Analyzing Model Dependencies for Rule-based Regression Test Selection

Qurat-ul-ann Farooq[1], Steffen Lehnert[1], Matthias Riebisch[2]
Ilmenau University of Technology[1]
University of Hamburg[2]
{Qurat-ul-ann.Farooq,Steffen.Lehnert}@tu-ilmenau.de
riebisch@informatik.uni-hamburg.de

Abstract: Unintended side effects during changes of software demand for a precise test case selection to achieve both confidence and minimal effort for testing. Identifying the change related test cases requires an impact analysis across different views, models, and tests. Model-based regression testing aims to provide this analysis earlier in the software development cycle and thus enables an early estimation of test effort. In this paper, we present an approach for model-based regression testing of business processes. Our approach analyzes change types and dependency relations between different models such as Business Process Modeling Notation (BPMN), Unified Modeling Language (UML), and UML Testing Profile (UTP) models. We developed a set of impact rules to forecast the impact of those changes on the test models prior to their implementation. We discuss the implementation of our impact rules inside a prototype tool EMFTrace. The approach has been evaluated in a project for business processes on mobile devices.

1 Introduction

The lifetime of almost any software system is characterized by a continuous need for changes in order to keep them up-to-date. Unintended side effects during these changes of software introduce additional defects and errors to them. Tests as means for error detection however require a high effort. Regression testing aims to reduce this effort by limiting the test execution to a subset of the test cases that correspond to the changes [RH96].

Model-based regression testing (MBRT) has the potential to provide early assessments of test effort by finding the impact of changes using the dependencies between requirements and design models, implementation, and tests. Thus, the test effort can be reduced by starting the test activity before the actual implementation of changes [BLH09]. However, the representation of complex, process-based software systems demands for modeling different views to represent their structure, behavior and other relevant aspects.

These views represent different aspects of the same system, which results in an overlapping of concepts and introduces dependencies between models of different views. Examples of these views for business processes are the *Process View*, which represents the high level business processes of the system, the *Structural View* which represents the component, business resources, and other structural aspects of the system [PE00], and finally the *Test View* which represents the test cases, test data, and other test related aspects [FR12]. De-

pendencies across these views propagate the changes across several models and can also potentially impact the tests. Hence, it is crucial to analyze the dependency relations between models belonging to various views such as to deal with change propagation and to guide the test selection.

Unfortunately, most of the existing approaches use process code for regression testing [WLC08, LQJW10, LLZT07]. Therefore, an early forecast of the required regression testing effort and an early start of the testing activity are not possible. Moreover, cross view dependency relations are not considered for business processes, which results in imprecise test selection. Other model-based regression testing approaches which determine the model dependencies during impact analysis, require repeated dependency analysis for each change [MTN10, PUA06, BLH09], which is not feasible in limited time and budget constraints. Broadly, the problem we focus on in this paper is:

> If a change is applied on any model belonging to the structural or process view of a business process, what will be its impact on the tests.

The main contribution of our approach is twofold. Firstly, we forecast the impact of changes on the *Structural*, *Process* [SDE+10, PE00], and *Test* view [FR12] of business processes to support regression test selection. To do so, we combine various approaches for dependency detection, change modeling, and impact analysis. This allows us to acquire the impacted test elements, which are then classified as required for retest, unaffected or obsolete. Secondly, we develop a set of impact rules to react on various changes of the models of the structural view and the process view. Since we record the dependency relations prior to the test selection, the dependency relations are not required to be repeatedly identified every time the impact analysis is performed, thus increasing the efficiency of the overall process.

The remainder of this paper is organized as follows. Section 2 presents an introduction to the case study we are using in this paper and provides more details for the motivation of our work. Section 3 formulates the test selection problem. Section 4 presents an overview of our approach and elaborates on the dependency relations, changes, and the impact rules. Section 5 discusses the details of test selection and classification. Section 6 presents the tool that implements our approach. Evaluation of the approach on a framework and on a scenario from our case study is presented in Section 7. Related work is discussed in Section 8 and finally, Section 9 concludes the paper and outlines further work.

2 Field Service Technician Case Study

Before we discuss the problem of regression test selection for business processes in detail, we first introduce our case study briefly and then formulate our problem by focusing on several aspects of the case study.

The *Field Service Technician* case study was developed in a joint academic and industrial research project *Adaptive Planning and Secure Execution of Mobile Processes in Dynamic*

Scenarios (MOPS)[1]. Its goals are to automate the processes to assist field service orders, which includes the *planning*, *preparation*, and *execution* of field service orders, *management of field tours*, *management of tools and spare parts*, and *resource scheduling*. The case study is of medium size and complexity and consists of 25 processes and 35 components.

We modeled these processes, their interactions, and the services they utilize (*Process View*) using BPMN collaboration diagrams. The other structural aspects of the processes, for example the services provided by various participants and stakeholders of the processes and their interfaces, the data acquired by the processes, and the relations between the processes and business resources are modeled using UML class and component diagrams (*Structural View*) [SDE$^+$10, PE00, KKCM04] .

