A Holistic Model-driven Approach to Generate U2TP Test Specifications Using BPMN and UML

Qurat-ul-ann Farooq*, Matthias Riebisch†
*Department of Software Systems / Process Informatics, Ilmenau University of Technology
98684 Ilmenau, Germany
{qurat-ul-ann.farooq}@tu-ilmenau.de
†Department of Informatics, University of Hamburg
Vogt-Kölln-Str. 30, 22527 Hamburg, Germany
riebisch@informatik.uni-hamburg.de

Abstract*—Testing process-based information systems is cost intensive and challenging due to rapid technological advancement and increasing complexity of processes. A number of existing process-based test generation approaches use process code for test generation. They operate on lower levels of abstraction and start the test activity later in the development cycle, which is not feasible. Other model-based testing approaches focus only on the individual behavior of a process. They do not consider the structural aspects and process interactions; thus, are not able to capture different test views. In this paper, we present a model-driven test generation approach that uses UML class and component diagrams to model the structural aspects, and BPMN collaboration diagrams to model the collaborative behavior of business processes. Models from both views are used as input to generate the test specifications, which are expressed as of UML 2 Testing profile (U2TP) elements. To identify the correspondences between the process structure, behavior and the test view, we analyze the semantics of UML, BPMN, and U2TP. We developed mapping rules to realize these correspondences for the generation of U2TP test specifications from UML and BPMN models. Our mapping rules are implemented as model transformations using the VIATRA model transformation framework. We illustrate the approach using an example scenario to demonstrate its applicability.

***Keywords-MDA*; *Model-driven Testing*; *BPMN*; *U2TP*; *Business Process Test Generation*.**

I. INTRODUCTION

Testing enterprise software systems is essential to ensure the quality of the systems supporting the underlying business processes. However, due to the increasing complexity of processes and rapid technological advancement, testing requires high effort and huge investments. Furthermore, early testing is required to save project costs. This can be achieved by deriving the test specifications from the process design specifications. However, most of the existing business process-based testing approaches use low level artifacts for test generation, such as process code or web service description language (WSDL) [1][2][3][4], which is often not available in the early phases of software development.

To deal with this issue, Model-driven testing (MDT) [5] for enterprise business processes has been introduced [6][7]. It enables the test generation from the high level process models instead of process code; thus, enabling testing activity in the early phases of the development life cycle. This results in reduced costs and cross platform portability of the test suites. Model-driven testing uses the concept of model transformations to transform the platform independent design models into platform independent test suites. Later, concrete test specifications or test code can be generated from these test models [5]. Hence, to support the model-driven test generation for process-based information systems, there are three major requirements; (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.

In this paper, we present a model-driven test generation approach for process-based information systems. To meet the first requirement, the artifacts required to model different views of process-based information systems are to be analyzed [8]. These different system views include the *Process View*, *Resource/Structure View*, *Behavior View*, and the *User Interface view* [8][9]. The existing model-based testing approaches in the literature only focus on the behavioral view of the processes for the test generation. However, the information from other system views can also be used to generate parts of the test specification. For example, the structural system view can be modeled using the UML class diagram and component diagram. The information about the system structure can be obtained from these models to generate the test architecture and test data [10].

In our approach, to model the process and behavior view we use BPMN collaboration diagrams, while modeling the structural view of the system using the UML component and class diagrams. Both UML and BPMN are standards from the *Object Management Group* (OMG)[11], [12]. We use the process modeling guidelines from the *UML-based Web Engineering* (UWE) [13] approach and the *Service Oriented Architecture Modeling Language* (SoaML) [14].

To fulfill the second requirement, which is support for the test modeling, we use U2TP [10], which is also a test modeling standard from OMG. U2TP covers several important test modeling issues, such as, modeling the *Test Architecture*, *Test Behavior*, *Test Data*, and *Test Time*. In this paper, we focus primarily on the generation of U2TP test architecture and test behavior for business process-based testing. 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. These mappings are defined by analyzing the semantics of the BPMN collaboration diagrams, UML class and component diagrams, and the U2TP test models. The mapping rules are implemented in the form of model transformations. We used *Viatra Transformation Control Language* (VTCL), which is a model transformation language provided by the VIATRA model transformation framework [15] for the implementation of our transformations. The rest of this paper is organized as follows.

Section II discusses the related work and analyzes the strengths and weaknesses of business process-based testing approaches. Section III provides an overview of our approach. Section IV discusses our model-driven test case generation approach in detail and also presents the mapping rules for the test generation. Section V conferes the implementation details and Section VI illustrates the application of the approach on an example scenario. Finally, Section VIII concludes the paper and discusses the future directions of our work.

