A Higher Order Generative Framework for Weaving Traceability Links into a Code Generator for Web Application Testing

Piero Fraternali and Massimo Tisi

Politecnico di Milano, Dipartimento di Elettronica e Informazione
P.za L. Da Vinci, 32. I-20133 Milano, Italy
{piero.fraternali,massimo.tisi}@polimi.it

Abstract. Model Driven Engineering is extending its reach beyond the generation of code from Platform Independent Models (PIMs), to all the phases of the software life-cycle. This paper presents an approach to exploit PIMs to ease regression testing, whereby developers can record and replay testing sessions and investigate testing failures on the application model, thanks to traceability links automatically inserted in the generated code. The core of the approach is a modified version of the model transformation for code generation, obtained by applying a Higher Order Transformation (HOT), that is a transformation that takes in input a transformation (the original code generator) and produces another transformation (the augmented code generator). The HOT weaves into the code generator additional rules producing traceability clues that help developers link any error to the model features likely to cause it.

1 Introduction

Model Driven Engineering advocates the use of models as the primary artifact of the software life-cycle. Models incorporate the knowledge about the application at hand, independently of the technological platform of delivery. The knowledge embodied in the model is primarily used for forward engineering, that is, the progressive refinement towards the final implementation code. However, models have a range of application that goes beyond code generation [31]. They can be used as documentation, to estimate the size and effort of application development [4][5], and even as a support to testing [9,7,23,26,28].

In the domain of testing, the use of models mostly concentrates on automating the production and execution of test cases, while other activities, like model-based selective regression testing and behavioral result evaluation are less supported [24]. When testing and debugging an application, developers are used to think in terms of the functionality at the source code level, and want to trace any testing failure directly to the source code elements that are most likely to have caused it. In an MDE environment, the link between the occurrence of a testing failure and the source code is not there; developers specify the application at a high level, and the detailed source code is produced by a model transformation.

M. Gaedke, M. Grossniklaus, and O. Díaz (Eds.): ICWE 2009, LNCS 5648, pp. 340–354, 2009.

When a regression test fails, developers should be able to link the failure not to the platform-dependent, low-level code, but to the PIM that they have specified.

This paper presents a framework for addressing the problem of letting Model Driven Engineers manage the testing of their application without exiting the level of abstraction of MDE. The main contributions of the proposed framework consist of:

- An Higher-Order-Transformation (HOT), whereby the model to text transformation that produces the source code of the application from its PIM is modified, so that model traceability clues are automatically weaved into the generated code.
- A Navigation Recorder, whereby the developer can implement a test session as a navigation script. The recorder not only registers the navigation steps of the user, but also encodes correctness assertions automatically, exploiting the model traceability clues weaved into the generated code.
- A Test Session Player, embedded within the same IDE used by the developer for editing the PIM and generating the code, which allows one to modify the model and generate the code, play any previously recorded regression test session, and trace failures back to the PIM elements that have caused them.

The rest of the paper is organized as follows: Section 2 introduces the motivations of this work and presents a case study used throughout the paper; Section 3 illustrates the use of Higher Order Transformations for enabling the production of model traceability clues in a model-to-code transformation; Section 4 presents a browser's extension for recording testing sessions, enabling the automatic production of correctness assertions, and a plug-in extension of a MDE development tool, allowing the seamless integration of change management, code generation and regression testing. Section 5 briefly discusses the implementation work; Section 6 compares our contribution to the related work; Section 7 draws the conclusions.

2 Motivation and Case Study

Regression testing is the activity aimed at detecting software regressions, defined as those situations in which a program functionality that was previously working ceases to do so, as a consequence of a change in the software.

Regression testing is particularly relevant in modern Web development methodologies for several reasons: 1) Web applications are often delivered in short times and are subject to continuous evolution; 2) the enabling technologies are still in motion, which introduces further source of uncontrolled changes; 3) rapid prototyping in the early phase of development is often used, to help the stake-holders compare alternative functionalities.

In Web applications, testing sessions can be encoded as scripts that simulate the user's navigation. Such scripts operate on the platform-dependent realization of the application and reproduce the interaction-evaluation loop typical of Web browsing: the user inputs or selects values using the interface and assesses

the response computed by the system; if this is correct, she proceeds in the interaction.