Test suites to tests the individual behavior of the processes and their interactions are also required. The test suites which are being used for testing a stable version of processes are known as a *Baseline* test suite. We model the baseline test suites using UTP [2] , which allows us to model several aspects of the tests, such as the *Test Architecture*, *Test Behavior*, and *Test Data* [2] . To evolve the processes of the Service Technician case study, various changes are required to be introduced in these processes.

Section 7.2 presents an illustrative example which includes the details of views, dependencies across these views and some example change scenarios on which our approach is applied. Here, to further motivate the need for our approach, we briefly discuss one of the changes from the scenario presented in Section 7.2.

A yet unrectified functional error in the system demands replacing an existing service with the new one in a process. However, class-methods defined in class diagram implement the interfaces of components, which in turn provide services to the processes. Similarly, test cases in the test view also call these services during the test execution. Moreover, mocks and stubs are implemented to mock the behavior of the class-methods and are used by the test cases. If a service has to be replaced all such dependencies are required to be understood and utilized to find the impacted tests. Thus, the questions arise that how many such dependencies exist between these various views? If a change is to be introduced, which test cases are affected due to such dependencies, and how they are affected?

3 Problem Definition

We consider the discussed various views and dependencies to formulate the problem of regression test selection in the context of business processes.

> Given a process P defined by a set of models S_M= (B, C_D, C_{OD}), where B is a BPMN collaboration diagram representing the *Process View*, C_D is a class diagram, and C_{OD} is a component diagram representing the *Structural View* of P. Given a set of baseline test models T to test P representing the *Test View*, defined by a 2-tuple T=(T_a, T_b), where T_a is a class diagram representing the

[1] See: http://mops.uni-jena.de/us/Homepage-page-.html
[2] http://utp.omg.org/

Test Architecture in UTP and T_b= (b_1, b_2,...,b_n) is a set of activity diagram test cases to test P representing the *Test Behavior* in UTP. Given a set of Changes C=(c_1, c_2,...,c_n), where each $c_i \in$ C is a distinct change type applicable on any model in the set S_M. For any given $c_i \in$ C, the problem is to determine which elements of T will be affected by c_i, which can be represented by $T^{'}$. Moreover, for each element $x \in T^{'}$, it is required to determine how x can be classified for regression testing.

The classification of test elements decides whether to select these elements for regression testing or to omit them. Our classification of test elements is further explained in Section 5. The next section presents an overview of our approach to deal with the aforementioned regression test selection problem.

4 Overview of Our Model-based Regression Test Selection Approach

Our approach is comprised of four major steps which are discussed in the following and are also presented by Figure 1. In the first step, we elicitate and record the dependency

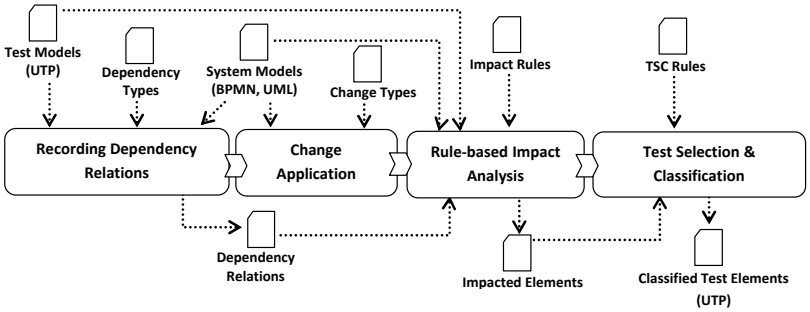

Figure 1: Overview of Regression Test Selection Approach

relations between the system models (S_M to S_M) and between the system models and test models (S_M to T). They remain valid as long as no changes occur, therefore they can be utilized for impact analysis tasks. The S_M to S_M dependencies are recorded using dependency detection rules, whereas the S_M to T dependencies are recorded during the test generation. Section 4.1 discusses the details of the dependency relations and how we record them.

The next step is to apply a change on any of the BPMN or UML models (S_M) and to assess its impact on the tests. We defined a set of change types to model changes as shown in Figure 1. Each model in the set S_M is analyzed for this purpose to identify the applicable changes. Section 4.2 discusses the details of these changes.

When a change is applied on a model, its corresponding impact rules are triggered to identify the potentially affected model elements and test elements. Our impact propagation rules analyze the interplay of change types and dependency relations to identify the affected elements. Section 4.3 elaborates on the structure and application of our impact

rules.

Finally, the impacted elements have to be analyzed to determine which test cases are required for regression testing and which test cases can be omitted. Therefore, we develop a set of test selection and classification rules (TSC Rules). For the classification of the test elements in a UTP model, we adapt and extend the test case classification scheme of Leung and White [LW89] and applied it on UTP test elements. The classification scheme and process are further explained in Section 5.

4.1 Recording Dependency Relations

To model them, we define the set of dependency relations as D=(d_1, d_2,...,d_n), where each d_i is a dependency relation defined by a 3-tuple (source, relation-type, target). The source and target of a dependency relation specify the elements of either S_M or T, which are related to each other. The relation-type defines the purpose of a dependency relation and clarifies its semantics. Dependency relations can be seen as relations between different types of models (*Cross-Model* dependency relations) or relations within the same model (*Intra-Model* dependency relations).

