Multi-level Tests for Model Driven Web Applications

Piero Fraternali[1] and Massimo Tisi[2]

[1] Politecnico di Milano, Dipartimento di Elettronica e Informazione
Milano, Italy
piero.fraternali@polimi.it
[2] AtlanMod, INRIA & Ecole des Mines de Nantes
Nantes, France
massimo.tisi@inria.fr

Abstract. Model Driven Engineering (MDE) advocates the use of models and transformations to support all the tasks of software development, from analysis to testing and maintenance. Modern MDE methodologies employ multiple models, to represent the different perspectives of the system at a progressive level of abstraction. In these situations, MDE frameworks need to work on a set of interdependent models and tranformations, which may evolve over time. This paper presents a model transformation framework capable of aligning two streams of transformations: the forward engineering stream that goes from the Computation Independent Model to the running code, and the testing stream that goes from the Computation Independent Test specification to an executable test script. The "vertical" transformations composing the two streams are kept aligned, by means of "horizontal" mappings that can be applied after a change in the modeling framework (e.g., an update in the PIM-to-code transformation due to a change in the target deployment technology). The proposed framework has been implemented and is under evaluation in a real-world MDE tool.

1 Introduction

In Model Driven Engineering (MDE), models incorporate knowledge about the application at hand, at a specific level of abstraction. An MDE environment usually comprises several models, connected by semantic relationships. The knowledge embodied in more abstract models is primarily used for forward engineering, that is, the progressive refinement towards models that are more concrete, and eventually towards the final implementation code. For instance, a well-known way of structuring the refinement process is provided by the Model Driven Architecture (MDA)[21] scheme that distinguishes three main levels of abstraction: Computation Independent Models (CIM), Platform Independent Models (PIM) and Platform Specific Models (PSM). The translation from one level to the following can be performed manually or, in generative software engineering, it can be driven by automatic transformations between models.

B. Benatallah et al. (Eds.): ICWE 2010, LNCS 6189, pp. 158–172, 2010.

Models have a range of application that goes beyond code generation [23]. In particular, several works use MDE as a support to testing [22,5,20,6]. In these works we can identify a common approach, consisting in producing a set of test cases from the analysis of a CIM or PIM and in executing them on the running software. When the process is automated, model transformations are used to build the testing artifacts. In these approaches the main challenge is in producing tests that have the highest chance of revealing errors.

In this paper we focus on the problem of defining, managing and executing test cases for applications modeled at multiple levels of abstraction, in automated MDE environments. We ignore the issue of generating the right test cases (for that topic, we refer the reader to the aforementioned papers), and concentrate on the problem of aligning the CIM-PIM-PSM transformation stream of the code generator to the parallel CIT-PIT-PST[1] stream used to produce and maintain test cases. In multi-level environments in which a certain number of CIMs, PIMs and PSMs have a parallel lifecycle, this problem is rather complex. For example, if one of the forward engineering transformations is updated, it is not obvious how to modify the "parallel" test transformations.

We introduce a model-transformation framework for test cases, and a prototype implementation of this framework that relies on concrete modeling languages: the CIM level consists of BPMN models [27], which express the multi-actor processes served by the application; at the PIM level, we use a specific Web application DSL, WebML [10], which expresses the data, business logic and front-end interface of the Web/SOA application that supports the business processes specified at the CIM level. The CIM to PIM to PSM transformation is provided by a commercial tool suite, called WebRatio [3]. The paper concentrates on the chain of transformations for producing tests and its contribution can be summarized as follows:

- suitable metamodels for representing test cases for Web applications at different levels of abstraction (CIM/BPMN and PIM/WebML);
- a mechanism for supporting automatic alignment of the Platform Independent Test specifications after the manual refinement of a partial CIM to PIM transformation;
- a mechanism to co-evolve the PIM to PSM transformation and the parallel PIT to PST transformation, which ensures that tests are automatically regenerated after the regeneration of the application code for a different platform.

The rest of the paper is organized as follows: Section 2 presents the case study used throughout the paper; Section 3 overviews the framework and describes the testing metamodels; Sections 4 illustrates how to keep test representations synchronized, when models and transformations evolve; Section 5 compares our contribution to the related work; Section 6 draws the conclusions.

[1] Computation Independent Test (CIT), Platform Independent Test (PIT), Platform Specific Test (PST).

2 Case Study

As a case study, we consider a simple application that manages the creation of an expense report by an employee and all the following approval steps.