II. Related Work

Most of the process-based test generation approaches derive the test cases from the process code. These approaches either generate test paths directly from the code, based on data and control flow information [16], [17], or translate the code into formal specifications languages like Petri-nets [2], [3], [18], to perform the model checking and test derivation. One of the major disadvantages of these approaches is that the tests cannot expose the deviations from the functional specifications, as the tests are directly derived from the code. Moreover, the testing activity can only be started after the development is complete, which increases the cost as well as the time allocated to the testing phase.

Werner et al. [4] use the WSDL process specifications for the test generation. They only consider the interfaces of the processes; thus, generate only black box test cases from them and do not consider the internal control flows or data flows of the system.

There are a few approaches focusing on model-based test generation for process-based systems. Bakota et al. [1] use a graph like notation for the process specification, where the nodes of the graph represent activities with distinct input and output parameters. The category partition method is used to derive test data for individual activities. They generate the test paths based on the data values and then convert them into the test frames. The approach presents interesting concepts but targets only the data-based process specification languages. The process models in BPMN support many additional activity types and events and they should also be considered during the test generation.

Heinecke et al. [19] present an approach for test generation, where a process is specified using activity diagrams. However, like Bakota et al, Heinecke et al. also do not support event-based process specifications for the test generation. The major distinctions of our approach from the approaches of Heinecke et al. and Bakota et al. are that we not only support events and various activity types during the test generation, we also use the concept of holistic modeling and test generation. Thus, we focus not only on the behavioral aspects, but also consider the structural aspects of the tests during the test generation.

Yuan et al. [7] present a model-driven test generation approach for process-based systems. Our approach is also using the same foundations as Yuan et al., however, their work is only an initial idea and lacks details regarding the test generation activities and the rules. Moreover, their work focuses only on the test architecture generation and lacks the test behavior generation aspect.

The model-based test generation approaches for testing business processes discussed in this section, do not consider a holistic view of the system during the test generation and rely only on behavioral artifacts, such as graphs or activity diagrams. However, as discussed earlier, the artifacts representing different views of a process-based system can provide input to generate different test views and thus, should be considered during the test generation. In the next section, we present our holistic model-driven approach that uses the artifacts from the structural, behavioral and process views of the system for the test generation.

III. Overview of the Approach

To perform effective model-based testing of process-based systems, the first step is to select an appropriate modeling methodology, and model the system architecture and business processes. After that, a quality assurance (QA) analyst can review the models for testability. This includes checking the models for completeness and validating any constraints required to generate the test specification. In the next step, a test generation tool can generate abstract test specifications using these models. We discuss these steps in detail in the following subsections.

A. Business Modeling using UWE and SoaML

As discussed earlier, we model the system architecture and the processes using the UWE [13] and SoaML [14] approaches. Since the focus of this paper is on the test generation by analyzing process interactions and structural aspects of the system, we briefly discuss the artifacts we use from these approaches and their specifics.

The UWE approach originally uses an activity like model to model the processes. However, instead of using that, we use BPMN collaboration diagrams and process diagrams to model the behavior of interacting and atomic processes. To model the structural aspects of the system, we use the UML class diagram. Using the UWE approach, each process itself is modeled as a class with a stereotype «ProcessClass» and the data and resources of the system are modeled as classes with a stereotype «entity». From the SoaML approach, we use the component models to define the high level structure of process-based systems.

As discussed earlier, once the models are complete, they should be analyzed for testability. Testability is an important property of the model because a less testable model can results in poor test cases. The testability requirements of our approach are discussed in the following.

→ Completeness: The artifacts used as input are complete and all the processes have their corresponding structures defined in the corresponding UML class diagram.

→ Since handling of multiple entry and exit points is complex, we restrict a collaboration diagram to exactly one start and end node within one pool to reduce additional complexity

→ To ease the testing, we restrict each Pool in the collaboration diagram, to have a corresponding component in the component diagram with well defined interfaces for access. A lane can map to a process class or a service class in the UML class diagram.

Once the models are complete and reviewed by a QA expert for testability, they can be used to generate the test specifications. In the next section, we discuss the foundations of our model-driven test generation approach, and in the later sections we elaborate the approach in more detail.

B. The Abstract Test Specification Generation

As discussed earlier, we use U2TP for the specification of the test architecture and test behavior. The following tasks are to be performed for model driven test generation using our approach.

1) Generate the test architecture by analyzing the structural system models, which are in our case UML class and component diagrams.
2) Transform the test architecture into a class diagram to support the test visualization.
3) Generate the test behavior from BPMN collaboration models in the form of test paths. These test paths can be constructed using path analysis algorithms from the graph-based test generation approaches using a certain coverage criterion.
4) In the next step, the generated paths are transformed into UML activity diagram paths. This transformation satisfies the second requirement stated in the introduction section of this paper. At this stage the tester can analyze each individual test path and add the additional information, such as test verdicts etc.

Test data can also be generated by using process constraints and data resources defined in the structural models; however, the test data generation is out of scope of the current paper. In the next section, we discuss our model-driven test generation process and the above discussed tasks in detail .

