Model-based Regression Testing of Evolving Software Systems

Dissertation with the aim of achieving the doctoral degree of

Doktor der Naturwissenschaften (Dr. rer. nat.)

Department of Software Engineering and Construction Methods
Faculty of Mathematics, Informatics, and Natural Sciences
University of Hamburg

submitted by

M.Sc. Qurat-Ul-Ann Farooq

Hamburg, 2015

Date of Oral Defense: 24.06.2016

The following evaluators recommend the admission of the dissertation:

Name: Univ.-Prof. Dr.-Ing. habil. Matthias Riebisch

Name: Prof. Dr. Uwe Aßmann

Abstract

Changes are required to evolve software systems in order to meet the requirements emerging from technological advances, changes in operating platforms, end user needs, and demands to fix errors and bugs. Studies have shown that about one third of a project's budget is spent to test these changes to ensure the system correctness. To minimize the testing cost and effort, model-based regression testing approaches select a subset of test cases earlier by tracing the changes in analysis and design models to the tests. However, modeling of complex software systems demands to cover various views, such as *structural view* and behavioral view. Dependency relations exist between these views due to overlapping and reuse of various concepts. These dependency relations complicate regression testing by propagating the changes across the models of different views and tests. Change impact analysis across the models and tests is required to identify the potentially affected test cases by analyzing the change propagation through dependency relations.

This thesis presents a holistic model-based regression testing approach, which exploits the interplay of changes and dependency relations to forecast the impact of changes on tests. A baseline test suite, the one used for testing the stable version of software, is required for the selection of test cases. To enable a test baseline, our approach supplements a model-driven test generation approach that uses model transformations to generate various test aspects. The approach uses BPMN and UML models and generates test models expressed as UML Testing Profile (UTP).

Dependency relations are recorded prior to the impact analysis by using a two-fold approach; during the generation of baseline test suite and by using a rule-based dependency detection approach. This prevents repeated search of dependency relations for each change and makes our approach less time extensive. Change impact analysis across tests is supported by integrating a rule-based impact analysis approach. The approach enables a set of rules, which analyze previously recorded dependency relations and change types to further propagate the impact of a change. To precisely define various changes in models, the approach also synthesizes a change taxonomy for a consistent representation of complex changes in the models. Finally, to distinguish between various potentially affected tests, our approach presents the concept of test classification rules. Test classification rules analyze the type of an affected element, the type of the applied change, and other related elements to decide whether the affected element is obsolete, unaffected, or required for regression testing.

To demonstrate the applicability of our approach in practice, we adapted our approach for the domain of business processes and support BPMN, UML, and UTP models. The tool support for our approach is available in two prototype tools; VIATRA Test Generation Tool (VTG) and EMFTrace. VTG generates UTP test baseline from BPMN and UML models using model transformations. EMFTrace is a tool, which was built initially to support the rule-based dependency detection among models. It is further extended to support the rule-based impact analysis and rule-based test classification. These tools help us to evaluate our approach on a case study from a joint industrial project to enable business processes of a *field service technician* on mobile platforms. The results of our evaluation show promising improvements with an average *reduction* of the test cases by 46% achieved with an average *precision* and *recall* of 93% and 87% respectively.

Kurzfassung

Software Systeme erfordern kontinuierliche Änderungen, um sie an die Anforderungen neuer Technologien, Einsatzumgebungen und Kundenwünsche anzupassen sowie um bestehende Defekte zu beheben. Dabei wurde durch Studien nachgewiesen, dass bis zu einem Drittel der gesamten Projektkosten allein auf das Testen der Software nach Änderungen entfallen, um weiterhin deren Korrektheit zu gewährleisten. Um diese Kosten zu minimieren, können durch modellbasiertes Testen nur die absolut notwendigen Testfälle durch die Nachverfolgung von Änderungen vorselektiert werden. Durch den reduzierten Testaufwand ist es darüber hinaus auch möglich, selbst in noch frühen Entwicklungsphasen systematische Regressionstests auszuführen. Die Modellierung komplexer Systeme setzt jedoch die Nutzung verschiedener Sichten voraus, so z.B. die Struktur- oder Verhaltenssicht eines Systems. Aufgrund von Überschneidungen und Abhängigkeiten dieser Sichten breiten sich Änderungen über mehrere Sichten und deren Modelle hinweg aus und erschweren somit das Regressionstesten. Mithilfe von Impact Analyse muss diese Änderungsausbreitung erfasst werden, um damit die für die Regressionstests einzuschließenden Testfälle zu identifizieren.

Diese Arbeit präsentiert einen umfassenden Ansatz für modellbasiertes Regressionstesten, der das Zusammenspiel verschiedener Typen von Abhängigkeiten und Änderungsoperationen für die Auswahl von Testfällen auswertet. Der Ansatz zeichnet die zu untersuchenden Abhängigkeiten zunächst auf, um eine wiederholte Abhängigkeitsanalyse pro Änderungsoperation zu vermeiden. Zur Ermittlung der Auswirkungen der Änderungen auf Modelle und Testfälle, integriert der Ansatz ein regelbasiertes Impact Analyse Verfahren, welches das Zusammenspiel von Änderungsoperationen und Abhängigkeitsbeziehungen analysiert. Die verwendeten Regeln bestimmen die Ausbreitung von Änderungen, indem sie die Abhängigkeitsbeziehungen zwischen Modellen untersuchen. Für die Definition von Änderungen stellt diese Arbeit eine Taxonomie von Änderungsoperationen bereit, die die konsistente Modellierung auch von komplexen Operation ermöglicht. Da das Vorhandensein einer Testbaseline eine wesentliche Voraussetzung des Regressionstestens ist, stellt der Ansatz zusätzlich ein modellgetriebenes Testerzeugungsverfahren vor, um eine solche Testbaseline durch Modelltransformationen zu erzeugen. Um zwischen den betroffenen Testfällen weiter differenzieren zu können, stellt diese Arbeit ein weiteres Konzept vor, um Testfälle entweder als Obsolet, Wiederverwendbar, Testbar, oder Neu zu kategorisieren.

Um die Anwendbarkeit des Ansatzes auch für praktische Probleme zu demonstrieren, wurde der Ansatz für die Domäne der Geschäftsprozesse erweitert und hinsichtlich der Unterstützung von BPMN, UML sowie dem UML Testing Profile (UTP) angepasst. Geeignete Werkzeugunterstützung wird dabei durch zwei Prototypen bereitgestellt: das VIATRA Test Generation Tool (VTG) und EMFTrace. Während VTG die Testbaseline aus BMPN- und UML-Modellen durch Modelltransformationen generiert, ermöglicht EMFTrace eine automatisierte Suche nach Abhängigkeiten zwischen diesen Modellen und bietet darüber hinaus Unterstützung für regelbasierte Impact Analyse sowie regelbasierte Testklassifizierung. Mithilfe beider Werkzeuge wurde eine Fallstudie im Rahmen eines industriellen Kooperationsprojektes durchgeführt, im Laufe derer typische Geschäftsprozesse von Servicetechnikern auf mobile Endgeräte abgebildet wurden. Die Ergebnisse dieser Studie zeigen dabei deutliche Verbesserungen hinsichtlich Precision und Recall und der Reduktion des Testaufwands sowie dem Abdeckungsgrad der Testfälle durch den vorgestellten Ansatz für modellbasiertes Regressionstesten.

Acknowledgment

I would like to thank all those who supported me throughout my doctorate. First of all, I would like to express my deepest gratitude for my supervisor Prof. Matthias Riebisch for his continuous mentoring, guidance, and encouragement. I cannot thank him enough for giving me this opportunity, keeping me motivated, giving required feedback, and having constructive discussions. I thank him for his patience and support during those times when I was not at my best. I would like to thank Prof. Uwe Aßman for giving his valuable time to review my thesis. I am really grateful to him for providing useful feedback to improve this menuscript. I am also very thankful to Steffen for being such a great support. I am really lucky to find a friend like him who was always so full of ideas and positive energy. I want to thank my colleagues at Ilmenau, who always supported me in every possible way. My sincere thanks to Prof. Phillipow, Nils, Heiner†, Patrick, and all others for everything.

I am very grateful to my colleagues at Hamburg. Though, we spent very little time together, all of you made me really welcome. My sincere thanks to Soliman and Sabastian for their useful comments during the thesis writeup. I want to take this opportunity to thank all my teachers and colleagues in Pakistan, who motivated me to peruse a doctorate and trusted my abilities. A lot of respect and gratitude for Prof. Zafar Malik for being an inspiration and a continuous source of encouragement. My deepest regards for my parents, my brothers, and the whole family for their encouragement and continuous love. Special thanks to my other half Naseer for his tremendous support, understanding, and patience. Finally, I would like to say thanks to my friends for being there, whenever I needed them during this long period of time. Heartiest thanks to Sana, Noman Bhai, Iram, Sadaf, and all others.

Table of Contents

List of Figures

List of Tables

Abbreviations

CPM	Category Partition Method
DFS	Depth First Search
EMF	Eclipse Modeling Framework
MBRT	Model-based Regression Testing
MBT	Model-based Testing
MDT	Model-driven Testing
MOPS	Adaptive Planning and Secure Execution of Mobile Processes in Dynamic Scenarios
OMG	Object Management Group
OWL	Web Ontology Language
PIM	Platform Independent Model
PIT	Platform Independent Test Model
RNS	Random Name Similarity-based test selection approach
SUT	System Under Test
TTCN	Testing and Test Control Notation
URN	User Requirement Notation
UTP	UML Testing Profile
VIATRA	Visual Automated Model Transformations
XMI	XML Meta-data Interchange

1

Introduction and Motivation

Software systems evolve continuously to accommodate new requirements, new technologies, and end user needs. These changes can adversely affect the quality of the software systems due to unintended side effects by introducing additional defects and errors. Testing as an instrument to detect these errors requires substantial effort and often consume a higher percentage of the project budget. Studies have shown that the cost spent on the testing budget can consume about one third of the total cost of the project [Har98][LW89][Whi00]. This includes the cost of developing tests, executing them, comparing their results, and then tracking the detected failures [OTPS98].

Regression testing aims to reduce the testing effort and consequently saves the cost by limiting the test execution to a subset of test cases corresponding to the changes [RH96]. One of the approaches to perform regression testing is to use different versions of source code, compare them to obtain a set of changes, and trace these changes to the test cases. Although, precision of the source code-based regression testing approaches might be higher, these approaches fail to deal with the complexity of larger systems. Moreover, overall efficiency of the regression testing process is compromised as the regression testing activity can only be started after the implementation. It is believed that starting the regression testing activity earlier in the development life cycle can reduce the testing cost by providing an early assessment of the test effort, enabling the test planning prior to the implementation, and early tracking the detected failures [BLY09].

Model-based regression testing (MBRT) is another complementary approach, which analyzes the changes in analysis and design models and traces them to the potentially affected tests. Thus, the overall test effort can be reduced by starting the regression testing activity before the actual implementation of the changes [BLY09]. However, the representation of complex software systems demands for modeling different views to represent their structure, behavior, and other relevant aspects [Gom11, FPK+12, PE00]. These views represent different aspects of the same system, which might result in overlapping of concepts and introduce dependency relations between models representing these views. Changes propagate through these dependency relations and can potentially impact the tests. Change impact analysis across models and tests is required to asses the change propagation through dependency relations to find potentially impacted subset of tests for regression testing [BLS02, BG00, WO03]. Dependency rela-

tions are of crucial importance to support change impact analysis, as they propagate changes across several models and tests.

Our analysis of the state of the art model-based regression testing approaches shows that they provide limited support for different types of dependency relations among models of various system views and tests. Moreover, if dependency relations are supported to some extent, they are repeatedly searched for each change, which is time extensive. An effective solution to deal with this issue is to record dependency relations prior to impact analysis, as the dependency relations can be reused for each change. However, this aspect is overlooked by the mainstream regression testing literature.

Besides the issue of dependency relations, the change support provided by these approaches is also insufficient. Most of the approaches use inconsistent and incomplete set of changes by considering only a set of basic changes in models, such as *add* and *delete*. Nevertheless, changes in models can be complex, for example, moving, merging, or replacing of model elements. Such complex changes are not supported by the existing model-based regression testing approaches.

Another important aspect neglected by the existing model-based regression approaches is that how the *baseline test suite* is represented and generated? A *baseline test suite* is the one used for testing the stable version of the software before the changes and its subset is required to be selected during regression testing. Recent developments in model-based testing emphasize the need of using test models to express various aspects of tests, such as *test architecture*, *test behavior*, and *test data* [BDG+07, SS09]. These aspects represent the *test view* of a system and a *baseline test suite* should constitute all these aspects. These aspects of the *test view* are required to be analyzed to provide a thorough coverage during regression testing. However, the existing model-based regression testing approaches lack the support for test models and standard test specification languages.

Furthermore, to enable the application of approaches for a wide range of models and test specification languages, the ability to integrate new models and test specification languages by means of extendable and flexible solutions is required [ZFKB12]. Similarly, the system modeling and test specification languages also evolve to support new concepts. In these cases, most of the existing model-based regression testing approaches require substantial effort to keep their tool-set stable and up-to-date and require modifications in the source code.

Considering the limitations of the state of the art model-based regression testing approaches, the need for a model-based regression testing approach to address these issues is evident. Various characteristics of the required approach include: (1) inherent support for dependency relations of various types among models of different system views and various test aspects, (2) support for complex changes in models, (3) explicit support for baseline test generation approach to cover various aspects of the *test view*, and finally (4) the capability to extend the approach easily to support new and evolved modeling and test specification languages.

1.1 Research Questions

The research questions presented in this section enable us to focus on the core issues and problems that we want to address in this thesis. Later, we derive the goals of our work based on these research questions.

RQ1: If a change is applied to a model belonging to a particular view of a software system, how can a systematic approach be used to propagate this change using dependency relations to identify a subset of required regression tests?

RQ2: How can different types of dependency relations between models of various views and tests be identified, recorded prior to impact analysis, and used to propagate changes to the tests, with the aim of finding impacted tests?

RQ3: How can simple and complex changes in models be defined and used to realize various change scenarios to initiate model-based regression testing?

RQ4: How can test models be generated from existing specification and design models to enable a test baseline providing coverage of various aspects of the *test view*?

RQ5: How can various parts of test models, affected by changes, be classified to distinguish between various types of regression tests?

RQ6: How can extensibility and flexibility be integrally supported to address the evolution of models and to generalize the approach for a wide range of modeling and test specification languages?

1.2 Scope

The scope of this thesis is limited to business processes, which are used to demonstrate various concepts presented in the thesis. We do not consider other domains like embedded systems or communication intensive systems. However, we believe that the concepts presented in the thesis are applicable to general software systems and can be adapted and extended to support other domains as well.

The focus of the thesis is further on using dependency relations to perform impact analysis across models and tests to select a subset of test cases for regression testing. Therefore, the thesis focuses on dependency relations across models representing different views of business processes. The thesis covers the models that represent *structural view*, *behavioral view*, and *process view*. Moreover, the thesis do not cover aspects related to source code, configurations, or deployment artifacts. Similarly, the approach presented in the thesis uses test models to express the *test view*. Although, test code is also used to some extent during the experimental evaluations, the approach do not provide support for automated test code generation. Thus, the test code used during the experimental evaluation is manually developed. Moreover, the focus of this work is on the selection of a test subset only and we do not aim for the test execution and

test result analysis techniques. These techniques are more relevant to the test code generation, which is not in the scope of our current work.

Different metrics, such as *cost* of test cases, *fault severity*, and *test execution history*, can also be used to identify the test cases required for regression testing. However, the approach presented in the thesis do not address such metrics, as we believe that they can be used for the prioritization of test cases and not for the selection of affected tests. To select the affected test cases, it is sufficient to analyze the change propagation through dependency relations. Similarly, other problems relevant to regression testing, such as test minimization and test prioritization, are also out of scope of this thesis.

1.3 Thesis Goals

The thesis aims to provide a structured and systematic model-based regression testing approach to answer the research questions presented above. The approach aims to reduce the test effort by selecting a subset of test cases and enables early testing by using models of various views and test models. Thus, an overall goal of the thesis is stated below as *Initial Goal* IG1.

IG1: Develop a *flexible* model-based regression testing approach to *reduce test effort* by selecting a subset of tests corresponding to the changes *earlier in software development*. The approach shall provide *better coverage* of models of various views and various test aspects. It shall use *dependency relations* between models of various views and tests to assess the impact of *changes* in models on tests for *selection and classification* of the affected tests.

The sub-goals of the thesis, from IG3 to IG9, adhere to the overall goal IG1 and cover the issues discussed earlier. The sub-goal IG2, however, is concerned with adaptation of the approach to business processes. We demonstrate the applicability of the presented solutions by adapting them to the domain of business processes, as depicted by IG2.

IG2: Adapt the approach and proposed solutions to business processes by extending them to support models, test models, change types, and dependency relations specific to business processes.

The foremost prerequisite of regression testing is the availability of an existing *baseline test suite*. To generate the *baseline test suite* for testing business processes, model-based and model-driven test generation approaches are required to be evaluated for their ability to cover various test aspects. These aspects include *test architecture*, *test behavior*, and *test data*. The rules to generate various test aspects are also required to analyze the correspondences between system models and test models.

IG3: Select/Develop a suitable method to systematically generate *baseline test models* for business processes to cover various test aspects.

To define and express changes in models, a generic, consistent, and unified representation of changes is required. It requires an investigation of existing change representations and taxonomies to analyze their support for various change types. This analysis shall focus on identifying any potential inconsistencies in the existing taxonomies and

synthesize them to enable a consistent and unified representation of changes applicable to models. Moreover, changes shall be defined and expressed for models of various business process views to realize different change scenarios.

IG4: Examine possible changes applicable to various models and define a unified and consistent change taxonomy to represent these changes. Apply the change taxonomy to define changes in the models of various business process views.

To analyze the impact of changes in various models on tests, an inherent support of various types of dependency relations is required to propagate changes through these dependency relations. It demands for understanding, identifying, expressing, and recording dependency relations of different types among various system and test models. To record these dependency relations, existing approaches for detecting and recording dependency relations are required to be analyzed for their ability to support system models and test models. These dependency relations are then required to be used to perform impact analysis by examining the change propagation through them. Further, potential solutions to record dependency relations and analyze the impact of changes in models on tests shall be extended for the domain of business processes.

IG5: Find/identify dependency relations between models of various views and tests and select appropriate methods to record these dependency relations.

IG6: Identify/Develop an appropriate method to propagate changes through the set of recorded dependency relations.

The selection of the affected test elements alone is not enough and a further classification based on the type of effect is required. Thus, the affected test elements are required to be classified to demonstrate if they are obsolete, required for retest, or unaffected. Such a classification of the affected test elements requires analysis of affected elements for conditions under which they belong to a specific classification. These conditions include constraints, such as the type of applied change, the type of the affected test element, and the status of related test elements.

IG7: Select a suitable classification scheme and define conditions to distinguish between different types of regression tests by examining the type of change and the status of other affected elements.

As discussed earlier, *flexibility* is a required characteristic of the approach to support the evolution of system modeling and test specification languages. The approach as well as tool support for the approach should enable flexible and extendible design to support easy integration of new models and test specification languages.

IG8: Develop inherently *flexible* and *extensible* solutions and corresponding tool support to enable the evolution of modeling and test specification languages.

To evaluate the quality of the solutions provided by the approach, it is required to assess various characteristics of the approach by developing and applying appropriate metrics. To do so, the approach shall be exercised on a case study from the domain of business processes and analyzed against various metrics to assess its quality. The case study is introduced in Section 2.3 of Chapter 2.

IG9: Evaluate quality of the presented solutions by developing appropriate metrics,

applying the approach on the selected case study, and analyzing the results of the case study to evaluate the metrics.

1.4 Contributions

The main contribution of the thesis is a holistic model-based regression testing approach, which identifies the required subset of regression tests by analyzing dependency relations among models and tests. Consequently, it facilitates the *forecast of the required effort* for regression testing at an early stage in the software development life cycle. Our approach is based on the hypothesis that the selection of test cases depends

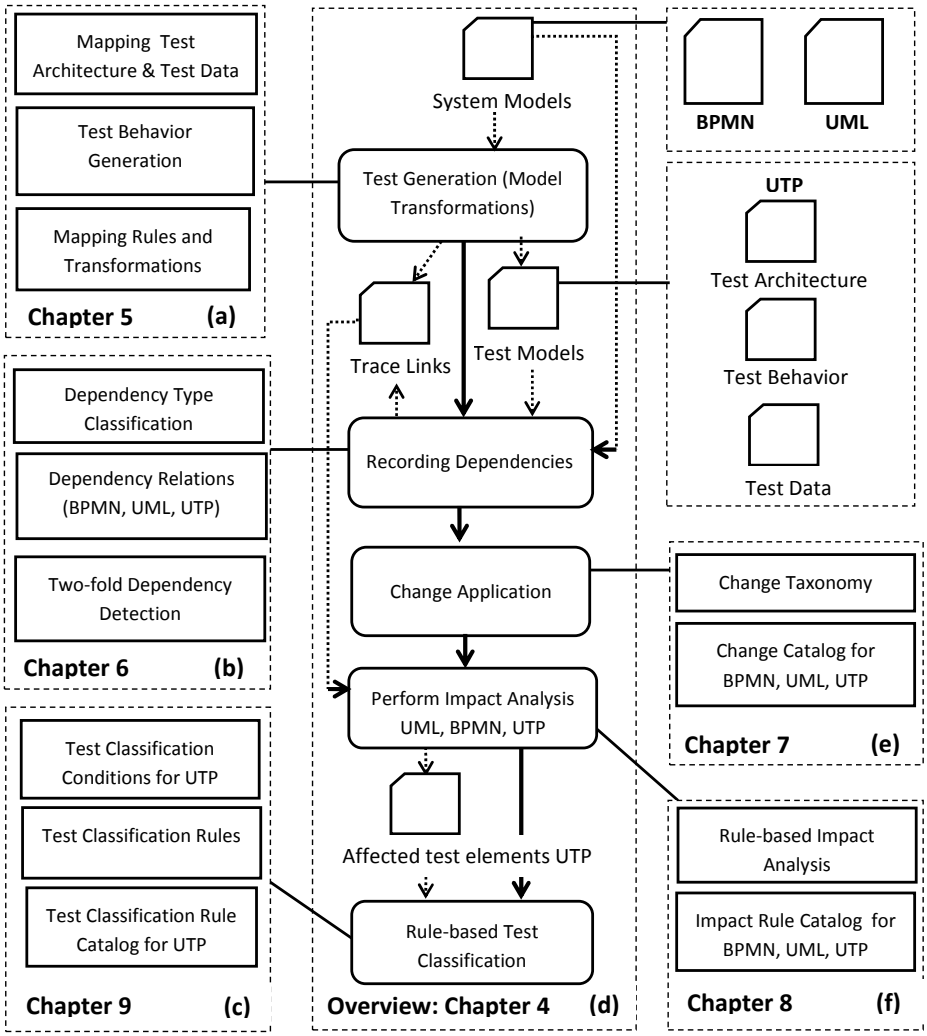


Figure 1.1: A Structured Overview of Thesis Contributions.

on the type of the impact, which is determined by evaluating the interplay of change types and dependency relations among the models of various system views and the *test view*. Thus, our approach integrates, reuses, and extends various approaches to record dependency relations, perform impact analysis, and classify tests. Figure 1.1 depicts an overview of the contributions in accordance with the structure of this thesis

and its middle part (d) depicts a simplified overview of our approach.

To enable the recording of dependency relations, our approach uses a two-fold approach, that is, during the test generation and using dependency detection rules. Our Contributions dealing with this aspect are also depicted in part (b) of Figure 1.1. Our approach enables the recording of dependency relations prior to the impact analysis, which is *less computational extensive* as compared to the approaches that perform repeated search of dependency relations for each change during the impact analysis. Furthermore, our approach provides *better coverage of the dependency relations* among models and tests by eliciting a comprehensive set of dependency relations of various types of models and tests. Thus, we presented a general classification of dependency relations of various dependency types. The classification is based on the purpose of dependency relations, identified by analyzing the context in which the dependency relations are required. We elicit 92 different types of dependency relations among UML, BPMN, and UTP models representing the *structural view*, *process view*, and *test view*. The corresponding dependency detection rules are also developed to detect these dependency relations.

A further contribution of our work is to develop a change taxonomy for a *unified and consistent representation of changes*, which helps to define and express the changes in the models belonging to various views. The contributions relevant to this aspect are also presented in part (e) of Figure 1.1. The change taxonomy defines two categories of changes; *atomic* changes representing the primary changes like add and delete, and *composite* changes representing more complex ones like move or merge. We adapted the change taxonomy to define changes in UML class diagrams, UML component diagrams, BPMN collaboration diagrams, and UTP test models. Thus, we identified a comprehensive list of 140 changes applicable to these models.

Change impact analysis between models and tests is achieved using a rule-based impact analysis approach to propagate the impact of changes through dependency relations among system models and test models. We originally developed this approach to support impact analysis across heterogeneous software artifacts based on the notion of various change types and dependency relations. However, it is required to determine the impact of changes on tests for the selection of affected tests. Therefore, the rule-based impact analysis approach suits in this context as well. The rule-based impact analysis approach uses the *interplay of dependency relations and change types* and provides a recursive impact propagation through dependency relations. We developed a comprehensive set of change impact rules to react on various changes in UML, BPMN, and UTP models. These contributions are also shown in part (f) of Figure 1.1.

Another relevant contribution of our work is to *enable the classification of the affected tests* by developing the concept of test classification rules. The contributions relevant to the test classification are presented in part (c) of Figure 1.1. The classification rules are able to specify various conditions on the affected test elements and impact reports produced during the impact analysis. When all the conditions stated in a rule meet, the rule classify the specific test element according to the given classification type. We use the classification types of Leung and White [LW89] for this purpose, as they cover all the required types for the test classification. Thus, we classify tests as `Obsolete`, `Reusable`, `Retestable`, `PartiallyRetestable`, or `New`. For the classification of UTP test models, we analyzed them for various classification conditions and developed

a set of test classification rules to classify various UTP test elements.

A further contribution of our work is to *generate the required baseline test suite*, which is a prerequisite for regression testing. The contributions relevant to this aspect are depicted in part (a) of Figure 1.1. Our model-driven test generation approach targets business processes and uses UML class diagrams, component diagrams, and BPMN collaboration diagrams to generates various test models expressed in UTP. The generated test suite consists of various UTP models representing the *test architecture*, *test behavior*, and *test data* [UTP11]. Thus, our approach provides *better coverage of test aspects* as compared to other state of the art business process testing approaches. We developed a set of mapping rules to express the correspondences between BPMN, UML, and UTP models. These mapping rules are then implemented as model transformations to support automatic test generation. As discussed earlier, dependency relations between models and their corresponding tests are also preserved during the test generation.

To demonstrate the applicability of our approach in practice, it is complemented by tool support provided by VIATRA Test Generation (VTG) tool and EMFTrace. VTG automates our baseline test generation approach using Visual Automated Model Transformations (VIATRA) framework. Whereas, EMFTrace is an eclipse based tool-set that was initially developed for rule-based dependency detection. It is later extended to support rule-based change impact analysis and rule-based test classification by reusing the existing rule processing infrastructure of EMFTrace.

Finally, the ideas presented in the thesis are evaluated by applying our approach on a case study from joint academic and industrial project. The case study automates the business processes relevant to field service technicians on mobile platforms. We analyzed our approach to evaluate the *precision*, *recall*, *reduction*, and *coverage* of our approach and our findings yield promising results by showing an average reduction of test cases by 46% achieved with an average precision and recall of 93% and 87% respectively. The approach also provides an average coverage of test elements up to 21%, which is significantly better than a name similarity-based approach, which provide 5% coverage when applied on the same change scenarios.

1.5 Thesis Structure

The remainder of the thesis is structured as follows. *Chapter 2* presents preliminaries and fundamental concepts, defines the problem of regression testing, and discusses various aspects of the problem. It also introduces *Field Service Technician* case study, which automates the business processes of field service technicians on mobile platforms. *Chapter 3* presents an analysis of the state of the art in the field of model-based regression testing by developing and applying a comprehensive evaluation criteria. It also presents analysis and discussions on various topics relevant to this thesis.

Figure 1.1 highlights the chapters, which contain the core contributions of the thesis. These contributions are presented in the former section as well. The contributions of a particular chapter are highlighted using dashed boxes in Figure 1.1. Thus, *Chapter 4* presents our holistic model-based regression testing approach, discusses various activities of the approach, and presents its adaptation to business processes. *Chapter 5-9*

correspond to the core activities of the proposed approach. These are the generation of a *baseline test suite*, the application of changes on models, recording of dependency relations, performing impact analysis, and classifying regression tests. These are already discussed in detail in the former chapter and are also highlighted in Figure 1.1.

Chapter 10 discusses the tool support provided by our approach and discusses how we used and extended EMFTrace to support our approach. It also presents VTG our baseline test generation tool for business processes. *Chapter 11* presents the experimental evaluation of our approach on the *Mobile Field Service Technician* case study and presents the results of our approach and discusses various validity threats. *Chapter 12* finally concludes the thesis by revisiting the contributions of our work, discussing the critical issues, and presenting the future directions.

2

Fundamentals and Preliminaries

This chapter presents the fundamental concepts by defining the regression testing problem and discussing various aspects relevant to the problem. The discussion on these preliminary concepts helps to understand the nature of the problem, provides an insight to the relevant issues, and helps to establish a foundation for our state of the art analysis.

2.1 Model-based Regression Testing Problem

Rothermel *et al*. [RH94a] define the regression testing problem as: "find a way making use of T, to get confidence in the correctness of $P^{'}$". Where $P^{'}$ is a modified version of a program P, and T is the test suite for P. Thus, the definition of Rothermel *et al*. focuses on the use of an existing test suite to test a modified program. Yoo and Harman [YH10] further classify the regression testing problem into three sub problems: *test minimization problem*, *test prioritization problem*, and *test selection problem*.

The *test minimization problem* focuses on removing the redundant test cases from the original test suite T to find a reduced subset $T^{'}$. The *test prioritization problem* focuses on an ordering of the test cases according to some desirable properties, such as the rate of fault detection, risk, and cost. Finally, the *test selection problem* focuses on the identification of the test cases relevant to the modified parts of the system.

In this thesis, we focus on the selection of tests affected by the changes propagating through dependency relations, hence we deal specifically with the *test selection problem*. The *test minimization problem* and the *test prioritization problem* are out of scope for our work.

Test Selection Problem– is defined by Yoo and Harman [YH10] as follows:

Given: a program P, a test suite T to test P, and the modified version of P, $P^{'}$.
Problem: Find a subset of $T, T^{'}$ to test $P^{'}$. (2.1)

The above presented definition is similar the one presented by Rothermel *et al*. [RH94a]. It focuses on the selection of a subset of test cases from a baseline test suite to test a modified program. The above presented definitions consider the source code of a program under test and the regression testing approaches of Rothermel *et al*. [RH94a] and Yoo and Harman [YH10] are also based on program source code.

Model-based regression testing is different from the code-based testing as it uses analysis and design models to identify changes in a software system instead of the program source code. These changes are then traced to the tests corresponding to the modified parts of the system. Thus, it is required to define the regression testing problem with a focus on models by incorporating the fundamental issues relevant to the models. Therefore, we define the model-based regression testing problem by adapting the definition of Yoo and Harman [YH10] and extending it to incorporate the notion of *models*, *changes* in models, and *dependency relations* among models.

Given: a Software System S defined by a set of Models S_M.
Given: a baseline test suite Tdefined by a set of Models T_M.
Given: a set $C = (c_1, c_2, ..., c_n) \mid \bigwedge_{1}^{n} c_i \in C$ is a change applicable on any model in M.
Given: a set $D = (d_1, d_2, ..., d_n) \mid \forall d_i \in D$ is a dependency relation between elements of S_M and T_M.
Problem: for any given $c_i \in C$, find $T_M^{'}$, by identifying elements of T_M affected by c_i using D. (2.2)

According to the problem definition presented above, model-based regression testing requires a set of system models representing various views of a software, a baseline test suite, a set of changes applicable to the models, and a set of dependency relations between various models. The problem is to use the set of dependency relations to assess the impact of any change $c_i \in C$ on the tests and find a subset $T^{'}$ of the baseline test suite T to test the modified system. This problem definition explicitly integrates the dependency relations, which are required to propagate the impact of a change c_i to the tests.

Selection of a subset of test cases for regression testing is required, even if, all the test cases for a modified version of a system can be regenerated using an automated tool support without substantial effort. The reason is that the time required to configure, execute, and analyze the test cases can still be saved by executing a subset of tests instead of the complete test suite. Similarly, for the integration and system level testing, it is very challenging to fully regenerate the tests, due to the complexity of test cases, number of required test components, test drivers, and test stubs.

Test Classification Problem– focuses on the identification of the affected test cases for regression testing. However, to distinguish between different types of affected test cases and to decide whether these test cases are valid or invalid for the modified system, a further distinction among the affected test cases is required. To facilitate such a distinction, Leung and White [LW89] introduced a test classification scheme, which covers different types of test cases required to be distinguished for regression testing. A significant number of regression testing approaches use this classification scheme [BLY09, FBH+10, NR07], as the classification scheme supports all the required classification types to distinguish between the affected tests. According to the classification of Leung and White [LW89], the test cases in T should be classified into `Obsolete`, `Reusable`, `Retestable`, and `New` test cases for regression testing in $T^{'}$.

As the name suggests, `Obsolete` test cases are no more required and should be removed from T'. The `Reusable` test cases are unaffected from the current changes; thus to be kept in $T^{'}$. However, they should not be re-executed during regression testing. The `Retestable` test cases are the ones affected by the changes. They should be included in $T^{'}$ and should be re-executed for regression testing. Finally, the `New` test cases are the ones which are to be added in $T^{'}$ as they correspond to the newly introduced elements of the system.

Since the test selection alone is not enough and classification of test cases inside T is required to understand the nature of regression test cases, we define the *test classification problem* as an important aspect to support regression testing. The *test classification problem* considering the classification scheme of Leung and White [LW89] is defined as follows:

Given: a test suite T.
Problem: $\forall x \in T$ decide if $x \in$ `Obsolete` $\vee$ `Reusable` $\vee$ `Retestable` $\vee$ `New`. (2.3)

According to the definition presented by the Equation 2.3, for a given baseline test suite T, the test classification problem is to decide how every test element x in T is classified either as `Obsolete`, `Reusable`, `Retestable`, or *New*.

Test Classification Problem for MBRT– is defined by Equation 2.5 in the context of models by adapting the test classification scheme of Leung and White [LW89].

Given T defined by $T_M, IR = r_1, r_2, r_3...r_m),\ let\ (O, U, R, P, A).$ (2.4)

Find $T^{'}$ defined by $T^{'}_M = (x_1, x_2, x_3, ..., x_n) \bigwedge_{i=0} n\ x \in (O \vee U \vee R \vee P \vee A).$ (2.5)

As presented in Equation 2.5, the test suite T is defined by a set of models T_M, a set of *affected elements* IR, which is produced after performing the impact analysis for the selection of the affected test elements. Further, the set O refers to the set of `Obsolete` elements, which are no more valid for $T^{'}$, thus should be removed for T'. The set U

represents the set of `Reusable` elements in $T^{'}$, which are not affected by a change. R is the set of `Reusable` elements, which are affected by the change and should be used to retest the system and should be included in $T^{'}$. The set P represents the `PartiallyRetestable` elements. We introduce the notion of `PartiallyRetestable` to address the composite test elements, such as *test components*. A *test component* might consist of various *mock* or *stub* operations. If one of these is `Retestable`, the *test component* will be then *PartiallyRetestable*.

Thus, an element x is `PartiallyRetestable`, if atleast one of its constituents is `Reusable` and atleast one of its child elements is `Retestable`. In the case of a `PartiallyRetestable` element, x should remain in $T^{'}$, whereas its affected constituents should be updated and used during regression testing.

Finally, A is the set of elements that are required to be added in $T^{'}$ to update it. The type of the element x and the affected elements relevant to x determine how x will be classified. Each element in UTP has to be analyzed to define the conditions under which that element can belong to either O, U, R, P, or A.

2.2 Problem Analysis

The above presented definitions of regression test selection (Equation 2.2) and classification (Equation 2.3) reflect various important aspects. These include, the use of *models* to express various views of a system, a *baseline test suite* to represent the test aspects, *dependency relations* among models, and the notion of *changes*. An understanding of these aspects is necessary to develop the foundations of the solutions to address the problems presented above and to outline the major requirements for our analysis of the state of the art model-based regression testing approaches. Moreover, we use business processes as an application domain to demonstrate the applicability of the concepts presented in this thesis. Thus, these aspects are required to be discussed for business processes as well to address the domain specific requirements.

2.2.1 The Role of System Views and Models

A software system can be perceived by different perspectives, known as views. IEEE standard 14700 [IEE00] defines the term view for the software systems as follows:

> **View:** A representation of a whole system from the perspective of a related set of concerns.

Both terms *view* and *viewpoint* are used interchangeability to describe a *software view* in the literature. The IEEE standard itself maps a *view* to exactly one *viewpoint*. Hence, we use the term *view* throughout this thesis to describe a software *view* defined by a specific *viewpoint*.

According to Hilliard [Hil99], a view can be characterized by its purpose, scope, and constituent elements. Thus, the purpose of modeling a view is important to understand the view, define its scope, and decide the required models to express it. In the

following, we present a list of views and the purposes they serve to emphasize the role of various views in the software development. Some of these views also exist with a different name in the literature but serve the same purpose, thus they are combined according to the purpose they adhere. The details of the relevant literature from which we extracted these views is presented in Section 3.2.5 of Chapter 3.

1. **PV0: Organizational View–**To describe business strategies and organizational structures [PE00].
2. **PV1: Requirement & Conceptual View–**To elaborate the software requirements, use cases, conceptual entities, and their interactions [Gom11, HNS99].
3. **PV2: Process View–**To express high level processes of the system, their interactions, the roles and participants involved, and the services used by processes [Kru95, PE00].
4. **PV3: Structural View–**To model the structure of the resources, data, modules, high level components, and functions of the system [PE00, Gom11].
5. **PV4: Behavioral View–**To demonstrate the functional behavior of a module and interactions between modules [PE00].
6. **PV5:UI View** To model user interfaces, navigation among various UI entities, and UI components of a system [KKCM04].
7. **PV6: Implementation View–**To represent the source code of the software and various implementation platforms [RCVD09, HNS99].
8. **PV7:Allocation View–**To describe the configurations of the software, the physical components on which the software would be deployed, and the allocation of software components to the physical components [Cle10].
9. **PV8: Test View–**To model the *test architecture*, *test behavior*, *test data*, and various other test related aspects of a system [UTP11].

Models to Express Views– Model-based development requires various models to express different software views. Thus, each of the above presented views can be expressed using one or more models belonging to various modeling languages. As an example, we take class diagrams, which are widely used to model *structural view* of a software system in different domains. They are used to model classes, their constituent operations, attributes, the interactions among various classes, the hierarchical relationships between classes, and composition of various classes etc. Since class diagram model the structural aspects of a system, they belong to the *structural view* [PE00]. Similarly, the *structural view* also consists of high level software components, which can be modeled using UML component diagrams.

Views and Models for Business Processes– Business processes require a high level representation of processes, where processes might also use services provided by various participants and interact with various other processes. Thus, a *process view* is required to model the high level processes and their interactions. Various approaches use UML activity diagrams to model the flow of the processes and their interactions [PE00, EFHT05].

In their approach for modeling the architecture of business information systems, Penker *et al*. [PE00] proposed four different views for modeling business processes. These are, *vision view*, *process view*, *structural view*, and *behavioral view*. The *vision view* can be mapped to the PV0 and PV1 presented above. In this thesis, we focus on the *structural view* and *process view*, as these views express the core design of a business processes.

Penker *et al*. [PE00] as well as many other business process modeling approaches [SDE+10, EFHT05] modeled the *structural view* using UML class and component diagrams. A *process view* is required to model the high level processes, their interactions, the participants which interact with processes, and the services provided by the participants. Penker *et al*. modeled the *process view* using UML activity diagram, where the activity diagram describes the flow of a process and its various tasks. The activity partitions model various participants of the process.

Another alternative to model the business processes is using Business Process Modeling Notation (BPMN) collaboration and process diagrams [BPM10], which are widely used to model business processes. In this thesis, we model the *process view* using BPMN collaboration diagrams and the *structural view* using UML class and component diagrams. Both UML and BPMN are standards from the *Object Management Group* (OMG) [OMG14].

2.2.2 Cross View Dependency Relations

Although different models are used to model several views of a system, the concepts used by the models cannot be completely isolated from each other. Overlapping and reuse of various concepts in the models belonging to different views results in dependency relations among models and views.

Example Dependency Relations from Business Processes– As an example, we take the previous example of the *structural view* and the *process view* from the modeling approach of Penker *et al*. [PE00]. Penker *et al*. [PE00] model various business resources as classes in a class diagram. The business processes use business resources defined in the class diagrams to model the data used by the processes. In this way, a business resources is defined inside one model and is used in another model. This suggest a dependency relation between the business resources belonging to the *structural view* and the process data belonging to the *process view*.

In the context of regression testing, tests representing the *test view* of a business process consist of test cases, test data elements, test components, and other test elements. Test data can be extracted by analyzing the process data, as also discussed later in Section 5.2.4 of Chapter 5. Thus, the test data elements indirectly relate to the business resources defined inside the corresponding class diagram. If a business resource is changed in the class diagram, this change would affect the process data, which in-turn affects the test data. Consequently, test cases using the test data would also be affected and they are required to be analyzed for potential side effects.

Another example is of a `Participant` of a process. A `Participant` provides various services to a process. However, a `Participant` can also be defined as a component in a UML component diagram [SDE+10], which is a case of overlapping of

concepts. Thus, it also suggests a dependency relation between the *Process* using the *Participant* and the corresponding component. Hence, all such dependency relations are required to be understood, made explicit, and used to propagate changes and identify the affected tests.

2.2.3 The Notion of Change

Changes are drivers and triggers for regression testing. Thus, they are required to be understood to commence various change scenarios during regression testing. Changes can be simple, such as deleting an `Attribute` from a `Class`, or they might be more complex, such as merging two classes into one. Thus, identifying and distinguishing between various types of changes is required to realize various change scenarios.

Distinguishing between the simple and complex change types is necessary to asses their impact on various models and tests. To understand the changes, various properties of changes are required to be analyzed. These include the nature of change, its complexity, and its applicability on models based on its granularity. To do so, it is required to investigate the existing change classifications and change taxonomies and assess them for the above mentioned properties.

In the context of model-based regression testing, the information about changes is traditionally extracted from the models belonging to different views of a system. Thus, the changes are obtained from analysis and design models by comparing two versions of a model. These changes are then used to perform impact analysis among models and tests. Thus, it is also required to analyze the capability of existing model-based regression testing approaches for the change support they provide.

Example Changes in Business Processes– To support model-based regression testing of business processes, changes in various models belonging to the business processes are to be identified, analyzed, and defined. Since we focus on the *structural view* and the *process view*, the changes in models belonging to these views are required to be considered. The changes in *structural view* include the changes in the elements of UML class and component diagrams, such as components, classes, services, etc. The changes in the *process view* consists of the changes in BPMN collaboration diagrams, such as processes, tasks, participants, etc.

Some examples of primitive changes are addition and deletion of classes, attributes, and operations. Renaming classes, components, and other elements is another example of a primitive change. Examples of complex changes include moving of a service from one component to another or swapping a service call in a process with another service.

2.2.4 The Test Baseline Using Model-Based Testing

The definitions of the test selection problem (Equation 2.2) and the test classification problem (Equation 2.3) inherently require a baseline test suite from which a subset has to be selected for regression testing. To generate a test baseline representing the *test*

view, model-based testing (MBT) approaches require a set of models of different views of a system to generate tests [DNT08].

Various models are used to generate test cases for different testing levels. This also results in correspondences between various models and the test suites generated from them. These corresponding elements also help to establish dependency relations between the system models and tests [NZR09]. Thus, for the applicability of model-based regression testing approaches, it is important to answer two fundamental questions. (1) How the baseline test suite is generated? (2) What are the constituents of baseline test suite? or how the baseline test suite is represented?

Representing Test Baseline– For a complete test specification, the representation of various aspects of a *test view* is required, such as *test architecture*, *test behavior*, and *test data*. This ensure the *better coverage* of various test aspects during regression testing. Test specification languages, such as TTCN [TTC13] or UTP [UTP11], are used for this purpose.

Recent developments in model-based and model-driven testing support the test specifications in the form of test models [UTP11]. The use of test modeling languages helps to specify the tests on higher level of abstraction. Consequently, the test design activity is performed before the implementation of test cases. The specification of tests in the form of models provides several benefits, such as *portability and interoperability* resulting from higher level of abstraction, *better comprehension* due to the visual modeling, *reduced training costs* due to their similarity to UML which is a widely accepted modeling language [RW03, LZG05]. For model-based regression testing, the test models provide better traceability between the system models and test models, as they belong to the same level of granularity. Thus, the existing model-based regression testing approaches are required to be analyzed for their support of various aspects of the *test view* and test modeling and specification languages.

Generating Test Baseline– Besides the test representation, the baseline test generation method is also of crucial importance. It is of crucial importance to answer various questions regarding the baseline test generation method. For example, how the baseline tests are generated? Which models are used to generate the test baseline and which test aspects are covered? Which algorithms are being used to generate the test cases. There is a need to analyze existing model-based regression testing approaches to assess their ability to answer these questions.

To represent the *test view* of business processes, the questions of how baseline test suite is generated and represented are required to be answered. The *test view* of a business processes should cover various cases of process execution defined by the processes test cases. Moreover, the interaction of various test components to simulate the services requires by the processes is also required to be modeled. Moreover, test data required for the execution of different test cases should also be modeled.

The information from the models belonging to various views can be used to identify these different test aspects using model-based testing approaches. For this purpose, correspondences between the elements of models of *structural view* and *behavioral view* and *test view* are required to be identified. Our proposed baseline test generation approach for business processes is discussed in detail in the Chapter 5.

For the representation of various test aspects, we use UTP [UTP11], which is standard test modeling language from OMG [OMG14]. The factors due to which we selected UTP for the test specification are its wide acceptance in industry and academia, the support available in the form of documentation, its ease of implementation in the form of a UML profile, ease of use as familiarity with UML is enough to understand the UTP notation, and finally its coverage of various test aspects.

2.3 Introduction to Mobile Field Service Technician Case Study

In this section, we introduce the *Mobile Field Service Technician* case study, which we use throughout this thesis as an example and also for the evaluation of our own approach. The case study was developed as part of a joint academic and industrial research project *Adaptive Planning and Secure Execution of Mobile Processes in Dynamic Scenarios* (MOPS) [MOP12]. The aim of MOPS project is to bring the business processes in the world of mobile devices. One of the goals of the MOPS project is to develop several case studies, which require the automation of business processes on mobile platforms.

The *Mobile Field Service Technician* case study is one of the case studies developed as part of the MOPS project. The main goal of the *Mobile Field Service Technician* study is to automate the core business processes relevant to the field service technicians, such as *planning*, *preparation*, and *execution*, of field service orders. Moreover, the business processes related to the *management of field tours*, *management of tools and spare parts*, and *resource scheduling* are also of crucial importance.

An example of one of these processes is the planning of a field tour, as presented in the following.

Problem Demonstration on HandleTourPlanningProcess– Tour planning is an important business process of *Mobile Field Service Technician* case study. We would refer to this process throughout this thesis for the demonstration of various concepts. A *service technician* uses a hand-held mobile device to plan field tours to cover various *service orders*. A mockup of the mobile device used in the project, showing a field tour, is depicted in Figure 2.1. Figure 2.1 shows a field tour (Tagestour), which consists of a list of service orders (Auftragsliste) to be handled at different times and venues. A service technician can plan a field tour based on several strategies, such as the shortest route from the start to the destination, maximum coverage of service contracts from the start to the destination, or the coverage of only high priority services.

The case study has undergone a lot a changes throughout its course and evolved to enhance the processes to cover new scenarios, meet the expectations of various project partners, and to improve various existing processes. Here, we briefly discuss one of the changes from the above presented scenario to motivate the need of our approach.

A yet unresolved functional error in the system demands replacing an existing service with the new one in a process. However, for the replacement of this service, several changes are required in various models. The participants providing services to the pro-

Figure 2.1: The Mobile Device Depicting a Field Tour. (In German)

cess are represented as components in component diagrams [SDE+10]. The services provided by these components are represented as operations of the classes implementing the component interfaces. Test cases testing the process also requires these services during the test execution. Moreover, mocks and stubs might simulate these services to assists the test cases.

If a service has to be replaced, all such dependencies are required to be understood and utilized to find the affected tests. Thus, there is a need to answer these two fundamental questions. First is that how many such dependencies exist between the various views discussed above? The second question is that if a change is to be introduced, which test cases are affected due to such dependencies, and how they are affected?

2.4 Chapter Summary

This chapter formalizes the test selection and classification problems to enable model-based regression testing and elaborates the relevant aspects. The problem definitions inherently include the fundamental aspects of using models of various system views, dependency relations among models, and the notion of change. Since a test baseline is a primary artifact that is used for regression testing, we also discuss the importance of the baseline test generation and test representation using test modeling language UTP.

These preliminary and foundation concepts help us to understand basic requirements to enable MBRT. Thus, provide a clear road map to evaluate the state of the art MBRT approaches in the next chapter. Further, we introduced our *Mobile Field Service Technician* case study, which automates business processes on mobile platforms and is used throughout this thesis for the concept demonstration.

3

Analysis and Evaluation of the State of the Art

We divided this chapter into two main sections. Since the scope of this thesis is limited to model-based regression testing, Section 3.1 presents a through evaluation of the state of the art model-based regression testing approaches by using a comprehensive evaluation criteria. Section 3.2 analyzes the literature relevant to other topics covered in the various parts of this thesis.

3.1 Evaluation of Model-based Regression Testing Approaches

We first present various research questions, which are required to be answered to evaluate the state of the art MBRT approaches.

Research Questions for Evaluation– The evaluation research questions presented in the following are derived from the research questions presented in Chapter 1 and the concepts discussed in 2.

1. **ERQ1:** Which views and models are covered by the approaches? (see Section 2.2.1 of Chapter 2)
2. **ERQ2:** What testing level is addressed by the approach?
3. **ERQ3:** What is the application domain of the approach? (see Section 1.2 of Chapter 1)

4. **ERQ4:** How the baseline test suite is generated and represented? (see RQ4 in Section 1.1 in chapter 1)
5. **ERQ5:** Are dependency relations supported and used for impact analysis? (see RQ2 in Section 1.1 of Chapter 1)
6. **ERQ6:** Which types of dependency relations are used by the approaches (see RQ2 in Section 1.1 of Chapter 1)
7. **ERQ7:** How dependency relations are recorded? (see RQ2 in Section 1.1 of Chapter 1)
8. **ERQ8:** Which simple and complex changes in models are considered? (see RQ3 in Section 1.1 of Chapter 1)
9. **ERQ9:** How the changes are identified and recorded? (see RQ2 in Section 1.1 of Chapter 1)
10. **ERQ10:** How the affected test elements belonging to various parts of test suites are obtained and classified? (see RQ5 in Section 1.1 of Chapter 1)
11. **ERQ11:** How easy it is to enhance and evolve the approach? and to which extent extensibility is supported? (see RQ6 in Section 1.1 of Chapter 1)
12. **ERQ12:** Which standards are supported for modeling, test specification, and tool implementation?
13. **ERQ13:** Whether automation and tool support is provided for the approach or not?

3.1.1 Evaluation Criteria for Model-based Regression Testing Approaches

This section presents the evaluation criteria we developed to answer the above presented research questions. The evaluation criteria is presented in Figure 3.1 and consists of 5 main criteria. Each criterion is further composed of several sub-criteria, which correspond to a set of inquiries. An inquiry is a distinct question that has to be answered to evaluate a specific criterion. The five main criteria are *Scope*, *Coverage*, *Core Methodology*, *Applicability*, and *Extensibility*.

Scope and Coverage– criteria provide answers to the questions *ERQ1-ERQ3*. The criterion *Scope* further consist of two sub criteria; *Domain* and *Testing Level*.

Coverage– is the criterion which provides information about various views covered by the approaches and the input and output models required by the approach.

Core Methodology– is the most significant criterion as it addresses the core MBRT issues, such as support for test baseline, changes, dependency relations, and the test classification of the approach under evaluation. This criterion addresses the research questions *ERQ2...ERQ8*.

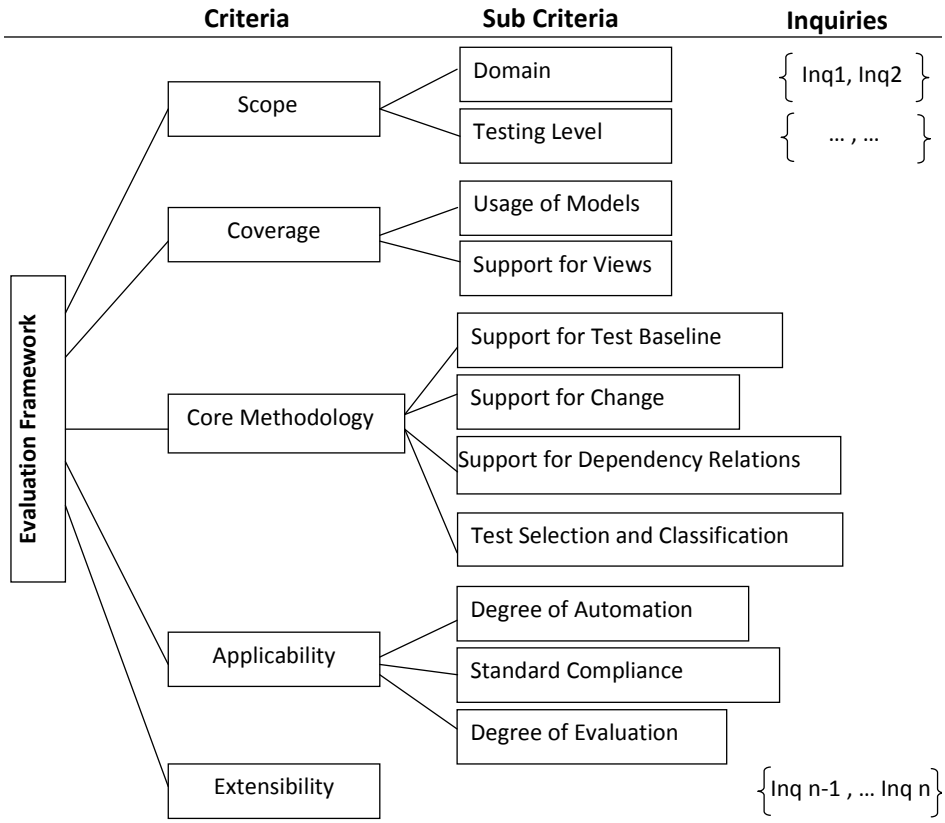


Figure 3.1: The Evaluation Criteria of MBRT Approaches.

Applicability– is the criterion that covers the aspects to determine whether the approach under consideration is applicable in practice or not. Thus, it evaluates the standard compliance, the automation support, and the extent to which evaluations are performed.

Extensibility– is the criterion which evaluates the flexibility of the approach and the ability to extend it with minimal effort. Table 3.1 provides the set of inquiries for each criterion and their sub-criteria presented in Figure 3.1.

Table 3.1: The Inquiries Corresponding to Criteria

Criteria	Sub Criteria	Inquiries
Scope	Domain	**Inq.1:** The approach is suitable of which types of the systems?
	Testing Level	**Inq.2:** What is the testing level addressed by the approach?
Coverage	Support For Views	**Inq.3:** The approach covers which of the views? (structural, behavioral, others) **Inq.4:** Which aspects of the *test view* are covered by the approach?
	Use of Models	**Inq.5:** What are the input models used by the approach? **Inq.6:** Does the approach require any additional inputs other than Models?
Core	Support For Test Baseline	**Inq.7:** How are the tests expressed? **Inq.8:** What is the base line test suite generation method?
	Support for Change	**Inq.9:** Does the approach discusses the change detection mechanism? **Inq.10:** Whether the approach provides sound change definitions for modifications in the system?

		Inq.11: How many change types are considered by the approach?
	Support for Dependency Relations	**Inq.12:** What type of dependency relations are supported? **Inq.13:** How the dependency relations are recorded? **Inq.14:** Does the approach considers dependency types? **Inq.15:** How dependency-relations are stored?
	Test Selection and Classification	**Inq.16:** Does the approach provide logic of test case selection and classification?
Applicability	Degree of Automation	**Inq.17:** Were the ideas defined by some algorithmic details or not? **Inq.18:** Does the approach provide tool support or not? **Inq.19:** What is the implementation platform for the tool if implemented?
	Standard Compliance	**Inq.20:** Is the input of approach compliant to any standards? **Inq.21:** Is the approach compliant to any test specifications standard?
	Degree of Evaluation	**Inq.22:** Is the approach evaluated on any case study or does any experimental evaluation was present?
Extensibility		**Inq.23:** Does the approach rely on a specific change identification method? **Inq.24:** Is it easy to extend the impact analysis logic? **Inq.25:** Dependency relations recording and impact analysis is tightly coupled or not?

3.1.2 Evaluation of Model-based Regression Testing Approaches based on the Evaluation Criteria

To evaluate the model-based regression testing approaches, we first select a set of model-based regression testing approaches and then evaluate them for each inquiry presented in Table 3.1. For the selection of the approaches for the evaluation, we used the following exclusion criteria.

Study Selection and Exclusion– We do not consider the studies that use source code or formal specifications as input. We only evaluate the approaches which takes visual analysis and design models as input to perform model-based regression test selection. In total we selected 18 studies published in 26 research papers which fulfill this criteria.

Evaluation based on Criteria– We used the Inq.1 to group the selected studies based on their application domain, as presented in the form of tables in Appendix A. Thus, a large number of approaches are object oriented and UML-based. A significant number of approaches are specific to state-based systems and a small number of approaches belong to the component-based and specification-based systems. The domain of business processes is, however, neglected by the model-based regression testing approaches. Although, there are other code-based regression testing approaches for business processes, which are presented in Section 3.2.1.

We present the cumulative evaluation results of all other inquiries in the form of bar charts for a better visualization in Figure 3.2, 3.3, and 3.4. The detailed results of other inquiries are also presented in Table A.1, A.2, and A.3 in Appendix A. The x axis of the bar charts shows the results for each inquiry presented in Table 3.1 and y axis depicts the percentage of the approaches which adhere to a certain answer of the respective inquiry. The various possible answers of an inquiry are represented as a distinct cate-

gory.

The answers of *Inq.2* shown in Figure 3.2 depict the percentage of the approaches that adhere to a certain testing level. The larger set of approaches targets the unit testing level. However, a reasonable number also addresses integration and system level regression testing. The results of *Inq.3* reveal that the support of *structural view*, *behavioral*

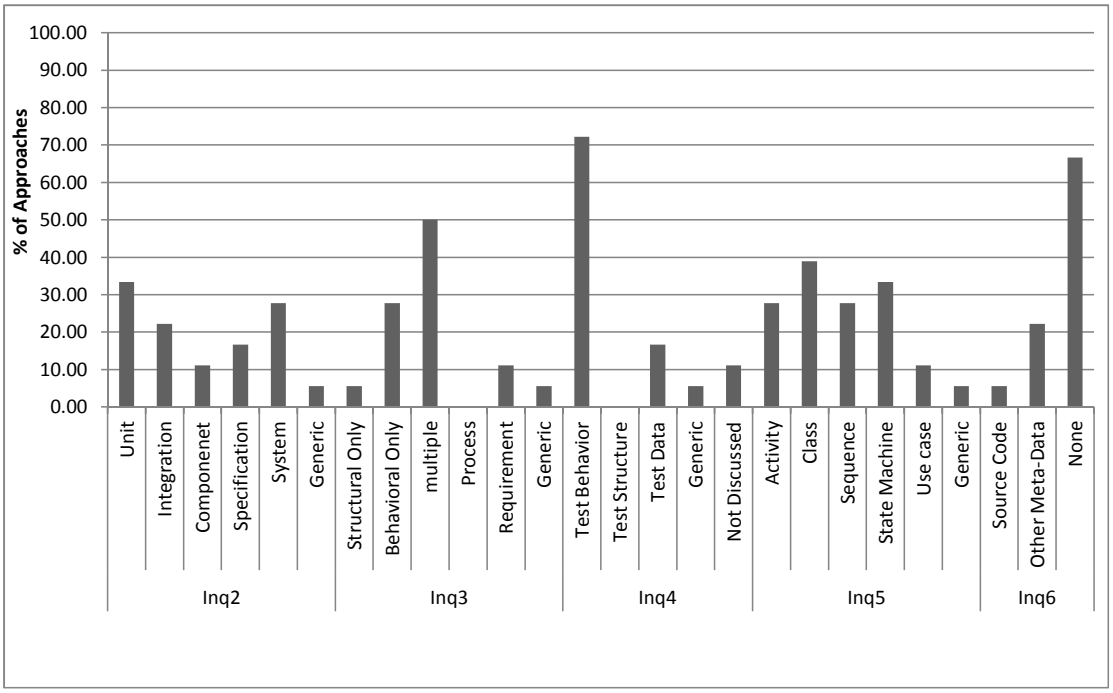


Figure 3.2: The Evaluation of MBRT Approaches for Inq. 2 to Inq.6.

view, and *requirement view* is already available to some extent in the exiting model-based regression testing approaches. However, the *process view*, which is crucial for business processes and in our context is not supported. Support of some other views presented in Section 2.2.1 in Chapter 2, such as *allocation view*, *implementation view*, and *UI view*, is also not available. However, these views target later phases of software development life cycle and though important for overall development life cycle, are not relevant to model-based regression testing.

It is to be noted that the approach of Zech *et al*. [ZFKB12] is generic and can support EMF-based models. This work is significant for us, as our approach is also generic for EMF-based models. However, the work of Zech *et al*. [ZFKB12] is based on OCL queries, which are more complex compared to our rule-based approach. Similarly, we use the notion of dependency relations and complex change types, which is not present in their work.

The results of *Inq.4* depict that most of the MBRT approach only support the *test behavior* and a few of them also support *test data*. However, this support is limited to the data constraints only. The *test architecture* is not considered at all. The approach of Zech *et al*. [ZFKB12] can be used to support UTP test models but their ideas are only limited to a number of small examples. A systematic approach to cover all these views is lacking in the state of the art approaches.

The results of *Inq.5* and *Inq.6* summarize the input data and format used by the approaches. The results of *Inq.6* show that most of the MBRT approaches use the information from models only and do not require any other meta-data as input. The additional meta-data on the one hand might improve the accuracy of the results but on the other hand it might be an additional overhead to acquire this data.

According to the results of *Inq.7* shown in Figure 3.3, the conformance of the approaches to the test specifications standards is lacking. As discussed earlier, this can result in the poor quality of test baseline and also makes the application of approaches difficult in practice. Moreover, according to *Inq.8*, a significant number of approaches do not provide any information about their baseline test generation methods. Most of the approaches, which use path-based algorithms for test generation are also only for unit level testing. For other complex testing scenarios, such as during integration and system testing, it is crucial to know the structure of the test baseline and how it can be generated.

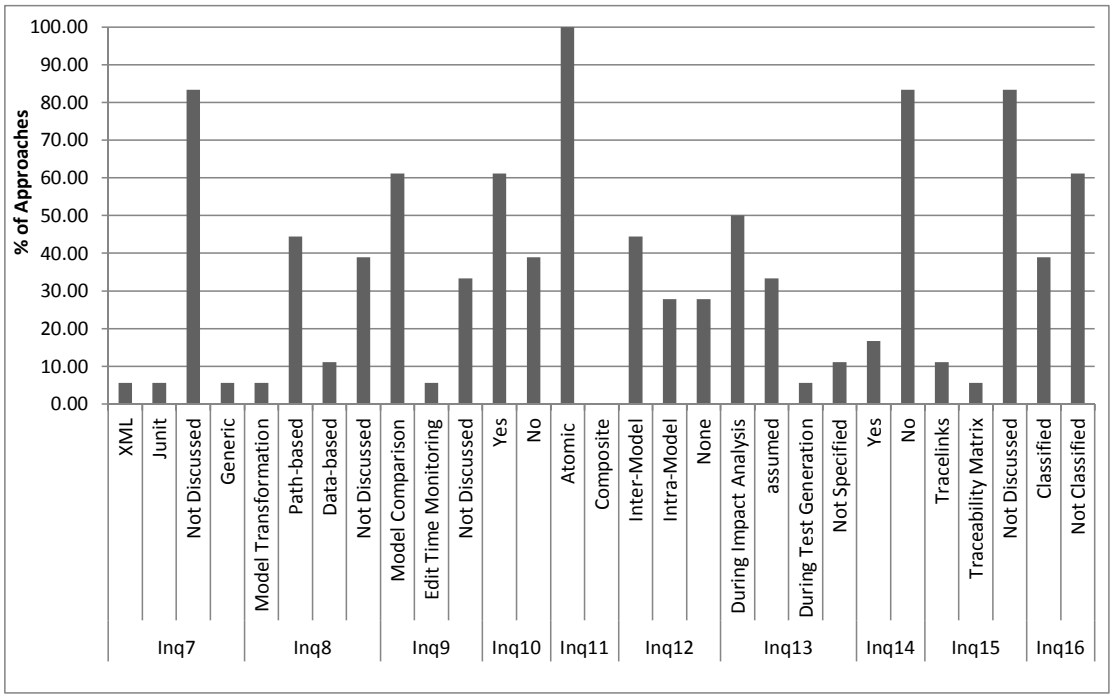


Figure 3.3: The Evaluation of MBRT Approaches for Inq.7 to Inq.16.

The inquiries *Inq.9-Inq.11* focus on the change support. According to the results, most of the existing approaches either rely on model comparison or do not discuss the change identification method explicitly. Further, only the atomic changes are considered by the approaches and no complex changes are supported. Moreover, the concrete change specifications are also not provided by many approaches.

The inquires *Inq.12-Inq.15* investigate the support of approaches for dependency relations. According to the results, although a number of approaches support inter-model dependencies, they are not made explicit and cannot be reused. Moreover, if the dependency relations are supported to an extent, for each change they are repeatedly searched compromising the efficiency of the approaches. A solution is to record them

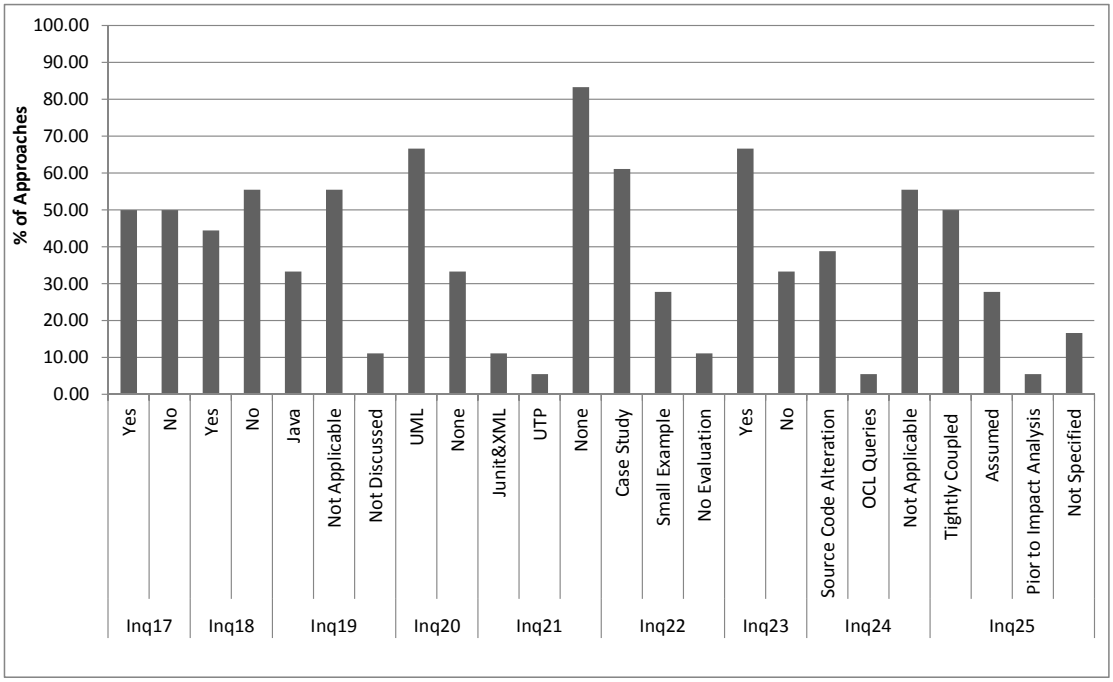


Figure 3.4: The Evaluation of MBRT Approaches for Inq. 17 to Inq.25.

prior to the impact analysis, as done by Naslavsky *et al*. [NZR09] and our approach presented in this thesis. Further, none of the approaches distinguish between various types of dependency relations.

For the test classification, the results of *Inq.16* show that a number of approaches discuss some mechanism to classify tests. Since they cover only the *test behavior* aspect, these classifications cannot be considered complete, as discussed earlier.

The inquiries *Inq.17-Inq.19* analyze the aspect of tool support. About half of the approaches provide some algorithmic details or prototype implementations of their tools. Java is the most used implementation platform. The inquiries *Inq.20* and *Inq.21* evaluate the approaches for their support of standards. A number of approaches use UML for system modeling. However, as discussed earlier as well, standard compliance for test specification is very limited. Only a few approaches use JUnit to express tests and most of the approaches only rely on self improvised notations.

The inquiry *Inq.22* presents the degree of evaluation provided by the approaches. The results are promising and a large number of approaches provide some basic evaluation of their approaches. However, an empirical evaluation of various approaches one one or more similar case studies can be an interesting direction for the further research in the area of model-based regression testing. The inquiries *Inq.23-Inq.25* evaluate the approaches for extensibility. The results of *Inq.23* show that most of the approaches rely on state-based change detection or model comparison for their approaches. Hence, they depend on some tool to compare the state of the models.