Navigation can be recorded using a state-of-the-practice *Record & Play* tool. Several such tools exist (e.g. Selenium [29] and TestGen4Web [30]), which implement an event-handler that listens to the events occurring inside the browser and then generate a test script (usually in XML format) that contains one or more assertions to be verified after each navigation step. An example of interaction that could be recorded as a test script is:

1. Go to the Google home page
2. Verify that the page title is "Google"
3. Fill the input form with the string "WebTest"
4. Press the "I'm Feeling Lucky" button
5. Assert that the string "WebTest" must appear in the returned page

Specifying an assertion requires an extension of the browser. The test scripts generated by the navigation recorder can then be executed, using one of several Web test environments available, e.g., Canoo WebTest [12], Cactus [32], HTMLUnit [19] and JWebUnit [21], which replay the test session and verify the assertions, highlighting failures.

The problem of this approach is that the evaluation of the testing session breaks the MDE abstraction level, because the testing sessions are defined in terms of the platform-specific realization of the application, and not at the level of the platform-independent models produced by the designers. This semantic mismatch hampers the task of linking failures back to the model elements that are likely to cause them. Furthermore, the testing sessions based on the realization of the system may depend on technological details and not only on the application functions: for example, an assertion on the page content may be sensible to the specific markup used for rendering the application look & feel. After a change of the presentation, such an assertion would fail, even if the functionality and content of the page are still valid.

The present work aims at supporting the definition and evaluation of test sessions in a MDE context, by:

- providing a (possibly automatic) way to preserve the elements of the conceptual model in the definition of the platform-dependent testing session;
- allowing the user to translate the use cases into navigation sessions without worrying about the presence of the models in the background;
- supporting the execution of regression testing from the replay of navigation sequences, with the possibility for the modeler to inspect the failures and trace their possible causes to the model elements.

The proposed approach is illustrated with respect to an exemplary MDE methodology, based on a Domain Specific Language targeted to Web application development, called WebML [13]. We use WebML to model a simplified Web application, derive testing sessions, generate the code with model-to-implementation traceability links, and perform regression testing with the support of the application model. As a case study, we consider a Product Catalog Web application, for publishing and managing content about furniture. The

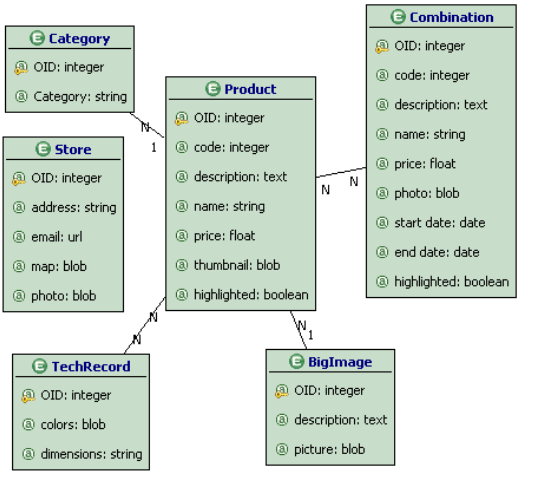

Fig. 1. Data model of the Product Catalog Application

home page contains the product and offer of the day, with a link to access their details, and a form for logging in. From the home page, several other pages are reachable, which allow one to browse the content of the catalog.

Figure 1 shows the data model of the case study, using the simplified E-R notation of WebML; the *Product*, *Combination*, and *Store* constitute the core entities of the data schema; products are clustered in *Categories* and associated with *Images* and a *Technical Record*.

A Web 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*.

Figure 2 shows a fragment of the site view for publishing the content of the Product Catalog application. The *Home* page contains two data publishing components (*data units*), which display selected attributes of a product and of a combination object, and one *entry unit*, which denotes a data entry form. The units have outgoing links, which enable navigation and parameter passing. For example, the *ProductOfTheDay* data unit has an outgoing link that permits the user to reach the *Product Page*, where all the details of the product displayed in the home page are shown. The *Product* page contains further content units, connected to the *Product details* data unit by *transport links* (represented as dashed arrows), which only allow parameter passing and are not rendered as navigable anchors.