IV. THE MODEL-DRIVEN TEST GENERATION PROCESS

To generate the test specifications, we adapted the classic model-driven test generation process by Dai et al. [5]. This

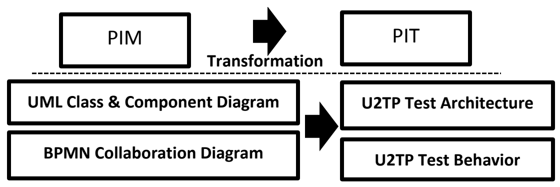


Figure 1. The Example Software Architecture of a Banking Example

process involves the transformation of a platform independent model (PIM) into a platform independent test model (PIT). The upper part of Figure 1 shows this pattern from Dai et al. [5], where a PIM model is transformed to a PIT model.

The lower part of Figure 1 shows our adaptation of the process for the generation of U2TP test architecture and test behavior using UML and BPMN models. The test models in U2TP should cover the structural and behavioral aspects of the test system. These aspects can be derived from the structural and behavioral specification of the system. To do this, we transformed the platform independent UML class and component diagrams representing the structure and resources of the processes into U2TP test architecture models. For the test behavior generation, the platform independent BPMN Collaboration diagrams and process diagrams are transformed into U2TP test behavior, which represents the abstract platform independent test specification.

To define these transformations, the mapping relations between the elements of source and target languages are to be identified. We identify these mapping relations by analyzing the correspondences between the relevant elements of these languages. The elements and semantics of the UML class and component diagrams, BPMN process and collaboration diagrams, and U2TP test models are defined in their respective meta-models[11], [12], [10].

We define the mapping relations in the form of mapping rules. 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 (*Source Element*, *Target Element*, *Rule Preconditions*, *Rule Postconditions*), where the *Source Element* is an element of the source PIM model, i.e., UML or BPMN, and the *Target Element* is an element in the target PIT model, i.e., U2TP. The *Preconditions* define any constraints for the execution of the rule and the *Postconditions* define the changes in the state of the target test models, such as the addition of new elements. An example of such a rule is presented in Listing 1. In the next subsections, we discuss both U2TP test architecture and test behavior generation activities, and the mapping rules we developed to define the transformations in detail.

A. Generation of U2TP Test Architecture

The test architecture is a representation of the structural aspects of a test system. To define the test architecture, U2TP provides several elements. These elements are: *System Under Test* (SUT), *Test Arbiter*, *Test Scheduler*, *Test Context*, and *Test Components*. To specify the test architecture to test a process, these elements are required to be specified. For this purpose, we analyze the elements representing the process structure and derive the test structure from these elements.

```
Source Element: Class, Test package: $CD^{SA}$
Target Element: Class, Association: $CD^{TA}$
Preconditions:
1. $\exists Class.C \in CD^{SA}$
2. ∃ Stereotype.ProcessClass ∈ C
3. ∃ U2TP.TestPackage $(TP \mid TP \in CD^{TA})$;
4. C ∈ TP
5. ∃ BPMN.Collaboration Diagram $(CD \mid CD \in C)$;
Postconditions:
6. $\exists Class.T \in CD^{TA}$
7. T.Name=C.Name
8. ∃ Stereotype.SUT ∈ T
9. $\exists Dependency.Import.A \in CD^{TA}$
10. $A \in TP, A \in T$
```

Listing 1. A Mapping Rule for SUT

To represent the test architecture, U2TP proposes the UML class diagram notation with stereotypes for the U2TP elements. We refer to the class diagram representing the test architecture as CD^{TA}, and the class diagram representing the system architecture as CD^{SA} in the our mapping rules. In the following, we discuss, how we derived the test architecture elements from the UML class and component diagrams, which represent the system structural aspects.

In the UWE process modeling approach, a class corresponding to a collaborative process is represented by a stereotype «ProcessClass». Since a *Process Class* defines the process representing interactions between several participants, it is a candidate class for testing the collaborative process. This means that this class can be treated as a system under test or process under test. Hence, we map each *Class* with a stereotype «ProcessClass» in the CD^{SA} to a *System under Test (SUT)* in the CD^{TA}. This mapping is realized by the mapping rule presented in Listing 1.

According to the rule, there are three preconditions in the *Preconditions* part. The first precondition (Line 1 and 2) states that a class named C with the stereotype «ProcessClass» should exist in the system architecture class diagram CD^{TA}. The second precondition (Line 3 and 4) indicates that a corresponding *Test Package* should be present in the test architecture class diagram, and the third precondition (Line 5) ensures that a BPMN collaboration diagram is present that corresponds to the process class C. The presence of a test package is required due to the reason that each test related element of a particular process class is packaged into one particular test package for better test organization [5].