However, the result of *Inq.25* show that all the approaches require the changes in source code to extend the rules for performing change impact analysis. This is a rigid solution and require a lot of effort to extend the tool support if new dependency relations

and scenarios are identified or new modeling or test specification languages are required to be integrated. Finally, the results of *Inq.25* show that only one approach, the approach of Naslavsky *et al.* [NZR09], records the dependency relations prior to the impact analysis and all the other approaches repeat the dependency detection for each change. As discussed earlier, this requires more time and efficiency of the approach is compromised.

Limitations of the State of the Art– To summarize the above discussion, we state the limitations of the state of art in the following points briefly.

1. Limited support for baseline test generation
2. Limited support for complex changes
3. Limited support for dependency relations and types
4. Lack of support for the *process view*
5. Limited support for test modeling languages and test views
6. Repeated search of dependency relations
7. Limited extensibility
8. Non-compliance to standards

3.2 Analysis of State of the Art in Other Relevant Areas

In this section, we present the state of the art relevant to various areas covered by this thesis in the following subsections.

3.2.1 Analysis of Business Process-based Regression Testing Approaches

A number of business process-based regression testing approaches use process code, such as BPEL for regression test selection [WLC08, LQJW10, LLZT07]. They start the test selection activity after the changes are already implemented. Hence, an early forecast of the required effort and an early start of testing activity are not possible. Moreover, they overlook the dependency relations across views, which might also produce less accurate results.

A number of approaches in the literature only target the unit testing of web services used by the processes [RT07b, RT07a, KH09, TFM06]. However, they do not address the changes and their impact on the process tests.

Some research in the field of impact analysis also focuses on the relationship of business processes with other artifacts. Ginige *et al*. [GG09] consider the relations between BPEL processes and the WSDL web service specifications. Since we do not use process code for regression testing, these dependency relations cannot contribute to our work.

Wang *et al*. [WYZS12] use dependency relations between the process layer and the service layer for impact analysis. They only consider one type of dependency and in comparison, we support a comprehensive set of other dependency relations between processes, services, components, and test suites.

3.2.2 Analysis of Test Generation Approaches for Business Processes

This section covers the state of the art approaches for the test generation of business processes, as we generate our test baseline using our proposed model-driven test generation approach as part of our work.

Bakota *et al*. [BBG+09] and Heinecke *et al*. [HGGF10] use an activity-like notation for process specification and test generation. Bakota *et al*. [BBG+09] use category partition method, whereas, Heinecke *et al*. [HGGF10] use a path-based approach for test derivation. However, these test generation approaches cannot be applied on BPMN processes as they have their own syntax, semantics, and many additional activity types requiring additional investigation. Moreover, BPMN supports the concepts of events, whereas, the model used by these approaches is purely data-based.

Schiefer *et al*. [SSS06] presents a framework which supports test generation for event based processes. The concept of event based testing is valuable for us but the paper focuses only on the execution framework and not on the test generation aspects. Werner *et al*. [WGTZ08] use WSDL specifications for test generation. They only consider interfaces of the processes, thus generate only black box test cases, and overlook control or data flows of the system.

Yuan *et al*. [Yua08] present a model-driven approach for test generation for business processes. Our approach also uses same foundations as Yuan *et al*. [Yua08] but their work is only an initial idea and lacks details about test generation activities, various mappings, mapping rules. Moreover, the focus of their work is only on test architecture generation and other aspects, such as test behavior generation and test data generation, are missing.

Apart form the above presented analysis, most of the above discussed approaches do not consider a holistic view of the system for the test generation They use a single artifact, such as process specification in the form of a graph or an activity diagram. However, as discussed earlier, the information for different test views can be obtained from the artifacts representing different views of business processes. Thus, there is a need for a test generation approach that uses the information from various views of business processes and generate various test aspects.

3.2.3 Analysis of Change Classification Schemes

In this section, we discuss the change classifications used by regression testing, impact analysis, and some other general change classifications schemes. Further, we discuss

the approaches dealing with BPMN collaboration diagrams, as we use them as an application domain.

Changes Covered for Regression Testing– Changes covered for regression testing are already discussed earlier in Section 3.1.2. Our evaluations conclude that most of the MBRT approaches use only *add*, *delete*, and *modify* change types. Some tools for regression testing of JUnit classes, such as *JUnit-CIA* by Störzer *et al*. [SRRT06] and *Chianti* by Ren *et al*. [RST+03, RSTC04], also use atomic change operations like *addition* and *deletion* of classes to determine affected test cases.

Changes Covered for Impact Analysis– We briefly discuss here the change types considered for impact analysis. A detailed discussion on these approaches can be seen in one of our works [LFR12]. A number of existing impact analysis approaches for source code use atomic changes, such as addition of elements, deletion of elements, and changes in data values and properties of elements [KGH+, FG06, SLT+10]. However, the major issue with all these works is lack of support for composite changes. Further, all the changes they consider are in the context of source code.

Various approaches for impact analysis across models support change types to express model changes. Briand *et al*. [BLOS06] also consider changes in various UML models, such as class diagrams, sequence diagrams, and use case diagrams. However, the set of change types they support is also limited to the atomic change types. The approach of Xing and Stroulia [XS04a, XS04b, XS05] is based on differencing of UML class models, with the overall goal of detecting class co-evolution patterns. The change types they support are *add*, *delete*, *move*, *rename*, and *signature changes* of classes, methods, and attributes. Although *move* is supported, however, many other composite changes types are neglected by their work.

Other Change Classification Schemes– The change taxonomy of Buckley *et al*. [BMZ+05] distinguishes between two major categories of changes; structural and semantic change types. The structural changes cover the addition, subtraction, and alteration of software entities. The concept of distinguishing between structural and semantic changes is interesting but we only focus on structural changes and do not consider semantic changes in our work.

The work of Mäder [Mäd10] on traceability maintenance supports elementary change operations, such as addition of elements, deletion of elements, and property modifications. Some composite developer activities they support consist of *replace*, *merge*, and *split* operations. Baldwin and Clark [BC00] propose a set of atomic operations to modularize a software by application of various refactorings. The authors distinguish between six distinct refactorings: *splitting*, *substitution*, *augmenting*, *excluding*, *inversion*, and *porting*. Their proposed set of operations contains a mixture of atomic and composite types. However, the atomic operations are incomplete, as the modification of properties is not covered. The set of composite operations lack the *merge* and the simple *move* operators.

Changes Covered for BPMN Collaboration Diagrams– Gerth *et al*. [GKLE10] presented an approach for detection of changes in the business process models for conflict resolution of changes. They considered three change operations, that are, *insert*, *delete*, *move* for detecting changes in process models. Weber *et al*. [WR08] proposed various change patterns to enable refactoring of business processes.

However, these change patterns can be realized by sequence of atomic and composite change types presented in this thesis. Since process refactoring is out of scope for this thesis, we do not consider these change patterns in our work.

3.2.4 Analysis of Impact Analysis Approaches

In this section, we briefly discuss the impact analysis approaches in the literature. A detailed discussion on the impact analysis approaches can be found in one of our works on rule-based impact analysis [LFR13b].

Typical impact analysis approaches are focused on only one type of software artifact, such as source code, or certain UML diagrams. For example, a call graph analysis cannot be applied on requirement specifications. Due to this limitation, most of the proposed approaches are not able to detect impacts in heterogeneous software artifacts. Their applicability in the context of regression testing is also difficult due to this. A previous study [Leh11] reveal ed, that from 150 studied impact analysis approaches only 19 are able to analyze atleast two types of artifacts. Various techniques have been proposed to assess the propagation of impacts, such as program slicing [VBF07], call graph analysis [RT01], analysis of execution traces [OAH03], impact analysis using similarity match algorithms [BLOS06, KSD09, KD11], information retrieval [ACCDL00, PMFG09], and the mining of software repositories [YMNCC04, ZWDZ05].

Ibrahim *et al*. [IIMD05] proposed an approach for impact analysis spanning classes, packages, tests, and requirements. To perform impact analysis, traceability relations are established between the artifacts based on similar names, domain knowledge, and explicit relationships. However, they provide semi-automated traceability analysis only, which limits the applicability of their approach. They further neglect UML, BPMN, and UTP models, thus are not applicable in our context.

Lindvall *et al*. [LS96, LS98] also use vertical and horizontal traceability relations for impact analysis. De Lucia *et al*. [DLFO08] discussed the need of traceability support for impact analysis and the two main problems of traceability analysis, that are, link recovery and link evolution.

Most impact analysis approaches are further limited by the amount of change types they support. They treat different types of changes equally and assume that they result in the same consequences. Only few works, such as the approach of Keller *et al*. [KSD09, KD11], treat different types of changes separately during the impact analysis process. However, their set of change operations is not comprehensive either.

A closely related area to impact analysis is also dependency detection. Imtiaz *et al*. [III08] analyzed various traceability techniques for their support of multi-perspective impact analysis, which is based on various criteria. These are support for horizontal and vertical traceability, support for different types of software artifacts, and their degree of automation. According to results of Imtiaz *et al*., rule-based traceability mining techniques are suitable for dependency detection among the different types of artifacts, which can also be fully automated.

3.2.5 Analysis of Approaches for Support of Software Views

Gomaa [Gom11] presented 7 different views of a system that can be modeled using UML diagrams. The *usecase View* describes the functional requirement of the system, the *static view* describes classes and their relationships, and the *dynamic interaction view* describes objects and communication between them. The *dynamic state-machine view* depicts states and internal control of a component, the *structural component view* defines components and their interconnections, the *dynamic concurrent view* describes communication between distributed components, and finally the *deployment view* defines the configurations of a system.

Razavizadeh *et al*. [RCVD09] use *implementation view* for extracting architectural descriptions from it. Four different architectural views are defined in the 4+1 view model [Kru95]: the *logical view*, the *development view*, the *process view*, and the *physical view*.

The *logical view* defined as an object model using classes and their relationships. The *process view* captures concurrency and synchronization aspects and defines tasks and inter-task communications. The *development view* defines the static organization of a software and its development environment by using components and connectors.

Hofmeister *et al*. [HNS99] use 4 different views: the conceptual view, the *module view*, the *code view*, and the *execution view* to define software architecture. The *conceptual view* of the software define by UML component diagram to define high level components, UML class diagrams to define the conceptual classes, UML sequence diagrams to show component interactions, and state machines to define the protocol for the component ports.

The *module view* defines the decomposition of subsystems into individual modules and is described by UML class diagram. The *execution view* defines the concrete configurations required to execute the system and is defined using UML class and Object diagram. Finally, the *code view* shows how the concrete source code files are organized into directories. Clements [Cle10] discuss *Module*, *Component & Connector*, and *Allocation* views to define the modular structures, components and their relationships and hardware allocations. In the following, we explicitly discuss the views of business oriented systems, as business processes are the application domain of this thesis.

Views of Business Oriented Software Systems– Penker and Eriksson [PE00] present 4 different views to model business architecture. The *vision view* models high level business strategies and business requirements and the *process view* models high level business processes. The *structure view* defines the structure of the business resources, processes, and services. Finally, *behavior view* defines states of various business objects and interaction between processes. Koch *et al.* [KKCM04] use various views to model business processes. These are *UI view* composed of *navigation* and *presentation* views, *process view*, and *structural view*. The *process* and *structural* views are similar to the Penker and Errikson's [PE00] approach.

Ferdian [PE00] describes different views of business process, such as *data view*, *function view*, *organization view*, and *control view*. The control view binds the *function* and *data view*. Moreover, he discusses several other views to describe a software system, such

Table 3.2: Different Software Views and Their Adherence to Purposes

Authors	Purposes						
	PV1	PV2	PV3	PV4	PV5	PV6	PV7
Penker and Eriksson [2000]	Vision View	Process View	Structure View	Behavior View	—	—	—
Koch et al. [2004]	—	Process View	Structural View	—	UIView	—	—
Ferdian [2001]	Organization View		Data View, Fuction View	Control View	—	—	—
Clements [2010]	—	—	Module View Component and Connector View	—	—	—	Allocation View
Hofmeister et al. [1999]	Conceptual View		Module View	—	—	Code View	Execution View
Kruchten [1995]		Process View	Logical View Development View	—	—	—	Physical View
Razavizadeh et al. [2009]	—	—	—	—		Implementation View	—
Gomaa [2011]	Use case view	—	Structural Component View	Dynamic Interaction View Dynamic State Machine View Dynamic Cuncurrent View	—	—	Deployment View

as *usecase view*, *design view*, *process view*, *implementation view*, and *deployment view*.

Another important view neglected by the architecture modeling approaches is *test view*. A number of test modeling and execution languages [HKO07, UTP11, MH03, PM91] are used to represent the test related aspects of a software system. Since our goal is to enable model-based regression testing, the test view of the system is of crucial importance to us.

All the above mentioned views, however, adhere to various purposes, which were introduced earlier in Section 3.2.5 of Chapter 2. Table 3.2 in-lines the above discussed views according to the purpose they serve. Table 3.2 presents how different views discussed in Section 3.2.5 in Chapter 3 adhere to the list of purposes we collected.

3.2.6 State of the Art Business Process Modeling Approaches

This section briefly presents various approaches for modeling business processes and discusses the modeling methods and artifacts supported by the approaches. The UWE approach [KKCM04] uses an activity like model to model processes. Each process itself is modeled as a class with a stereotype «ProcessClass ». The data and resources required by the processes are modeled as classes with a stereotype «entity ». The focus of the approach is to integrate the UI navigation models with the business processes.

The SoaML approach [EtCM$^+$10, SDE$^+$10] models the enterprise business applications by modeling the business architecture models and software architecture models. The business architecture is modeled using BPMN collaboration and process models, and business goals and capabilities. Moreover, a high level service architecture is modeled using UML collaboration diagrams. The service architecture is further expressed as service interface models in class diagrams, service interface behavior using UML interaction/sequence diagrams, and software components using UML structure/component diagrams. The aim of the approach is to integrate both business modeling (BPMN) and service modeling (SoaML) approaches to model service oriented enter-

prise applications.

Engels *et al*.[EFHT05] use UML activity diagrams to model the process behavior. The constraints on the processes are modeled using OCL. The data used by processes is defined as classes inside UML class diagrams, and concrete data instances are modeled using UML object diagrams. The organizational structure is also modeled using UML class and object diagrams. Interactions of business processes are modeled using swim lanes, which is conceptually similar to the Pools in BPMN collaboration diagrams. UML structure/component diagrams are used to model the high level business components and their interfaces.

As discussed in previous section, Penker *et al*. [PE00] present four different views to model business architecture. They use various UML models to model these various aspects, such as UML activity, class, object, sequence, and collaboration diagrams.

Auer *et al.* [AGEG09] modeled user interfaces using dialogs. Other approaches used task models for interface modeling.

3.2.7 Analysis of State of the Art on Test Dependencies

A number of approaches in the literature use dependency relations to support various test related activities. Sneed [Sne04] presented an approach for the reverse engineering of the test cases to support software migration. They used dependency relations among various system artifacts and tests to support software migration. Some of these dependency relations include, subsystem to test procedure, test procedure to test case,test case to use case, and test case to component etc.

Paul *et al.* [PYTB01] presented various categories of dependency relations to support functional regression testing using textual test scenarios. The different categories presented by them are: *functional dependence*, which expresses a relationship between various test scenarios, *input/output dependence*, which expresses common inputs and outputs shared by test scenarios, *persistent data dependence*, if the input/output data of tests is persistent, *execution dependence*, if the execution order of test cases is dependent on each other, and *condition dependence*, if the tests share preconditions/postconditions.

Kuhn *et al*. [KRH+08] talk about dependencies among the unit test cases and discover them for the purpose of fault localization. They use annotations to declare explicit dependencies inside the test cases. Jungmayr [Jun02] discuss the dependencies between components to increase the testability of the components. They suggest to remove certain dependencies to reduce the need of stubs for testing. They call them test critical dependencies but the work does not suggests any new dependency types and only aims to reduce existing dependencies between components to support testing.

Honeyman [Hon82] as well as Johnson and Klug [JK84] discuss the dependencies among database fields and quires for testing the databases. *Control low dependencies* due to conditional statements and *data flow dependencies* due to assignments to program variable are discussed and used for testing program code by various approaches, such as by Leung and White [WL92] and by Podgurski and Clarke [PC89]. Podgurski and Clarke [PC89] also discuss two other types of dependencies for testing program code, that are,

syntactic dependence if a statement is in the chain of control and data dependence, and *semantic dependence*, if they have a semantic relationship which might not be visible by program syntax.

3.3 Chapter Summary

The analysis presented in this chapter shows that the existing MBRT approaches do not provide through coverage of various types of dependency relations among models belonging to various views and tests. Furthermore, for each change, the existing approaches perform a repeated search of dependencies during the impact analysis. Detecting dependency relations prior to impact analysis can significantly improve the efficiency and execution time required for larger systems. Similarly, impact analysis activity is not separated from the detection of dependency relations. Thus, it is not possible to use existing dependency detection approaches with them, which makes the design less flexible.

Moreover, the existing approaches only support primitive changes, such as addition and deletion, and neglect more complex changes applicable to models. In addition, there is no support for test modeling languages and consequently, impact of changes on various test views is also not supported. Similarly, only a limited number of approaches support any standard test specification language. Most of the approaches do not refer to any baseline test generation method. Thus, it is difficult to apply them on practical scenarios. The tools provided by the approaches also lack extensibility, if the modeling languages or test specification languages evolve. Changes in the source code of the tools are required to support new conditions, dependency relations, and new language features, which results in additional effort. Thus, there is a strong need of a new model-based regression testing approach to overcome these issues.

4

Overview of Proposed Model-based Regression Testing Approach

This chapter presents a broad overview of our approach and the activities required to enable our approach. The approach complements the results obtained by our evaluation of the state of the art, presented in Section 3.1.2 of Chapter 3. Our evaluation results demand for a model-based regression testing approach, which inherently supports various types of dependency relations across several system views and the *test view*. These dependency relations are required to be recorded prior to the impact analysis, to abstain from repeated search of dependency relations for each change. The approach shall also provide support for different types of changes applicable to models. Moreover, a desirable characteristic of the approach is inherent flexibility to extend and adapt the approach for different modeling, test specification, and test execution languages. Thus, based on these requirements we establish the hypothesis for our approach in the following.

Hypothesis– The hypothesis of our approach states that a potentially affected subset of tests, after a change is being introduced, can be selected and classified by considering following aspects.

1. The type of the applied change.
2. The type of impact determined by change impact analysis considering three distinct aspects; *type of the applied change*, *type of the element* on which the change is applied, and *dependency relations* relevant to this element.
3. The type of the affected test element determined by the change impact analysis
4. The type of the test classification and classification conditions respective to the type of the affected test element.

Further, our approach supports the concept of rules to enable various activities. These rules are extensible and can be enhanced to support various modeling and test specification languages.

4.1 Proposed Model-based Regression Testing Approach

Our approach follows a systematic procedure by first defining generic steps and activities to enable model-based regression testing. This enables us to discuss the conceptual foundations of our approach, independent of a particular application domain. For the realization of our approach to support domain specific requirements, it is then adapted to the domain of business processes. Thus, the adaptation demonstrates concrete application of the generic concepts. This section presents a general overview of our approach and Section 4.2 discusses the adaptation of our approach for the domain of business processes.

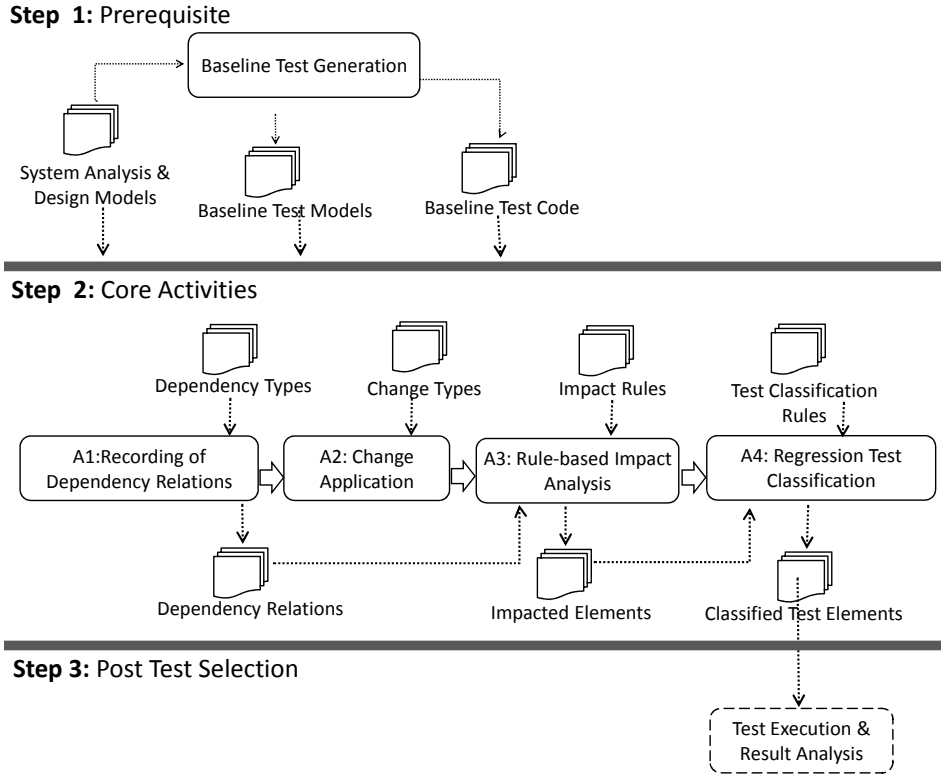


Figure 4.1: Overview of Model-based Regression Testing Approach.

Our approach is comprised of three major steps, as depicted in Figure 4.1. Step 1 provides a test baseline, as the need for a test baseline is evident to support regression testing. Our evaluation of the state of the art exhibits that the baseline test suite should cover various test aspects, which is supported by *Step 1*. *Step 2* in Figure 4.1 enables core regression testing activities. These activities are (1) application of changes on models,

(2) recording of dependency relations, (3) change impact analysis, and (4) the classification of tests for regression testing.

Finally, the last step, *Step 3* in Figure 4.1, depicts the post test selection activities. These activities enable the execution of the classified test cases and analysis of obtained test results. The execution of regression tests and the analysis of the test results can be achieved by using the same test execution platforms, which were used to test the baseline test suite prior to the changes. A number of existing test execution tools can be used for this purpose, such as TTWorkbench[1] . However, test execution is a separate domain and is not in the scope of our current work. Therefore, further discussion on this step in the context of regression testing is not required. In the following, we elaborate on the activities belonging to *Step 1* and *Step 2* to provide further insight to our approach.

4.1.1 Baseline Test Generation

To support the generation of a test baseline from models, it is required to use these models by extracting the test relevant information to generate the test suites. Therefore, we propose a model-driven test generation approach, which uses the system models and transform them to the test models to obtain a set of baseline test models. Model-driven test generation approaches are based on the notion of platform-independent models, which are transformed to the platform-independent test models. As discussed in the previous chapter, we use UTP models to express the *test view* and various aspects of UTP are required to be covered during the test generation.

Thus, to generate the *test architecture* in UTP, the information about the structure of the software system is required, which is obtained from UML class and component diagrams. Similarly, UTP *test behavior* can be extracted by analyzing the control flow in BPMN collaboration diagrams. *test data* in UTP is also required to be generated to provide the test inputs and expected test results. Therefore, we analyze the data used by the processes and various business entities and resources defined in the class diagrams.

Further, in model-driven test generation, mapping rules are required to transform the system models to the test models for the test generation. To support the generation of various test aspects, we analyzed the elements of system models and test models to identify the correspondences and realized these correspondences as mapping rules. These mappings rules are then translated to model transformations for the execution. Further details of our baseline test generation approach and the mapping rules we developed to transform the system models to UTP test models are presented in detail in Chapter 5.

4.1.2 Recording of Dependency Relations

The activity *Recording of Dependency Relations* in Figure 4.1 records the dependency relations between the set of system models S_M and the set of test models T_M prior

[1] http://www.testingtech.com/products/ttworkbench.php

to the impact analysis. There are two major benefits of using this approach. Firstly, various existing approaches can be used for dependency recovery and dependency recording independent of impact analysis [MM03, LFOT07, BLR11]. Secondly, every time a change is applied the required dependency relations are not repeatedly searched during the impact analysis reducing the search overhead.

Thus, we record the dependency relations of various types between system and test models using two different approaches. The dependency relations across system models (S_M to S_M) are recorded using a rule-based approach for dependency detection [BLR11]. The approach was originally proposed for dependency detection between requirements, user goals, and UML design models. However, the approach supports rules, which are able to capture dependency relations based on a set of conditions. These rules are based on a generic structure and can be extended to detect dependency relations for other artifacts as well. We extend the set of dependency detection rules to detect dependency relations between the models belonging to the *structural view* and *process view*.

The dependency relations between system models and test models (S_M to S_T) are recorded during the test generation. As discussed in the previous section, we generate the tests by extracting the relevant information from the models belonging to S_M. During the test generation, dependency relations between source system models and target test models can also be recorded and preserved, as done by Naslavsky *et al*. [NR07]. Therefore, we record the dependency relations between various system models and UTP test models during the test generation.

Moreover, to record various types of dependency relations, a distinction between various types is necessary to identify various classes of relations. This is particularly important later during the impact analysis because different type of dependency relations might result in different type of impacts. Therefore, we also provide a classification of various types of dependency relations, which is based on the purpose of the dependency relations. Chapter 6 discusses various aspects relevant to dependency relations, the classification and types of dependency relations, the approaches to record dependency relations, and dependency relations between BPMN, UML, and UTP models in detail. Once the dependency relations between the models are explicitly recorded, the next step is to apply a change on any model element to assess its impact.

4.1.3 Change Application

The *Change Application* is the second activity of step 2, as depicted in Figure 4.1. In our approach, we use a predefined change catalogue and select the required changes to be applied on a model to start the regression test selection. In this way, we do not require the changes to be implemented first on the models. Therefore, our approach is not based on the model comparison as compared to the existing approaches in the literature. This enables us to deal with complex changes as well. Model comparison is mostly able to detect basic changes in the models, such as add and delete. The complex model changes, such as move and merge, are very hard to detect using the model comparison. Figure 4.1 represents this catalogue as *change types*. The models in the set are required to be analyzed to identify the changes applicable on them. The set of

changes applicable to the set S_M was defined as set C in the problem definition expressed by Equation 2.2 in Section 2.1 in Chapter 2. These changes are required to be well understood, extracted for each required model, and defined unambiguously.

To define various types of changes applicable to models, a unified representation of changes applicable to models is required. To support this, we present a generic change taxonomy, which can be used to represent *atomic* and *composite* changes in models. The taxonomy analyzes various change types from the literature and unifies them for a consistent representation of change types across all the models. Further details of the *Change Application* activity, the change taxonomy, and its application on the domain of business processes is presented in detail in Chapter 7.

4.1.4 Rule-based Impact Analysis

After a change is applied on any model, the next task is to initiate the *Rule-based Impact Analysis* activity, as depicted by Figure 4.1. To analyze the propagation of changes across models, the dependency relations recorded earlier are utilized by the rules.

We initially developed the rule-based impact analysis approach to support impact analysis across heterogeneous software artifacts [LFR13a]. The approach is based on the rules that use interplay to change types and dependency relations to propagate impact of a change. Our hypothesis for the test selection and classification is based on the affected test elements determined by analyzing various change types and dependency relations. Therefore, the rule-based impact analysis approach can be used to analyze the impact of changes in the models on the test models.

According to Figure 4.1, rule-based impact analysis requires a set of rules, a set of predefined change types, and previously recorded dependency relations. Impact rules asses the impacted elements by analyzing various conditions under which a change $c_i \in C$ propagates through dependency relations. Thus, a set of affected elements from the models of S_M and T is obtained. The impact rules produce a chain of reactions and trigger other changes and their corresponding impact rules as well. This process can consume a chain of dependency relations and carries on until no further dependency relations are found for a particular impact rule.

At this point, we already obtain the set of affected test elements, which addresses the *test selection* problem presented in Section 2.1 of Chapter 2. Further details of the *rule-based impact analysis* activity and its application on the models from the domain of business processes is presented in Chapter 8.

4.1.5 Regression Test Classification

The final activity of step 2 shown in Figure 4.1 is *Regression Test Classification*, as the selection of affected test elements alone is not enough and a further classification of affected tests is required. Once, a set of elements affected by a change is available after the impact analysis, it is required to analyze the affected test elements to determine how they are affected? and how they should be used for regression testing?

Thus, to distinguish between various types of test cases for regression testing, we use the classification scheme proposed by Leung and White [LW89]. The baseline test suite T is defined by a set of models T_M and classification of the elements of T_M is required to obtain the final regression test suite $T^{'}$. To classify the elements of T_M, it is required to analyze these elements to define the conditions under which the affected test elements can be classified.

We propose the concept of *text classification rules*, which are based on various classification conditions for a test element, the type of impact produced by the impact analysis activity, and the type of applied changes. The rules analyze these various elements to decide how an affected test element should be used during regression testing, that is, if it is `Obsolete`, `Reusable`, `Retestable`,`PartiallyRetestable`, or `New`. These rules are easy to extend as compared to the existing approaches, which require changes in source code if new test elements are required to be covered or any change in the classification conditions is required [BLY09, FIMR10, NZR10]. The details of the test classification activity and test classification rules are presented in Chapter 9.

The next section presents a scenario to motivate the need of our approach for business processes and how we adapted our approach to business processes is presented in Section 4.2.

4.2 Adapting the Approach to Business Processes

To demonstrate the applicability of our approach, in this section, we present how our approach is adapted to the domain of business processes. Therefore, we first present a motivating scenario, which is taken from the *HandleTourPlanningProcess* from the *Mobile Field Service Technician* case study introduced earlier in Section 2.3. We use this scenario throughout this thesis to explain various aspects of our approach. After that, we define the concrete problem of regression testing for business processes and discuss how our approach can be adapted to provide the concrete for the domain of business processes.

4.2.1 Motivating Scenario for Business Processes

Models to express various views of business processes are prepared during the analysis and design phase of software development and they serve as a working specification of the design of business processes. These views are also discussed earlier in Chapter 2. The views we specifically consider in our approach are presented in Figure 4.2. Figure 4.3 presents the excerpts of the various views of the tour planning process.

Tour Planning Process View– Part (a) of Figure 4.3 presents the *process view* and provides an excerpt of the *HandleTourPlanningProcess*. The process takes *start* and *end location* of the *field tour*, which is to be planned, and plans a tour based on various strategies. The scenario in Figure 4.3 presents only the excerpt of the process for the case when the strategy for planning a tour is based on the *shortest distance*. However, other cases are also possible, such as routes covering maximum number of *service orders*

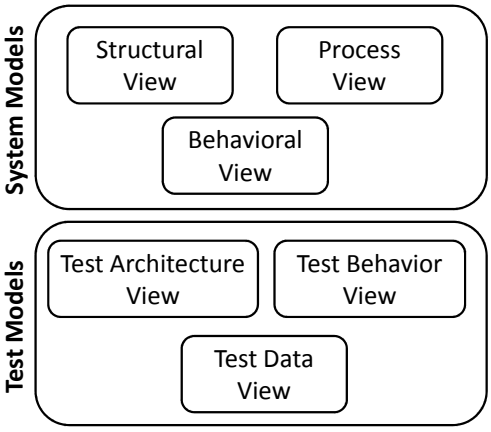


Figure 4.2: Coverage of Views for Business Processes.

or the routes covering only high priority *service orders*.

When a *route* is obtained based on the *shortest distance* strategy, the next task is to obtain the *service orders* for this particular *route* and create a *tour plan*. The *tour plan* can then be customized by the *service technician* and finally the *tour plan* is saved. The tasks *Get Shortest Route* and *Get Service Orders For Route* call two services provided by the `RoutePlaner` and `ServicePlanner`, which are *participants* of the *process*.

Tour Planning Structural View– The part (b) and (c) of Figure 4.3 present the *structural view* of the process. According to part (b) a `Class` is modeled corresponding to the process *HandleTourPlanningProcess* with the stereotype «ProcessClass » and various data elements used by the process are also defined as classes with the stereotype «entity » [KKCM04]. One of the entity classes is `TourPlan` as depicted in part (b) of Figure 4.3.

Similarly, different *stakeholders* and *participants* of the processes are defined as *components* [SDE+10], shown in part (c) of Figure 4.3. Thus, one of the components is `ServicePlanner`, which provides the service `getServiceOrders` to the *HandleTourPlanningProcess*. Thus, a class `ServicePlanningManager` implements the service `getServiceOrders` of the `ServicePlanner` component. Similarly, the component `RoutePlanner` provides the service `getShortestRoute`, which is called by the process. These services can be modeled as operations in the classes implementing the component interfaces.

Tour Planning Test View– The part (d) of the Figure 4.3 depicts an excerpt of the *test view* of the *HandleTourPlanningProcess*. It consists of three test components and a `TestContext`. The `TestContext` contains two test cases, *testShortestStrategy* and *testNoRouteForShortestStrategy*. These test cases are derived from the collaboration diagram depicted in Part (a) of Figure 4.3 using a path-based strategy, discussed in Section 5.2.3 of Chapter 5.

The test components, `RoutePlannerTCom`, `ServicePlannerTCom`, and `HTPPTCom`, simulate the behavior of the actual components from the *structural view*. They simulate the services provided by the components as *mock operations*. How we generate these test aspects using our model-driven test generation approach is discussed in detail in

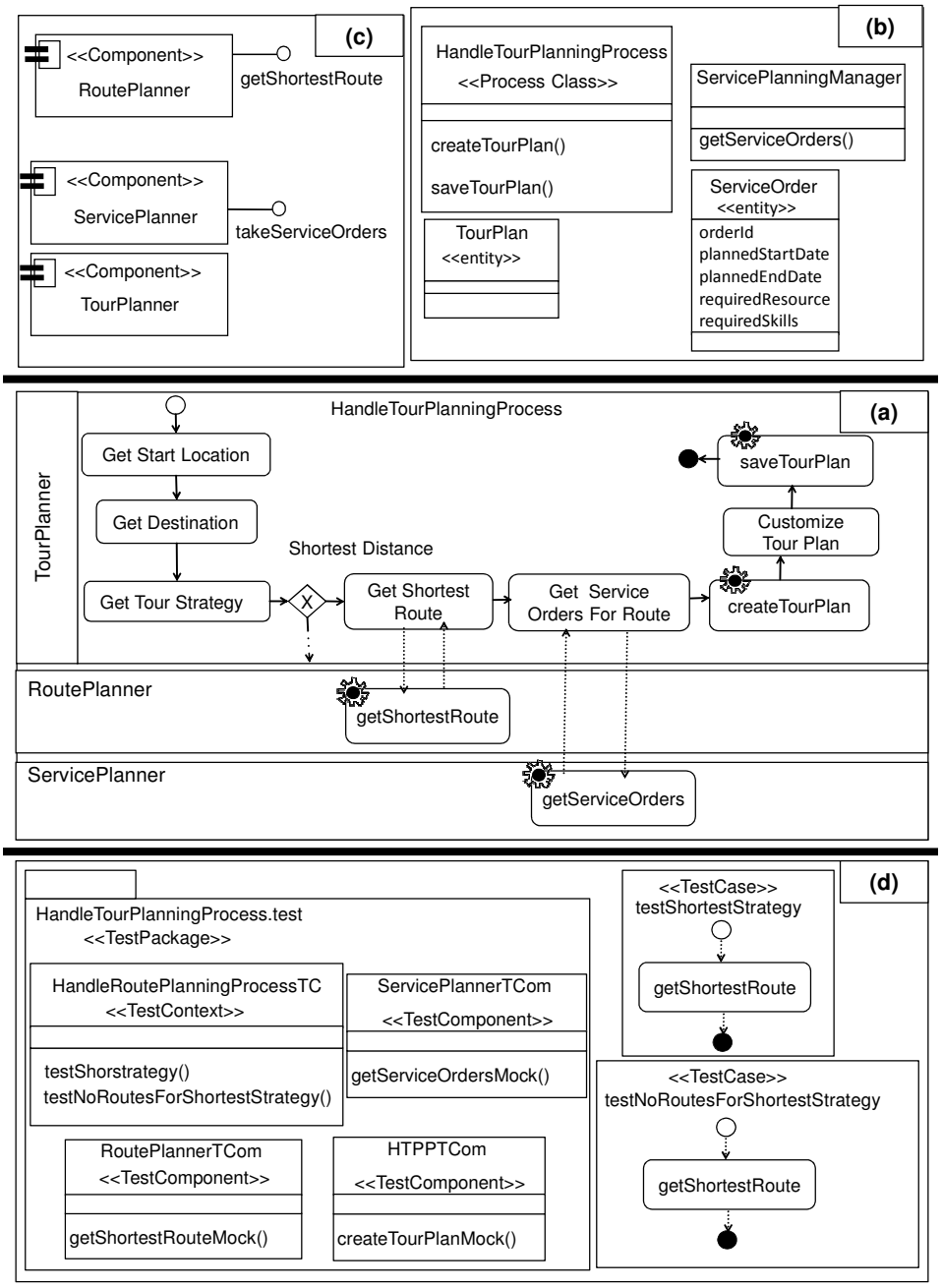


Figure 4.3: A Scenario Representing System Models belonging to Various Views of Business Processes.

Chapter 5.

Dependency Relations for Models of Different Views– From the above presented views, one can already notice some of the dependency relations between the elements of these views, as depicted in the following.

- between `Participant` and `Component`–As the participants in a BPMN collaboration diagram are *equivalent to* the components in UML component diagram.
- between `Task` and `Service`–As tasks *call* services provided by participants.
- between `Component` and `TestComponent`–As the test components *simulate* the behavior of actual components.

These example dependency relations provide an idea of various *types* of dependency relations that exist between models of different views.

Change Scenario and Need for Regression Testing– Considering the above discussed views, we apply the changes and analyze their impact on the *test view* to demonstrate the need of our approach. The operation *getServiceOrder()* from the class `ServicePlanningManager` does not take any argument, however requires a `Route` to compute the *service orders* covering that `Route`. The test component `ServicePlannerTCom` implements a corresponding `MockOperation` which simulates the behavior of the actual *getServiceOrder* operation. Therefore, the impacts on the *test view* would be following:

1. The change would affect the `takeServiceOrdersMock` in the test component `ServicePlannerTCom`.
2. Any test case that uses the `takeServiceOrdersMock` is affected as well, hence in this case both operations in the `TestContext` are affected.

Thus, the problem is to determine that if a change is applied on any of these views, which test elements are affected by the change and are required to be selected for regression testing. Moreover, how the test elements should be classified for regression testing, that is, whether they become obsolete, they are required for retesting, new elements are required, or they remain unaffected.

Both of the test cases presented in Figure 4.3 become `Retestable` and are required to rerun for the regression cycle. There are total 10 test cases in the actual test suite of *HandleTourPlanningProcess* as reported in Section 11.2 of Chapter 11. In the present change scenario, the remaining 8 test cases remain unaffected and are not required to retest.

4.2.2 Adapting Problem and Solution for Business Processes

Considering the scenario presented in the previous section, for the adaptation of our approach to business processes, we first map the problem definition presented in Section 2.1 of Chapter 2 to business processes. Afterward, we discuss how various activities of our approach presented in Section 4.1 are adapted to support business processes.

Problem Definition for Business Processes– Equation 4.1 presents the problem of model-based regression test selection and classification by adapting the

problem definitions presented in Section 2.1 of Chapter 2 and focusing on models of various views of business processes.

Given: $V = (PV, SV, TV)$ a set of different software views.
$PV = (pv_1, pv_2 ... pv_n) \mid \forall\, pv_i \in PV\, pv_i$ is a model expressing the *process view*.
$SV = (sv_1, sv_2 ... sv_n) \mid \forall\, sv_i \in SV\, sv_i$ is a model expressing the *structural view*.
$TV = (tv_1, tv_2 ... tv_n) \mid \forall\, tv_i \in TV\, tv_i$ is a model expressing the *test view*.
Given: a process P defined by $S_M = (b, C_D, C_{OD}) \mid$ type(b)=collaboration diagram $\wedge$
$b \in\ PV\ \wedge$ type(C_D)=class diagram $\wedge$ type(C_{OD})=component diagram $\wedge$
$(C_D \wedge C_{OD}) \in\ SV$.
Given: $T = (T_a, T_b, T_d) \in\ TV$ to test P.
T_a expresses *the test architecture* $\wedge\, T_d$ expresses *the test data*.
$T_b = (b_1, b_2, ..., b_n) \mid \forall\, b_i \in T_b,\ b_i$ is a test case representing *test behavior*.
Given: $C = (c_1, c_2, ..., c_n) \mid \forall c_i \in C,\ c_i$ is a change type for any model in S_M.
Problem: find $T^{'}$ for any given $c_i \in C$ by identifying elements of T affected by c_i using D.
determine $\forall x \in T^{'}$ how x can be classified? (4.1)

According to Equation 4.1, the *structural view* and the *behavioral view* of a given process is defined by class diagram, component diagram, and collaboration diagram. Moreover, the *test view* of the process is defined using a set of models T_a, T_b, and T_d representing the *test architecture*, *test behavior*, and *test data* respectively.

Moreover, a set of changes applicable to the models is defined as C. The problem is that if a change $C_i \in C$ is applied to any of the models belonging to the set S_M, which elements in the models of *test architecture*, *test behavior*, and *test data* will be affected. Further, how these elements can be classified to assist regression testing.

Adaptation of Proposed Approach to Business Processes– The first activity *Baseline Test Generation* in Figure 4.1 takes BPMN collaboration diagram, UML class diagram, and UML component diagram as input. The output of the baseline test generation activity is test models to test the BPMN collaboration diagrams expressed in UTP. Our model-driven approach presented in Chapter 5 details down the test generation using these models, the mapping rules we developed, and the example scenarios from our case study for the concept demonstration.

To enable the second activity *Recoding Dependency Relations* presented in Figure 4.1, dependency relations are required to be recorded between BPMN, UML, and UTP models. The details of how we identified, analyzed, and recorded these dependency relations is discussed in Chapter 6.

The activity *Change Application* presented in Figure 4.1 requires a set of predefined change types for BPMN and UML models. As discussed in Section 4.1.3, we develop a change taxonomy to define the generic changes in models. We used the same taxonomy to define the changes in UML and BPMN models and discuss them in Chapter 7.

To adapt the activity *Rule-based impact analysis* presented in Figure 4.1, impact rules are required to be developed to react on the changes defined for BPMN and UML models. The details of how we developed these rules is presented in Chapter 8. The set of impact rules is also presented in Appendix E. Finally, the last activity *regression test classification* requires the analysis of UTP test models to identify the conditions for classification. Chapter 9 presents the details of the tests classification, the definitions to classify UTP elements and the concept of test classification rules.

4.3 Relation of the Approach with General SDLC

In this section, we briefly discuss the relation of our model-based regression testing approach with general software development life cycle (SDLC). In one of our earlier works, we presented a generic process for regression testing [FR11]. We refine this process in the context of our approach, as presented in Figure 4.4. The upper part of Figure 4.4 depicts the basic SDLC, which is comparable to the traditional waterfall development model [Som10]. Baseline analysis and design models are prepared during the *Analysis* and *Design* phases of SDLC. During the testing phase, model-driven testing is used to generate baseline test models.

Usually, change requests are made during the maintenance phase. After the initial assessment of a change, model-based regression testing can be started to analyze the test effort required for the changes. During the regression testing activity, various activities of our approach are commenced by recording dependency relations, applying change, analyzing the change impact, and finally selecting and classifying a subset of baseline test suite for regression testing. After that the desired change is designed and implemented. The selected subset of tests is then used for the test execution and test result analysis to get the confidence that changes have no adverse effects on the system.

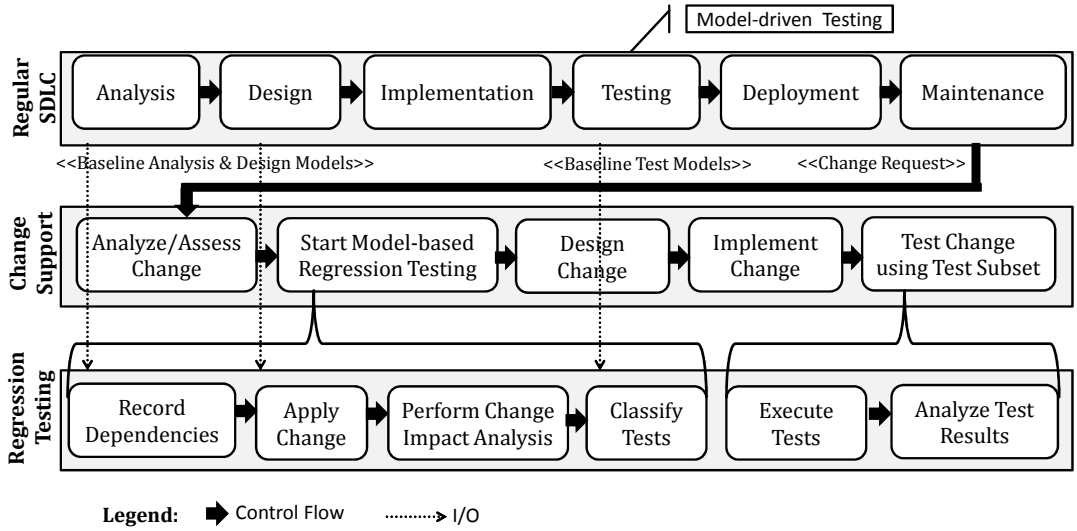


Figure 4.4: Relation of our approach to SDLC.

After a change cycle is complete, it can be repeated to accommodate other changes to support incremental development. The adherence to waterfall SDLC depicted in Fig-

ure 4.4 is only an example of integration of our approach with a basic development process. To integrate it with other widely used development processes, for example RUP [Kru00] or agile process SCRUM [Sch95], the approach shall be adapted accordingly.

4.4 Chapter Summary

In this chapter, we provide an overview of our proposed regression testing approach, which uses dependency relations between system models and test models recorded prior to the regression test selection and classification. The approach consists of three major steps, the establishment of prerequisites, the core activities to support regression testing, and post test selection activities. In this thesis, we focus on the first two steps. The establishment of prerequisites step focuses on the generation of a test baseline using a model-driven test generation approach.

The core regression testing step consists of various other activities: *Recording of Dependency Relations*, *Change Application*, *Rule-based Impact Analysis*, and *Test Selection and Classification*. Dependency relations are used to propagate impact of changes by using a set of impact rules, which can be triggered when a change is applied on a model. The set of impacted test elements are then be analyzed to classify the test cases required for regression testing.

We demonstrate the applicability of our approach by adapting it to the domain of business processes. For this purpose, we first discussed the need of adaptation of our approach by presenting a motivating scenario from *Mobile Service Technician* case study. The activities of our model-based regression testing approach are then analyzed to enable our approach in the context of business processes. The subsequent chapters discuss each activity supported by our approach in detail to provide further insight to our approach.

5

Baseline Test Generation–A Model-driven Test Generation Approach

This chapter presents our model-driven test generation approach to generate a test baseline for business processes, which is one of the contributions of this thesis. Step 1 of our model-based regression testing approach presented in Section 4.1 of Chapter 4 states the baseline test generation as a prerequisite step. The test baseline generated in this step will be used for test selection and classification later in our approach.

Our test generation approach is targeted for the domain of business processes. Testing business processes is essential to ensure the quality of the systems supporting the underlying business processes. Due to the increasing complexity of processes and rapid technological advancement, testing requires high effort and huge investments. Early testing can significantly reduce the testing costs [BDG+07]. Model-driven testing (MDT) of business processes is, therefore, introduced [SWK09, Yua08], which support early testing by deriving the tests from analysis and design models.

Our analysis of the state of the art shows that most of the business process testing approaches derive test cases from process code, for example, *BPEL* [YLY+06, YLS06]. Some approaches also translate the process code into some formal specification, such as PROMELA [GFTR06, ZZK07a, ZZK07b] or petri-nets [DYZ06], and then perform model checking and test derivation. There are three main disadvantages of these approaches.

Firstly, the tests can not expose the deviations of the code from the functional specifications. Secondly, testing activity can only be started after the development is complete, which increases time and cost for testing. Thirdly, various test views discussed throughout this thesis are not supported by these approaches and the focus is only on the *test behavior*. Therefore, a model-driven test generation approach for business processes is needed to overcome these deficiencies of the state of the art approaches.

5.1 Requirements to Enable Model-driven Test Generation for Business Processes

Model-driven testing uses the concept of model transformations to transform the platform independent design models into platform independent test suites [BDG+07]. Thus, there are three major requirements to support model-driven test generation for business processes. These are **(1)** the availability of a platform independent process modeling language, **(2)** the availability of a test modeling language to support test visualization and documentation, and **(3)** the support for model transformations for the test generation.

To meet the first requirement, support for the artifacts required to model different views of process-based information systems is required [PE00]. As discussed earlier, to model the *process view* and *behavioral view*, we use BPMN collaboration diagrams. The *structural view* of the system is modeled using the UML class and component diagrams.

To fulfill the second requirement we use UTP test models for the test specification. UTP supports modeling of several test related aspects, such as *test architecture*, *test behavior*, *test data*, *test time*, and several others. Finally, to support the third requirement, we identify the correspondences between the design artifacts and test models and develop the mapping rules using these corresponding elements.

5.2 Our Approach for Model-driven Testing of Business Processes

To generate the test specifications, we adapt the general model-driven test generation process by Dai *et al*. [BDG+07], as depicted in part (a) of Figure 5.1. This process

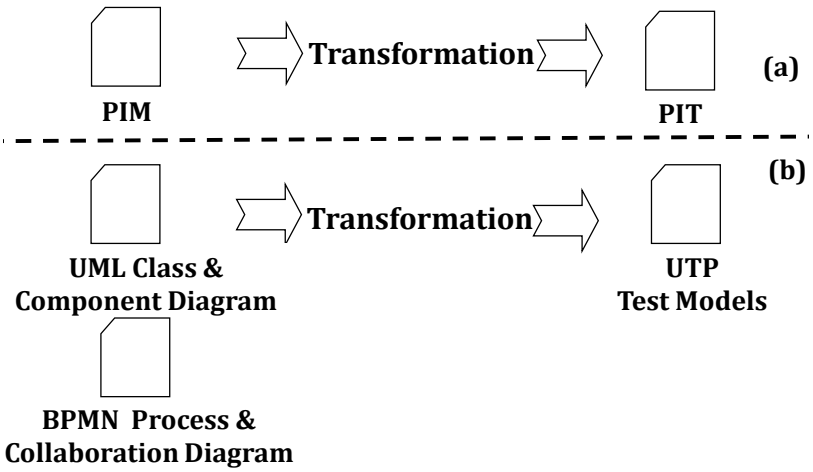


Figure 5.1: Model-driven Testing using BPMN, UML, and UTP.

involves the transformation of a platform independent model (PIM) to a platform independent test model (PIT). The lower part of Figure 5.1 shows our adaptation of the process. Our approach transforms UML and BPMN models to generate test specifications represented as UTP models, which is a PIT model.

We support three distinct aspects from UTP, which are of core importance to the testing activity. These are *test architecture*, *test behavior*, and *test data* elements. Thus, we transform the UML class and component diagrams representing the structure and resources of the processes into *test architecture models* represented in UTP.

For the generation of *test behavior*, information from BPMN collaboration diagram or process diagram as PIM is used. Finally, the *test data* can also be generated by analyzing the resources modeled as classes in UML class diagrams and data elements from BPMN collaboration and process diagrams.

5.2.1 Mappings and Mapping Rules

To define the model transformations, mappings between the elements of source and target models are required to be identified. We identify these mappings by analyzing the correspondence between the relevant elements of UML, BPMN, and UTP. Model transformations require the translation of mappings into concrete rules to realize various transformation scenarios. In principle, a mapping rule realizes a mapping relation, that represents a correspondence between the relevant source model and the target model of a particular transformation.

We define a mapping rule as a 4-tuple (`SourceElement`, `TargetElement`, `Preconditions`, `Postconditions`). The `SourceElement` is an element of the source PIM model, the `TargetElement` is an element in the target PIT model. The `Preconditions` is the set of constraints defined for the execution of the rule. Finally, the `Postconditions` is the set of conditions that define the required changes in the state of the target test models, such as creation of new test elements.

To identify the mappings and to develop the corresponding mapping rules, we analyzed the elements in the source models to generate various aspects of the test specification. According to the definition 4.1 presented in Section 4.2 of Chapter 4, the set B=(b_1, b_2, b_3...b_n) for all (i=1,2,3...n) represents the set of collaboration diagrams, where each b_i represents the *process view* of a process P. The *structural view* of a process is defined using a class diagram C_D and a component diagram C_{OD}. The test suite T represents the *test view*, where T_a represents *test architecture*, T_b represents *test behavior*, and T_d represents *test data*. The mappings and rules required to generate *test architecture*, *test behavior*, and *test data* form these models of the *process view* and the *structural view* are presented below.

5.2.2 UTP Test Architecture Generation

The *test architecture* T_a in UTP represents the structure of a *test model*. UTP defines various concepts related to *test architecture* and provide a set of model elements to represent

these concepts in the form of a UML class diagram decorated with UTP stereotypes. These concepts include `SystemUnderTest` (`SUT`), `TestArbiter`, `TestScheduler`, `TestContext`, and `TestComponent`.

Table 5.1: Mappings between UML,BPMN, and UTP Test Architecture Elements.

UML & BPMN Elements	**Test Architecture Elements**	**Mapping Rule**
Package:UML	TestPackage:UTP	Listing 5.1
Class:UML «ProcessClass »	SUT:UTP	Listing B.1
Class:UML «ProcessClass »	TestContext:UTP	Listing B.2
Participant: BPMN:Collaboration	TestComponent:UTP	Listing B.3
Lane:BPMN	TestComponent:UTP	Listing B.4
TestPath:BPMN	TestCase:UTP	Listing B.6

UTP uses UML class diagrams with test related stereotypes to represent the *test architecture* T_a of a system [BDG+07]. We refer to this class diagram as CD_{ta} representing T_a. To generate each element of the *test architecture*, we first analyze the elements of system models to extract the relevant information and provide the corresponding mappings and develop mapping rules. Table 5.1 presents the mapping and a reference to the corresponding mapping rules. In the following, we present the *test architecture* elements in UTP and discuss how they are generated from the elements of system class and component diagram defined earlier as C_D and C_{OD}. We also use a set of dependency relations D to access various related elements for defining the mapping rules.

Generation of TestPackage– A `TestPackage` in UTP is required to group the related test elements into one package for better organization [BDG+07]. As discussed earlier in Section 4.2 of Chapter 4, the process modeling approach UWE represents the structure of a collaborative process as a `Class` inside C_D with a stereotype «ProcessClass » [KKCM04]. Thus, the test-related elements to test a `ProcessClass` can be grouped together inside one `TestPackage`.

Due to this, we generate a `TestPackage` corresponding to a `ProcessClass`, which groups the test related artifacts to test the process it represents. The mapping rule to realize this mapping is presented in Listing 5.1.

According to the mapping rule in Listing 5.1, there are two preconditions specifying the existential constraints. The first precondition states that a `ProcessClass` x exists in the system class diagram CD_D (Line 5). The second and third preconditions ensure that a BPMN collaboration diagram $b1$ exists corresponding to the `ProcessClass` x (Line 6-7).

Listing 5.1: Mapping Rule to Generate a TestPackage

```
1 Mapping-Rule TA001: Source:ProcessClass->Target:TestPackage
2 Description:
3 /* A TestPackage is generated to a ProcessClass to contain its test related elements
      */
4 Preconditions:
5  ∃x ∈ C_D | type(x)=Class:UML «ProcessClass »
6  ∃b1 ∈ B | type(b1)=Collaboration:BPMN
7  ∃d1 ∈ D | source(d1)=x, target(d1)=b1, type(d1)=Equivalence
8 postconditions:
9  create p ∈ CD_ta | type(p)=UML:Package «TestPackage »
```

```
10  p.name= x.FQN +"TP"
11  set p.Containment=true
12  createTraceLink(x,"Derivation", p)
13  createTraceLink(p,"Tests",b1)
```

In the postcondition part, a `TestPackage` p is created and its named after the original class x and a string *"TP"* is concatenated with it (Line 9-10). Moreover, the property `Containment` of the newly created `TestPackage` is set to true (Line 11), as this `TestPackage` would later contain other test elements. The next two postconditions (Line 12-13) create required dependency relations between various elements. This helps to access the related elements during the transformations as well as preserving these dependency relations for the later reuse, as discussed later in Chapter 6.

Generation of SUT– The `SUT` in UTP represents the system to be tested, which in our case is a process. This `ProcessClass` defines a collaborative process representing interactions between several participants. These interactions are required to be tested to test the functional behavior of the collaborative process. Hence, each `ProcessClass` can be translated to a SUT for a test suite which tests the functional behavior of that process.

We, therefore, map a `ProcessClass` in C_D to a `SUT` in the CD_{ta}. This mapping is realized by the mapping rule presented in Listing B.1 in Appendix B. According to the mapping rule, the preconditions pose various existential constraints. The first three constraints are similar to the rule discussed in the previous section. The third and forth constraints require that a `TestPackage` already exists to test the `SUT`. The creation of the `TestPackage` is already discussed in the previous section. Thus, the forth precondition checks if a `Link` exists between the `ProcessClass` and `TestPackage`, which was created previously in the last postcondition of the rule presented in Listing 5.1. Finally, in the last preconditions the existence of the operations inside the relevant `ProcessClass` is ensured.

The postconditions create the test element `SUT` and give it a similar name as the `ProcessClass`, as it represents the process to be tested. All the operations of the `ProcessClass` are then mapped to `SUT`.

There is a need to relate the `SUT` to the `TestPackage` which tests it. Baker *et al*. [BDG+07] suggest to create an *import* dependency between `SUT` and`TestPackage` so that the tests are able to access `SUT` during the test execution. Thus, an import dependency *dep* is also created. Finally, the required dependency relations are created to relate various source and target elements.

Generation of TestContext– A `TestContext` is UTP contains the test cases and is responsible of the order in which the test cases should be executed. Similar to the `SUT`, a `TestContext` is also mapped to the `ProcessClass`. Listing B.2 in Appendix B presents the mapping rule to generate the `TestContext`. All the test case in the `TestContext` class are added later during the test behavior generation, presented in Section 5.2.3. However, the test control can be defined manually after all the tests are generated.

Generation of Test Components– Test components in UTP simulate the

behavior of the actual components and mock any interaction of the test cases with the outside world. The test components can be generated by analyzing the associations of the `ProcessClass` inside C_D.

However, all the process interactions might not be visible inside class diagrams; as the process may also interact with high level components, GUI windows etc. Similarly, the method called by the processes might be required to mocked or simulated inside test components. This information can also not be obtained from the class diagrams alone. Thus, to generate the test component, the process interactions are required to be analyzed from BPMN collaboration diagrams. In the following, we discuss the interaction modeling scenarios for BPMN collaboration diagrams and provide the mappings to extract test components.

Case 1: Process Interacts with a Participant.

To model the collaborators of a process, BPMN provides the element `Participant`. A `Participant` can represent a software component, a GUI component, or a service provider component. A `Process` can interact with a `Participant` by using a `MessageFlow` element. However, in this case, it is not visible which service of the component is being called, or how the message would be received by the component. Thus, the `Participant` in this case represents a blackbox component. The mapping rule realizing this scenario is presented in Listing B.3 in Appendix B.

Case 2: Process Interacts with a Lane.

A `Lane` inside a `Participant` can either represent a `Class` realizing a `Component` or it represents a `ServiceClass`. A `ServiceClass` can implement a service interface of a `Component` represented by the `Participant`. Therefore, a call to a `Lane` will be received by a `ServiceTask` inside a `Lane`, which correspond to a method or a service. Listing B.4 in Appendix B provides a mapping rule for this scenario.

5.2.3 UTP Test Behavior Generation

The generation of *test behavior* comprises of two major tasks; the generation of test paths from BPMN collaboration diagram, and the transformation of these test paths into UML activity diagram test cases. The generation of the test paths is dependent on the selected coverage criteria. However, the activity of mapping the test paths onto activity diagram test cases should be independent of the coverage criteria and path generation strategies. Thus, to enable such reuse, we design the process of test behavior generation into three sub-activities, as depicted in Figure 5.2.

Extend Model with Distance– According to Figure 5.2, in the first activity, we prepare the BPMN collaboration diagram for test path generation by extending it with some information required by the path extraction algorithms. The algorithm we use in this work uses the distance information for the selection of test paths. Thus, the distance of each node from the end node is computed and the nodes are annotated with this information.

The output of this activity is a diagram that is extended with distance information. In the second activity, the test paths are extracted from the extended collaboration dia-

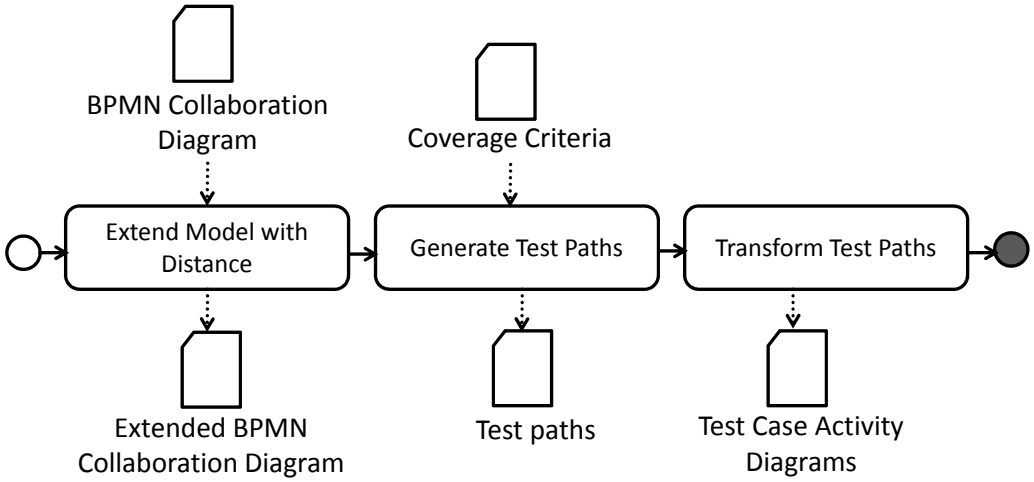


Figure 5.2: Test Behavior Generation Activities.

gram based on some coverage criterion. Finally, during the third activity, the generated paths are transformed to UML activity diagrams to support the visualization and test documentation.

As discussed earlier, the mapping of the test paths onto UML activity diagrams is independent of the test path generation activity, hence can be reused with multiple coverage criteria. In the following sections, we discuss the test path generation and mapping the test paths into UML activity diagram in detail.

Generate Test Paths– To enable the test path generation from BPMN collaboration diagram, we use the *branch coverage* criterion. The criterion covers all the gateways in the diagram for all possible outcomes, and traverses each loop once. To compute the test paths, we first compute the shortest path in the collaboration diagram by using the Dijkstra algorithm [SSD06]. The algorithm starts by selecting the start node in the diagram and then selects its child node which has the shortest distance from the end node.

This process is repeated for each selected node until the end node is reached. The reasons for using the Dijkstra algorithm for computing the shortest path first is that in our test suite, the first test case will always contain the shortest execution path. In the case of limited testing budget, execution of this test case can provide the confidence about at least one process path execution with the minimum cost overhead.

For the calculation of other paths in the diagram we use a Depth First Search (DFS) algorithm with backtracking [Awe85]. The reason of selecting the DFS is its ability to cover all the branches of a graph by visiting all children nodes of a node and backtracking when the end node or an already visited node is found. When a node is added to a path all the information of the node is also copied. If the node corresponds to a send, receive, or service task, the information about the related participants is also copied with that task.

BPMN collaboration diagrams support parallelism by using the parallel gateways. How these parallax activities will be executed is decided at the execution time. For example, all branches of the gateway can be executed in a sequential order [BBG+09]. Since the

decision to treat parallelism is based on execution strategies, we also defer this decision for the test code generation. We maintain the abstraction of test models by adding the whole parallel fragment in the test cases. After the generation of the test paths, the next activity is to map each test path onto a UML activity diagram, as discussed in the next section.

Transform Test Paths to UMLAD– The next activity is the transformation of test paths in to UML activity diagrams. The mappings between the test cases represented by the UTP activity diagrams and test paths extracted from BPMN collaboration diagram requires an analysis of elements of both diagrams for similarity of concepts. BPMN collaboration diagrams comprise of set of nodes (start, end, and intermediate), tasks and activities, participants and lanes, flow elements (control flows, message flows), gateways, and several data elements.

Mapping rules are required to be defined between all these elements and their corresponding activity diagram elements. The mappings between UTP activity diagrams and BPMN processes are presented in Table B.1 in Appendix B. The corresponding mappings rule are also enlisted in Appendix B. Discussion on all the mapping rules is not possible due to space constraints. However, for more detailed explanations a related masters thesis can be consulted [Gro11]. For the demonstration purpose, in the following, we provide an example of a mapping rule that maps a `StartEvent` in BPMN collaboration diagram to the corresponding activity diagram elements.

The `StartEvent` and `EndEvent` in a BPMN collaboration diagram can be mapped to the `InitialNode` and `FinalNode` in the UML activity diagram. However, BPMN collaboration diagrams have many different types of start and end events, such as `MessageStartEvent`, `EmptyStartEvent`, `TimerStartEvent`, and many others. Since the UML activity diagram has only one `InitialNode`, it can be mapped to only one type of `StartEvent` in BPMN collaboration diagram, that is, `EmptyStartEvent`. For the rest, more complex patterns need to be identified in the UML activity diagram. In the following, we discuss the mapping for a more complex `StartEvent`, that is, `MessageStartEvent`.

An Example of Mapping the MessageStartEvent

A `MessageStartEvent` is initiated upon the receipt of a message. Thus, it can be mapped to an `InitialNode` followed by an `AcceptEventAction`, with a `MessageEvent`. A `ControlFlow` in the activity diagram joins both `InitialNode` and `AcceptEventAction`. The corresponding mapping rule is presented in Listing 5.2. Similar to the mapping rule in Listing 5.1 presented earlier, the mapping rule 5.2 also consists of a set of *preconditions* and *postconditions*.

The *preconditions* state that a `MessageStartEvent` exists in a BPMN collaboration diagram and a corresponding `Activity` element exists, which depicts the root element of the test case activity diagram. In the *postconditions*, an `InitialNode`, a `AcceptEventAction`, and a `ControlFlow` is created and added in the test case activity diagram. The name of `AcceptEventAction` is assigned to the `InitialNode` and the *source* and *target* of the `ControlFlow` are set of join the `InitialNode` and `AcceptEventAction`.

Listing 5.2: A Rule for Mapping Messagestartevent/SignalStartEvent in BPMN

```
Mapping Rule 019: MessageStartEvent|SignalStartEvent-> InitialNode,ControlFlow,
    AcceptEventAction
Description:
/*Since a MessageStartEvent and a SignalStartEvent wait for incoming message, they
    are mapped to an InitialNode followed by an AcceptEventAction, which are joined
    by a ControlFlow in an activity diagram test case.*/
PreConditions:
∃ta ∈ T_M | type(ta)= Actvity, streotype(ta)=TestCase
∃P ∈ S_M |type(P)=Collaboartion
∃d1 ∈ D |source(d1) =ta, target(d1)=P, type(d1)=Tests
∃se ∈ S_M | type(se)=MessageStartEvent|SignalStartEvent
∃d2 ∈ D |source(d2) =P, target(d2)=se type(d2)=Containment

PostConditions:
create anode | type(anode)=ActivityInitialNode:UML
create cf | type(cf)=ControlFlow:UML
create aea | type(aea)=AcceptEventAction:UML
ta.add(anode)
ta.add(cf)
ta.add(aea)
anode.name=se.name
source(cf)=anode
target(cf)=aea
```

5.2.4 UTP Test Data Generation

Test data generation requires extraction of *test data* for test cases and definition of the data elements inside UTP. Test data generation is a vast field and scope of this thesis limits the discussion on test data generation techniques. However, in one of our works, we explored the test data generation for business processes in detail [Rav13]. For the concept demonstration, we only briefly discuss the steps to generate *test data* from business process models.

Table 5.2: Mappings Data Elements.

Collaboration Elements	**Activity Elements**
DataObject	Input/Output Pin (with tasks)
DataInput	InputPin of base ActivityPartition
DataOutput	OutputPin of base ActivityPartition
DataStore	DataStoreNode

The first step is to map the BPMN data elements to UTP test cases. Table 5.2 presents the correspondences between data elements of both models. In the step 2, a *test data* class diagram is created to represent the *test data definitions* for a particular process. This diagram contains data pools and data partitions from the data elements and process constraints using the *Category Partition Method*(CPM) [OB88]. CPM is a well known method to generate data partition from variables by identifying sets of data classes based on common traits. An example of data pools and partitions is available in Section 5.3 Finally, test verdicts are modeled by combining all the data outputs and analyzing the expected results.