The cross-model dependency relations originating from *Structural* and *Process* views can be categorized into four categories. These categories are also displayed by Figure 2 and are discussed in the following.

Structural View to Structural View: expresses the relationships between UML class and

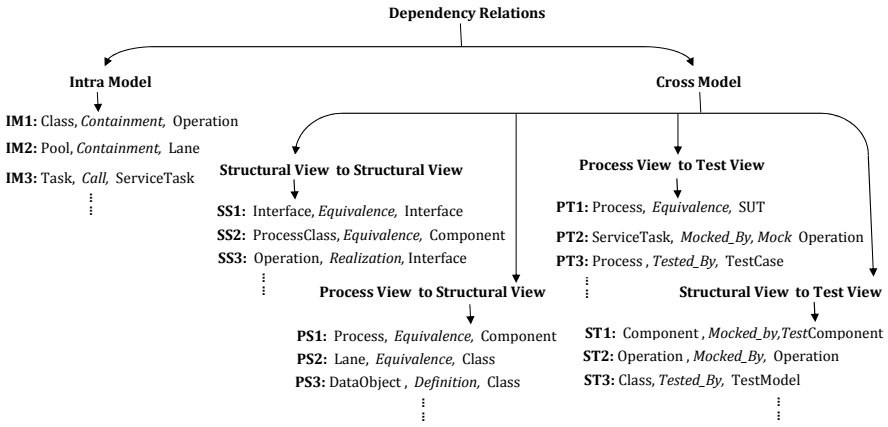

Figure 2: Categories of Cross-Model and Intra-Model Dependency Relations

component diagrams (S_M to S_M). As an example of this category, consider the dependency relation SS1:(Interface, Equivalence, Interface) depicted by Figure 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*.

Process View to Structural View: This category covers dependency relations between BPMN collaboration diagrams and UML class and component diagrams (S_M to S_M). As

an example, consider a *Process* in the BPMN collaboration diagram. It can be defined as a component in a component diagram, where the component will define the interfaces provided and acquired by the process [SDE+10]. This is depicted in Figure 2 as PS1, *(Process, Equivalence, Component)*.

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*.

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)*. It expresses a situation where the behavior of a *Component* defined by a UML component diagram is simulated by a *Test Component* in UTP test architecture.

Intra-Model. This category covers of dependency relations within one model, such as a relation between two elements of a class diagram. Examples are *IM1* and *IM2* as depicted by Figure 2. The dependency relation *IM1* expresses that a class in a UML class diagram can contain operations. A similar dependency relation of type *Containment* is suggested by *IM2*, which expresses that a *Lane* element is contained by a *Pool* element in a BPMN collaboration diagram. To record these dependency relations, we use two different methods as discussed in the following subsections.

Recording Dependency Relations During Test Generation: Dependency relations belonging to the test view (Categories *Process View to Test View* and *Structural View to Test View*) can be recorded during the test generation [NZR10]. Our baseline test suites are generated using a model-driven approach that uses information from BPMN collaboration diagrams and UML class diagrams to generate UTP test architecture and test behavior [FR12]. The UTP test architecture is in the form of UML class diagrams with UTP stereotypes and test behavior is in the form of UML activity diagram test cases generated using path traversal algorithms. During the test generation, the relations between source and target models are also preserved. Each test case in UTP corresponds to a path in BPMN collaboration diagram, thus mappings between the source and target elements also provide the required dependency relations. An example is a *Service Task* in a BPMN collaboration diagram that maps to a *CallOperationAction* in an activity diagram test case. This dependency relation between the *ServiceTask* and *CallOperationAction* is recorded at the time of test generation.

Recording Dependency Relations Using Detection Rules: The intra-model dependency relations (Categories *Structural View to Structural View* and *Process View to Structural View*) do not involve any test models. To record them, we utilize a rule-based approach that was introduced in our previous works [LFR13]. This approach relies on a set of predefined detection rules that are applied on a software and elicitate dependencies between its software artifacts. Each rule is designed for detecting a specific dependency relation using conditions encoded in the rule itself. These conditions allow the rules to query the attributes (e.g. identifiers), relations (e.g. inheritance-relations) and the structure (e.g. parent-child-relations) of 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.

4.2 Change Application

Our rule-based approach requires that changes applicable on the models are well understood and explicitly specified. Therefore we define types of changes belonging to the set S_M by the concept of *Atomic* and *Composite* changes [LFR12] to define the changes for each model in the set S_M. Atomic changes are the basic unit of change, and cover the ***Addition***, ***Deletion***, and ***Updating*** the properties of model elements. A composite change on the other hand is composed of several other atomic or composite changes. The composite changes are: ***Moving***, ***Replacing***, ***Swapping***, ***Merging***, and ***Splitting*** of model elements [LFR12]. The names of these composite changes are self explanatory, whereas their actual definition is context dependent. Every change type $c_i \in C$ is an instance of the aforementioned atomic or composite change types. As discussed earlier, the structural aspects of a process, for example the local or provided operations, can be defined inside a *Class* of a UML class diagram, which is therefore referred to as a *ProcessClass* [KKCM04]. An example change type in this context is *Add Operation in ProcessClass*, which is an instance of the atomic change type *Add*. It adds an operation inside a *ProcessClass* which can provide services to a process.