The WebML PIMs can be automatically translated into a running application, by means of the WebRatio tool suite [3]. The WebRatio code 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

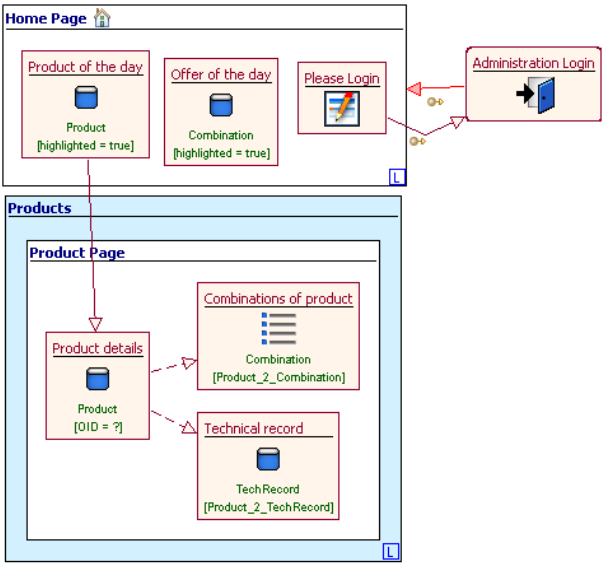

Fig. 2. Site view of the Product Catalog Application

(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.

In the case study, a testing session is expressed at high level using the concepts that appear in the application model. In the subsequent Sections, we will use the following example:

```
1. Go to the Home Page of the Product Catalog
2. Check that the ProductOfTheDay data unit displays the 'Aladdin' item
3. Navigate the outgoing link of the ProductOfTheDay unit
```

Despite its simplicity, the above test can reveal several bugs. Step 1 checks that the Home page is correctly generated and that the communication between the client and the Web server works properly. Step 2 verifies that the item extracted from the database is correct. Step 3 tests the navigation from the Home Page to another Web page, verifying that the link in the Home Page exists and has proper parameters and that that the destination page is computed properly.

With an implementation-oriented approach, an equivalent case must be encoded manually, by navigating the generated HTML pages and asserting conditions on the HTML content (e.g. images, input forms, strings, etc.). Furthermore, the resulting script depends on the graphical layout. For example, step (3) requires evaluating an XPath expression over the page markup: the evaluation of some XPath expressions may change if the page layout is updated (even if functionality does not change).

3 HOT for Weaving Traceability Links into the Code Generation Transformation

One way of circumventing the semantic gap between the application model and the implementation subjected to regression testing is enhancing the implementation with traceability clues, which have no functional meaning but can help linking the occurrence of a failure to the model elements more likely to bear responsibility.

In the context of MDE, this task can be achieved by a Higher Order Transformation, that is, a transformation that acts on the transformation used for generating the code.

Figure 3 pictorially illustrates the HOT framework: the code generation process can be seen as a model-to-model transformation (T1 in Figure 3) that maps an input model at level M1 (the WebML model of the application) into a an executable model (the Java2EE code). T1 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.

Adding traceability to the generative framework of Figure 3 requires preserving the relationship between the elements of the input model and the elements of the output model derived from them. Traceability links can be stored: 1) in the input model; 2) in the output model; 3) in a separate ad-hoc model.

In this paper, we have opted for the second solution, but in our case the transformation T2, which produces an output model comprising the needed traceability links, is dynamically generated from T1. In this way, T1 can still be used to produce the concise and efficient code needed for application execution, but the traceability links needed for regression testing can be obtained by using T2.

With this solution, the major problem is to ensure the consistency between T2 and T1, so that the code produced for testing is exactly equivalent to the production code, modulo the presence of traceability links.

This result can be attained by deriving T2 automatically from T1 by means of a HOT, as depicted in Figure 4.