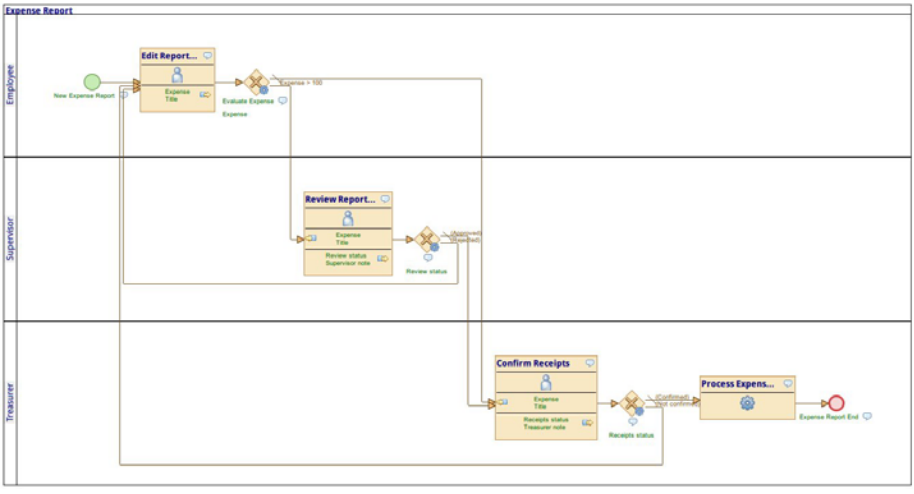

Fig. 1. BPMN model of the Product Catalog Application

The case study is first modeled at the CIM level by the BPMN model shown in Figure 1. The model has three lanes, representing the actors that take part to the process, i.e. employee, supervisor and treasurer. The application process starts with the *Edit Report* activity that allows the employee to insert the title and the total amount of the expense. The values are stored in the *Title* and *Expense* parameters and evaluated by a condition in the following gateway. If the expense exceeds 100$ then the process flow goes to the supervisor, while a smaller expense is directly managed by the treasurer. In the first case the supervisor checks the report parameters and sets the *Review status* parameter to "Approved" or "Rejected" (*Review Report* activity). If the value is "Approved" then the flow goes to the treasurer, otherwise the rejection is sent back to the employee. Finally the treasurer has to set the "Receipts status" parameter in the *Confirm Receipt* activity, and explain in the "Treasurer notes" parameter the reasons of this choice. If the value of "Receipts status" is "Confirmed" then the expense report is directly sent to the company account system by the *Process Expense* activity.

The model in Figure 1 is created using the BPMN modeling tool in the WebRatio toolsuite [3]. The toolsuite can automatically translate this process model into a Web application model, represented in the WebML language. The generated WebML application is specified on top of a data model by means of one or more *site views*, comprising *pages*, possibly clustered into *areas*, and containing various kinds of data publishing components (*content units* in the WebML jargon) connected by *links*. The WebRatio generator from BPMN to WebML creates:

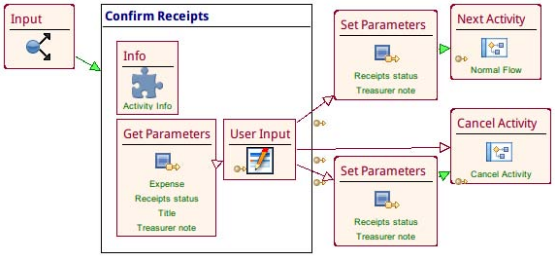

Fig. 2. Generated WebML hypertext for the Confirm Receipt activity

- a generic data model for the process execution (with entities like *User*, *ActivityInstance*, *Parameter*),
- two standard site views for authentication and process administration,
- one site view for each lane, for the orchestration of the process,
- one module, i.e. a composable partial site-view, for each activity.

Figure 2 shows for example the WebML translation of the *Confirm Receipt* activity. The *Input* unit represents the entry point of the module. The units have outgoing links, which enable navigation and parameter passing. For example, *Input* activates the *Confirm Receipts* page. The page displays the name of the current activity (by the *Info* unit) and retrieves from the application context the needed parameters by the *Get Parameters* unit. The retrieved parameters are *Title* and *Expense*, needed to evaluate the expense report, but the unit also looks for pre-existing values of *Receipt Status* and *Treasurer notes*, that could have been saved by the treasurer in a previous access to this activity. The link outgoing from *Get Parameters* communicates these values to the *User Input* unit that denotes a data entry form. The parameter values are used to pre-fill four corresponding input fields in the form. The user can edit these values and select one of the three outgoing navigable links. He can 1) store the new values of the parameters and pass to the *Next Activity*, giving the control to the corresponding module or 2) cancel the activity without touching the current value of the parameters or 3) save a temporary value for the *Receipt status* and *Treasurer notes* before cancelling the activity.