In the postcondition part, a type *Class* T with a stereotype «SUT» corresponding to the class C is created in the test architecture class diagram CD^{TA} (Line 6, 7, and 8). Since elements of a test package require the SUT for the test execution [5], an import dependency A is created between the test package TP and the SUT class T (Line 9 and 10).

To map the other elements in the test architecture, such as the *Test Context* and the *Test Components*, we developed mapping rules like the one shown Listing 1. The rules to generate the test components are more complex as they require additional information from the UML component diagram and the BPMN collaboration diagram. Due to the space limitations, it is not possible to include all of these rules in the paper; however, a subset of these rules is available on our website for additional reading[20]. After establishing the mappings for the U2TP test architecture, the next step is to develop the mappings to generate U2TP test behavior, which is explained in the next section.

B. Generation of U2TP Test Behavior from BPMN Collaboration Diagrams

In this section, we discuss our test behavior generation process and the mapping rules we developed for the test behavior generation. Our test behavior generation process comprises of two major tasks; the generation of test paths from BPMN collaboration diagrams, 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 the activity diagram test cases should be independent of the coverage criteria and test path generation strategies. So that it can be reused with different coverage criteria and path generation activities. To enable such reuse, we split our test generation process into three sub activities, as depicted in Figure 2.

According to Figure 2, the first activity prepares 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.

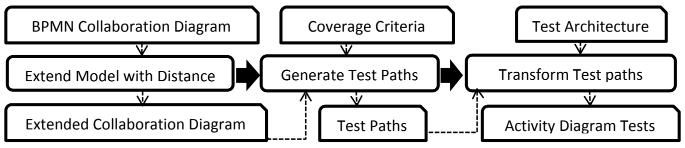


Figure 2. Test Behavior Generation Activities

The output of this activity is a diagram extended with distance information. In the second activity, the test paths are extracted from the extended collaboration diagram based on some coverage criterion. In this paper, we use the branch coverage criterion, which is further explained in the next subsection. Finally, during the third activity, the generated paths are transformed into UML activity diagrams to support the visualization and test documentation.

In U2TP, the test behavior can be specified using either the UML sequence diagrams or the UML activity diagrams. In this work, we selected the UML activity diagram notation for the test behavior specification due to its natural similarity to the process specifications. As discussed earlier, the mapping activity to map the test paths onto UML activity diagrams is independent of the test path generation activity; thus, it can be reused with multiple coverage criteria. In the following subsections, we discuss the test path generation and the details of mapping the test paths onto UML activity diagrams.

1) Generate Test paths: To enable the test path generation from BPMN collaboration diagrams, we use the branch coverage criterion. This criterion covers all the gateways in the diagram for all possible outcomes, and covers each loop once. For computing the test paths, we first compute the shortest path in the collaboration diagram by using the Dijkstra algorithm [21]. 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 [22]. The reason for selecting the DFS is its ability to cover all the branches of a graph by visiting all child nodes of a node and to backtrack, 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 pools is also copied with that task. BPMN collaboration diagrams can contain parallelism by using the parallel gateways in the diagram. To deal with it, one of the simplest strategies can be to place all the branches of the gateway in a sequential order [1]. However, more complex execution strategies can exist. We are treating all the branches of a parallel gateway as one test case and defer the decision of their execution strategies to the concrete test generation activity. After the generation of the test paths, the next activity is to map each test path onto a UML activity diagram, which is discussed in the next section.

2) Transform Test Paths to UMLAD: The transformation of the test paths extracted in the previous steps to a UML activity diagram requires the identification and definition of mappings between the elements of BPMN collaboration diagram and UML activity diagram. These mappings can be identified by analyzing the constructs of BPMN collaboration diagram and the corresponding elements in the UML activity diagram. Based on these correspondences, we develop the mapping rules, which realize the correspondences or mapping functions.

```
Source Element: Collaboration: Message Start Event
Target Element: Activity: Initial Node, AcceptEventAction,
    SignalEvent, Controlflow
Preconditions:
∃ BPMN.Pool (X)∨ BPMN.Lane(X);
∃BPMN.EventStartMessage(Start | Start ∈ X);
∃UML.ActivityPartition(AP | AP ∈ X);
Postconditions:
∃ UML.InitialNode(I | I ∈ AP);
∃ UML.AcceptEventAction(accept | accept ∈ AP, name(accept)
    =name(Start)+'AcceptAction');
∃ UML.OutputPin(OP | OP ∈ accept);
∃ UML.Trigger(trigger| trigger ∈ accept, name(trigger)=
    name(Start)+'Trigger');
∃ UML.SignalEvent(te | te ∈ trigger);
∃ ControlFlow(cf | cf(I, accept));
```

Listing 2. An excerpt of the Mapping Rule for Message Start Event

BPMN collaboration diagrams comprise of a set of start, end, and intermediate nodes, tasks and activities, pools and lanes, control flows and message flows, gateways, and several data elements. A mapping function or mapping rule is required to be defined between each of these elements and their corresponding activity diagram elements.