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

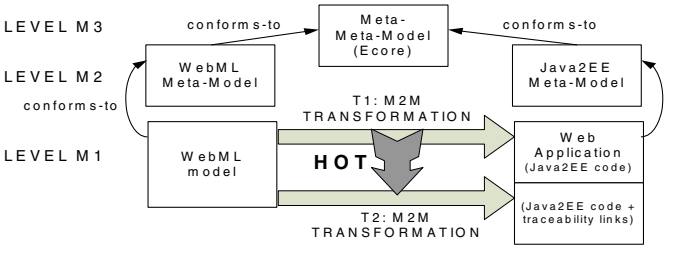

Fig. 3. Using HOT to weave traceability links into the code generator transformation

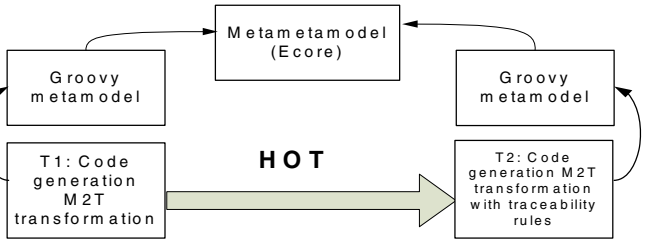

Fig. 4. Input and Output Models of the HOT

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.

Figure 5 shows the internal structure of the input model of the HOT (i.e., the original Groovy code generation transformation).

The transformation 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

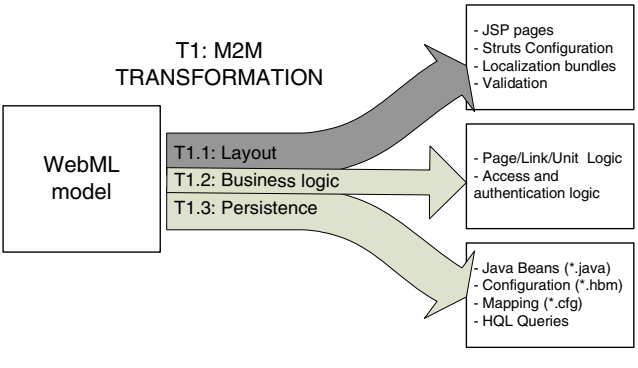

Fig. 5. Structure of T1 transformation

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 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 design of the HOT requires deciding where to store the traceability links in the output model (the Java2EE code) and what information to use for the trace links. In the present version of the HOT, the following design decisions have been taken:

- The traceability link information amounts to the id, name and published values of the content units appearing within the pages of the WebML model, and to the id, name and parameters of navigable links.
- Such traceability links are stored into the View elements of the output model, so that they can be easily added to the recording of the user's navigation.

The above-mentioned design choices entail that the HOT takes only the layout sub-transformation in input, because this is the only one that produces the View elements. 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).

An example can help illustrate the modified behavior of T2 with respect to T1.

The *ProductOfTheDay* data unit of Figure 2 can be represented by the following fragment of the input model[1]:

```
<DataUnit id="dau16" name="Product of the day">
 <Selector id="dau16sel">
   <AttributesCondition attributes="att23"
      name="highlight"/>
 </Selector>
</DataUnit>
```

Transformation T1 (for an XHTML implementation of the View) maps the data unit into JSP code that produces the following mark-up fragment, for a specific product named "Aladdin":

```
<table>
 <tr>  <td>Aladdin</td> </tr>
 <tr>  <td>1500</td> </tr>
 <tr>  <td><img src="images/small_table.jpg"/></td> </tr>
</table>
```

Transformation T2, derived from T1, maps the data unit into JSP code that produces a mark-up fragment enhanced with traceability links:

```
<div id="testUnit id:dau16 name:Product of the day">
 <table>
 <tr>   <td><div id="testAttribute id:att10 name:name
        type:string unitName:Product of the day">
         Aladdin
            </div></td> </tr>
```

[1] WebML has both a visual notation and an XML syntax, and is also equipped with a MOF metamodel; for simplicity, in the example, we use the XML syntax.

```
 <tr>   <td><div id="testAttribute id:att11 name:price
        type:float unitName:Product of the day">
         1500
            </div></td> </tr>
 <tr>   <td><div id="testAttribute id:att12 name:thumbnail
         type:blob  unitName:Product of the day">
         <img src="images/small_table.jpg" />
            </div></td> </tr>
 </table>
</div>
```

The trace clues, inserted in rendition-neutral DIV elements, link the output model (e.g., an XHTML table cell containing the string 'Aladdin") to the input model (e.g., the *name* attribute published by the *ProductOfTheDay* data unit).