The WebML model enriches the BPMN process scheme with operational details. For example, the parameter saving functionality is not explicitly defined by the BPMN model but added automatically to the WebML design by the WebRatio generator. Furthermore, the application developer can manually modify the generated WebML model to add collateral functions not described at the CIM level. For example, it could be useful to give to the treasurer the possibility to review the history of past expense reports before taking a decision. To model this functionality the designer edits the WebML model to obtain the diagram in Figure 3. In the final model the treasurer can navigate a new link that takes him to the Expense Log page, showing a table of all the registered expenses (by the

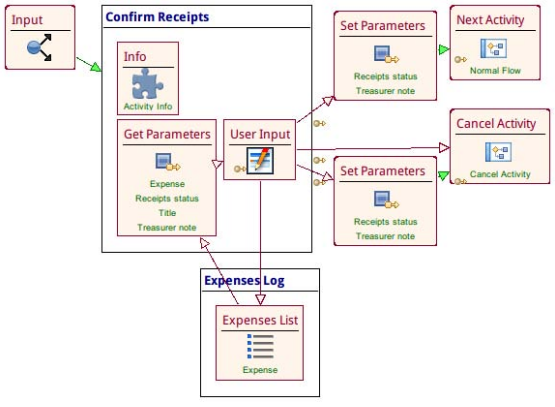

Fig. 3. Edited WebML hypertext for the Confirm Receipt activity

Expenses List index unit). From this page the user has to return to the Confirm Receipts page, to take a definite decision.

The final WebML PIMs can be automatically translated into a running application, by means of the WebRatio code generator. The generator produces all the implementation artifacts for the Java2EE deployment platform, exploiting the popular MVC2 Struts presentation framework and the Hibernate persistence layer. In particular, the View components can utilize any rendition platform (e.g., HTML, FLASH, Ajax), because the code generator is designed to be extensible: the generative rules producing the components of the View adopt a template-based style and thus can incorporate examples of layout for the various WebML elements (pages and content units) coded in arbitrary rendition languages. The user provided templates (like the main code generator) are written in the Groovy language, which allows a Java-like syntax encapsulated into scriptlets, to create template-based transformations.

Once the generated application has been deployed, the application models can be exploited to generate sets of testing sessions, to optimize some testing accuracy metric, e.g., by using techniques like the ones in [9]. For instance, the testing policy could require all the paths of the BPMN model to be exercised by at least one test. A testing session generated at the CIM level is expressed using the concepts that appear in the BPMN model. In the subsequent sections, we will use the following example:

```
1. an employee starts the process instance
2. the employee creates the report named "Car Rental" for 50$
3. a treasurer receives a report named "Car Rental"
4. the treasurer accepts the receipt
5. the expense report is sent to the company account system
```

From this high-level test we want to generate the correspondent platform-independent and executable versions.

3 Model-Driven Test Representation

Figure 4 shows an overview of the models involved in our framework. For each one of the MDA abstraction levels, we define both a metamodel of the Web application and a metamodel of the test case:

- at the CIM level, the modeling language is BPMN and the Computation Independent Tests (CITs) are modeled in our BPMN-Test metamodel;
- at the PIM level, the modeling language is WebML and the Platform Independent Tests (PITs) are modeled in our WebML-Test metamodel;
- at the PSM/code level, Platform Specific Tests (PSTs) are Web navigations represented as scripts of a Web testing suite (e.g. the Canoo WebTest system[2]).

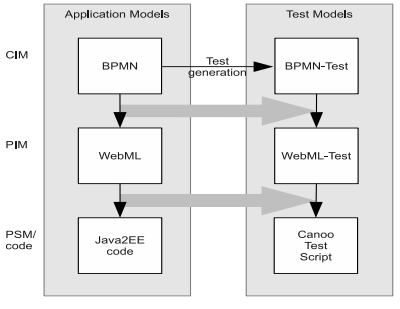

Fig. 4. Overview of the transformation framework

In the design of the test case metamodels we seek maximum simplicity and extensibility. The metamodels are based on a common core. They comprise a container class *TestSuite* that can be decorated with information about the application configuration. *TestSuite* contains multiple *Tests* composed by ordered sets of *Steps*. Each *Step* specifies the identifier of an application session, e.g. useful for distinguishing actions performed by concurrent users of the system. *Step* is specialized in two abstract classes that have to be subclassed for each concrete test case metamodel: an *ActionStep* activates some elements of the application model, referenced by an identifier, while an *AssertionStep* represents the evaluation of a predicate over the application state. Given an application domain, new *ActionStep* or *AssertionStep* subclasses can always be defined for domain-specific tests. Excerpts from the BPMN-Test and WebML-Test metamodels are shown in Figures 5 and 6. It is easy to identify in the two metamodels a reference to the specific concepts of the respective application models. A BPMN-Test model allows one to initiate a process instance, follow its links and insert values in the process instance variables. The only assertion provided checks the value of a process instance variable, but new assertions can be introduced by defining new subclasses. Finally a *Not* assertion allows one to negate the truth value