As discuss earlier, processes require services to commence a process. An example change type in this context is *Replace a Service*, which is an instance of the composite change type *Replace*. Since this is a composite change type, it requires removing the existing *ServiceTask* in BPMN collaboration diagram and adding a new one in the place of the previous *ServiceTask*. It should also update any calls to former *ServiceTask* with the new *ServiceTask*. A change can be selected from the list of pre-defined change types to initiate the impact analysis and test selection process. Hence, the estimation of required regression testing effort is possible even before a change is actually implemented on a model. Hence, our approach is not dependent on any specific change detection strategy such as model comparison as compared to other MBRT approaches[BLH09, NZR10].

4.3 Rule-based Impact Analysis

When a change is introduced to a model, its dependency relations are to be analyzed for change propagation across related models and tests. To do so, we developed a set of impact analysis rules for studying the propagation of changes between the changed model and all related, thus possibly impacted, models using the recorded dependency relations.

An impact rule can be regarded as a 5-tuple R=(c_t,m_e, ED, QD, RD), where c_t defines the change type that acts as the *Change Trigger* for the impact rule and m_e defines the model element on which c_t is applied. The *ElementDefinition*-part (ED) is defined as ED=(e_1,e_2 ,...,e_n), where each e_i is an element from one of the models belonging the set S_M or T. The *QueryDefinition*-part (QD) is defined as QD=(q_1, q_2,...q_n), where each 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 pre-defined operations is *modelRelatedTo(a, t, b)*, which checks if a dependency relation of type *t* exists between a model element *a* and another model element *b*. Finally, the *ResultDefinition*-part (RD) is defined as RD=(a_1, a_2,...,a_n), where each a_i is an action that reports an impact (*Reporting Action*). A *Reporting Action* can further trigger a new change that may also trigger additional impact rules.

The actual execution and processing of our impact rules is accomplished in a recursive manner [LFR13]. First, the initial change (*Change Trigger*) is selected for execution. Rules which react on this kind of change are then being executed and produce a set of impact reports. Each impact report equals a 3-tuple; the source of a change, the change type, and the affected element. 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 an ordered set of impact reports.

4.3.1 Example Impact Rule Illustration

To illustrate the aforementioned concept, we present a scenario and an impact rule which can be applied in the context of this scenario. As discussed earlier in section 4.2, the

```
<rule id="BehavioralImpactRule055">
    <elements>
        <element Type="Class" Alias="e1"/>
        <element Type="TestComponent" Alias="e2"/>
        <element Type="Component" Alias="e3"/>
        <element Type="AtomicChangeType" Alias="e4"/>
    </elements>                                              } Element Definition
    <query>
        <and>
Change Trigger → ValueEquals(e4::name, "Add Operation in Process Class")
            ModelRelatedTo(e3, "Mocked_By", e2)
                ModelRelatedTo(e1, "Equivalence", e3)        } Conditions
        </and>
    </query>
    <results>
        ReportImpact(e1, "Add  MockOperation in TestComponent", e2)  } Impact Report
    </results>
</rule>
```

Listing 1: An Example Impact Rule for the Atomic Change Type "Add Operation"

structure of a process can be modeled using a *ProcessClass* in the system class diagram. However, each *Process* can also be defined as a *Component* in a UML component diagram [SDE$^+$10], which defines its required and provided interfaces which are implemented as operations in the *ProcessClass* corresponding to the process. Thus, a *ProcessClass* that defines a process and its corresponding *Component* are "equivalent"to each other (*SS2* in Figure 2). Furthermore, a *TestComponent* can "mock" the behavior of the *Component* (*ST1* in Figure 2) to test a certain process. In case the *Participant* involved in a collaborative process is changed, its related *Component* and consequently the related *TestComponent* would be affected as well.

The example impact rule depicted in Listing 1 realizes this scenario. The element *e4* is a *Change Trigger* and the name of its associated change type is *"Add Operation in ProcessClass"*. The *Element Definition* part defines the elements to be evaluated by the impact rule, i.e. *Class*, *Component*, and *TestComponent*. The conditions part applies constraints on the model elements defined in the *Element Definition* part. The two main conditions use the *modelRelatedTo*-operation to check if the dependency relations (ProcessClass, *Equivalence*, Component) and (Component, *Mocked_By*, TestComponent) exist for the model elements specified in the element definitions.

If the conditions are satisfied, the next change is triggered and the impact is further propagated as defined in the created *Impact Report*. The impact report states that the source of the change was an element *e1*, a *ProcessClass*, and that the change is propagated to the element *e2*, a *TestComponent*. The next change trigger will be the impact rule corresponding to the change *"Add MockOperation in TestComponent"*, which will be applied on the element *e2*.

5 Test Selection and Classification

We classify a UTP test suite into four types of test cases: *Obsolete*, *Reusable*, *Retestable*, and *New* as suggested by Leung et al. [LW89]. To classify the composite model elements, we also introduce the notion of *Partially Retestable* elements to the classification as explained in the following.

