelation' = TagRelation \cup \{e? \mapsto Delete\}$

DOI: 10.1002/smr

APPENDIX C: FORMALIZING CHANGE PROPAGATING RULES

Formal definition for propagating delta operations in the batch and incremental manner are presented in the schema *BatchDeltaTrans* and schema *IncrementalDeltaTrans*, respectively.

BatchDeltaTrans

$$
\begin{array}{l}
\Delta ApplicableATC \\
\Delta RemovableATC \\
\Delta NewATC \\
\Delta UpdatableATC \\
applicable, removable, newtest: \mathbb{P}\ ATC \\
\hline
\forall delta: TotalDeltaSig \\
\quad \bullet\ delta\ .\ 2 = DelMT \\
\quad \Rightarrow removable' \\
\qquad = removable \cup \{\ s: ATC \mid s \in AllTest \land delta\ .\ 1 \in\ totalTrace\ s\ \} \\
\quad \land\ delta\ .\ 2 = UpdateMT \\
\quad \Rightarrow updatable' \\
\qquad = updatable \cup \{\ s: ATC \mid s \in AllTest \land delta\ .\ 1 \in totalTrace\ s\ \} \\
\quad \land\ delta\ .\ 2 = AddMT \\
\quad \Rightarrow newtest' \\
\qquad = newtest \cup \{\ s: ATC \mid s \notin AllTest \land delta\ .\ 1 \in totalTrace\ s\ \} \\
\quad \land\ applicable' \\
\qquad = applicable \\
\qquad \cup \{\ s: ATC \mid s \in AllTest \land delta\ .\ 1 \notin totalTrace\ s\ \}
\end{array}
$$

IncrementalDeltaTrans

$$
\begin{array}{l}
\Delta ApplicableATC \\
\Delta RemovableATC \\
\Delta NewATC \\
\Delta UpdatableATC \\
d?: DeltaSIG \\
deltats: \mathbb{P}\ TestMetaModel \\
applicable, removable, newtest: \mathbb{P}\ ATC \\
\hline
\exists s: TotalDeltaSig \mid s = d? \\
\quad \bullet\ d?\ .\ 2 = DelMT \\
\quad \Rightarrow removable' \\
\qquad = removable \cup \{\ s: ATC \mid s \in AllTest \land d?\ .\ 1 \in\ totalTrace\ s\ \} \\
\quad \land\ d?\ .\ 2 = UpdateMT \\
\quad \Rightarrow updatable' \\
\qquad = updatable \cup \{\ s: ATC \mid s \in AllTest \land d?\ .\ 1 \in totalTrace\ s\ \} \\
\quad \land\ d?\ .\ 2 = AddMT \\
\quad \Rightarrow newtest' \\
\qquad = newtest \cup \{\ s: ATC \mid s \notin AllTest \land d?\ .\ 1 \in totalTrace\ s\ \} \\
\quad \land\ applicable' \\
\qquad = applicable \cup \{\ s: ATC \mid s \in AllTest \land d?\ .\ 1 \notin totalTrace\ s\ \}
\end{array}
$$

DOI: 10.1002/smr

REFERENCES

1. Thomas D. MDA: revenge of the modelers or UML utopia? *IEEE Software* 2004; **21**(3):15–17.
2. Gargantini A, Riccobene E, Scandurra P. Combining formal methods and MDE techniques for model-driven system design and analysis. *International Journal on Advances in Software* 2010; **3**(1-2):1–18.
3. Harrold MJ. Testing evolving software. *Journal of Systems and Software* 1999; **47**(2-3):173–181.
4. Briand LC, Labiche Y, He S. Automating regression test selection based on UML designs. *Information & Software Technology* 2009; **51**(1):16–30.
5. Spivey JM. *The Z Notation: A Reference Manual*, Second edn. Prentice Hall: Englewood Cliffs, NJ, 1992.
6. Malik P, Utting M. CZT: a framework for Z tools, formal specification and development in Z and B. *Lecture Notes in Computer Science* 2005; **3455**:65–84.
7. Saaltink M. The Z/EVES system. In Proceedings of the 10th International Conference of Z Users on The Z formal Specification Notation, Lecture Notes in Computer Science 1997; **1212**: 72–85.
8. Yoo S, Harman M. Regression testing minimization, selection and prioritization: a survey. *Journal of Software: Testing, Verification and Reliability* 2012; **22**(2):67–120.
9. Rothermel G, Untch RH, Chu C, Harrold MJ. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering* 2001; **27**(10):929–948.
10. Object Management Group. MOF query/views/transformations specification 1.0, 2008. http://www.omg.org/spec/QVT/1.0/PDF.
11. Object management group. MOF 1.3 Specification, 1999. http://www.omg.org/spec/MOF/.
12. Hausmann JH, Heckel R, Sauer S. Extended model relations with graphical consistency conditions. UML Workshop on Consistency Problems in UML-based Software Development, 2002; 61–74.
13. Stevens P. Bidirectional model transformations in QVT: semantic issues and open questions. *Software and Systems Modeling* 2010; **9**(1):7–20.
14. Schürr A, Klar F. 15 years of triple graph grammars. *Proceedings of the 4th International Conference on Graph Transformation. Lecture Notes in Computer Science* 2008; **5214**:411–425.
15. Object Management Group. UML 2.0 testing profile specification, version 1.0, 2005 http://www.omg.org/spec/UTP/1.0/PDF.
16. Dai ZR. Model-driven testing with UML2.0. Proceedings of Second European Workshop on Model Driven Architecture with an Emphasis on Methodologies and Transformations, 2004; 179–187.
17. Heckel R, Küster JM, Taentzer G. Confluence of typed attributed graph transformation systems. In: Graph Transformation, First International Conference. Lecture Notes in Computer Science Volume **2505**, 2002; 161–176.
18. Mens T, Gorp PVA. Taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science* 2006; **152**:125–142.
19. Mens T. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering* 2002; **28**(5): 449–462.
20. Buckley J, Mens T, Zenger M, Rashid A, Kniesel G. Towards a taxonomy of software change. *Software Maintenance and Evolution* 2005; **17**(5):309–332.
21. Bergmann G, Ráth I, Varró G, Varró D. Change-driven model transformations. *Software & Systems Modeling* 2012; **11**(3):431–461.
22. Myers G. *The Art of Software Testing*, (2nd edn) John Wiley & Sons, Inc.: New Jersey, 2004.
23. Zhu H, Hall PA, May HR. Software unit test coverage and adequacy. *ACM Computing Surveys* 1997; **29**(4):336–427.
24. Utting M, Pretschner A, Legeard B. A taxonomy of model-based testing. Technical Report 04/2006, Department of Computer Science, The Universiy of Waikato (New Zealand).
25. Darabos A, Pataricza A, Varró D. Towards testing the implementation of graph transformations. *Electronic Notes in Theoretical Computer Science* 2008; **211**:75–85.
26. Jones J, Harrold M, Stasko J. Visualization of test information to assist fault localization. Proc. of ICSE, ACM 2002; 467–477.
27. Woodcock J, Davies, J (1996). *Using Z: Specification, Refinement, and Proof*. Prentice Hall, ISBN-13:978-0139484728.
28. Cristià M, Rodriguez Monetti P. Implementing and applying the Stocks–Carrington framework for model-based testing. Springer-Verlag 2009 ICFEM; volume 5885: 167–185.
29. Varró D, Balogh A. The model transformation language of the VIATRA2 framework. *Science of Computer Programming* 2007; **68**(3):214–234.
30. Balsamo S, Marzolla M. Performance evaluation of UML software architectures with multiclass queueing network models. WOSP 2005; 37–42.
31. Jalote P. *An Integrated Approach to Software Engineering*, (3rd edn) Springer: New York, 2006.
32. B'Far R, Fielding R. Mobile computing principles: designing and developing mobile applications with UML and XML 2004. ISBN-13: 978-0521817332.
33. Rothermel G, Harrold MJ. Analysing regression test selection techniques. *IEEE Transactions on Software Engineering* 1996; **22**(8):529–551.
34. Naslavsky L, Richardson DJ. Using traceability to support model-based regression testing. Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering 2007; 567–570.