[2] `http://webtest.canoo.com`

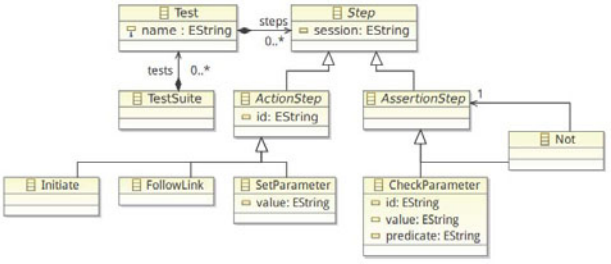

Fig. 5. BPMNTest Metamodel

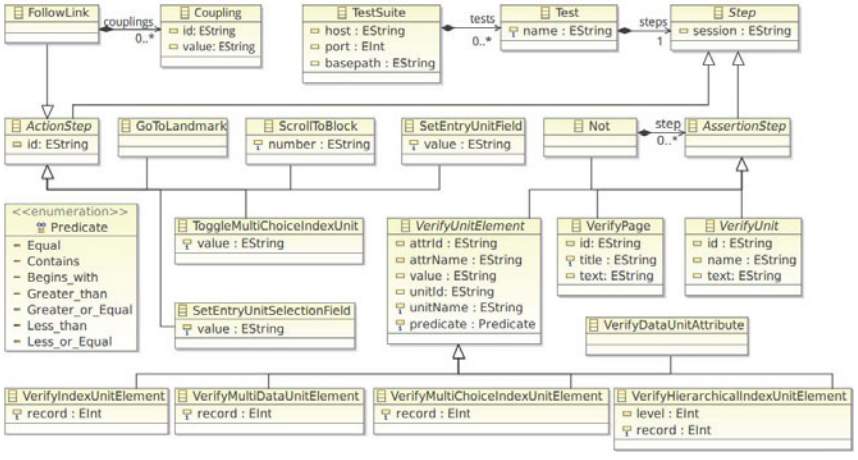

Fig. 6. WebML-Test Metamodel

of a referenced assertion. The WebML-Test metamodel is more complex, as expectable. *ActionSteps* include the activation of links (providing an optional set of correspondent parameter couplings), of landmark elements[3], of input fields, selections and scrolling. *AssertionSteps* allow one to check information about: 1) the current page (i.e. id, title and contained text), 2) currently visualized units (i.e. id, name and contained text), 3) a single element of a currently visualized unit, provided the id of the unit, of the attribute and the numerical coordinates of the record in the table or tree represented by the unit (e.g., to check that the third element of an index contains a given value).

The test models are associated to a default semantics, according to which the test is successful if: 1) it is possible to execute all the *ActionSteps*, 2) no *AssertionStep* evaluates to false. Going back to our case study, the described BPMN test scenario is a single *Test* with this sequence of *Steps* elements:

```
1a. initiate (session='1', id='lane1')
2a. setParameter (session='1', id='title', value='Car Rental')
```

[3] Landmarks are global navigation targets, like the home page or the entry pages of main application areas.

```
2b. setParameter (session='1', id='expense', value='50')
3c. followLink (session='1', id='link2')
3a. initiate (session='2', id='lane3')
3b. checkParameter (session='2', id='title', value='Car Rental',
    predicate='equal')
3c. checkParameter (session='2', id='expense', value='50',
    predicate='equal')
4a. setParameter (session='2', id='Receipts status', value='true')
4b. followLink (session='2', id='link4')
4c. followLink (session='2', id='link5')
```

4 Synchronizing Test Representations

The vertical arrows in Figure 4 represent refinement transformations. Transformations in the right column refine the specification of the test case. A BPMN test case, conforming to the BPMN-Test metamodel is translated in one or more WebML test cases, conforming to the WebML-Test metamodel. A model of a WebML test is translated into a Web testing script.