As presented in Section 3, the regression test selection problem consists of two fundamental parts: the identification of elements affected by a change $c_i \in C$, and the classification of these impacted elements. Let $x \in T^{'}$ be an element impacted by a change $c_i \in C$. Let I be the set of reporting actions produced by the impact rules after the application of c_i on x. Let (O, U, R, P, and A) represent the sets of *Obsolete*, *Reusable*, *Retestable*, *Partially Retestable*, and *New* elements. The test classification problem is to determine whether x belongs to O, U, R, P, or A. Where O refers to the set of obsolete elements, which are no more valid for $T^{'}$. The set U represents the set of *Reusable* elements in $T^{'}$, which are not affected when the change c_i is applied. R is the set of elements which are affected by the change c_i and should be used to retest the process P after any required modifications and should be included in $T^{'}$.

We further extend the definition of *Retestable* by using another set of *Partially Retestable* elements. An element $x \in P$ is partially retestable, if at least one of its constituents is *Reusable* and at least one of its child elements is *Retestable*. The 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 determines how it 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. We analyzed the UTP elements and define the classification conditions for them. As an example, below we present some of the conditions under which a *Test Component* element in UTP might belong to one of the classifications.

Example of Classifying a Test Component – Let $x = tc \in T^{'}$ be a UTP *TestComponent*, and let M be the set of *MockOperations* belonging to tc. In case a change c_i is applied on tc, it will be considered as *Obsolete* if the following conditions are met. There exists an impact report $r \in I$ caused by tc, such that the change type of r is *Delete TestComponent*. The origin of the change type *Delete TestComponent* can vary due to the dependency relations.

Element tc is considered *Reusable* if $tc \notin O, R, P, or A$. This means that no reporting action $r \in I$ exists for tc. The element tc will be *Retestable* if $\forall m \in M$ m is *Retestable*. This means that, for a test component to be retestable, all of its mock operations should be affected by c_i. Otherwise, tc will be *Partially Retestable* under the following conditions; **(1)** $\exists m \in M, n \in M$ such that m is *Reusable* and n is *Retestable*, **(2)** $\exists r \in I$, such that the change type of r is *PropertyUpdate* for tc, **(3)** A *MockOperation* is added to M inside a $r \in I$. Finally, tc will be considered as *New* if $\exists r \in I$, such that the change type of r is *(Add TestComponent)* for tc.

6 Tool Support

We implemented our approach in a prototype tool called EMFTrace [3] . EMFTrace is an Eclipse-based tool which is built upon the Eclipse Modeling Framework [4] (EMF), and was initially developed for dependency detection. Our tool offers features for importing models from a variety of tools and modeling languages. It is capable of analyzing various different types of software artifacts for dependency relations. This dependency analysis is implemented by dependency detection rules as introduced in Section 4.1, which are executed by the rule processing component integrated in EMFTrace.

EMFTrace has been extended to allow for rule-based impact analysis [LFR13] as presented in Section 4.3.1. Therefore, the existing rule-processing infrastructure is reused and extended to allow for the generation of impact reports. To perform the test selection, we further extended EMFTrace by implementing a test selector prototype plug-in. This plug-in allows to analyze the impacted elements produced by the impact analyzer and classifies the affected test elements.

7 Evaluation and Application on a Case study

To evaluate our approach, we applied the criteria of the framework of Rothermel *et al* [RH96], which are employed by several MBRT approaches for evaluation [BLH09, NZR10]. The framework consists of 4 major criteria; *Inclusiveness*, *Precision*, *Efficiency* and *Generality*. *Inclusiveness* is the measure to which the modification revealing tests are included. In contrast, *Precision* determines the presence of false positives. *Efficiency* is defined in terms of time and space requirements of the approach, its automatability, the cost of calcu-

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

lating modifications, and the costs that occur during the preliminary and critical phases of testing. *Generality* is the ability of the approach to perform in various practical scenarios [RH96]. Further, we also applied our approach on a case study that automates business processes on mobile devices.

7.1 Evaluation based on Rothermel et al.'s Criteria

Inclusiveness and Precision: Our approach considers 114 different dependency relations, hence all modification revealing test cases covered by them will be considered for retest. Since we cover a comprehensive set of dependency relations, there is less risk of missing any possibly impacted test elements. We explicitly separate the non-modification revealing test cases, i.e. the *Reusable* test cases. Hence, the precision of our approach is considerably higher than the *Retest-All* approach. One of our previous studies show a precision of 80% for the rule-based impact analysis [LFR13]. We expect the same precision, as our approach is also based on the rule-based impact analysis.

Efficiency: We provide the tool support through EMFTrace (see Section 6), which saves the time required for manual analysis. The worst case eventuates if each model of the system is dependent on any other model of the system. Hence there are $n(n-1)/2$ dependency relations for n models, which defines how often each impact rule is executed due to recursion. The time complexity of a single rule computes to $O(n^k)$ where k is the number of elements queried by the rule. Thus, the final time complexity equals $O(m \cdot n^{k+2})$, where m represents the number of rules. Moreover, since the approach is able to forecast the number of test cases affected by a change in earlier phases of development, it can save test costs compared to the approaches that require more effort in later critical testing phases.