To show how the HOT is implemented in a generic way, we illustrate the creation of the traceability link for a content unit. The HOT locates the following instruction in T1:

```
<%printRaw(executeTemplate(templateFile.absolutePath,
    ["params" : unitLayout.parameters,
                    "templateType" : "unit"])) %>
```

The instruction is an explicit call to the Groovy transformation rules for the unit content. It will be translated by the HOT to a new version in T2 that contains an additional DIV element:

```
<div id="testUnit_id:<%=unitId%>_name:<%=unitName%>_">
        <%printRaw(executeTemplate(templateFile.absolutePath,
            ["params" : unitLayout.parameters,
                            "templateType" : "unit"])) %>
</div>
```

This translation is achieved by the following HOT rule:

```
rule UnitLink {
  from
    matched : GroovyMM!Scriptlet (
    matched.statements->recursiveExists(p |
    p.oclIsKindOf(GroovyMM!MethodInvocation) and s.name='printRaw' and
    s.arguments->exists(e | e.oclIsKindOf('GroovyMM!MethodInvocation') and
      e.name='executeTemplate' and
      e.arguments->at(2).oclIsKindOf('GroovyMM!Map') and
      e.arguments->at(2).elements->exists(t |
        t.key='templateType' and
        t.value.oclIsKindOf('GroovyMM!String') and t.value.value='unit')
  to
    div : GroovyMM!Tag (
      name <- 'div', attributes <- Sequence{id}, children <- Sequence{c}),
    id : GroovyMM!TagAttribute (
      name <- 'id', value <-'testUnit_id:<%=unitId%>_name:<%=unitName%>_'
    )
    c : GroovyMM!Scriptlet (
        statements <- matched.statements
    )
}
```

The HOT rule matches any Groovy scriptlet that prints the result of an executeTemplate call to a unit template, i.e. a call with a parameter *templateType = 'unit'*. The output pattern of the rule is a *Tag* named *div* containing a *TagAttribute* named *id* representing an encoding of the traceability link. The matched scriptlet is finally copied as a child of this Tag.

4 Test Session Recording and Execution

The modified T2 transformation produces traceability links in the generated code, so that the resulting application can be exploited to record model-aware testing sessions.

For recording the test sessions, a *Test Session Recorder* has been designed, by extending the TestGen4Web Firefox add-on [30], so to recognize the trace links in the page rendition and save them in the final test script automatically, without any user's intervention.

As an example, consider the testing session of Section 2. Once the recording is stopped, the navigation is saved in an XML file compliant with the syntax of Canoo WebTest, shown below:

```
/*step 1*/
<testInfo type="trace" info="page1"/>
<echo message="Go to the URL: http://www.acme.com"/>
<wrInvoke url="http://www.acme.com"/>

/*step 2*/
<verifyXPath text=".*Aladdin.*"
  xpath="//div[@id='testUnit_id:dau16
          _name:Product of the day_']
     //div[@id='testAttribute_id:att10
      _name:name_type:string
      _unitName:Product of the day_']"/>

/*step 3*/
<testInfo type="input" info="Aladdin"/>
<testInfo type="trace" info="ln30"/>
<wrClicklink fieldIndex="0"
  label="More.." exactmatch="true"
  description="Click the link labeled More.."/>
```

The test script contains, besides the usual Canoo tags, additional information coming from the trace links.

Each step is annotated by the ID of the model element involved (e.g, as in `<testInfo type="trace" info="page1">`). Assertion steps, e.g., step (2), are expressed by means of XPath expressions that do not depend on the graphical layout, but only on the identifiers of the model elements. If the code is regenerated with a different style or layout, the assertion remains valid.