The horizontal arrows represent the synchronization mechanisms between application transformations and test transformations that is the main contribution of this paper. It is important to remark that this kind of synchronization is not always necessary in generic model-driven testing. If the application transformation is *complete*, i.e. it generates automatically the whole target model, and *fixed*, i.e. it does not change over time, then no synchronization mechanism is required. This is a common case in transformation environments. Several applications are based on a single stable transformation that refines an input model, generating a complete output model. Notable examples are compilers, optimizers, analyzers. In all these cases the transformation logic is fixed, and a corresponding fixed transformation can be easily written also for the test cases. In the cases in which the main transformation is not complete (i.e. it is *partial*) or not fixed (i.e. it is *user-defined* or *evolving*), a synchronization mechanism becomes necessary. In Section 4.1 we propose a solution for partial transformations, using the case study BPMN to WebML. Section 4.2 investigates applications with user-defined and evolving transformations using the case study WebML to code.

4.1 Synchronizing Tests with Partial Transformations

Sometimes the main model transformation is *partial*, meaning that it generates only a skeleton of the target model, leaving to the modeler the task of completing the modeling artifact. In these cases, the abstract test case can be easily translated into a skeleton of the concrete test case by a fixed transformation. However, only by means of a synchronization mechanism it is possible to deal with testing the manual additions to the application model.

As exemplified in Section 2, the transformation BPMN to WebML is a case of partial transformation, since the developer may manually intervene on the generated model to add complementary activities to the main workflow. Hence,

the transformation between CIT and PIT can't be directly derived by analyzing the CIM-to-PIM transformation. While creating the WebML test we need to take in account also the current state of the WebML model.

In our case study, each *Step* of the BPMN test sequence can be easily translated in one or more *Steps* for testing the generated WebML model. For instance, steps 3b-4b can be transformed automatically into the following steps over the WebML module in Figure 2:

```
3b. verifyEntryUnitElement (session='2', unitID='enu12',
    attrName='title', value='Car Rental', predicate='equal')
3c. verifyEntryUnitElement (session='2', unitID='enu12',
    attrName='expense', value='50d', predicate='equal')
4a. setEntryUnitField (session='2', id='fld12', value='yes')
4b. followLink (session='2', id='ln21')
```

However, manual modifications of the WebML application model could impact the previously generated test set in two ways:

- the test could loose the completeness property, due to the occurrence of new paths in the modified WebML model that would not be subject to test;
- the test could be no longer applicable to the new model, e.g., there could be no link ln21 exiting from the entry unit enu12.

For example, while the above-mentioned test sequence is still applicable to the model in Figure 3, it would not test for errors in the presentation of the list of past expenses. The solution for making the BPMN-Test to WebML-Test transformation aware of manual modifications to the application model is shown in Figure 7. The CIT to PIT transformation is given a composite structure, made of two steps: **T1a.** A first set of *standard CIT to PIT rules* implements the default refinement from BPMN-Test to WebML-Test, following the same logic used for the forward engineering from BPMN to WebML models. These transformation rules match the elements of the BPMN test sequence, retrieve additional information from the BPMN model, if necessary, and apply a default translation to each test step, mirroring the logic in the forward engineering. **T1b.** A second set of *PIT extension rules* implements an algorithm for checking test executability and for extending test coverage to the newly introduced parts of the application model. The algorithm analyzes each test step generated by the standard CIT to PIT rules and checks it with respect to the modified WebML model. If the step is not executable from the modified WebML model or, in case of *followlink* test steps, if the new model presents alternative paths, the algorithm updates the test with a policy that depends on the desired depth of the testing. Otherwise, the step is simply copied to the result. In our prototype, the test update policy chooses non deterministically an alternative link to follow with respect to the non-executable link, or a subset of the newly introduced navigation paths. The algorithm stops when: a) all the BPMN-Test steps have been analyzed (success) or b) there is no way to proceed with the test extension or the number of steps in the test exceeds a threshold (failure).

In the case study, T1a would generate the steps 3b-4b shown above. T1b would copy 3b-4a to the output script, and would start the coverage algorithm

for the step 4b, since new alternative paths have been added to the application model, so to add, in at least one of the updated test cases, the path towards the manually added page that shows the expense list.

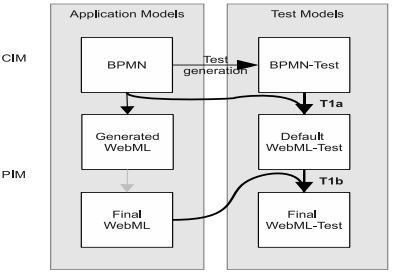

Fig. 7. Transformation framework implementation (BPMN to WebML)

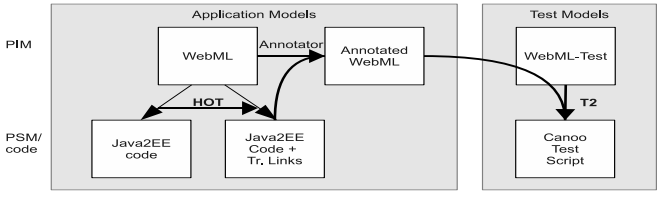

Fig. 8. Adaptation framework to align the PIT to PST transformation

The *PIT extension rules* are able to handle any manual modification to the application model, with the only limitation to elements removal (e.g., the deletion of the User Input unit). Units can instead be repositioned in the model, other units can be interposed between them, and the topology of links can be altered.