Generality: We consider two factors when analyzing the generality of our approach. First, our rules are easier to extend without requiring any change to the underlying rule execution engine and tooling. Hence, our concept offers improved extensibility when compared to other approaches that require changes in their respective tools, which is often a tedious task. Moreover, our approach does not require any explicit model comparison, thus making it easier to integrate with any change detection mechanism than other approaches.

7.2 Application of Our Approach on a Scenario from the Field Service Technician Case study

In the following, we present the application of our approach on a process *TourPlanning-Process* from the *Field Service Technician* case study. The *Tour Planning Process* is responsible for planning a field tour based on various strategies. Figure 3 shows small cutouts of this process, its related classes, components, UTP test models, and their dependencies.

Process View: the box numbered as **(a)** in Figure 3 shows a part of the BPMN collaboration diagram representing the *TourPlanningProcess*. It shows three process participants,

i.e. *TourPlanner*, *RoutePlanner*, and *ServicePlanner*. The participant *Tour Planner* commences the process and collaborates with other participants to create a *TourPlan*. The part presented in Figure 3 focuses on a situation where the *TourPlan* is created based on the shortest available plan between start and end destinations. The participants *RoutePlanner* and *ServicePlanner* provide two services modeled as *Service Tasks*: the *getShortestRoute* service, which provides a shortest route between a given start and destination location, and the *getServiceOrders* service which returns the list of service orders covering a particular route. For our discussion we will concentrate on the dependencies of these participants and services to other models.

Structural View: This view is represented in Figure 3 by the parts **(b)** and **(c)**. Within this view, the *Process View* participants *RoutePlanner*, *TourPlanner*, and *ServicePlanner* are modeled as UML components, following the SoaML modeling method [SDE+10]. The

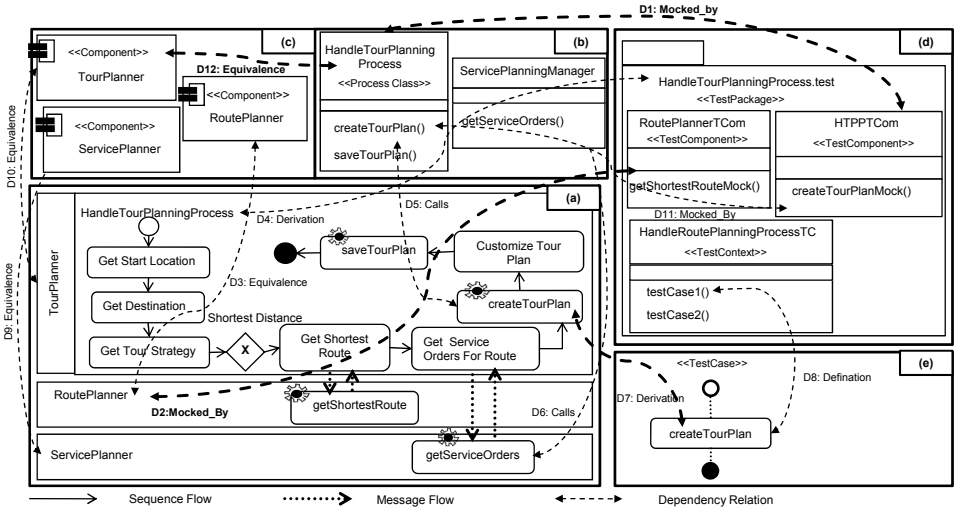

Figure 3: A part of the tour planning process in MOPS, its different views and dependency relations

UML class diagram shown in part **(b)** presents two classes: *HandleTourPlanningProcess*, and *ServicePlanningManager*. The first class represents the structural definition of the process itself and the later class implements an interface of the *ServicePlanner* component. It provides the service *getServiceOrders()* modeled as an operation. **Test View:** Finally, part **(d)** and **(e)** show the *Test View* of the *HandleTourPlanningProcess*. The *Test Architecture* as shown in part **(d)** includes a *TestPackage* and a *TestContext* class corresponding to the *HandleTourPlanningProcess*. The UTP class *TestContext* contains the definitions of all the test cases required for testing the process. However, Figure 3 shows only two of them as an example. The actual test case specification in UTP is represented by an UML activity diagram. Part **(e)** shows the exemplification of a test path, used to test the execution path shown in the process view. The path is derived by applying a path search algorithm using our test generation strategy [FR12].

Dependency Relations Across Views: The dashed arrows in Figure 3 show the dependency relations among the elements of the models. An example dependency relation is

D2, which is shown in bold in Figure 3 and represents a dependency relation of type *Mocked_by* between the participant *RoutePlanner* and its corresponding *TestComponent*. It implements the dependency relation ST1 as presented in Figure 2.