Trace links are also enhanced with dynamic information about the objects appearing in the navigated page. For instance, step (3) shows the case of the navigation of a link, where the `<testInfo type="input" info="Aladdin">` annotation stores the name of the object that is associated with the navigated link as a parameter. In this way, session recording can take advantage of the dynamic information coming from the objects of the data model, and blend it with the information on the user's interactions with the page widgets (e.g, single or multiple selections from indexes, selections from combo boxes, and so on).

The final element of the proposed regression testing environment is the *Regression Testing Plug-in*, a component of the WebRatio tool suite that allow modelers to perform regression testing from within the same tool they use for design and code generation.

The Regression Testing Plug-in executes the recorded scrips using the Canoo WebTest platform and collects the outcome of the execution, linking each step to the model elements it refers to.

The plug-in exploits the information stored inside the test script by the Navigation Recorder to reflect the user's navigation onto the WebML model, thanks to the identifier of the elements; the plug-in can replay a session visually and can overlay the dynamic information on the navigated objects over the model elements, as shown in Figure 6.

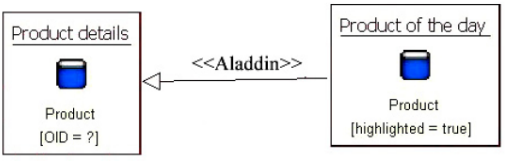

Fig. 6. Visual replay of the testing session with dynamic information overlaid on the WebML model

The replay of a testing session from within the WebRatio IDE is achieved by a client/server connection between the WebRatio Regression Testing Plugin and the Canoo test environment.

The WebRatio plug-in acts as a server and starts the test environment as a client. The client, in turn, opens a new socket to communicate with the server sending to it the testing session trace. Once the test execution ends, the server collects all the identifiers of the WebML elements that have been reached during the test execution together with the information on the outcome of each step. The WebRatio plug-in presents the regression test results in a tabular pane (see Figure 7), where each row displays the identifiers of the WebML elements, their input and a description in natural language of the current step.

Using the provided visualizations, developers can monitor the regression steps and correlate them to the involved elements of the WebML model. In the case of a test failure, the plug-in also catches the exceptions launched from the test

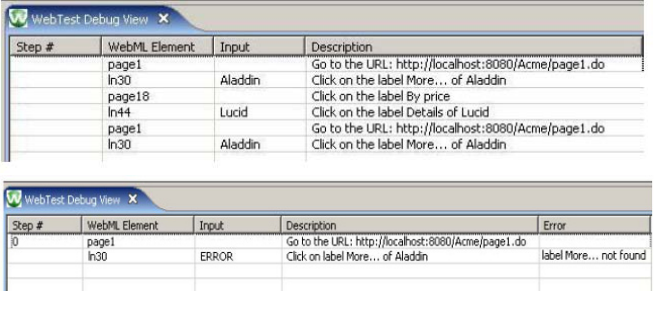

Fig. 7. Tabular representation of a test: success (top) and failure (bottom)

environment, and reports the cause of the errors in the debugging pane (as shown in the bottom part of Figure 7).

5 Implementation

The HOT has been implemented using the ATL language and the AmmA [10] 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 Test Session Recorder has been implemented extending the Firefox TestGen4Web add-on, using XUL and Javascript. In particular, the Javascript module that generates the output has been modified to produce XML files compliant with Canoo WebTest. Furthermore, its code has been refactored to manage every type of assertions in a separate sub-module.

The WebRatio Regression Testing Plug-in has been implemented by means of: 1) a a Java component that runs the Canoo WebTest environment, taking the test script as input, and elaborates the information received from the test execution; 2) an Eclipse view that visualizes the execution outcome. The communication between the test execution platform and WebRatio is regulated using auxiliary Groovy tags inserted in the test script by the Test Session Recorder.

6 Related Work

The task of optimizing the regression testing phase has been addressed in literature especially from the point of view of selective regression testing [27], i.e. of optimizing the regression test set removing superfluous tests. The importance of model-based specifications, for generating and selecting test cases, is already recognized [15]. The HOT framework presented in this paper, as a general approach to embed high-level information in low level code, can be naturally used to address these concerns. In this paper we presented also an original application of the method that facilitates the manual development of regression test cases.