5.3 Applying Test Generation Approach on HandleTourPlanningProcess

To present concrete examples of the concepts discuss above, we discuss here the test aspects related to the *Field Service Technician* scenario presented in Section 4.2.1 of Chapter 4. Figure 5.3 presents an excerpt of the *test architecture*, *test behavior*, and *test data* for

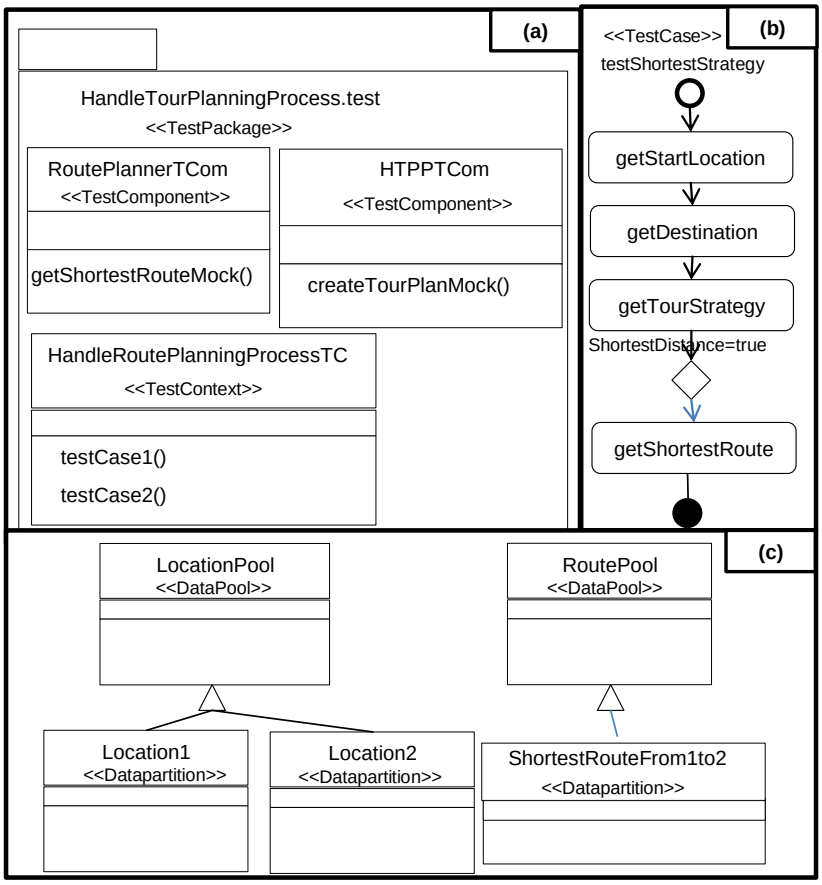


Figure 5.3: An excerpt of the *test architecture* and *test behavior* for Tour Planning Process.

the scenario discussed in Section 4.2.1 of Chapter 4. These aspects can be generated by using the mapping rules discussed above.

The *test architecture* depicted in part (a) of Figure 5.3 represents the elements `TestPackage`, `TestContext`, and two `TestComponent` elements. The test components are generated from the participants of the *HandleTourPlanningProcess* from the scenario presented in Section 4.2.1 of Chapter 4.

The *test behavior* is presented in part (b) of Figure 5.3 which is generated by applying the path extraction and mapping rules discussed above. Part (b) of Figure 5.3 shows one particular test case that test the scenario when the strategy *Shortest Distance* is selected. Finally, the *test data* shown in part (c) of Figure 5.3 depicts the data pools and partitions required for the execution of the test case shown in part (b). The data pools

are generated from the data elements `Location` and `Route` but their corresponding partitions are created manually.

5.4 Chapter Summary

In this chapter, we presented a holistic model-driven test generation approach to generate the test baseline for testing business processes. We generate the *test architecture*, *test behavior*, and *test data* by defining mapping rules between the elements of UML class diagram, component diagram, and BPMN collaboration diagram. The *test architecture* and *test data* generation required direct transformation of elements of source models into target models. To extract the *test behavior*, however, we first extract the test paths from BPMN collaboration diagrams using a path-based algorithm. These test paths are then transformed into U2P activity diagram test cases. To enable the transformations, we analyzed the elements of BPMN collaboration diagrams, UML class and component diagrams, and UTP test models to identify the corresponding elements and then developed mapping rules based on them. An interesting future research area is to provide a way to translate the UTP test specifications for business processes into concrete test scripts to support test execution.

6

Recording of Dependency Relations–between Models and Tests

We use dependency relations to propagate changes across models and tests to identify the affected test elements. Therefore, it is of crucial importance to our work to show how these dependency relations are recorded. *Recording Dependency Relations* is the first activity in the Step 2 of our approach presented in Section 4.1 of Chapter 4. The main contribution of this chapter is to present various solutions for recording dependency relation across models and tests by analyzing different types of dependency relations that can occur between software artifacts. Thus, we present a taxonomy of dependency relations that classifies various types of dependency relations.

We use the taxonomy as a basis to identify a comprehensive set of dependency relations between various views of business processes. Moreover, we adapt two different approaches to record these dependency relations. Firstly, the dependency relations are recorded during the test generation. Secondly, we use a rule-based dependency detection approach, that uses a set of rules which analyze various conditions to identify and record various types of dependency relations.

Our analysis of model-based regression testing approaches presented in Chapter 3 shows a lack of support for various types of dependency relations across views. Similarly, the state of the art approaches repeatedly search dependency relations for each change during the impact analysis. Another limitation of the state of the art is lack of support for dependency relations of various types. None of the model-based regression testing approaches support the notion of type of dependency relations. Therefore, our approach addresses these issues by recording the dependency relations of various types prior to the impact analysis.

Therefore, we first present the fundamentals of dependency relations to gain a better comprehension of the concept and then discuss how we identified and recorded dependency relations in our approach.

6.1 Fundamentals of Dependency Relations

To get a through understanding of dependency relations, following questions are required to be answered.

1. **Q1:** How a dependency relation can be defined? What are the basic building blocks and structure of a dependency relation?
2. From where the dependency relation can emerge? What are the origins of the dependency relations and how to find them?
3. **Q2:** What types and categories of dependency relations exist?
4. **Q3:** How the dependency relations relevant to tests can be recorded?

In the following, we discuss the relevant details to answer the above mentioned research questions.

Defining Dependency Relations– Earlier in the problem definition 2.1, the set of dependency relations is refereed to as D, where each $d_i \in D$ is a dependency relation between the system and test models. In the following, we define a dependency relation, as we use and perceive it in the rest of this thesis.

> **Definition–**A dependency relation $d_i \in D$ can be defined as a 3-tuple (Source, RelationType, Target). The *Source* and *Target* specify the model elements belonging to the models of different system views, which are related to each other. The relation-type defines the purpose of a dependency relation and clarifies its semantics.

According to the definition above, a dependency relation not only points to the related model elements but also contain a purpose to ease the comprehension of the context of a dependency relation. To understand the purpose of dependency relations and to find various dependency relations among models and tests, it is important to identify the genesis of dependency relations.

6.1.1 Origin of Dependency Relations

Keeping in view the dependency relations in the context of model-based testing, we identified the following origins of dependency relations, which can be of interest while finding dependency relations among system models and tests. The different origins of dependency relations are presented in the following:

1. arising from different views of a system.
2. originating form modeling and development methodologies.
3. emerging from meta-models.
4. arising from test generation Approaches.

Relations From Views and Modeling Methodologies– We already discussed the dependency relations originating from different views of a system in

detail in Section 2.2.2. Modeling methodologies define how various models belonging to different views are developed and also define various constraints to enforce better modeling practices. Dependency relations can also originate due to these constraints imposed by modeling methodologies. Since the dependency relations emerging from views and modeling methodologies are closely related, we are discussing them side by side.

An example is, that a `Participant` in the *process view* is *equivalent* to a `Component` defined in *structural view*. This dependency relation is suggested by the modeling methodology of Sadovykh *et al*. [SDE+10]. Another example is of Koch *et al*. [KKCM04] requiring that the *structural view* of each `Process` is modeled as a `Class` in UML class diagram with a stereotype «ProcessClass ». This suggests a relationship between a *process view* and *structural view*, where the `Class` *defines* the corresponding `Process`.

Relations Emerging from Meta-Models– A meta-model defines a model, the elements that constitute a model, and the possible relationships between those elements. For example, UML and BPMN are defined by their respective meta-models [UML07, BPM10] provided by OMG [OMG14]. Meta-models explicitly state various dependencies and relations among elements of models. These relations are also referred to as "*structural relations*" by De Lucia *et al*. [DLFO08]. For example, UML meta-model suggests *inheritance* and *composition* relations between *classes*. It also defines the *containment* relation between the *classes* and their contained *operations* and *attributes*.

BPMN meta-model also explicitly suggests *containment* relations between *processes* and their contained *tasks*, *gateways*, and various *data elements*. However, all the relations are not explicitly stated and some dependency relations can also be extracted by analyzing indirect relations. For example, consider the case where a `Process` contains a `Task` and a `ServiceTask`, which are connected by a `MessageFlow` element. If the *source* of the `MessageFlow` is the `Task` element and the *target* of the `MessageFlow` is the `ServiceTask`, then it means that the *Task* is calling the `ServiceTask`. This suggests the *call* relation between the `Task` and `ServiceTask`.

Relations Arising From Test Generation Approaches– As demonstrated earlier in Chapter 5, model-based testing uses analysis and design models to generate test cases. Thus, the correspondences between *source models* and *target tests* suggest potential dependency relations among them. Such dependency relations are recognized and used for regression test selection in the works of Naslavsky *et al*. [NR07] as well. These relations are further discussed in Section 6.2.2 and 6.3.1.

6.1.2 Classification of Dependency Types

The discussion in this section aims to answer the Q3 from the list of questions presented at the start of this section. Each dependency relation serves a particular purpose and the purpose determines the type of a dependency relation.

Purpose of Dependency Relations– The purpose of dependency relations helps to improve the comprehension of dependency relations and distinguish the different type of relationship between same type of elements. An example of such a de-

pendency type is between an `Operation` and `Class`. A `Class` can contain an owned operation, which specifies a dependency relation whose purpose is to describe a *structural* relationship between the `Class` and `Operation`. Similarly, a `Class` might use an `Operation` of another class. The purpose of this relationship is different as it specifies a usage scenario. Therefore, the purpose of a dependency relation helps to distinguish between various dependency relations. Moreover, it helps to understand the context of the dependency relation, in which the dependency relation is used. To further clarify the concept, we take an example from BPMN collaboration diagrams. A `ServiceTask` in BPMN collaboration diagram can *call* an `Operation` corresponding to the service in UML class diagram. Similarly, an `Operation` in a `TestComponent` can *simulate* the behavior of a `ServiceTask`. in a test suite can test a service task inside a BPMN collaboration diagram. These two dependency relations are among the same element types, that are, `Operation` and `ServiceTask`. However, they reflect different purposes and consequently are used in different contexts. Thus, they can be expressed using two different type of relations, that are, *tests* and *simulates*.

Taxonomy of Dependency Types– Based on the purpose of the dependency relations, we classified the dependency relations into different types. During one of our studies, we identified a set of 50 different types of dependency relations that can occur between software artifacts [LFR13b]. We categorized these relations into a taxonomy of dependency relations which consists of different clusters based on their purpose. Figure 6.1, we present the taxonomy and discuss the high level clusters of dependency types in the following. The details of these clusters and subtypes are available as a separate white paper with this thesis [LFR13b]. The taxonomy consists of three levels and each level contains the subtypes of dependencies of its parent level. The first level contains 9 high level dependency clusters. These are *Abstraction*, *Definition*, *Similarity*, *Structure*, *Realization*, *Behavior*, *Evolutionary*, *Conditional*, and *Causation*.

Abstraction– defines the relation between elements, where one elements represents an abstraction of the other elements. Such as, an element is *refinement* of other or an element is a *parent* of other.

Definition– is a type, which suggests an element defines another element.

Similarity– suggests *similar* or *equivalent* elements.

Structure– suggests *structural* relationships between elements. An example is the relationship between the element `Class` and `Operation` in object oriented software systems. A `Class` *contains* an `Operation`, which is a *structural* dependency relation.

Realization– depicts the set of dependency types, where elements *realize* or *implement* other elements. Another type that falls in this cluster is *instance-of* relation between an element and its type.

Behavior– is a set of dependency relations, where elements are behaviorally dependent on other. For example, operations *calling* other operations, elements *creating* other elements, or elements *simulating* the behavior of other elements. A number of test dependencies also fall in this category. These dependency types include *tests*, *simulate*, and *assert*.

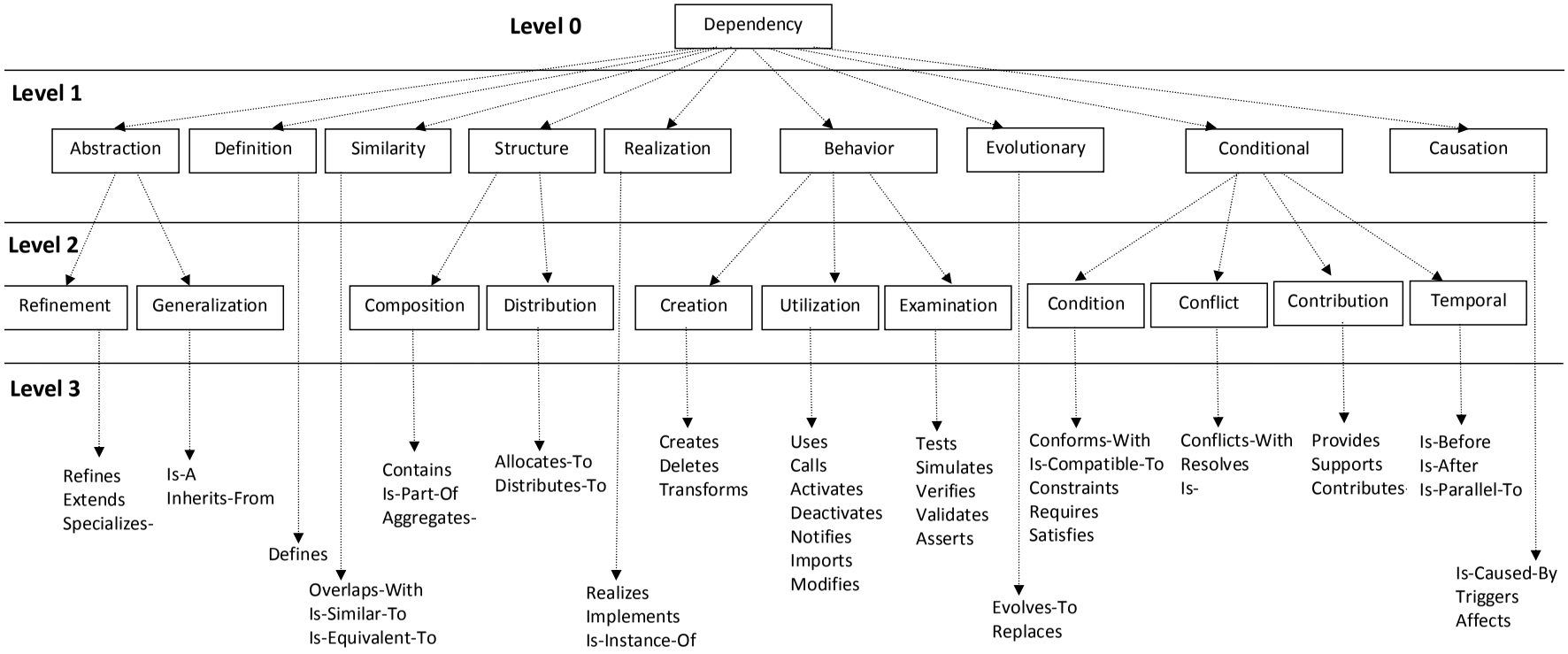


Figure 6.1: A Taxonomy of Dependency Relations.

Evolutionary– dependencies define evolution of an element to another element. These relationships can be seen in evolutionary modeling languages, such as in feature modeling [TTJ+09].

Conditional– category defines the relations, where the existence of an element depends on the existence of other. For example, if one element *requires* another element, *provides* another element, or *conflicts* with another element.

Causation– defines *cause* and *effect* relationships between elements.

In the following, we discuss how the dependency relations of different types are being used in the context of regression testing. We present example dependency relation types and examples of their usage from the domain of business processes using BPMN, UML, and UTP models. The discussion on test related dependencies is also explicitly presented in Section 3.2.7 of Chapter 3.

- **Equivalence:** An `Interface` in a UML component diagram is *equivalent* to an `Interface` in UML class diagram.
- **Derivation:** A `TestPackage` in a UTP test model is *derived* from a `ProcessClass` in a UML class diagram.
- **Tests:** A `TestCase` in a UTP test model *tests* a `Process` in a BPMN collaboration diagram.
- **Mocks:** A `TestComponent` in UTP *test architecture mocks* the behavior of the `Component` in UML component diagram.
- **Definition:** A `Process` in a BPMN collaboration diagram is *defined* by a `ProcessClass` in a UML class diagram.
- **Calls:** A `ServiceTask` in a BPMN Collaboration diagram *calls* an `Operation` in a UML class diagram.

Concrete examples of these types are presented in Section 6.3.3, while demonstrating the concepts on a scenario from our case study.

6.2 Dependency Relations among Business Process Models and Tests

To further elaborate on various types of dependency relations for business processes and tests, we further categorized them into two major categories; *cross-model* and *intra-model* dependency relations, as depicted by Figure 6.2. The intra-model dependency relations are within one specific model, whereas, the cross-model dependency relations relate more than one models.

We are interested in the dependency relations among models belonging to the sets S_M and T. Thus, we categorize cross-model dependency relations in two subcategories, as depicted by Figure 6.2. These are S_M to S_M dependency relations and the S_M to T dependency relations. We discuss these categories in detail in the following.

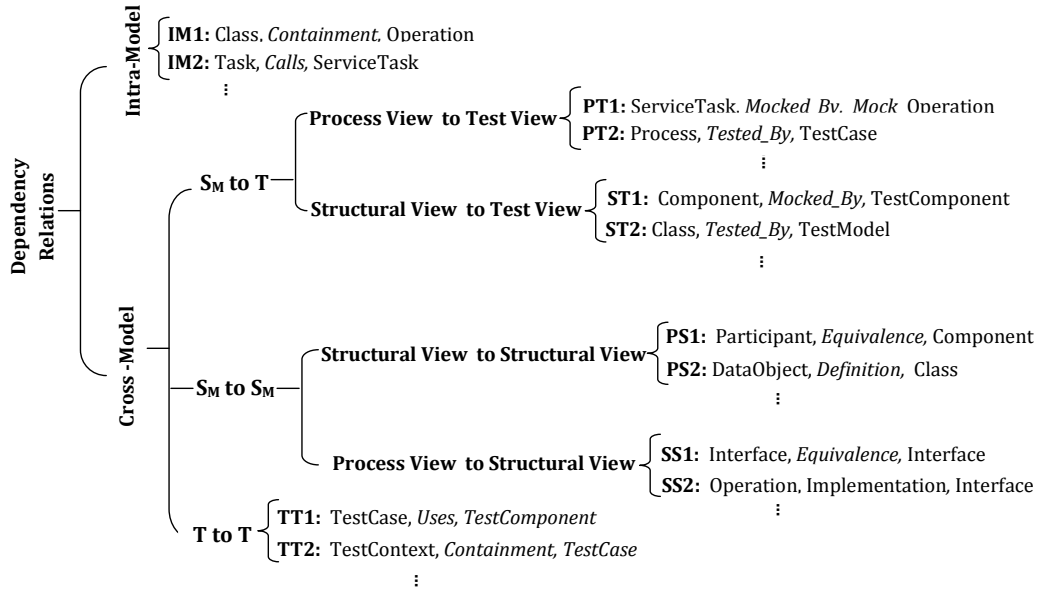


Figure 6.2: Categories of Dependency Relations between Models and Tests.

6.2.1 Intra-Model Dependency Relations

The category *intra-model* consists of dependency relations within one model, such as a relation between two elements of a class diagram. Hence, an *intra-model* model dependency relation d1=(source:A, DependencyType, target:A) depicts a relationship in which both source and target elements belongs to the same model A. Examples of these relations are *IM1*, *IM2*, and *IM3*, as depicted by Figure 6.2.

The dependency relation *IM1* expresses that a `Class` in a UML class diagram can contain a set of *operations*. A similar dependency relation of type *containment* is suggested by *IM2* that expresses that a `Lane` element is *contained by* a `Participant` element in a BPMN collaboration diagram. The dependency relation *IM3* expresses a *call* relation between a service-calling `Task` and the `ServiceTask` that represents the service being called in a BPMN collaboration diagram.

6.2.2 Cross-Model Dependency Relations

The *cross-model* dependency relations are between elements of different models belonging to any of the views. Hence, a *cross-model* dependency relation d2=(source:A, DependencyType, target:B) depicts a relationship, in which, the *source* element belongs to a model A and the *target* element belongs to the model B and $A \neq B$. As discussed earlier, the *cross-model* dependency relations are further divided into two categories; S_M to S_M and S_M to T dependency relations. The category S_M *to* S_M expresses the dependency relations among system models. Whereas, the category S_M *to* T expresses the dependency relations between system models and test models.

The need for these dependency relations is already discussed in Chapter 4 in Section 4.1.2. Since the set S_M consists of the models belonging to the *structural view* and *process*

view of the system, the S_M to S_M dependency relations an be categorized into *structural view to structural view* and *process view to structural view* dependency relations, as shown in Figure 6.2. Further the S_M to T can be categorized as *process view to test view* and *structural view to test View* dependency relations.

S_M to S_M Dependency Relations– In the following, we present the subcategories of S_M to S_M dependency relations and provide examples of them for the concept demonstration.

The category *structural view to structural view* expresses the relationships between UML class and component diagrams (S_M to S_M). As an example, consider the dependency relation *SS1:(Interface, Equivalence, Interface)* depicted by Figure 6.2. It specifies that an Interface of a component in a UML component diagram can also be presented as a concrete interface in a UML class diagram. However, both express the same `Interface`.

The category *process view to structural view* consists of dependency relations between BPMN collaboration diagrams and UML class and component diagrams (S_M to S_M). One example of such dependency relations as depicted in Figure 6.2 is *PS1:(Participant, Equivalence, Component)*. This suggests the situation where a `Component` can act as a `Participant` of a collaborative process. The `Pocess` can call services and methods defined in that *Component* as well [SDE+10].

S_M to T Dependency Relations– The category *process view to test view* consists of dependency relations between the elements of BPMN collaboration diagrams and UTP test models (S_M to T). One example of such a relation is *PT3:(Process, Tested_By, TestCase)*, which suggests that a `Process` can be tested by a UTP `Test Case`. Finally, the *intra-model* dependency relations are the dependency relations inside one particular model belonging to any view.

The category *structural view to test view* contains the dependency relations between the elements of UML class and component diagrams and UTP test models (S_M to T). An example of such a relation is *ST1:(Component, Mocked By, Test Component)*. This dependency relation expresses a situation where the behavior of a `Component` defined by a UML component diagram is simulated by a `TestComponent` in UTP *test architecture*.

T to T Dependency Relations– These dependency relations are the relations between various test models belonging to the *test view* of the system. As discussed in the previous chapter, various test aspects are expressed using a set of different test models, such as class diagrams to express the *test architecture* and *test data*, activity diagrams to express the *test behavior*, and object diagrams to express various test configurations. The relation between these various models belong to the category of dependency relations T to T.

The dependency relation TT1 in Figure 6.2 describes a situation when a `TestCase` uses a `TestComponent` by calling one of its mock operations. The second dependency relation in Figure 6.2 expresses the situation where a `TestContext` contains various test cases to test a particular process. We collected a set of different types of dependency relations between the UTP *test architecture*, *test behavior*, and *test data* elements

and between the other system models. The set of these various dependency relations is available in Appendix C in Table C.1.

6.3 Recording Dependency Relations for Tests

Recording the dependency relations is important to recognize ans identify the dependency relations among models and tests. We identified following four different techniques to record dependency relations among models.

Analyzing Test Execution Traces– During the execution of test cases, the execution traces can be recorded to check the parts of the program executed during a test run [WHLA97]. This can help to establish the links between the parts of the program and their corresponding test cases. However, such an approach cannot be used for recording dependency relations among models and tests.

Mining Co-evolution Histories– In this approach, the co-evolution between production code and the accompanying tests is analyzed by exploring a project's versioning system, code coverage reports, and size-metrics [ZRDD08]. However, a co-evolution of source and test code might not suggest the accurate dependency relations as the developers and testers might be working on two different problems at the same time.

Recording During Test Generation– In model-based and model-driven testing, the tests are generated from the system analysis and design models. Therefore, mappings between the elements of analysis and design models and test models also suggest the dependency relations between them. These relations can be recorded during test generation, as done by Naslavsky *et al*. [NR07]. Since we use model transformations for test generation, we also record dependency relations among source and target models during these model transformations.

Recording Using Dependency Detection Rules– *Bode et al.* [BLR11] presented an approach to record dependency relations by using a set of dependency detection rules to record traceability relations among models. These rules evaluate certain conditions, such as name similarity checks, conditions on attribute values, etc. If the conditions meet, the rules create a dependency relations among the specified source and target elements. This approach is very useful to create the S_M to S_M dependency relations, thus we employ this approach as well. In the following we discuss the both methods we employ for recording the dependency relations in detail.

6.3.1 Recording Dependency Relations During Test Generation

As discussed earlier dependency relations between source models and target test models can be preserved during the transformation process [NZR10], as depicted in Figure 6.3. In our approach the UML and BPMN models are PIM models, whereas, the UTP test models are the PIT models. Hence, during the test generation, we record the de-

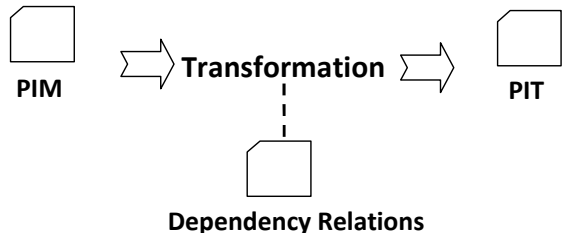


Figure 6.3: Examples of Cross-Model and Intra-Model Dependency Relations.

pendency relations belonging to the categories *process view to test view* and *structural view to test view* . Our test generation approach takes BPMN collaboration diagrams as input, and generates the activity diagram test paths from them using a path traversal algorithm. Hence, the dependency relations of the category *process view to test view* are recoded during this activity. Dependency relations belonging to the category *structural view to test view* are recorded mostly during the generation of *test architecture* and *test data*.

Listing 6.1: An excerpt of the Mapping Rule for the Generation of a TestPackage.

```
8  postconditions:
9   create p ∈ CD_TA | type(p)=UML:Package «TestPackage »
10  p.name= x.FQN +"TP"
11  set p.Containment=true
12  createTraceLink(x,"Derivation", p)
13  createTraceLink(p,"Tests",b1)
```

Listing 6.1 presents an excerpt of the mapping rule discussed in Chapter 5 for the generation of a UTP `TestPackage`. The excerpt only shows the postconditions of the rule, where the last postcondition creates a `Link` between the newly created `TestPackage` and the source `ProcessClass` from which it is generated.

The creation of a `Link` requires the specification of atleast three aspects. These are the `SourceElement`, the `TargetElement`, and the `LinkType`. Thus, in the last line of the mapping rule in Listing 6.1 x is the `SourceElement`, which represents the `ProcessClass`, which is the `SourceElement` of this mapping rules as well. The element p is the `TargetElement`, which refers to the newly created `TestPackage`. Finally, *Derivation* is the type of the dependency relation, which reflects that the `TestPackage` is derived from the `ProcessClass`.

6.3.2 Recording Dependency Relations Using Detection Rules

Since the S_M to T dependency relations are recorded during the test generation activity, it is required to record the S_M to S_M dependency relations as well. To record them, we utilize a rule-based approach, which we introduced in our previous works [LFR13b]. This approach was initially developed for dependency detection among UML, *Use Requirement Notation* (URN), and *Web Ontology Language* (OWL) models. However, we extended the set of dependency detection rules to record the intra-model dependency relations and (S_M to S_M) dependency relations in our approach.

A dependency detection rule checks if two elements meet certain conditions and then

creates a dependency relation between them. These conditions include name similarity checks, conditions on attribute values, conditions for analyzing the structure of models (For example, parent-child-relations) or searching for existing dependency relations between model elements.

A dependency detection rule consists of three essential parts. The *ElementDefnition* part declares the model elements used by the rule. The *ConditionDefinition* part specifies the constraints of model elements, which are required to meet to establish a link between the elements. Finally, the *ActionDefinition* part specifies an action *createLink*, which creates a link of a particular type between the specified source and target elements.

Illustration of an Example Dependency Detection Rule– To illustrate the concept of dependency detection rules, we take the example of the dependency relation *PS1* from Figure 6.2. As explained earlier, this dependency relation explains the relation of type `Equivalence` between the element `Participant` in a BPMN collaboration diagram and the element `Component` in UML class diagram. As the participants in the collaboration diagrams might use the services of components defined in component diagrams.

Listing 6.2: An example dependency detection rule to relate a `Participant` to a `Component`.

```
1 <RuleModel:Rule Description=creates links between participants and components inside
      Component diagram RuleID=traceabilitycreationrule006>
2   <Elements Type=Component Alias=e1/>
3   <Elements Type=Participant Alias=e2/>
4    <Conditions>
5     <BaseConditions type=valueEquals Source=e1::name Target=e2::name/>
6   </Conditions>
7   <Actions name=createLink ResultType=Equivalence SourceElement=e1 TargetElement=e2/>
8 </RuleModel:Rule>
```

To record this relation, a rule has to match the name of the `Participant` with the name of the `Component`. If the names matches, a dependency relation of type `Equivalence` between the `Participant` and `Component` has to be created.

Listing 6.2 presents the rule corresponding to the above discussed situation. A rule is declared with the keywords *RuleModel:Rule*. The attributes of the rule are *Description* and a unique *RuleID* to identify the rule (Line1). After that the elements required to process the rule are declared with the keyword *Elements* (Line 2-3). The type of the attribute is specified with the *type* attribute and an *Alias* is given to the element to identify and access it. The scope of the *Alias* is limited to the current rule only.

A condition required to process are declared inside the *Conditions* tag. Thus, a *BaseCondition* checks if the name of the element `Component` and the element `Participant`, specified as "e1" and "e2", matches (Line 5). The type of the condition is *valueEquals*, which checks if two string values match or not. Finally, an action of the type *createLink* is specified inside the *Actions* tag. The *createLink* action creates and stores a trace link, which expresses a dependency relation of type `Equivalence` between e1 and e2.

As discussed earlier, more complex dependency detection rules involving multiple model elements and complex conditions are also possible. The list of the dependency

detection rules we developed to extract various inter-model and test dependencies are presented in the Appendix C.

6.3.3 Demonstrating Dependency Relations for HandleTourPlanningProcess

The set of dependency relations between various models of *HandleTourPlanningProcess* is very large. However, for the concept demonstration, we present a few dependency relations between the *process view*, *structural view*, and *test view* of the `HandleTourPlanningProcess` in Figure 6.4. The dependency relations are presented as a dotted two headed arrow linking the source and target elements of the dependency relation. The type of a dependency relation is also presented as a label on the arrow. An example of these dependency relations is *D1*, which presents an *Equivalence* relation between the `Component` *RoutePlanner* in UML component diagram and the `Participant` *RoutePlanner* in the BPMN collaboration diagram. The type of relationship suggests the similarity relationship between both elements.

Another example of relationship between the `Participant` *RoutePlanner* and a UTP `TestComponent` is the dependency relation *D5*. The type of this relationship is *mocks* which suggest that the *RoutePlannerTCom* `TestComponent` mocks the behavior of the `Participant` *RoutePlanner* by simulating its provided services.

6.4 Chapter Summary

This chapter presents our contributions to record dependency relations prior to impact analysis and test selection and classification. Thus, we defined fundamentals of dependency relations by defining dependency relations, discussing the origin and types of dependency relations, and analyzing various methods for recording them between models and tests.

We identified and discussed various types of dependency relations between the models of *structural view*, *process view*, and *test view*. A comprehensive set of 92 various types of dependency relations is presented, which are recorded during test generation and using dependency detection rules. We also developed a set of 60 dependency detection rules to record these dependency relations.

In contrast to most of the model-based approaches in the literature, we record dependency relations prior to the impact analysis, which enables a flexible design of the approach. It allows us to use diverse approaches for recording dependency relations to provide a comprehensive coverage of dependency relations. Moreover, it enables the reuse of dependency relations for multiple changes without repetition, which saves the execution time. Once the dependency relations are available, they can be used to perform impact analysis to support regression testing as discussed later.

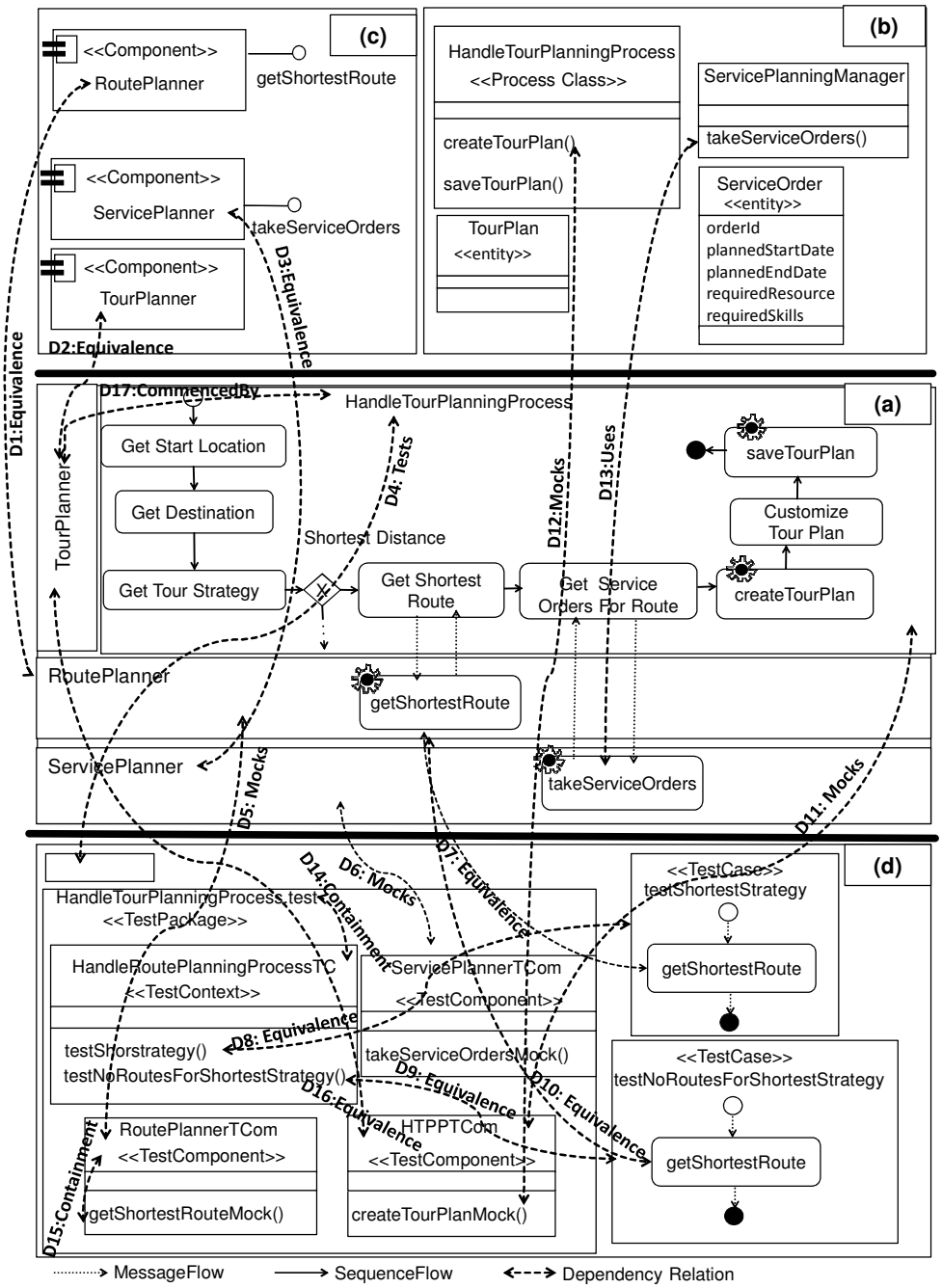


Figure 6.4: Example Dependency Relations between Various Views of Handle Tour Planning Process.

7

Change Application

Changes are of fundamental importance for our approach, as they trigger the need for regression testing. *Change Application* is the second activity of Step 2 of our approach, presented in Section 4.1 of Chapter 4. To apply changes on models, the models in context are required to be analyzed thoroughly for simple and more complex changes. Therefore, this chapter first discusses the *change application* in the context of our approach in the following section. The *Change Taxonomy* is presented in Section 7.2, to assist the definition of model changes. The changes in UML and BPMN models are then defined using our change taxonomy and applied on the scenario from our case study for the concept demonstration.

7.1 Applying Changes from Pre-Defined Change Catalogue

Our approach enable changes by providing a pre-defined set of changes to the test experts, from which they can select a change and execute it on the models to assess its impact on tests. The concept is depicted in Figure 7.1. The first task in Figure 7.1 is to define changes applicable to different models. Defining the changes for models requires an analysis of models to identify the potential changes applicable to various model elements. This also requires an understanding to simple and complex change types, that can be applied on various elements. In the next task, a specific change type is selected from the set of available changes and finally it is applied on the selected model to trigger possible consequences.

For regression testing, such an approach support basic changes as well as more complex changes. It is a major advantage as compared to the state of the art approaches,

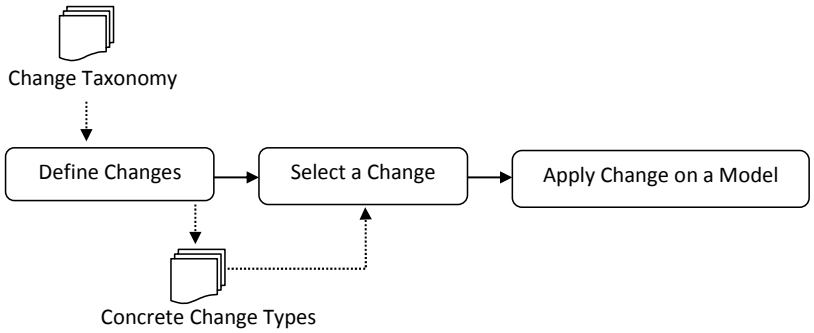


Figure 7.1: Tasks to Support Change Application in our Approach.

which use *model comparison* for change detection [BLY09, FIMR10, NZR09]. Thus, two versions of the same model are compared to detect the differences between them [SM08, KDRPP09]. It results in two main disadvantages.

Firstly, *model comparison* only supports very basic changes, such as *addition* and *deletion*, of model elements. The detection of complex changes, such as *moving* or *merging*, of model elements is very difficult by model comparison and requires complex heuristics. Secondly, in the case when a change is considered infeasible because it is estimated that it would result in huge regression testing effort, it requires additional effort by reverting it from models.

Our approach for the *Change Application* and the change taxonomy is enabled by the tool support in EMFTrace. *Change Catalogue* for a specific model can be created in EMFTrace according to our change taxonomy. EMFTrace allows selecting a particular change for a specific model element to trigger it. Triggering a change initiates the impact analysis activity, which is discussed in Chapter 8. Further, discussion on EMFTrace is presented in Section 10.2 in Chapter 10. To define various change type, a unified and consistent representation of various change types is necessary. Therefore, we define a *change taxonomy*, which consists of various change types to support the definition of changes for models. In the next section, we present our change taxonomy, which enables us to define the set of concrete changes in UML, BPMN, and UTP models presented in Section 7.3.

7.2 A Taxonomy of Change Types

In this section, we present our change taxonomy which we developed to define various aspects related to the changes. Our analysis of the state of the art, discussed in Section 3.2.3 of Chapter 3, reflects that there is a lot of duplicate work in terms of proposed change types and taxonomies. The changes defined by taxonomies are inconsistent to each other. Therefore, we felt the need for providing a taxonomy to improve the existing change classifications, unify various change types, and provide consistent definitions to promote reuse. This unified change representation can be used to support several development activities, such as regression testing, change impact analysis, and consistency checking.

We first define a *change* as we perceive it in the following and then discuss various facets of our change taxonomy.

> **Definition–**A *change* is an *operation*, which can be applied on an *artifact* to change one or more elements of this artifact. The *consequence* of a change is dependent of the *type* of a change.

According to the above presented definition, a change can be perceived as an *operation*, which can be applied to a number of elements belonging to a certain artifact, such as a model, source code, or any other artifact. Similarly, the definition also emphasize the need of distinguishing between various types of changes. Since a change can be applied to a single element or more elements, it also points to the granularity of the change.

Our proposed change taxonomy is based on 4 different facets of a change; *Abstraction*, *Granularity*, *Type*, and *Scope* of a change. These facets are also presented in Figure 7.2. The first facet *Abstraction* of a change defines, whether a change is *Generic*, or *Concrete*. A *generic* change is only a high level change. To realize this change it has to made concrete by defining it for one or more elements of a particular artifact.

As an example, consider the change type *Add*, which is a generic change type. However, when applied to a particular element, such as the element *Class* from UML class diagram, the concrete form would be *Add Class*. The second facet is *Granularity* and the third facet is *Type* of a change. Based on the granularity of a change, we can divide the change types in two main categories. These are *Atomic changes* and *Composite Changes*, as also shown in Figure 7.2. As the name suggests, the atomic change types

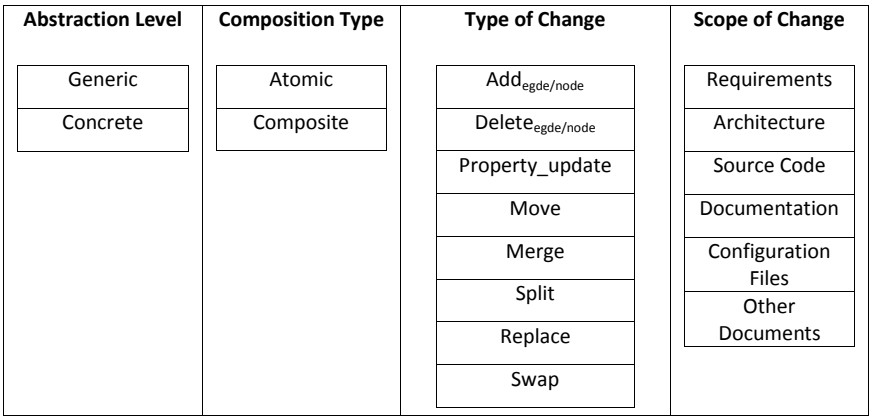


Figure 7.2: Change Types.

are the basic unit of change, and composed of three basic change types; add, delete, and property update, which refers to the update of properties of elements of different artifacts. The composite change types are more, replace, split, merge, and swap.

Finally, the *Scope* of a change determines the type of artifact on which the change has to be applied. Before we explain the various atomic and composite change types, we first exemplify one change type as defined by the change taxonomy for the purpose o concept demonstration. The change "set the return type of method X to *integer*" according to our taxonomy would be classified as follows.

- Abstraction level: concrete.
- Composition type: atomic.
- Type of operation: property_update.
- Scope of change: Architecture, Source Code.

7.2.1 Representing Models as Labeled Graph

To apply the changes on models, we first define a model. A model can be defined as a directed graph of elements, that is, $G = (V, E)$. The elements in the models are perceived as a set of nodes (V) and relations between model elements as edges (E) of the graph Attributes are added as property labels to the nodes and edges. This kind of representation to most of the modeling languages, including UML, BPMN, and UTP. The property labels, assigned to the nodes represent the properties of model elements. However, there are two properties which are independent of domain: the *name* and the *type* of an artifact. Thus, each artifact has atleast these two properties. The set of properties of a node is denoted as p_j.

$$\forall v_i \in V : v_i = \{p_j | j \in \mathbb{N}, j \geq 2\}.$$

Relations which exist between entities can also be enhanced with properties p_l, such as the type of relation (E.g., *Implemented_By*, *Inherits* or *Defined_By*).

$$\forall e_k \in E : e_k = (a, b, \{p_l | l \in \mathbb{N}\}),\ where\ a, b \in V.$$

Based on the above definition of models, we now define the change types applicable on models.

7.2.2 Change Types

In the following, we explain the change types in our change taxonomy, also depicted in Figure 7.2. One of our related publications discuss the application on the change taxonomy on software in general for various development activities [LFR12]. It also presents, how the change types presented in the state of the art taxonomies are mapped onto our change taxonomy.

Atomic Change Types– The set of *atomic change types* is denoted as OP_{atomic}, and is defined as follows.

$$OP_{\text{atomic}} := \{add_{\text{node}}, delete_{\text{node}}, add_{\text{edge}}, delete_{\text{edge}}, update\_property\}.$$

This set is similar to the one proposed by Fluri and Gall [FG06]. In contrast to Fluri and Gall, we do not consider *move* as an atomic operation, since it can be modeled by *delete* and *add* operations. We further distinguish between the *addition* and *deletion* of

nodes and *edges*.

Add and Delete: As the names suggest, *Add* and *Delete* refer to the adding and deleting of *nodes* and *edges*. These are also referred to as *Augmentation* and *Remove* in the literature [BC00]. The former change type adds a new model element and the later removes an existing model element from a model. An example is adding or deleting an `Operation` in a `Class` in UML class diagram, or a `Participant` in a BPMN collaboration diagram.

Property Update: A Property update is a change type, which refers to a change in attributeproperty of a node or an edge. We present a few of them as an example by considering the elementnode *Class* from UML class diagrams.

- Declare a `Class` as *abstract*.
- Change the visibility of a `Class`, `Operation`, or `Attribute` to either *public*, *protected* or *private*.
- Rename a `Class`.
- Change the type of an `Attribute`, the return type of an `Operation` or the type of a method parameter.
- Change the modifiers of an `Attribute`, e.g., declare it as *static* or *final*.

Composite Change Types– Our proposed composite change types consist of sequences of atomic operations and are based on previous research on regression testing [FR11] and traceability maintenance [MRP06]. As previously stated, we consider *move* as an composite change type, since it can be modeled by a sequence of *add* and *delete* operations. Proposed composite change types also share some similarities with the refactoring activities proposed by Baldwin and Clark [BC00]. Therefore, the set of composite operations $OP_{\text{composite}}$ is defined as follows.

$$OP_{\text{composite}} := \{move, replace, split, merge, swap\}.$$

- Move - move one sub-graph to another node.
- Replace - replace one sub-graph by another sub-graph.
- Split - split one sub-graph into several sub-graphs.
- Merge - merge several sub-graphs into one.
- Swap - exchange two sub-graphs.

We discuss the *composite change types* in the following and discuss how they can be modeled as a sequence of atomic operations. The function $P(x)$ denotes the "direct parent" of node x, that is, one of its predecessors which is related to x via a relation of the type *definedBy*, *consistsOf* or similar. For example, a method *m* can have more than one predecessor, that is, one class *C* it belongs to and several other artifacts, which call this method (E.g., related via *Calls*). Thus, we consider *C* as the "parent" of *m*, and *m* as the "child" of *C* likewise.

Move

We support two versions of the *move* change type. First, the developer can move an entire sub-graph x to another node y, e.g., when moving a class to another package,all

methods and attributes are also moved.

$$move(x, y) := delete_{\text{edge}}(x, P(x)), add_{\text{edge}}(x, y).$$

On the other hand, a developer might want to move a node x to another node y only, leaving potential child nodes x_i in place, e.g., when moving an attribute up to the parent class.

$$move'(x, y) := \bigwedge_{i=0}^{n} move(x_i, P(x)), move(x, y).$$

Depending on the context, the *move* change type may also be modeled as a sequence of $delete_{\text{node}}$ and add_{node} operations. This might be required in the context of model comparison and state-based change identification. Move is also referred to as *Change of Hierarchy* by Baldwin *et al*. [BC00]. Since if we move any model element from one place to another, it will be placed from one container to another, which is a form of *change in hierarchy* as well. Fowler [Fow99] proposes moving of methods as a refactoring step, where Gorp *et al*. [VGSMD03] use move operations to pull methods up into a superclass

Replace

In a similar fashion, a developer can replace the entire sub-graph x by another sub-graph y, e.g., when replacing the `Service` corresponding to a `ServiceTask` in a *Process*. This is also discussed later in Section 7.4, when defining change scenarios for an example from our case study.

$$replace(x, y) := delete_{\text{node}}(x), move(y, P(x)).$$

The developer may also replace a node x by node y only, that is, leaving all the child nodes x_i of x in place.

$$replace'(x, y) := \bigwedge_{i=0}^{n} move(x_i, y), replace(x, y).$$

Other examples can be found in the work of Sunyé *et al*. [SPLTJ01], who replace UML state machine transitions, and in the work of Westhuizen and Hoek [WH02], who replace architectural elements during architectural evolution.

Split

The change type *Split* breaks an element into n number of elements. It creates a set of n nodes ($n \in \mathbb{N}, n \geq 2$), which are of the same type as x, and moves all child elements of x to the respective new node x'_i. An example for this operation is the extraction of a class from another class. A new class is created and all methods and attributes which should be extracted are moved to the new class. A tuple (s_a, d_b) denotes, that the s_a*-th*

sub-graph of the node x should be moved to the d_b-*th* sub-graph of the resulting set x'.

$$split(x, n, (s_0, d_0) \dots (s_m, d_m)) := \bigwedge_{i=0}^{n} add_{\text{node}}(x'_i, P(x)),$$
$$\bigwedge_{j=0}^{m} move(y_{s_j}, x'_{d_j})),\ where\ x = P(y_{s_j}).$$

Another example is of Sunyé *et al*. [SPLTJ01], who extract sub-states from UML state machines by splitting them into a set of states.

Merge

The inverse operation to *split* is *merge*, which bundles n entities $(n \in \mathbb{N}, n \geq 2)$ of the same type into one, where y_{i_j} is the j-*th* sub-graph of x_i.

$$merge(x_0 \dots x_n) := \bigwedge_{i=1}^{n} (\bigwedge_{j=0}^{m} move(y_{i_j}, x_0)).$$

The changes like *merge* and *split* are hard to detect using model comparison, as only the added and deleted elements can be easily detected and heuristics are required to infer if composite changes are also present.

Swap

Swapping allows exchanging two entities by another.

$$swap(x, y) := move(x, P(y)), move(y, P(x)).$$

We support exchanging nodes only, which attaches the child nodes of x, x_i, to y, and vice versa.

$$swap'(x, y) := \bigwedge_{i=0}^{n} move(x_i, y), \bigwedge_{j=0}^{m} move(y_j, x),$$
$$swap(x, y).$$

Further combinations of atomic and composite changes presented here are possible to realize more complex refactorings. However, a discussion on refactorings is out of scope of our current discussion.

7.3 Application of Taxonomy on Models of Structural and Process View

As discussed earlier, the change taxonomy presented above help to define uniform changes for various artifacts. Thus, it can be used as a basis to define changes inside various models of interest. Our approach uses the models belonging to *structural view*, *process view*, and *test view* of the system. Thus, changes in the models belonging to these views can be defined using the above proposed taxonomy.

7.3.1 Adapting Change Taxonomy for Models of Structural View

As discussed earlier, we define the *structural view* of the system using UML class and component diagrams. Thus, the changes in various elements of both models need to be explicitly specified. Changes in UML class diagram are widely discussed in the literature on impact analysis and regression testing [BLS03, BLOS06, PUA06, FIMN07]. However, all these approaches only consider *Atomic* change types and do not support the *Composite* change types. The change types we defined for the elements of both UML class and component diagram are presented in Appendix D.

However, for the demonstration of concept, in the following, we consider an example of the element `Class` from UML class diagram and discuss various changes extracted for this element by applying our change taxonomy. Both atomic and composite change types for the element `Class` are listed below.

Atomic Changes in a Class

According to the our change taxonomy, the *atomic* change types for the element `Class` are *Add*, *Delete*, and *Property_Update*. The first two changes are referred to as *Add Class* and *Delete Class*respectively.

Property_Update: is an *Atomic* change type which can be applied to various *Properties* of the element `Class` as visible from the UML meta-model [UML07]. UML meta-model states various properties of a class, e.g., *name* as a `Class` is inherited from the element `NamedElement` in the meta-model. The *Property_Update* change type applied on the property name results in the change type *Rename a Class*. Another important property of the element `Class` is `abstract` and the corresponding *Property_Update* change type is *Make Class abstract*. Since the list of properties of the *Class* is long, all the property update change types for the element `Class` are listed in Table D.2in Appendix D.

Composite Changes in a Class

When applying our taxonomy on the element class, the composite change types which are to be extracted are *move*, *merge*, *split*. The change types *replace* and *swap* are not relevant in the context of classes.

Move a Class: is a scenario when a `Class` is moved from one *Component* or another or

from one *Package* to another. Thus, two concrete change types of type *move* are: *Move Class to Package*, *Move Class to Component*. The application of the `move` change type on the element *Class* would result in affecting all its relationships to the other classes. However, this question is relevant to the *impact analysis* problem, hence is addressed in Chapter *chapter:impactanalysis*.

Merge Classes: is a scenario, where two classes are merged together. The merging of two classes. The corresponding concrete change type listed in Table D.2in Appendix D is *Merge Two Classes*. Since it is a composite change type, it is further composed of many other change types. All the attributes and operations of the source class have to be added to the target class. Similarly, the properties of the source class are also to be applied on the target class. However, the affects on attributes, operations,and relationship of the class would be assessed during *impact analysis*, thus are not relevant to *change identification* problem.

Split a Class: is a composite change type. In contrast to the *Merge* operation, the *Split* operation splits a class into n number of classes. This operation can also be seen as extraction of another class from an existing class. The *Split* operation would result in creation of a new class and moving the selected methods and attributes of the existing class to the newly created class.

7.3.2 Adapting Change Taxonomy to the Models of Process View

To model the changes for the *process view*, it is required to identify and define define changes applicable to BPMN collaboration diagrams by adapting our change taxonomy.

BPMN collaboration diagrams are composed of various model elements, such as processes, participants, lanes, tasks, gateways, message flows, sequence flows, and several data elements. The list of several atomic and composite change types applicable to BPMN collaboration diagrams are presented in Table D.1 of Appendix D. Here we discuss the element *Participant* from BPMN collaboration diagram and discuss the atomic and composite changes applicable to it in the following.

Atomic Changes for the Participant

As discussed earlier, the element *Participant* in a BPMN collaboration diagram represents various contributors of a collaborative process and can refer to an *actor*, *stack holder*, a *system component*, an *external component*, or *subsystem*.

Thus, the atomic change types are *Add a Participant in Collaboration*, *Delete a Participant from Collaboration*, and *Rename a Participant*. The rename might be applied in the result of renaming a component or any other other element to which the *Participant* refers. Deleting a *Participant* from a collaboration results in the deletion of all incoming and outgoing message flows, and also in the deletion of all its service task.

Composite Changes for the Participant

We analyzed each element of BPMN collaboration diagram to access which composite change types are applicable on those elements.

- ***Merge Two Participants:*** Merge Two participants into one. This change will cause the merging of all the contained service tasks and incoming and outgoings message flows of the `Participant` as well. In the result, those service tasks will get affected as well.
- ***Split a Participant:*** Split a Participant into two Participants. Application of those change type creates another `Participant` by splitting an existing `Participant` into two. Similar to the merge change type, the split change type also affects the service tasks contained in the old `Participant` and its incoming and outgoing message flows. As some of them might be required to move into the newly created `Participant`.
- ***Move and Swap a Participant:*** These change types are not relevant to the element `Participant`, as moving a `Participant` or swapping it with another inside a collaboration would only affect its graphical representation and does not affect its semantics. All the `Participant` elements belong to the same hierarchy in the model, hence, these change types only change the graphical placement of the element `Participant`.
- ***Replace a Participant***: Replacing a `Participant` with another will not only replace the old `Participant` with the new one, it will also affect its contained service tasks and incoming and outgoing message flows.

7.4 Demonstrating Changes for HandleTourPlanning-Process

In the previous chapters, we already discussed the generation of the test models and recording of dependency relations for the business process *HandleTourPlanningProcess*. This section introduces a change scenario to fix a problem in the specification of the process and discusses the concrete changes we planned based on the change taxonomy we introduced above. The relevant views of *HandleTourPlanningProcess* are already presented in Figure 4.3 of Chapter 4.2.1. We developed further change scenarios for our case study for various *perfective* and *corrective* [WO03] maintenance activities, which are also presented in Table D.4 in Appendix D. In Figure 7.3, we present

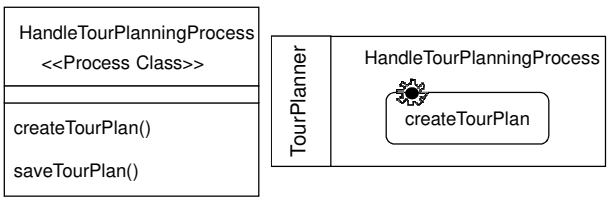


Figure 7.3: The Elements of the HandleTourPlanningProcess Relevent to the Change Scenarios.

a `ServiceTask` *createTourPlan* and its corresponding operation `Class` *HandleTourPlanningProcess* «ProcessClass ». Thus, the *ServiceTask* represents an element of the *process view*, whereas, its corresponding operation modeled inside a class represents elements of *structural view*.

The operation *createTourPlan()* creates and initializes a *TourPlan* object and returns it to the calling process. We identified that the creation of a *TourPlan* in *HandleTourPlanningProcess* not only requires the creation and initialization of the *TourPlan* object, but it should also assign the *Tour* and the *ServiceOrders* selected for the *Tour* to the *TourPlan*. Otherwise, an empty *TourPlan* object would be kept in the system, which would violate the constraints of the system design. Thus, the replacement of the old *createTourPlan* operation is required with a new operation, which takes the parameters `currentTour` and a list of *SericeOrders*.

This change scenario also requires modified operation contracts. However, to simplify, we do not discuss the operation contracts here and focus only on the addition of a new operation. Thus, this change scenario leads to a number of changes, two of which are presented in the following. Other changes relevant to this scenario are also presented in Appendix D.

1. ***Change1:***
   Name of Change: Add Operation.
   Abstraction Level: Concrete.
   Composition Type: Atomic.
   Scope: Design/Model Level.
   Source: *createTourPlan*(`Tour` currentTour, `ServiceOrders.<List>`)
   Target: `HandleTourPlanningProcess` «ProcessClass ».
2. ***Change2:***
   Name of Change: Replace ServiceTask.
   Abstraction Level: Concrete.
   Composition Type: Composite.
   Scope: Design/Model Level.
   Source: createTourPlan().
   Target: *createTourPlan*(`Tour` currentTour, `ServiceOrders.<List>`)

In the above presented changes, the *Change 1* describes the scenario when a new `Operation` is added inside the `HandleTourPlanningProcess` «ProcessClass ». The name of this change is *Add an Operation*, it is defined as a *Concrete* change as we are applying it on concrete model elements. It is modeled as an *Atomic* change type and its *Scope* is limited to the design models.

The *source* and the *target* attributes define the model element which is added (That is, an `Operation`) and the target `Container` element in which it is to be added (That is, `Class`) respectively. The changes are named as *Add Operation* and *Replace Service Task* In the subsequent chapters, we discuss the change impact analysis and regression test classification, corresponding to the changes discussed in this chapter.

7.5 Chapter Summary

The contributions of this chapter are three-fold. Firstly, it establishes the need of change application and a change taxonomy in the context of our approach. Secondly, it presents our change taxonomy, which is based on a set of *atomic* and *composite* change types. We present three atomic changes types *add*, *delete*, and *property_update*, and present several

examples of their application in the context of models. The set of composite change types consists of five changes: *move*, *merge*, *split*, *replace*, and *swap*.

Thirdly, it adapts the change taxonomy to define changes for UML and BPMN models. To do so, we analyzed the model elements from BPMN and UML models for the applicability of each change defined in our change taxonomy. To demonstrate the application of changes on our case study, we developed various change scenarios and defined the changes to realize these change scenarios using the changes defined for UML and BPMN Models.

8

Rule-based Impact analysis Across Tests

Rule-based Impact Analysis is is the third activity of Step 2 in our approach and the it is performed in response to the application of changes. We developed our rule-based impact analysis approach to analyze the impact of changes in heterogeneous software artifacts by exploiting the interplay of changes and dependency relations [LFR13a]. Thus, the rule-based impact analysis approach solves the *test selection* problem presented in Section 2.1 of Chapter 2 by identifying the test elements affected by a change.

In the context of regression testing, the impact analysis is performed specifically to cover the changes affecting the test models. As discussed earlier in Chapter 4 our approach is based on the notion that potentially affected test cases can be identified by considering three distinct aspects. These are *type of change*, *type of the element* on which the change is applied, and the *dependency relations* relevant to that element. Our rule-based impact analysis approach complements this notion and relies on these three aspects to propagate the impact of a change on models.

Our change impact analysis approach can be used to deal with various software development issues, such as determining the architectural erosion and software consistency checking [LFR13a]. However, we particularly focus on how to use the approach to propagate changes to the *test view* to find the affected test elements. Thus, the rules consider the dependency relations among system models and test models. Consequently, they are able to identify the affected elements of test models by processing the relevant dependency relations. These affected test elements are later analyzed to classify them for regression testing in Chapter 9 to address the *test classification* problem. This chapter contributes by first providing an insight to our rule-based impact analysis approach and then presents the adaptation of the concept to define impact analysis rules for the domain of business processes.

8.1 Insight to Rule-based Impact Analysis Approach

As mentioned earlier, the basis of rule-based impact analysis approach is the interplay of change types and dependency relations. A dependency relation refers to two related elements. Hence, if one is affected by the change, its related element might be

considered possibly impacted. The decision, whether, the related element is impacted or not is based on the type of dependency relation, type of the applied change, and further constraints to specify the conditions under which an element can be considered impacted. Thus, to enable rule-based impact analysis, changes, dependency relations, and conditions are required to be analyzed. Figure 8.1 presents an example to clarify

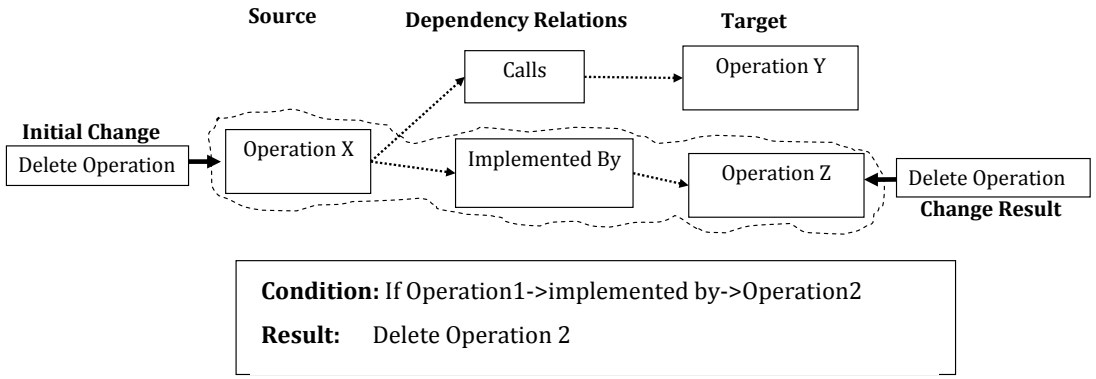


Figure 8.1: Interplay of Changes and Dependency Relations for Rule-based Impact Analysis.

the notion of impact propagation by considering the factors mentioned above. According to Figure 8.1, an `Operation` might have two different types of dependency relations with other operations. First is that an `Operation` might call another `Operation` during its course of control, depicted by Operation X and Operation Y. Similarly, an `Operation` in a `Class` might implement an `Operation` of an `Interface`, depicted by Operation X and Operation Z. If an `Operation` is deleted, it would not effect the `Operation` it calls.

However, if an `Operation` is deleted, the `Operation` implementing it is required to be deleted as well. In the case depicted in Figure 8.1, if Operation X is deleted, the Operation Z is to be deleted as well. The lower part of Figure 8.1 depicts this condition. It also shows the result, which requires deletion of `Operation` Z.

8.1.1 Defining Impact Rules

By the above presented analogy, an impact rule defined for a particular change type retrieve a number of impacted elements by retrieving relevant dependency relations if the conditions satisfy. These dependency relations can point to further elements and further changes can be triggered on the retrieved elements to recursively perform impact analysis on the retrieved elements. This is the pivotal concept of our approach and is defined as follows. An impact rule operates on a change $c_i \in C$ and processes dependency relations $D = d_1, d_2, d_3...d_n$ under various conditions to find the impacted elements. This the following equation present the scenario of recursive impact analy-

sis.

$$Given\ c_i \in C \text{ applicable to } elem \in S_M. \tag{8.1}$$

$$Given \bigwedge_{i=0} m\ d_i\ \in D^{elem} \mid D^{elem} \text{ set of dependencies for } elem. \tag{8.2}$$

$$\bigwedge_{k=0} m\ target = findTarget(c_i, elem, D_k^{elem})). \tag{8.3}$$

$$repeat for elem = target. \tag{8.4}$$

According to 8.1, the c_j is a change applicable to the element a from a model M_a which belongs to the set of system models. Further, according to 8.2, D_a is the set of dependency relations of the element a, where the number of dependency relations in the set are n. The equation 8.3 defines the impact propagation scenario, where the impact of the change c is propagated through all the dependency relations of a in the set D_a. Structure of the impact rules is defined as follows.

Based on the above mentioned aspects, an impact rule R, as we define it in Equation 8.6-8.10 is a 5 tuple (c_t, m_e, ED, QD, RD).

$$R = (c_t, m_e, ED, QD, RD). \tag{8.5}$$

$$c_t \in C \wedge c_t\ applies\ on\ S_M. \tag{8.6}$$

$$m_e \in S_M. \tag{8.7}$$

$$ED = (e_1, e_2, ..., e_n) \mid \bigwedge_{i=0} n\ e_i \in S_M \vee T. \tag{8.8}$$

$$QD = (q_1, q_2, ...q_m) \mid \bigwedge_{j=0} m\ \exists e_i\ constrained\ by\ q_j. \tag{8.9}$$

$$RD = (a_1, a_2, ..., a_p) \mid \bigwedge_{k=0} reports\ impact \wedge a_i = (source, target, impact, result). \tag{8.10}$$

Change Trigger– According to the equations 8.6 and 8.7, the element c_t is a change type that acts as the *Change Trigger* for the impact rule and the element m_e defines the concrete model element from the set of system models on which c_t is applied.

Element Definition– According to the equation 8.8 ED is the *Element Definition* part. where each e_i in the set ED is an element from one of the models belonging the set S_M or T These elements include the element on which the change is applied and also those elements which are being used in the *Condition Definition* part to process dependency relations. For example, consider the example change *Add Operation* in a design class. It requires the definition of element `Class` and the element `Operation`. However, adding an operation in a class might require adding an operation into the implementation class due to the *implementation* relation between them. In this case, the implementation class is also required to be defined in the *Element Definition* part, for example, the `ClassDeclaration` element in java.

Query Definition– According to the equation 8.9 QD is the *Query Definition* part and each query-definition q_i specifies a condition on the elements belonging to the set ED. These conditions include logical conditions which can filter the elements selected by the rule, for example *AND*, *OR*, *XOR*, and pre-defined operations to query the attributes of models and the relations between models.

One example of these predefined operations is *modelParentOf(a, b)*, which checks if a model element *a* is parent of another model element *b*. Similarly, another important predefined operation is *modelRelatedTo(a,dType, b)*, which queries the dependency relations where source of the relation is a, target of the relation is b, and the dType refers to the type of the dependency relation. The types of dependency relations are already elaborated in detail in Chapter 6. The conditions in the *Query Definition* should include atleast one condition that uses the *modelRelatedTo* operation to propagate the change further.

Result Definition– Finally, according to the equation 8.10 RD is the *Result Definition* part, where each result definition a_i is an action that reports an impact. This action further consists of a set of elements. The source element tells the *source* and target depicts the source and target elements on which change is applied. For example, the change type Add Operation requires a source element, that would be the `Class` in which `Operation` is added. The *target* would be the `Operation`, which is to be added. The *impact* depicts the set of elements which will be potentially affected in the result of this impact rule. Finally, the *result* refers to the next change, which is to be applied on the obtained affected elements to further continue the impact analysis.

8.1.2 Impact Analysis Process and Activities

The impact analysis process starts with the application of a change on a model, as depicted in Figure 8.2. Therefore, first a model from the set of system model is selected. After that, the required change is selected and applied from the *Change Catalogue*. The details about how the *Change Catalogue* is being created is already discussed in Chapter 7.

The actual execution of impact rules is accomplished in a recursive manner [LFR13b]. First, the impact rules corresponding to the initial change (*Change Trigger*) are selected and applied. The impact rules process the set of dependency relations expressed as *Trace Links*, which were earlier recorded, as presented in Chapter 6. These impact rules processes the relevant dependency relations to produce the impacted model and test elements and corresponding *impact reports*.

Each impact report is then again treated as the initial change (*Change Trigger*) and processed accordingly. Consequently, further impact reports might be created. The final result produced by this impact analysis process is a set of impact reports, containing the elements defined by the *Result Definition* part of the rule.

Dealing with Cyclic Dependencies– Cyclic dependencies between software artifacts may lead to infinite loops during impact propagation. We address this problem by maintaining two lists storing 3-tuples (changed element, change type, im-

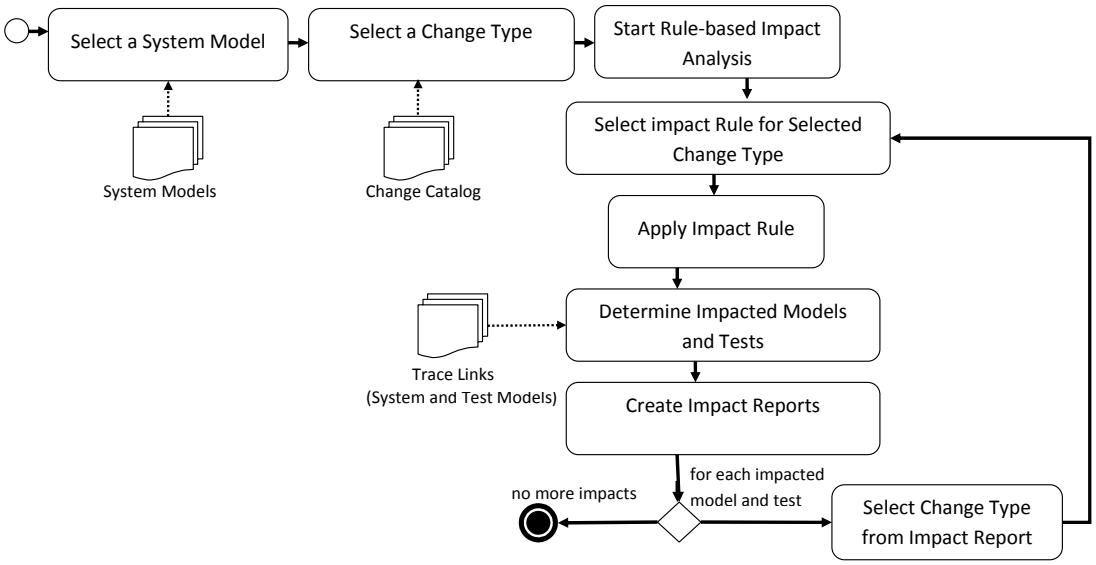


Figure 8.2: Tasks for Rule-based Impact Analysis.

pacted element) to record the current progress of impact propagation, which is inspired by the *A**-path finding algorithm [HNR68]. The *ClosedList* contains all already *explored* 3-tuples or impact paths. The *OpenList* contains all 3-tuples which might lead to new impact paths and should be further explored.