In the following, we provide an example of a mapping rule that maps a start event in BPMN collaboration diagram to the corresponding activity diagram elements. The start and end nodes in a BPMN collaboration diagram can be mapped to the *Initial Node* and *Final Node* in the UML activity diagram. However, BPMN collaboration diagrams have many different types of start and end nodes, such as, *Message Start Event*, *Empty Start Event*, *Timer Start Event*, and many others. Since the UML activity diagram has only one Initial Node type, it can be mapped to only one type of start event in BPMN collaboration diagrams, i.e., *Empty Start Event*. For the remaining events, more complex patterns are required to be identified in the UML activity diagrams.

An example of such event is the *Message Start Event*.

The mapping rule for the *Message Start Event* is depicted in Listing 2. According to Listing 2, a *Message Start Event* can be mapped to an *Initial Node* followed by an *Accept event Action* in a UML activity diagram. The preconditions stated in the preconditions part specify that in the BPMN collaboration diagram, either a *Pool* or a *Lane* X exists and the *Message Start Event* belongs to that *Pool/Lane*. The next precondition is that an *Activity partition* exists corresponding to the *Pool/Lane* X. We map a *Pool* or a *Lane* in the BPMN collaboration diagram to an *ActivityPartition* element in the UML activity diagram. This *ActivityPartition* is a container for other elements such as *Tasks*, *Events* and *ControlFlows* in the activity diagram.

The postcondition part is rather complex. It states that an *Initial Node* in the UML activity diagram is created, which is followed by an *AcceptEventAction*. The trigger of this action is a *Signal Event*. A *Control Flow* is added between the *Initial Node* and *Accept Event Action* to maintain the sequential dependency between both. Due to the space limitations, it is not possible to discuss all the elements and their respective mapping rules in this paper. However, Table I depicts a subset of the mappings between the elements of BPMN collaboration diagram and UML activity diagram. A complete set of mapping rules is available at our project website [20]. After each test path is mapped to an activity diagram, the test case definitions are added to the *Test Context* class as test operations in the test architecture class diagram. The interface of these operations can be generated by analyzing input data required by each path.

Table I
MAPPING SUBSET:COLLABORATION ONTO ACTIVITY DIAGRAM

Collaboration Elements	**Activity Elements**
Pool	ActivityPartition
SequenceFlow	ControlFlow
Gateway	DecisionNode
EmptyTask/None	Action
SendTask when target is a non-service Task or Pool	SendSignalAction
SendTask when target is a service Task	CallOperationAction
ReceiveTask	AcceptEventAction
A Task calling a ServiceTask	CallOperationAction
EmptyStartEvent/EmptyEndEvent	InitialNode/FinalNode

V. TRANSFORMATION IMPLEMENTATION USING VIATRA

We implemented our test generation approach and the mappings using the model transformation framework VIATRA [15]. VIATRA provides a rule-based language *VIATRA Textual Command Language* (VTCL) for the implementation of model transformation rules. The rule-based structure of VIATRA makes it suitable for the implementation of our mapping rules.

The basic construct of VTCL is a *Graph Transformation Rule* (GT-Rule). A *GT-Rule* is comprised of a precondition part, a postcondition part and an action part. When a precondition pattern of a *GT-Rule* is matched in the source model, it creates an image of the postcondition pattern in the target model. Our mapping rules can be seen as an abstract form of a *GT-Rule* and are easily translatable to the concrete executable *GT-Rules*. The models we use to implement the transformations conform to the meta-models of UML, BPMN, and U2TP.All the meta models are implemented as Eclipse plugins using the EMF framework [23].