Our application makes use of traceability links to connect the generated implementation with model-based specifications. The concept of traceability links has been widely investigated in literature. A first classification of traceability has been made between *traceability in the small* and *traceability in the large* [8]. The former is intended to handle the trace information between model elements, i.e. information about how different elements of source and target models are linked together; the latter traces information between models in the whole, in order to have information about relationships between distinct models. In some approaches the traceability mechanism is implicitly embedded in the tool's algorithms [11],[25], while other approaches represent traceability relationships explicitly, e.g., [18]. In this latter case, the location where the links are stored, can be the source and/or target model, or separate (e.g, by means of a GUIDE in each model element and traceability information separate from the source

and target models). Our approach realizes traceability in the small representing explicitly the traceability links in the target model.

Transformation frameworks can address traceability during the design of transformations [14], either by providing dedicated support for traceability (e.g., Tefkat [22], QVT [2]), or by encoding traceability as any other link between the input and output models (e.g., VIATRA [33], GreAT [6]). Traceability links may be encoded manually in the transformation rules (e.g., [22]), or inserted automatically (e.g., [2]). The HOT-based approach that we propose can be used to add traceability support to languages like groovy, that do not provide any built-in support to automatic or manual traceability links.

With respect to hard-coding the traceability mechanism when developing the transformation, our use of a HOT favors reusability and extension, because the feature to be weaved into the transformation is managed separately.

A general traceability system using HOTs is already implemented in [20], 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). In other analogous solutions, such as [17], the traceability links are represented by an ad-hoc extension of a standard metamodel for modeling correspondences, the Atlas Weaving Metamodel [16]. Our approach differs from these in merging traceability links within the target metamodel, i.e. the generated implementation code. We showed how this technique is useful in the Web domain to derive model-based test cases from hypertext navigations.

Finally Aspect Oriented Development can be considered as a particular case of HOT. Using a generic transformation language for defining the HOT, our approach has a higher expressing power and flexibility, allowing the definition of complex HOT rules.

7 Conclusions

In this paper we have presented a framework for supporting regression testing in MDE environment. The framework supports the phases of: 1) recording a testing session with a conventional Record & Play tool; 2) replaying the recorded session from within the same IDE that is used for application modeling and code generation; 3) tracing the failures of a test session to the model elements most related to them. The core of the approach is the connection between the conceptual model, which the developer uses to specify and build the application, and the generated code, which is exploited to record and play the testing session. Such a connection is established by traceability links between the input model and the generated code, automatically inserted by a modified version of the code generator. This modified version is itself produced automatically, by exploiting the powerful paradigm of Higher Order Transformation (HOT), which are transformations that operate on other transformation. The resulting framework enables MDE developers to perform regression testing in an effective way, without breaking the level of abstraction entailed by the use of models as the principal artifact of design.

The ongoing and future work will focus on: 1) Extending the HOT to obtain a code generator capable of producing application code instrumented for the step-by-step debugging of the sequences of operations, which are now executed as black boxes; 2) Structuring the HOT in a modular way, so that it is possible to weave different orthogonal aspects in the code generator, e.g., the insertion of performance verification code or of security code (e.g. alternative URL encoding and encryption policies). 3) Supporting selective regression testing [24]: when a change is made, the collaborative work function of WebRatio can be used to identify the list of differences between the original and modified model and to select a minimal set of sessions to execute. From an analysis of differences, it could also be possible to launch the extended code generator and session recorder to automatically synthesize the sessions needed for covering the new parts of the model.

Acknowledgment. We wish to thank Alessandro Baffa for the implementation work and the WebRatio Team for the evaluation of the testing framework.