If a possible new impact path is found, the *ClosedList* is searched for a tuple containing the same elements and change types. If such a tuple is found, further propagation on this path is stopped. This concept and the related algorithmic details are further discussed in our work on heterogeneous impact analysis [LFR13b].

8.2 Impact Rules Covering Business Process Views

The dependency relations between various views of business processes are already discussed in detail in Chapter 6. To see how changes are propagated through these dependency relations, we consider the *HandleTourPlaningProcess* scenario used in the earlier chapters. According to the scenario a `Component` might act as `Participant` in a collaborative process to provide services to that process [SDE+10]. This dependency relation was earlier presented as *PS1* in Figure 6.2, in Section 6.2 of Chapter 6. It demonstrate a dependency relation between *process view* and *structural view*. In the

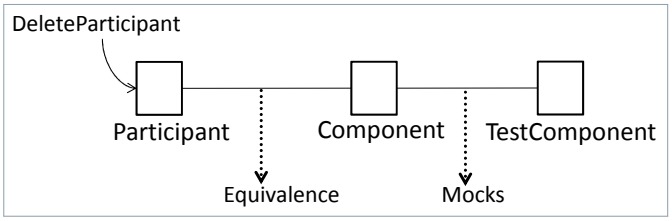
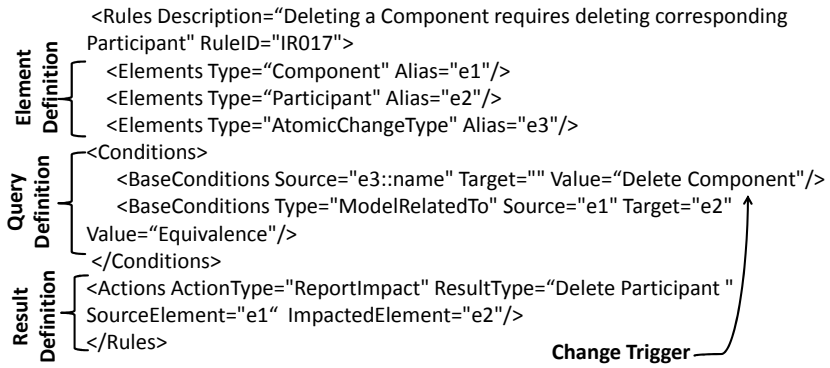
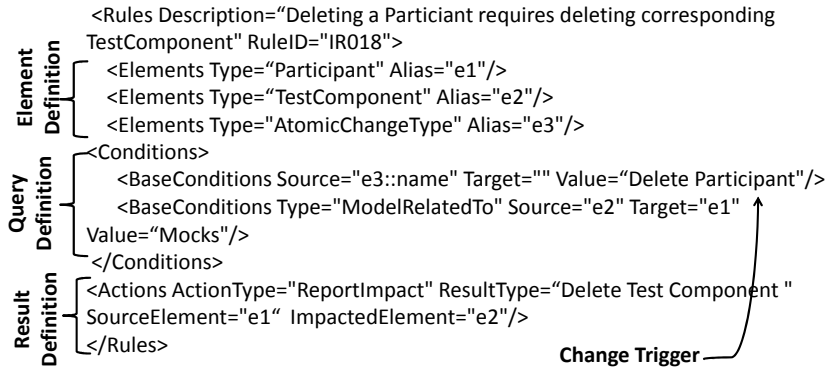


Figure 8.3: Impact Propagating Through Chain of Dependency Relations.

context of test, a `TestComponent` might mock the `Participant` to simulate its behavior for testing. This dependency relation was also discussed and presented as *ST1* in Figure 6.2 in 6.2 of Chapter 6. It related the elements of *process view* and *test view*.



(a) Component affects Participant



(b) Participant affects TestComponent

Figure 8.4: Consequent Impact Rules for Structural and Process View.

If a `Participant` is deleted, it would require the deletion of corresponding `Test Component` as well. This scenario is also depicted in Figure 8.3. The rules realizing this scenario are presented in Figure 8.4.

The element *e3* in Figure 8.4a is a *Change Trigger* and the name of its associated change type is *Delete Component*. The *Element Definition* part defines the elements to be evaluated by the impact rule, that are, `Participant` and `Component`. The conditions part applies constraints on the model elements defined in the *Element Definition* part. One condition in the rule checks the type of the applied change type to trigger the rule. The other condition uses the predefined operation *modelParentOf* to check if the dependency relations PS1:(Participant, Equivalence, Component) exists between any of the components and participants in the models.

If the conditions are satisfied, the next change depicted in the *Result Definition* part is triggered. In the rule presented in Figure 8.4a, the resulting change is *Delete participant*, which will be applied to all the retrieved participants after the execution of rule. After the execution of rule depicted in Figure 8.4a, the rule depicted in Figure 8.4b would be triggered for all the related participants. This would in the result invoke the rules corresponding to the *Delete TestComponent* change type for all the related components.

The impact rules presented above enable us to propagate changes using inter-model and intra-model dependency relations for the models belonging to various software views including the *test view*. These rules realize and complement our theory that test selection can be enabled by propagating impact using the type of applied changes, dependency relations, and the type of model elements. This was a major goal of our work stated in Section 1.3 of Chapter 1. The list of the impact rules reacting to all the changes inside the models of interest is rather long. Therefore, we present a part of these rules in Appendix E for further consultation.

8.3 Demonstrating Rule-based Impact Analysis on HandleTourPlanningProcess

We presented a change scenario from the *TourPlanningProcess* earlier in Chapter 7. In the following, we present the impact rules adhering to the changes from the scenario. Two different changes were taken from the scenario. First is adding an operation into a `ProcessClass` and the other is replacing a`ServiceTask` with another `ServiceTask`.

8.3.1 Impact Analysis for the Application of Change 1

For the Change 1 *Add Operation* in `ProcessClass`, when an `Operation` is added in a `ProcessClass`, a chain of dependency relations is required to be analyzed, as presented in Figure 8.5 As the `ProcessClass` in UML class diagram represents a

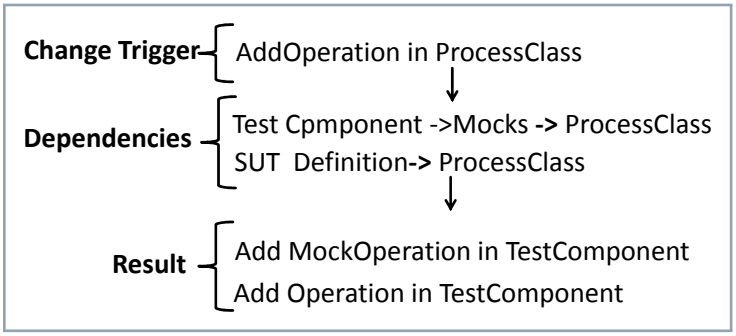


Figure 8.5: Dependencies and Results of Applying the Change AddOperation.

`Process` in BPMN collaboration diagram. This corresponding `Process` is commenced by a `Participant`. For the test purpose, as discussed earlier in this chapter, a `TestComponent` can simulate the behavior of a `Participant`. Thus, is an operation is added to a `ProcessClass`, a corresponding `MockOperation` is to be added in that `TestComponent` as well.

Similarly, the `ProcessClass` also provides a definition of the `SUT` in the *test architecture*. Therefore, as shown in Figure 8.5, a corresponding operation is required to be

added in that SUT as well. Thus, two rules are triggered by the change *Add Operation* in `ProcessClass`. These rules are presented in Table 8.1.

Table 8.1: Impact Rules for the Change AddOperation.

Rule	IR006
Description	Adding an operation in class requires adding mock operations in corresponding test components
Elements	e1:Class «ProcessClass », e2:Operation, e3:Class «TestComponent »
Change Type	
Conditions	(modelRelatedTo(e3,Mocks,e1) OR modelRelatedTo(e1,Tests,e2))
Actions	reportImpact(e1, Add MockOperation,e2, e3 | e2)
Rule	IR008
Description	Adding an operation in ProcessClass requires a corresponding operation in SUT
Elements	e1:Class «ProcessClass »,e2:Operation, e3:Class «SUT »
Change Type	
Conditions	(modelRelatedTo(e1,Definition,e3))
Actions	reportImpact(e1, Delete ProcessClass, e2)

The results of these rules would trigger a set of other changes and corresponding impact rules, as shown in the actions of the rules presented in Table 8.1. These rules are also available in Appendix E. When applying the change *Add Operation* on the

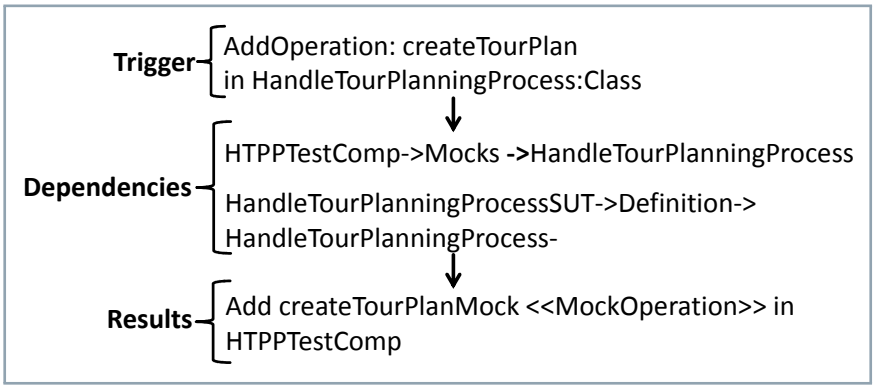


Figure 8.6: Dependencies and Result of Applying Change 1 on HandleTourPlanningScenario.

`ProcessClass` *HandleTourPlanningProcess*, the result will be similar to the scenario depicted in Figure 8.6. The dependency relations consumed in the case depicted in Figure 8.6 can be seen in Figure 6.4 in Chapter 6 as *D18*, *D17*, and *D16* respectively. The `TestComponent` *HTPPTCom* will be considered as affected by the application of this change and a corresponding impact report would be produced.

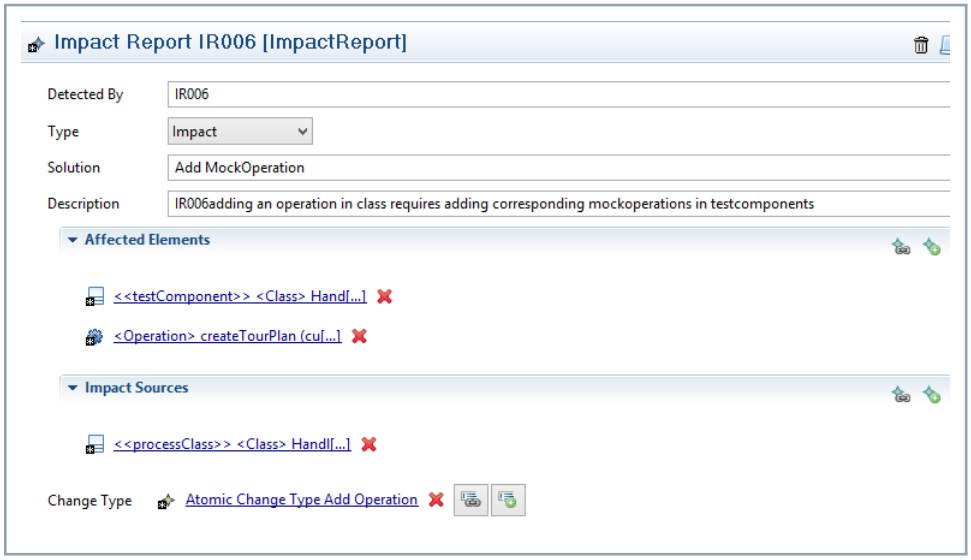


Figure 8.7: Impact Report for Add Operation in ProcessClass.

8.3.2 Impact Analysis for the Application of Change 2

As discussed in the previous chapter, the change 2 to realize the change scenario for *HandleTourPlanningProcess* requires the replacement of an `Operation` corresponding to the `ServiceTask` {`TourPlan` *createTourPlan()*} with the new `Operation` {`TourPlan` *createTourPlan*(`Tour` *currentTour*, `ServiceOrders`«`List`» *so*)}.

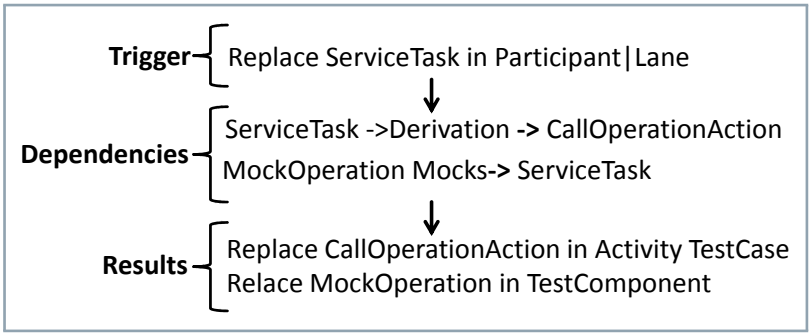


Figure 8.8: Dependencies and Result of Applying Change Replace ServiceTask.

Figure 8.8 illustrate the required dependencies to be processed for the application of the change *Replace ServiceTask* and the resulting changes to be triggered. A `MockOperation` can mock the behavior of the `ServiceTask` in the test system. This corresponding `MockOperation` will be affected and is thus required to be deleted from its parent `TestComponent`. A new `MockOperation` is required to be added in the corresponding `TestComponent`. The dependencies consumed for this scenario are also depicted in Figure 8.8.

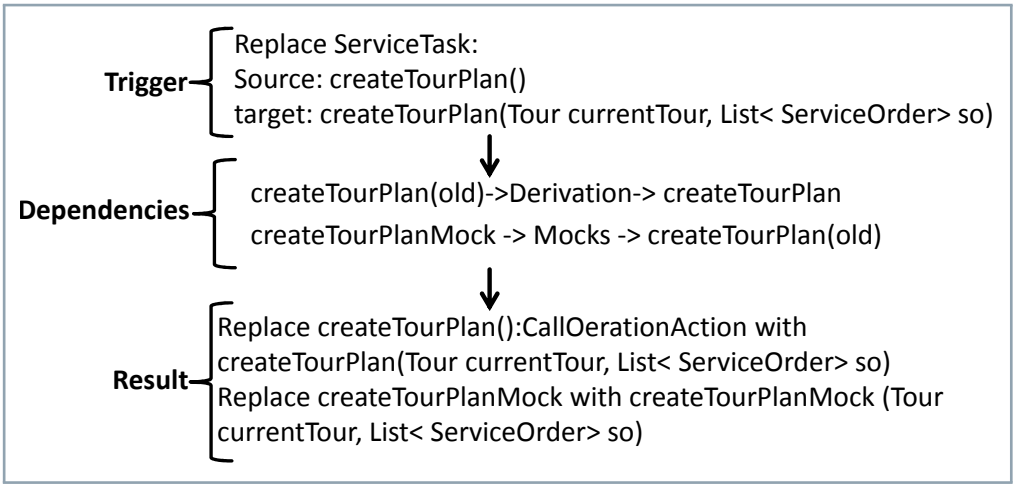


Figure 8.9: Dependencies and Result of Applying Change 2 on HandleTourPlanningScenario.

Table 8.2: Impact Rules for the change Replace ServiceTask.

Rule	IR007
Description	Replacing a ServiceTask requires replacement of corresponding CallOperationAction in tests
Elements	e1:ServiceTask, e2:ServiceTask, e3:CallOperationAction
Change Type	Replace ServiceTask (Composite)
Conditions	(ModelRelatedTo(e1,Derivation,e3))
Actions	reportImpact(e1, Replace CallOperationAction, e2, e3|e2)
Rule	IR010
Description	Replace a `ServiceTask` with another `ServiceTask` affects corresponding mockoperation
Elements	e1:`ServiceTask`, e2:`Operation`, e3:`ServiceTask`,e4:`Operation`
Change Type	Replace `ServiceTask` (Composite)
Conditions	(ModelRelatedTo(e2,Mocks,e1) OR ModelRelatedTo(e4,Mocks,e3))
Actions	reportImpact(e1, Replace MockOperation, e3,e2|e4)

Figure 8.9 depicts the application of the actual change on the `ServiceTask: TourPlan`*createTourPlan()*. The corresponding `MockOperation {createTourPlanMock()}` from the `TestComponent` *HTPPTTCom* gets affected due to this change.

Similarly, if this `ServiceTask` is used in some `TestCase`, that `TestCase` will also get affected. In this case, the test cases *testShortestStrategy* and *testNoRoutesForShortestStrategy* both use this `MockOperation`, and thus would get affected as well. However, to simplify the discussion we do not discuss this scenario in detail here.

8.4 Chapter Summary

This chapter provides the details about the impact analysis approach we employ to perform impact analysis across models and tests. We presented the concept of impact rules and how they utilize various dependency relations to react on changes. The impact rules produce a set of impact reports, which exhibit the list of affected elements in the result of change, how they are affected, and what next changes they trigger. This list of impact reports although show the affected test elements along with other affected elements. However, to decide how these affected test elements should be treated, we still need a classification mechanism to distinguish between various types of affected test elements. In the next chapter, we discuss the test classification mechanism we use to support our regression testing approach.

9

Regression Test Classification

The classification of tests to support regression testing is the last activity in Step 2 of our approach, presented in Section 4.1 of Chapter 4. The affected test elements obtained after the change impact analysis activity are required to be distinguished based on how they will be used during regression testing. Thus, a distinction is necessary between the test elements which became obsolete, are required to be executed during regression testing, and are unaffected. Our hypothesis, presented in Chapter 4, stated that the classification of an affected test element depends on four factors: the *type of the applied change*, the *type of the affected test element* determined by the impact analysis, the *classification type*, and the applicable *test classification conditions*. Based on the hypothesis, in the following section, we present our test classification approach that uses rules based on these factors to classify the affected test elements.

9.1 Rule-based Test Classification

To classify the test cases for regression testing, we propose a set of rules, which decide how an affected test element should be classified. The test classification rules are used to analyze the affected test elements to check how they are affected and if they are potentially affecting other test elements due to various conditions.

Thus, the test classification rules analyze the impact reports produced by the impact analysis performed earlier, check if a specified affected test element exists, and assess it for various specified conditions. If the conditions satisfy, the respective test element is classified according to the classification type specified in the rule.

Our analysis of the state of the art regression testing approaches reflect that they implement the test classification conditions as part of the source code. Thus, adding new conditions, covering new test elements, or incorporating new test specification and implementation languages require changes inside the source code of the tool. This results in lack of ability to support extensibility and customization of the test classification logic. Our rule-based test classification approach improves on this aspect and uses the concept of easily extendible rules. These rules are also able to process various conditions, asses dependency relations, and query various test elements, impact reports, and other model elements. The structure of the *test classification rules* is very similar to the dependency detection rules and the impact analysis rules presented earlier in Chapter 6 and Chapter 8.

9.1.1 Concept of Test Classification

We further explain the concept of the rule-based test classification using an example of a `TestCase` and a `MockOperation`. If a `MockOperation` is deleted during the impact analysis process, it will be considered `Obsolete`. However, any `TestCase`, which uses this `MockOperation` should be selected for retest during the regression testing, by analyzing and changing the `TestCase` accordingly. Figure 9.1 elaborates on the concept of impact analysis by using this example.

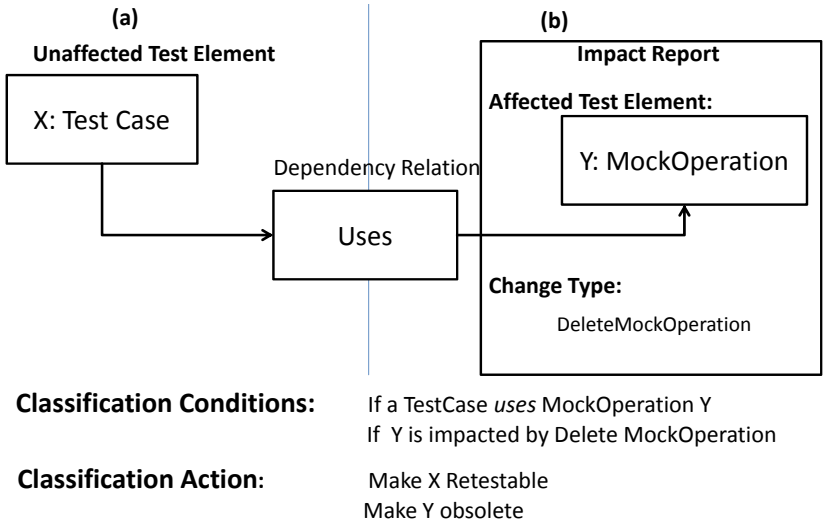


Figure 9.1: Classifying a Test Element.

The right hand side of Figure 9.1 (b) depicts an impact report, which presents an affected `MockOperation` Y, where the result change type in the impact report is *Delete MockOperation*. There is a test case `X`, which was initially unaffected during the impact analysis process. The test classification conditions present in the lower part of Figure 9.1 specify two constraints. First is that if a `TestCase` exist which uses a `MockOperation`

`Y`. The left hand side shows an element `X`, which is a `TestCase` and a dependency relation between `X` and `Y` of type *uses* exist, making the classification condition true. The second condition is that Y should be an affected element and the change type in the impact report should be *Delete MockOperation*. This condition also satisfies. Since both conditions satisfy, the test case `X` can be classified as `Retestable` and the `MockOperation Y` can be classified as `Obsolete`, as specified in the *classification action*.

9.1.2 Test Classification Rules

To realize the above presented scenario, test classification rules are required to be defined and used. Similar to the impact analysis rules, the test classification rules require the evaluation of a set of conditions. If these conditions are met, the result is a classification action specify the test element to be classified and the classification type that should be assigned to the test element. Thus, the most important components of a test classification rule are as follows: (1) the specification of test elements, impact reports, and other model elements, (2) classification conditions, and (3) test classification actions. Another important aspect is the test classification type to be specified in the test classification actions. We use the classification scheme of Leung and White [LW89], which consists of all the types required for the test classification. These types are discussed later with the *test classification actions*.

The structure of test classification rules is similar to the dependency detection rules and impact analysis rules, as they use the same analogy of evaluating logic conditions and performing classifications actions based on these conditions. Thus, similar to the impact analysis rules, a test classification rule also consist of the *element definitions*, the *classification conditions*, and the *test classification action* parts. For the purpose of demonstration, Listing 9.1 presents an excerpt of a test classification rule, which classifies a `TestComponent` as `PartiallyRetestable`.

Element Definition– part specifies the model elements, test elements, and impact reports to be used by the rules. The `ElementDefinition` part of a test classification rule should specify one or more impact reports or refer to already classified test elements, which are to be accessed inside the classification rule. These impact reports are produced during the impact analysis process and are used to analyze the affected test elements. Line 2-6 of the impact rule shown in Listing 9.1 is the element definition part of the rule and specifies various elements required by the rule.

Test Classification Conditions– consist of logical conditions and other predefined operations on models. These include checking the parent-child relationships among elements, checking the name similarity and matching the attribute values. For the creation of classification conditions, each test element is required to be analyzed to identify the cases in which it could be classified according to a classification type. Line 7-15 of Listing 9.1 show the test classification conditions of the presented rule, which are required to be evaluated to true for the classification of the specified elements.

Test Classification Actions– assign a test classification type to a classified test element. As discussed earlier, we use the test classification scheme of Leung and White [LW89] for various test classification types. Thes classification types proposed

by Leung and White [LW89] are `Obsolete`, `Reusable`, `Retestable`, and `New`. The `Obsolete` and `New` test elements are self explanatory. The `Retestable` test elements are affected and required for regression testing. The `Reusable` test elements are valid but not required for regression testing.

Listing 9.1: A Rule To Classify a TestComponent as Partially Retestable.

```
1  <RuleModel:Rule Description="Makes␣a␣TestComponent␣Retestable" RuleID="TCR003">
2    <Elements Type="ClassifiedTestElement" Alias="e1"/>
3    <Elements Type="MockOperation" Alias="e2"/>
4    <Elements Type="ClassifiedTestElement" Alias="e3"/>
5    <Elements Type="MockOperation" Alias="e4"/>
6    <Elements Type="TestComponent" Alias="e5"/>
7      <Conditions>
8          <BaseConditions Type="ModelEquals" Source="e1::afftectedTestElement" Target="
               e2" Value=""/>
9      <BaseConditions Source="e1::cType" Target="" Value="reusable"/>
10     <BaseConditions Source="e3::cType" Target="" Value="retestable"/>
11     <LogicConditions>
12       <BaseConditions Type="ModelRelatedTo" Source="e5" Target="e2" Value="
             Containment"/>
13       <BaseConditions Type="ModelRelatedTo" Source="e5" Target="e4" Value="
             Containment"/>
14     </LogicConditions>
15   </Conditions>
16  <Actions ActionType="TestClassification" ImpactedElement=""/>
17   <Actions xsi:type="RuleModel:TestClassificationAction" SourceElement="e5"
         classificationType="partiallyRetestable"/>
18 </RuleModel:Rule>
```

However, the test elements can also be composite, such as *test components*, are composed of *mock operations*. Therefore, to classify the composite test elements affected by changes, we use another classification type `PartiallyRetestable` as well. A test element is considered `PartiallyRetestable` if atleast one of the constituent of a composite test element is affected and atleast one of its constituents remains unaffected.
types required for the classification of test elements. Line 16-18 in Listing 9.1 show the specification of a test classification action, which classifies a `TestComponent` as `Obsolete`

A test classification action creates `TestClassificationReport/s`, which specify the classified test element, the classification type, and any relevant impact reports used for deciding the test classification. The meta-model presented in 9.2 defines the elements, relevant to the implementation of *test classification reports*.

The Test Classification Meta-Model– depicts the element `TestClassificationReport` and its interaction with various other relevant classes. Thus, according to Figure 9.2, a `TestClassificationReport` has three primary attributes. A *name*, a *creationDate*, and an element `TestClassificationType` applicable to a the test element to be classified. It also refers to a UTP `TestElement` which is to be classified in the `TestClassificationReport`. The `TestClassificationType` is an enumeration literal, which specifies the types from the test classification scheme of Leung and White [LW89].

The class `TestClassificationReport` also refers to the `TestClassificationRule` class, which resulted in the creation of a concrete `TestClassificationReport`. It

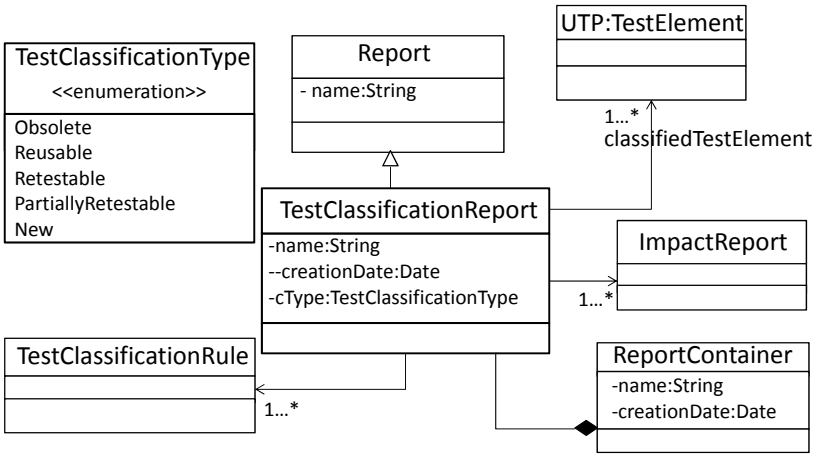


Figure 9.2: The Test Classification Meta-Model.

also refers to any impact reports, which were being processed during the test classification. The class `ReportContainer` packages all the test classification reports in one container inside EMFTrace GUI. Figure 10.6 shows a screen shot of EMFTrace showing a `TestClassificationReportContainer` with a `TestClassificationReport` element, which classifies the test element as `Reusable`. The actual specification of the test classification rules is similar to the dependency detection and impact rules.

9.1.3 Test Classification Process

Based on the example and concept presented above, we present the process of test classification in Figure 9.3. According to Figure 9.3, the test classification process starts by selecting the impact reports produced by the impact analysis process and the required test classification rules. The *test classification rules* are based on the same analogy presented in Figure 9.1, and are discussed later in the subsequent section. The actual test classification process takes the test classification rules and processes them one by one, until all the rules are covered. For the processing of rules, first all the elements specified in the rule are obtained by querying the models according to the specified type. After that the *classification conditions* specified in the rule are accessed.

Listing 9.2: Pseudo Code for processing Test Classification Rules.

```
1 classifyTests(rules, models) {
2   for(int i = 0; i < rules.size(); i++) {
3         do{
4      results=results+processTestClassificationActions(rule);
5         }
6         while(applyRule(selectedRules, models) );
7     }
8   removeAndMergeDuplicates();
9   fillReportContainer(results);  }
```

The querying of elements and the processing of conditions is achieved similar to the impact rules presented in the Chapter 8. Therefore, the test classification rules can also specify various logic conditions and can use pre-defined operation discussed in Section 8.1 of Chapter 8. If the conditions specified in the rule are satisfied, then a

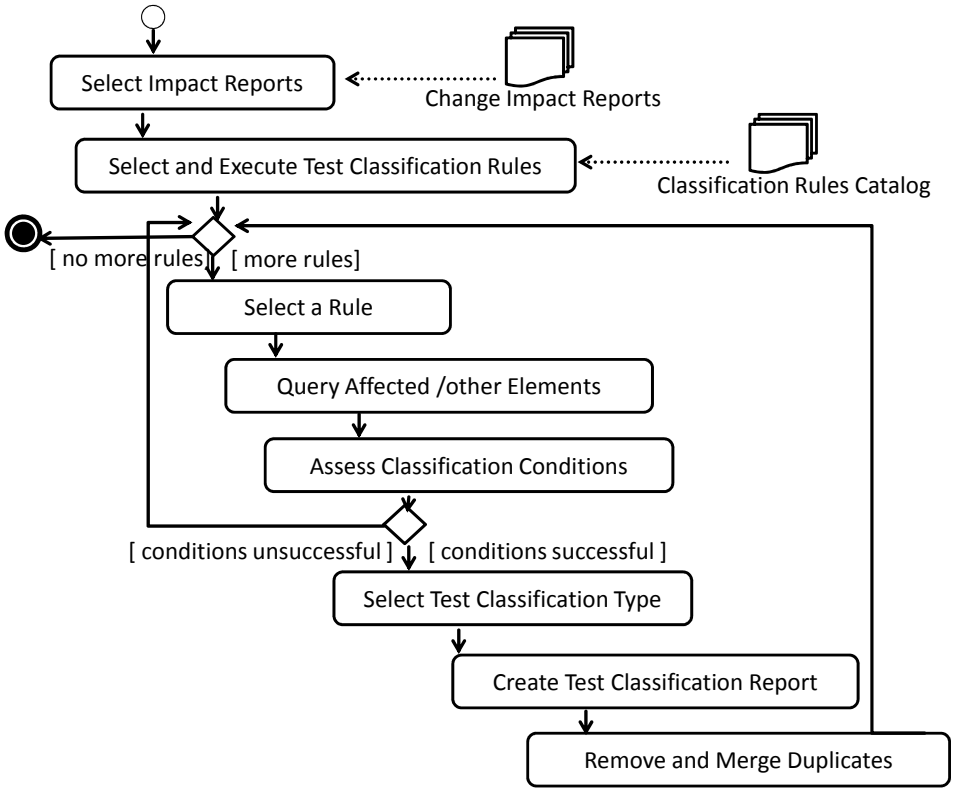


Figure 9.3: Process for the Classification of Test Elements.

`TestClassificationReport` is created. The `TestClassificationReports` classify the selected test element, which fulfills the conditions according to the specified classification type. Finally, any duplicate *test classification reports* are merged to remove duplicates.

Listing 9.2 presents the pseudo code, which reflects how the test classification rules are processed. The operation *classifyTests()* takes the test classification rules and set of models. The models include system and test models as well as impact reports produced during the impact analysis earlier.

For each rule, the rule is applied to evaluate all the conditions and retrieve the results meeting these conditions. The results are then used to process the test classification actions and create the test classification reports. In the Line 8 of Listing 9.2, the results are then analyzed for any duplicate test classification reports, which might be produced during the result processing. Finally, Line 8 fills the `TestClassificationContainer` with the created test classification reports Listing 9.3 presents the pseudo code to process the test classification results and create the required test classification reports.

Listing 9.3: Pseudo Code to Process Test Classification Results.

```
1 processTestClassificationResults(rule, tuples,index){
2  for(int j = 0; j < tuples.length; j++){
3    src= tuples.getSource(index);
4    dst=tuples.getTarget(index);
5    ir=tuple.getImpactReports(index);
6 //Creates ClassificationReports;
7       create  tcr;
8    set tcr.impactReports=impactReports();
```

```
9    set tcr.AffectedTestElement =src;
10 //ct is TestClassificationType specified in input rule
11    ct=rule.getClassificationType();
12    switch(ct){
13 //Decides Which Classification Type is applicable
14     case "added":{
15     set tcr.classificationType=NEW; break; }
16     case "obsolete":{
17       set tcr.classificationType=OBSOLETE; break;  }
18     case "reusable":{
19      set tcr.classificationType=RESUABLE; break;  }
20     case "retestable":{
21       set tcr.classificationType=RETESTABLE; break;  }
22     case "partiallyRetestable":{
23       set tcr.classificationType=PARTIALLY_RETESTABLE; break; } } }
24  results.add(tcr);
25  return results;   }
```

These test classification reports can the be analyzed by the test analyst to check how many test cases are required for retest. This provides an early assessment of the effort required to maintain, execute, and analyze the tests if the change is introduced. The feasibility of the change can be assessed by considering the effort required for testing, which consumes a higher percentage of project budget.

9.2 Classification of UTP Test Elements

As we use UTP test models for the test specification in our approach, they are required to be analyzed to identify the classification conditions under which the UTP elements are classified. Therefore, we analyze the UTP *test architecture* and *test behavior* elements by identifying the classification conditions for each classification type discussed earlier.

9.2.1 Classification of UTP Test Architecture Elements

We analyzed the UTP *test architecture* elements for the conditions in which they will be classified. We first present some initial constants and then below we present UTP elements and their classification conditions.

$$\text{Let } IR = (ir_1, ir_2, ir_3 ... ir_n) \text{ be a set of impact reports of size } n. \tag{9.1}$$

$$\text{Let } T_M = (tm_1, tm_2, tm_3 ... tm_m) \text{ be a set of baseline test models of size } m. \tag{9.2}$$

$$\tag{9.3}$$

Classifying a TestModel– A `TestModel` contains all test related elements of UTP, thus is a container of all the test related elements. A `TestModel` in UTP can be classified for regression test selection as follows.

Case Obsolete: A `TestModel` is considered `Obsolete` if the following holds true.

Let tm be an element `TestModel` in UTP. (9.4)

Let $D_{tm} = d_1, d_2, d_3...d_k$ be the set of dependency relations of tm `TestModel` of size k. (9.5)

$\exists\, r \in IR \mid changeType(r) =$ `Delete TestModel` $\wedge$

$source(r) = tm.$ (9.6)

$\vee$

$\exists\, r \in IR \mid changeType(r) =$ `Delete TestPackage` for $tp \in T_M.$ (9.7)

$\mid type(tp) =$ `TestPackage` $\wedge$ (9.8)

$\exists\, d \in D_{tm} \mid type(d) = Containment\ \wedge\ target(d) = tp..$ (9.9)

According to the above mentioned equations, tm will be obsolete in two different scenarios. Firstly, if the `TestModel` tm is deleted itself. Secondly, when a `TestPackage` corresponding to tm is deleted. Thus, according to Equation 9.6, tm is considered `Obsolete` if an `ImpactReport` exists that contains a change type `Delete TestModel` for tm.

Moreover, according to Equations 9.7, 9.8, and 9.9, if a `TestPackage` that has a dependency relation of type *Containment* with tm is deleted, tm will be considered `Obsolete`. Listing 9.4 shows the test classification rule that satisfies the test classification condition defined by Equation 9.7. Similar test classification rules for other test classification conditions presented in this chapter are also developed and presented in Table E.3 in Appendix E.

Listing 9.4: A Test Classification Rule to Classify a TestModel.

```
1  <RuleModel:Rule Description="A␣TestModel␣is␣Obsolete␣if␣its␣corresponding␣TestPackage
       ␣is␣Deleted" RuleID="TCR025">
2    <Elements Type="Model" Alias="e1"/>
3    <Elements Type="TestModel" Alias="e2"/>
4    <Elements Type="Package" Alias="e3"/>
5    <Elements Type="ImpactReport" Alias="e4"/>
6    <Actions xsi:type="RuleModel:TestClassificationAction" ActionType="
         TestClassification" ResultType="obsolete" SourceElement="e1" TargetElement="e1"
          classificationType="retestable"/>
7    <Conditions>
8      <BaseConditions Type="ModelEquals" Source="e2::base_Package" Target="e1"/>
9      <BaseConditions Type="ModelRelatedTo" Source="e1" Target="e3" Value="Containment"
           />
10     <BaseConditions Type="ValueEndsWith" Source="e3::name" Target="" Value="TP"/>
11     <BaseConditions Type="ReferenceExists" Source="e4::AffectedElements" Target=""
         Value="e3"/>
12     <LogicConditions Type="Or">
13       <BaseConditions Source="e4::Solution" Target="" Value="Delete␣TestPackage"/>
14       <BaseConditions Source="e4::Solution" Target="" Value="Delete␣Package"/>
15     </LogicConditions>
16   </Conditions>
17 </RuleModel:Rule>
18 }
```

Case Retestable: The cases in which a `TestModel` tm is considered `Retestable` is

defined as follows.

$$\neg(status(tm) = \texttt{Obsolete}) \wedge \tag{9.10}$$
$$\exists\, tc \in T_M \mid type(tc) = \texttt{TestContext}. \tag{9.11}$$
$$\exists\, d \in D_{tm} \mid type(d) = Containment \ \wedge \ target(d) = tc. \tag{9.12}$$
$$status(tc) = \texttt{Retestable}. \tag{9.13}$$
$$\exists\, tp \in T_M \mid type(tp) = \texttt{TestPackage} \ \wedge \ status(tp) = \texttt{Retestable}. \tag{9.14}$$
$$\exists\, d1 \in D_{tm} \mid type(d1) = Containment \ \wedge \ target(d1) = tp. \tag{9.15}$$

If the `TestModel` tm is not classified as `Obsolete` (9.10). Or the `TestContext` (9.11, 9.12, and 9.13) and `TestPackage` (9.14, 9.15) corresponding to this `TestModel` is `Retestable`.

Case PartiallyRetestable: A `TestModel` tm is `PartiallyRetestable` in the following cases.

$$\exists\, tc \in T_M \mid type(tc) = \texttt{TestContext}. \tag{9.16}$$
$$\exists\, d \in D_{tm} \mid type(d) = Containment \ \wedge \ target(d) = tc. \tag{9.17}$$
$$status(tc) = Retestable. \tag{9.18}$$
$$\exists\, tp \in T_M \mid type(tp) = \texttt{TestPackage} \ \wedge \ status(tp) = \texttt{Retestable}. \tag{9.19}$$
$$\exists\, d1 \in D_{tm} \mid type(d1) = Containment \ \wedge \ target(d1) = tp. \tag{9.20}$$

According to the above mentioned equations, a `TestModel` is considered `PartiallyRetestable` if either the `TestContext` (9.16, 9.17, 9.18) or the `TestPackage` (9.20) corresponding to this `TestModel` are `PartiallyRetestable`.

Case Reusable: A `TestModel` is `Reusable` if none of its elements are affected by any change. The following equation describes this scenario.

$$\neg(\exists\, r \in IR \mid changeType(r).source.type = \texttt{TestModel} for\ tm). \tag{9.21}$$

Case New: Following equation describes a scenario when a `TestModel` is considered `New`.

$$\exists\, r \in IR \mid changeType(r) = \texttt{Add TestModel} \text{ for } tm. \tag{9.22}$$

Classifying a Test Package– In the following, we discuss the classification of the element `TestPackage` tp in UTP. For this we first define the required constants and then provide definitions for various classification scenarios.

$$Let\ tp \in T_M \mid type(tp) = \texttt{TestPackage}. \tag{9.23}$$
$$Let\ tc \in T_M \mid type(tc) = \texttt{TestContext}. \tag{9.24}$$
$$Let\ TCM = (tcm_1, tcm_2, tcm_3 ... tcm_n),\ where \bigwedge_{i=0} n\ type(tcm_i) = \texttt{TestComponent}. \tag{9.25}$$

$$Let\ D_{tp} = (d_1, d_2, d_3 ... d_m),\ where \bigwedge_{j=0} m\ type(d_j) = \texttt{DependencyRelation}$$

for the `TestPackage`. (9.26)

$$\bigwedge_{k=0} n\ (\bigwedge_{l=0} m(\exists\ d_l \in D_{tp} \mid type(d_l) = Containment,\ target(d_l) = tc_k)). \quad (9.27)$$

$$\exists\ d \in D_{tp} \mid type(d) = Containment, target(d) = tc. \quad (9.28)$$

Case Obsolete: A `TestPackage` *tp* is considered `Obsolete` in the following cases.

$$\exists\ r \in IR \mid changeType(r) = \texttt{Delete TestPackage} \wedge source(r) = tp \quad (9.29)$$

$$\vee$$

$$status(tc) = \texttt{Obsolete}. \quad (9.30)$$

According to the Equation 9.29, a `TestPackage` *tp* in UTP is considered `Obsolete` if an `ImpactReport` exists which consists of a change *Delete TestPackage* for *tp*. A `TestPackage` *tp* is considered `Obsolete` as well when the `TestContext` belonging to *tp* is `Obsolete`, as shown by Equation 9.30.

Case Retestable: A `TestPackage` *tp* is considered `Retestable` in the situation when its `TestContext` is either `Retestable`, as shown by Equation 9.31.

$$status(tc) = (\texttt{Retestable}. \quad (9.31)$$

Case PartiallyRetestable A `TestPackage` *tp* is considered `PartiallyRetestable` in the following cases.

$$status(tc) = (\texttt{PartiallyRetestable} \vee \texttt{New}) \quad (9.32)$$

$$\vee$$

$$\bigwedge_{i=0} n\ status(tc_i) = (\texttt{Obsolete} \vee \texttt{Retestable} \vee \texttt{PartiallyRetestable} \vee New). \quad (9.33)$$

According to the above presented equations, a `TestPackage` *tp* is considered `PartiallyRetestable` if either its `TestContext` is `PartiallyRetestable` or `New`. Moreover if any of its test components are affected,then as well it well be considered `PartiallyRetestable`. That is, that the affected `TestComponent` is either `Obsolete`, `Retestable`, `PartiallyRetestable`, or `New`.

Case Reusable: A `TestPackage` *tp* is considered `Reusable` if none of its test components and its `TestContext` is affected by any change as presented by the following

equations.

$$\neg(\exists\, r1 \in IR \mid changeType(r1).source.type = \texttt{TestPackage} for\ tp) \tag{9.34}$$

$$\vee$$

$$\neg(\exists\, r2 \in IR \mid changeType(r2).source.type = \texttt{TestContext} for\ tc) \tag{9.35}$$

$$\vee$$

$$\bigwedge_{i=0} n\ \neg(\exists\, r3 \in IR \mid changeType(r3).source.type = \texttt{TestComponent} for\ tc_i). \tag{9.36}$$

Case New: A `TestPackage` tp is considered `New` if a change type for adding it exists, as shown in the following equation.

$$\neg(\exists\, r \in IR \mid changeType(r) = \texttt{Add TestPackage} \text{ for } tp). \tag{9.37}$$

Classifying a SUT– For the classification of a `SUT` sut, we first define the following constants to define sut and its contained operations. We then present different classification cases for classifying `SUT`.

$$Let\ sut \in T_M \mid type(tc) = \texttt{SUT}. \tag{9.38}$$

$$LetM_{op} = (op_1, op_2, op_3 ... op_n), where \bigwedge_{i=0} n\ type(op_i) = \texttt{Operation}. \tag{9.39}$$

$$LetD_{sut} = (d_1, d_2, d_3 ... d_m), where \bigwedge_{j=0} mtype(d_j = \texttt{DependencyRelation}. \tag{9.40}$$

$$\bigwedge k = 0n\ (\bigwedge_{l=0} m(\exists\, d_l \in D_{sut} \mid type(d_l) = Containment,\ target(d_l) = op_k)). \tag{9.41}$$

Case Obsolete: The sut is considered `Obsolete` in the following cases.

$$\exists\, r \in IR \mid changeType(r) = \texttt{Delete SUT} \text{ for } sut. \tag{9.42}$$

The change type `DeleteSUT` would be triggered in response to the deletion of its corresponding process or the process class in the system class diagram.

Case Retestable: The sut is considered `Retestable` in the following cases.

$$\bigwedge_{k=0} n\ \neg(op(k) \in M_{op} = \texttt{Reusable}). \tag{9.43}$$

According to Equation 9.43, a `SUT` sut is considered `Retestable` if all the operations inside `SUT` are affected by any change. We define this by using the $\neg$ property and this states that all the operations belonging to the sut should not be `Reusable`, that is, affected by a change.

Case PartiallyRetestable: The sut is considered `PartiallyRetestable` in the fol-

lowing cases.

$$\exists\, op1 \in M_{op} \mid status(op1) = (\texttt{Obsolete} \vee \texttt{Retestable}) \wedge \tag{9.44}$$

$$\exists\, op2 \in M_{op} \mid status(op2) = \texttt{Reusable}. \tag{9.45}$$

$$\vee$$

$$\exists\, op3 \in M_{op} \mid status(op3) = \texttt{New}. \tag{9.46}$$

According to the above presented equations, there are two scenarios in which sut would be considered `PartiallyRetestable`. In the first scenario, as defined by the Equation 9.44, and Equation 9.45, the sut would be considered `PartiallyRetestable` if atleast one operation in sut is either `Obsolete` or `Retestable` and atleast one operation in sut is `Reusable`. This means that atleast one operation affected by a change and atleast one operation unaffected by a change should exist in sut to make it `PartiallyRetestable`. In the second scenario, the sut would be considered `PartiallyRetestable`, if a `New` operation is added to it, as new test cases would be required to test this operation.

Case Reusable: An `SUT` sut is considered `Reusable` if it remains unaffected and none of its operations are affected by a change. The following equations describe the scenarios, when an `SUT` is considered `Reusable`.

$$\neg(\exists\, r \in IR \mid changeType(r).source.type = \texttt{SUT} for\ st). \tag{9.47}$$

$$\bigwedge_{i=0} n\ (op_i \in M_{op}\ status(op_i) = \texttt{Reusable}). \tag{9.48}$$

Case New: A `SUT` st is considered `New` if the following holds for it.

$$\exists\, r \in IR,\ \exists\, tp \in T_M \mid type(tp) = \texttt{TestPackage}. \tag{9.49}$$

$$changeType(r) = \texttt{Add SUT} \wedge\ source(r) = tp. \tag{9.50}$$

Classifying a TestContext– To classify a `TestContext` in UTP, we first define it and its constituents in the following.

$$Let\ tc \in T_M \mid type(tc) = \texttt{TestContext}. \tag{9.51}$$

$$Let\ TC_t = (t1_1, t_2, t_3 ... t_n), where \bigwedge_{i=0} n\ type(op_i) = \texttt{TestCase}. \tag{9.52}$$

$$Let\ D_{tc} = (d_1, d_2, d_3 ... d_m), where \bigwedge_{j=0} m\ type(d_j = \texttt{DependencyRelation}. \tag{9.53}$$

$$\bigwedge k = 0n\ (\bigwedge_{l=0} m(\exists\, d_l \in D_{tc} \mid type(d_l) = Containment,\ target(d_l) = t_k)). \tag{9.54}$$

Case Obsolete: A `TestContext` tc is considered `Obsolete` in the following cases.

$$\exists\, r \in IR \mid changeType(r) = \texttt{Delete TestContext} \text{ for } tc \tag{9.55}$$

$$\vee$$

$$\bigwedge_{i=0} n\ status(t_i) = \texttt{Obsolete}. \tag{9.56}$$

According to the above mentioned equations, a `TestContext` tc will be considered `Obsolete` in two cases. The first case is when a change operation *Delete TestContext* is explicitly available in the set of impact reports IR (9.55). The other case is when all of the test cases inside the `TestContext` tc are already `Obsolete` 9.56.

Case Retestable: Since a `TestContext` is a container of all the test cases to test a process, it would be considered `Retestable` if all the test cases inside that `TestContext` are `Retestable`. The Equation 9.57 describe this case for a `TestContext` tc.

$$\bigwedge_{i=0} n\ status(t_i) = \texttt{Retestable}. \tag{9.57}$$

PartiallyRetestable: A `TestContext` is considered `PartiallyRetestable` in the following cases.

$$\exists\, t1, t2 \in TC \mid status(t1) = (\texttt{Reusable} \vee \texttt{Obsolete}) \ \wedge\ status(t2) = \texttt{Retestable} \tag{9.58}$$

$$\vee$$

$$\exists\, r \in IR \mid changeType(r) = \texttt{Add TestCase for } tc. \tag{9.59}$$

$$\tag{9.60}$$

A `TestContext` tc is considered `PartiallyRetestable` in two different scenario. Firstly, if atleast one `TestCase` inside that `TestContext` is `Reusable`, and atleast one `TestCase` is either `Retestable` or `Obsolete`. Secondly, if atleast one `TestCase` is added inside tc it would become `PartiallyRetestable` as this `TestCase` has to be executed for regression testing.

Case Reusable: A `TestContext` tc is considered `Reusable` if all the test cases inside tc are `Reusable` and no change is applied on the `TestContext` as defined by the following equations.

$$\neg(\exists\, r \in IR \mid changeType(r).source.type = \texttt{TestContext}\, for\ tc). \tag{9.61}$$

$$\bigwedge_{i=0} nstatus(t_i) = \texttt{Reusable}. \tag{9.62}$$

Case New: A `TestContext` $tc$is classified as new if an impact report exists for its addition in a `TestPackage`

$$\exists\, r \in IR,\ \exists\, tp \in T_M \mid type(tp) = \texttt{TestPackage}. \tag{9.63}$$

$$changeType(r) = \texttt{Add TestContext} \ \wedge\ source(r) = tp. \tag{9.64}$$

Classifying a Mock Operation– In the following, we define how a `MockOperation` inside a `TestComponent` can be classified. Since a `MockOperation` is not composed of further complex sub-elements, there is no such case in which `MockOperation` can be classified as `PartiallyRetestable`. For the other classification

types, however, we define the conditions in the following. We first define a `MockOperation` and required constants and then present its classification scenarios.

$$Let\ mo \in T_M \mid type(mo) = \texttt{MockOperation} \tag{9.65}$$

$$Let\ PAR_{mo} = (p_1, p_2, p_3 ... p_n), where \bigwedge_{k=0} n\ type(p_k) = \texttt{Parameter}. \tag{9.66}$$

$$Let\ D_{mo} = (d_1, d_2, d_3 ... d_m), where \bigwedge_{i=0} m\ type(d_i) = \texttt{DependencyRelation}. \tag{9.67}$$

$$\bigwedge_{i=0} n\ (\bigwedge_{j=0} m(\exists\ d_j \in D_{mo} \mid type(d_j) = Containment,\ target(d_j) = p_i)). \tag{9.68}$$

$$Let\ PRE_{mo} = (pr_1, pr_2, pr_3 ... pr_x), \bigwedge_{y=0} x\ type(pr_y) = \texttt{Constraint}. \tag{9.69}$$

$$\bigwedge_{a=0} x\ (\bigwedge_{b=0} m(\exists\ d_b \in D_{mo} \mid type(d_b) = Containment,\ target(d_b) = pr_a)). \tag{9.70}$$

$$Let\ POS_{mo} = (po_1, po_2, po_3 ... po_t), \bigwedge_{s=0} t\ type(po_s) = \texttt{Constraint}. \tag{9.71}$$

$$\bigwedge_{x=0} t\ (\bigwedge_{y=0} m(\exists\ d_y \in D_{mo} \mid type(d_y) = Containment,\ target(d_y) = pr_x)). \tag{9.72}$$

Case Obsolete: A `MockOperation` *mo* is considered as `Obsolete` if the following holds to true.

$$\exists\ r \in IR \mid changeType(r) = \texttt{Delete MockOperation} \wedge source(r) = mo. \tag{9.73}$$

A `MockOperation` would be considered `Obsolete` if a change operation *Delete MockOperation* is applied atleast once on the `MockOperation`. This is similar to the case of the deletion of an `Operation` as defined by [BLS02, FR11] in their works. However, we do not consider a `MockOperation` as `Obsolete` if a `Parameter` is added or deleted from it as done by [BLS02] and [FR11]. The reason to do so is that from the test point of view, if a parameter is added or deleted from a `MockOperation`, it would be required to retest the test cases using this `MockOperation`. Hence, instead of making it `Obsolete`, we would classify it as `Retestable` in case of addition or deletion of parameters, as also discussed later.

Case Retestable: A `MockOperation` *mo* will be considered `Retestable` in the fol-

lowing cases.

$$\exists\, r1 \in IR \mid changeType(r1) = \texttt{Add Parameter} \;\wedge\; source(r1) = mo \tag{9.74}$$

$$\vee$$

$$\exists\, r2 \in IR, \exists\, p1 \in PAR_{mo} \mid changeType(r2) = \texttt{Delete Parameter} \wedge source(r2) = mo \;\wedge\; target(r2) = p1 \tag{9.75}$$

$$\vee$$

$$\exists\, r3 \in IR,\; \exists\, pr \in PRE_{mo} \mid changeType(r3) = \texttt{Change PreCondition} \wedge source(r3) = mo, target(r3) = pr \tag{9.76}$$

$$\vee$$

$$\exists\, r4 \in IR,\; \exists\, po \in POS_{mo} \mid changeType(r4) = \texttt{Change PostCondition} \wedge source(r4) = mo, target(r4) = pr. \tag{9.77}$$

As defined in the above presented equations, a `MockOperation` in UTP will be classified as `Retestable` in the following different cases.

In the first case, a `MockOperation` will be considered as `Retestable` if a `Parameter` is added inside that `Operation`. The addition of a `Parameter` will be recognized if a corresponding `ImpactReport` exists where source of the impact report is mo and the *changeType* of that `ImpactReport` is *Add Parameter*, as defined by the Equation 9.74.

In the second case, if a parameter is deleted from an `MockOperation`, in that case as well, it would be considered `Retestable` and all the test cases calling mo would be selected for a retest as well. The Equation 9.75 describes this scenario. In the third and forth case, if the `Precondition` or the `PostCondition` of the `MockOperation` mo is changed, in this case as well mo would be considered `Retestable` as suggested by the Equation 9.76 and Equation 9.77.

Further, a `MockOperation` will also be considered `Retestable` if any of its own properties or any property of its `Parameter` is changed. The following equations present two of such cases. In the first case, if a property of the mo is changed, for example, if mo is made abstract, then as well, mo would be considered `Retestable`, as suggested by Equation 9.78. In the other case, if type of a `Parameter` of mo is changed then it would be considered `Retestable`, as suggested by Equation 9.79

$$\exists\, r5 \in IR \mid changeType(r5) = \texttt{MakeMockOperationAbstarct} \wedge source(r5) = mo \tag{9.78}$$

$$\vee$$

$$\exists\, r6 \in IR, \exists p2 \in PAR_{mo} \mid changeType(r6) = \texttt{ChangeParameterType} \wedge source(r6) = p2. \tag{9.79}$$

Case Reusable: A `MockOperation` is considered`Reusable` if it remains unaffected and there are no corresponding impact reports for it as defined in the following.

$$\neg(\exists\, r \in IR \mid changeType(r).source.type = \texttt{MockOperation} for\ mo). \tag{9.80}$$

Case New: A `MockOperation` mo is considered `New` if an `ImpactReport` for its addition exists, as suggested by Equation 9.82 and Equation **??**. According to the equation, the *source* of the `ImpactReport` has to be a `TestComponent` in which the `MockOperation` has to be added.

$$\exists\, r \in IR,\ \exists\, tc \in T_M \mid type(tc) = \texttt{TestComponent}. \tag{9.81}$$

$$changeType(r) = \texttt{Add MockOperation} \wedge source(r) = tp. \tag{9.82}$$

Classifying a Test Component– To classify a UTP `TestComponent` tc, the changes applied to a test component and its constituent `MockOperations` have to be analyzed. The following set of equations defines various cases in which a `TestComponent` can be classified.

$$Let\ tc \in T_M \mid type(tc) = \texttt{TestComponent}. \tag{9.83}$$

$$Let\ M_{tc} = (m_1, m_2, m_3 ... m_n),\ where \bigwedge_{i=0} n\ type(m_i) = \texttt{MockOperation}. \tag{9.84}$$

Equation 9.83 and 9.84 define the primitive concepts. They define a `TestComponent` tc belonging to the set of *baseline test models*. Further, a set of `MockOperations` M_{tc} exists for the `TestComponent` tc. Thus, if a change $c \in C$ is applied, it would make the tc either `Obsolete`, `Reusable`, `Retestable`, `PartiallyRetestable`, or `New` in the following cases.

Case Obsolete: The following equation defines the case when tc is `Obsolete`.

$$\exists\, r \in IR\ for\ tc \mid changeType(r) = \texttt{Delete TestComponent}.. \tag{9.85}$$

The origin of the change type *Delete TestComponent* can vary, as it depends on the dependency relations, which are exercised.

Case Retestable: The element tc will be `Retestable` if the following holds true.

$$\forall m \in M_{tc}\ m \in (R \vee P). \tag{9.86}$$

This means that a test component is considered `Retestable` if all of its mock operations are also `Retestable` or `PartiallyRetestable`. Otherwise, tc will be `PartiallyRetestable` under the conditions discussed in the following. If a `TestComponent` is `Retestable`, all the test cases using this `TestComponent` would also be considered affected.

Case PartiallyRetestable: The following equations define when a `TestComponent`

is considered PartiallyRetestable.

$$\exists\, m \in M_{tc}, n \in M_{tc}, \mid m \in \texttt{Reusable} \;\wedge\; n \in \texttt{Retestable}. \tag{9.87}$$

$$\vee$$

$$\exists\, r \in IR \mid changeType(r) = \texttt{PropertyUpdate} \text{ for } tc \tag{9.88}$$

$$\vee$$

$$\exists\, r \in IR \mid changeType(r) = \texttt{Add MockOperation} \text{ for } tc.. \tag{9.89}$$

Thus, according to the above presented equations, a TestComponent will be considered PartiallyRetestable in three different cases. The first case is when atleast one of its constituent mock operations is affected and atleast one of its constituent remains unaffected 9.87.

The second case is one any property of the TestComponent is changed, for example, a *rename* change operation is applied 9.88. Finally, the third case is when any new MockOperation is added in a TestComponent 9.89.

Case New: Finally, tc will be considered as New if a *Add TestComponent* change type exists in any impact report as presented in Equation 9.90.

$$\exists\, r \in IR \mid changeType(r) = \texttt{Add TestComponent} source(r) = tc. \tag{9.90}$$

Case Reusable: To define the reusebility of a test element we introduce the concept of an *immaculate* element. An *immaculate* element is the one which does not falls into the category of any of the Obsolete, Retestable, PartiallyRetestable, or New, as shown in Equation 9.91.

$$\exists\, e \in T_M \; e \notin O, R, P, A. \tag{9.91}$$

Thus, Equation 9.92 presents the case in which tc is Reusable. According to the Equation 9.92, the case Reusable is applicable when tc is immaculate and no other impact reports exists for tc.

$$tc = immaculate \;\wedge\; \neg(\exists\, r \in IR \; for \; tc). \tag{9.92}$$

9.2.2 Classification of UTP Test Behavior Elements

As discussed earlier, the behavior of the tests in UTP can be expressed using UML activity diagram specifying test scenarios and test cases. In our test generation approach presented in Chapter 5, the activity diagrams representing the test cases are generated from BPMN collaboration diagrams and consequently express the test cases to test a process. In the following, we first present the classification definitions for the TestCase definition inside the *test architecture* and then discuss the classification of their corresponding activity diagram test cases.

Classifying a TestCase– A TestCase is defined inside the class diagram representing the *test architecture* of a business process. It is defined as an Operation

inside the TestContext class as also discussed in Chapter 5. A TestCase can be classified similar to an Operation, since it is a test operation. It would be affected by other relevant aspects, such as the data used by it and its behavior defined by a behavioral activity diagram. Hence, to classify a TestCase, we first define it and its relevant aspects.

$$Let\ tcase \in T_M \mid type(tcase) = \texttt{TestCase}. \tag{9.93}$$

$$Let\ tc \in T_M \mid type(tc) = \texttt{TestContext}. \tag{9.94}$$

$$Let\ ac \in T_M \mid type(ac) = \texttt{Activity}. \tag{9.95}$$

$$Let\ D_{tcase} = (d_1, d_2, d_3 ... d_n), where \bigwedge_{k=0} n\ type(d_k) = \texttt{DependencyRelation}. \tag{9.96}$$

$$\exists\ d1 \in D_{tcase} \mid type(d1) = Equivalence, target(d1) = ac. \tag{9.97}$$

$$\exists\ d1 \in D_{tcase} \mid type(d1) = Contained_By, target(d1) = tc. \tag{9.98}$$

$$Let\ PAR_{tcase} = (p_1, p_2, p_3 ... p_m), where \bigwedge_{r=0} m\ type(p_r) = \texttt{Parameter}. \tag{9.99}$$

$$\bigwedge_{i=0} m\ (\bigwedge_{j=0} n(\exists\ d_j \in D_{tcase} \mid type(d_j) = Containment,\ target(d_j) = p_i)). \tag{9.100}$$

Case Obsolete: A TestCase *tcase* is considered as Obsolete if the following holds to true.

$$\exists\ r \in IR \mid changeType(r) = \texttt{Delete TestCase} \wedge source(r) = tcase \tag{9.101}$$

$$\vee$$

$$status(ac) = \texttt{Obsolete}. \tag{9.102}$$

A TestCase *tcase* will be considered Obsolete is an ImpactReport for its deletion exists or if the *test behavior* corresponding to tc is Obsolete. Similar to a MockOperation, addition and deletion of parameters would not make a TestCase obsolete rather it would make it Retestable, as discussed later as well.

Case Retestable: A TestCase is considered Retestable in the following cases.

- If its corresponding Activity is Re-testable.
- If a Constraint on a MockOperation changes.
- If the type of any Parameter of the MockOperation is changed.
- If any parameter is added or deleted from the MockOperation.
- If the Verdict related to the MockOperation changes.
- If any input data element changes.

.

Case Reusable: A TestCase is considered Reusable if it remains unaffected and there are no corresponding impact reports for it as defined in the following, and if it

does not falls to any other classifications.

$$\neg(\exists\, r \in IR \mid source(r) = tcase) \tag{9.103}$$

$$\vee$$

$$\neg(tcase = (Obsolete \ \wedge \ Retestable)). \tag{9.104}$$

Case New: A `TestCase` is considered `New` if the change operation *Add TestCase* exists in the set of Impact Reports for a particular `TestContext` in the set of impact reports.

$$\exists\, r \in IR \mid changeType(r) = \texttt{Add TestCase} \wedge source(r) = tc. \tag{9.105}$$

Classifying a Activity Diagram Test Case– A UML activity diagram corresponding to a `TestCase` represents the *test behavior* and is expressed as t in Equation 9.106. In the following, we discuss various classification scenarios for t.

$$Let\ t \in T_M \mid type(t) = \texttt{Activity} \ \ \text{«}\texttt{TestCase}\ \text{»}. \tag{9.106}$$

Case Obsolete: A test case t will be considered `Obsolete` if either it is explicitly deleted from the model or the path of the process tested by t is no longer valid. This is a case when any `SequenceFlow` in the process is deleted triggering the deletion of the corresponding `ControlFlow` in t. The following equations define these scenarios.

$$\exists\, r1 \in IR \mid changeType(r1) = \texttt{Delete TestCaseActivity} \text{ for } t. \tag{9.107}$$

$$\vee$$

$$\exists\, c \in T_M \mid type(c) = ControlFlow,\ \exists\, d \in D_t \mid type(d) = Containment \wedge target(d) = c. \tag{9.108}$$

$$\exists\, r2 \in IR \mid changeType(r2) = \texttt{Delete ControlFlow} \text{ for } c. \tag{9.109}$$

According to Equation (9.107), if an impact report with a change type *Delete TestCaseActivity* exists for t, then t will be considered `Obsolete`. The second case is when a `ControlFlow` inside t is deleted making it `Obsolete` and is represented by the Equation 9.108 and Equation 9.109.

According to Equation 9.108, a `ControlFlow` c exists and is contained by t. Equation 9.108 further clarifies the conditions by stating that an `ImpactReport` with a change type *Delete ControlFlow* exists for the `ControlFlow` c contained by t.

Case Retestable:

$$\exists\ \in T_M a \mid type(a) = Action\ \wedge\ \exists\, d1 \in D_t \mid source(d1) = t\ \wedge\ target(d1) = a \\ \wedge\ type(d1) = Containment. \tag{9.110}$$

$$\exists\, m \mid type(m) = \texttt{MockOperation}\ \wedge\ \exists\, d2 \in D_t \mid source(d2) = a \wedge \\ target(d2) = m\ \wedge\ m \in (O \vee R). \tag{9.111}$$

$$\vee$$

$$\exists\, tc1 \mid type(tc1) = \texttt{TestComponent}\ \wedge\ tc1 \in (O \vee R). \tag{9.112}$$

$$\exists\, d3 \in D_t\ \wedge\ source(d3) = t\ \wedge\ target(d3) = tc1\ \wedge\ type(d3) = requires. \tag{9.113}$$

$$\vee$$

$$\exists\, do \in T_M \mid type(do) = \texttt{DataObject}\ \wedge\ do \in (O, R, P, N). \tag{9.114}$$

$$\exists\, d4 \in D_t \mid source(d4) = t\ \wedge\ target(d4) = do, type(d4) = Containment. \tag{9.115}$$

Case New: Finally, t will be considered as `New` if a *Add TestCaseActivity* change type exists in any impact report as presented in Equation 9.116.

$$\exists\, r \in IR \mid changeType(r) = \texttt{Add TestCaseActivity} \text{ for } t. \tag{9.116}$$

Case Reusable: The test case t is reusable if t is immaculate, or if t is immaculate and a rename change operation is applied on it as expressed by Equation 9.117 and Equation 9.118. If a change operation *Rename TestCaseActivity* is applied on t, then it requires the maintenance of the test model but actually does not affects the test logic.

The definition of an immaculate element is already presented in the Equation 9.91 in Section 9.2.1 while discussing the reusability of a `TestComponent`.

$$t = immaculate \forall sub \in tsub = immaculate. \tag{9.117}$$

$$\vee$$

$$t = immaculate\ \wedge\ \exists\, r1 \in IR \mid changeType(r1) = \texttt{Rename TestCaseActivity} \text{ for } t. \tag{9.118}$$

Results of Test Classification for HandleTourPlanningProcess–

The *TestCase1* represented as an activity diagram is classified as `Retestable`, as it is required to be retested due to a change in its called `Operation`. Other test cases remain unaffected because they do not call this operation. The `HTPPTCom TestComponent` and the `TestContext` class will be classified as `PartiallyRetestable`. All other elements of the *test view* will be considered as `Reusable`.

9.3 Chapter Summary

This chapter demonstrates the test classification mechanism employed by our approach for the classification of *baseline test suite*. The chapter first elaborates the classification scheme we used to classify the test elements in to `Obsolete`, `Reusable`, `Retestable`, `PartiallyRetestable`, and `New`. We further analyzed the test elements of UTP for

the conditions in which various test classification scenarios are applicable to various UTP elements. We also presented the classification definitions for these elements.

These classification definitions are then refined and mapped to the test classifications rules we developed to assist the implementation and execution. The *test classification rules* are generic, customizable, extensible, and can be defined to classify various elements of a *baseline test suite*. In contrast to the state of the art, the rules can be extended to support new classification scenarios and test languages without requiring modifications in source code.

10 Automation and Tool Support

To reduce the manual effort, automation of various activities of our approach is required . Thus, we provide tool support to enable our approach in two different ways. Firstly, by presenting a prototype tool *VIATRA Test Generation Tool* (VTG) to enable the baseline test generation.

Secondly, by using and extending EMFTrace[1] , a tool based on Eclipse Modeling Framework (EMF)[2] that provides traceability support between various software artifacts. As mentioned earlier, the goal of providing tool support is to reduce the manual effort required to commence various activities of our approach. Thus, the tool support shall enable the generation of test models from system models, preserve and record dependency relations between models and tests, support impact analysis, classify tests using our rule-based approach.

Requirements to enable Tool Support– The major requirements for the tool support are stated in the following.

1. Enable model to model transformations to realize mapping rules for the generation of UTP test models from BPMN and UML models.
2. Preserve traceability links between test models and system models during the test generation.
3. Provide support for rule-based dependency detection between UML and BPMN models and UTP test models.
4. Enable impact analysis across BPMN, UML, and UTP models using dependency relations by providing support for the execution of impact rules.
5. Support import and export of dependency relations between models and tests to reuse them for impact analysis.
6. Support the test classification of UTP models by enabling execution of test classification rules.
7. Generate reports to enable the analysis of classified elements of test models.