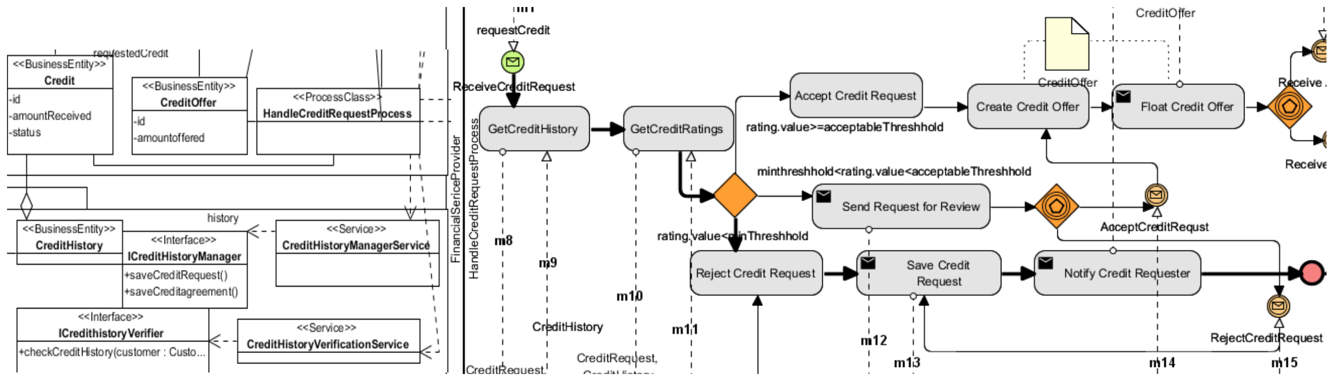
```
gtrule singlePool(inout Pool, inout Activity)={
  precondition pattern isPool(Pool)={
   bpmn.metamodel.bpmn.Pool(Pool);
   neg find isLaneInPool(Pool,Lane);}
  postcondition pattern matchPartitionToPool(VisKind,
        Partition ,Pool,Activity ,String)={
   bpmn.metamodel.bpmn.Pool(Pool);
   uml.metamodel.uml.ActivityPartition(Partition) in
          Activity;
   uml.metamodel.uml.VisibilityKind(VisKind) in Partition;
   uml.metamodel.uml.NamedElement.name(NamedElem,Partition
          ,String);
   uml.metamodel.uml.NamedElement.visibility(Vis,Partition
          ,VisKind);}
  action{
   rename(Partition ,name(Pool));
   setValue(VisKind,"public");}}
```

Listing 3. An excerpt of the *GT-Rule* in VTCL

Listing 3 presents a *GT-Rule* for the mapping *"Pool maps to ActivityPartition"*, as depicted in the first row of Table I. The graph transformation rule, *"singlePool"* shown in Listing 3 takes a *Pool* and an *Activity* as input. The input *Activity* is the base element of the activity diagram meta-model. The rule *singlePool* consists of a precondition pattern and a postcondition pattern. The precondition pattern states that *Pool* is an element in the BPMN meta-model. The second line of the precondition pattern checks if there is a lane inside the pool or not. The postcondition pattern creates an *ActivityPartition* inside the element *Activity*, and instantiates its properties. In the action part of the *GT-Rule*, name of the *Pool* is assigned to the created *ActivityPartition*, and the property "visibilitykind" is specified as public. A prototype Eclipse plug-in implementing the transformations is available at our project website[20].

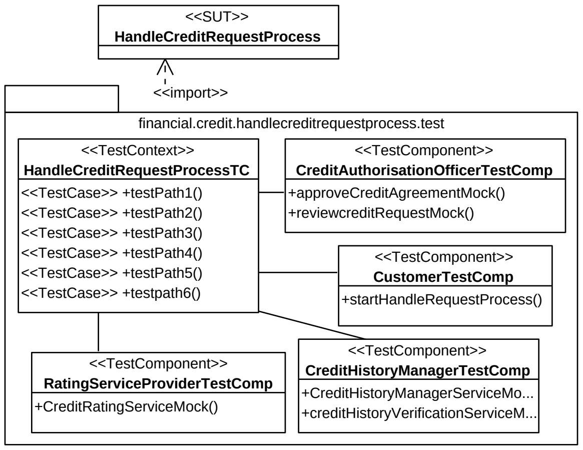
VI. CONCEPT ILLUSTRATION BY EXAMPLE

To illustrate the applicability of our approach, we introduce an example credit request scenario and apply our approach on it. The left side of Figure 3 depicts an excerpt of the class diagram of the credit request application. The diagram contains a class *HandleCreditRequestProcess* with a stereotype «ProcessClass». A part of the collaboration diagram corresponding to this process class is depicted on the right hand side of Figure 3. Figure 4 represents the U2TP test architecture class diagram. The class *HandleCreditRequest-Process* with the stereotype «SUT» represents the system under test, and is derived by applying the mapping rule in Listing 1. The SUT class has an import dependency to the test package *financial.credit.handlecreditrequestprocess.test*



<<BusinessEntity>> Credit
-id
-amountReceived
-status
<<BusinessEntity>> CreditOffer
-id
-amountoffered
<<ProcessClass>> HandleCreditRequestProcess
<<BusinessEntity>> CreditHistory
history
<<Interface>> ICreditHistoryManager
+saveCreditRequest()
+saveCreditagreement()
<<Service>> CreditHistoryManagerService
<<Interface>> ICredithistoryVerifier
+checkCreditHistory(customer : Custo...
<<Service>> CreditHistoryVerificationService
FinancialSericeProvider
HandleCreditRequestProcess
requestCredit
ReceiveCreditRequest
GetCreditHistory
GetCreditRatings
Accept Credit Request
rating.value>=acceptableThreshhold
Create Credit Offer
Float Credit Offer
CreditOffer
Receive
minthreshhold<rating.value<acceptableThreshhold
Send Request for Review
AcceptCreditRequst
rating.value<minThreshhold
Reject Credit Request
Save Credit Request
Notify Credit Requester
RejectCreditRequest
CreditHistory
CreditRequest
m8
m9
m10
m11
m12
m13
m14
m15