References

1. MoDisco home page, `http://www.eclipse.org/gmt/modisco/`
2. QVT 1.0., `http://www.omg.org/spec/QVT/1.0/`
3. WebRatio, `http://www.webratio.com`
4. Abrahao, S., Pastor, O.: Measuring the functional size of web applications. Int. J. Web Eng. Technol. 1(1), 5–16 (2003)
5. Abrahão, S.M., Mendes, E., Gómez, J., Insfrán, E.: A model-driven measurement procedure for sizing web applications: Design, automation and validation. In: MoDELS, pp. 467–481 (2007)
6. Agrawal, A., Karsai, G., Shi, F.: Graph transformations on domain-specific models. Technical report, ISIS (November 2003)
7. Baerisch, S.: Model-driven test-case construction. In: ESEC-FSE Companion 2007: 6th Joint Meeting on European SE Conf. and the ACM SIGSOFT Symp. on the Foundations of SE, pp. 587–590. ACM, New York (2007)
8. Barbero, M., Del Fabro, M.D., Bézivin, J.: Traceability and provenance issues in global model management. In: 3rd ECMDA-Traceability Workshop (2007)
9. 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)
10. Bézivin, J., Jouault, F., Touzet, D.: An introduction to the ATLAS model management architecture. Research Report LINA(05-01) (2005)
11. Briand, L., Labiche, Y., Soccar, G.: Automating impact analysis and regression test selection based on uml designs. In: IEEE International Conference on Software Maintenance, p. 252 (2002)
12. Canoo. Canoo Web Test (2008), `http://webtest.canoo.com`
13. Ceri, S., Fraternali, P., Bongio, A., Brambilla, M., Comai, S., Matera, M.: Designing Data-Intensive Web Applications. Morgan Kaufmann, USA (2002)
14. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: OOPSLA 2003 Workshop on Generative Techniques in the Context of MDA (2003)
15. Dick, J., Faivre, A.: Automating the generation and sequencing of test cases from Model-Based specifications. In: Larsen, P.G., Woodcock, J.C.P. (eds.) FME 1993. LNCS, vol. 670, pp. 268–284. Springer, Heidelberg (1993)

16. Del Fabro, M.D., Bézivin, J., Jouault, F., Breton, E., Gueltas, G.: Amw: a generic model weaver. In: 1ére Journée sur l'Ingénierie Dirigée par les Modèles (IDM 2005) (2005)
17. GMT Project. Amw use case - traceability (February 2008), http://www.eclipse.org/gmt/amw/usecases/traceability
18. Hartman, A., Nagin, K.: The AGEDIS tools for model based testing. SIGSOFT Softw. Eng. Notes 29(4), 129–132 (2004)
19. HTMLUnit Team. HTMLUnit (2008), http://htmlunit.sourceforge.net/
20. Jouault, F.: Loosely coupled traceability for atl. In: European Conference on Model Driven Architecture (ECMDA), workshop on traceability (2005)
21. JWebUnit Team. JWebUnit (2008), http://jwebunit.sourceforge.net/
22. 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)
23. Li, N., Ma, Q.-q., Wu, J., Jin, M.-z., Liu, C.: A framework of model-driven web application testing. In: COMPSAC 2006, Washington, DC, USA, pp. 157–162. IEEE Computer Society, Los Alamitos (2006)
24. Naslavsky, L., Richardson, D.J.: Using traceability to support model-based regression testing. In: ASE 2007, pp. 567–570. ACM, New York (2007)
25. Nebut, C., Fleurey, F., Le Traon, Y., Jezequel, J.: Automatic test generation: A use case driven approach. IEEE Transactions on SE 32(3), 155, 140 (2006)
26. 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)
27. Rothermel, G., Harrold, M.J.: Analyzing regression test selection techniques. IEEE Transactions on Software Engineering 22(8), 529–551 (1996)
28. Saad, M.A., Kamenzky, N., Schiller, J.: Visual scatterUnit: A visual model-driven testing framework of wireless sensor networks applications. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 751–765. Springer, Heidelberg (2008)
29. Selenium Project. Seleniumhq (2008), http://seleniumhq.org/
30. Vinay Srini. Testgen4web (2008), http://developer.spikesource.com/blogs/vsrini/2008/06/testgen4web_update_10_1.html
31. Stahl, T., Voelter, M., Czarnecki, K.: Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons, Chichester (2006)
32. The Apache Jakarta Project. Cactus (2008), http://jakarta.apache.org/cactus
33. Varró, D., Varró, G., Pataricza, A.: Designing the automatic transformation of visual languages. Sci. Comput. Program. 44(2), 205–227 (2002)