[1] https://sourceforge.net/projects/emftrace/
[2] http://www.eclipse.org/modeling/emf/

8. Provide import and export facility for BPMN, UML, and UTP models.

To enable the first two requirements, we developed a prototype test generation tool *VIATRA Test Generation Tool* (VTG). VTG enables the implementation of mapping rules for test generation as model transformations using the *Visual Automated Model Transformations* (VIATRA) framework [VIA11]. The tool also preserves the traceability links between system and test models during the model transformations. Further details of VTG are presented in Section 10.1.

To fulfill the other requirements, we use and extend EMFTrace. EMFTrace was originally developed to support traceability between various software artifacts [BLR11]. EMFTrace uses a model repository EMFStore[1] and provides various facilities, such as import/export, editing, and update of models. Models in EMFTrace are conformed to *XML Meta-data Interchange* (XMI) format and adapters for various modeling tools, such as Visual Paradigm [2] and Eclipse UML2 Tools [3], are available in EMFTrace. EMFTrace suits very well in our context, as it provides the facility to detect dependency relations using the rule-based approach discussed earlier in Section 6.3.2 of Chapter 6. Similarly, it already provides import facility for UML models and provides export facility of any model inside the model repository.

In one of our works, we extended EMFTrace to enable impact analysis across heterogeneous software artifacts [LFR13a]. In this thesis, we use EMFTrace to record dependency relations and perform impact analysis between UTP, UML, and BPMN models. Further, we extend EMFTrace to support the test classification by developing the test classification rules. The details of how we use and extend EMFTrace are presented in Section 10.2.

10.1 Tool Support for our Baseline Test Generation Approach using VTG

As mentioned earlier, the relevant mappings for the test generation in our approach are implemented as model transformations. The details of our test generation approach and required mappings are already presented in Chapter 5. Figure 10.1 depicts the architecture of our prototype tool VTG.

As depicted in Figure 10.1, the tool takes BPMN and UML models as input and produces UTP test models by applying a set of model transformations. These model transformations are based on the mapping rules presented in Chapter 5 and Appendix B. The model transformations also preserve a set of trace links to link the source and target model elements of the transformations, as discussed in Section 6.3.1 of Chapter 6.

In the VIATRA framework, all models are saved in a *Model Space*, which is an internal model repository. Models are expressed in VIATRA Textual Meta-modeling language (VTML) that expresses any model in the form of entities and relations. For example, a

[1] http://eclipse.org/emfstore/
[2] http://www.visual-paradigm.com
[3] http://eclipse.org/modeling/mdt/?project=uml2tools

UML `Class` is expressed as an `Entity` and an `Association` between two classes is expressed as a `Relation` in VTML. To transform a source model into a target model, transformation rules are required, which can be developed in *VIATRA Textual Command Language* (VTCL).

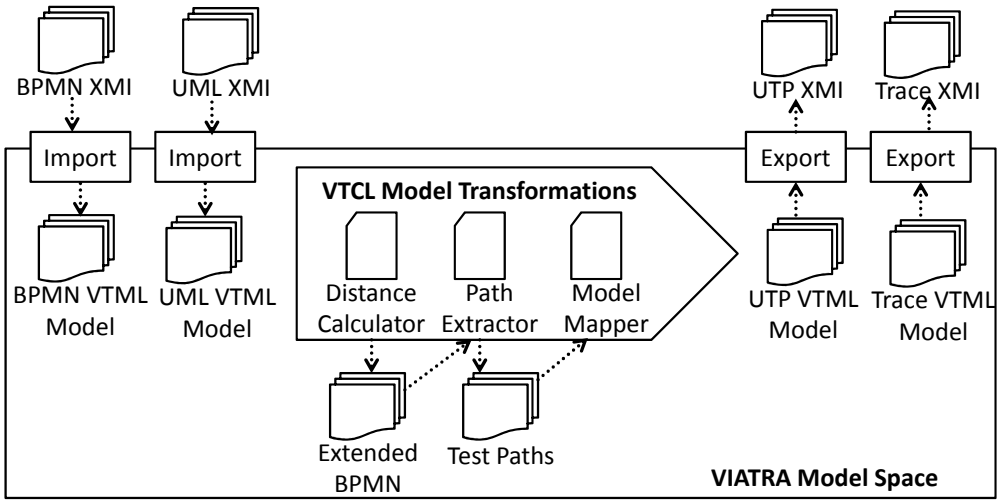


Figure 10.1: Architecture of the Baseline Test Generation Tool-VTG.

We developed a set of VTCL model transformations to realize our baseline test generation approach presented in Chapter 5. In our baseline test generation approach, most of the *test architecture* elements have one to one correspondences between system models and UTP test models. Thus, a model mapping transformation is sufficient to generate the *test architecture* elements. For example, a `TestContext` in UTP corresponds to the `ProcessClass` in UML class diagram.

However, the generation of *test behavior* requires the extraction of test paths from BPMN collaboration diagrams. These test paths are then required to be transformed to activity diagram test cases in UTP. Thus, the model transformations are also required to be developed accordingly. We developed three different transformations, which complement our test generation approach presented in Chapter 5.

Distance Calculator Transformation– The *Distance Calculator Transformation* calculates the distance of each BPMN element from the end element and attaches it to the elements in the form of textual annotations. This extended BPMN model is then used for path extraction by the *Path Extractor* transformation. Listing 10.1 depicts an excerpt of the *Distance Calculator* in VTCL.

Listing 10.1: An excerpt of DistanceCalculator.vtcl.

```
1 rule visitChildren( in Node) = seq //Visit Children of 'Node' {
2   forall Child with find unvisitedChild(Node,Child) do //take all unvisited child
        nodes
3   let Dist_N = undef, Dist_C = undef, New_Dist_C = undef in //declaration of
        variables
4   seq{
5    choose N_Distance with find GetDistance(Node,N_Distance) do
6    seq{
7     update Dist_N = toInteger(value(N_Distance)); //Get Distance of Node}
8    choose C_Distance with find GetDistance(Child,C_Distance) do
```

```
9    seq{
10    update Dist_C = toInteger(value(C_Distance)); //Get Distance of Child}
11      update New_Dist_C = Dist_N +1; //Calculate new distance of child
12   if (Dist_C > New_Dist_C) //if you encounter that new distance is shorter than old
         one:
13   seq{
14    call setDistance(Child,New_Dist_C); //set new distance
15    call visitChildren(Child); //visit children of child node (recursive)}
16  }
17 }
```

The Listing 10.1 presents a recursive rule, which visits its children nodes and assign them a value based on the hierarchical distance from the parent node. The keyword *seq* is similar to a do-while loop in standard programming languages and the keyword *call* denotes calls to other rules. Thus, the distance of a node and all its children nodes is obtained. Each child node has a distance of parent node+1. If the newly calculated distance is shorter then the old one, it is assigned to the node and the process is recursively repeated for all child nodes.

Path Calculator Transformation– The *Path Calculator Transformation* then uses the distance-based path calculation strategy discussed in Chapter 5 and extracts test paths from the extended BPMN collaboration diagram. It uses the backtracking form the end node and selects the paths having lesser distance from the end node, thus calculates the shortest path to test first. All the other paths are also calculated in the similar fashion to provide coverage of all the decision nodes of a process.

Model Mapper Transformation– Once the test paths are available, the *Model Mapper Transformation* maps the source BPMN and UML elements to UTP test models. The *Model Mapper* automates the mapping rules presented in Section Appendix B and generates the UTP activity diagram test cases from the extracted test paths. To realize the second requirement for tool support stated in the start of this chapter, the *Model Mapper* also preserves the corresponding *source* and *target* elements and saves them in a *Trace Model*. *Trace Model* stores the *source* element, the *target* element, and the *type* of the dependency relation.

To demonstrate how elements in source BPMN and UML models are mapped to the corresponding UTP elements, we present an example VTCL rule, which implements the mapping rule in Listing B.7 in Appendix B. The rule maps the element `Participant` in BPMN to the element `ActivityPartition` in UTP activity diagram test case.

The (a) part Figure 10.2 presents the mapping rule depicted Listing B.7 in Appendix B and (b) part depicts an excerpt of the corresponding transformation in VTCL. VTCL consists of a set of graph transformation rules *GTRules*, which consist of a precondition pattern, a postcondition pattern, and a set of *actions*. The *GTRule* `matchParticipant` in part (b) of Figure 10.2 consist of a precondition pattern, which checks if the element to be transformed is an element of BPMN meta-model. The postcondition pattern `matchSwimlane` ensures that after the transformation a `Participant` and corresponding `ActivityPartition` exists and a link between both is also created.

In the *actions* part of the rule, name of the `Participant` is assigned to the name of the `ActivityPartition` and the newly created `ActivityPartition` and `TraceLink` is saved/moved to the models. Thus, VTG enables our baseline test generation ap-

```
Mapping Rule 008: Participant -> ActivityPartition
Preconditions:
∃tp ∈ TP | type(tp)=TestPath
∃act ∈ AC | type(act)=Activity:UML «TestCase »
∃d1 ∈ D |source(d1)=tp, target(d1)=act, type(d1)=Derivation
∃par ∈ tp | type(par)=Participant:BPMN
Postconditions:
create ap | type(ap)=ActivityPartition:UML
ap.name=par.name
act.add(ap)
create Tracelink(ap,"Derivation", par)
create Tracelink(act,"Containment", ap)
```

(a) Mapping Rule

```
gtrule matchParticipant(out Participant)={
precondition pattern isParticipant(Participant)={
 bpmn.metamodel.bpmn.Participant(Participant);
}
postcondition pattern
matchSwimlane(Partition,Participant,Link)={
bpmn.Participant(Participant);
uml.ActivityPartition(Partition);
trace.Link(Link);
trace.Link.Source(Src,Link,Participant);
trace.Link.Target(Trg,Link,Partition);
}
action{
rename(Partition,name(Participant));
move(Partition,uml.models);
rename(Link,name(Pool)+"->"+name(Partition));
move(Link,trace.models);
}}
```

(b) Transformation Rule

Figure 10.2: The Mapping Rule and Corresponding Transformation Rule for Participant.

proach. The generated test models and trace links are later used by EMFTrace to enable our regression testing approach.

10.2 Tool Support for our Regression Testing Approach by using EMFTrace

As discussed earlier, we use EMFTrace to detect dependency relations among models and tests using the rule-based dependency detection approach implemented in EMFTrace. In one of our works, we extended EMFTrace to support rule-based impact analysis across heterogeneous various software artifacts based on the concepts presented in Chapter 8 [LFR13a]. In this thesis, we developed rules to support impact analysis across BPMN, UML, and UTP models using the impact analysis capabilities of EMFTrace. Further, we extended EMFTrace to support test classification rules for regression testing.

Architecture of Extended EMFTrace Tool– Figure 10.3 represents the architecture of the extended EMFTrace tool and the logical components of EMFTrace. The shaded elements in 10.3 are the ones, which are reused without introducing any changes to them. The other elements are either newly introduced or extended to support regression testing in particular. The EMFStore repository is the central storage of all the models and traceability links between them. EMFTrace consists of a generic rule processing engine, depicted as `RuleProcessor` in Figure 10.3, which enables execution of logical conditions and some other comparison functions to compare various properties of model elements. The `RuleProcessor` component is used for the processing of *dependency detection rules*, *impact rules*, and *test classification rules*.

Processing of the impact analysis rules is responsibility of the `ImpactAnalyzer` component. We extended EMFTrace to enable the processing of *test classification rules*, which are being implemented by the `TestClassifier` component. The `Result-`

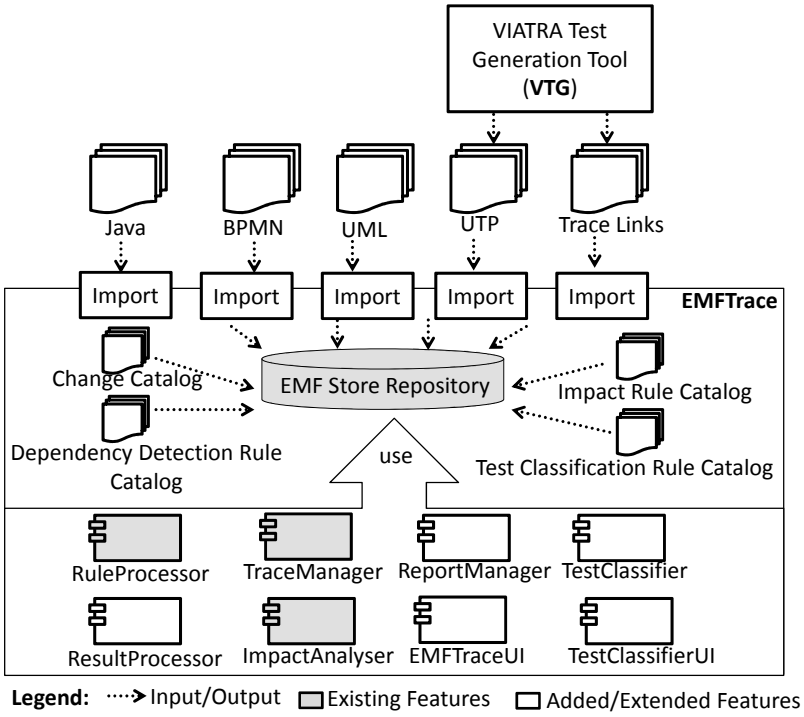


Figure 10.3: Architecture of the Extended EMFTrace Tool.

`Processor` and `ReportManager` components are responsible of processing results of these rules and creation of *impact reports* and *test classification reports*. As the name suggests, the `EMFTraceUI` and `TestClassifierUI` components provide the user interface to enable various user actions. Before we discuss how EMFTrace is used and extended for dependency detection, impact analysis, and test classification, we first present the models and meta-models required to enable our approach in EMFTrace.

Required Models and Meta-Models– Figure 10.4 depicts the models and meta-models required for implementing our approach in EMFTrace. These models consist of three categories, which are *system models*, *test models*, and *EMFTrace internal models*. The category *system models* consists of BPMN and UML models and these models should be consistent with the BPMN and UML meta-models. We use the UML and BPMN meta-models from the project *Model Development Tools* (MDT[1]).

The category *test models* consists of UTP models. UTP used class diagrams to represent the *test architecture* and *test data*. The *test behavior* is represented using activity diagrams to model test cases with UTP specific stereotypes. The UTP test models represent the test baseline which can be generated using the VTG Tool, as depicted in Figure 10.3. We implemented the UTP meta-model as a profile for UML using EMF Framework, which is available as a Eclipse plugin. Finally, the third category *EMFTrace internal models* consists of the models to express changes, dependency relations, impact rules, test classification rules, impact reports, and test classification reports. These models are used for the implementation of impact analysis and test classification rules.

[1]http://eclipse.org/modeling/mdt/?project=uml2

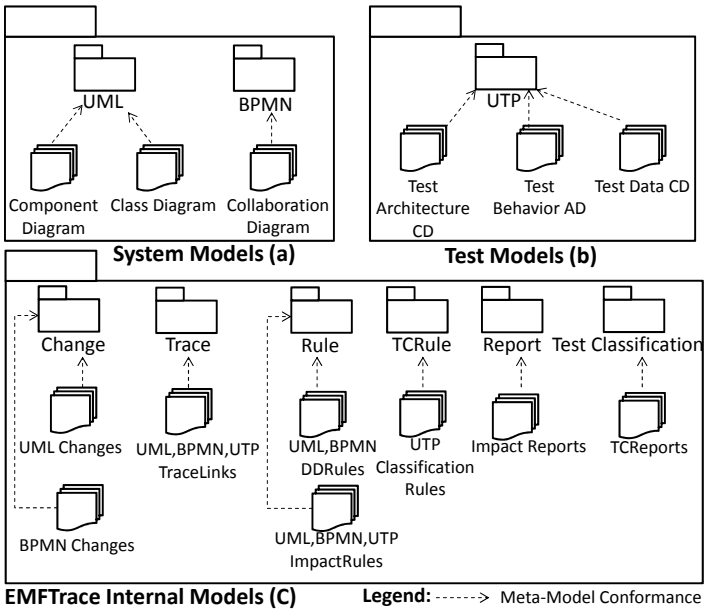


Figure 10.4: Required Models and Meta-Models for the Implementation of Approach.

10.2.1 Using EMFTrace for Dependency Detection and Rule-based Impact Analysis

To implement the dependency detection rules to support our approach, we use existing features of EMFTrace for rule-based dependency detection. The logical component `TraceManager` is responsible to record dependencies between various artifacts based on rules. Thus, the dependency detection rules for UML, BPMN, and UTP models are depicted as *Dependency Detection Rule Catalog* in Figure 10.3. EMFTrace uses the `RuleProcessor` component to to execute the dependency detection rules.

The implementation of rule-based impact analysis is part of one of our works [LFR13a] to support impact analysis across heterogeneous software artifacts. Thus, the `Impact-Analyzer` component uses the same `RuleProcessor` to process the impact rules by evaluating various conditions to identify the elements impacted by the change. A built-in function *modelRelatedTo* is used to asses if the dependency relations exists between two model elements are not. The details of the impact analysis approach are already presented in Chapter 8. The `ImpactAnalyzer` component generates the *impact reports* using the `ReportManager` component.

To support the impact analysis in our approach, in this thesis, we implemented impact rules to react on changes in BPMN and UML models to find their impact on UTP test models. Thus, the *Change Catalog* depicted in Figure 10.3 defines the change types applicable to UML, BPMN, and UTP models. The `Change Catalog` consists of the changes defined earlier in Chapter 7.

The *Impact Rule Catalog* depicted in Figure 10.3 consists of the impact rules applicable to these changes in UML, BPMN, and UTP models. These rules use the dependency relations recorded using dependency detection rules and VTG tool during the test generation and propagate the impact of these changes to other models and tests.

These rules are presented Appendix E. The component `ImpactAnalyzer` processes the impact rules and creates *impact reports* using the `ReportManager` component.

10.2.2 Extending EMFTrace for Test Classification

As discussed earlier, the `TestClassifier` component is responsible for the implementation of test classification rules. Thus, a set of test classification rules are developed in EMFTrace and can be executed on the impact reports produced during the impact analysis to decide how impacted test elements can be classified.

The `TestClassifier` uses the existing `RuleProcessor` component of EMFTrace to process the rules and to evaluate the specified conditions. However, it extends the `ResultProcessor` component to implement the actions specific to the test classification. The `ReportManager` component is also extended to enable the generation of *Test Classification Reports* based on the results produced by the `ResultProcessor`.

Figure 10.5 presents the high level interactions of various classes for the realization of test classification rules. According to Figure 10.5, the `TestClassifierUI` obtains an instance of `TestClassifier` class from the `Activator` class. The `Activator` is used for dynamic instantiation of various classes to support loose coupling between the user interface classes and core logic. After that, the method *classifyTests()* of the

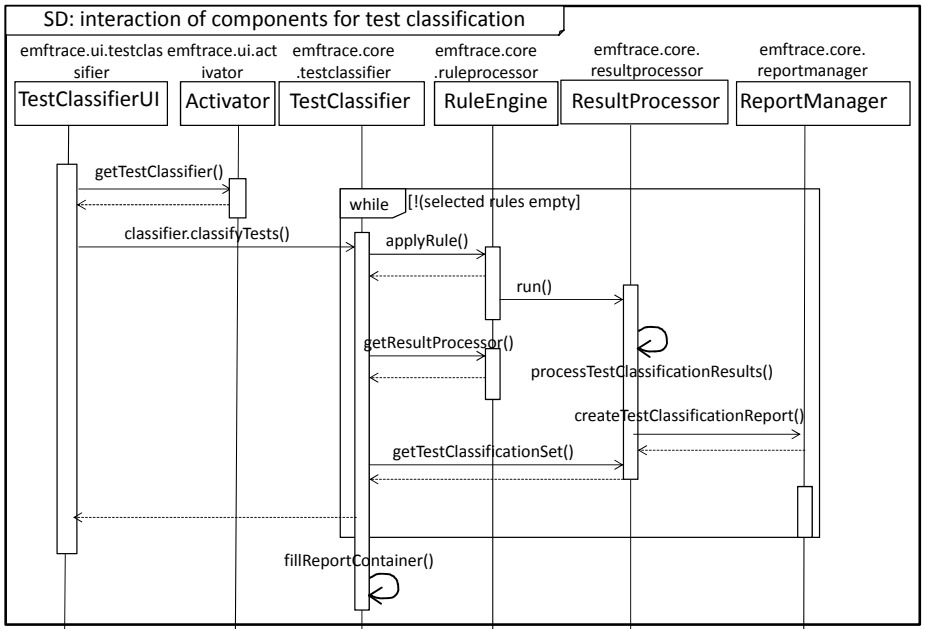


Figure 10.5: A Sequence Diagram Depicting High Level Interactions of Classes to Implement Test Classification Rules.

class `TestClassifier` is called. This method implements the core test classification logic. It takes the set of test classification rules and calls *applyRule()* method of the class `RuleEngine` of EMFTrace. The *applyRule()* method evaluates the conditions of the rules and all the elements, which satisfy these conditions are obtained.

The `RuleEngine` class also uses the various other components, such as an `Element-Processor` component to get the elements specified in the rules, a `ConditionPro-`

cessor to evaluate conditions in the rule, and a RuleValidator to check if the rules are structurally correct. However, for the sake of simplification, these are not shown in Figure 10.5.

Consequently, the *applyRule()* operation calls the *processTestClassificationResult()* operation of the ResultProcessor to process the elements obtained after the evaluation of conditions. The ResultProcessor then extracts the relevant information from a list of results and calls the ReportManager to create the *Test Classification Reports*. Listing 10.2 presents an excerpt of the source code of the method *processTestClassificationResult()*.

Listing 10.2: An Excerpt of the Source Code for Processing Test Classification Results.

```
1  public void processTestClassificationResult(Project project, Rule rule, List<List<
       EObject>> results, int[][] tuples, int index){
2    classificationSet.clear();
3    int srcIdx = ListHelper.getIndexForElement(rule, rule.getActions().get(index).
         getSourceElement());
4      int dstIdx = ListHelper.getIndexForElement(rule, rule.getActions().get(index).
           getTargetElement());
5      List<Integer> irIdxes=getIRIndex(results, srcIdx, dstIdx);
6      if( results.get(srcIdx).isEmpty() || results.get(dstIdx).isEmpty()) return;
7    TestClassificationAction currentAction=(TestClassificationAction)rule.getActions().
         get(index);
8    TestClassificationType classificationType=TestClassificationType.get(
         currentAction.getClassificationType().getValue());
9    for(int j = 0; j < tuples.length; j++){
10     if( tuples[j][srcIdx] == -1 || tuples[j][dstIdx] == -1 )   continue;
11     EObject src = results.get(srcIdx).get(tuples[j][srcIdx]);
12     EObject dst = results.get(dstIdx).get(tuples[j][dstIdx]);
13     List<ImpactReport> iReports=new ArrayList();;
14          if(irIdxes!=null){
15       for(int k=0; k<irIdxes.size(); k++){
16              ImpactReport temp= (ImpactReport)results.get(irIdxes.get(k)).get(
                    tuples[j][irIdxes.get(k)]);
17                   if(temp!=null)iReports.add(temp); }
18          }
19          EObject impactedTestElement=null;
20          if( src == null || dst == null || src == dst ) continue;
21          impactedTestElement=src;
22          ClassifiedTestElement cte=createClassifiedTestElement(iReports,
                impactedTestElement,dst,classificationType);
23          classificationSet.add(reportManager.createTestClassificationReport(cte),
                rule) ; }
24 }
```

According to the method in Listing 10.2, the method *processTestClassificationResult()* takes a *project*, a *rule* that is being processed, a list of *results*, and a two dimensional array of *tuples*. The *results* and *tuples* are produced by the ConditionProcessor, while processing the rules. The *tuples* is an array, which only contains indexes of matching elements, whereas, the actual elements are saved in the list *results*.

The method first gets the index of source and target elements of the rule (line 4 and 5). It also takes the index of the places in the *tuples*, where the matched *impact reports* processed by the rule are stored (line 6). How the specified elements are classified is obtained by accessing the ClassificationType specified by the rules (line 8-9). Then all the tuples are visited to obtain the values for the indexes specified earlier to get a matching *source element*, *target element*, and a set of processed *impact reports*. The

source element `src` is that test element, which is to be classified by the current rule.

These elements are then passed to the `ReportManager`to create the corresponding *test classification reports* (line 24-25). The test classification meta-model depicted in Fig-

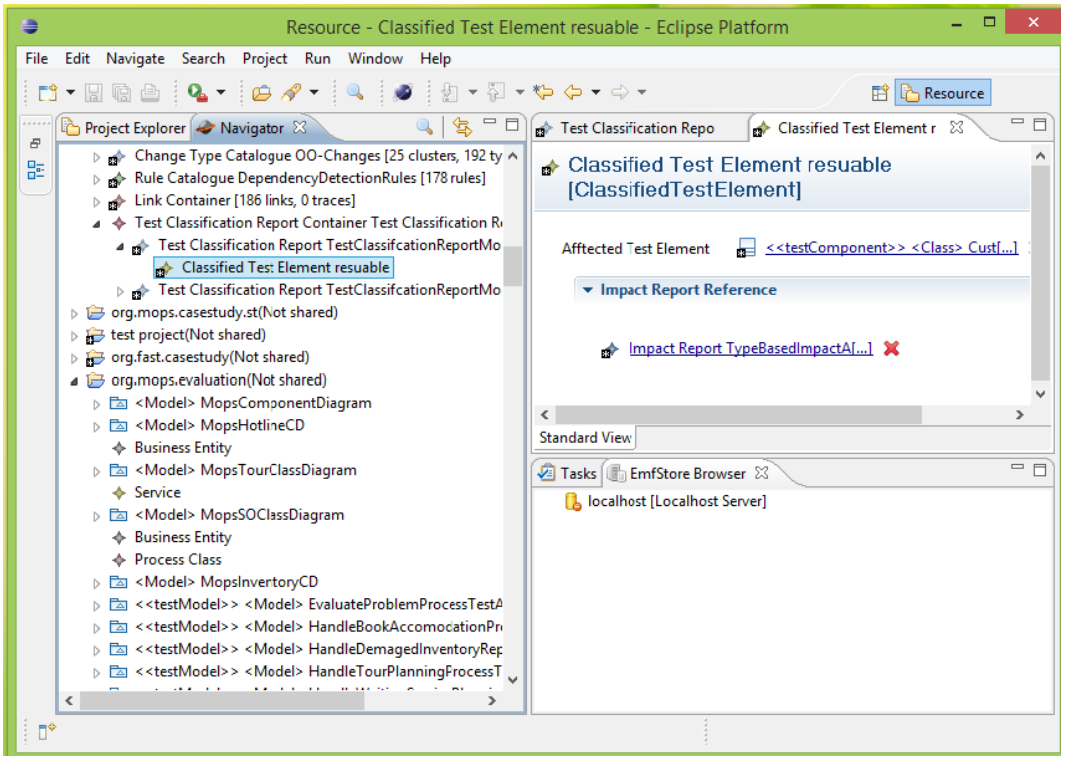


Figure 10.6: Test Classification Report in EMFTrace for a Reusable Test Element.

ure 9.2 of Chapter 9 already described the structure of the test classification reports. A test classification reports is contained in a `ClassificationReportContainer` and refers to a set of impact reports, the classified test element, and the assigned classification type. Figure 10.6 shows an impact report, which shows a *reusable* `TestComponent`.

10.3 Chapter Summary

This chapter presents the details of automation and tool support we provide to automate various activities of our approach. We implemented our baseline test generation approach in a tool TVG using a a model transformation framework VIATRA. TVG also produces a set of dependency relations between BPMN, UML, and UTP models.

To support rule-based dependency detection, impact analysis, and test classification, we use and extend EMFTrace. EMFTrace is a tool built in Eclipse using EMF and EMFStore frameworks. A set of rules to detect dependency relations, perform impact analysis, and classify tests for BPMN, UML, and UTP are developed in EMFTrace. We present the architecture of extended EMFTrace and discuss how we reuse and extended its various components to support the test classification. Thus, we provide a generic facility to develop and extend test classification rules for various test models in EMFTrace.

11 Evaluation

This chapter presents the evaluation of our approach to assess its performance. The main objective of the evaluation is to prove our early hypothesis that *using the explicit dependency relations to analyze the changes propagating to tests can yield better results during regression testing*. Therefore, we apply our approach on a case study from a joint academic and industrial project. We design the experimental evaluation by following the guidelines for conducting software experiments by Juristo *et al*. [JM10]. Hence, we first develop an evaluation protocol to establish the basis of the experimental evaluation. The evaluation protocol defines various requirements, parameters, and metrics, which are necessary to ensure an effective experimental evaluation. The protocol is then applied to our case study and the results are analyzed by using the specified metrics.

11.1 The Evaluation Protocol

For the evaluation, we first present our initial hypothesis and extract various evaluation requirements from the initial hypothesis. Stating an initial hypothesis helps to focus on the goals of the evaluation [JM10].

Initial Hypothesis– Our initial hypothesis is stated as follows.

1. HP1: The *precision* and *recall* of our approach is better than the name similarity-based regression testing approaches.
2. HP2: Our approach provides more *coverage* of modified test elements, and reduces the test suite depending on the number of applied changes and the type of a change.

The above presented hypothesis features two important aspects crucial to the evaluation of our approach. Firstly, the *correctness* of the results, which can be established by analyzing the *precision* and *recall* of the approach. Since most of the existing model-based regression testing approaches are name similarity-based, we want to demonstrate that the *precision* and *recall* of our approach is better than those approaches.

Secondly, the *completeness* of the approach, which can be assessed by analyzing its capability to cover the tests during regression testing as well as how much the base line test suite is reduced for regression testing. These factors are covered by the evaluation requirements stated below. The experimental evaluation and the analysis of evaluation results are required to be consistent with these evaluation requirements.

Evaluation Requirements– We extract the following evaluation requirements form the initial hypothesis.

1. Measuring Precision: What is the extent to which the unaffected test elements are omitted by the approach?
2. Measuring Recall: What is the extent to which the affected test elements are selected by the approach? *Recall* is also stated as *inclusiveness* by Rothermel *et al*. [RH94b]
3. Measuring Coverage: How much coverage of modified test elements is achieved?
4. Measuring Reduction: To which extent the number of required regression test cases is reduced?

To answer the questions corresponding to the evaluation requirements, corresponding metrics are required to be developed to satisfy the evaluation requirements [OSH04]. Therefore, the experimental evaluation aims to analyzes the validity of the initial hypothesis by considering the above stated evaluation requirements using various evaluation metrics. However, before we present the evaluation metrics, we first present *invariable parameters* for the evaluation.

Invariable Parameters– The invariable parameters are required to understand the nature and scope of the experiment and they remain fixed throughout the experimental evaluations [JM10].We present the invariable parameters of our experimental evaluation in the following.

Experimental Object and its Scope– The experimental object for our evaluation is the *Field Service Technician* case study, introduced earlier in Chapter 2. The case study is taken from a joint academic and industrial project MOPS. The aim of the case study is to automate the business processes on mobile platforms to facilitate the mobility and various tasks of a field service technician during the field work. The scope of our case study is limited to the business processes for mobile platforms.

The selection of the case study is influenced by various factors. The foremost aspect is the availability of a set of domain models, process models, and other relevant information directly from the industrial partners. This is particularly important for the application of our approach in the industrial context, as it is applied to real world scenarios developed by the domain experts.

Size of the Case Study– The case study is of medium size and consists of various process models, system models, test models, and other required artifacts. The number of these models and artifacts is depicted in Table 11.1. The upper part of the table depicts the system models and the required project artifacts. The lower part of the table depicts the required test models and artifacts. Thus, the test models consist of 75 test behavior models, 229 test components, and 184 test stubs and mocks. Moreover, test code in Java is also available for some parts of the test models. Parts of the case

Table 11.1: Statistics about size of the case study.

System Models and Project Artifacts			
BPMN Processes	**UML Components**	**UML Classes**	**EMFTrace Project Size**
18	49	181	42892 Lines
Test Models and Artifacts			
UTP Test Behavior Models	**UTP Test Components**	**UTP Mocks & Stubs**	**Java Test Code**
75	229	184	Test Classes:107 Test Methods:485

study evolved continuously by the author as well as various students, who worked on this project during their masters thesis. Therefore, various scenarios to support the needs of evolution are identified from the case study.

Tools for Experiment Execution– We use EMFTrace for the execution of our case study. All the required changes and rules to perform impact analysis, record dependency relations, and classify tests are also implemented in EMFTrace. The set of rules consists of 95 dependency detection rules, 70 impact analysis rules, and 35 test classification rules for BPMN, UML, and UTP models. These rules are presented in Appendix E as well. All the models, tests, and other artifacts are stored in a project in the EMFTrace model repository. The size of the EMFTrace project is also presented in Table 11.1.

Benchmarks for Result Evaluation– Another important aspect is how we compare the results of the evaluation to draw meaningful conclusions from the experimental evaluation. For this purpose, we take two distinct measures. Firstly, we developed a manual *oracle* to define the expected outcomes of our evaluation. The *oracle* is developed by analyzing various change scenarios for potentially affected test elements manually. The results of our case study are then compared with the oracle to analyze the correctness of the results.

Secondly, we compare the results of our approach with a random test selection approach based on the concept of name similarity matches. A similar random name similarity-based approach (RNS) was also used by [YJH08] to analyze their test reduction approach. Moreover, most of the existing regression testing approaches also use name similarity matches to find the elements affected from changes. Therefore, comparison with such an approach gives us an idea about how the existing approaches would perform in comparison to our approach.

11.1.1 Evaluation Metrics

The evaluation metrics presented in this section adhere to the evaluation requirements presented earlier.

Precision and Recall– These are the standard information retrieval concepts and are used by several approaches for the evaluation [RH94b, HH04]. To define both *precision* and *recall*, we first define the set S and R. The set S represents the set of test elements, which is selected by a given regression testing approach, when a change is applied. The set R defines the set of test elements which is required to be selected by a regression testing approach. The set I denotes the intersection of both S and R, that is, $S \cap R$. The regression testing approach, which is under consideration is denoted by

A. The *precision* of an approach A is denoted as P_A and *recall* of A is denoted as R_A. Based on these definitions, we define the *precision* and *recall* as follows:

$$P_A = \frac{I = S \cap R}{S} \quad R_A = \frac{I = S \cap R}{R} \tag{11.1}$$

Coverage– Coverage can be defined as the ability of a given regression testing approach to cover various test elements during the test classification. A regression testing approach is required to cover various aspects of the *test view*, namely the *test architecture*, *test behavior*, and *test data*. We define the *coverage* of test elements as a relation induced by a regression testing approach A. A similar approach to define *coverage* is also used by Harrold *et al*. [HRRW01] to determine the *coverage* of program statements by tests.

We define a *coverage relation* as $T \times C$ with $covers_A = (t, c)$ true if and only if the test element t is covered by a test classification report c. The *coverage relation* $covers_A(t, c)$ can be represented as a matrix C^A whose rows represents the elements of the test suite T and columns represents the elements of a the test classification set C. An element $c_{i,j}$ of C^A can be defined as:

$$c_{i,j} = \begin{cases} 0 & \text{if } covers^A(i,j) \\ 1 & \text{otherwise} \end{cases} \tag{11.2}$$

The *cumulative coverage* of the test elements can be defined as follows.

$$CC = \sum_{i=1}^{|T|} \sum_{j=1}^{|C|} co_{i,j} \tag{11.3}$$

Reduction– It measures the extent to which the original test suite is reduced after the application of a given regression testing approach. A reduced set of the test cases is denoted by $T^{'}$, which are the test cases affected by changes.

To measure the *reduction*, we consider only behavioral test elements, that is, *test cases*, *mock operations*, and *SUT operations*. The reason is that the effort for developing these behavioral test elements is higher due to their complexity, as they implement the core logic of various test scenarios.

Thus, $M \subseteq T$ consists of the set of *test cases*, *mock operations*, and *SUT operations* from T. $M^{'} \subseteq T^{'}$ denotes the subset of $T^{'}$ which consists of all the classified *test cases*, mock operations, and *SUT operations*. The percentage *reduction* denoted by $Reduce_A$ for a regression testing approach can be measured by using the following formula.

$$Reduce_A = 100 - (\frac{M^{'}}{M} \times 100) \tag{11.4}$$

Although, the above mentioned evaluation metrics answer the evaluation requirements presented earlier, we present two more metrics, which are proposed by Rothermal *et al*. for the evaluation of regression testing approaches.

Efficiency and Generality– Efficiency is defined in terms of time and space requirements of the approach, its computability, the cost of calculating modifications, and the costs that occur during the critical phases of testing. Generality is defined as the ability to work in a wide range of practical applications and identifiable classes of programs [RH96]. These both metrics depends on many subjective variables, such as the degree of applicability and the expected cost of modifications. Therefore, we do not explicitly calculate them during our experimental evaluation. However, a discussion on various aspects influencing the generality and efficiency of our approach is presented later in this chapter.

After defining the required evaluation metrics, we now specify how the experimental evaluation is commenced and how the required data is gathered.

11.1.2 The Experiment Execution Process

Following steps are taken to execute the experiment on our case study.

–Set-up and Preliminary Work

1. Identify changes and their order of application to commence each change scenario.
2. Prepare the test oracle for the comparison of results for each change.
3. Import all artifacts, changes, and rules in EMFTrace.
4. Import dependency relations from our tool VTG.

–Experiment Execution and Data Collection

1. Apply the dependency detection rules to record all the required dependencies between models and tests.
2. Select a change scenario, apply changes listed in the change scenario one by one, and record the impact reports for each change.
3. Analyze the impact reports to add any required elements demanded by the impact reports and apply further changes.
4. Once all the changes in a scenario are consumed, start the execution the test classification rules on the impact reports.
5. Record the test classification reports for each scenario and repeat the change scenarios

–Result Analysis and Metric Computation

1. Compare the impact reports of each change with the oracle to first analyze the correctly identified effected test elements.
2. Compare the test classification reports for the scenario and record the correct and false results.
3. Analyze the recorded results by calculating the evaluation metrics presented earlier.

Executing RNS Approach– To compare our results with the RNS approach, we implemented a prototype of the approach in EMFTrace. The approach compares the name of each model element on which a change is applied to the process test cases. The approach then selects the test cases randomly to reduce the selected subset of matched

tests. The test cases are classified according to the applied change type. If the initial change is *delete* or *add*, then all the selected tests are classified as *obsolete* or *new*. Otherwise, all the selected tests are classified as *Retestable*. The results of each change are separately recorded for each change scenario.

11.2 Evaluation Results

This section presents the results of our evaluation on the change scenarios from the field service technician case study. We present the results to analyze the correlation of the results obtained by various metrics according to our initial evaluation requirements. Thus, we first present the results of two different change scenarios. After that, we present the cumulative result of all the scenarios to present the big picture.

11.2.1 Evaluation Results of Change Scenario 1

The change scenario 1 (CS1) was discussed throughout this thesis to elaborate various concepts presented in the thesis. The scenario consists of 4 atomic and one composite change operation. Figure 11.1 presents the results of various metrics in the result of CS 1 of our approach as well as the RNS approach. The blue curve depicts the results of

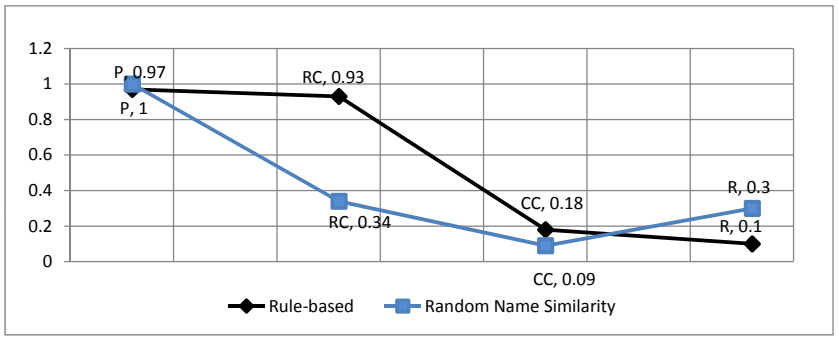


Figure 11.1: Precision, Recall, Coverage,and Reduction achieved on CS1.

RNS approach, whereas, the black curve depicts the results of our rule-based regression testing approach. The results are mapped on the scale form 0 to 1. The results presented in Figure 11.1 show that the *precision* of both approaches in this scenario is quite good. This means that all affected test cases are correctly identified and classified. The RNS approach was able to detect all the affected tests, whereas, our approach identified and correctly classified about 0.97 percent test elements. However, a huge difference is in the *recall* of both approaches.

The results of our approach are very good yielding 0.93 percent *recall*, whereas, the *recall* of RNS approach is as low as 0.35. The results of the scenario CS1 support our claim that dependency types help to propagate correct results is a valid explanation of this phenomena. Similarly, the *coverage* of the test elements provided by our approach are twice higher in comparison to the RNS approach. The reason is that the dependency relations in our approach cover a large number of dependency relations between different system and test models. However, the RNS approach reduces the original test suite to 0.1, which is quite significant reduction. Our approach reduces the test baseline to 0.3 on a scale of one, which is still quite reasonable. However, the low *recall* of

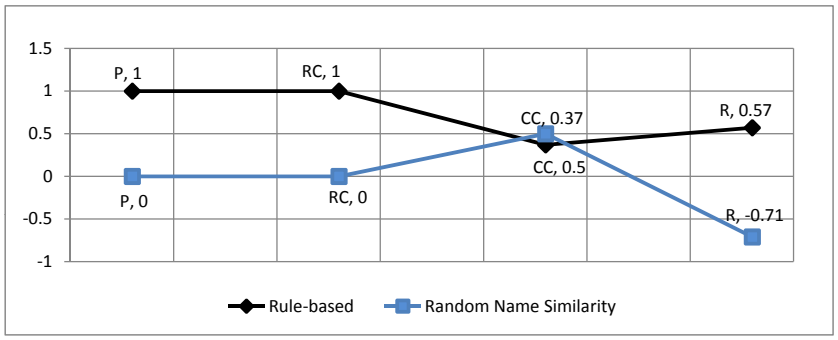


Figure 11.2: Precision, Recall, Coverage,and Reduction achieved on CS2.

the RNS approach raises questions on the validity of the reduced test subset. A very similar tendency can also be seen in the results of change scenario 2 (CS2), depicted in Figure 11.2.

11.2.2 Cumulative Evaluation Results

In the following we present the cumulative results of 10 different change scenarios, which are applied during the evaluation.

Results for Precision and Recall– As discussed earlier *precision* and *recall* are used to measure the correctness of the approaches. Therefore, *recall* measures the extent to which the affected tests are included in the regression test suite. However, *precision* determines the presence of false positives, that is, the inclusion of those test elements not defined in the oracle. The *precision* and *recall* of our approach largely depends on its ability to record the dependency relations among models and test correctly. To ensure this, we cover 114 different types of dependency relations, which are recorded using two different approaches. Due to this, the risk of missing any modified test elements is precisely very low. However, all approach do not covers all artifacts required during several stages of software development, which can result in additional side effects, consequently effecting the *precision* and *recall*.

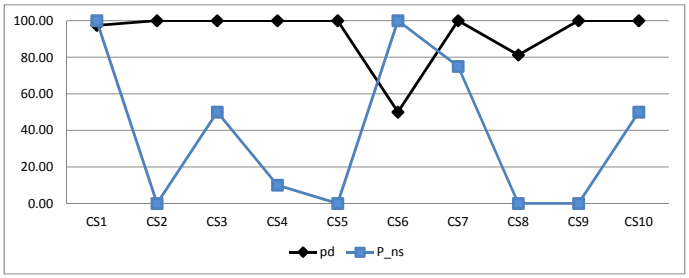


Figure 11.3: Cumulative Results for the Precision of Approaches.

Figure 11.3 shows the cumulative results showing the *precision* of our dependency-based approach and the RNS approach. The results of the parameter *precision*, in our approach, are almost linear in the most high ranges, which means that it performed consistently well in most of the scenarios. Thus, the average *precision* of our approach based on these 10 scenarios is recorded as 92.87%. The *precision* of the RNS approach

shows an inconsistent trend, often falling to the lowest ranges. The RNS approach mostly performed well in the scenarios, where direct relations among the changed elements and test cases was present. Dependency relations of different types are not integral to this approach, hence it performed very low in the cases where inter-model dependencies among structural and process models were affecting the tests indirectly. Thus, the results yield an average *precision* of 38.50% of the RNS approach. The scenario

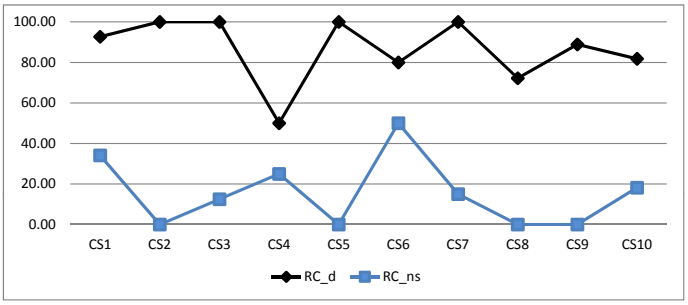


Figure 11.4: Cumulative Results for the Recall of Approaches.

six depicted as C6 shows an interesting result, where our approach performed not very well and the name similarity-based approach was able to select and classify the tests. This shows a particular case, in which no test elements were affected and required to be classified due to the nature of change. A conceptual error in one of the rules, however, wrongly classified a set of test cases. The RNS approach, however, was not able to find any impact, which was caused by the inter-model dependency relation. Therefore, it was luckily able to produce the desired results in this scenario. The results of the *recall* of our approach and RNS approach are displayed in Figure 11.4. The average *recall* obtained by our approach as shown by our results is 86.56%.

However, the results obtained by the RNS approach consistently fall in the lower range, due to either omitting the required results, or wrongly classifying the tests due to missing information about dependency relations. Thus, the average *recall* obtained by the RNS approach in these scenarios is fairly low yielding 15.48% *recall*. The concrete results in the form of a table are also presented in Table 11.2. Moreover, all the test classification reports obtained for these change scenarios are also presented in the Appendix G.

Table 11.2: Variables and Results for Change Scenarios.

R	S_d	$I_d = R \cap S_d$	P_d=$\frac{I_d}{S_d}$	RC_d=$\frac{I_d}{R}$	S_{ns}	$I_{ns} = R \cap S_{ns}$	P_{ns}=$\frac{I_{ns}}{S_{ns}}$	RC_{ns}=$\frac{I_{ns}}{R}$	CC_d	CC_{ns}	PR_d	PR_d
Change Scenario 1												
41	39	38	97.44	92.68	14	100	100	100	18.29	8.54	10	30
Change Scenario 2												
27	27	27	100	100	5	0	0	0	37.70	8.20	57.14	-72
Change Scenario 3												
8	8	8	00	100	2	1	50	12	11.90	2.38	75	75
Change Scenario 4												
4	2	2	100	50	10	1	10	25	2.44	12.20	0	90
Change Scenario 5												
2	2	2	100	100	0	0	0	0	4.76	0	100	100
Change Scenario 6												
10	16	8	50	80	5	5	100	50	22.22	13.89	0	38
Change Scenario 7												
20	20	20	100	100	4	3	75	15	20	6.67	0	57
Change Scenario 8												
18	16	13	81.25	100	0	0	0	0	42.11	0	0	100
Change Scenario 9												
27	24	24	100	80	0	0	0	0	18.95	0	27.78	100
Change Scenario 10												
22	18	18	100	100	8	4	50	18.18	30	10	100	42

These results explain the low *recall* of the RNS approach. The selected and classified test elements by the RNS approach are depicted by the variable S_{ns} and the correctly selected and classified test elements are presented by the variable I_{ns}. The values on S_{ns} and I_{ns} reflect that for a number of change scenarios either no test elements were obtained or the set of obtained test elements is wrongly classified. As explained earlier, this is due to missing the iter-model relations among the models of *structural* and *process view*, which have no direct relations to tests. The basis of our test classification results is the rule-based impact analysis performed across the system models and tests models. In one of our studies, the same impact analysis approach showed a *precision* and *recall* of 80% for the rule-based impact analysis of heterogeneous software artifacts including UML and Java [LFR13b]. Thus, the results of *precision* and *recall* obtained by our experimental evaluations show the same trends and are consistent to our previous findings.

Results for Coverage and Reduction– As defined earlier, the *coverage* of an approach is defined as the extent to which the test elements relevant to various test aspects are covered during regression testing. However, *reduction* is the extent to which the baseline test suite is reduced after the selection and classification of the affected tests. However, both *coverage* and *reduction* cannot be analyzed in the isolation and the results should always be interpreted in relation to the *precision* and *recall*.

High *coverage* is not necessarily a good thing if the *recall* of the approach is low. It shows that those test elements are also covered by the approach, which are not required. The low *coverage* and low *recall* also indicate that it might be the result of not covering adequate test elements, which is indicative of a problem.

The same concept is applicable while computing the parameter *reduction* as well. Although high *reduction* is very desirable during regression testing, as we want to minimize the test effort as much as possible. However, similar to the *coverage*, high *reduction* with low *recall* suggests that the approach is not including a number of potentially affected test cases in the regression test suite. This might result in undetected side effects

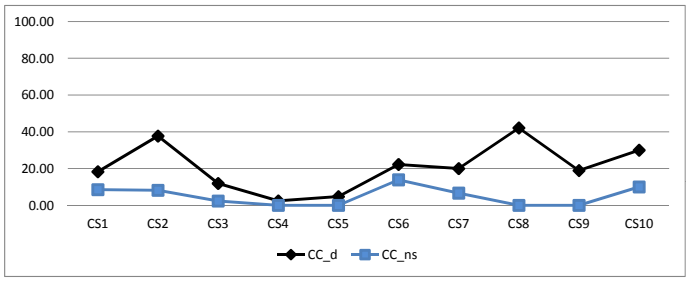


Figure 11.5: Evaluation Results for the Coverage.

of the changes. The low *reduction* with low *recall* might also indicate that the approach is including those tests in the regression test suite, which could possibly be omitted.

However, the *coverage* is computed differently from the *reduction*. It specifies all the potentially affected test elements selected after analyzing changes, regardless of how they are classified for the regression testing later. Figure 11.5 shows the *coverage* achieved by both approach for various change scenarios. The *coverage* achieved by our approach is consistently higher than the RNS approach The average *coverage* achieved by our approach is 20.84% and the average *coverage* of the RNS approach is 4.97%. The reason is that our approach covers various test aspects, such as the *test architecture*, *test behavior*, and *test data*. Whereas, the RNS approach only consider the test cases reflecting the *test behavior*, similar to the other state of the art regression testing approaches.

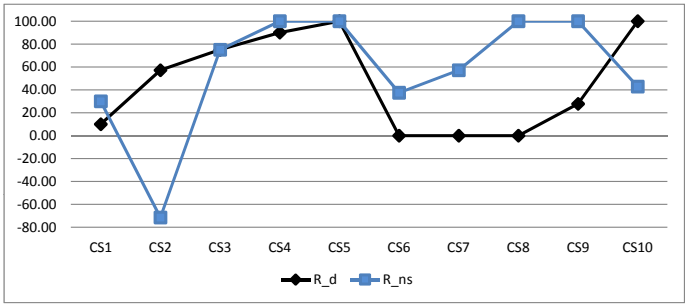


Figure 11.6: Evaluation Results for the Reduction.

Figure 11.6 depicts the results for the *reduction* achieved by the approaches. According to the figure, the results of our approach are comparable to the RNS approach. However, the *reduction* capability of the RNS approach is slightly higher than our approach. However, as stated earlier, a comparison of the *reduction* with the achieved *recall* is necessary to interpret the achieved *reduction*. A comparison with the *recall* values yields that our approach still performs better as the *recall* is higher than the RNS approach when the *reduction* is low. However, the average *reduction* achieved by our approach is still slightly higher than the RNS approach due to a negative value depicted in Figure 11.6. The average *reduction* achieved by our approach is 45.99% and the average *reduction* achieved by the RNS approach is 57.11%. The negative value of RNS is due to adding the new test cases, which were not required by the test oracle.

The evaluation results presented above are promising and imply that our approach can yield good performance on the systems comparable to the one used in the evaluation.

In the following, we briefly analyze various factors which can influence the efficiency and generality of our approach. Moreover, we also evaluate our approach based on the same evaluation criteria using which we evaluated the existing state of the art approaches in Chapter 3 as presented in Table 11.3 This provides a comparison of our approach with the existing state of the art approaches.

Results for Efficiency and Generality– of our approach is subject to various factors. For the generality, our approach first develop the abstract conceptual foundations of all the concepts it integrates and then apply them to address the domain specific requirements. Thus, the concepts are applicable to the general software systems. Although, concrete artifacts are developed for the domain of business processes only.

However, our approach can be generalized to the other domain easily, as it is based on the extensible rules to detect dependencies, perform impact analysis, and classify tests. The rules require no modification to the tool to adapt to a new modeling or test specification language. Similarly, new change types and dependency relations can also be integrated easily without requiring any additional tool support.

The efficiency our approach is mainly influenced by the tool support provided by EMFTrace, presented in Chapter 10. The worst case time complexity of the rule execution inside EMFTrace is $O(n^2)$, which is solvable in polynomial time, where n is the number of models in the repository. The best case and worst case space complexity is always $O(n)$. Moreover, the approach can save significant test costs compared to the approaches that require more effort in the later critical testing phases, since it is able to forecast the number of test cases affected by a change in the earlier phases of software development.

Table 11.3: Overall Analysis Based on Evaluation Criteria for MBRT Approaches.

Inquiries	**Result of Own Approach**
Inq.1: The approach is suitable of which types of the systems?	Generic and applied to business processes.
Inq.2: What is the testing level addressed by the approach?	Process *unit* and *integration* tests
Inq.3: The approach covers which of the views?	*structural View*, *behavioral view*, *process view*
Inq.4: Which aspects of the test views are covered by the approach?	*test architecture*, *test behavior*, *test data*
Inq.5: What are the input models used by the approach?	UML class and component diagrams, BPMN collaboration diagram, UTP test models
Inq.6: Does the approach require any additional inputs other than models?	*Trace links* recorded during test generation
Inq.7: How are the tests expressed?	UTP test models
Inq.8: What is the base line test suite generation method?	Model-driven test generation
Inq.9: Does the approach discusses the change detection mechanism?	No changes directly selected and applied
Inq.10: Whether the approach provides sound change definitions for modifications in the system?	Yes, Change taxonomy and its application
Inq.11: How many change types are considered by the approach?	3 atomic and 5 composite change types.
Inq.12: what type of dependency relations are supported?	92 concrete dependency relations, 50 types
Inq.13: How the dependency relations are recorded?	during test generation as well as using dependency detection approach
Inq.14: Does the approach considers dependency types?	yes, 50 different types
Inq.15: How dependency-relations are stored?	in a separate model
Inq.16: Does the approach provide logic of test case selection and classification?	yes, test classification rules
Inq.17: Were the ideas defined by some algorithmic details or not?	yes
Inq.18: Does the approach provide tool support or not?	VTG and EMFTrace
Inq.19: What is the implementation platform for the tool if implemented?	VIATRA, Java, Eclipse EMF
Inq.20: Is the input of approach compliant to any standards?	OMG standards: BPMN and UML
Inq.21: Is the approach compliant to any test specifications standard?	OMG Standard: UTP

Inq.22: Is the approach evaluated on any case study or does any experimental evaluation was present?	Field Service Technician Case study, medium size
Inq.23: Does the approach rely on a specific change identification method?	No, changes selected from predefined catalog
Inq.24: Is it easy to extend the impact analysis logic?	Impact rules are extensible without source code modification
Inq.25: Dependency relations recording and impact analysis is tightly coupled or not?	Dependency recording prior to impact analysis.

11.3 Threats to Validity

A fundamental question regarding the evaluations is that how valid are the results? Therefore, an analysis of the validity threats is required to check whether the results of the evaluation are valid and in accordance to the initial evaluation requirements and hypothesis [WRH+00]. Wohlin *et al*. classify the validity threats in four different categories. We discuss them in the context of our experimental evaluation in the following.

Internal Validity– The internal validity describes the validity within a given environment by analyzing the degree of influence through side effects, which might negatively effect the experiment results [WRH+00]. To minimize these side effects, we used a single controlled environment, predefined tasks, and same artifacts during our evaluation. Further, we have not changed any hardware or software environment to further restrict the external side effects. However, following factors may affect the internal validity of our evaluation.

Firstly, the *precision* of dependency relations obtained during the dependency recording phase can effect the validity of selected test cases. In case of missing or wrongly recorded dependency relations, the selected and classified tests might also be invalid. To minimize this side effect, we used two different means of recording the dependency relations, that is, during the test generation and using a rule-based dependency detection approach. Moreover, we developed a manual oracle to analyze the unwanted results to further minimize this threat.

The second aspect is the side effect introduced by the selected modeling method. In practice, various modeling methods can be used for the specification and design of the system and each modeling method would might require relationships among different model elements. This side effect can not be treated in the current settings as it is not possible to consider all the available modeling methods in practice due to the effort required. However, to minimize this side effect, during the development of the case study, the guidelines from two well accepted modeling methods are followed. These are the methods of Penker *et al*. [PE00]and SoaML business modeling approach [SDE+10].

External Validity– The factors which obstruct the generalization of the results of the evaluation are widely known as threats to the external validity [WRH+00]. We already discussed the *generality* of our approach earlier in the previous section. We do not claim that the results of the conducted evaluations can be generalized for all other domains. However, considering the heterogeneity of the used models and artifacts, it can be safely assumed that for the larger systems involving complex interactions, the similar experiments will probably yield equivalent results in the terms of *precision*, *recall*, *coverage*, and *reduction* achieved. Moreover, our approach is more adaptable due

to extensible rules, thus can be generalized to other domains as well.

Construct validity– The construct validity is concerned with the relation between the theory and collected observations [WRH+00]. In theory, we have shown the relation between the dependency relations among models and tests, which propagate the changes to models of different views and in the result affect the test cases. Our theory also aims to reduce the existing test suite to a smaller subset affected by the changes, and to provide better *coverage* of various views and test aspects.

During our evaluation, we examine the *precision* and *recall* of the selected tests, applying our initial theory. For that, we used various change types and dependency relations to observe the change preparation through them. Similarly, we use another measure *reduction* to check how much the tests are reduced. Finally, another metric *coverage* measures how many test aspects are covered by an approach. Therefore, our observations during the experiment are also consistent with our theory.

Conclusion validity– Conclusion validity is concerned with the reliability of conclusions drawn form the experimental evaluation [WRH+00]. conclusion validity can depends upon several factors, for example, the sample size, the heterogeneity of subjects, and the quality of measures. A general threat to the conclusion validity is the *fishing and the error rate* threat, where the experiment is conducted to get a specific outcome [WRH+00]. To reduce the influence on the outcome, we developed and applied a set of heterogeneous change scenarios considering various maintenance activities.

Similarly, the sample size in our case consists of a reasonably large set of number of UML, BMMN, and BPMN model. The quality of the models can also be considered satisfactory as they were taken from a joint academic and industrial project and were created by professionals and experts. Another potential threat that can affect the conclusion validity is the quality of measures. Quantitative measures are often considered better than the qualitative measures when performing evaluations [WRH+00]. We used a set of both quantitative and qualitative metrics to and hence the quality of our measures can also be considered as adequate.

11.4 Chapter Summary

The chapter presents the details of experiment and evaluation we performed to assess various characteristics of our approach. To do so, we analyzed our approach using a number of quantitative and qualitative metrics. The evaluation is performed on the *Field Service Technician* case study from a joint industrial and academic project. The objective of the case study was to automate the business processes on mobile platforms. The quantitative metrics we evaluated during our experiment are *precision*, *recall*, *coverage*, and *reduction* achieved by our approach.

Thus, over approach provides an average *reduction* of test cases by 46% achieved with an average *precision* and *recall* of 93% and 87% respectively. These results are fairly good as compared to a random name similarity based approach, which yield 57% *reduction* with an average *precision* and *recall* of 39% and 15% respectively. Our approach also provides an average *coverage* of test elements up to 21%, which is significantly better than the name similarity-based approach, which provides 5% coverage when applied on the same change scenarios. The qualitative metrics we subjectively evaluated are

efficiency and *generality* of our approach.

As shown by our evaluations, in some of the cases, the *reduction* achieved by our approach is not very significant. The reason is that our approach aims to select all those test elements, which are potentially affected by the changes and do not use any other test reduction technique. Thus, all potentially affected tests are included in the regression test suite. However, after the classification of test cases by applying our approach, a further prioritization can be made based on the other metrics, such as cost of the test cases, fault detection effectiveness, and test cases associated with higher risk. Since the existing test prioritization approaches can be adapted for this purpose, we consider this aspect out of scope for our current work.

12

Conclusion and Future Work

We conclude this thesis by synthesizing our contributions, examining the implications of our work, and exploring the potential future research directions. Section 12.1 provides a summary of the core contributions by revisiting our accomplishments. Section 12.2 presents a critical review of various important issues and Section 12.3 presents the potential future research directions.

12.1 Summary of Contributions

The foremost contribution of this thesis is to provide a holistic model-based regression testing approach, which is based on the notion of dependency relations and complex change types. A subset of test cases corresponding to the changes is selected for regression testing by examining the potential changes and dependency relations earlier in the software development life cycle. It helps to *forecast the required test effort earlier*, prior to the implementation of changes. Thus, an *early test planning and resource allocation* is possible, which is believed to reduce the overall testing costs. The approach synthesizes various methods and concepts to assist the model-based regression testing process.

Proposed Model-based Regression Testing Approach– Our model-based regression testing approach is based on the notion that different types of dependency relations exist between models representing several views of a software system and tests. These dependency relations can be used to propagate the changes across models and tests to identify the potentially impacted test cases. Therefore, our approach employed a systematic course of activities to enable the selection of a subset of test cases for regression testing using these dependency relations. A *holistic coverage* of various software views is provided by supporting the *structural view*, *process view*, and *test view*.

Firstly, the test baseline is generated from a set of models by developing a model-driven test generation approach. After that the dependency relations between the models and the test baseline are recorded. The recording of these dependency relation is accomplished during the test generation and by using a rule-based dependency detection approach.

For a consistent and unified representation of changes in models, we proposed a change taxonomy, which defines various *atomic* and *composite* changes applicable to models. These changes and dependency relations are then used to identify the potentially affected test cases by using a rule-based impact analysis approach. The rule-based impact analysis approach recursively propagates the changes through the dependency relations recorded earlier and identifies the potentially affected test cases.

To decide whether the affected tests are obsolete, required for retest during regression testing, or new, we proposed a rule-based test classification method. The test classification rules operate on the affected test elements and analyze the type of affected test elements, any related affected elements, and the type of initially applied change to decide how a test element should be classified.

The major benefits of our approach are discussed in the following. As discussed earlier, our approach provides an early forecast of the required test effort, thus enables the early test planning and resource allocation. It is *less computational extensive*, as the dependency relations are not repeatedly searched after every change. The approach provides *better coverage* by considering the *structural view* and *process view* as well as various aspects of the *test view*, as discussed in the next section. Moreover, a comprehensive set of dependency relations and change types is also covered to provide a better assessment of the potentially affected test cases.

We designed our approach to enable *generic and flexible solutions* to answer the research question RQ6, presented in Chapter 1. Thus, our approach is based on a set of *extensible rules* to record dependency relations, perform impact analysis, and classify tests. Moreover, all the solutions presented in this thesis are first discussed on an abstract level and are then adapted for the domain of business processes. This enables us to discuss the generic solutions first and then demonstrate their implementation on a particular application domain. Furthermore, all the above mentioned rules can be created and executed using our tool EMFTrace without requiring any modification in the source code of EMFTrace. This also enhances the flexibility of our approach. In the following, we present further related contributions of this thesis.

Model-driven Baseline Test Generation– One of the related contributions of our work is to provide an approach for the generation of the test baseline. The research question RQ4, presented in Chapter 1, demands for a test baseline covering different test aspects. To accomplish this, we developed a model-driven test generation approach, which takes BPMN and UML models as input to generate UTP test models using model to test transformations. We developed the required mapping rules between BPMN, UML, and UTP models and implemented these rules as model transformations for the test generation. Our test generation approach covers *test architecture*, *test behavior*, and *test data* to test the collaborative processes. The *test behavior* is represented by activity diagram test cases generated using a distance-based path traversal algorithm.

The approach is implemented in our prototype tool VTG, which uses the VIATRA framework to realize the required model transformations. Our test generation approach supports *early testing*, *uses standard* modeling and test specification languages, and provides *better coverage* of the test view. Finally, it supports *higher abstraction* by using models for the test generation as well as the test specification.

Change Taxonomy– Developing a change taxonomy to express various changes in models is also another contribution of our work. The research question RQ3, presented in Chapter 1, establishes the need for considering the simple and complex changes applicable to models during regression testing. These changes are required to realize various change scenarios. For a unified and consistent representation of changes, our approach proposes a change taxonomy, which consists of *atomic* and *composite* changes

applicable to models. Our change taxonomy is composed of eight fundamental change types which can be defined for models. Most of the existing model-based regression testing approaches are dependent on model comparison, hence can deal only with very basic changes, such as addition, deletion, and update of attributes of model elements. Our approach is based on the application of changes using a predefined change catalog, which is defined using our change taxonomy. This enables early start of the regression testing activity resulting in early estimation of the required regression testing effort, even before a change is implemented on models. We used our change taxonomy to define the changes applicable to BPMN, UML, and UTP models.

Inherent Support for Dependency Relations for Rule-based Impact Analysis– Another contribution of our work is to integrate a rule-based impact analysis approach, which uses dependency relations of various types between models and tests to answer the research question RQ2, presented in Chapter 1. To do so, first we identified various types of dependency relations among models of different views and tests and use them to perform impact analysis. As discussed earlier, these dependency relations are recorded prior to the impact analysis by using a rule-based dependency detection approach and also recorded during the baseline test generation.

To distinguish between various types of dependency relations, our approach presents a taxonomy of dependency relations. The taxonomy classifies various dependency relations based on their purpose. A comprehensive set of dependency relation is identified between BPMN, UML, and UTP models by using the taxonomy of dependency types. Once dependency relations between models and tests are available, we employed a rule-based impact analysis approach to propagate the impact of changes across models and tests. The rule-based impact analysis approach was developed initially to support impact analysis across heterogeneous software artifacts and uses change propagation through dependency relations to access the impact. We used this approach to find impact of changes in BPMN and UML models on UTP tests by using the dependency relations recorded earlier. We elicit a comprehensive set of rules to support dependency detection and impact analysis for BPMN, UML, and UTP models.

Rule-based Test Classification– The research question RQ5, presented in Chapter 1, inquires how the affected test elements identified during the impact analysis activity can be classified to distinguish them for reuse during regression testing. To answer this question, we present the concept of test classification rules. The test classification rules analyze and classify the affected test elements based on their type, any related affected element, and the type of the initially applied change. These rules are able to specify various conditions on the models, impact reports produced during the impact analysis, and test models. These conditions define the cases under which the baseline test models and test cases are classified as `Obsolete`, `Reusable`, `Retestable`, `PartiallyRetestable`, and `New`. To classify various test elements of UTP test specifications, we thoroughly analyzed them to identify different test classification conditions.

These definitions are then translated to create the test classification rules. The test classification rules are implemented in EMFTrace, a tool, which was initially developed for the dependency detection among software artifacts. It was further enhanced to support the rule-based impact analysis. We integrated the concept of test classification rules in EMFTrace as well to support our approach.

12.2 Critical Review

In the following, we discuss various critical issues regarding our approach and the validity of our evaluation results.

Reliability of Dependency Relations and Impact Analysis– The reliability of the test selection and classification in our approach depends on the reliability of dependency relations and the rule-based impact analysis approach. For a through coverage of dependency relations, we record them using two different approaches. This greatly reduces the chance of missing any potential dependency relations between models and tests. A study conducted in our research group, which apply the same rule-based dependency detection and impact analysis approach for object-oriented software artifacts, yields 90% precision. This also complements the results produced by the evaluations presented in this thesis.

Availability of Models– An assumption of our approach is the availability of a set of system models, which should be up-to-date and consistent to each other. In some projects, a complete set of models might not be available due to limited resources. In such cases, where the resource constraints do not allow extensive modeling, the critical parts of system should be modeled at minimum to get the benefits of early estimation of test effort, early testing, and early bug tracking. This helps to ensure that atleast critical system parts are well tested. Studies have also shown that model-based testing can reveal defects earlier and save project costs in the long run [BLW05]. Therefore, it is wise to invest on early modeling and regression testing to save costs in the long run.

Predefined Rules– Our approach requires a set of predefined rules and if they do not cover a specific model, the impact of changes on that model cannot be determined. However, our approach is easier to extend, as it is based on extensible rules. In comparison to other state of the art approaches, different rules required by our approach can be extended without requiring modifications in the source code of the tool. Therefore, new rules can be easily implemented to cover any additional models.

Validity of Evaluation Results– We evaluated our approach on a joint academic and industrial case study of medium size and complexity. Thus, the experimental evaluation is based on real scenarios and changes. Thus, over approach provides an average reduction of test cases by 46% achieved with an average precision and recall of 93% and 87% respectively. Our approach also provides an average *coverage* of test elements up to 21%, which is significantly better than the name similarity-based approach that achieved 5% coverage when applied on the same change scenarios.

To provide a thorough coverage of various models and tests, a comprehensive set of rules for dependency detection, impact analysis, and test classification is developed, which also greatly reduces the chances of missing impacts on test elements. During the evaluation, we compared our results against a name similarity-based approach and using a manually developed oracle. To avoid any potential bias in the evaluation, we explicitly discuss the measures we taken to reduce the validity threats, as discussed in Section 11.3 of Chapter 11.

We do not claim that the findings of our evaluation can be generalized. However, we can safely assume that similar experiments on the systems involving complex inter-

actions will likely yield similar results in terms of the *precision*, *recall*, *coverage*, and *reduction* achieved.