Figure 3. An Excerpt of the Credit Request Collaboration and Component Architecture



<<SUT>> HandleCreditRequestProcess
<<import>>
financial.credit.handlecreditrequestprocess.test
<<TestContext>> HandleCreditRequestProcessTC
<<TestCase>> +testPath1()
<<TestCase>> +testPath2()
<<TestCase>> +testPath3()
<<TestCase>> +testPath4()
<<TestCase>> +testPath5()
<<TestCase>> +testpath6()
<<TestComponent>> CreditAuthorisationOfficerTestComp
+approveCreditAgreementMock()
+reviewcreditRequestMock()
<<TestComponent>> CustomerTestComp
+startHandleRequestProcess()
<<TestComponent>> RatingServiceProviderTestComp
+CreditRatingServiceMock()
<<TestComponent>> CreditHistoryManagerTestComp
+CreditHistoryManagerServiceMo...
+creditHistoryVerificationServiceM...

Figure 4. The Credit Request Test Architecture

as defined in the postcondition of the mapping rule in Listing 1.

The classes with the stereotype *TestComponent* shown in Figure 4 are the test components, which are required to test the process *HandleCreditRequestProcess*. These test components are derived from the *Pools* and *Lanes* of the collaboration diagram and their corresponding structural specifications defined in the class and component diagrams.Due to the limited space, these pools are not shown in the collaboration illustrated by Figure 3.

An example test component in Figure 4 is the *CreditHistoryManagerTestComp*, which is responsible of processing the message calls sent by the tasks *GetCreditHistory* and *SaveCreditRequest* in the *HandleCreditRequestProcess*. The test components are derived from the pools receiving the messages.The test component *CreditHistoryManagerTestComp* mocks the two services *CredithistorymanagerService* and *CredithistoryVerificationService*. These services are shown as service classes in the class diagram shown in figure 3. The test package also contains the *Test Context* class and the required test components. The test context class contains six test cases, which are generated by applying the test generation steps explained in the Section IV-B. One of such test paths is depicted with the bold control flow in Figure 3. The UML activity diagram test case corresponding to this test path is depicted in Figure 5.

In the activity diagram, the service tasks are assigned to *CallOperationAction*, send tasks are assigned to the *SendSignalAction*, and receive tasks are assigned to the *AccepteventAction*.These mappings are consistent with the mappings shown in Table I. The sent and received messages are assigned to the ports of the respective actions in the UML activity diagram test case. For the sake of simplicity, no data flows were shown in figure 3, and no *ActivityPartitions* are shown in the test case shown in Figure 5.

VII. Acknowledgment

We thank Stefan Gross for developing the prototype discussed n section V as part of his masters thesis.

VIII. Conclusion and Future Work

In this paper, we presented a holistic model-driven test generation approach for testing business process-based information systems. For the test generation, first we transform the elements of UML class and component diagrams into U2TP test architecture elements. After that, we generate the test behavior by transforming the BPMN collaboration diagrams into test paths and then transforming these test paths into U2TP activity diagram test cases. To do this, we analyzed the elements of BPMN collaboration diagrams, UML class and component diagrams, and U2TP test models to identify the corresponding elements and then developed mapping rules based on them. We implemented the mapping rules in a prototype, using model transformations provided by the VIATRA framework. One of the benefits of our approach is that we used models for the test generation as well as the test specification, which results in support for early testing and better test documentation. A limitation of our work is that at present we do not support the test data by analyzing the data elements in the BPMN models and their corresponding structures. However, this is part of our ongoing work and we plan to address this issue in the future.

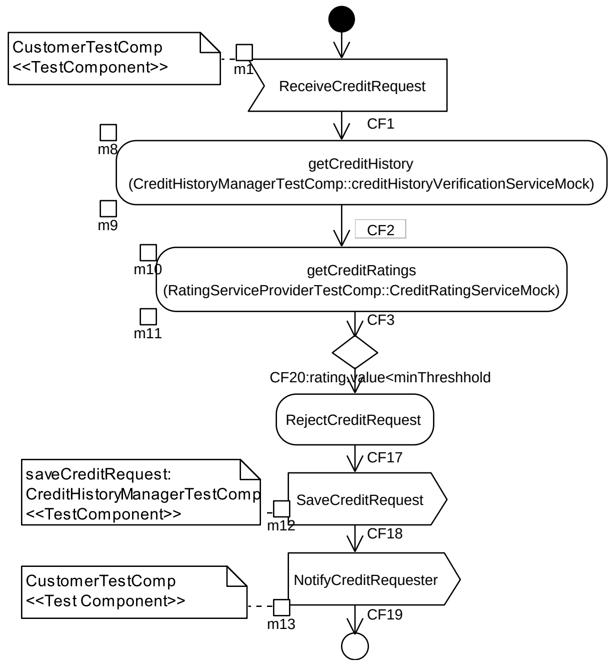


Figure 5. A Test path of the Credit Request Process as UML Activity Diagram Test Case

References

[1] T. Bakota, A. Beszédes, T. Gergely, M. I. Gyalai, T. Gyimóthy, and D. Füleki, “Semi-automatic test case generation from business process models,” 11th Symposium on Programming Languages and Software Tools, 2009.