Change Application and Impact Analysis for Replacement of a Service: To illustrate

Table 1: An excerpt of the results of applying *Change 2* on the scenario

	Rule 1	**Rule 2**
Change Operation	Replace ServiceTask	Replace CallOperationAction
Source	createTourPlan: ServiceTask	createTourPlan: CallOperationAction
Target	createTourPlan: CallOperationAction	Operation
Dependency Relation	ServiceTask,*Derivation*, CallOperationAction	CallOperationAction,*Equivalence*, TestOperation
Triggered Change	Replace CallOperationAction	Replace TestOperation

the implementation of our To illustrate the utilization of the dependencies and rules we discuss the implementation of the replacement of an operation. The service task *createTourPlan* was discussed with part **(a)** of Figure 3. Its corresponding *Operation* is represented by the *HandleTourPlanningProcess* in part **(b)** by a UML class with the stereotype ≪`ProcessClass`≫. The operation *createTourPlan()* creates and initializes a *TourPlan* object and returns it to the calling process. However, we identified that the creation of a *TourPlan* in *HandleTourPlanningProcess* not only requires 1the 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. However, we want to keep the existing *createTourPlan* operation due to its utilization in another scenario. The change scenario can now be implemented by evaluating the rules and dependency relations, leading to the following changes.

Change 1 - Atomic (Add): Add a new operation OP: "TourPlan createTourPlan(Tour currentTour, ServiceOrders <`List so`>);" in the *ProcessClass HandleTourPlanningProcess*. This corresponds to the *AddOperation in ProcessClass* change type discussed in Section 4.2.

Change 2 - Composite (Replace): Replace the operation corresponding to createTourPlan ServiceTask in the HandleTourPlanningProcess collaboration diagram with OP. The *Replace ServiceTask* change type is also referred to in Section 4.2.

The application of *Change 1* will activate the impact rule depicted in Listing 1. That will in turn utilize the dependency relations D1 and D12 depicted in figure 3. The rule would then suggest to add a corresponding mock operation inside the test component *HTPPTCom* by triggering the change type *Add MockOperation in TestComponent*, as suggested by the rule in Listing 1.

The application of the *Change 2* requires to trigger the change type *Replace ServiceTask*, which is also discussed in Section 4.2. Table 1 represents partial results in the case when the *Change 2* will be applied to replace a *ServiceTask*. The first column presents the triggering of the first rule, while the second one represents the rule triggered as a result of applying the first rule. When the first rule replaces the *ServiceTask createTourPlan*, it will affect the corresponding *CallOperationAction* in the test case activity diagram due to the

dependency *D7: (createTourPlan, Derivation, createTourPlan)*, as depicted in Figure 3. Finally, *Rule 2* will be triggered, which suggests replacing the *CallOperationAction* inside the activity diagram test case. The rule for replacing the CallOperationAction triggers another rule to replace its corresponding *TestOperation* inside the test code and so on.
Test Selection and Classification for the Scenario: 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. As a new MockOperation is added inside the *HTPPTCom TestComponent*, it would classified as *Partially-Retestable*. This classification is consistent with the case 3 presented in the classification discussed in Section 5.

7.3 Threats to Validity

We identified two major factors that can influence the results achieved by our approach. The first factor is the accuracy of the dependency relations recorded by our dependency detection approach. Although our rules cover several types of constraints for a precise detection of dependency relations, the similarity of the names of model elements, however, still plays a significant role. If proper naming conventions are not followed during the modeling phase, some dependency relation might remain undetected and our approach might produce imprecise results. The other factor is the size of the test suite. If the baseline test suite is already too small, the reduction of cost and effort achieved by our approach might not be significant compared to the retest-all approach. However, the results can still be used to update the baseline test suite.

8 Related Work

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 and cannot forecast the required cost and effort earlier.

Our recent investigations on change impact analysis revealed the lack of support for the interplay of different types of models and software artifacts [LFR13]. A few works, such as the one of 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. However, along with such dependencies, we support a more comprehensive set of other dependency relations between processes, services, components, and test suites.

A number of MBRT approaches only consider intra-model relations inside a single artifact and their impacts on tests [CPU07, CPU09, KTV02, TJJM00]. These approaches are valuable for unit level testing; however, they cannot predict the indirect impacts resulting from the changes in other system artifacts. A large number of MBRT approaches consider

the inter-model relations between artifacts for test selection [MTN10, PUA06, BLH09]. However, they do not record these relations prior to the impact analysis. To perform test selection more than once, each time the relations have to be repeatedly searched; thus, increasing the test selection time.

Recent work by Naslavsky *et al.* [NZR10] makes the dependency links explicit by storing them in a separate model during the test generation process prior to the test selection. However, this approach can only record dependency relations between the design models from which tests are generated and the tests themselves. We are not only using this approach, but also recording other inter-model dependency relations between several design models using additional dependency detection rules. Moreover, our approach further compares to all above mentioned approaches in following ways.

Firstly, these approaches perform the discovery of dependency relations during the impact analysis activity. In our approach, we separate these two activities by discovering the dependency relations in the first phase and later performing the impact analysis. In this way, the discovery part can be reused for other maintenance activities, such as consistency checking of models. Secondly, all above discussed approaches are based on model comparison for test selection. They cannot deal with the changes directly captured from a model editor. Our approach can be integrated with both. Once a change is available, independent from the detection mechanism, the impact analysis activity can be started.