12.3 Future Work

We identified several potential research directions for the future work during this thesis, which are discussed in this section briefly.

Round Trip Change Support for Test Code– One of the potential future works is to provide a round trip change support, if changes are made to the test code directly. Currently, our approach can be used if tests are generated from models and are consistent to models. However, agile software development practices may allow the testers to change the test code without considering the models first. As a consequence, the design models and test models corresponding to the test code might become inconsistent. To deal with this issue, rules can be developed to propagate the impact of changes in test code to the test models and consequently to the design models. For this purpose, potential dependency relations between test models and test code are required to be identified. This round trip approach can also help to assess the *model coverage* provided by the test cases, which is another important problem in the domain of testing.

Support for Test Prioritization Metrics– During our work, we select all potentially affected tests cases. Therefore, the tests selected by our approach are not necessarily a minimal subset. In different scenarios, it might be desirable to further prioritize the selected test cases corresponding to parts of the system with high risk, high cost, or fault severity history etc. Although, test prioritization is not in the current scope of our work, in the future the concept of rule-based test selection can be extended to support these metrics as well. We already conducted an initial investigation in a master's thesis by analyzing the cost of test derived from business processes. However, this is still a preliminary work and requires further research in this area.

Support for Automatic Test Code Generation– At present, our approach focuses largely on the tests models. For the test execution, these test models are required to be translated to the concrete test code. During our evaluations, we use some test code written in Java with UTP specific annotations and developed rules for it as well. However, we translated the test models to the test code manually. To automate this aspect, the existing test code generation approaches can be analyzed and extended to support the test code generation from UTP test models for business processes.

Generalization to other Domains– Finally, to generalize our findings, there is a need to apply our approach on other domains, such as embedded systems or telecommunication systems. Our approach can be applied to other domains by extending the set of rules to support dependency detection rules, impact analysis, and test classification. The telecommunication domain can be one of the potential application domains, as model-based testing is widely used in this domain. Tests are often generated from class diagrams, sequence diagrams, and message sequence charts. These tests are usually expressed using Testing and Control Notation (TTCN). Thus, dependencies between these models and TTCN tests are required to be recorded and corresponding rules are required to be developed for the application of our approach.

A

State of the Art Analysis Tables

Table A.1: Analysis based on the criteria C1 and C2 in the category *Scope*.

Approaches	INQ 2	INQ 3	INQ 4	INQ 5	INQ 6	INQ 7	INQ 8	INQ 9	INQ 10	INQ 11
State-based										
Chen *et al.* [CPU07, CPU09]	unit	BV	TB	EFSM	SDL	transition sequence	path-based	NS	no	add, delete, modify
Korel *et al.* [KTV02]	unit	BV	TB	EFSM	SDL	transition sequence	path-based	NS	no	add, delete
Beydeda and Gruhn [BG00]	unit	BV	TD	class SM	source code	def-use pairs	graph traversal for data flow	NS	no	modify
Farooq *et al.* [FIMN07, FIMR10]	unit, integration	BV,SV	TB	SM, CD	none	XML	classification tree method	MC	yes	add, delete, property-update
Component-based										
Muccini *et al.* [MDR06, Muc07]	component	BV	NS	CHARMY SM	none	NS	NS	MC	yes	add, delete, modify
Wu and Offutt [WO03]	component	BV, SV	NS	SM	none	NS	NS	MC	no	add, delete, modify
Specification-based										
Gorthi *et al.* [GPCL08]	acceptance	RV	TB	structured AD	risk data	activity paths	NS	MC	no	add, delete, modify
Chen and Probert [CPS02, CP03]	acceptance	RV	TB	AD	risk data	activity paths	NS	MC	no	modify
Filho *et al.* [FBH+10]	functional	RV,BV, SV	TB	AD, SD, CD	risk, feature value, TD	test procedures	path-based, CPM	edit time monitoring	no	modify
UML-based and Object Oriented										
Mansour and Takkoush [MT07]	unit, integration, system	BV, SV	TB	AD	none	method call sequences	NS	MC	no	modify
Briand *et al.* [BL02, BLS03, BLY09]	system	RV, BV, SV	TB	use case, SD, CD	none	NS	NS	MC	yes	addition, deletion, modify property

Continued on next page–See legends at table end

Table A.1 Continued from previous page

Approaches	INQ 2	INQ 3	INQ 4	INQ 5	INQ 6	INQ 7	INQ 8	INQ 9	INQ 10	INQ 11
Deng *et al.* [DSW04]	system	BV, SV	TB	Use case, SD, AD, CD	none	NS	path-based	NS	no	modify
Ali *et al.* [ANMU07]	system	BV, SV	TB	SD,CD	none	control flow paths	graph traversal	MC	yes	modify
Pilskalns *et al.* [PUA06]	model testing	BV, SV	TD	SD, CD	none	sequence of conditions	non-binary domain analysis	NS	yes	add, delete, modify
Naslavsky *et al.* [NR07, NZR09, NZR10]	integration	BV, SV	TB	SD, CD	none	message call sequences, JUnit skeletons	model transformations	MC	yes	add, delete, modify
Traon *et al.* [TJJM00]	integration	SV	TB	CD	none	stub sequences	graph traversal-based	NS	no	modify
Martins and Vieira [MV05]	unit	BV	TB	AD	none	control flow edges	control flow analysis	MC	no	add, remove
Zech *et al.* [ZFKB12]	generic	generic	generic	generic	none	UTP, Telling TestStories (TTS) models	NS	MC	no	NS

LEGENDS:

TB	: Test Behavior	TA	: Test Architecture	TD	: Test Data	NS	: Not Specified
EFSM	: Extended Finite State Machine	AD	: Activity Diagram	SD	: Sequence Diagram	CD	: Class Diagram
SM	: State Machine	BV	: Behavioral View	SV	: Structural View	RV	: Requirement View
CPM	: Category Partition Method	SDL	: Specification Description Language	MC	:Model comparison		

Table A.2: Analysis based on criteria C5 and C6 in the category *Core*.

Approaches	INQ 12	INQ 13	INQ14	INQ 15	INQ 16	INQ 17	INQ 18	INQ 19
State-based								
Continued on next page–See legends at table end								

Table A.2 Continued from previous page

Approaches	INQ 12	INQ 13	INQ14	INQ 15	INQ 16	INQ 17	INQ 18	INQ 19
Chen *et al.* [CPU07, CPU09]	IM	during impact analysis	data and control	on the fly computation	affected	algorithms presented	implemented, NS	NS
Korel *et al.* [KTV02]	IM	during impact analysis	data and control	on the fly computation	affected	no algorithms presented	no	NA
Beydeda and Gruhn [BG00]	IM	during impact analysis	data	no preservation, on the fly computation	affected	algorithms presented	none	NA
Farooq *et al.* [FIMN07, FIMR10]	IRM	during impact analysis	none	no preservation, on the fly computation	obsolete, reusable, retestable, new	no algorithms presented	START prototype	Eclipse plug-ins, Java
Component-based								
Muccini *et al.* [MDR06, Muc07]	IM (CHARMY SM)	during impact analysis	none	no preservation, on the fly computation	retestable, new	algorithms presented	plug-ins in CHARMY environment	Java
Wu and Offutt [WO03]	IM	NS	data and control	NS	affected, new	no algorithms presented	none	NA
Specification-based								
Gorthi *et al.* [GPCL08]	requirements to test only	assumed between requirements to tests	none	NS	affected, new	algorithms presented	implemented, NS	NS
Chen and Probert [CPS02, CP03]	requirements to test only	assumed between requirements to tests	none	traceability matrix	affected, new, prioritized	no algorithms presented	none	NA
Filho *et al.* [FBH+10]	IM	assumed	none	traceability links	obsolete, reusable, retestable, new	no algorithms presented	TDE-UML	Java
Object Oriented and UML-based								
Mansour and Takkoush [MT07]	IRM	during impact analysis	none	on the fly computation	affected	algorithms presented	none	NA
Briand *et al.* [BL02, BLS03, BLY09]	IRM	during impact analysis (name Similarity matches)	none	on the fly computation	obsolete, reusable, retestable	no algorithms presented	RTS prototype	Java
Deng *et al.* [DSW04]	IRM	NS	none	NS	affected	no algorithms presented	none	NA
Ali *et al.* [ANMU07]	IRM	during impact analysis (EC-CFG constructed)	none	on the fly computation	obsolete, reusable, retestable	no algorithms presented	none	NA

Continued on next page–See legends at table end

Table A.2 Continued from previous page

Approaches	INQ 12	INQ 13	INQ14	INQ 15	INQ 16	INQ 17	INQ 18	INQ 19
Pilskalns *et al.* [PUA06]	model to test only	assumed between model to tests	none	NS	obsolete, reusable, retestable, new	algorithms presented	none	NA
Naslavsky *et al.* [NR07, NZR09, NZR10]	IRM	during baseline test generation	none	traceability links	obsolete, reusable, retestable	algorithms presented	MBSRT tool	Eclipse plug-ins , Java
Traon *et al.* [TJJM00]	IM	during impact analysis (Test Dependency Graph)	contractual, implementation	on the fly computation	affected	no algorithms presented	none	NA
Martins and Vieira [MV05]	model to test only	assumed between requirements to tests	none	NS	obsolete, reusable, retestable	algorithms presented	none	NA
Zech *et al.* [ZFKB12]	model to test only	assumed between models and tests	none	NS	no	tool support discussed	Model Versioning and Evolution (MoVE) platform	Java, Eclipse

LEGENDS:

IM : Intra-Model IRM : Inter-Model TD : Test Data
NS : Not Specified NA : Not Applicable

Table A.3: Analysis based on criteria C9 and C10 in the category *Applicability*.

Approaches	INQ 20	INQ 21	INQ 22	INQ 23	INQ 24	INQ 25
State-based						
Chen *et al.* [CPU07, CPU09]	SDL	none	7 different case studies. size of largest test suite is 1691	NS	CMD	TC
Korel *et al.* [KTV02]	SDL	none	small example	NS	NA	TC
Beydeda and Gruhn [BG00]	Class SM	none	small example	NS	NA	TC
Farooq *et al.* [FIMN07, FIMR10]	UML	XML	Student Enrollment System case study, 723 test cases	MC	CMD	TC
Component-based						

Continued on next page–See legends at table end

Table A.3 Continued from previous page

Approaches	INQ 20	INQ 21	INQ 22	INQ 23	INQ 24	INQ 25
Muccini *et al.* [MDR06, Muc07]	CHARMY SM	none	Cargo Router System (10 components)	MC	CMD	TC
Wu and Offutt [WO03]	UML	none	small example	NS	NA	NS
Specification-based						
Gorthi *et al.* [GPCL08]	Activity like notation	none	Retail System case study, 342 test cases	MC	CMD	assumed
Chen and Probert [CPS02, CP03]	Activity like notation	none	3 IBM Sphere components, 306 test cases	MC	NA	assumed
Filho *et al.* [FBH+10]	UML	none	none	edit time monitoring	CMD	assumed
Object Oriented and UML-based						
Mansour and Takkoush [MT07]	UML	none	multiple case studies, size of largest test cases 90	MC	NA	TC
Briand *et al.* [BL02, BLS03, BLY09]	UML	none	596 test cases	MC	CMD	TC
Deng *et al.* [DSW04]	UML	none	none	NS	NA	NS
Ali *et al.* [ANMU07]	UML	none	small example	MC	NA	TC
Pilskalns *et al.* [PUA06]	UML	none	Transcode component, 32 classes, 52 test cases	NS	NA	NS
Naslavsky *et al.* [NR07, NZR09, NZR10]	UML	JUnit	PIMs and Acqualush case studies (size NS)	MC	CMD	PRI
Traon *et al.* [TJJM00]	UML	none	SMDS server in telecommunication domain 38 classes	NS	NA	TC
Martins and Vieira [MV05]	UML	none	Common C++ library,2 classes, 24 methods, 6 versions	MC	NA	assumed
Zech *et al.* [ZFKB12]	UML, EMF	UTP	small examples	MC	through OCL queries	assumed

LEGENDS:

NS : Not Specified
NA : Not Applicable
MC : Model Comparison
SDL : Specification Description Language
SM : State Machine
CMD : Code Modification Needed
PRI : Prior to impact analysis
TC : Tightly Coupled

B

Mappings and Mapping Rules for UTP Test Generation

B.1 Mapping rules for Test Architecture Generation

Input Models:
B= set of BPMN collaboration diagrams representing the process view
C_D UML class diagram representing the structural view
C_{OD}= UML component diagram representing the structural view
D=Set of dependency relations
Output Models:
CD_{TA}=UTP test architecture class diagram
D=Set of dependency relations

Listing B.1: A Mapping Rule to Generate SUT.

```
1  Mapping-Rule 002: ProcessClass->SUT
2  Description:
3  /*A SUT is created against the ProcessClass and all the operations of the
       ProcessClass are also mapped to the SUT.*/
4  Preconditions:
5   ∃x ∈ C_D | type(x)=Class:UML «ProcessClass »
6   ∃p ∈ B | type(p)=Collaboration:BPMN
7   ∃d1 ∈ D |source(d1)=x, target(d1)=p, type(d1)=Equivalence
8   ∃tp ∈ CD_TA | type(p)=Package:UML «TestPackage »
9   ∃d2 ∈ D |source(d2)=x, target(d2)=tp, type(d2)=Derivation
10  ∃OP ∈ x = (op_1, op_2, op_3...op_n) | ∀op_i ∈ OP type(op_i)=Operation:UML (∀i = 0...n)
11 Postconditions:
12  create su ∈ CD_TA | type(su)=Class:UML «SUT »
13  su.name=x.name
14  ⋀_{i=1}^{n} [op_{i} \in n
15  create op_i^sut | type(op_i^sut)=Operation:UML
16  op_i^sut.name=op_i.name
17  mapParameters(op_i, op_i^sut)
18  su.add(op_i^sut)
19  createTraceLink(su,"Containment", op_i^sut)]
20  create dep ∈ CD_TA | type(dep)=Dependency:UML
21  source(i)=su, target(dep)=tp, type(dep)=import
22  createTraceLink(x,"Definition", su)
23  createTraceLink(tp,"Tests", su)
```

Listing B.2: A Rule for Generating a TestContext.

```
Mapping Rule 003: ProcessClass -> TestContext
Description:
/*A TestContext is created against a ProcessClass and added to the corresponding
    TestPackage.*/
Preconditions:
 ∃x ∈ C_D |type(x) =Class:UML «ProcessClass »
 ∃p ∈ B |type(p) =Collaboration:BPMN
 existsd1 ∈ D | source(d1)=x, target(d1)=p, type(d)=Equivalence
 ∃tp ∈ T_M | type(tp)=Class:UML «TestPackage »
 existsd2 ∈ D | source(d2)=x, target(d2)=tp, type(d2)=Derivation
 Postconditions:
 create tc | type(tc)=Class:UML «TestContext »
 tc.name=x.name+"TC"
 p.add(tc)
 createTraceLink(x,"Derivation", tc)
 createTraceLink(p,"Containment", tc)
```

Listing B.3: A Rule to Generate a TestComponent (EmptyParticipant).

```
Mapping Rule 004: Participant->TestComponent
Description:
/*For a Participant of a Process, which contains no Lanes, a TestComponent is created
    .
A component corresponding to the Participant is required, whose interfaces are mapped
     to the MockOperations of the new TestComponent.*/
Preconditions:
∃p ∈ B | type(p)=Collaboartion:BPMN
∃par ∈ P |type(par)=Participant:BPMN
¬(∃la ∈ par | type(la) = Lane)
∃pc ∈ C_D | type(pc)=Class:UML «ProcessClass »
∃d0 ∈ D | source(d0)=p, target(d0)=pc, type(d1)=Equivalence
∃com ∈ C_OD | type(x)=Component:UML
∃d1 ∈ D | source(d1)=par, target(d1)=com, type(d1)=Equivalence
∃tp ∈ T_M, where type(tp)=Package:UML «TestPackage »
∃d2 ∈ D | source(d2)=pc, target(d2)=tp, type(d2)=Derivation
∃IN ∈ com = (int_1, int_2, int_3...int_n) | ∀int_i ∈ IN type(int_i)=Interface:UML (∀i = 1...n)
∃tc ∈ T_M | type(tc)=Class:UML «TestContext »
∃d3 ∈ D | source(d3)=pc, target(d3)=tc, type(d3)=Derivation
Postconditions:
create tcom | type(tcom)=Class:UML «TestComponent »
tcom.name=com.name+"TestComp"
∀int_i ∈ IN
[create mo | type(mo)=Operation:UML «MockOpeartion »
mo.name=int_i.name+"Mock"
mapParameters(int_i,mo)
tcom.add(mo),
createTraceLink(tcom,"Containment", mo)
createTraceLink(mo,"Mocks",int_i)]

tp.add(tcom)
create a ∈ T_M, where type(a)=Association:UML, source(a)=tc, target(a)=tcom
createTraceLink(tp,"Containment", tcom)
createTraceLink(tc,"Usage", com)
createTraceLink(par,"Derivation", tcom)
createTraceLink(tcom,"Mocks", com)
```

Listing B.4: A Rule to Generate a TestComponent (Participant with Lanes).

```
Mapping Rule 005: Participant (with Lanes) ->TestComponent
Description:
/*Each Lane in a Participant is mapped to a TestComponent.
 All the ServiceTasks inside these Lanes are translated to MockOperations.
```

```
 The definitions of these ServiceTasks is obtained from the class diagram from the
     class realizing a component.*/
Preconditions:
∃p ∈ B |type(p)=Collaboartion:BPMN
∃par ∈ p |type(par)=Participant:BPMN
∃L = l1, l2, l3, ..., ln |type(li)=Lane:BPMN ∀i = 1...n
∀li∃SRi = sr1, sr2, sr3, ..., srm | type(srj)=ServiceTask (∀j = 1...m)
∃pc ∈ CD | type(pc)=Class:UML «ProcessClass »

∃d0 ∈ D | source(d0)=P, target(d0)=pc, type(d1)=Equivalence
∃com ∈ COD |type(com)=Component:UML
∃comClass ∈ CD | type(comClass)=Class:UML
∃d1 ∈ D | source(d1)=par, target(d1)=com, type(d1)=Equivalence
∃d2 ∈ D | source(d2)=com, target(d2)=comClass, type(d1)=Realization
∃tp ∈ TM | type(tp)=Class:UML «TestPackage »
∃d3 ∈ D | source(d3)=pc, target(d3)=tp, type(d3)=Derivation
∃tc ∈ TM |type(tc)=Class:UML «TestContext »
∃d4 ∈ D | source(d4)=pc, target(d4)=tc, type(d4)=Derivation
Postconditions:
∀li ∈ L, [
create tcom where type(tcom)=Class:UML «TestComponent »
createTraceLink(li,"Derivation", tcom)
∀srj ∈ SR
[create mo | type(mo)=Operation:UML «MockOpeartion »
mo.name=srj.name+"Mock"
get m for sr(j) /*from comClass by using trace link of type Equivalence */
mapParameters(m,mo)
tcom.add(mo),
createTraceLink(tcom,"Containment", mo)]
create TraceLink(mo,"Mocks", srj)
create a ∈ TM, where type(a)=Association:UML, source(a)=tc, target(a)=tcom
```

Listing B.5: A Rule to Generate a TestCase Definition.

```
Mapping Rule 006: Activity -> TestCase
Description:
/*An Activity representing the behavior of a TestCase is used to generate its
    definition inside the TestContext.
The TestContext class contains TestCases to test its corresponding process.*/

Preconditions:
∃tb ∈ TM | type(tb)= Activity:UML  «TestCase »
∃p ∈ SM |type(p)=Collaboartion:BPMN
∃d1 ∈ D |source(d1) =tb, target(d1)=p, type(d1)=Tests
∃tc ∈ TM |type(tc)=Class:UML «TestContext »
∃pc ∈ SM | type(pc)=Class:UML «ProcessClass »
∃d2 ∈ D |source(d2) =p, target(d2)=pc, type(d1)=Equivalence
∃d3 ∈ D |source(d3) =pc, target(d3)=tc, type(d1)=Derivation
Postconditions:
create tcase | type(tCase)=UML:Operation «TestCase:UTP »
tcase.name=tb.name
mapTestCaseParameters(tcase, tb)
tc.add(tcase)
create TraceLink(tcase,"Definition", tb)
create TraceLink(tc,"Containment", tcase)
```

B.2 Mapping Rules for Test Behavior Generation

Input Models:
TP: Set of test paths extracted from BPMN collaboration diagrams
B= set of BPMN collaboration diagrams representing the process view
C_D UML class diagram representing the structural view

C_{OD}= UML component diagram representing the structural view
CD_{TA}=UTP test architecture class diagram
Output Models:
AC=Set of Activity Diagrams representing a test case D=Set of dependency relations

Table B.1: Mappings between Collaboration and Activity diagram.

BPMN Elements	**Activity Elements**	**Mapping Rule**
TestPath (derived from BPMN)	Activity	Listing B.6
Participant	ActivityPartition	Listing B.7
Lane	ActivityPartition	Listing B.8
SequenceFlow	ControlFlow	Listing B.9
ExclusiveGateway	ConditionalNode	Listing B.10
EventBasedGateway	ConditionalNode	Listing B.11
Task	OpaqueAction	Listing B.12
ScriptTask	OpaqueAction	Listing B.12
BusinessRuleTask	OpaqueAction	Listing B.12
ManualTask	OpaqueAction	Listing B.12
UserTask	OpaqueAction	Listing B.12
Task (calling ServiceTask)	Call Operation Action	Listing B.13
SendTask	SendSignalAction	Listing B.14
SendTask (Service Calling)	CallOperationAction	Listing B.15
ReceiveTask	AcceptEventAction	Listing B.16
StartEvent	InitialNode	Listing B.17
MessageStartEvent	InitialNode followed by AcceptEventAction	Listing 5.2
SignalStartEvent	InitialNode followed by AcceptEventAction	Listing 5.2
ConditionalStartEvent	InitialNode followed by a ControlFlow with a condition	Listing B.18
EndEvent	CallOperationAction followed by ActivityFinalNode	Listing B.19
MessageEndEvent	AcceptEventAction followed by ControlFlow and CallOperationAction followed by ActivityFinalNode	Listing B.20

Listing B.6: A Rule to Generate a TestCase Activity for a TestPath.

```
Mapping Rule 007: TestPath  -> Activity
 Description:
 /*This activity contains all other elements presents in a TestCase.
 This Activity element is a container for all other test elements.*/
Preconditions:
∃tp ∈ TP | type(tp)=TestPath
∃col ∈ B | type(pr)=Process:BPMN
∃pc ∈ B | type(pc)=Class:UML
∃d1 ∈ D |source(d1)=pr, target(d1)=tp, type(d1)=Tests
∃d2 ∈ D |source(d2)=pr, target(d2)=pc, type(d1)=D
Postconditions:
create act, | type(act)=Activity:UML  «TestCase »
act.name=pr.name + "TestCase"+total+1
create TraceLink(act,"Tests", pr)
create Tracelink(act,"Derivation", pc)
```

Listing B.7: A Rule for Mapping a Participant.

```
Mapping Rule 008: Participant -> ActivityPartition
Description:
/* A Participant in a test path is mapped to a ActivityPartition in a test case.*/
Preconditions:
∃tp ∈ TP | type(tp)=TestPath
∃act ∈ AC | type(act)=Activity:UML «TestCase »
```

```
∃d1 ∈ D |source(d1)=tp, target(d1)=act, type(d1)=Derivation
∃par ∈ tp | type(par)=Participant:BPMN
Postconditions:
create ap | type(ap)=ActivityPartition:UML
ap.name=par.name
act.add(ap)
create Tracelink(ap,"Derivation", par)
create Tracelink(act,"Containment", ap)
```

Listing B.8: A Rule for Mapping a Lane.

```
Mapping Rule 009: Lane -> ActivityPartition
Description:
/* A Lane in a test path is mapped to a ActivityPartition in a test case.*/
Preconditions:
∃tp ∈ TP | type(tp)=TestPath
∃act ∈ AC | type(act)=Activity:UML «TestCase »
∃d1 ∈ D |source(d1)=tp, target(d1)=act, type(d1)=Derivation
∃ln ∈ tp | type(ln)=Lane:BPMN
Postconditions:
create ap | type(ap)=ActivityPartition:UML
ap.name=ln.name
act.add(ap)
create Tracelink(ln,"Derivation", ap)
create Tracelink(act,"Containment", ap)
```

Listing B.9: A Rule for Mapping a SequenceFlow.

```
Mapping Rule 010 SequenceFlow -> ControlFlow
Description:
/*A SequenceFlow in BPMN collaboration diagram is mapped to a ControlFlow in activity
     diagram test case.The name and conditions on sequence flows are also mapped
    respectively */
Preconditions:
∃ta ∈ T_M | type(ta)= Actvity, streotype(ta)=TestCase
∃P ∈ S_M |type(P)=Collaboartion
∃d1 ∈ D |source(d1) =ta, target(d1)=P, type(d1)=Tests
∃sf ∈ S_M | type(sf)=SequenceFlow, source(sf)=e1, target(sf)=e2 \mid e1, e2 \in S_{M}
∃d2 ∈ D |source(d2) =P, target(d2)=sf type(d2)=Containment
∃d3, d4 ∈ D | source(d3)=P, target(d3)=e1, type(d3)=Containment
source(d4)=P, target(d4)=e2, type(d4)=Containemnt
∃s1, s2 ∈ T_M
∃d5, d6, d7, d8 ∈ D |
source(d5)=ta, target(d5)=s1, type(d5)=Containment
source(d6)=ta, target(d6)=s2, type(d6)=Containment
source(d7)=e1, target(d8)=s1, type(d7)=Derivation
source(d8)=e2, target(d7)=s2, type(d8)=Derivation

PostConditions:
create cf, | type(cf)=ControlFlow:UML
mapConditions(sf,cf)
source(cf)=s1, target(cf)=s2
ta.add(cf)
create TraceLink(sf,"Derivation", cf)
create Tracelink(ta, Containment,cf)
```

Listing B.10: A Rule for Mapping an ExclusiveGateway in BPMN.

```
Mapping Rule 011 ExclusiveGateway -> ConditionalNode
Description:
/*
```

An ExclusiveGateway in BPMN collaboration diagram is mapped to a ConditionalNode. The name of the ConditionalNode is assigned the name of the SequenceFlow following the gateway.
/
Preconditions:
$\exists ta \in T_M$ | **type(**ta**)=** Actvity, streotype**(**ta**)=**TestCase
$\exists P \in S_M$ |**type(**P**)=**Collaboartion
$\exists d1 \in D$ |**source(**d1**)** =ta, **target(**d1**)=**P, **type(**d1**)=**Tests
$\exists gway \in S_M$ |**type(**gway**)=**ExclusiveGateway
$\exists sf \in S_M$ | **type(**sf**)=**SequenceFlow, **source(**sf**)=**gway
$\exists d2 \in D$ |**source(**d2**)** =P, **target(**d2**)=**sf **type(**d2**)=**Containment
$\exists d2 \in D$ |**source(**d2**)** =P, **target(**d2**)=**gway **type(**d2**)=**Containment

PostConditions:
create cnode, | **type(**cnode**)=**ConditionalNode:UML
name**(**cnode**)=**condition**(**sf**)**
condition**(**cnode**)=**condition**(**sf**)**
ta.add**(**cnode**)**
create TraceLink**(**gway,"Derivation", cnode**)**
create Tracelink**(**ta, Containment,cnode**)**

Listing B.11: A Rule for Mapping an EventBasedGateway in BPMN.

Mapping Rule 012 EventBasedGateway **->** ConditionalNode
Description:
*/**
An EventBasedGateway is assigned to a ConditionalNode with the value true. Instead of the conditions, the AcceptEventActions following this gateway will decide the flow.
/
Preconditions:
$\exists ta \in T_M$ | **type(**ta**)=** Actvity, streotype**(**ta**)=**TestCase
$\exists P \in S_M$ |**type(**P**)=**Collaboartion
$\exists d1 \in D$ |**source(**d1**)** =ta, **target(**d1**)=**P, **type(**d1**)=**Tests
$\exists gway \in S_M$ |**type(**gway**)=**EventBasedGateway
$\exists d2 \in D$ |**source(**d2**)** =P, **target(**d2**)=**gway **type(**d2**)=**Containment

PostConditions:
create cnode, | **type(**cnode**)=**ConditionalNode:UML
name**(**cnode**)=**name**(**gway**)**
condition**(**cnode**)=**true
ta.add**(**cnode**)**
create TraceLink**(**gway,"Derivation", cnode**)**
create Tracelink**(**ta, Containment,cnode**)**

Listing B.12: A Rule for Mapping Various Tasks in BPMN.

Mapping Rule 013: Task**|**ScriptTask**|**ManualTask**|**BusinessRuleTask**|**UserTask **->** OpaqueAction
Description:
*/*The ScriptTask, ManualTask, BusinessRuleTask, and UserTask are mapped to an OpaqueAction.*/*
Preconditions:
$\exists ta \in T_M$ | **type(**ta**)=** Actvity, streotype**(**ta**)=**TestCase
$\exists P \in S_M$ |**type(**P**)=**Collaboartion
$\exists d1 \in D$ |**source(**d1**)** =ta, **target(**d1**)=**P, **type(**d1**)=**Tests
$\exists st \in S_M$ | **type(**st**)=**Task$\vee$ScriptTask$\vee$ManualTask$\vee$BusinessRuleTask$\vee$UserTask$\vee$
$\exists d2 \in D$ |**source(**d2**)** =P, **target(**d2**)=**st, **type(**d2**)=**Containment
$\exists M = (m_1, m_2, m_3, ..., m_n)$ | $\forall m_i \in Mtype(m_i) = MessageFlow, source(m_i) = st$
$\forall i = 1...n$
PostConditions:
create oa, | **type(**oa**)=**OpaqueAction:UML
oa.name**=**st.name
ta.add**(**oa**)**

```
create TraceLink(st,"Derivation", oa)
create Tracelink(ta, Containment,oa)
mapMessages(oa, M)
```

Listing B.13: A Rule for Mapping a Task Calling a Service.

```
Mapping Rule 014: Task (calling a Service) -> CallOperationAction
Description:
/*A task invoking a Service is mapped to a CallOperationAction in a test case
    activity diagram.
This CallOperationAction calls the MockOperation corresponding to the called
    ServiceTask*/
Preconditions:
∃tp ∈ TP | type(tp)=TestPath
∃act ∈ AC | type(act)=Activity:UML «TestCase »
∃d1 ∈ D |source(d1)=tp, target(d1)=act, type(d1)=Derivation
∃ta ∈ tp | type(ta)=Task:BPMN
∃ser ∈ tp | type(ser)=ServiceTask:BPMN
∃d2 ∈ D |source(d2)=ta, target(d2)=ser, type(d2)=Calls
∃op ∈ C_D | type(op)=UML:Operation
∃d3 ∈ D |source(d3)=ser, target(d3)=op, type(d3)=Equivalence
∃mo ∈ T_A | type(mo)=UML:Operation «MockOperation »
∃d4 ∈ D |source(d4)=op, target(d4)=mo, type(d4)=Mocks
∃tCom ∈ T_A | type(tCom)=UML:Class «TestComponent »
∃d5 ∈ D |source(d5)=tCom, target(d5)=mo, type(d5)=Containment
Postconditions:
create ca | type(ca)=CallOperationAction:UML
ca.name=ta.name
set ca.Operation=mo
act.add(ca)
create Tracelink(ta,"Derivation", ca)
create Tracelink(act,"Containment", ca)
create Tracelink(ca,"Calls", mo)
create Tracelink(act,"Requires", tCom)
```

Listing B.14: A Rule for Mapping a SendTask in BPMN.

```
Mapping Rule 015: SendTask -> SendSignalAction
Description:
/*A non service calling SendTask is mapped to a SendSignalAction in the test case
    activity diagram. A dependency relations between the TestComponent corresponding
    to the receiver Participant is also created*/
Preconditions:
∃tp ∈ TP | type(tp)=TestPath
∃ta ∈ AC | type(ta)= Activity:UML «TestCase »
∃p ∈ S_M |type(p)=Collaboartion:BPMN
∃d1 ∈ D |source(d1) =ta, target(d1)=p, type(d1)=Tests
∃st ∈ tp | type(st)=SendTask:BPMN
∃d2 ∈ D |source(d2) =P, target(d2)=st, type(d2)=Containment
∃M = (m_1, m_2, m_3, ..., m_n) | ∀m_i ∈ M type(m_i)=MessageFlow, source(m_i)=st (∀i = 1...n)
[⋀_{j=1}^{n}[¬(∃m_{j}∈ M \mid type(target(m_j))=ServiceTask)
∧∃m_{j}∈ M∃par ∈ p | type(par)=Participant:BPMN target(m_j)=par]
∃tcom | type(tcom)Class:UML «TestComponent » ]
∃d3 ∈ D |source(d3) =par, target(d3)=tcom, type(d3)=Mocks
Postconditions:
create ssa, | type(ssa)=SendsignalAction:UML
ssa.name=st.name
mapMessageFlows(M, ssa)
ta.add(ssa)
create TraceLink(ta,"Containment", ssa)
create Tracelink(st,"Derivation", ssa)
create Tracelink(ssa,"Communication", tCom)
create Tracelink(act,"Requires", tcom)
```

Listing B.15: A Rule for Mapping a SendTask (Service Calling).

Mapping Rule Rule 016: Send Task **(**ServiceCalling) **->** CallOperationAction
Description:
/ Since, call to a service is synchronous and the the target waits for the reply, thus a service calling SendTask is mapped to a CallOperationAction */*
Preconditions:
$\exists tp \in TP$ | **type(**tp)**=**TestPath
$\exists act \in AC$ | **type(**act)**=** Activity:UML «TestCase »
$\exists p \in S_M$ |**type(**p)**=**Collaboartion:BPMN
$\exists d1 \in D$ |**source(**d1) **=**act, **target(**d1)**=**p, **type(**d1)**=**Tests
$\exists st \in tp$ | **type(**st)**=**SendTask:BPMN
$\exists d2 \in D$ |**source(**d2) **=**P, **target(**d2)**=**st, **type(**d2)**=**Containment
$\exists M = (m_1, m_2, m_3, ..., m_n)$ | $\forall m_i \in M$ **type(**m_i)**=**MessageFlow, **source(**m_i)**=**st $(\forall i = 1...n)$
$[\bigwedge_{j=1}^{n} \exists$m_**{**j**}**$\in M \exists st$ | **type(**st)**=**ServiceTask:BPMN $\wedge$ **target(**m_j)**=**st**]**
$\exists par$ |**type(**par)**=**Participant:BPMN
$\exists d3 \in D$ |**source(**d3) **=**par, **target(**d3)**=**st, **type(**d3)**=**Containment
$\exists tcom$ | **type(**tcom)Class:UML «TestComponent » **]**
$\exists d4 \in D$ |**source(**d4) **=**par, **target(**d4)**=**tcom, **type(**d4)**=**Mocks
$\exists mo \in tcom$ | **type(**mo)**=**UML:Operation «MockOperation »
$\exists d5 \in D$ |**source(**d5)**=**st, **target(**d5)**=**mo, **type(**d5)**=**Mocks
Postconditions:
create coa, | **type(**coa)**=**CallOperationAction:UML
coa.name**=**st.name
set coa.Operation**=**mo
mapMessages**(**M, coa)
act.add**(**coa)
create TraceLink**(**act,"Containment", coa)
create Tracelink**(**st,"Derivation", coa)
create Tracelink**(**coa,"Calls", mo)
create Tracelink**(**act,"Requires", tCom)

Listing B.16: A Rule for Mapping a ReceiveTask in BPMN.

Mapping Rule 017: ReceiveTask **->** AcceptEventAction
Description:
*/*A ReceiveTask is mapped to an AcceptEventAction in the TestCase as this action can accept the incoming messages.*/*
Preconditions:
$\exists ta \in T_M$ | **type(**ta)**=** Actvity, streotype**(**ta)**=**TestCase
$\exists P \in S_M$ |**type(**P)**=**Collaboartion
$\exists d1 \in D$ |**source(**d1) **=**ta, **target(**d1)**=**P, **type(**d1)**=**Tests
$\exists rt \in S_M$ | **type(**rt)**=**ReceiveTask
$\exists d2 \in D$ |**source(**d2) **=**P, **target(**d2)**=**rt, **type(**d2)**=**Containment
PostConditions:
create aea, | **type(**aea)**=**CallOperationAction:UML
aea.name**=**rt
ta.add**(**aea)
create TraceLink**(**ta,"Containment", aea)
create Tracelink**(**rt,"Derivation", aea)
mapOutputPin**()** */*to the message*/*

Listing B.17: A Rule for Mapping a StartEvent in BPMN

Mapping Rule 018: StartEvent **->** ActivityInitialNode
Description:
*/*A StartEvent in BPMN collaboration diagram is mapped to an ActivityInitialNode in activity diagram test case*/*
Preconditions:
$\exists ta \in T_M$ | **type(**ta)**=** Actvity, streotype**(**ta)**=**TestCase
$\exists P \in S_M$ |**type(**P)**=**Collaboartion
$\exists d1 \in D$ |**source(**d1) **=**ta, **target(**d1)**=**P, **type(**d1)**=**Tests
$\exists se \in S_M$ | **type(**se)**=**StartEvent
$\exists d2 \in D$ |**source(**d2) **=**P, **target(**d2)**=**se **type(**d2)**=**Containment

```
PostConditions:
create anode, | type(anode)=ActivityInitialNode:UML
ta.add(anode)
anode.name=se.name
create TraceLink(sa,"Derivation", anode)
create Tracelink(ta, Containment,anode)
```

Listing B.18: A Rule for Mapping ConditionalStartEvent in BPMN.

```
Mapping Rule 020: ConditionalStartEvent-> InitialNode,ControlFlow
Description:
/*A ConditionalStartEvent is mapped to an InitialNode and the outgoing ControlFlow of
     the InitialNode contains the condition of the ConditionalStartEvent */
PreConditions:
∃ta ∈ T_M | type(ta)= Actvity, streotype(ta)=TestCase
∃P ∈ S_M |type(P)=Collaboartion
∃d1 ∈ D |source(d1) =ta, target(d1)=P, type(d1)=Tests
∃se ∈ S_M | type(se)=ConditionalStartEvent
∃d2 ∈ D |source(d2) =P, target(d2)=se type(d2)=Containment

PostConditions:
create anode | type(anode)=ActivityInitialNode:UML
create cf | type(cf)=ControlFlow:UML
ta.add(anode)
ta.add(cf)
anode.name=se.name
condition(cf)=condition(se)
source(cf)=anode
create TraceLink(sa,"Derivation", anode, cf)
create Tracelink(ta, Containment,anode, cf)
```

Listing B.19: A Rule for Mapping an EndEvent in BPMN.

```
Mapping Rule 021: EndEvent -> ActivityFinalNode
Description:
/*An EndEvent in BPMN collaboration diagram is mapped to ActivityFinalNode in
    activity diagram test case.*/
Preconditions:
∃ta ∈ T_M | type(ta)= Actvity, streotype(ta)=TestCase
∃P ∈ S_M |type(P)=Collaboartion
∃d1 ∈ D |source(d1) =ta, target(d1)=P, type(d1)=Tests
∃se ∈ S_M | type(ee)=EndEvent
∃d2 ∈ D |source(d2) =P, target(d2)=ee type(d2)=Containment

PostConditions:
create enode, | type(enode)=ActivityFinalNode:UML
ta.add(enode)
enode.name=ee.name
create TraceLink(sa,"Derivation", enode)
create Tracelink(ta, Containment,enode)
```

Listing B.20: A Rule for Mapping a MessageEndEvent in BPMN.

```
Mapping Rule 022: MessageEndEvent -> AcceptEventAction, ControlFlow,
    ActivityFinalNode
Description:
/*Similar to the MessageStartEvent, a MessageEndEvent waits for a message and then
    ends the process. Thus it is mapped to an AcceptEventAction followed by
    ActivityFinalNode*/
Preconditions:
```

$\exists ta \in T_M$ | **type(**ta)**=** Actvity, streotype**(**ta)**=**TestCase
$\exists P \in S_M$ |**type(**P)**=**Collaboartion
$\exists d1 \in D$ |**source(**d1) **=**ta, **target(**d1)**=**P, **type(**d1)**=**Tests
$\exists se \in S_M$ | **type(**ee)**=**MessageEndEvent
$\exists d2 \in D$ |**source(**d2) **=**P, **target(**d2)**=**ee **type(**d2)**=**Containment

PostConditions:
create aea | **type(**aea)**=**AcceptEventAction:UML
create cf | **type(**cf)**=**ControlFlow:UML
create enode | **type(**enode)**=**ActivityFinalNode:UML
ta.add**(**aea)
ta.add**(**cf)
ta.add**(**enode)
enode.name**=**ee.name
source(cf)**=**aea
target(cf)**=**enode
create TraceLink**(**sa,"Derivation", aea, cf, enode)
create Tracelink**(**ta, Containment,aea, cf, enode)

C

List of Dependencies Between System Views

Table C.1: Dependency Relations for TestView of Business Processes.

Source UTP Element	Target Element	Dependency Type	Target View	Description	Rules
TestCase	Activity:UTP «TestCase »	Definition	TBV	A TestCase defines the structure of the activity diagram representing the test behavior.	MRule:B.5, MRule:B.15, DDRule014
TestCase	TestComponent:UTP	Requires	TAV	A TestCase requires a TestComponent, if the original Component required is not available.	MRule:B.13, MRule:B.14
TestCase	Process:BPMN	Tests	BPV	A TestCases tests a Process under test.	DDRule020
TestContext	TestComponent:UTP	Usage	TAV	The TestContext uses various test components to execute its test cases.	MRule:B.3, DDRule013
TestContext:UTP	TestControl:UTP	Containment	TAV	The TestContext contains a TestControl operation, which defines the sequence of execution of test cases contained by the TestContext.	DDRule027 (Indirect)
TestContext	TestCase:UTP	Containment	TAV	A TestContext is the container for test cases required to test SUT.	MRule:B.5, DDRule012
TestContext:UTP	Class:UML «ProcessClass »	Derivation	BSV	The TestContext is derived from a ProcessClass during the test generation.	MRule:B.2
TestContext, TestComponent	TestModel:UTP	Containemnet	TAV	A TestModel is an indirect parent of both TestComponent and TestContext.	DDRule025
TestComponent	MockOperation:UTP	Containment	TAV	Test components contain mock operations, which simulate the behavior of operations required by test cases.	MRule:B.3, MRule:B.4, DDRule015
TestComponent	Lane:BPMN	Derivation	BPV	A TestComponent can be derived from a Lane.	MRule:B.4, DDRule016
TestComponent	Participant:BPMN	Derivation	BPV	A TestComponent can be derived from a Participant of a collaborative process.	MRule:B.3, DDRule001
TestComponent	Component:UML	Mocks	BSV	A TestComponent mocks the behavior of a Component.	MRule:B.3
TestPackage	TestContext:UTP	Containment	TAV	A TestPackage is a container for the TestContext [BDG+07].	MRule:B.2, DDRule007
TestPackage	SUT:UTP	Tests	TAV	A TestPackage is required to test a SUT.	MRule:B.1, DDRule010
TestPackage	TestComponent:UTP	Containment	TAV	A TestPackage contains test components to test a SUT [BDG+07].	MRule:B.3, MRule:B.4, DDRule11
TestPackage	Class:UML «ProcessClass »	Derivation	BSV	A TestPackage is derived from a ProcessClass during the test generation.	MRule:5.1
TestModel	TestPackage:UTP	Containment	TAV	A TestPackage is contained in a TestModel, which tests a particular process	DDRule008

Continued on next page–See legends at table end

Table C.1 Continued from previous page

Source UTP Element	Target Element	Dependency Type	Target View	Description	Rules
TestModel	SUT:UTP	Containment	TAV	A TestModel contains the SUT, which corresponds to a Process.	DDRule009
TestModel	Class:UML «ProcessClass »	Tests	BSV	A TestModel is derived from a ProcessClass during the test generation and it tests the process corresponding to the ProcessClass.	DDRule053
TestModel	Collaboration:BPMN	Tests	BPV	For each Collaboration in BPMN collaboration diagram, a TestModel is specified which contains all test related information of the collaborative process.	DDRule006
SUT	Class:UML «ProcessClass »	Definition	BSV	The ProcessClass provides the definition of a SUT, which is the process to be tested.	MRule:B.1, DDRule021
SUT	Operation (in SUT)	Containment	TAV	The *SUT* contains the available operations of the process to be tested.	MRule:B.1, DDRule027 (Indirect)
Operation (in SUT)	Opeartion:UML	Equivalnece	BSV	The operations provided by SUT are defined in the corresponding ProcessClass.	DDRule021
MockOperation	Operation:UML	Mocks	BSV	A MockOperation mocks the behavior of an actual operation from UML class diagram for a particular TestContext.	MRule:B.4, MRule:B.3
MockOperation	SericeTask:BPMN	Mocks	BPV	A MockOperation is derived from a ServiceTask during the test generation.	DDRule054
MockOperation	SendTask:BPMN	Mocks	BPV	MockOperation is derived from a SendTask during the test generation.	DDRule062
MockOperation	ReceiveTask:BPMN	Mocks	BPV	MockOperation is derived from a ReceiveTask during the test generation.	DDRule064
Activity «TestCase »	Process:BPMN	Tests	BPV	An activity diagram test cases tests a Process	MRule:B.6
Activity «TestCase »	Process:BPMN	Derivation	BPV	An activity diagram test cases is derived from a Process during the test generation	MRule:B.6
Activity «TestCase »	ActivityPartition: UTP	Containment	TBV	An activity diagram test case can consists of one or more activity partitions.	DDRule043, MRule:B.7, MRule:B.8
CallOperationAction	Operation:UTP	Calls	TAV	A CallOperationAction inside an activity diagram test case calls mock operations and operations of SUT.	MRule:B.13, MRule:B.15, DDRule044
AcceptEventAction	ReceiveTask:BPMN	Derivation	BPV	A ReceiveTask in BPMN collaboration diagram is mapped onto an AcceptEventAction in activity diagram test case during the test generation.	MRule:B.16
SendSignalAction	SendTask:BPMN	Derivation	BPV	A SendTask in BPMN collaboration diagram is mapped onto an SendSignalAction in activity diagram test case during the test generation	MRule:B.14

Continued on next page–See legends at table end

Table C.1 Continued from previous page

Source UTP Element	Target Element	Dependency Type	Target View	Description	Rules
CallOperationAction	SendTask:BPMN	Derivation	BPV	A SendTask whose outgoing MessageFlow goes to a ServiceTask is mapped onto a CallOperationAction during the test generation.	MRule:B.15
Activity «TestCase »	SendSignalAction:UML	Containment	TBV	An activity diagram test case contains a set of SendSignalActions derived from send tasks in a collaboration diagram.	MRule:B.14
SendSignalAction	TestComponent:UTP	Communication	BPV	A SendSignalAction communicates with the test components, which act as message receivers.	MRule:B.14
InitialNode, AcceptEventAction	SignalStartEvent: BPMN	Derivation	BPV	A SignalStartEvent in BPMN collaboration diagram is mapped onto an InitialNode followed by an AcceptEventAction in the test case activity diagram.	MRule:5.2
InitialNode	StartEvent	Derivation	BPV	An InitialNodein an activity diagram test case is derived from a StartEvent in BPMN collaboration diagram.	MRule:B.17
ControlFlow	SequenceFlow:BPMN	Derivation	BPV	A ControlFlow in an activity diagram test case is derived from a SequenceFlow from BPMN collaboration diagram.	MRule:B.9
InitialNode, AcceptEventAction	MessageStartEvent:BPMN	Derivation	BPV	A MessageStartEvent in BPMN collaboration diagram is mapped onto and InitialNode followed by an AcceptEventAction in the activity diagram test case.	MRule:5.2
AcceptEventAction, ActivityFinalNode	MessageEndEvent: BPMN	Derivation	BPV	A MessageEndEvent in BPMN collaboration diagram is mapped onto an AcceptEventAction followed by an InitialNode in the activity diagram test case.	MRule:B.20
ActivityPartition	Lane:BPMN	Derivation	BPV	An ActivityPartition in an activity diagram test case is derived from a Lane from BPMN collaboration diagram during the test generation	MRule:B.8
OpaqueAction	Task: BPMN	Derivation	BPV	A simple Task in BPMN is mapped onto an OpaqueAction in UML activity diagram test case.	MRule:B.12
OpaqueAction	BusinessRuleTask: BPMN	Derivation	BPV	A BusinessRuleTask is mapped onto an OpaqueAction in UML activity diagram test case.	MRule:B.12
OpaqueAction	ScriptTask: BPMN	Derivation	BPV	A ScriptTask is mapped onto an OpaqueAction in UML activity diagram test case.	MRule:B.12
OpaqueAction	ManualTask: BPMN	Derivation	BPV	A ManualTask is mapped onto an OpaqueAction in UML activity diagram test case.	MRule:B.12
OpaqueAction	UserTask: BPMN	Derivation	BPV	A UserTask is mapped onto an OpaqueAction in UML activity diagram test case	MRule:B.12
Activity «TestCase »	OpaqueAction:UML	Containment	TBV	An activity diagram test case contains a set of OpaqueActions derived from various tasks in BPMN collaboration diagram.	MRule:B.12, MRule:5.2, MRule:B.20, DDRule042

Continued on next page–See legends at table end

Table C.1 Continued from previous page

Source UTP Element	Target Element	Dependency Type	Target View	Description	Rules
InitialNode, ControlFlow	ConditionalStartEvent:BPMN	Derivation	TBV	A ConditionalStartEvent is mapped onto an InitialNode followed by a ControlFlow, where the ControlFlow contains the condition of the ConditionalStarteEvent.	MRule:B.18
ActivityFinalNode	EndEvent:BPMN	Derivation	BPV	An EndEvent in a BPMN collaboration diagram maps onto an ActivityFinalNode in the test case activity diagram during the test generation.	MRule:B.19, DDRule048
DecisionNode	ExclusiveGateway:BPMN	Derivation	BPV	An ExclusiveGateway in a BPMN collaboration diagram maps onto an DecisionNode in the test case activity diagram during the test generation.	MRule:B.10
Activity «TestCase »	DecisionNode:UML	Containment	TBV	The activity diagram test case contains a set of decision nodes derived from the BPMN collaboration diagram.	MRule:B.10, MRule:B.11, DDRule042
DecisionNode	EvenBasedGateway:BPMN	Derivation	BPV	An EventBasedGateway in BPMN is mapped onto a DecisionNode in the activity diagram test case.	MRule:B.11
Activity «TestCase »	ControlFlow:UML	Containment	TBV	An activity diagram test case contains a set of control flows to model the test execution flow	MRule:B.9, MRule:5.2, MRule:B.18, MRule:B.20, DDRule042
Activity «TestCase »	CallOperationAction: UTP	Containment	TBV	An activity diagram test case contains CallOperationAction to call various operations of SUT and test components during the test execution.	MRule:B.13, MRule:B.15, DDRule044
CallOperationAction: UTP	Task:BPMN	Derivation	TBV	A CallOperationAction in an activity diagram test case can be derived from a Task calling a service.	MRule:B.13, DDRule051
Activity «TestCase »	InitialNode:UML	Containment	TBV	An activity diagram test case contains an InitialNode to show the start of the test execution.	MRule:B.17, MRule:5.2, MRule:B.18, DDRule045
InitialNode:UTP	StartEvent:BPMN	Derivation	TBV	An InitialNode in an activity diagram test case is derived from a StartEvent in BPMN collaboration diagram.	DDRule049
Activity «TestCase »	ActivityFinalNode:UTP	Containment	TBV	An activity diagram test case contains a ActivityFinalNode to represent the completion of the test execution.	MRule:B.19, MRule:B.20, DDRule046
ActivityPartition	Participant:BPMN	Derivation	BPV	An ActivityPartition in a test case activity diagram is derived from a Participant of BPMN collaboration diagram during the test generation.	MRule:B.7

Continued on next page–See legends at table end

Table C.1 Continued from previous page

Source UTP Element	Target Element	Dependency Type	Target View	Description	Rules
TestModel	TestDataModel	Uses	TDV	A TestDataModel is used by a test architecture model, as it defines the data pools and partitions required for various test cases.	DDRule057
Activity «TestCase »	TestDataModel	Uses	TDV	An activity diagram test case uses the concrete test data specified in a TestDataModel.	DDRule058
DataPool	Class	Specializes	TDV	A DataPool specializes a class and defines various cases of test data corresponding to the original class.	DDRule059
DataPartition	DataPool	Extends	TDV	A DataPartition extends a DataPool to define various cases of a test data.	DDRule060 (Indirect)

LEGENDS:

BSV : Business Structure View TAV : Test Architecture View MRule : Mapping Rule
BPV : Business Process View TBV : Test Behavior View

Table C.2: Dependency Relations for Process View and Structural View.

Source Element	Source View	Target Element	Target View	Dependency Type	Description	Rules
Class:UML	BSV	Interface:UML	BSV	Realization	The provided interface of a component, in our case models a distinct service provided, or a set of operations provided. An interface is realized by a Class in UML class diagram.	DDRule041
Class:UML	BSV	Operation:UML	BSV	Containment	A class contains a set of Operations	DDRule027
Component:UML	BSV	Participant:BPMN	BPV	Equivalence	In SoaML modeling method, a component is created for each Participant, as a Participant can be a role, a software component, or a GUI component	DDRule002
Component:UML	BSV	Interface:UML	BSV	Provides	A component provides interfaces to access its services	DDRule041
Component:UML	BSV	Interface:UML	BSV	Requires	A component requires other interfaces to realize its services	
Component:UML	BSV	Class:UML	BSV	Implementation	A Class in UML class diagram can implement a Component from UML component diagram	DDrule005
Collaboration: BPMN	BPV	Participant:BPMN	BPV	Containment	The parent container of a participant is a collaboration. These relations are enforced by the meta-models to package the related elements	DDRule039
Interface:UML	BSV	Interface:UML	BSV	Equivalence	An interface of a component can be defined in the UML class diagram by defining its operations in detail	DDRule038
Interface:UML	BSV	Operation:UML	BSV	Containment	An interface can contain a set of operations	DDRule027

Continued on next page–See legends at table end

Table C.2 Continued from previous page

Source Element	Source View	Target Element	Target View	Dependency Type	Description	Rules
Lane:BPMN	BPV	Class:UML	BSV	Definition	A service class in UML class diagram implements the service tasks of lanes, since a Participant can represent a Component, we model a Class belonging to that Component as a Lane	DDRule018
Operation:UML	BSV	Parameter:UML	BSV	Containment	A operation contains a list of parameters	DDRule037
Participant:BPMN	BPV	Process:BPMN	BPV	Initiation	A Participant initiates a business process	DDRule003
Participant:BPMN	BPV	ServiceTask:BPMN	BPV	Provision	A Participant provides services as ServiceTasks	DDRule065
Process:BPMN	BPV	Task:BPMN	BPV	Containment	A process can use a set of tasks to implement various abstract steps	DDRule029
Process:BPMN	BPV	SendTask:BPMN	BPV	Containment	A process can send messages to other participants or lanes interacting with it	DDRule030
Process:BPMN	BPV	ReceiveTask:BPMN	BPV	Containment	A process can receive messages sent by other participants or lanes interacting with it	DDRule031
Process:BPMN	BPV	ServiceTask:BPMN	BPV	Containment	A process can provide a set of services modeled as ServiceTasks.	DDRule028
Process:BPMN	BPV	StartEvent:BPMN	BPV	Containment	A process contains exactly one StartEvent in our approach to ease the test generation.	DDRule032
Process:BPMN	BPV	EndEvent:BPMN	BPV	Containment	A process can contain exactly one EndEvent to ease the test generation	DDRule033
Process:BPMN	BPV	SequenceFlow:BPMN	BPV	Containment	A process contains a set of sequence flows between various tasks and elements	DDRule034
Process:BPMN	BPV	EventBasedGateway :BPMN	BPV	Containment	A process can contain a set of event-based gateways to models event-based decisions.	DDRule035
Process:BPMN	BPV	ExclusiveGateway :BPMN	BPV	Containment	To model decision based on values of various conditions, a process can use a set of exclusive gateways	DDRule036
Process:BPMN	BPV	Lane:BPMN	BPV	Containment	A process contains a set of Lanes to distinguish between various roles of participants. In our approach, a Participant can be a component and a Lane can be a class which belongs to this component	DDRule017
ProcessClass:UML	BSV	Process:BPMN	BPV	Definition	A process is defined by a ProcessClass in the structural view	DDRule004
Package:UML	BSV	Package:UML	BSV	Containment	A Package can contain other sub packages	DDRule023
Package:UML	BSV	Class:UML	BSV	Containment	A Package can contain a set of classes	DDRule047
Lane:BPMN	BPV	DocumentRoot:BPMN	BPV	Containment	A DocumentRoot indirectly conatins a set oflanes	DDRule024
Lane:BPMN	BPV	ServiceTask:BPMN	BPV	Provision	Lane provides services in the form of service tasks	DDRule026
ServiceTask:BPMN	BPV	Operation:UML	BSV	Definition	The definition of the service called by a ServiceTask is provided as a Operation in the class corresponding to the provider Participant or Lane	DDRule026
Task:BPMN	BPV	ServiceTask:BPMN	BPV	Calls	A Task in a Process can call a service of another Participant using ServiceTask	DDRule050

Continued on next page–See legends at table end

Table C.2 Continued from previous page

Source Element	Source View	Target Element	Target View	Dependency Type	Description	Rules
`Class:UML`	BSV	`Class:UML`	BSV	Extends	This dependency defines a Parent-Child relationship between two classes.	DDRule060
`SendTask:BMN`	BPV	`Operation:UML`	BSV	Equivalence	A `SendTask` is defined as an `Operation` in the `Class` corresponding to its parent `Lane`	DDRule061
`SendTask:BPMN`	BPV	`Lane:BPMN`	BPV	Communicates	A `SendTask` is contained by its parent `Lane`	DDRule061
`ReceiveTask:BMN`	BPV	`Operation:UML`	BSV	Equivalence	A `ReceiveTask` is defined as an `Operation` in the `Class` corresponding to its parent `Lane`	DDRule063
`ReceiveTask:BPMN`	BPV	`Lane:BPMN`	BPV	Communicates	A `ReceiveTask` is contained by its parent `Lane`	DDRule063

LEGENDS:

BSV : Business Structure View BPV : Business Process View

D

Change Types and Scenarios

Table D.1: Generic Change Types for BPMN Collaboration Diagrams (*process view*.

ModelElement	ChangeType	Comosition	Description	Rules
Participant	Add	Atomic	Add Participant in Collaboration	
Participant	Delete	Atomic	Delete participant from Collaboration	
Participant	PU:name	Atomic	Rename a Participant	
Participant	Merge	Composite	Merge Two participants into one (To merge n+1 participants, the merge operation has to be applied n times)	
Participant	Split	Composite	Split a Participant into two Participants	
Participant	Others	Composite	Move, Swap, and Replace are not applicable on participants	
Process	Add	Atomic	Add a Process in Collaboration	
Process	Delete	Atomic	Delete a Process from Collaboration	
Process	PU:name	Atomic	Rename a Process	
Process	PU:type	Atomic	Change visibility of a process. Type specifies if a process is private or public	
Process	Merge	Composite	Merge two processes	
Process	Split	Composite	Split a Process into two	
Process	Swap, Replace, Move	Composite	Not applicable	
Task[1]	Add	Atomic	Add a task in a Process	
Task	Delete	Atomic	Delete a task from the model	
Task	PU:name	Atomic	Rename a task	
Task	Move	Composite	Move a task from one lane to another	
Task	Move	Composite	Move a task from one Participant to another (only valid for service tasks)	
Task	Split	Composite	Split a task into two	
Task	Merge	Composite	Merge two tasks	
Task	Replace	Composite	Replace a Task with another (only applicable to the service tasks)	
Task	Swap	Composite	Swap is not applicable to tasks	
SequenceFlow	Add	Atomic	Add a sequence flow between two tasks. Similar to the change pattern add control dependency by Weber et al. [WR08]	
SequenceFlow	Delete	Atomic	Delete a Sequence Flow between two tasks. Similar to the change pattern remove control dependency by Weber et al. [WR08]	
SequenceFlow	others	Composite	Move|Merge|Replace|Swap are not applicable on sequence flows	
SequenceFlow	PU:Condition	Atomic	Update the conditional expression on a SequenceFlow	
MessageFlow	Add	Atomic	Add a message flow between two tasks	
MessageFlow	Delete	Atomic	Remove a a message flow between two tasks	
MessageFlow	Others	Composite	Move|Merge|Replace|Swap are not applicable on a MessageFlow	
Lane	Add	Atomic	Add a lane inside a Process	
Lane	Delete	Atomic	Delete a Lane from a Process	
Lane	PU:name	Atomic	Rename a Lane	
Lane	Merge	Composite	Merge two lanes	
Lane	Split	Composite	Split a Lane into two	
Lane	Swap|Replace	Composite	Not Applicable	
Gateway	Add	Atomic	Add a gateway inside a process	

[1] tasks include Send, Receive, Service, Manual, Script, and User tasks

Gateway	Delete	Atomic	Delete a gateway from a process	
Gateway	PU:name	Atomic	Rename a gateway	
Gateway	Move	Composite	Move a gateway from one lane to another	
Data Store	Add	Atomic	Add a DataStore inside a Collaboration	
Data Store	Delete	Atomic	Delete a DataStore from a Collaboration	
Data Store	Delete	Atomic	Delete a DataStore from a Collaboration	
Data Store	PU:name	Atomic	Rename a DataStore	
Data Store	Merge	Composite	Two Data stores are merged	
Data Store	Split	Composite	A data store is splitted into two	
Data Store	Replace	Composite	Replace a data store with another	

Table D.2: Generic Change Types for Models of Structural View

ModelElement	**ChangeType**	**Composition**	**Description**	**Rules**
Class	Add	Atomic	Add a class in a model	
Class	Delete	Atomic	Delete a class from a model	
Class	Swap	Composite	Swap a class with another class	
Class	Split	Composite	Split a class into two classes	
Class	Merge	Composite	Merge two classes into one	
Class	Move	Composite	Move a Class from one Package to another	
Class	Move	Composite	Move a Class from one Component to another	
Class	PU:name	Atomic	Rename a class	
Class	PU:abstract	Atomic	Make a class abstract	
Class	PU:visibility	Atomic	Change visibility of a class	
Class	PU:static	Atomic	Make a class static	
Property	Add	Atomic	Add an attribute to a class	
Property	Delete	Atomic	Delete an attribute from a class	
Property	PU:name	Atomic	Rename an attribute of a class	
Property	PU:final	Atomic	Change final modifier of an attribute	
Property	PU:static	Atomic	Make an attribute static by changing the static modifier	
Property	PU:type	Atomic	Change type of an attribute	
Property	PU:visibility	Atomic	Change visibility of an attribute	
Property	Move	Composite	Move an attribute from one class to another	
Operation	Add	Atomic	Add an operation to a class	
Operation	Delete	Atomic	Remove an operation from the class	
Operation	Move	Composite	Move an operation from one class to another	
Operation	PU:returnType	Atomic	Change return type of an operation	
Parameter	PU:Type	Atomic	Change data type of a parameter of an operation	
Parameter	Delete	Atomic	Delete a parameter from an Operation	
Parameter	Add	Atomic	Add a new parameter in an Operation	
Operation	PU:Static	Atomic	Make an operation static	
Operation	PU:name	Atomic	Rename an Operation	
Parameter	PU:name	Atomic	Rename a parameter of an Operation	
Parameter	PU:final	Atomic	Change final modifier of a parameter of an Operation	
Operation	Add	Atomic	Add an abstract Operation in a class	
Operation	Delete	Atomic	Remove an abstract method from the class	
Association(UML Property with an ownedEnd)	Add	Atomic	Add an Association/composition/Aggregation between two classes/interface	
Association	Delete	Atomic	Association is a Property with an owned end in UML. Delete an Association/composition/Aggregation between two classes/interface	
Association	PU:multiplicity	Atomic	Change the multiplicity of an association end	
Association	PU: visibility	Atomic	Change the Visibility of an association end	
Inheritance	Add	Atomic	Add an Inheritance relation between two classes/interface	
Inheritance	Add	Atomic	Remove an Inheritance relation from two classes/interface	
Interface	Add	Atomic	Add an Interface to a Model	

Interface	Delete	Atomic	Delete an Interface from the model	
Interface	PU:name	Atomic	Rename an interface	
Component	Add	Atomic	Add a component in a model	
Component	Delete	Atomic	Delete a component from a model	
Component	Split	Composite	Split a component in to two	
Component	Merge	Composite	Merge two components	
Interface	Add	Atomic	Add Interface (Required/Provided) to a Component	
Interface	Delete	Atomic	Remove Interface (Required/Provided) of a Component	
Interface	Move	Composite	Move Interface (Required/Provided) from one Component to another	

Table D.3: Change Types for UTP *test view*.

ModelElement	**ChangeType**	**Composition**	**Description**	**Rules**
TestModel	Add	Atomic	Add a new TestModel	
TestModel	Delete	Atomic	Delete a TestModel	
TestModel	PU:name	Atomic	Rename a TestModel	
TestPackage	Add	Atomic	Add a TestPackage in a TestModel	
TestPackage	Delete	Atomic	Remove a TestPackage from a TestModel	
TestPackage	PU:name	Atomic	Rename a TestPackage	
TestContext	Add	Atomic	Add a TestContext in a TestPackage	
TestContext	Delete	Atomic	Remove a TestContext from a TestPackage	
TestContext	PU:name	Atomic	Rename a TestContext	
SUT	Add	Atomic	Add a SUT in a TestModel	
SUT	Delete	Atomic	Remove a SUT from a TestModel	
SUT	PU:name	Atomic	Rename a SUT	
TestCase	Add	Atomic	Add a TestCase in a TestContext	
TestCase	Delete	Atomic	Remove a TestCase from a TestContext	
TestCase	PU:name	Atomic	Rename a TestCase	
MockOperation	Add	Atomic	Add a MockOperation in a TestComponent	
MockOperation	delete	Atomic	Remove a MockOperation from a TestComponent	
MockOperation	U:name	Atomic	Rename a MockOperation	
TestComponent	Add	Atomic	Add a TestComponent in a TestPackage	
TestComponent	Delete	Atomic	Remove a TestComponent from a TestPackage	
TestComponent	PU:name	Atomic	Rename a TestComponent	
Activity	Add	Atomic	Add an activity in a Model	
Activity	Delete	Atomic	Remove an activity from a Model	
Activity	PU:name	Atomic	Rename an activity	
CallOperationAction	Add	Atomic	Add a CallOperationAction in an Activity	
CallOperationAction	Delete	Atomic	Remove a CallOperationAction from an Activity	
CallOperationAction	PU:name	Atomic	Rename an Activity	
ActivityPartition	Add	Atomic	Add an ActivityPartition in an Activity	
ActivityPartition	Delete	Atomic	Remove an ActivityPartition from an Activity	
ActivityPartition	PU:name	Atomic	Rename an ActivityPartition from an Activity	

Table D.4: Change Scenarios For Mobile Service Technician Case Study.