[2] J. Garcia-Fanjul, J. Tuya, and C. de la Riva, “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, pp. 83–94.

[3] Y. Zheng, J. Zhou, and P. Krause, “A model checking based test case generation framework forweb services,” in *Proceedings of the International Conference on Information Technology*, 2007, pp. 715–722.

[4] E. Werner, J. Grabowski, S. Troschutz, and B. Zeiss, “A TTCN-3-based web service test framework,” in *Workshop on Testing of Software - From Research to Practice*, 2008.

[5] Z. Dai, “An approach to model-driven testing: Functional and real-time testing with UML 2.0, U2TP and TTCN-3,” Ph.D. dissertation, TU Berlin, 2006.

[6] A. Stefanescu, S. Wieczorek, and A. Kirshin, “MBT4Chor: a Model-Based testing approach for service choreographies,” in *Model Driven Architecture - Foundations and Applications*, 2009, pp. 313–324.

[7] Q. Yuan, “A model driven approach toward business process test case generation,” *10th International Symposium on Web Site Evolution*, pp. 41–44, 2008.

[8] M. Penker and H. Eriksson, *Business Modeling With UML: Business Patterns at Work*, 1st ed. Wiley, Jan. 2000.

[9] D. Auer, V. Geist, W. Erhart, and C. Gunz, “An integrated framework for modeling Process-Oriented enterprise applications and its application to a logistics server system,” in *2nd International conference on Logistics and Industrial Informatics*, Sep. 2009, pp. 1 –6.

[10] OMG, “UML2 Testing Profile,” OMG Docment Formal/ptc/2011-07-20, Object Management Group, July 2005. [Online]. Available: http://www.omg.org/spec/UTP/1.0/

[11] ——, “Business Process Model and Notation (Beta 2),” Object Management Group, June 2010. [Online]. Available: http://www.omg.org/spec/BPMN/2.0/Beta2/PDF/

[12] ——, “UML 2.0 superstructure specification,” OMG document formal/2007-02-03, Object Management Group, 2007. [Online]. Available: http://www.omg.org/docs/formal/07-02-03.pdf

[13] N. Koch, A. Kraus, C. Cachero, and S. Meliá, “Integration of business processes in web application models,” *J. Web Eng.*, vol. 3, no. 1, pp. 22–49, 2004.

[14] 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 *6th Central and Eastern European Software Engineering Conference*, 2010, pp. 79 –85.

[15] VIATRA2, “Viatra2, visual automated model transformations framework,” Availabe at: http://www.eclipse.org/gmt/VIATRA2/, June 2011.

[16] J. Yan, Z. Li, Y. Yuan, W. Sun, and J. Zhang, “BPEL4WS Unit Testing: Test Case Generation Using a Concurrent Path Analysis Approach,” in *17th International Symposium on Software Reliability Engineering*, 2006, pp. 75 –84.

[17] Y. Yuan, Z. Li, and W. Sun, “A graph-search based approach to bpel4ws test generation,” in *International Conference on Software Engineering Advances*, 2006, p. 14.

[18] W.-L. Dong, H. Yu, and Y.-B. Zhang, “Testing BPEL-based web service composition using high-level petri nets,” in *10th IEEE International Enterprise Distributed Object Computing Conference*, 2006, pp. 441 –444.

[19] A. Heinecke, T. Griebe, V. Gruhn, and H. Flemig, “Business process-based testing of web applications.” ser. Lecture Notes in Business Information Processing, vol. 66, 2010, pp. 603–614.

[20] “B2u project website,” Available at: http://www.theoinf.tu-ilmenau.de/ qurat/B2UProject/subsite/index.htm, Last Accessed: 09.11.2012, 2012.

[21] J. Sneyers, T. Schrijvers, and B. Demoen, “Dijkstras algorithm with fibonacci heaps: An executable description,” in *20th Workshop on Logic Programming*, 2006, pp. 182–191.

[22] B. Awerbuch, “A new distributed depth-first-search algorithm,” *Information Processing Letters*, vol. 20, no. 3, pp. 147 – 150, 1985.

[23] EMF, “Eclipse Modeling Framework ,” Last Accessed: 09.11.2012. [Online]. Available: http://www.eclipse.org/modeling/emf/?project=emf