4.2 Synchronizing Tests with User-Defined and Evolving Transformations

If the application transformation is *user-defined* or it is *evolving* in time, then adaptation of the PIT to PST transformation is required. This is the typical case of Web applications, in which the PIM can be translated into code in several ways, depending on a number of implementation choices. Notably, on most model-driven Web environments, the implementation transformation depends on the presentation style defined by the graphical designer, which is subject to frequent changes. The code generation process can be seen as a model-to-model transformation that maps an input model at PIM level (e.g., the WebML model of the application) into a an executable model (e.g., the Java2EE code). It is normally a lossy transformation: since its purpose is to produce the code to be actually executed, no extra information is added to the output model and the links between the input and output artifacts are lost.

The transformation from WebML to code is organized into three sub-transformations. The *Layout Transformation* generates a set of JSP pages (one for each page of the WebML model) and miscellaneous elements required by the target platform: Struts configuration (i.e. the controller in the Struts MVC architecture), localization bundles, and form validators. The *Business Logic Transformation* generates a set of XML files (logic descriptors) describing the run-time behavior of the elements of the source model, mainly pages, links, and units. In addition, this transformation produces secondary artifacts, such as the access/authentication logic. The *Persistence Transformation* produces the standard Hibernate artifacts: Java Beans and configuration mapping (one for each entity of the source model) as well as the overall database configuration.

The sub-transformations are based on Groovy. Being the output a set of structured XML and JSP/HTML files, the Groovy generators use a template-based approach: each sub-transformation comprises templates similar to the expected output (e.g., XML or HTML) enriched with scriptlets for looking-up the needed elements of the source model.

The adaptation problem to be solved occurs whenever the code generation produces an implementation with a different way of performing a test step. In this case, also the testing scripts generated from the PIT need to be updated, to automatically align the test session to the updated implementation.

For example, continuing the case study from the previous section, step 4b requires the test to follow a WebML link (ln21) outgoing from an entry unit. The designer may re-generate the application code with a new Groovy template, which alters the presentation of the unit: link 21 could be rendered differently, e.g. as a button instead of an HTML anchor tag. The different rendering could require different activations from the physical test script. For example, in the Canoo WebTest suite, scripts are represented as XML files and links and buttons are activated by specifying different tags, respectively `<clickLink xpath="..."/>` `<clickButton xpath="..."/>`, where the *xpath* attributes is filled by the PIT to PST transformation, in order to point to the correct link or button. In principle, since one cannot make assumptions about the PIM to PSM transformation, which can incorporate any arbitrary code generation rule, the adaptation framework should be able to analyze the code of the transformation itself to detect the new code generation rules. However such an analysis would be remarkably complex. For this reason we propose approach to synchronize the PIT to PST transformation with an evolving PIM to PSM transformation, which relies only on the generated code, and not on the internal structure of the PIM to PSM transformation code. Figure 8 pictorially illustrates the framework.

The key to such a solution is the a posteriori explicitation of the relationship between PIM concepts and the PSM primitive used to render them; this task is performed by an *Annotator* transformation, which enriches the WebML model with the references to the PSM concepts. For the Annotator to remain generic (i.e., not depend on the target implementation platform) another step is required: being able to trace each model concept to the (arbitrary) implementation code produced by the PIM to PSM transformation. This problem is solved by a Higher

Order Transformation (HOT), which automatically weaves traceability links into the PIM to PSM mapping. Therefore, the control flow of the adaptation framework goes as follows: 1) the designer applies the PIM to PSM transformation to generate the code, which may invalidate previous test cases; 2) the framework uses the HOT to mutate the PIM to PSM transformation and produce an augmented PIM to PSM mapping that creates traceability links; 3) the augmented PIM to PSM transformation is executed to produce an augmented implementation code with embedded traceability links; 4) the Annotator transformation uses the PIM (WebML model) and the PSM (J2EE code) augmented with traceability links and produces an annotated PIM model, in which the relationship between PIM concepts and their PSM rendition is made explicit and declarative; 5) the PIT to PST transformation is parametric and exploits the information in the annotated PIM to produce a test case that mirrors the platform dependent primitives used to render the PIM concepts. 6) the test cases automatically generated in this way can be run against the new application implementation.