ID	ModelElement
Change Scenario 1	
Maintenance Type	Perfective
Scenario Description	In the *HandleTourPlanningProcess*, the creation of a *TourPlan* requires assigning the *Tour* and the *ServiceOrders* selected for the *Tour* to the *TourPlan*. Otherwise, an empty *TourPlan* object would be kept in the system, which would violate the constraints of the system design. Thus, the replacement of the old *createTourPlan* operation is required with a new operation, which takes the parameters `currentTour` and a list of *SericeOrders*.
Required Changes	**Change 1** name: Add Operation abstraction: Concrete composition: Atomic source: createTourPlan(`Tour` currentTour, `ServiceOrders` so) :Operation target: HandleTourPlanningProcess:Class **Change 2** name: Add Parameter abstraction: Concrete composition: Atomic source: currentTour:Parameter, type=Tour target: createTourPlan:Operation **Change 3** name: Add Parameter abstraction: Concrete composition: Atomic source: so:Parameter, type=ServiceOrder<List> target: createTourPlan:Operation **Change 4** name: Add ServiceTask abstraction: Concrete composition: Atomic source: HandleTourPlanningProcess:Process target: createTourPlan:ServiceTask **Change 5** name: Replace ServiceTask abstraction: Concrete composition: composite source: createTourPlan():Operation target: *createTourPlan* (`Tour` currentTour, `ServiceOrders` so)
Change Scenario 2	
Maintenance Type	Perfective
Scenario Description	In the *EvaluateProblemProcess*, it is required to check if some service technicians were assigned the device already and check their availability on priority, there is no such check and it needs to be incorporated in the process.
Required Changes	**Change 1** name: Add Service Task abstraction: Concrete composition: Atomic source: getPriorityServiceTechnician:ServcieTask target: EvaluateProblemProcess:Lane **Change 2** name: Add ServiceTask abstraction: Concrete composition: Atomic source: assignPriorityServiceTechnicaintoSO:ServcieTask target: EvaluateProblemProcess:Lane **Change 3** name: Delete SequenceFlow abstraction: Concrete composition: Atomic source: saveServiceOrder:ServiceTask target: saveServiceQoutation:ServiceTask **Change 4** name: Add SequenceFlow abstraction: Concrete composition: Atomic source: between saveServiceQoutation:ServcieTask target: getPriorityServiceTechnician:ServiceTask **Change 5** name: Add SequenceFlow abstraction: Concrete composition: Atomic source: getPriorityServiceTechnician:ServiceTask target: assignPriorityServiceTechnicianSO:ServiceTask

	Change 6 name Add SequenceFlow abstraction Concrete composition Atomic source assignPriorityServiceTechnicianSO:ServcieTask target saveServiceOrder:ServcieTask
Change Scenario 3	
Maintenance Type	Corrective
Scenario Description	The operation createServiceDiagnosisReport requires a list of possibleSolutions of type Solution in the interface corresponding to Class ServiceReportHandler. Moreover, it is required to call this operation for the further processing of cancellation of the ServiceOrder during the ServiceExecutionProcess.
Required Changes	**Change 2** name Add Parameter abstraction Concrete composition Atomic source possibleSolutions:Parameter target createServiceDiagnsisReport:Operation **Change 1** name Add ServiceTask abstraction Concrete composition Atomic source createServiceDiagnosisReport ServiceTask target ServiceReportHandler:Participant **Change 3** name Add MessageFlow abstraction Concrete composition Atomic source Create Diagnosis Report :Task target createServiceDiagnsisReport :ServiceTask
Change Scenario 4	
Maintenance Type	Corrective
Scenario Description	The HandleTourPlanningProcess uses a service getRoutesForInterval, which is ot more available. Another service getRoutesForDuration returns the routes for a specified duration. Thus, the reuirement is to replace the ServiceTask with the other.
Required Changes	name Replace ServiceTask abstraction Concrete composition Composite source getRoutesForInterval:ServiceTask target getRoutesForDuration:ServiceTask
Change Scenario 5	
Maintenance Type	Perfective
Scenario Description	The Class ServiceOrderSchedular and ServicesSchedular both serve the same purpose of scheduling related tasks for Service Orders. Merge both classes for a better design.
Required Changes	name Merge Class abstraction Concrete composition Composite source Merge ServicesSchedular:Class target ServiceOrderSchedularClass:Class
Change Scenario 6	
Maintenance Type	Corrective
Scenario Description	In HandleBookAccomodationProcess a ServcieTechnician cannot Handle the payment of booked accommodations, and cannot access to the service *payAccomodationExpenses*. Instead, he can send the invoice to the PaymentAuthorizationOfficer's and request for a payment.
Required Changes	**Change 1** name Rename Task abstraction Concrete composition Atomic source Pay for Booking:Task **Change 2** name Replace ServiceTask abstraction Concrete composition Composite source payAccomodationExpenses:ServcieTask target requestForAccomodationBookingPayment:ServcieTask
Change Scenario 7	
Maintenance Type	Corrective
Scenario Description	ReturnInventoryForServiceOrderProcess requires calling a Service to verify the item conditions and any liabilities from the ServciePlanner component
Required Changes	**Change 1** name Add Participant in Process abstraction Concrete composition Atomic source ServicePlanner:Participant target ReturnInventoryForServiceOrderProcess:Process

	Change 2 name Add ServiceTask abstraction Concrete composition Atomic source verifyItemConditions:ServiceTask target ServicePlanner:Participant **Change 3** name Add MessageFlow abstraction Concrete composition Atomic source Verify Item Conditions:Task target verifyItemConditionsServiceTask
Change Scenario 8	
Maintenance Type	Perfective
Scenario Description	The components Recruitment and Recruiter are required to be merged as they serve the similar purpose.
Required Changes	name Merge Components abstraction Concrete composition Composite source Recruitement:Component target Recruiter:Component
Change Scenario 9	
Maintenance Type	Perfective
Scenario Description	Rename the Component ServcieCoordinatorNotifier to CoordinatorNotifier
Required Changes	name Rename Component abstraction Concrete composition Atomic source ServcieCoordinatorNotifier:Component target CoordinatorNotifier:Component
Change Scenario 10	
Maintenance Type	Perfective
Scenario Description	Rename ReturnInventoryForServiceOrderProcess to ReturnServcieOrderInventoryProcess
Required Changes	name Rename Process abstraction Concrete composition Atomic source ReturnInventoryForServiceOrderProcess:Process target ReturnServcieOrderInventoryProcess:Process

E

Rules

Dependency Detection Rules

Table E.1: Rules for Detecting Dependency Relations between Models and Tests.

Rule-ID	DDRule001
Description	creates links between participants and test components
Elements	e1:DocumentRoot, e2:Model, e3:Participant,e4:Class
Conditions	modelRelatedto(e2,Containment,e4) AND modelRelatedto(e1,Containment,e3) AND valueStartsWith(e4.name, e3.name) AND (valueEndsWith(e4.name,'TCom') OR valueEndsWith(e4.name,'TestComp'))AND ((modelRelatedto(e1,Tested_By,e2) OR modelRelatedto(e2,Tests,e1))
Actions	createLink(e3, Derivation, e4)
Rule-ID	DDRule002
Description	creates links between participants and corresponding components
Elements	e1:Participant, e2:Component
Conditions	valueEquals(e1.name, e2.name)
Actions	createLink(e1, Equivalence, e2)
Rule-ID	DDRule003
Description	creates links between participants and processes they initiate
Elements	e1:Participant, e2:Process
Conditions	valueNotNull(e2::name) AND modelEquals(e1::processRef, e2)
Actions	createLink(e1, Initiation, e2)
Rule-ID	DDRule004
Description	creates link between a Process and the ProcessClass defining the Process
Elements	e1:Process,e2:Class, e3:ProcessClass
Conditions	modelEquals(e3::base_Class, e2) AND valueEquals(e1::name, e2::name)
Actions	createLink(e1, Definition, e2)
Rule-ID	DDRule005
Description	creates links between components and their corresponding classes in CD
Elements	e1:Component, e2:Class
Conditions	valueEquals(e1::name,e2::name)
Actions	createLink(e1, Implementation, e2)
Rule-ID	DDRule006
Description	links BPMN collaboration root element and the corresponding TestModel
Elements	e1:DocumentRoot, e2:Model, e3:TestModel
Conditions	valueContains(e2::name, e1::name) AND valueEndsWith(e2::name, TestArchitecture) AND modelEquals(e3::base_Package, e2)
Actions	createLink(e2, Tests, e1), createLink (e1, Tested_By, e2)
Rule-ID	DDRule007
Description	creates link between a TestContext and TestPackage
Elements	e1:Class, e2:Package, e3:TestContext, e4:TestPackage
Conditions	modelEquals(e4::base_Package, e2) AND modelEquals(e3::base_StructuredClassifier, e1) AND modelDirectParentOf(e2, e1)
Actions	createLink(e2, Containment, e1)
Rule-ID	DDRule008
Description	creates link between a TestPackage and SUT
Elements	e1:Package, e2:Model, e3:TestPackage, e4:TestModel
Conditions	modelEquals(e3::base_Package, e1) AND modelEquals(e4::base_Package, e2) AND modelDirectParentOf(e2, e1)
Actions	createLink(e2, Containment, e1)
Rule-ID	DDRule009
Description	creates link between SUT and Test Model
Elements	e1:Model, e2:Class, e3:TestModel, e4:SUT
Conditions	modelEquals(e3::base_Package, e1) AND modelEquals(e4::base_Classifier, e2) AND modelDirectParentOf(e1, e2)
Actions	createLink(e1, Containment, e2)

Rule-ID	DDRule010
Description	creates link between TestPackage and SUT
Elements	e1:Model, e2:Package, e3:Class, e4:TestModel, e5:TestPackage, e6:SUT
Conditions	modelEquals(e4::base_Package, e1) AND modelEquals(e5::base_Package, e2) AND modelEquals(e6::base_Classifier, e3) AND modelDirectParentOf(e1, e2) AND modelDirectParentOf(e1, e3)
Actions	createLink(e3, Tests, e2)
Rule-ID	DDRule011
Description	creates links between TestPackage and TestComponent
Elements	e1:Package, e2:Class, e3:TestPackage, e4:TestComponent
Conditions	modelEquals(e3::base_Package, e1) AND modelEquals(e4::base_StructuredClassifier, e2) AND modelDirectParentOf(e1, e2)
Actions	createLink(e1, Containment, e2)
Rule-ID	DDRule012
Description	creates link between the TestContext and TestCase
Elements	e1:Class, e2:Operation, e3:TestContext
Conditions	modelEquals(e3::base_StructuredClassifier, e1) AND modelDirectParentOf(e1, e2)
Actions	createLink(e1, Containment, e2)
Rule-ID	DDRule013
Description	creates links between a TestContext and related TestComponents
Elements	e1:Class, e2:Package, e3:TestComponent, e4:Class, e5:TestContext
Conditions	modelEquals(e3::base_StructuredClassifier, e1) AND modelEquals(e5::base_StructuredClassifier, e4, Containment) AND modelRelatedTo(e2, e1) AND modelRelatedTo(e2, e4, Containment)
Actions	createLink(e4, Usage, e1)
Rule-ID	DDRule014
Description	Creates Link between a test behavior and its definition
Elements	e1:Class, e2:Operation, e3:TestContext, e4:Process, e5:Activity
Conditions	valueNotNull(e4::name) AND modelEquals(e3::base_StructuredClassifier, e1) AND valueContains(e1::name, e4::name) AND modelRelatedTo(e1, Containment, e2) AND valueContains(e5::name, e4::name) AND valueEquals(e5::name, e2::name)
Actions	createLink(e2, Definition, e5)
Rule-ID	DDRule015
Description	creates links between TestComponent and mock operation
Elements	e1:Class, e2:Operation, e3:TestComponent
Conditions	modelEquals(e3::base_StructuredClassifier, e1) AND modelDirectParentOf(e1, e2)
Actions	createLink(e1, Containment, e2)
Rule-ID	DDRule016
Description	A TestComponent is derived from a Lane
Elements	e1:Lane, e2:DocumentRoot,e3:Model, e4:Class, e5:TestComponent
Conditions	modelEquals(e5::base_StructuredClassifier, e4)AND modelRelatedTo(e2, Containment, e1), AND (modelRelatedTo(e3, Tests, e2) OR modelRelatedTo(e2, Tested_By, e3)) AND modelRelatedTo(e3, Containment, e4)
Actions	createLink(e1, Derivation, e4)
Rule-ID	DDRule017
Description	creates links between a process and its contained Lane
Elements	e1:Process, e2:Lane
Conditions	modelParentOf(e1,e2)
Actions	createLink(e1, Containment, e2)
Rule-ID	DDRule018
Description	creates link between a Lane and the corresponding class implementing its services
Elements	e1:Lane, e2:Class
Conditions	valueEquals(e1::name, e2::name)
Actions	createLink(e2, Definition, e1)
Rule-ID	DDRule019
Description	creates links between a TestModel and the Process it tests
Elements	e1:Process, e2:Model, e3:DocumentRoot, e4:Definitions
Conditions	modelDirectParentOf(e4, e1) AND modelDirectParentOf(e3, e4) AND valueNotNull(e1::name), AND (modelRelatedTo(e2, Tests, e3) OR modelRelatedTo(e3, Tests, e2))
Actions	createLink(e2, Tests, e1)
Rule-ID	DDRule020
Description	creates traceability links between a process and the test cases to test the process
Elements	e1:Process, e2:Activity
Conditions	valueEndsWith(e2::name, e1::name) AND valueStartsWith(e2::name, testCase)
Actions	createLink(e2, Tests, e1), createLink(e1, Tested_By, e2)
Rule-ID	DDRule021
Description	creates link between SUT and corresponding ProcessClass
Elements	e1:Class, e2:ProcessClass, e3:Class, e4:SUT
Conditions	modelEquals(e4::base_Classifier, e3) AND modelEqualse2::base_Class, e1) AND valueEquals(e1::name, e3::name)

Actions	createLink(e1, Definition, e3)
Rule-ID	DDRule022
Description	creates a traceability link between an operation of ProcessClass and an Operation of SUT
Elements	e1:Class, e2:ProcessClass, e3:Class, e4:SUT, pOP:Operation, sOP:Operation
Conditions	modelEquals(e2::base_Class, e1) AND modelEquals(e4::base_Classifier, e3) AND modelDirectParentOf(e1, pOP) AND modelDirectParentOf(e3,sOP) AND valueEquals(sOP::name, pOP::name)
Actions	createLink(sOP, Equivalence, pOP)
Rule-ID	DDRule023
Description	creates a link between a package and its sub packages
Elements	e1:Package, e2:Package
Conditions	modelDirectParentOf(e1, e2)
Actions	createLink(e1, Containment, e2)
Rule-ID	DDRule024
Description	creates link between Lane and its root element
Elements	e1:Lane, e2:DocumentRoot
Conditions	modelParentOf(e2, e1)
Actions	createLink(e2, Containment, e1)
Rule-ID	DDRule025
Description	creates links between TestComponent, TestContext and its TestModel
Elements	e1:Class, e2:Model, e3:TestModel
Conditions	modelEquals(e3::base_Package, e2) AND modelParentOf(e2, e1)
Actions	createLink(e2, Containment, e1)
Rule-ID	DDRule026
Description	creates links between lanes, ServiceTasks, and corresponding Operations
Elements	e1:Lane, e2:Class, e3:Operation, e4:Process, e5:ServiceTask
Conditions	modelRelatedTo(e2, Definition, e1) AND valueEquals(e5::name, e3::name) AND modelRelatedTo(e2, Containment, e3), modelRelatedTo(e4, Containment, e5), modelRelatedTo(e4, Containment, e1)
Actions	createLink(e3, Equivalence, e5), createLink(e1, Provision, e5)
Rule-ID	DDRule027
Description	creates links between classes/interfaces and their owned operations
Elements	e1:Class | Interface, e2:Operation
Conditions	modelDirectParentOf(e1, e2)
Actions	createLink(e1, Containment, e2)
Rule-ID	DDRule028
Description	creates Links between Processes and ServiceTasks
Elements	e1:Process, e2:ServiceTask
Conditions	modelDirectParentOf(e1, e2)
Actions	createLink(e1, Containment, e2)
Rule-ID	DDRule029
Description	creates links between process and its contained tasks
Elements	e1:Process, e2:Task
Conditions	modelDirectParentOf(e1, e2)
Actions	createLink(e1, Containment, e2)
Rule-ID	DDRule030
Description	creates traceability links between process and the send tasks conatined by it
Elements	e1:Process,e2:SendTask
Conditions	modelDirectParentOf(e1,e2)
Actions	createLink(e1, Containment, e2)
Rule-ID	DDRule031
Description	creates traceability links between process and the receive tasks conatined by it
Elements	e1:Process,e2:ReceiveTask
Conditions	modelDirectParentOf(e1,e2)
Actions	createLink(e1, Containment, e2)
Rule-ID	DDRule033
Description	creates link between StartEvent and Process
Elements	e1:Process,e2:StartEvent
Conditions	modelDirectParentOf(e1,e2)
Actions	createLink(e1, Containment, e2)
Rule-ID	DDRule033
Description	creates link between EndEvent and Process
Elements	e1:Process,e2:EndEvent
Conditions	modelDirectParentOf(e1,e2)
Actions	createLink(e1, Containment, e2)
Rule-ID	DDRule034
Description	creates link between Process and the SequenceFlows it contains
Elements	e1:Process,e2:SequenceFlow
Conditions	modelDirectParentOf(e1,e2)
Actions	createLink(e1, Containment, e2)
Rule-ID	DDRule035

Description	creates link between process and event based gateway
Elements	e1:Process,e2:EventBasedGateway
Conditions	modelDirectParentOf(e1,e2)
Actions	createLink(e1, Containment, e2)
Rule-ID	DDRule036
Description	creates link between process and ExclusiveGateway
Elements	e1:Process,e2:ExclusiveGateway
Conditions	modelDirectParentOf(e1,e2)
Actions	createLink(e1, Containment, e2)
Rule-ID	DDRule037
Description	creates a link between an operation and its contained parameters
Elements	e1:Operation, e2:Parameter
Conditions	modelDirectParentOf(e1, e2)
Actions	createLink(e1, Containment, e2)
Rule-ID	DDRule038
Description	creates link between an component interface and an interface in Class diagram
Elements	e1:Interface, e2:Interface
Conditions	valueEquals(e1::name, e2::name)
Actions	createLink(e1, Equivalence,e2)
Rule-ID	DDRule039
Description	creates a traceability link between a Particiant and a Collaboration
Elements	e1:Collaboration, e2:Participant
Conditions	modelDirectParentOf(e1, e2)
Actions	createLink(e1, Containment, e2)
Rule-ID	DDRule040
Description	creates a traceability link between component and its provided interface
Elements	e1:Interface, e2:Component; e3:Port
Conditions	modelEquals(e3::type, e1) AND modelDirectParentOf(e2, e3)
Actions	createLink(e2,Provision, e1)
Rule-ID	DDRule041
Description	creates a link between an interface and the class implementing the interface
Elements	e1:Interface, e2:Class
Conditions	valueStartsWith(e1::name,"I") AND valueEndsWith(e1::name, e2::name)
Actions	createLink(e1, Realization, e2)
Rule-ID	DDRule042
Description	creates a link beween activity and its contained Control Flows
Elements	e1:Activity, e2:ControlFlow
Conditions	modelDirectParentOf(e1, e2)
Actions	createLink(e1, Containment, e2)
Rule-ID	DDRule043
Description	creates a link between an activity and its contained activity partition
Elements	e1:Activity, e2:ActivityPartition
Conditions	modelDirectParentOf(e1, e2)
Actions	createLink(e1, Containment, e2)
Rule-ID	DDRule044
Description	creates a link between an activity and its contained Call operation actions
Elements	e1:Activity, e2:CallOperationAction
Conditions	modelDirectParentOf(e1, e2)
Actions	createLink(e1, Containment, e2)
Rule-ID	DDRule045
Description	creates links between an activity and its contained InitailNode
Elements	e1:Activity, e2:InitialNode
Conditions	modelDirectParentOf(e1, e2)
Actions	createLink(e1, Containment, e2)
Rule-ID	DDRule046
Description	creates links between an activity and its contained ActivityFinalNode
Elements	e1:Activity, e2:ActivityFinalNode
Conditions	modelDirectParentOf(e1, e2)
Actions	createLink(e1, Containment, e2)
Rule-ID	DDRule047
Description	creates a link between Package and its contained classes
Elements	e1:Package, e2:Class
Conditions	modelDirectParentOf(e1, e2)
Actions	createLink(e1, Containment, e2)
Rule-ID	DDRule048
Description	creates link between ActivityFinalNode and BPMN EndEvent
Elements	e1:ActivityFinalNode, e2:Activity, e3:Process, e4:EndEvent
Conditions	modelDirectParentOf(e3, e4) AND modelRelatedTo(e2, Tests, e3) AND modelDirectParentOf(e2, e1)
Actions	createLink(e4, Derivation, e1)

Rule-ID	DDRule049
Description	creates link between test InitialNode and BPMN StartEvent
Elements	e1:InitialNode, e2:Activity, e3:Process, e4:StartEvent
Conditions	modelDirectParentOf(e3, e4) AND modelRelatedTo(e2, Tests, e3) AND modelDirectParentOf(e2, e1)
Actions	createLink(e4, Derivation, e1)
Rule-ID	DDRule050
Description	creates a link between a Task and Service it calls
Elements	e1:Task, e2:MessageFlow, e3:ServiceTask
Conditions	modelEquals(e2::sourceRef, e1) AND modelEquals(e2::targetRef, e3)
Actions	createLink(e1, Calls, e3)
Rule-ID	DDRule051
Description	creates link between test CallOperationAction and BPMN Task
Elements	e1:CallOperationAction, e2:Activity, e3:Process, e4:Task
Conditions	modelDirectParentOf(e3, e4) AND modelRelatedTo(e2, Tests, e3) AND modelDirectParentOf(e2, e1)
Actions	createLink(e4, Derivation, e1)
Rule-ID	DDRule052
Description	creates link between OpaqueAction and Task
Elements	e1:OpaqueAction, e2:Activity, e3:Process, e4:Task
Conditions	modelDirectParentOf(e3, e4) AND valueEquals(e1::name,e4::name) AND modelRelatedTo(e2, Tests, e3) AND modelDirectParentOf(e2, e1)
Actions	createLink(e4, Derivation, e1)
Rule-ID	DDRule053
Description	creates a link between a ProcessClass and a TestModel
Elements	e1:Class, e2:Model, e3:Process
Conditions	modelRelatedTo(e2, Tests, e3) AND modelRelatedTo(e3, Definition, e1)
Actions	createLink(e2, Tests, e1)
Rule-ID	DDRule054
Description	Mock operation is derived from a ServiceTask
Elements	e1:Operation, e2:Class, e3:ServiceTask, e4:Lane
Conditions	valueEndsWith(e1::name, "Mock") AND valueStartsWith(e1::name, e3::name) AND modelDirectParentOf(e2, e1) (modelRelatedTo(e4, Provision, e3) OR modelRelatedTo(e3,Provision, e4)) AND modelRelatedTo(e4, Derivation, e2)
Actions	createLink(e3, Mocks, e1)
Rule-ID	DDRule055
Description	creates links between the Participant and its corresponding implementation Class
Elements	e1:Participant,e2:Class,e3: Component
Conditions	modelRelatedTo(e2, Implementation,e3) AND (modelRelatedTo(e1, Equivalence,e3) OR modelRelatedTo(e3, Equivalence,e1))
Actions	createLink(e2, Implementation, e1)
Rule-ID	DDRule057
Description	creates link between a TestArchitectureModel and TestDataModel
Elements	e1:Model, e2:Model, e3:Process
Conditions	valueStartsWith(e2::name, e3::name) AND valueEndsWith(e2::name, "TestData") AND modelRelatedTo(e1,Tests, e3)
Actions	createLink(e1, Usage, e2)
Rule-ID	DDRule058
Description	creates links between a TestCase and TestDataModel
Elements	e1:Activity, e2:Model, e3:Process
Conditions	valueStartsWith(e2::name, e1::name) AND valueEndsWith(e2::name, "TestData") AND modelRelatedTo(e1,Tests, e3)
Actions	createLink(e1, Usage, e2)
Rule-ID	DDRule059
Description	creates link between a DataPool and the Class it specializes
Elements	e1:Class, e2:Class, e3:DataPool
Conditions	modelEquals(e3::base_Classifier, e1) AND valueStartsWith(e1::name,e2::name)
Actions	createLink(e1, Specializes, e2)
Rule-ID	DDRule060
Description	creates link between a class and its super class
Elements	e1:Class, e2:Class, e3:Generalization
Conditions	modelDirectParentOf(e1, e3) AND modelEquals(e3::general, e2)
Actions	createLink(e1, Extends, e2)
Rule-ID	DDRule061
Description	creates links between lanes, SendTasks, and corresponding Operations
Elements	e1:Lane, e2:Class, e3:Operation, e2:Process, e5:SendTask
Conditions	modelRelatedTo(e2, Definition,e1) AND valueEquals(e5.name,e3.name) AND modelRelatedTo(e2, Containment,e3) AND modelRelatedTo(e4, Containment,e5) AND modelRelatedTo(e4, Containment,e1)
Actions	createLink(e3,Equivalence , e5), createLink(e1, Communicates, e5)
Rule-ID	DDRule062
Description	Mock operation is derived from a SendTask

Elements	e1:Operation, e2:Class, e3:SendTask, e4:Lane | Participant
Conditions	valueEndsWith(e1.name, 'Mock') AND valueStartsWith(e1.name, e3.name) modelRelatedTo(e2, Containment,e1) AND (modelRelatedTo(e4, Communicate,e3) OR modelRelatedTo(e3, Communicate,e4)) AND (modelRelatedTo(e4, Derivation,e2) OR modelRelatedTo(e2, Derivation,e4))
Actions	createLink(e1, Mocks, e3)
Rule-ID	DDRule063
Description	creates links between lanes, ReceiveTasks, and corresponding Operations
Elements	e1:Lane
Elements	e2:Class
Elements	e3:Operation
Elements	e4:Process
Elements	e5:ReceiveTask
Conditions	modelRelatedTo(e3, Equivalence,e5) AND modelRelatedTo(e1, Communicate,e5) AND modelRelatedTo(e2, Definition,e1)
Actions	createLink(e1, , e2)
Rule-ID	DDRule064
Description	Mock operation is derived from a ReceiveTask
Elements	e1:Operation, e2:Class, e3:ReceiveTask, e4:Lane | Participant
Conditions	valueEndsWith(e1::name, Mock) AND valueStartsWith(e1::name, e3::name) AND modelRelatedTo(e2, Containment,e1) AND modelRelatedTo(e4, Communicate,e3) AND modelRelatedTo(e4, Derivation,e2)
Actions	createLink(e1, Mocks, e3)
Rule-ID	DDRule065
Description	Creates Link between a Participant and its provided services
Elements	e1:Participant, e2:Process, e3:ServiceTask, e4:DocumentRoot
Conditions	modelRelatedTo(e2, Initiation,e1) AND modelRelatedTo(e4, Containment,e1) AND modelRelatedTo(e4, Containment,e2) AND modelRelatedTo(e2, Containment,e3)
Actions	createLink(e1, Provision, e3)
Rule-ID	DDRule066
Description	Between the services of participants and their implementation
Elements	e1:Participant, e2:ServiceTask, e3:Class, e4:Operation, e5:Interface
Conditions	modelRelatedTo(e1, Provision,e2) AND modelRelatedTo(e3, Implementation,e1) AND valueEquals(e2::name, e4::name) AND modelRelatedTo(e5, Realization,e3) AND (modelRelatedTo(e3, Containment,e4) OR modelRelatedTo(e5, Containment,e4))
Actions	createLink(e2, Equivalence, e4)
Rule-ID	DDRule67
Description	creates link between a Mock operation and a ServiceTask provided by a participant
Elements	e1:Participant, e2:ServiceTask, e3:Class, e4:Operation
Conditions	modelRelatedTo(e1, Provision,e2) AND modelDirectParentOf"(e3, e4) AND valueEndsWith(e4::name,"Mock") AND modelRelatedTo(e3, Derivation,e1)
Actions	createLink(e4, Mocks, e2)
Rule-ID	DDRule068
Description	Creates dependency relation between Participant and DocumentRoot
Elements	e1:Participant, e2:DocumentRoot
Conditions	modelParentOf(e2, e1)
Actions	createLink(e2, Containment, e1)
Rule-ID	DDRule069
Description	Creates link between element process and documentroot
Elements	e1:Process, e2:DocumentRoot
Conditions	modelParentOf(e2, e1)
Actions	createLink(e2, Containment, e1)
Rule-ID	DDRule070
Description	Creates Links between class corresponding to lane or participant and test component
Elements	e1:Lane | Participant, e2:Class, e3:Class
Conditions	modelRelatedTo(e1, Derivation,e3) AND (modelRelatedTo(e2, Implementation,e1) OR modelRelatedTo(e2, Definition,e1)
Actions	createLink(e3, Mocks, e2)
Rule-ID	DDRule071
Description	creates a link between a process and SUT
Elements	e1:Class, e2:Class, e3:Process
Conditions	(modelRelatedTo(e2, Definition,e1) OR modelRelatedTo(e1, Definition,e2)) AND (modelRelatedTo(e3, Definition,e1) OR modelRelatedTo(e1, Definition,e3))
Actions	createLink(e3, Derivation, e2)
Rule-ID	DDRule072
Description	creates links between Lanes and Tasks
Elements	e1:Lane, e2:Task
Conditions	(modelEquals(e2::lane,e1))
Actions	createLink(e1, Executes, e2)
Rule-ID	DDRule073
Description	creates link between Lane and ServiceTasks

Elements	e1:Lane, e2:ServiceTask
Conditions	(modelEquals(e2::lane, e1))
Actions	createLink(e1, Provision, e2)
Rule-ID	DDRule074
Description	creates links between SendTask, ReceiveTask and Lane
Elements	e1:Lane, e2:ReceiveTask | SendTask
Conditions	(modelEquals(e2::lane, e1))
Actions	createLink(e1, Communicate, e2)
Rule-ID	DDRule075
Description	create links between a task and mock operation
Elements	e1:Process, e2:EndEvent,e3:Lane | Participant,e4:Task,e5:Class,e6:Operation
Conditions	(valueStartsWith(e6::name, e4::name) AND (NOT(modelRelatedTo(e1,Containment,e2)) AND (modelRelatedTo(e5,Containment,e6) AND (modelRelatedTo(e3,Derivation,e5) AND (modelRelatedTo(e1,Containment,e4) AND (modelRelatedTo(e3,Initiation,e1) OR (modelRelatedTo(e1,Containment,e3))))
Actions	createLink(e6, Mocks, e4)
Rule-ID	DDRule077
Description	creates links bertween operatons of interfaces and mockoperatios
Elements	e1:Class, e2:Interface, e3:Operation, e4:Class, e5:Operation
Conditions	(modelRelatedTo(e2, Realization,e1) AND modelRelatedTo(e2, Containment, e3) AND modelRelatedTo(e4, Mocks, e1) AND modelRelatedTo(e4, Containment, e5) AND valueStartsWith(e5::name,e3::name) AND ValueEndsWith(e5::name,Mock))
Actions	createLink(e5, Mocks, e3)
Rule-ID	DDRule076
Description	creates link between operations of a class and corresponding test componenet
Elements	e1:Class, e2:Class, e3:Operation, e4:Operation
Conditions	(modelRelatedTo(e1, Mocks,e2) AND modelRelatedTo(e1, Containment, e3) AND modelRelatedTo(e2, Containment, e4) AND valueStartsWith(e3::name, e4::name))
Actions	createLink(e3, Mocks, e4)
Rule-ID	DDRule078
Description	creates links between a test model and its java implemenattaion
Elements	e1:Model, e2:Package
Conditions	(valueStartsWith(e1::name,TestArchitecture) AND (e1::name, e2::name))
Actions	createLink(e2, Implementation, e1)
Rule-ID	DDRule079
Description	creates link between a test package and its corresponding java implementation
Elements	e1:Package, e2:Package, e3:TestPackage
Conditions	(modelEquals(e3::base_Package,e1) AND (e1::name, e2::name))
Actions	createLink(e1, Implementation, e2)
Rule-ID	DDRule080
Description	creates links between java package and implementation classes
Elements	e1:Package, e2:ClassDeclaration
Conditions	(modelDirectParentOf(e1,e2))
Actions	createLink(e1, Containment, e2)
Rule-ID	DDRule081
Description	creates link between java classes and methods
Elements	e1:ClassDeclaration, e2:MethodDeclaration
Conditions	(modelDirectParentOf(e1,e2))
Actions	createLink(e1, Containment, e2)
Rule-ID	DDRule082
Description	creates link between SUT and corresponding java implemenatation
Elements	e1:Class, e2:Model, e3:ClassDeclaration,e4:Package,e5:SUT
Conditions	(ModelEquals(e5::base_Classifier,e1) AND ModelRelatedTo(e2,Containment,e1) AND ModelRelatedTo(e4,Containment,e3) AND ValueStartsWith(e3::name,e1::name)) AND (ModelRelatedTo(e2, Implementation, e4) OR ModelRelatedTo(e4, Implementation, e2))
Actions	createLink(e3, Implementation, e4)
Rule-ID	DDRule083
Description	creates links between TestContext, TestComponents, and its java implementation
Elements	e1:Class, e2:Package, e3:ClassDeclaration,e4:Package,e5:TestContext | TestComponent
Conditions	(ModelEquals(e5::base_StructuredClassifier,e1) AND ModelRelatedTo(e2,Containment,e1) AND ModelRelatedTo(e4,Containment,e3) AND (e1::name,e3::name)) AND (ModelRelatedTo(e2, Implementation, e4) OR ModelRelatedTo(e4, Implementation, e2))
Actions	createLink(e3, Implementation, e1)
Rule-ID	DDRule084
Description	creates link between MockOperation and corresponding java methods and SUT operations
Elements	e1:Class, e2:ClassDeclaration, e3:Operation,e4:MethodDeclaration
Conditions	(e4::name,e3::name) AND (ModelRelatedTo(e1,Containment,e3) AND ModelRelatedTo(e2,Containment,e4)) AND (ModelRelatedTo(e2, Implementation, e1))
Actions	createLink(e4, Implementation, e3)

Rule-ID	DDRule085
Description	creates link between receive tasks and send signal action
Elements	e1:ReceiveTask, e2:AcceptEventAction, e3:Activity,e4:Process
Conditions	(ModelRelatedTo(e3,Tests,e4) OR ModelRelatedTo(e4,Tested_By,e3)) AND (ModelRelatedTo(e4,Containment,e1) AND ModelRelatedTo(e3,Containment,e2))
Actions	createLink(e1, Derivation, e2)
Rule-ID	DDRule086
Description	creates links between sequence flows and control flows (startevent and task)
Elements	e1:SequenceFlow, srcSF:StartEvent, dstSF:Task, e2:ControlFlow, srcA:InitialNode, dstA:OpaqueAction
Conditions	(ModelEquals(e1::sourceRef,srcSF) AND ModelEquals(e1::targetRef,dstSF)) AND (ModelEquals(e2::source,srcA) AND ModelEquals(e2::target,dstA)) AND (ModelRelatedTo(srcSF, Derivation, srcA) OR ModelRelatedTo(srcA, Derivation, srcSF)) AND (ModelRelatedTo(dstSF, Derivation, dstA) OR ModelRelatedTo(dstA, Derivation, dstSF))
Actions	createLink(e1, Derivation, e2)
Rule-ID	DDRule087
Description	creates links between sequence flows and control flows
Elements	e1:SequenceFlow, srcSF:ServiceTask, dstSF:Task, e2:ControlFlow, srcA:CallOperationAction, dstA:OpaqueAction
Conditions	(ModelEquals(e1::sourceRef,srcSF) AND ModelEquals(e1::targetRef,dstSF)) AND (ModelEquals(e2::source,srcA) AND ModelEquals(e2::target,dstA)) AND (ModelRelatedTo(srcSF, Derivation, srcA) OR ModelRelatedTo(srcA, Derivation, srcSF)) AND (ModelRelatedTo(dstSF, Derivation, dstA) OR ModelRelatedTo(dstA, Derivation, dstSF))
Actions	createLink(e1, Derivation, e2)
Rule-ID	DDRule088
Description	creates links between sequence flows and control flows
Elements	e1:Activity, e2:Process, e3:AcceptEventAction, e4:ReceiveTask, srcA:CallOperationAction, dstA:OpaqueAction
Conditions	(e3::name,e4::name) AND (ModelDirectParentOf(e1,e3) AND ModelRelatedTo(e2, Containment, e4)) AND (ModelRelatedTo(e2,Tests,e1) OR ModelRelatedTo(e1,Tests,e2))
Actions	createLink(e3, Derivation, e4)
Rule-ID	DDRule089
Description	controlflow and sequenceflow
Elements	e1:SequenceFlow, e2:ControlFlow, srcSF:ServiceTask, dstSF:ReceiveTask, srcA:CallOperationAction, dstA:AcceptEventAction
Conditions	(ModelEquals(e1::sourceRef,srcSF) AND ModelEquals(e1::targetRef,dstSF)) AND (ModelEquals(e2::source,srcA) AND ModelEquals(e2::target,dstA)) AND (ModelRelatedTo(srcSF, Derivation, srcA) OR ModelRelatedTo(srcA, Derivation, srcSF)) AND (ModelRelatedTo(dstSF, Derivation, dstA) OR ModelRelatedTo(dstA, Derivation, dstSF))
Actions	createLink(e1, Derivation, e2)
Rule-ID	DDRule091
Description	ControlFow and SequenceFlow ServiceTask CallOperattionAction
Elements	e1:SequenceFlow, e2:ControlFlow, srcSF:ServiceTask, dstSF:ServiceTask, srcA:CallOperationAction, dstA:CallOperationAction
Conditions	(ModelEquals(srcSF::outgoing,e1) AND ModelEquals(dstSF::incoming,e1)) AND (ModelEquals(srcA::outgoing, e2) AND ModelEquals(dstA::incoming, e2)) AND (ModelRelatedTo(srcSF, Derivation, srcA) AND ModelRelatedTo(dstSF, Derivation, dstA))
Actions	createLink(e1, Derivation, e2)
Rule-ID	DDRule092
Description	creates links between Participants and their Lanes
Elements	e1:Process, e2:Participant, e3:Lane
Conditions	(ModelRelatedTo(e2, Initiation, e1) AND ModelRelatedTo(e1, Containment, e3))
Actions	createLink(e2, Containment, e3)
Rule-ID	DDRule093
Description	cretaes a link between the callOperationAction and the MockOperations they call
Elements	e1:CallOperationAction | SendSignalAction, e2:Task | SendTask, e3:ServiceTask, e4:Operation
Conditions	(ModelRelatedTo(e4, Mocks, e3) AND ModelRelatedTo(e2, Calls, e3) AND ModelRelatedTo(e2, Derivation, e1))
Actions	createLink(e1, Calls, e4)
Rule-ID	DDRule094
Description	creates Link between SendSignalAction and Send Task
Elements	e1:SendTask, e2:Process, e3:SendSignalAction, e4:Activity
Conditions	(ModelRelatedTo(e4, Containment, e3) AND (e1::name, e3::name)) AND (ModelDirectParentOf(e2, e1) AND ModelRelatedTo(e4, Tests, e2))
Actions	createLink(e1, Derivation, e3)
Rule-ID	DDRule095
Description	Creates Link between Sequence Flow emerging from Decision Node to Taks and corresponding ControlFlow
Elements	e1:SequenceFlow, e2:Task, e3:ControlFlow, e4:ConditionalNode, e5:CallOperationAction

Conditions	((e1::name,e4::name)) AND (ModelEquals(e3::source, e4) AND ModelEquals(e3::source, e5)) AND (ModelEquals(e1::targetRef, e2) AND ModelRelatedTo(e2, Derivation, e5))
Actions	createLink(e1, Derivation, e3) AND createLink(e1, Derivation, e4)

Impact Rules

Table E.2: Impact Rules between Models and Tests.

Rule	IR001
Description	Deleting a ProcessClass affects corresponding TestModel
Elements	e1:Class, e2:Model, e3:TestComponent
Change Type	Delete ProcessClass
Conditions	(ModelRelatedTo(e2,Tests,e1) OR ModelRelatedTo(e1,Tests,e2))
Actions	reportImpact(e1, Delete ProcessClass, e2)
Rule	IR002
Description	Deleting an operation deletes corresponding ServiceTask
Elements	e1:ServiceTask, e2:Operation
Change Type	Delete Operation
Conditions	(ModelRelatedTo(e2,Equivalence,e1) OR ModelRelatedTo(e1,Equivalence,e2))
Actions	reportImpact(e2, Delete ServiceTask, e1)
Rule	IR003
Description	Deleting a ServiceTask deletes corresponding MockOperation
Elements	e1:Process, e2:Class
Change Type	Delete ServiceTask
Conditions	(ModelRelatedTo(e2,Mocks,e1))
Actions	reportImpact(e1, Delete ServiceTask, e2)
Rule	IR004
Description	Deleting a Process affects corresponding ProcessClass
Elements	e1:Participant, e2:Class, e3:TestComponent
Change Type	Delete Process
Conditions	(ModelRelatedTo(e2,Definition,e1) OR ModelRelatedTo(e1,Definition,e2))
Actions	reportImpact(e1, Delete Participant, e2)
Rule	IR005
Description	Deleting a participant impacts test components
Elements	e1: Class, e2:Model, e3:TestComponent
Change Type	Delete Participant
Conditions	(ModelRelatedTo(e2,Derivation,e1) OR ModelRelatedTo(e1,Derivation,e2))
Actions	reportImpact(e1, Delete ProcessClass, e2)
Rule	IR006
Description	Adding an operation in class requires adding corresponding mockoperations in testcomponents
Elements	e1:Class, e2:Operation, e3:Class
Change Type	Add Operation
Conditions	(ModelRelatedTo(e3,Mocks,e1) OR ModelRelatedTo(e1,Tests,e2))
Actions	reportImpact(e1, Add MockOperation,e2, e3|e2)
Rule	IR007
Description	Replacing a ServiceTask requires replacement of corresponding CallOperationAction in tests
Elements	e1:ServiceTask, e2:ServiceTask, e3:CallOperationAction
Change Type	Replace ServiceTask (Composite)
Conditions	(ModelRelatedTo(e1,Derivation,e3))
Actions	reportImpact(e1, Add SUTOperation, e2, e3|e2)
Rule	IR008
Description	Adding an operation in ProcessClass requires a corresponding operation in SUT
Elements	e1:Class,e2:Operation, e3:Class
Change Type	Add Operation
Conditions	(ModelRelatedTo(e1,Definition,e3))
Actions	reportImpact(e1, Delete ProcessClass, e2)
Rule	IR009
Description	Adding a parameter in an operation requires adding a parameter in the corresponding mockoperation
Elements	e1:Operation, e2:Parameter, e3:Operation
Change Type	Add Parameter
Conditions	(ModelRelatedTo(e3,Mocks,e1))
Actions	reportImpact(e1, Add Parameter in MockOperation, e2, e3|e2)

Rule	IR010
Description	Replace a Service Task with another ServiceTask affects corresponding mockoperation
Elements	e1:ServiceTask, e2:Operation, e3:ServiceTask,e4:Operation
Change Type	Replace ServiceTask (Composite)
Conditions	(ModelRelatedTo(e2,Mocks,e1) OR ModelRelatedTo(e4,Mocks,e3))
Actions	reportImpact(e1, Replace MockOperation, e3,e2 | e4)
Rule	IR011
Description	Adding a MockOperation in test component requires corresponding method in implementation class
Elements	e1:Operation, e2:Class, e3:ClassDeclaration
Change Type	Add MockOperation
Conditions	(ModelRelatedTo(e3,Implementation,e2))
Actions	reportImpact(e2, Add MOImplementation, e1,e3 | e1)
Rule	IR012
Description	Adding an operation in SUT requires corresponding implementation method in test code
Elements	e1:Operation, e2:Class, e3:ClassDeclaration
Change Type	Add SUTOperation
Conditions	(ModelRelatedTo(e3,Implementation,e2))
Actions	reportImpact(e2, Add SUTOperationImplementation, e1, e3 | e1)
Rule	IR013
Description	Adding a parameter in MockOperation requires adding parameter in corresponding implementation
Elements	e1:Operation, e2:Parameter, e3:MethodDeclaration
Change Type	Add Parameter in MockOperation
Conditions	(ModelRelatedTo(e3,Implementation,e1))
Actions	reportImpact(e1, Add Parameter in MOImplementation, e2, e3 | e2)
Rule	IR014
Description	Adding an parameter in an operation requires update in corresponding SUT Operation
Elements	e1:Operation, e2:Parameter, e3:Operation
Change Type	Add Parameter
Conditions	(ModelRelatedTo(e3,Equivalence,e1))
Actions	reportImpact(e1, Add Parameter in SUTOperation, e2, e3 | e2)
Rule	IR015
Description	Adding Parameter in SUTOperation requires update in its implementation
Elements	e1:Operation, e2:Parameter, e3:MethodDeclaration
Change Type	Add Parameter in SUTOperation
Conditions	(ModelRelatedTo(e3,Implementation,e1))
Actions	reportImpact(e1, Add Parameter in SUTImplementation, e2, e3 | e2)
Rule	IR016
Description	Adding a ServiceTask in a Process requires adding a corresponding MockOperation
Elements	e1:Process, e2:Lane, e3:Class, e4:ServiceTask
Change Type	Add ServiceTask
Conditions	(ModelRelatedTo(e2,Derivation,e3) AND ModelRelatedTo(e1,Containment,e2))
Actions	reportImpact(e1, Add MockOperation, e4, e3)
Rule	IR017
Description	Adding a Parameter requires adding corresponding data pools or data partitions
Elements	e1:Operation, e2:Parameter, e3:Model, e4:Model
Change Type	Add Parameter
Conditions	ModelRelatedTo(e3,Usage,e4)) AND (modelParentOf(e3,e1))
Actions	reportImpact(e1, Add DataPool, e2, e4)
Rule	IR018
Description	Adding a ServiceTask requires corresponding Operation in class
Elements	e1:Process, e2:ServiceTask, e3:Class
Change Type	Add ServiceTask
Conditions	(ModelRelatedTo(e1,Definition,e3) AND ModelRelatedTo(e1,Tests,e2))
Actions	reportImpact(e1, Add Operation, e2,e3 | e2)
Rule	IR019
Description	Adding a ServiceTask requires corresponding operation in SUT
Elements	e1:Process, e2:Class, e3:ServiceTask
Change Type	Add ServiceTask
Conditions	(ModelRelatedTo(e1,Derivation,e2) AND ModelRelatedTo(e1,Tests,e2))
Actions	reportImpact(e1, Add SUTOperation, e3, e2 | e3)
Rule	IR020
Description	Deleting a SequenceFlow requires deleting the corresponding ControlFlow in the test case
Elements	e1:SequenceFlow, e2:ControlFlow, e3:TestComponent
Change Type	Delete SequenceFlow
Conditions	ModelRelatedTo(e2,Derivation,e1)
Actions	reportImpact(e1, Delete ControlFlow, e2)
Rule	IR021
Description	Adding a sequence flow in the process requires adding sequence flow to the corresponding test cases
Elements	e1:ServiceTask, e2:ServiceTask, e3:CallOperationAction, e4:CallOperationAction, e6:Activity

Change Type	Add SequenceFlow
Conditions	(ModelRelatedTo(e2,Derivation,e1) AND modelDirectParentOf(e6, e3) AND modelDirectParentOf(e6, e4))
Actions	reportImpact(e1, Add ControlFlow, e2, e3 | e4)
Rule	IR022
Description	Adding a ServiceTask in a process of Participant requires corressponding MockOperation in the test component
Elements	e1:Process, e2:ServiceTask, e3:Participant, e4:Class
Change Type	Add ServiceTask
Conditions	(ModelRelatedTo(e3,Initiation,e1) AND ModelRelatedTo(e3,Derivation,e24))
Actions	reportImpact(e1, Add MockOperation, e2, e4 | e2)
Rule	IR023
Description	Adding a MessageFlow between a Task and Service Task requires replacement in corresponding test
Elements	e1:Task, e2:ServiceTask, e3:OpaqueAction | CallOperationAction
Change Type	Add MessageFlow
Conditions	ModelRelatedTo(e1,Derivation,e3)
Actions	reportImpact(e1, Replace OpaqueAction, e2, e3 | e2)
Rule	IR024
Description	Replacing a ServiceTask requires replacing mockOperation (if new operation is missing)
Elements	e1:ServiceTask, e2:ServiceTask, e3:Operation
Change Type	Replace ServiceTask
Conditions	ModelRelatedTo(e3,Mocks,e1)
Actions	reportImpact(e1, Replace MockOperation, e2, e2 | e3)
Rule	IR025
Description	Merging two classes require merging corresponding participants in a Process
Elements	e1:Class, e2:Class, e3:Participant, e4:Participant,e5:Process
Change Type	Merge Classes
Conditions	(ModelRelatedTo(e3,Initiation,e5) AND ModelRelatedTo(e4,Initiation,e5)AND ModelRelatedTo(e1,Implementation,e3)AND ModelRelatedTo(e2,Implementation,e4))
Actions	reportImpact(e1, Merge Participants, e2, e3 | e4)
Rule	IR026
Description	Merging Of participants requires merging of corresponding Lanes
Elements	e1:Participant, e2:Participant, e3:Lane, e4:Lane, e6:Process
Change Type	Merge Participants
Conditions	(ModelRelatedTo(e1,Containment,e3) AND ModelRelatedTo(e2,Containment,e4) AND ModelRelatedTo(e1,Initiation,e6) AND ModelRelatedTo(e2,Initiation,e6))
Actions	reportImpact(e1, Merge Lanes, e2,e3 | e4)
Rule	IR027
Description	Merging two classes result in rename of Participnat (if both are not in a Process)
Elements	e1:Class, e2:Class, e3:Participant, e5:Participant
Change Type	Merge Classes
Conditions	(ModelRelatedTo(e2,Implementation,e3) AND NOT(ModelRelatedTo(e1,Implementation,e5))
Actions	reportImpact(e1, Rename Participant, e2, e3)
Rule	IR028
Description	Renaming a Participant requires renaming a TestComponent
Elements	e1:Participant, e2:Class, e3:TestComponent
Change Type	Rename Participant
Conditions	ModelRelatedTo(e2,Derivation,e1) A)
Actions	reportImpact(e1, Rename TestComponent, e2)
Rule	IR029
Description	Renaming a Task Renames corresponding CallOperationAction in TestCase
Elements	e1:Task, e2:CallOperationAction
Change Type	Rename Task
Conditions	ModelRelatedTo(e1,Derivation,e2)
Actions	reportImpact(e1, Rename CallOperationAction, e2)
Rule	IR030
Description	Renaming a task requires renaming corresponding OpaqueAction in TestCase
Elements	e1:Task, e2:OpaqueAction, e3:TestComponent
Change Type	Rename Task
Conditions	ModelRelatedTo(e1,Derivation,e2)
Actions	reportImpact(e1, Rename OpaqueAction, e2)
Rule	IR031
Description	Adding a Participant in a Process requires corresponding test component in the test architecture
Elements	e1:Process, e2:Participant, e3:Model
Change Type	Add Participant
Conditions	modelRelatedTo(e3,Tests,e1)
Actions	reportImpact(e1, Add TestComponent'', e2,e3 | e2)
Rule	IR032
Description	Adding a test component in a model requires test component code

Elements	e1:Model, e2:Class, e3:Package
Change Type	Add TestComponent
Conditions	ModelRelatedTo(e3,Implementation,e1)
Actions	reportImpact(e1, Add TestComponentImplementation, e2,e3 | e2)
Rule	IR033
Description	Deleting a component requires deleting corresponding Participant
Elements	e1:Component, e2:Participant
Change Type	Delete Component
Conditions	(ModelRelatedTo(e1, Equivalence, e2) OR ModelRelatedTo(e2, Equivalence, e1))
Actions	reportImpact(e1, Delete Participant, e2)
Rule	IR034
Description	Renaming a component requires renaming the corresponding Participant
Elements	e1:Component, e2:Participant
Change Type	Rename Component
Conditions	(ModelRelatedTo(e1, Equivalence, e2) OR ModelRelatedTo(e2, Equivalence, e1))
Actions	reportImpact(e1, Rename Participant, e2)
Rule	IR035
Description	Merging a Component requires merging corresponding Participants
Elements	e1:Component, e2:Component, e3:Participant, e4:Participant, Collaboration
Change Type	Merge Components
Conditions	(ModelRelatedTo(e1, Equivalence, e3) OR ModelRelatedTo(e3, Equivalence, e1)) AND (ModelRelatedTo(e2, Equivalence, e4) OR ModelRelatedTo(e4, Equivalence, e2)) AND (ModelRelatedTo(e5, Containment, e3) AND ModelRelatedTo(e5, Equivalence, e4))
Actions	reportImpact(e1, Rename Participant, e2)
Rule	IR036
Description	Merging of two lanes require moving corresponding ServiceTasks to the merged Lane.
Elements	e1:Lane, e2:Lane, e3:ServiceTask
Change Type	Merge Lanes
Conditions	ModelRelatedTo(e2, Provision, e3)
Actions	reportImpact(e1, Move ServiceTask, e2, e3 | e1)
Rule	IR037
Description	Merging two Participants requires merging corresponding TestCmponents
Elements	e1:Participant, e2:Participant, e3:Class, e4:Class, e5:Package
Change Type	Merge Participants
Conditions	(ModelRelatedTo(e1, Derivation, e3) AND ModelRelatedTo(e2, Derivation, e4)) AND (ModelRelatedTo(e5, Containment, e3) AND ModelRelatedTo(e5, Containment, e4))
Actions	reportImpact(e1, Merge TestComponents, e2, e3 | e4)
Rule	IR038
Description	Merging two Lanes requires merging corresponding TestComponents
Elements	e1:Lane, e2:Lane, e3:Class, e4:Class, e5:Package
Change Type	Merge Lanes
Conditions	(ModelRelatedTo(e1, Derivation, e3) AND ModelRelatedTo(e2, Derivation, e4)) AND (ModelRelatedTo(e5, Containment, e3) AND ModelRelatedTo(e5, Containment, e4))
Actions	reportImpact(e1, Merge TestComponents, e2, e3 | e4)
Rule	IR039
Description	Merging Two Lanes requires renaming source component (if source have no TC)
Elements	e1:Lane, e2:Lane, e3:Class, e4:Class, e5:Package
Change Type	Merge Lanes
Conditions	(ModelRelatedTo(e5, Containment, e4) AND ModelRelatedTo(e2, Derivation, e4)) AND (ModelRelatedTo(e1, Derivation, e3) NOT ModelRelatedTo(e5, Containment, e3))
Actions	reportImpact(e1, Rename TestComponent, e2, e4)
Rule	IR040
Description	Moving a ServiceTask to another Lane requires moving the corresponding MockOperation
Elements	e1:ServiceTask, e2:Lane, e3:Operation, e4:Class, e5:Package
Change Type	Move ServiceTask
Conditions	(ModelRelatedTo(e3, Mocks, e1) AND ModelRelatedTo(e2, Derivation, e4)) AND ModelRelatedTo(e4, Containment, e3)
Actions	reportImpact(e1, Move MockOperation, e2, e3 | e4)
Rule	IR041
Description	Renaming a TestComponent requires renaming the TCImplementation
Elements	e1:Class, e2:ClassDeclaration
Change Type	Rename TestComponent
Conditions	ModelRelatedTo(e2, Implementation, e1)
Actions	reportImpact(e1, Rename TestComponentImplementation, e2)
Rule	IR042
Description	Renaming a Process requires renaming its corresponding test model
Elements	e1:Process, e2:Model
Change Type	Rename Process
Conditions	ModelRelatedTo(e2, Tests, e1)

Actions	reportImpact(e1, Rename TestModel, , e2)
Rule	IR043
Description	Renaming a TestModel requires renaming corresponding implemntation package
Elements	e1:Model,e2:Package
Change Type	Rename TestModel
Conditions	ModelRelatedTo(e2, Implementation, e1)
Actions	reportImpact(e1, TestPackageImplementation, , e2)
Rule	IR044
Description	Renaming a Process requires renamindg corresponding SUT
Elements	e1:Process,e2:Class,e3:SUT, e4:Model
Change Type	Rename Process
Conditions	ModelEquals(e3::base_Classifier,e2) AND ModelRelatedTo(e4, Implementation, e2) AND ModelRelatedTo(e4, Tests, e1))
Actions	reportImpact(e1, Rename SUT, , e2)
Rule	IR045
Description	Renaming an SUT requires renaming corresponding implementation
Elements	e1:Class,e2:ClassDeclaration,e3:SUT
Change Type	Rename SUT
Conditions	(ModelEquals(e3::base_Classifier,e1) AND (ModelRelatedTo(e2, Implementation, e1))
Actions	reportImpact(e1, Rename SUTImplementation, , e2)
Rule	IR046
Description	Renaming a Process requires renaming its corresponding TestPackage
Elements	e1:Process,e2:Package,e3:Model
Change Type	Rename Process
Conditions	(ModelRelatedTo(e3, Containment, e2) AND (ModelRelatedTo(e3, Tests, e1))
Actions	reportImpact(e1, Rename TestPackage, , e2)
Rule	IR047
Description	Renaming a TestPackage requires renaming the corresponding implementation
Elements	e1:Package,e2:Package
Change Type	Rename TestPackage
Conditions	ModelRelatedTo(e1, Implementation, e2)
Actions	reportImpact(e1, Rename TestPackageImplementation, , e2)
Rule	IR048
Description	Renaming a Process requires renaming its TestContext
Elements	e1:Process,e2:Class,e3:TestContext,e4:Model
Change Type	Rename Process
Conditions	ModelEquals(e3::base_StructuredClassifier, e2) AND (ModelRelatedTo(e4, Containment,e2) AND ModelRelatedTo(e4, Tests,e1))
Actions	reportImpact(e1, Rename TestContext, , e2)
Rule	IR049
Description	Renaming a TestContext requires renaming its implemenattion in test code
Elements	e1:Class,e2:ClassDeclaration
Change Type	Rename TestContext
Conditions	ModelRelatedTo(e2, Implementation,e1)
Actions	reportImpact(e1, Rename TestContextImplementation, , e2)
Rule	IR050
Description	Renaming a Process requires renaming the test cases of process
Elements	e1:Process,e2:Activity
Change Type	Rename Process
Conditions	(ModelRelatedTo(e1, Tested_By,e2) OR ModelRelatedTo(e2, Tests,e1))
Actions	reportImpact(e1, Rename TestCase, , e2)
Rule	IR051
Description	Adding a Process requires a corresponding TestPackage in the TestModel
Elements	e1:DocumentRoot,e2:TestModel
Change Type	Add Process
Conditions	ModelRelatedTo(e1, Tested_By,e2)
Actions	reportImpact(e1, Add TestPackage, , e2)
Rule	IR052
Description	Adding a Process requires adding corresponding SUT in TestModel
Elements	e1:DocumentRoot,e2:TestModel
Change Type	Add Process
Conditions	ModelRelatedTo(e1, Tested_By,e2)
Actions	reportImpact(e1, Add SUT, , e2)
Rule	IR053
Description	Moving an Operation from one class to another requires moving the ServiceTasks of Participants and Lanes
Elements	e1:Class,e2:Class, e3:Participant | Lane, e4:Participant | Lane
Change Type	Move Operation
Conditions	(ModelRelatedTo(e3, Implementation,e1) AND ModelRelatedTo(e4, Implementation,e2))
Actions	reportImpact(e1, Move ServiceTask, e2, e3 | e4)

Rule	IR054
Description	Moving a ServiceTask from one Lane | participant to another requires moving mockoperations of testcomponents
Elements	e1:Lane | Participant,e2:Lane | Participant, e3:Class, e4:Class
Change Type	Move ServiceTask
Conditions	(ReferenceExists(e1, Derivation,e3) AND ModelRelatedTo(e2, Derivation,e4))
Actions	reportImpact(e1, Move MockOperation, e2, e3 | e4)
Rule	IR055
Description	Deleting a TestPackage requires deletions of TestCompoents
Elements	e1:Package,e2:Class
Change Type	Delete TestPackage
Conditions	(ModelRelatedTo(e1, Containment,e2) AND ValueEndsWith(e1::name, TP))
Actions	reportImpact(e1, Delete TestComponent, , e2)
Rule	IR056
Description	Splitting a Participant requires Splitting corresponding testcomponents
Elements	e1:Participant,e2:Participant, e3:Class
Change Type	Merge Processes
Conditions	ModelRelatedTo(e2, Derivation,e3)
Actions	reportImpact(e1, Split TestComponent, e2, e3)
Rule	IR057
Description	Merging two Processes requires moving the lanes of Target Process to the source Process
Elements	e1:Process,e2:Process, e3:Lane
Change Type	Split Participant
Conditions	ModelRelatedTo(e2, Containment, e3)
Actions	reportImpact(e1, Move Lane, e2, e3 | e1)
Rule	IR058
Description	Splitting a Task into two requires splitting the corresponding MockOperation
Elements	e1:Task | SendTask | ReceiveTask | ServiceTask,e2:Task | SendTask | ReceiveTask | ServiceTask, e3:Operation
Change Type	Split Task
Conditions	ModelRelatedTo(e3, Mocks, e1)
Actions	reportImpact(e1, Split MockOperation, e2, e3)
Rule	IR059
Description	Adding a Task in a Process requires a new TestCase to cover it in the corresponding TestContext
Elements	e1:Process,e2:Class, e3:Model, e4:TestContext
Change Type	Add Task
Conditions	(ModelRelatedTo(e3, Tests, e1) AND (e3, Containment,e2))
Actions	reportImpact(e1, Add TestCase, , e4)
Rule	IR060
Description	Splitting a Lane requires splitting corresponding TestComponent
Elements	e1:Lane,e2:Lane, e3:TestComponent
Change Type	Split Lanes
Conditions	(e2, Derivation, e1)
Actions	reportImpact(e1, Split TestComponent, e2, e3)
Rule	IR061
Description	Merging two tasks requires merging the corresponding MockOperation
Elements	e1:Task | ServiceTask | SendTask | ReceiveTask,e2:Task | ServiceTask | SendTask | ReceiveTask, e3:Operation, e4:Operation, e5:Class
Change Type	Merge Tasks
Conditions	(ModelRelatedTo(e5,Containment,e3) AND ModelRelatedTo(e5,Containment,e4)) AND (ModelRelatedTo(e3,Mocks,e1) AND ModelRelatedTo(e4,Mocks,e2))
Actions	reportImpact(e1, Merge MockOperation, e2, e3 | e4)
Rule	IR0643
Description	Deleting a Lane requires deleting the corresponding TestComponent
Elements	e1:Lane,e2:Class
Change Type	Delete Lane
Conditions	ModelRelatedTo(e1, Derivation, e2)
Actions	reportImpact(e1, Delete TestComponent, , e22)
Rule	IR062
Description	Adding a Lane in a Participant requires corresponding TestComponent in the TestModel
Elements	e1:Participant,e2:Model,e3:DocumentRoot
Change Type	Add Lane
Conditions	(ModelRelatedTo(e3, Containment, e1) AND ModelRelatedTo(e3, Tested_By, e2))
Actions	reportImpact(e1, Add TestComponent, , e2)
Rule	IR064
Description	Renaming a Lane requires renaming corresponding Testcomponent
Elements	e1:Lane,e2:Class
Change Type	Rename Lane
Conditions	ModelRelatedTo(e1, Derivation, e2)

Actions	reportImpact(e1, Rename TestComponent, , e2)
Rule	IR066
Description	Deleting a Gateway requires deleting CondtionalNodes in test behaviors
Elements	e1:ExclusiveGateway,e2:ConditionalNode
Change Type	Delete Gateway
Conditions	ModelRelatedTo(e1, Derivation, e2)
Actions	reportImpact(e1, Delete ConditionalNode, , e2)
Rule	IR067
Description	Adding a Gateway requires new TestCase coverage
Elements	e1:Process,e2:Model
Change Type	Add Gateway
Conditions	ModelRelatedTo(e2, Tests, e1)
Actions	reportImpact(e1, Add TestCase, , e2)
Rule	IR068
Description	Changing Type of a Gateway requires deletion of ConditionalNodes in TestCases
Elements	e1:ExclusiveGateway,e2:ConditionalNode
Change Type	Update GatewayType
Conditions	ModelRelatedTo(e1, Derivation, e2)
Actions	reportImpact(e1, Delete ConditionalNode, , e2)
Rule	IR069
Description	Adding a DataElement in a Process requires corresponding datapool in test data
Elements	e1:Process,e2:Model
Change Type	Add DataElement
Conditions	(ModelRelatedTo(e1, Derivation, e2) AND ValueEndsWith(e2::name, TestData))
Actions	reportImpact(e1, Add DataPool, , e2)
Rule	IR070
Description	Adding a DataElement in a Process requires new DataPartition in test data
Elements	e1:Process,e2:Model
Change Type	Add DataElement
Conditions	ModelRelatedTo(e2, Tests, e1)
Actions	reportImpact(e1, Add DataPartition, , e2)

Test Classification Rules

Table E.3: Test Classification Rules for UTP Test Elements.