DOI: 10.1002/smr

35. Engström E, Skoglund M, Runeson P. Empirical evaluations of regression test selection techniques: a systematic review. Proceedings of the Empirical software engineering and measurement. ACM-IEEE international symposium, 2008; 22–31.
36. Farooq Q. A model driven approach to test evolving business process based systems. ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems, 2010; 16–24.
37. Wu Y, Offutt J. Maintaining evolving component based software with UML. Proceeding of 7th European conference on software maintenance and reengineering, IEEE 2003; 133–142.
38. Chen Y, Probert R, Sims D. Specification-based regression test selection with risk analysis. Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research. ACM 2002; page 1.
39. Hemmati H, Arcuri A, Briand L. Reducing the cost of model-based testing through test case diversity. Proceedings of the 22nd International Conference On Testing Software and Systems. Lecture Notes in Computer Science Volume 6435, 2010; 63–78.
40. Korel B, Koutsogiannakis G, Tahat L. Model-based test prioritization heuristic methods and their evaluation. Proceedings of the 3rd International Workshop on Advances in Model-Based Testing, ACM 2007; 34–43.
41. Hemmati H, Arcuri A, Briand L. Achieving scalable model-based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology* 2013; **22**(1):6.
42. Tahat L, Korel B, Harman M, Ural H. Regression test suite prioritization using system models. *Journal of Software Testing, Verification & Reliability* 2012; **22**(7):481–506.
43. Farooq Q, Iqbal MZ, Malik ZI, Nadeem A. An approach for selective state machine based regression testing. Proceeding of AMOST. ACM 2007; 44–52.
44. Object Management Group. BPMN 2.0 Specification, 2009. http://www.omg.org/spec/BPMN/2.0/.
45. Pilskalns O, Uyan G, Andrews A. Regression testing UML designs. ICSM. 22nd IEEE International Conference on 2006; 254–264.
46. Yuan Q, Wu J, Liu C, Zhang L. A model driven approach toward business process test case generation. 10th International Symposium on Web Site Evolution. IEEE 2008; 41–44.
47. Ammann P, Offutt J. Using formal methods to derive test frames in category-partition testing. Proceeding of the 9th Annual Conference on Computer Assurance. IEEE Computer Society 1994; 69–80.
48. Carrington D, Stocks P. A tale of two paradigms: formal methods and software testing. In Z User Workshop, Cambridge Workshops in Computing. Springer-Verlag 1994; 51–68.
49. Stocks PA, Carrington D. Test templates: a specification-based testing framework. Proceeding of 15th Int. Conf. on Software Engineering 1996; 405–414.
50. Chen C, Chapman R, Chang KH. Test scenario and regression test suite generation from object-Z formal specification for object oriented program testing. Proceedings of the 37th Annual Southeast Regional Conference 1999, DOI: 10.1145/306363.306408.
51. Duke R, Rose G. *Formal Object-Oriented Specification Using Object-Z.* MacMillan: Basingstok, 2000.
52. Liang H. Regression testing of classes based on TCOZ specifications. Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems 2005; 450–457.
53. Mahony B, Dong JS. Timed communicating object Z. *IEEE Transactions on Software Engineering* 2000; **26**(2): 150–177.

DOI: 10.1002/smr