In Step 2, we exploit our previous work on traceability weaving [15], and we implement an extended version of our Higher Order Transformation (HOT) for traceability. A HOT is a transformation that acts on another transformation, in our case on the transformation used for generating the code. Adding traceability to the generative framework requires preserving the relationship between the elements of the input model and the elements of the output model derived from them. The input of the HOT is the M2M transformation that produces the implementation code. This transformation can be seen as a model, represented by the chosen transformation language (Groovy, in our case study). The output is another transformation, derived by extending the input model with extra elements (additional code generation rules and templates) for producing the traceability links in the implementation code. The HOT must apply to the relevant original transformation rules and produce extended rules such that: 1) they generate the same output elements as the original rules; 2) they add the needed traceability links to the output. The HOT takes only the layout subtransformation in input, because this is the only one that produces the View elements exercised by the testing script. The traceability links are stored within presentation-neutral, transparent elements (e.g., HTML DIV elements) added to the View artifacts of the output model (namely, the JSP pages).

Once the traceability links are stored into the output code, the *Annotator* parses each element of the WebML model, looks for the associated element in the generated code and adds an annotation to the WebML element (e.g. it would add "type=button" or "type=link" to ln21). Finally, the PIT to PST transformation translates WebML tests into Canoo tests. T2 is parametrized by the element types stored in the annotated WebML model.

The HOT has been implemented using the ATL language and the AmmA [7] framework. To integrate the Groovy language in the transformation framework, a Groovy metamodel has been developed extending the JavaAbstractSyntax metamodel provided by the MoDisco project [1]. The Annotator has been implemented in Java and the PIT to PST transformation is written in ATL.

To summarize, the design of the proposed transformation scheme has the following benefits:

- thanks to the HOT approach the user can freely develop a Groovy template for the website generation;
- the template analysis logic is contained in the HOT and Annotator, and it is kept separated from the test generation logic of T2;
- the template analysis is remarkably semplified by the fact that instead of interpreting the Groovy code, the Annotator has to interpret only the result of this code, i.e. the HTML/JSP.

The main limitation of our current approach is the supposed one-to-one relationship between the PIM and the PSM model (i.e. one PIM element translates into one PSM element, with an arbitrary internal complexity). While this assumption is verified in most WebML applications, an extended version of the algorithm could be advisable for more complex cases.

5 Related Work

The three-layers parallel transformation flow in Figure 4 is first introduced in [13] and the model transformations that compose it are studied in several works, as surveyed in [24]. One of the most popular tasks in this area is test script generation from application requirements, for which an extensive list of references can be found in [14]. In these approaches requirements are modeled by activity diagrams [16], sequence diagrams [9] or natural text [8]. [24] introduces a Functional Requirement Metamodel similar to our BPMN-Test. Our work differentiates from these in being the only one to investigate the automatic synchronization between refinement transformations of application and test cases.

Similar problems to our framework are addressed in the field of model and transformation co-evolution, for instance in [18], [26] and [11]. While some of the design issues are shared with these works, our proposal addresses the peculiar relationship between the model of an artifact and the model of a test case.

Our framework makes use of traceability links to connect a generated model element with its source and avoid the direct analysis of generation code. Transformation frameworks can address traceability during the design of transformations [12], either by providing dedicated support for traceability (e.g., Tefkat [19], QVT [2]), or by encoding traceability as any other link between the input and output models (e.g., VIATRA [25], GreAT [4]). Traceability links may be encoded manually in the transformation rules (e.g., [19]), or inserted automatically (e.g., [2]). A HOT-based traceability system for ATL is already implemented in [17], where the HOT adds to each original transformation rule the production of a traceability link in an external ad-hoc traceability model (conforming to a small traceability metamodel). The approach that we propose is inspired from [17] and can be used to add traceability support to a language like Groovy, that does not provide any built-in support to automatic or manual traceability links.

6 Conclusions

In this paper we have addressed the problem of managing complex model-driven development and testing environments by automatically aligning model transformations. As an application, we have considered the problem of testing Web applications specified at the CIM level with BPMN and at the PIM level with WebML. The proposed framework consists of four "vertical" transformations (CIM-to-PIM and PIM-to-PSM) applied to the forward engineering and to the production of test scripts, which are kept aligned by two "horizontal" transformations that are capable to reinforce integrity after a change of the WebML model produced from the BPMN process diagram and after the update of the WebML-to-Java transformation that yields the executable application. A prototype of the framework has been implemented in Java and ATL. The ongoing and future work will concentrate on the performance validation of the current prototype on very large projects, on its integration with the WebRatio development tool suite, and on the provision of effective mechanisms for evaluating the coverage of a test set with respect to the CIM, PIM and PSM of the application. As a particularly important direction of work, the illustrated framework could be exploited to promote a Test Driven Development approach for MDE.