Rule	TCR001:Retestable
Description	Deleting an Activity makes the activity test case obsolete
Elements	e1:CallOperationAction, e2:ImpactReport, e3:Activity
Conditions	(ReferenceExists(e2::AffectedElements,e1) AND valueEquals(e2::Solution,"Replace CallOperationAction") AND (modelRelatedTo(e3,Containment,e1)))
Action	TestClassificationAction=e3:retestable
Rule	TCR002:Obsolete
Description	If CallOperationAction of a test case is affected it becomes Retestable
Elements	e1:Activity, e2:ImpactReport
Conditions	(ReferenceExists(e2::AffectedElements,e1) AND valueEquals(e2::Solution,"Delete ActivityTestCase")
Action	TestClassificationAction=e1:obsolete
Rule	TCR003:PRetestable
Description	Adding a MockOperation makes the parent TestComponent Partially Retestable
Elements	e1:Class «TestComponent », e2:ImpactReport
Conditions	(ReferenceExists(e2::AffectedElements,e1) AND valueEquals(e2::Solution,"Add MockOperation")
Action	TestClassificationAction=e1:partiallyRetestable
Rule	TCR004:PRetestable
Description	Adding an Operation in SUT makes it PRetestable
Elements	e1:lass «SUT », e2:ImpactReport
Conditions	(ReferenceExists(e2::AffectedElements,e1) AND valueEquals(e2::Solution,"Add SUTOperation")
Action	TestClassificationAction=e1:partiallyRetestable
Rule	TCR005:PRetestable
Description	Adding a Method in Java Test Component Makes it partially Retestable
Elements	e1:ClassDeclaration, e2:ImpactReport
Conditions	(ReferenceExists(e2::AffectedElements,e1) AND valueEquals(e2::Solution,"Add MOImplementation")
Action	TestClassificationAction=e1:partiallyRetestable

Rule	TCR006:Retestable
Description:	Adding a Parameter in a MockOperation and SUT operation makes it Retestable
Elements	e1:Operation, e2:ImpactReport
Conditions	(ReferenceExists(e2::AffectedElements,e1) OR(valueEquals(e2::Solution,"Add Parameter in MockOperation") AND valueEquals(e2::Solution,"Add Parameter in SUTOperation")
Action	TestClassificationAction=e1:retestable
Rule	TCR007:Retestabale
Description	Adding a Parameter in Java MockOperation and SUT makes it retestable
Elements	e1:MethodDeclaration, e2:ImpactReport
Conditions	(ReferenceExists(e2::AffectedElements,e1) OR(valueEquals(e2::Solution,"Add Parameter in MOImplementation") AND valueEquals(e2::Solution,"Add Parameter in SUTImplementation")
Action	TestClassificationAction=e1:retestable
Rule	TCR008:pRetestable
Description	Adding a DataPool in a data model makes it Partially Retestable
Elements	e1:Model, e2:ImpactReport
Conditions	(ReferenceExists(e2::AffectedElements,e1) AND valueEquals(e2::Solution,"Add DataPool")
Action	TestClassificationAction=e1:partiallyRetestable
Rule	TCR009:pRetestable
Description	:Replacing a Mockoperation in TestComponent makes it Partially Retestable
Elements	e1:Class «TestComponenet », e2:Operation, e3:ImpactReport
Conditions	(ReferenceExists(e3::AffectedElements,e2) AND valueStartsWith(e2::name", 'Mock') AND valueEquals(e3::Solution,"Replace MockOperation") AND modelRelatedTo(e1,Containment, e2)
Action	TestClassificationAction=e1:partiallyRetestable
Rule	TCR010
Description	Deleting a controlFlow in TestCase makes it Retestable if a sequence is added after that between the nodes
Elements	e1:Activity, e2:CallOperationAction,e3:ControlFlow, e4:ImpactReport, e5:ImpactReport
Conditions	(ReferenceExists(e5::AffectedElements,e2) AND valueEquals(e4::Solution,"Delete ControlFlow") AND modelEquals(e4::AffectedElements, e3) AND modelEquals(e3::source, e2) AND modelRelatedTo(e1,Containment, e2), valueEquals(e5::Solution,"Add ControlFlow")
Action	TestClassificationAction=e1:retestable
Rule	TCR011:retestable
Description	Replacing an OpaqueAction with another action makes the test cases retetsable
Elements	e1:OpaqueAction, e2:ImpactReport, e3:Activity
Conditions	(ReferenceExists(e2::AffectedElements,e1) AND valueEquals(e2::Solution,"Replace OpaqueAction") AND modelDirectParentOf(e3, e1)
Action	TestClassificationAction=e3:retestable
Rule	TCR012:retestable
Description	Replacing a ServiceTask requires the test cases corresponding to its Call Operation Action as Retestable
Elements	e1:erviceTask, e2:CallOperationAction, e3:ImpactReport, e4:Activity, e5:Task
Conditions	valueEquals(e3::Solution,"Replace MockOperation") AND modelEquals(e3::ImpactSources, e1) AND modelDirectParentOf(e4,e2) AND modelRelatedTo(e5,Calls, e1) AND modelRelatedTo(e5,Derivation, e2
Action	TestClassificationAction=e4:retestable
Rule	TCR013:retestable
Description	Renaming a CallOperationAction makes a TestCase Retestable
Elements	e1:CallOperationAction,e2:Activity, e3:ImpactReport
Conditions	(ReferenceExists(e3::AffectedElements,e1) AND valueEquals(e3::Solution,"Rename CallOperationAction") AND modelDirectParentOf(e2, e1)
Action	TestClassificationAction=e2:retestable
Rule	TCR014
Description	Renaming a TestComponent makes it PartiallyRetestable
Elements	e1:Class, e2:ImpactReport
Conditions	(ReferenceExists(e2::AffectedElements,e1) AND valueEquals(e2::Solution,"Rename TestComponent")
Action	TestClassificationAction=e1:partiallyRetestable
Rule	TCR015
Description	Moving a ServiceTask to Lane makes the corresponding MockOperation Retestable
Elements	e1:ServiceTask, e2:Operation, e3:ImpactReport
Conditions	(ReferenceExists(e3::AffectedElements,e1) AND valueEquals(e3::Solution,"Move ServiceTask") AND ModelRelatedTo(e2,"Mocks",e1)
Action	TestClassificationAction=e2:retestable
Rule	TCR016
Description	Moving a ServiceTask to Lane makes the corresponding TestComponent PRetestable
Elements	e1:Lane, e2:Class, e3:ImpactReport
Conditions	(ReferenceExists(e3::AffectedElements,e1) AND valueEquals(e3::Solution,"Move ServiceTask") AND ModelRelatedTo(e1,"Derivation",e2)
Action	TestClassificationAction=e2:partiallyRetestable
Rule	TCR017
Description	Moving a ServiceTask to Lane makes the source Component PRetestable
Elements	e1:Lane, e2:Class, e3:ImpactReport

Conditions	(ModelEquals(e3::ImpactSources,e1) AND valueEquals(e3::Solution,"Move ServiceTask") AND ModelRelatedTo(e1,"Derivation",e2)
Action	TestClassificationAction=e2:partiallyRetestable
Rule	TCR018
Description	A MockOperaton corresponding to a moved service task is Retestable
Elements	e1:Operation, e2:ServiceTask, e3:ImpactReport
Conditions	(ReferenceExists(e3::AffectedElements,e2) AND valueEquals(e3::Solution,"Move ServiceTask") AND ValueEndsWith(e1::name, "Mock") AND ModelRelatedTo(e1,"Mocks",e2)
Action	TestClassificationAction=e1:retestable
Rule	TCR019
Description	A test case will be retestable if it calls a MockOperation corresponding to moved ServiceTasks
Elements	e1:CallOperationAction, e2:ServiceTask, e3:Operation, e4:ImpactReport, e5:Activity
Conditions	(ReferenceExists(e4::AffectedElements,e2) AND valueEquals(e4::Solution,"Move ServiceTask") AND ModelRelatedTo(e5, "Containment", e1) AND ModelRelatedTo(e3, "Mocks", e2) AND ModelRelatedTo(e1, "Calls", e3)
Action	TestClassificationAction=e5:retestable
Rule	TCR020
Description	Renaming a TestComponentImplementation makes it PRetestabe
Elements	e1:ClassDeclaration, e2:ImpactReport
Conditions	(ReferenceExists(e2::AffectedElements,e1) AND valueEquals(e2::Solution,"Rename TestComponentImplementation")
Action	TestClassificationAction=e1:partiallyRetestable
Rule	IR021
Description	Make All the Testcases PartiallyRetestable if they call a MockOperation of a Renamed TestComponent
Elements	e1:Activity, e2:Class, e3:Operation, e4:CallOperationAction | SendSignalAction, e5:ImpactReport
Conditions	(ReferenceExists(e5::AffectedElements,e2) AND valueEquals(e5::Solution,"Rename TestComponent") AND ModelRelatedTo(e2, "Containment", e3) AND ModelRelatedTo(e4, "Calls", e3) AND ModelRelatedTo(e1, "Containment", e4)
Action	TestClassificationAction=e1:partiallyRetestable
Rule	TCR022
Description	A TestCase is considered reusable if it is renamed and not retestale
Elements	e1:Activity, e2:ImpactReport
Conditions	(ReferenceExists(e2::AffectedElements,e1) AND valueEquals(e2::Solution,"Rename TestCase")
Action	TestClassificationAction=e1:reusable
Rule	TCR023
Description	A TestModel is considered Obsolete if a change delete Model exists for it
Elements	e1:Model, e2:ImpactReport, e3:TestModel
Conditions	(ModelEquals(e3::base_Package,e1) AND ReferenceExists(e2::AffectedElements,e1) AND (valueEquals(e2::Solution,"Delete Model") OR valueEquals(e2::Solution,"Delete TestModel"))
Action	TestClassificationAction=e1:obsolete
Rule	TCR024
Description	A TestModel is Obsolete if its corresponding TestPackage is Deleted
Elements	e1:Model,e2:TestModel, e3:Package, e4:ImpactReport
Conditions	modelEquals(e2::base_Package,e1) AND modelRelatedTo(e1,Containment, e3) AND valueEndsWith(e3::name,"TP") AND ReferenceExists(e4::AffectedElements,e3) AND (valueEquals(e4::Solution,"Delete Package") OR valueEquals(e4::Solution,"Delete TestPackage")
Action	TestClassificationAction=e1:obsolete
Rule	TCR025
Description	Classifies a TestModel as Retestable
Elements	e1:Model,e2:TestModel, e3:Class, 4:TestContext, e5:Package, e6:ImpactReport, e7:ClassifiedTestElement
Conditions	modelEquals(e2::base_Package, e1) AND modelEquals(e4::base_StructuredClassifier, e3) AND ValueEndsWith(e5::name, "TP") AND ModelRelatedTo(e1, Containment, e5) AND ModelRelatedTo(e1, Containment, e3) AND (NOT(ReferenceExists(e6::AffectedElements, e1) AND valueEquals(e6::Solution,"Delete TestModel") AND (valueEquals(e7::cType,"Obsolete") OR ModelEquals(e7::AffectedTestElement, e5)) AND (valueEquals(e7::cType,"Obsolete") OR ModelEquals(e7::AffectedTestElement, e3))))
Action	TestClassificationAction=e1:retestable
Rule	TCR026
Description	Classifies a TestModel as PartiallyRetestable
Elements	e1:Model, e2:TestModel, e3Class, e4:TestContext, e5:ClassifiedTestElement, e7:Package, e8:ClassifiedTestElement
Conditions	ModelEquals(e2::base_Package, e1) AND ModelEquals(e4::base_StructuredClassifier, e3) AND ValueEndsWith(e7::name, "TP") AND ModelRelatedTo(e1, Containment, e3) AND ModelRelatedTo(e1, Containment, e7) AND ModelEquals(e5::AffectedTestElement, e3) AND valueEquals(e5::cType, "Retestable) AND ModelEquals(e8::AffectedTestElement, e7) AND valueEquals(e8, "Retesable")
Action	TestClassificationAction=e1:partiallyRetestable
Rule	TCR027
Description	Adding a processes requires a New TestModel
Elements	e1:Process, e2:ImpactReport

Conditions	(ReferenceExists(e2::AffectedElements,e1) AND valueEquals(e2::Solution,"Add TestModel")
Action	TestClassificationAction=e1:added
Rule	
Description	
Elements	e1:Package, e2:ImpactReport
Conditions	(ReferenceExists(e2::AffectedElements,e1) AND valueEquals(e2::Solution,"Delete TestPackage")
Action	TestClassificationAction=e1:obsolete
Rule	TCR029
Description	A TestPackage is Retestable if its testcontext is retestable
Elements	e1:Package, e2:TestPackage, e3:ClassifiedTestElement, e4:Class, e5:TestContext
Conditions	ModelEquals(e2::base_Package, e1) AND ModelEquals(e5::base_StructuredClassifier, e4) AND ModelRelatedTo(e1, Containment,e4) AND ModelEquals(e3::AffectedTestElement, e4) AND valueEquals(e3::cType, "retestable)
Action	TestClassificationAction=e1:retestable
Rule	TCR030
Description	Classifies a SUT as Obsolete
Elements	e1:Class, e2:SUT, e3:ImpactReport
Conditions	ModelEquals(e1::base_Classifier, e1) AND ReferenceExists(e3::AffectedElements, e1) AND valueEquals(e2::Solution,"Delete SUT")
Action	TestClassificationAction=e1:obsolete
Rule	TCR031
Description	Classifies a SUT as PartiallyRetestable
Elements	e1:Class, e2:SUT, e3:Operation, e4ClassifiedTestElement, e5:Operation, e6:ClassifiedTestElement
Conditions	ModelEquals(e2::base_Classifier, e1) AND ModelRelatedTo(e1, Containment, e3) AND ModelRelatedTo(e1,Containment, e5) AND ModelEquals(e4::AffectedTestElement, e3) AND ModelEquals(e6::AffectedTestElement, e5) AND valueEquals(e6::cType, "reusable") AND (valueEquals(e4::cType, obsolete") OR valueEquals(e4::cType, "retestable"))
Action	TestClassificationAction=e1:partiallyRetestable
Rule	TCR032
Description	Classifies a TestContext as Obsolete
Elements	e1:Class, e2:TestContext, e3:ImpactReport
Conditions	(ModelEquals(e2::base_StructuredClassifier, e1) AND ReferenceExists(e3::AffectedElements,e1) AND valueEquals(e3::Solution,"Delete TestContext")
Action	TestClassificationAction=e1:obsolete
Rule	TCR033
Description	Classifies a MockOperation as Obsolete
Elements	e1:Class, e2:TestComponent, e3:Operation, e4:ImpactReport
Conditions	ModelEquals(e2::base_StructuredClassifier, e1) AND ModelRelatedTo(e1, Containment, e3) AND ReferenceExists(e4::AffectedElements,e3) AND (valueEquals(e4::Solution,"Delete MockOperation") OR valueEquals(e4::Solution,"Delete Operation"))
Action	TestClassificationAction=e3:obsolete
Rule	TCR034
Description	Make a MockOperation Retestable if a Parameter is added
Elements	e1:Class, e2:TestComponent, e3:Operation, e4:ImpactReport
Conditions	(ModelEquals(e2::base_StructuredClassifier, e1) AND ModelRelatedTo(e1,Containment, e3) AND ReferenceExists(e4::AffectedElements,e3) AND valueEquals(e4::Solution,"Add Parameter")
Action	TestClassificationAction=e3:retestable
Rule	TCR035
Description	Makes a MockOperation Retestable if a Parameter is Deleted from it
Elements	e1:Class, e2:TestComponent, e3:Operation, e4:Parameter, e5:ImpactReport
Conditions	(ModelEquals(e2::base_StructuredClassifier, e1) AND ModelRelatedTo(e1, Containment, e3) AND ModelRelatedTo(e3, Containment, e4) AND ReferenceExists(e5::AffectedElements,e4) AND valueEquals(e5::Solution,"Delete Parameter")
Action	TestClassificationAction=e3:retestable

F

List of Own Publications

1. Qurat-Ul-Ann Farooq, Steffen Lehnert, and Matthias Riebisch: *Analyzing Interplay of Changes and Dependencies for Rule-based Regression Test Selection*, In: Modellierung2014, GI-Edition - Lecture Notes in Informatics Vol. 225, Köllen, pp. 305-320, Vienna, Austria, Mar. 19-21, 2014. (Conference Paper, Peer Reviewed)
2. Steffen Lehnert, Qurat-ul-ann Farooq, and Matthias Riebisch: *Rule-based Impact Analysis for Heterogeneous Software Artifacts*. In: Proc. 17th European Conference on Software Maintenance and Reengineering (CSMR2013), pp. 209-218 Genova, Italy, Mar. 5-8, 2013. (Conference Paper, Peer Reviewed)
3. Qurat-ul-ann Farooq, Matthias Riebisch: *Model-Based Regression Testing: Process, Challenges and Approaches*, In Book: *Emerging Technologies for the Evolution and Maintenance of Software Models*, Jörg Rech, Christian Bunse (Eds.) IGI Global, ISBN: 9781613504383, pp. 254-297, Mar. 27, 2012. (Book Chapter, Peer Reviewed)
4. Qurat-ul-ann Farooq, Matthias Riebisch: *A Holistic Model-driven Approach to Generate U2TP Test Specifications Using BPMN and UML*, In: Proc. Fourth International Conference on Advances in System Testing and Validation Lifecycle (VALID 2012), The, Lisbon, Portugal, pp. 85-92, Nov. 18-23, 2012. (Conference Paper, Peer Reviewed)
5. Stefan Lehnert, Qurat-Ul-Ann Farooq, and Matthias Riebisch: *A Taxonomy of Change Types and its Application in Software Evolution*, In: Proc. 19th IEEE International Conference on the Engineering of Computer Based Systems (ECBS 2012), Novi Sad, Serbia, pp. 98-107, Apr. 11-13, 2012. (Conference Paper, Peer Reviewed)
6. Matthias Riebisch, Stephan Bode, Qurat-Ul-Ann Farooq, and Steffen Lehnert: *Towards Comprehensive Modelling by Inter-Model Links Using an Integrating Repository.* In: Proc. 8th IEEE Workshop on Model-Based Development for Computer-Based Systems – Covering Domain and Design Knowledge in Models within the 18th IEEE International Conference on Engineering of Computer-Based Systems (ECBS2011), Las Vegas, Apr. 27-29, 2011. (Workshop Paper, Peer Reviewed)
7. Aneela Jabeen, Sidra Tariq, Qurat-Ul-Ann Farooq, and Zafar I. Malik: *A lightweight aspect modelling approach for BPMN*, In: Proc. 14th International IEEE Multitopic Conference (INMIC), pp.255-260, Dec. 22-24, 2011. (Conference Paper, Peer Reviewed)
8. Qurat-ul-ann Farooq, Muhammad Zohaib Z. Iqbal, Zafar I. Malik, and Matthias Riebisch: *A Model-Based Regression Testing Approach for Evolving Software Systems with Flexible Tool Support*, In: Proc. IEEE International Conference and Workshops on Engineering of Computer-Based Systems (ECBS), Oxford, United Kingdom, pp. 41-49, Mar.22-26 2010. (Conference Paper, Peer Reviewed)
9. Qurat-ul-ann Farooq: *A Model Driven Approach for Testing Evolving Business Process based Applications*, In: Proc. Doctoral Symposium at International Conference on

Model Driven Engineering Languages and Systems (MODELS), 2010. (Doctoral Symposium Paper)

10. Nosheen Sabahat, Qurat-ul-ann Farooq, and Zafar I. Malik: *Analyzing Impact of Change in Sequence diagrams on State-machine based Regression Testing*, In: Proc. IASTED International Conference on Software Engineering, Innsbruck, Austria, pp. 226-233, Mar. 2010. (Conference Paper, Peer Reviewed)
11. Stephan Bode, Qurat-Ul-Ann Farooq, and Matthias Riebisch: *Evolution Support for Model-Based Development and Testing–Summary*, In: Joint Proceedings of EMDT2010, IWK2010, Ilmenau, Germany, Sept. 13-16, 2010. (Workshop Paper, Not Peer Reviewed)

Co-supervised Thesis

1. Stefan Groß: *Generierung von U2TP-Testfallbeschreibungen aus BPMN Workflowmodellen auf Basis von Eclipse*, Ilmenau University of Technology, Masters Thesis, 2011 (German Only)
2. Rahul Ravindranath: *Mapping Business Resources to UTP*, Ilmenau University of Technology, Masters Thesis, 2013
3. Suraj Maranahalli: *Cost Based Test Prioritization for Business Processes*, Ilmenau University of Technology, Masters Thesis, 2013

G

The Experiment Data

Due to the size of the artifacts and generated reports, it is not possible to include them all here. However, all the evaluation data, case study material, and tool source code is available in the electronic form in the DVD provided with this thesis. In the following, only final test classification reports, generated for each change scenario are presented.

Test Classification Results: Change Scenario 1

TestClassificationReportName:**TCR-C1-001**, AffectedElements: «SUT »HandleTourPlanningProcess , ImpactReportReference: cs1-c1-IR008, ClassificationType: PartiallyRetestable.
TestClassificationReportName:**TCR-C1-002**, AffectedElements: ClassDeclaration HandleTourPlanningProcessTestComp , ImpactReportReference: cs1-c1-IR011, ClassificationType: PartiallyRetestable.
TestClassificationReportName:**TCR-C1-003**, AffectedElements: MockOperation createSOTourPlanMock, ImpactReportReference: cs1-c2-IR009, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C1-004**, AffectedElements: «TestComponent » HandleTourPlanningProcessTestComp, ImpactReportReference: cs1-c1-IR006, ClassificationType: PartiallyRetestable.

Test Classification Results: Change Scenario 2

TestClassificationReportName:**TCR-C2-001**, AffectedElements: «TestComponent » EvaluateProblemProcessTestComp , ImpactReportReference: cs2-c2-IR016,cs2-c1-IR016 , ClassificationType: PartiallyRetestable.
TestClassificationReportName:**TCR-C2-002**, AffectedElements: «SUT » EvaluateProblemProcess, ImpactReportReference:cs2-c2-IR008, cs2-c2-IR019, cs2-c1-IR019, ClassificationType: PartiallyRetestable.
TestClassificationReportName:**TCR-C2-003**, AffectedElements: ClassDeclaration EvaluateProblemProcessTestComp , ImpactReportReference: cs2-c1-IR011, cs2-c2-IR011, ClassificationType: PartiallyRetestable.
TestClassificationReportName:**TCR-C2-004**, AffectedElements: SUTOperation createTourPlan, ImpactReportReference: c2-IR014, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C2-005**, AffectedElements: MethodDeclaration createTourPlan, ImpactReportReference: c2-IR015, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C2-006**, AffectedElements: MethodDeclaration createSOTourPlanMock, ImpactReportReference: c2-IR013, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C2-007**, AffectedElements: Model HandleTourPlanningProcessTestData, ImpactReportReference: c2-IR017, ClassificationType: PartiallyRetestable.

TestClassificationReportName:**TCR-C2-008**, AffectedElements: testCase6#EvaluateProblemProcess, ImpactReportReference: cs2-c3-IR020-3, cs2-c4-IR021-6, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C2-009**, AffectedElements: testCase3#EvaluateProblemProcess, ImpactReportReference: cs2-c3-IR020-1, cs2-c4-IR021-5, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C2-010**, AffectedElements: testCase5#EvaluateProblemProcess, ImpactReportReference: cs2-c3-IR020-2, cs2-c4-IR021-4, ClassificationType: Retestable.

Test Classification Results: Change Scenario 3

TestClassificationReportName:**TCR-C3-001**, AffectedElements: «TestComponent » ServiceReportHandlerTestComp , ImpactReportReference: cs3-c1-IR022, ClassificationType: PartiallyRetestable.
TestClassificationReportName:**TCR-C3-002**, AffectedElements: ClassDeclaration ServiceReportHandlerTestComp, ImpactReportReference: cs3-c1-IR011, ClassificationType: PartiallyRetestable.
TestClassificationReportName:**TCR-C3-003**, AffectedElements: SUTOperation createTourPlan, ImpactReportReference: c3-c3-IR014, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C3-004**, AffectedElements: MockOperation createServiceDiagnosisReportMock, ImpactReportReference: cs3-c2-IR009, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C3-005**, AffectedElements: MethodDeclaration createTourPlan, ImpactReportReference: c3-IR015, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C3-006**, AffectedElements: MethodDeclaration createServiceDiagnosisReportMock, ImpactReportReference: cs3-c2-IR013, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C3-007**, AffectedElements: MethodDeclaration createSOTourPlanMock, ImpactReportReference: c3-IR013, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C3-008**, AffectedElements: «TestComponent » HandleTourPlanningProcessTestComp, ImpactReportReference: c3-IR016, ClassificationType: PartiallyRetestable.
TestClassificationReportName:**TCR-C3-009**, AffectedElements: testCase4#ServiceExecutionProcess , ImpactReportReference: cs3-c2-IR023, ClassificationType: Retestable.

Test Classification Results: Change Scenario 4

TestClassificationReportName:**TCR-C4-001**, AffectedElements: testCase8#HandleTourPlanningProcess , ImpactReportReference: c4-IR007 ClassificationType: Retestable.
TestClassificationReportName:**TCR-C4-002**, AffectedElements: testCase9#HandleTourPlanningProcess , ImpactReportReference: cs4-c1: IR024 ClassificationType: Retestable.

Test Classification Results: Change Scenario 5

TestClassificationReportName:**TCR-C5-001**, AffectedElements: testCase6#HandleTourPlanningProcess, ImpactReportReference: c5-IR007-1, ClassificationType: Retestable.
TestClassificationReportName:**TCR-c5-002**, AffectedElements: testCase3#HandleTourPlanningProcess, ImpactReportReference: c5-IR007-3, c5-IR007-12, c5-IR007-4, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C5-003**, AffectedElements: testCase4#HandleTourPlanningProcess, ImpactReportReference: c5-IR007-10, ClassificationType:Retestable.
TestClassificationReportName:**TCR-C5-004**, AffectedElements: testCase5#HandleTourPlanningProcess, ImpactReportReference: c5-IR007-8, c5-IR007-5, ClassificationType:Retestable.
TestClassificationReportName:**TCR-C5-005**, AffectedElements: testCase8#HandleTourPlanningProcess , ImpactReportReference: c5-IR007-13, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C5-006**, AffectedElements: testCase10#HandleTourPlanningProcess , ImpactReportReference: c5-IR007-7, c5-IR007-2, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C5-007**, AffectedElements: ,testCase7#HandleTourPlanningProcess, ImpactReportReference: c5-IR007-11, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C5-008**, AffectedElements: testCase1#HandleTourPlanningProcess, ImpactReportReference: c5-IR007-6, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C5-009**, AffectedElements: testCase2#HandleTourPlanningProcess , ImpactReportReference: c5-IR007-9, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C5-010**, AffectedElements: «TestComponent » HandleTourPlanningProcessTestComp, ImpactReportReference: c5-IR010-2, c5-IR010-1, ClassificationType: PartiallyRetestable.
TestClassificationReportName:**TCR-C5-011**, AffectedElements: «TestComponent » ServiceOrderSchedularTestCompp, ImpactReportReference: cs5-c1-IR028, ClassificationType: PartiallyRetestable.

Test Classification Results: Change Scenario 6

TestClassificationReportName:**TCR-C6-001**, AffectedElements: testCase5#HandleBookAccomodationProcess, ImpactReportReference: cs6-c1-IR029-6, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C6-002**, AffectedElements: testCase8#HandleBookAccomodationProcess, ImpactReportReference: cs6-c1-IR029-7, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C6-003**, AffectedElements: testCase4#HandleBookAccomodationProcess, ImpactReportReference: cs6-c1-IR029-1, ClassificationType: Retestable.

TestClassificationReportName:**TCR-C6-004**, AffectedElements: testCase7#HandleBookAccomodationProcess, ImpactReportReference: cs6-c1-IR029-3, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C6-005**, AffectedElements: testCase2#HandleBookAccomodationProcess, ImpactReportReference: cs6-c1-IR029-8, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C6-006**, AffectedElements: testCase1#HandleBookAccomodationProcess, ImpactReportReference: cs6-c1-IR029-2, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C6-007**, AffectedElements: testCase6#HandleBookAccomodationProcess, ImpactReportReference: cs6-c1-IR029-5, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C6-008**, AffectedElements: testCase3#HandleBookAccomodationProcess, ImpactReportReference: cs6-c1-IR029-4, ClassificationType: Retestable.

Test Classification Results: Change Scenario 7

TestClassificationReportName:**TCR-C7-001**, AffectedElements: «TestComponent » ServicePlannerTestComp, ImpactReportReference: cs7-c2-IR022, ClassificationType: PartiallyRetestable.
TestClassificationReportName:**TCR-C7-002**, AffectedElements: ClassDeclaration ServicePlannerTestComp, ImpactReportReference: cs7-c2-IR011, ClassificationType: PartiallyRetestable.
TestClassificationReportName:**TCR-C7-003**, AffectedElements: testCase7#ReturnInventoryForServiceOrder , ImpactReportReference: cs7-c3-IR023-2, cs7-c3-IR023-7, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C7-004**, AffectedElements: testCase2#ReturnInventoryForServiceOrder , ImpactReportReference: cs7-c3-IR023-8, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C7-005**, AffectedElements: testCase3#ReturnInventoryForServiceOrder , ImpactReportReference: cs7-c3-IR023-6, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C7-006**, AffectedElements: testCase6#ReturnInventoryForServiceOrder , ImpactReportReference: cs7-c3-IR023-10, cs7-c3-IR023-4, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C7-007**, AffectedElements: testCase4#ReturnInventoryForServiceOrder , ImpactReportReference: cs7-c3-IR023-3, cs7-c3-IR023-5, cs7-c3-IR023-9, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C7-008**, AffectedElements: testCase5#ReturnInventoryForServiceOrder , ImpactReportReference: cs7-c3-IR023-1, ClassificationType: Retestable.

Test Classification Results: Change Scenario 8

TestClassificationReportName:**TCR-C8-001**, AffectedElements: «TestComponent » ApplicationRegistrationHandlerTestComp, ImpactReportReference: cs8-c1-IR039, ClassificationType: PartiallyRetestable.
TestClassificationReportName:**TCR-C8-002**, AffectedElements: MockOperation cancelRegistrationMock, ImpactReportReference: cs8-c1-IR036-3, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C8-003**, AffectedElements: MockOperation saveRegistrationMock, ImpactReportReference: cs8-c1-IR036-2, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C8-004**, AffectedElements: MockOperation verifyApplicationMock, ImpactReportReference: cs8-c1-IR036-4, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C8-005**, AffectedElements: MockOperation startApplicationRegistrationMock, ImpactReportReference: cs8-c1-IR036-1, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C8-006**, AffectedElements: testCase4#RegisterJobApplicantProcess, ImpactReportReference: cs8-c1-IR036-1, cs8-c1-IR036-3, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C8-007**, AffectedElements: testCase1#RegisterJobApplicantProcess, ImpactReportReference: cs8-c1-IR036-1, cs8-c1-IR036-3, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C8-008**, AffectedElements: testCase2#RegisterJobApplicantProcess, ImpactReportReference: cs8-c1-IR036-1, cs8-c1-IR036-2, ClassificationType: Retestable.
TestClassificationReportName:**TCR-C8-009**, AffectedElements: testCase3#RegisterJobApplicantProcess, ImpactReportReference: cs8-c1-IR036-1, cs8-c1-IR036-2, ClassificationType: Retestable.

Test Classification Results: Change Scenario 9

TestClassificationReportName:**TCR-C9-001**, AffectedElements: «TestComponent » ServiceCoordinatorNotifierTestComp, ImpactReportReference: c9-c1-IR028-3, ClassificationType: PartiallyRetestable.
TestClassificationReportName:**TCR-C9-002**, AffectedElements: «TestComponent » ServiceCoordinatorNotifierTestComp (RegisterProblemOnlineProcess), ImpactReportReference: c9-c1-IR028-1, ClassificationType: PartiallyRetestable.
TestClassificationReportName:**TCR-C9-003**, AffectedElements: ClassDeclaration ServiceCoordinatorNotifierTestComp (RegisterProblemOnlineProcess), ImpactReportReference: c9-c1-IR041-2, ClassificationType: PartiallyRetestable.
TestClassificationReportName:**TCR-C9-004**, AffectedElements: ClassDeclaration ServiceCoordinatorNotifierTestComp (EvaluateProblemProcess), ImpactReportReference: c9-c1-IR041-1, ClassificationType: PartiallyRetestable.
TestClassificationReportName:**TCR-C9-005**, AffectedElements: testCase8#HandleWaitingServicePlanningProcess, ImpactReportReference: c9-c1-IR028-3, ClassificationType: PartiallyRetestable.
TestClassificationReportName:**TCR-C9-006**, AffectedElements: testCase7#HandleWaitingServicePlanningProcess, ImpactReportReference: c9-c1-IR028-3, ClassificationType: PartiallyRetestable.
TestClassificationReportName:**TCR-C9-007**, AffectedElements: testCase3#HandleWaitingServicePlanningProcess, ImpactReportReference: c9-c1-IR028-3, ClassificationType: PartiallyRetestable.
TestClassificationReportName:**TCR-C9-008**, AffectedElements: testCase9#HandleWaitingServicePlanningProcess, ImpactReportReference: c9-c1-IR028-3, ClassificationType: PartiallyRetestable.

TestClassificationReportName:**TCR-C9-009**, AffectedElements: testCase4#HandleWaitingServicePlanningProcess, ImpactReportReference: c9-c1-IR028-3, ClassificationType: PartiallyRetestable.
TestClassificationReportName:**TCR-C9-010**, AffectedElements: testCase5#HandleWaitingServicePlanningProcess, ImpactReportReference: c9-c1-IR028-3, ClassificationType: PartiallyRetestable.
TestClassificationReportName:**TCR-C9-011**, AffectedElements: testCase1#HandleWaitingServicePlanningProcess, ImpactReportReference: c9-c1-IR028-3, ClassificationType: PartiallyRetestable.
TestClassificationReportName:**TCR-C9-012**, AffectedElements: testCase1#RegisterProblemOnlineProcess, ImpactReportReference: c9-c1-IR028-2, ClassificationType: PartiallyRetestable.
TestClassificationReportName:**TCR-C9-013**, AffectedElements: testCase2#RegisterProblemOnlineProcess, ImpactReportReference: c9-c1-IR028-22, ClassificationType: PartiallyRetestable.
TestClassificationReportName:**TCR-C9-014**, AffectedElements: testCase1#RegisterJobApplicantProcess, ImpactReportReference: cs8-c1-IR039, ClassificationType: PartiallyRetestable.
TestClassificationReportName:**TCR-C9-015**, AffectedElements: testCase4#RegisterJobApplicantProcess, ImpactReportReference: cs8-c1-IR039 ClassificationType: PartiallyRetestable.

Test Classification Results: Change Scenario 10

TestClassificationReportName:**TCR-C10-001**, AffectedElements: testCase6#ReturnInventoryForServiceOrder , ImpactReportReference: c10-c1-IR050-4, ClassificationType: Reusable.
TestClassificationReportName:**TCR-C10-002**, AffectedElements: testCase6#ReturnInventoryForServiceOrder , ImpactReportReference: c10-c1-IR050-1, ClassificationType: Reusable.
TestClassificationReportName:**TCR-C10-003**, AffectedElements: testCase5#ReturnInventoryForServiceOrder , ImpactReportReference: c10-c1-IR050-6, ClassificationType: Reusable.
TestClassificationReportName:**TCR-C10-004**, AffectedElements: testCase2#ReturnInventoryForServiceOrder , ImpactReportReference: c10-c1-IR050-2, ClassificationType: Reusable.
TestClassificationReportName:**TCR-C10-005**, AffectedElements: testCase4#ReturnInventoryForServiceOrder , ImpactReportReference: c10-c1-IR050-2, ClassificationType: Reusable.
TestClassificationReportName:**TCR-C10-006**, AffectedElements: testCase3#ReturnInventoryForServiceOrder , ImpactReportReference: c10-c1-IR050-3, ClassificationType: Reusable.

Bibliography

[ACCDL00] Antoniol, G. ; Canfora, G. ; Casazza, G. ; De Lucia, A.: Identifying the Starting Impact Set of a Maintenance Request: A Case Study. In: *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*. Zurich, Switzerland, February 2000, S. 227–230 30

[AGEG09] Auer, D. ; Geist, V. ; Erhart, W. ; Gunz, C.: An Integrated Framework for Modeling Process-Oriented Enterprise Applications and Its Application to a Logistics Server System. In: *Logistics and Industrial Informatics, 2009. LINDI 2009. 2nd International*, 2009, S. 1 –6 33

[ANMU07] Ali, A. ; Nadeem, A. ; M.Z.Z.Iqbal ; Usman, M.: Regression Testing Based on UML Design Models. In: *13th Pacific Rim International Symposium on Dependable Computing*, 2007, S. 85–88 146, 147, 149

[Awe85] Awerbuch, Baruch: A new distributed Depth-First-Search algorithm. In: *Information Processing Letters* 20 (1985), Nr. 3, S. 147 – 150. – ISSN 0020–0190 53

[BBG+09] Bakota, Tibor ; Beszédes, Árpád ; Gergely, Tamás ; Gyalai, Milán I. ; Gyimóthy, Tibor ; Füleki, Dániel: *Semi-Automatic Test Case Generation from Business Process Models*. 11th Symposium on Programming Languages and Software Tools, 2009 28, 53

[BC00] Baldwin, Carliss Y. ; Clark, Kim B.: *Design Rules: The power of modularity*. 2000. – ISBN 9780262024662 29, 75, 76

[BDG+07] Baker, Paul ; Dai, Zhen R. ; Grabowski, Jens ; Haugen, Øystein ; Schieferdecker, Ina ; Williams, Clay: *Model-Driven Testing: Using the UML Testing Profile*. 1. 2007. – ISBN 3540725628 2, 47, 48, 50, 51, 161

[BG00] Beydeda, Sami ; Gruhn, Volker: Integrating White- and Black-Box Techniques for Class-Level Regression Testing. (2000) 1, 145, 147, 148

[BL02] Briand, L. ; Labiche, Y.: A UML-based approach to system testing. In: *Software and Systems Modeling* 1 (2002), Nr. 1, S. 10–42 145, 147, 149

[BLOS06] Briand, L. ; Labiche, Y. ; O'Sullivan, L. ; Sówka, M.: Automated impact analysis of UML models. In: *Journal of Systems and Software* 79 (2006), S. 339–352 29, 30, 78

[BLR11] Bode, Stephan ; Lehnert, Steffen ; Riebisch, Matthias: Comprehensive Model Integration for Dependency Identification with EMFTrace. In: *Joint Proceedings of the First International Workshop on Model-Driven Software Migration and the Fifth International Workshop on Software Quality and Maintainability*, 2011, S. 17–20 38, 66, 116

[BLS02] Briand, L. ; Labiche, Y. ; Soccar, G.: Automating Impact Analysis and Regression Test Selection Based on UML Designs. In: *IEEE International Conference on Software Maintenance,*, 2002, S. 0252 1, 107

[BLS03] Briand, L. C. ; Labiche, Y. ; Sullivan, L. O.: Impact Analysis and Change Management of UML Models. In: *Proceedings of the International Conference on Software Maintenance*, 2003, S. 256 78, 145, 147, 149

[BLW05] Baker, Paul ; Loh, Shiou ; Weil, Frank: Model-Driven Engineering in a Large Industrial Context –Motorola Case Study. In: *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems*, 2005. – ISBN 3–540–29010–9, 978–3–540–29010–0, S. 476–491 142

[BLY09] Briand, Lionel C. ; Labiche, Yvan ; Yue, Tao: Automated traceability analysis for UML model refinements. In: *Inf. Softw. Technol.* 51 (2009), February, S. 512–527. – ISSN 0950–5849 1, 12, 40, 72, 145, 147, 149

[BMZ+05] Buckley, Jim ; Mens, Tom ; Zenger, Matthias ; Rashid, Awais ; Kniesel, Günter: Towards a taxonomy of software change: Research Articles. New York, NY, USA : John Wiley & Sons, Inc., September 2005. – ISSN 1532–060X, S. 309–332 29

[BPM10] BPMN: *Business Process Model and Notation (BPMN)*. Available at: http://www.omg.org/spec/BPMN/2.0, 2010 15, 60

[Cle10] Clements, Paul: *Documenting software architectures: views and beyond*. 2. ed. 2010. – ISBN 0–321–55268–7 14, 31

[CP03] Chen, Yanping ; Probert, Robert: A Risk-based Regression Test Selection Strategy. In: *Proceeding of the 14th IEEE International Symposium on Software Reliability Engineering*, 2003, S. 305–306 145, 147, 149

[CPS02] Chen, Yanping ; Probert, Robert L. ; Sims, D. P.: Specification-based regression test selection with risk analysis. In: *Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research*, 2002, 1 145, 147, 149

[CPU07] Chen, Yanping ; Probert, Robert L. ; Ural, Hasan: Regression test suite reduction using extended dependence analysis. In: *Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting*, 2007, S. 62–69 145, 147, 148

[CPU09] Chen, Yanping ; Probert, Robert L. ; Ural, Hasan: Regression test suite reduction based on SDL models of system requirements. In: *Journal of Software Maintenance and Evolution: Research and Practice* 21 (2009), Nr. 6, S. 379–405 145, 147, 148

[DLFO08] De Lucia, A. ; Fasano, F. ; Oliveto, R.: Traceability management for impact analysis. In: *Proceedings of Frontiers of Software Maintenance*, 2008, S. 21–30 30, 60

[DNT08] Dias Neto, Arilo C. ; Travassos, Guilherme H.: Supporting the selection of model-based testing approaches for software projects. In: *Proceedings of the 3rd international workshop on Automation of software test*. New York, NY, USA : ACM, 2008 (AST '08). – ISBN 978–1–60558–030–2, 21–24 17

[DSW04] Deng, D. ; Sheu, P.C.-Y. ; Wang, T.: Model-based testing and maintenance. In: *Multimedia Software Engineering, 2004. Proceedings. IEEE Sixth International Symposium on*, 2004, S. 278–285 146, 147, 149

[DYZ06] Dong, Wen-Li ; Yu, Hang ; Zhang, Yu-Bing: Testing BPEL-based Web Service Composition Using High-level Petri Nets. In: *10th IEEE International Enterprise Distributed Object Computing Conference*, 2006, S. 441 –444 47

[EFHT05] Engels, Gregor ; Förster, Alexander ; Heckel, Reiko ; Thöne, Sebastian: *Process Modeling using UML*. 2005. – 83–117 S. – ISBN 9780471741442 14, 15, 33

[EtCM+10] Elvesæ ter, Brian ; Carrez, Cyril ; Mohagheghi, Parastoo ; Berre, Arne-Jø r. ; Svein G., Johnsen ; Solberg, Arnor: Model-Based Development with SoaML / University of Oslo, Norway. 2010. – Forschungsbericht 32

[FBH+10] Filho, Roberto S. S. ; Budnik, Christof J. ; Hasling, William M. ; McKenna, Monica ; Subramanyan, Rajesh: Supporting Concern-Based Regression Testing and Prioritization in a Model-Driven Environment. In: *Proceedings of Computer Software and Applications Conference* (2010), S. 323–328 12, 145, 147, 149

[FG06] Fluri, Beat ; Gall, Harald C.: Classifying Change Types for Qualifying Change Couplings. In: *Proceeding of the 14th IEEE International Conference on Program Comprehension*, 2006, S. 35–45 29, 74

[FIMN07] Farooq, Qurat-ul-ann ; Iqbal, Muhammad Zohaib Z. ; Malik, Zafar I. ; Nadeem, Aamer: An Approach for Selective State-Machine- based Regression Testing. In: *Proceedings of the 3rd International Workshop on Advances in Model-based Testing*, 2007, S. 44–52 78, 145, 147, 148

[FIMR10] Farooq, Qurat-Ul-Ann ; Iqbal, M. ; Malik, Z.I. ; Riebisch, M.: A Model-Based Regression Testing Approach for Evolving Software Systems with Flexible Tool Support. In: *17th IEEE International Conference and Workshops on Engineering of Computer Based Systems*, 2010, S. 41–49 40, 72, 145, 147, 148

[Fow99] Fowler, Martin: *Refactoring: Improving the Design of Existing Code*. 1999. – ISBN 9780201485677 76

[FPK+12] Fischer, Klaus ; Panfilenko, Dima ; Krumeich, Julian ; Born, Marc ; Desfray, Philippe: Viewpoint-Based Modeling-Towards Defining the

Viewpoint Concept and Implications for Supporting Modeling Tools. In: *EMISA*, 2012, S. 123–136 1

[FR11] *Kapitel* 10. In: Farooq, Qurat-Ul-Ann ; Riebisch, Matthias: *Model-Based Regression Testing: Process, Challenges and Approaches*. IGI Global, 2011, S. 254–297 45, 75, 107

[GFTR06] Garcia-Fanjul, J. ; Tuya, J. ; Riva, C. de l.: Generating test cases specifications for BPEL compositions of web services using SPIN. In: *Proceedings of the International Workshop on Web Services: Modeling and Testing*, 2006, S. 83–94 47

[GG09] Ginige, Jeewani A. ; Ginige, Athula: An Algorithm for Propagating-Impact Analysis of Process Evolutions. In: *Information Systems: Modeling, Development, and Integration* Bd. 20. 2009. – ISBN 978–3–642–01111–5, S. 153–164 27

[GKLE10] Gerth, Christian ; Küster, JochenM. ; Luckey, Markus ; Engels, Gregor: Precise Detection of Conflicting Change Operations Using Process Model Terms. In: *Model Driven Engineering Languages and Systems* Bd. 6395. 2010. – ISBN 978–3–642–16128–5, S. 93–107 29

[Gom11] Gomaa, Hassan: *Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures*. Cambridge University Press, 2011. – ISBN 9781139494731 1, 14, 31

[GPCL08] Gorthi, Ravi P. ; Pasala, Anjaneyulu ; Chanduka, Kailash K. ; Leong, Benny: Specification-Based Approach to Select Regression Test Suite to Validate Changed Software. In: *Asia-Pacific Software Engineering Conference*, 2008, S. 153–160 145, 147, 149

[Gro11] Gross, Stefan: *Generierung von U2TP-Testfallbeschreibungen aus BPMN-Workflowmodellen auf Basis von Eclipse (In German.)*, Ilmenau University of Technology, Diplomarbeit, 2011 54

[Har98] Harrold, Mary J.: Architecture-Based Regression Testing of Evolving Systems. In: *Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis* (1998), S. 73—77 1

[HGGF10] Heinecke, Andreas ; Griebe, Tobias ; Gruhn, Volker ; Flemig, Holger: Business Process-Based Testing of Web Applications., 2010 (Lecture Notes in Business Information Processing), S. 603–614 28

[HH04] Hassan, Ahmed E. ; Holt, Richard C.: Predicting Change Propagation in Software Systems. In: *Proceedings of the 20th IEEE International Conference on Software Maintenance*, 2004, S. 284–293 127

[Hil99] Hilliard, Rich: Views and viewpoints in software systems architecture. In: *Position paper from the First Working IFIP Conference on Software Architecture, San Antonio*, 1999 13

[HKO07] Hartman, Alan ; Katara, Mika ; Olvovsky, Sergey: Choosing a test modeling language: a survey. In: *Proceedings of the 2nd international Haifa verification conference on Hardware and software verification and testing*, 2007, S. 204–218 32

[HNR68] Hart, P. E. ; Nilsson, N. J. ; Raphael, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. In: *IEEE Transactions on Systems Science and Cybernetics* 4 (1968), Nr. 2, S. 100–107 87

[HNS99] Hofmeister, Christine ; Nord, Robert L. ; Soni, Dilip: Describing Software Architecture with UML. In: *Proceedings of the First Working IFIP Conference on Software Architecture*, 1999, S. 145–160 14, 31

[Hon82] Honeyman, Peter: Testing Satisfaction of Functional Dependencies. In: *J. ACM* 29 (1982), Juli, Nr. 3, S. 668–677. – ISSN 0004–5411 33

[HRRW01] Harrold, M.J. ; Rosenblum, D. ; Rothermel, G. ; Weyuker, E.: Empirical studies of a prediction model for regression test selection. In: *IEEE Transactions on Software Engineering* 27 (2001), mar, Nr. 3, S. 248 –263. – ISSN 0098–5589 128

[IEE00] IEEE: 1471-2000-Recommended practice for architectural description of software-intensive systems / IEEE Computer Society. 2000. – Forschungsbericht. – 1–23 S. 13

[III08] Imtiaz, Salma ; Ikram, Naveed ; Imtiaz, Saima: Impact Analysis from Multiple Perspectives: Evaluation of Traceability Techniques. In: *Proceedings of the 3rd International Conference on Software Engineering Advances*, 2008, S. 457–464 30

[IIMD05] Ibrahim, Suhaimi ; Idris, Norbik B. ; Munro, Malcolm ; Deraman, Aziz: Integrating Software Traceability for Change Impact Analysis. In: *The International Arab Journal of Information Technology* 2 (2005), October, Nr. 4, S. 301–308 30

[JK84] Johnson, D.S. ; Klug, A.: Testing containment of conjunctive queries under functional and inclusion dependencies. In: *Journal of Computer and System Sciences* 28 (1984), Nr. 1, S. 167 – 189. – ISSN 0022–0000 33

[JM10] Juristo, Natalia ; Moreno, Ana M.: *Basics of Software Engineering Experimentation*. 1st. 2010. – ISBN 1441950117, 9781441950116 125, 126

[Jun02] Jungmayr, Stefan: Identifying Test-Critical Dependencies. In: *International Conferene on Software Maintenance*, 2002, S. 404–413 33

[KD11] *Kapitel* 2. In: Keller, Anne ; Demeyer, Serge: *Change Impact Analysis for UML Model Maintenance*. IGI Global, 2011, S. 32–56 30

[KDRPP09] Kolovos, Dimitrios S. ; Di Ruscio, Davide ; Pierantonio, Alfonso ; Paige, Richard F.: Different models for model matching: An analysis of approaches to support model differencing. In: *Proceedings of the ICSE Workshop on Comparison and Versioning of Software Models*, 2009, S. 1–6 72

[KGH+] Kung, D. ; Gao, J. ; Hsia, P. ; Wen, F. ; Toyoshima, Y. ; Chen, C.: Change impact identification in object oriented software maintenance 29

[KH09] Khan, Tamim A. ; Heckel, Reiko: A Methodology for Model-Based Regression Testing of Web Services. In: *Academic & Industrial Conference on Practice And Research Techniques, Testing*, 2009, S. 123–124 27

[KKCM04] Koch, Nora ; Kraus, Andreas ; Cachero, Cristina ; Meliá, Santiago: Integration of business processes in web application models. In: *J. Web Eng.* 3 (2004), Mai, Nr. 1, S. 22–49 14, 31, 32, 41, 50, 60

[KRH+08] Kuhn, Adrian ; Rompaey, Bart V. ; Haensenberger, Lea ; Nierstrasz, Oscar ; Demeyer, Serge ; Gälli, Markus ; Leemput, Koenraad V.: JExample: Exploiting Dependencies between Tests to Improve Defect Localization., Springer, 2008 (Lecture Notes in Business Information Processing). – ISBN 978–3–540–68254–7, S. 73–82 33

[Kru95] Kruchten, Philippe: The 4+1 View Model of Architecture. In: *IEEE Softw.* 12 (1995), November, Nr. 6, S. 42–50. – ISSN 0740–7459 14, 31

[Kru00] Kruchten, Philip: *The Rational Unified Process: An Introduction*. Addison Wesley, 2000 46

[KSD09] Keller, Anne ; Schippers, Hans ; Demeyer, Serge: Supporting Inconsistency Resolution through Predictive Change Impact Analysis. In: *Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*. Denver, Colorado, USA, October 2009 30

[KTV02] Korel, B. ; Tahat, L.H. ; Vaysburg, B.: Model based regression test reduction using dependence analysis. In: *Software Maintenance, 2002. Proceedings. International Conference on*, 2002. – ISBN 1063–6773, S. 214–223 145, 147, 148

[Leh11] Lehnert, Steffen: A Review of Software Change Impact Analysis / Ilmenau University of Technology, Department of Software Systems / Process Informatics. 2011. – Forschungsbericht 30

[LFOT07] Lucia, Andrea D. ; Fasano, Fausto ; Oliveto, Rocco ; Tortora, Genoveffa: Recovering Traceability Links in Software Artifact Management Systems Using Information Retrieval Methods. In: *ACM Trans. Softw. Eng. Methodol.* 16 (2007), September, Nr. 4. – ISSN 1049–331X 38

[LFR12] Lehnert, Steffen ; Farooq, Qurat-Ul-Ann ; Riebisch, Matthias: A Taxonomy of Change Types and its Application in Software Evolution. In: *Proceedings of the 19th Annual IEEE International Conference on the Engineering of Computer Based Systems*, 2012, S. 98–107 29, 74

[LFR13a] Lehnert, Steffen ; Farooq, Qurat-Ul-Ann ; Riebisch, Matthias: Rule-based Impact Analysis for Heterogeneous Software Artifacts. In: *Proceedings of the 17th European Conference on Software Maintenance and Reengineering*, 2013 39, 83, 116, 119, 121

[LFR13b] Lehnert, Steffen ; Farooq, Quratulann ; Riebisch, Matthias: Towards a Taxonomy of Software Dependencies for Impact Analysis. In: *White Paper* (2013), S. 1–14 30, 61, 67, 86, 87, 133

[LLZT07] Liu, H. ; Li, Z. ; Zhu, J. ; Tan, H.: Business Process Regression Testing. In: *Service-Oriented Computing ICSOC* (2007), S. 157–168 27

[LQJW10] Li, Bixin ; Qiu, Dong ; Ji, Shunhui ; Wang, Di: Automatic test case selection and generation for regression testing of composite service based on extensible BPEL flow graph. In: *Proceedings of the International Conference on Software Maintenance*, 2010, S. 1–10 27

[LS96] Lindvall, Mikael ; Sandahl, Kristian: Practical implications of traceability. In: *Softw. Pract. Exper.* 26 (1996), October, S. 1161–1180. – ISSN 0038–0644 30

[LS98] Lindvall, M. ; Sandahl, K: Traceability aspects of impact analysis in object-oriented systems. In: *Journal of Software Maintenance: Research and Practice* 10 (1998), January, S. 37–57 30

[LW89] Leung, H.K.N. ; White, L.: Insights into regression testing [software testing]. In: *Proceedings of Conference on Software Maintenance*, 1989, S. 60–69 1, 7, 12, 40, 96, 97

[LZG05] Lin, Yuehua ; Zhang, Jing ; Gray, Jeff: A Testing Framework for Model Transformations. In: *Model-Driven Software Development - Research and Practice in Software Engineering*, 2005, S. 219–236 17

[Mäd10] Mäder, Patrick: *Rule-based maintenance of post-requirements traceability*, Ilmenau University of Technology, Germany, Diss., 2010 29

[MDR06] Muccini, Henry ; Dias, Marcio ; Richardson, Debra J.: Software architecture-based regression testing. In: *Journal of Systems and Software* 79 (2006), Oktober, Nr. 10, S. 1379–1396. – ISSN 0164–1212 145, 147, 149

[MH03] Massol, Vincent ; Husted, Ted: *JUnit in Action*. 2003. – ISBN 1930110995 32

[MM03] Marcus, Andrian ; Maletic, Jonathan I.: Recovering documentation-to-source-code traceability links using latent semantic indexing. In: *Software Engineering, 2003. Proceedings. 25th International Conference on* IEEE, 2003, S. 125–135 38

[MOP12] MOPS: *Adaptive Planning and Secure Execution of Mobile Processes in Dynamic Scenarios*. Availble at: http://mops.uni-jena.de/us/Homepage-page-.html, 2012 18

[MRP06] Mäder, Patrick ; Riebisch, Matthias ; Philippow, Ilka: Traceability for Managing Evolutionary Change. In: *Proceedings of the 15th International Conference on Software Engineering and Data Engineering (SEDE-2006)*, 2006, S. 1–8 75

[MT07] Mansour, Nashat ; Takkoush, Husam: UML based regression testing for OO software. In: *Proceedings of the 11th International Conference on Software Engineering and Applications*, 2007. – ISBN 978–0–88986–706–2, S. 96–101 145, 147, 149

[Muc07] Muccini, Henry: Using Model Differencing for Architecture-level Regression Testing. In: *Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications*, 2007. – ISBN 0–7695–2977–1, S. 59–66 145, 147, 149

[MV05] Martins, Eliane ; Vieira, Vanessa: Regression Test Selection for Testable Classes. In: *Dependable Computing - EDCC 5* Bd. 3463. 2005, S. 453–470 146, 148, 149

[NR07] Naslavsky, Leila ; Richardson, Debra J.: Using traceability to support model-based regression testing. In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 2007, S. 567–570 12, 38, 60, 66, 146, 148, 149

[NZR09] Naslavsky, L. ; Ziv, H. ; Richardson, D.J.: A model-based regression test selection technique. In: *IEEE International Conference on Software Maintenance*, 2009, S. 515–518 17, 26, 27, 72, 146, 148, 149

[NZR10] Naslavsky, L. ; Ziv, H. ; Richardson, D.J.: MbSRT2: Model-Based Selective Regression Testing with Traceability. In: *2010 Third International Conference on Software Testing, Verification and Validation*, 2010, S. 89–98 40, 66, 146, 148, 149

[OAH03] Orso, Alessandro ; Apiwattanapong, Taweesup ; Harrold, Mary J.: Leveraging Field Data for Impact Analysis and Regression Testing. In: *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE'03)*. Helsinki, Finland, 2003, S. 128–137 30

[OB88] Ostrand, T. J. ; Balcer, M. J.: The category-partition method for specifying and generating fuctional tests. In: *Commun. ACM* 31 (1988), Juni, Nr. 6, S. 676–686. – ISSN 0001–0782 55

[OMG14] OMG: *Object Management Group (OMG)*. Last Visited: October, 2014. `http://www.omg.org/`. Version: 2014 15, 18, 60

[OSH04] Orso, Alessandro ; Shi, Nanjuan ; Harrold, Mary J.: Scaling regression testing to large software systems. In: *SIGSOFT Softw. Eng. Notes* 29 (2004), October, S. 241–251 126

[OTPS98] Onoma, Akira ; Tsai, Wei ; Poonawala, Mustafa ; Suganuma, Hiroshi: Regression testing in an industrial environment. In: *Communications of ACM* 41 (1998), Nr. 5, S. 81–86. – ISSN 0001–0782 1

[PC89] Podgurski, A. ; Clarke, L.: The Implications of Program Dependencies for Software Testing, Debugging, and Maintenance. In: *SIGSOFT Softw. Eng. Notes* 14 (1989), November, Nr. 8, S. 168–178 33

[PE00] Penker, Magnus ; Eriksson, Hans-Erik: *Business Modeling With UML: Business Patterns at Work*. 1. 2000. – ISBN 0471295515 1, 14, 15, 31, 33, 48, 136

[PM91] Probert, Robert L. ; Monkewich, Ostap: TTCN: The International Notation for Specifying Tests of Communcations Systems. In: *Computer Networks and ISDN Systems* 23 (1991), S. 417–438 32

[PMFG09] Poshyvanyk, Denys ; Marcus, Andrian ; Ferenc, Rudolf ; Gyimóthy, Tibor: Using information retrieval based coupling measures for impact analysis. In: *Empirical Software Engineering* 14 (2009), Nr. 1, S. 5–32 30

[PUA06] Pilskalns, Orest ; Uyan, Gunay ; Andrews, Anneliese: Regression Testing UML Designs. In: *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, 2006, S. 254–264 78, 146, 148, 149

[PYTB01] Paul, Ray ; Yu, Lian ; Tsai, Wei-Tek ; Bai, Xiaoying: Scenario-Based Functional Regression Testing. In: *Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development*, 2001, S. 496 33

[Rav13] Ravindranath, Rahul: *Mapping Business Resources to UTP*, Ilmenau University of Technology, Diplomarbeit, 2013 55

[RCVD09] Razavizadeh, Azadeh ; Cîmpan, Sorana ; Verjus, Hervé ; Ducasse, Stéphane: Software System Understanding via Architectural Views Extraction According to Multiple Viewpoints. In: *Proceedings of the Confederated International Workshops and Posters on On the Move to Meaningful Internet Systems*, 2009, S. 433–442 14, 31

[RH94a] Rothermel, Gregg ; Harrold, Mary J.: A framework for evaluating regression test selection techniques. In: *Proceedings of the 16th international conference on Software engineering*, 1994, S. 201–210 10, 11

[RH94b] Rothermel, Gregg ; Harrold, Mary J.: Selecting Regression Tests for Object-Oriented Software. In: *Proceedings of the International Conference on Software Maintenance*, 1994, S. 14–25 126, 127

[RH96] Rothermel, G. ; Harrold, M.J.: Analyzing regression test selection techniques. In: *Software Engineering, IEEE Transactions on* 22 (1996), aug, Nr. 8, S. 529 –551 1, 129

[RST+03] Ren, Xiaoxia ; Shah, Fenil ; Tip, Frank ; Ryder, Barbara G. ; Chesley, Ophelia ; Dolby, Julian: Chianti: A Prototype Change Impact Analysis Tool for Java / Rutgers University, Department of Computer Science. 2003 (DCS-TR-533). – Forschungsbericht 29

[RSTC04] Ren, X. ; Shah, F. ; Tip, B.G.R. ; Chesley, O.: Chianti: A Tool for Change Impact Analysis of Java Programs. In: *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, 2004, S. 432–448 29

[RT01] Ryder, Barbara G. ; Tip, Frank: Change Impact Analysis for Object-Oriented Programs. In: *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE '01)*. Snowbird, Utah, USA, June 2001, S. 46–53 30

[RT07a] Ruth, Michael ; Tu, Shengru: A Safe Regression Test Selection Technique for Web Services. In: *Proceedings of the Second International Conference on Internet and Web Applications and Services*, 2007, S. 47 27

[RT07b] Ruth, Michael E. ; Tu, Shengru: Towards automating regression test selection for web services. In: *Proceedings of the 16th international conference on World Wide Web*, 2007, S. 1265–1266 27

[RW03] Rutherford, Matthew J. ; Wolf, Alexander L.: A Case for Test-code Generation in Model-driven Systems. In: *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, 2003, S. 377–396 17

[Sch95] Schwaber, Ken: SCRUM Development Process. (1995). – Presented at OOPSLA'95 Workshop on Business Object Design and Implementation 46

[SDE+10] Sadovykh, A. ; Desfray, P. ; Elvesaeter, B. ; Berre, A.-J. ; Landre, E.: Enterprise architecture modeling with SoaML using BMM and BPMN - MDA approach in practice. In: *6th Central and Eastern European Software Engineering Conference*, 2010, S. 79 –85 15, 19, 32, 41, 60, 65, 87, 136

[SLT+10] Sun, Xiaobing ; Li, Bixin ; Tao, Chuanqi ; Wen, Wanzhi ; Zhang, Sai: Change Impact Analysis Based on a Taxonomy of Change Types. In: *Proceedings of the IEEE 34th Annual Computer Software and Applications Conference*, 2010, S. 373–382 29

[SM08] Soto, Martin ; Munch, Jurgen: Using model comparison to maintain model-to-standard compliance. In: *Proceedings of the 2008 international workshop on Comparison and versioning of software models*, 2008, S. 35–40 72

[Sne04] Sneed, Harry M.: Reverse Engineering of Test Cases for Selective Regression Testing. In: *Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering*, 2004, S. 69 33

[Som10] Sommerville, Ian: *Software Engineering*. 9. Harlow, England : Addison-Wesley, 2010. – ISBN 978–0–13–703515–1 45

[SPLTJ01] Sunyé, Gerson ; Pollet, Damien ; Le Traon, Yves ; Jézéquel, Jean-Marc: Refactoring UML Models. In: *Lecture Notes in Computer Science* 2185 (2001), S. 134–148 76, 77

[SRRT06] Störzer, Maximilian ; Ryder, Barbara G. ; Ren, Xiaoxia ; Tip, Frank: Finding Failure-Inducing Changes in Java programs using Change Classification. In: *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, 2006, S. 57–68 29

[SS09] Stefanescu, Alin ; Schieferdecker, Ina: Model-Based Integration Testing of Enterprise Services. In: *Proceedings of the Testing: Academic and Industrial Conference - Practice and Research Techniques*, 2009, S. 56–60 2

[SSD06] Sneyers, Jon ; Schrijvers, Tom ; Demoen, Bart: Dijkstras algorithm with Fibonacci heaps: An executable description. In: *20th Workshop on Logic Programming*, 2006, S. 182–191 53

[SSS06] Schiefer, Josef ; Saurer, Gerd ; Schatten, Alexander: Testing Event-Driven Business Processes. In: *JCP* 1 (2006), Nr. 7, S. 69–80 28

[SWK09] Stefanescu, Alin ; Wieczorek, Sebastian ; Kirshin, Andrei: MBT4Chor: A Model-Based Testing Approach for Service Choreographies. In: *Model Driven Architecture - Foundations and Applications*. 2009, S. 313–324 47

[TFM06] Tarhini, A. ; Fouchal, H. ; Mansour, N.: Regression testing web services-based applications. In: *International Conference on Computer Systems and Applications*, 2006, S. 163–170 27

[TJJM00] Traon, Y. L. ; Jeron, T. ; Jezequel, J.-M. ; Morel, P.: Efficient object-oriented integration and regression testing. In: *IEEE Transactions on Reliability* 49 (2000), Nr. 1, S. 12–25 146, 148, 149

[TTC13] TTCN3: *The Testing and Test Control Notation (TTCN-3)*. ETSI ES 201 873-1, 2013 17

[TTJ+09] Tun, Thein T. ; Trew, Tim ; Jackson, Michael ; Laney, Robin ; Nuseibeh, Bashar: Specifying features of an evolving software system. In: *Softw. Pract. Exper.* 39 (2009), August, S. 973–1002. – ISSN 0038–0644 63

[UML07] UML: *Super-Structure Specification Unified Modeling Language*. Available at: http://www.omg.org/docs/formal/07-11-04.pdf, November 2007 60, 78

[UTP11] UTP: *UML Testing Profile (UTP)*. Available at: http://www.omg.org/spec/UTP/1.1/PDF, 2011 8, 14, 17, 18, 32

[VBF07] Vidács, László ; Beszédes, Árpád ; Ferenc, Rudolf: Macro Impact Analysis Using Macro Slicing. In: *Proceedings of the Second International Conference on Software and Data Technologies (ICSOFT '07)*, 2007, S. 230–235 30

[VGSMD03] Van Gorp, Pieter ; Stenten, Hans ; Mens, Tom ; Demeyer, Serge: Towards Automating Source-Consistent UML Refactorings. In: *Lecture Notes in Computer Science* 2863 (2003), S. 144–158 76

[VIA11] VIATRA2: *VIATRA2, VIsual Automated model TRAnsformations Framework*. Availabe at: http://www.eclipse.org/gmt/VIATRA2/, June 2011 116

[WGTZ08] Werner, Edith ; Grabowski, Jens ; Troschutz, Stefan ; Zeiss, Benjamin: A TTCN-3-based Web Service Test Framework. In: *Workshop on Testing of Software - From Research to Practice*, 2008 28

[WH02] Westhuizen, Christian van d. ; Hoek, André van d.: Understanding and Propagating Architectural Changes. In: *Third Working IEEE/IFIP Conference on Software Architecture*, 2002, S. 95–109 76

[Whi00] Whittaker, James A.: What Is Software Testing? And Why Is It So Hard? In: *IEEE Softw.* 17 (2000), Januar, Nr. 1, S. 70–79 1

[WHLA97] Wong, W.E. ; Horgan, J.R. ; London, S. ; Agrawal, H.: A study of effective regression testing in practice. In: *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, 1997, S. 264–274 66

[WL92] White, L.J. ; Leung, H.K.N.: A firewall concept for both control-flow and data-flow in regression integration testing. In: *Proceerdings., Conference on Software Maintenance*, 1992, S. 262–271 33

[WLC08] Wang, D. ; Li, B. ; Cai, J.: Regression Testing of Composite Service: An XBFG-Based Approach. In: *IEEE Congress on Services Part II*, 2008, S. 112–119 27

[WO03] Wu, Ye ; Offutt, Jeff: Maintaining Evolving Component-Based Software with UML. In: *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, 2003, S. 133 1, 80, 145, 147, 149

[WR08] Weber, Barbara ; Reichert, Manfred: Refactoring Process Models in Large Process Repositories. In: *Advanced Information Systems Engineering* Bd. 5074. 2008, S. 124–139 29, 168

[WRH+00] Wohlin, Claes ; Runeson, Per ; Höst, Martin ; Ohlsson, Magnus C. ; Regnell, Björn ; Wesslén, Anders: *Experimentation in software engineering: an introduction*. Norwell, MA, USA : Kluwer Academic Publishers, 2000. – ISBN 0–7923–8682–5 136, 137

[WYZS12] Wang, Yi ; Yang, Jian ; Zhao, Weiliang ; Su, Jianwen: Change impact analysis in service-based business processes. In: *Serv. Oriented Comput. Appl.* 6 (2012), Juni, Nr. 2, S. 131–149 28

[XS04a] Xing, Zhenchang ; Stroulia, Eleni: Data-mining in Support of Detecting Class Co-evolution. In: *Proceedings of the 16th International Conference on Software Engineering & Knowledge Engineering*, 2004, S. 123–128 29

[XS04b] Xing, Zhenchang ; Stroulia, Eleni: Understanding Class Evolution in Object-Oriented Software. In: *Proceedings of the 12th IEEE International Workshop on Program Comprehension*, 2004, S. 34–43 29

[XS05] Xing, Zhenchang ; Stroulia, Eleni: UMLDiff: an algorithm for object-oriented design differencing. In: *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005, S. 54–65 29

[YH10] Yoo, S. ; Harman, M.: Regression testing minimization, selection and prioritization: a survey. In: *Software Testing, Verification and Reliability* (2010) 10, 11

[YJH08] Yu, Yanbing ; Jones, James A. ; Harrold, Mary J.: An empirical study of the effects of test-suite reduction on fault localization. In: *Proceedings of the 30th international conference on Software engineering*, 2008, S. 201–210 127

[YLS06] Yuan, Yuan ; Li, Zhongjie ; Sun, Wei: A Graph-Search Based Approach to BPEL4WS Test Generation. In: *International Conference on Software Engineering Advances*, 2006, S. 14 47

[YLY+06] Yan, J. ; Li, Zhongjie ; Yuan, Yuan ; Sun, Wei ; Zhang, Jian: BPEL4WS Unit Testing: Test Case Generation Using a Concurrent Path Analysis Approach. In: *17th International Symposium on Software Reliability Engineering*, 2006, S. 75 –84 47

[YMNCC04] Ying, Annie T. ; Murphy, Gail C. ; Ng, Raymond ; Chu-Carroll, Mark C.: Predicting Source Code Changes by Mining Change History. In: *IEEE Transactions on Software Engineering* 30 (2004), September, Nr. 9, S. 574–586 30

[Yua08] Yuan, Qiulu: A model driven approach toward business process test case generation. In: *10th International Symposium on Web Site Evolution* (2008), S. 41–44 28, 47

[ZFKB12] Zech, Philipp ; Felderer, Michael ; Kalb, Philipp ; Breu, Ruth: A Generic Platform for Model-Based Regression Testing. In: Margaria, Tiziana (Hrsg.) ; Steffen, Bernhard (Hrsg.): *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*. Springer Berlin Heidelberg, 2012 (Lecture Notes in Computer Science 7609). – ISBN 978–3–642–34025–3, 978–3–642–34026–0, S. 112–126 2, 24, 146, 148, 149

[ZRDD08] Zaidman, Andy ; Rompaey, Bart V. ; Demeyer, Serge ; Deursen, Arie v.: Mining Software Repositories to Study Co-Evolution of Production & Test Code. In: *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, 2008, S. 220–229 66

[ZWDZ05] Zimmermann, Thomas ; Weissgerber, Peter ; Diehl, Stephan ; Zeller, Andreas: Mining Version Histories to Guide Software Changes. In: *IEEE Transactions on Software Engineering* 31 (2005), June, Nr. 6, S. 429–445 30

[ZZK07a] Zheng, Yongyan ; Zhou, Jiong ; Krause, Paul: Analysis of BPEL Data Dependencies. In: *33rd EUROMICRO Conference on Software Engineering and Advanced Applications* Bd. 0, 2007, S. 351–358 47

[ZZK07b] Zheng, Yongyan ; Zhou, Jiong ; Krause, Paul: An Automatic Test Case Generation Framework for Web Services. In: *JSW* 2 (2007), S. 64–77 47

List of Own Publications

1. Qurat-Ul-Ann Farooq, Steffen Lehnert, and Matthias Riebisch: *Analyzing Interplay of Changes and Dependencies for Rule-based Regression Test Selection*, In: Modellierung2014, GI-Edition - Lecture Notes in Informatics Vol. 225, Köllen, pp. 305-320, Vienna, Austria, Mar. 19-21, 2014. (Conference Paper, Peer Reviewed)

2. Steffen Lehnert, Qurat-ul-ann Farooq, and Matthias Riebisch: *Rule-based Impact Analysis for Heterogeneous Software Artifacts*. In: Proc. 17th European Conference on Software Maintenance and Reengineering (CSMR2013), pp. 209-218 Genova, Italy, Mar. 5-8, 2013. (Conference Paper, Peer Reviewed)

3. Qurat-ul-ann Farooq, Matthias Riebisch: *Model-Based Regression Testing: Process, Challenges and Approaches*, In Book: *Emerging Technologies for the Evolution and Maintenance of Software Models*, Jörg Rech, Christian Bunse (Eds.) IGI Global, ISBN: 9781613504383, pp. 254-297, Mar. 27, 2012. (Book Chapter, Peer Reviewed)

4. Qurat-ul-ann Farooq, Matthias Riebisch: *A Holistic Model-driven Approach to Generate U2TP Test Specifications Using BPMN and UML*, In: Proc. Fourth International Conference on Advances in System Testing and Validation Lifecycle (VALID 2012), The, Lisbon, Portugal, pp. 85-92, Nov. 18-23, 2012. (Conference Paper, Peer Reviewed)

5. Stefan Lehnert, Qurat-Ul-Ann Farooq, and Matthias Riebisch: *A Taxonomy of Change Types and its Application in Software Evolution*, In: Proc. 19th IEEE International Conference on the Engineering of Computer Based Systems (ECBS 2012), Novi Sad, Serbia, pp. 98-107, Apr. 11-13, 2012. (Conference Paper, Peer Reviewed)

6. Matthias Riebisch, Stephan Bode, Qurat-Ul-Ann Farooq, and Steffen Lehnert: *Towards Comprehensive Modelling by Inter-Model Links Using an Integrating Repository.* In: Proc. 8th IEEE Workshop on Model-Based Development for Computer-Based Systems – Covering Domain and Design Knowledge in Models within the 18th IEEE International Conference on Engineering of Computer-Based Systems (ECBS2011), Las Vegas, Apr. 27-29, 2011. (Workshop Paper, Peer Reviewed)

7. Aneela Jabeen, Sidra Tariq, Qurat-Ul-Ann Farooq, and Zafar I. Malik: *A lightweight aspect modelling approach for BPMN*, In: Proc. 14th International IEEE Multitopic Conference (INMIC), pp.255-260, Dec. 22-24, 2011. (Conference Paper, Peer Reviewed)

8. Qurat-ul-ann Farooq, Muhammad Zohaib Z. Iqbal, Zafar I. Malik, and Matthias Riebisch: *A Model-Based Regression Testing Approach for Evolving Software Systems with Flexible Tool Support*, In: Proc. IEEE International Conference and Workshops on Engineering of Computer-Based Systems (ECBS), Oxford, United Kingdom, pp. 41-49, Mar.22-26 2010. (Conference Paper, Peer Reviewed)

9. Qurat-ul-ann Farooq: *A Model Driven Approach for Testing Evolving Business Process based Applications*, In: Proc. Doctoral Symposium at International Conference on Model Driven Engineering Languages and Systems (MODELS), 2010. (Doctoral Symposium Paper)

10. Nosheen Sabahat, Qurat-ul-ann Farooq, and Zafar I. Malik: *Analyzing Impact of Change in Sequence diagrams on State-machine based Regression Testing*, In: Proc. IASTED International Conference on Software Engineering, Innsbruck, Austria, pp. 226-233, Mar. 2010. (Conference Paper, Peer Reviewed)

11. Stephan Bode, Qurat-Ul-Ann Farooq, and Matthias Riebisch: *Evolution Support for Model-Based Development and Testing–Summary*, In: Joint Proceedings of EMDT2010, IWK2010, Ilmenau, Germany, Sept. 13-16, 2010. (Workshop Paper, Not Peer Reviewed)

Co-supervised Thesis

1. Stefan Groß: *Generierung von U2TP-Testfallbeschreibungen aus BPMN Workflowmodellen auf Basis von Eclipse*, Ilmenau University of Technology, Masters Thesis, 2011 (German Only)

2. Rahul Ravindranath: *Mapping Business Resources to UTP*, Ilmenau University of Technology, Masters Thesis, 2013

3. Suraj Maranahalli: *Cost Based Test Prioritization for Business Processes*, Ilmenau University of Technology, Masters Thesis, 2013

Eidesstattliche Versicherung

Declaration on oath

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Dissertationsschrifft selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

I hereby declare, on oath, that I have written the present dissertation on my own and have not used other then acknowledged resources and aids.

München, den 16. 09. 2016 **Unterschrift:** _____