9 Conclusion and Future Directions

In this paper, we presented a model-based regression testing approach for business processes. Our approach determines affected test cases by forecasting the impacts of changes prior to their implementation. For this purpose, we first record the dependency relations between UML models, BPMN models, and UTP test models. As another contribution, we developed a set of impact rules to forecast the impacts of changes and the resulting change propagation on different parts of a test suite. Tests are further classified to decide their inclusion for regression testing. We discussed the implementation of our approach in our prototype tool EMFTrace. To demonstrate the applicability of our approach, we applied it on several change scenarios in a case study on mobile field service technicians developed under the MOPS project. We further evaluated our approach according to the criteria of *Inclusiveness*, *Precision*, *Efficiency*, and *Generality*. Our future work targets on an extension of our impact rules to cover the concrete test scripts. Furthermore, we plan to analyze how risk, cost, and fault severity based approaches can be integrated with our approach for further test prioritization.

References

[BLH09] L. C. Briand, Y. Labiche, and S. He. Automating regression test selection based on UML designs. *Information and Software Technology.*, 51(1):16–30, 2009.

[CPU07] Yanping Chen, Robert L. Probert, and Hasan Ural. Regression test suite reduction using extended dependence analysis. In *SOQUA*, pages 62–69, 2007.

[CPU09] Yanping Chen, Robert L. Probert, and Hasan Ural. Regression test suite reduction based on SDL models of system requirements. *Journal of Software Maintenance and Evolution: Research and Practice*, 21(6):379–405, 2009.

[FR12] Qurat-Ul-Ann Farooq and Matthias Riebisch. A Holistic Model-driven Approach to Generate U2TP Test Specifications Using BPMN and UML. In *Valid 2012*, pages 85–92, 2012.

[GG09] Jeewani Anupama Ginige and Athula Ginige. An Algorithm for Propagating-Impact Analysis of Process Evolutions. In *UNISCON*, volume 20, pages 153–164. 2009.

[KKCM04] Nora Koch, Andreas Kraus, Cristina Cachero, and Santiago Meliá. Integration of business processes in web application models. *J. Web Eng.*, 3(1):22–49, 2004.

[KTV02] B. Korel, L.H. Tahat, and B. Vaysburg. Model based regression test reduction using dependence analysis. In *ICSM 2002*, pages 214–223, 2002.

[LFR12] Steffen Lehnert, Qurat-Ul-Ann Farooq, and Matthias Riebisch. A Taxonomy of Change Types and its Application in Software Evolution. In *ECBS 2012*, pages 98–107, 2012.

[LFR13] Steffen Lehnert, Qurat-Ul-Ann Farooq, and Matt Riebisch. Rule-based Impact Analysis for Heterogeneous Software Artifacts. In *CSMR 2013*, pages 209–218, 2013.

[LLZT07] H. Liu, Z. Li, J. Zhu, and H. Tan. Business Process Regression Testing. *Service-Oriented Computing*, pages 157–168, 2007.

[LQJW10] Bixin Li, Dong Qiu, Shunhui Ji, and Di Wang. Automatic test case selection and generation for regression testing of composite service based on extensible BPEL flow graph. In *ICSM 2010*, pages 1–10, 2010.

[LW89] H.K.N. Leung and L. White. Insights into regression testing [software testing]. In *Conference on Software Maintenance*, pages 60–69, 1989.

[MTN10] Nashat Mansour, Husam Takkoush, and Ali Nehme. UML-based regression testing for OO software. *Journal of Software Maintenance and Evolution: Research and Practice*, 2010.

[NZR10] L. Naslavsky, H. Ziv, and D.J. Richardson. MbSRT2: Model-Based Selective Regression Testing with Traceability. In *ICST 2010*, pages 89–98, 2010.

[PE00] Magnus Penker and Hans-Erik Eriksson. *Business Modeling With UML: Business Patterns at Work*. Wiley, 1 edition, January 2000.

[PUA06] Orest Pilskalns, Gunay Uyan, and Anneliese Andrews. Regression Testing UML Designs. In *ICSM 2006*, pages 254–264, 2006.

[RH96] G. Rothermel and M.J. Harrold. Analyzing regression test selection techniques. *Software Engineering, IEEE Transactions on*, 22(8):529–551, 1996.

[SDE+10] A. Sadovykh, P. Desfray, B. Elvesaeter, A.-J. Berre, and E. Landre. Enterprise architecture modeling with SoaML using BMM and BPMN - MDA approach in practice. In *CEE-SECR*, pages 79–85, 2010.

[TJJM00] Y. Le Traon, T. Jeron, J.-M. Jezequel, and P. Morel. Efficient object-oriented integration and regression testing. *IEEE Transactions on Reliability*, 49(1):12–25, 2000.

[WLC08] Di Wang, Bixin Li, and Ju Cai. Regression Testing of Composite Service: An XBFG-Based Approach. In *2008 IEEE Congress on Services Part II*, pages 112–119, 2008.

[WYZS12] Yi Wang, Jian Yang, Weiliang Zhao, and Jianwen Su. Change impact analysis in service-based business processes. *Serv. Oriented Comput. Appl.*, 6(2):131–149, 2012.