References

1. MoDisco home page, `http://www.eclipse.org/gmt/modisco/`
2. QVT 1.0, `http://www.omg.org/spec/QVT/1.0/`
3. Acerbis, R., Bongio, A., Brambilla, M., Butti, S.: Webratio 5: An eclipse-based case tool for engineering web applications. In: Baresi, L., Fraternali, P., Houben, G.-J. (eds.) ICWE 2007. LNCS, vol. 4607, pp. 501–505. Springer, Heidelberg (2007)
4. Agrawal, A., Karsai, G., Shi, F.: Graph transformations on domain-specific models. Technical report, ISIS (November 2003)
5. Baerisch, S.: Model-driven test-case construction. In: ESEC-FSE Companion '07: 6th Joint Meeting on European SE Conf. and the ACM SIGSOFT Symp. on the Foundations of SE, pp. 587–590. ACM, New York (2007)
6. Baresi, L., Fraternali, P., Tisi, M., Morasca, S.: Towards model-driven testing of a web application generator. In: Lowe, D.G., Gaedke, M. (eds.) ICWE 2005. LNCS, vol. 3579, pp. 75–86. Springer, Heidelberg (2005)
7. Bézivin, J., Jouault, F., Touzet, D.: An introduction to the ATLAS model management architecture. Research Report LINA(05-01) (2005)
8. Boddu, R., Mukhopadhyay, S., Cukic, B.: RETNA: from requirements to testing in a natural way. In: Proceedings of 12th IEEE International Requirements Engineering Conference, vol. 4, pp. 244–253 (2004)
9. Briand, L., Labiche, Y.: A UML-based approach to system testing. Software and Systems Modeling 1(1), 1042 (2002)
10. Ceri, S., Fraternali, P., Bongio, A., Brambilla, M., Comai, S., Matera, M.: Designing Data-Intensive Web Applications. Morgan Kaufmann, USA (2002)
11. Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A.: Automating Co-evolution in Model-Driven Engineering. In: 12th International IEEE Enterprise Distributed Object Computing Conference, pp. 222–231 (2008)

12. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: OOPSLA '03 Workshop on Generative Techniques in the Context of MDA (2003)
13. Dai, Z.R.: Model-driven testing with UML 2.0. Computer Science at Kent (2004)
14. Denger, C.M.M., Mora, M.M.: Test Case Derived from Requirement Specifications. Fraunhofer IESE Report, Germany (033) (2003)
15. Fraternali, P., Tisi, M.: A Higher Order Generative Framework for Weaving Traceability Links into a Code Generator for Web Application Testing. In: Gaedke, M., Grissnikalus, M., Diaz, O. (eds.) ICWE 2009. LNCS, vol. 5648, pp. 273–292. Springer, Hiedelberg (2009)
16. Hartmann, J., Vieira, M., Foster, H., Ruder, A.: A UML-based approach to system testing. Innovations in Systems and Software Engineering (1), 12–24 (2005)
17. Jouault, F.: Loosely coupled traceability for atl. In: European Conference on Model Driven Architecture (ECMDA), workshop on traceability (2005)
18. Lammel, R.: Coupled software transformations. In: First International Workshop on Software Evolution Transformations, Citeseer, p. 3135 (2004)
19. Lawley, M., Steel, J.: Practical declarative model transformation with tefkat. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 139–150. Springer, Heidelberg (2006)
20. Li, N., Ma, Q.-q., Wu, J., Jin, M.-z., Liu, C.: A framework of model-driven web application testing. In: COMPSAC '06, Washington, DC, USA, pp. 157–162. IEEE Computer Society Press, Los Alamitos (2006)
21. Miller, J., Mukerji, J., et al.: MDA Guide Version 1.0. 1. Object Management Group, 234 (2003)
22. Pretschner, A.: Model-based testing in practice. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 537–541. Springer, Heidelberg (2005)
23. Stahl, T., Voelter, M., Czarnecki, K.: Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons, Chichester (2006)
24. Torres, A.H., Escalona, M.J., Mejias, M., Gutiérrez, J.: A MDA-Based Testing: A comparative study. In: 4th International Conference on Software and Data Technologies, ICSOFT, Bulgary (2009)
25. Varró, D., Varró, G., Pataricza, A.: Designing the automatic transformation of visual languages. Sci. Comput. Program. 44(2), 205–227 (2002)
26. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, p. 600. Springer, Heidelberg (2007)
27. White, S.A.: Business process modeling notation. Specification, BPMI. org. (2004)