Journal of Web Engineering, Vol. 10, No. 1 (2011) 001–020
© Rinton Press

USING TRACEABILITY LINKS AND HIGHER-ORDER TRANSFORMATIONS FOR EASING REGRESSION TESTING OF WEB APPLICATIONS

PIERO FRATERNALI
Dipartimento di Elettronica ed Informazione, Politecnico di Milano, via Ponzio 24/5
20133, Milano, Italy
piero.fraternali@polimi.it

MASSIMO TISI
AtlanMod, INRIA & Ecole des Mines de Nantes, La Chantrerie 4, rue Alfred Kastler
BP 20722, F-44307 Nantes Cedex 3, France
massimo.tisi@inria.fr

Received June 1, 2010
Revised November 25, 2010

For Model-Driven Engineering to become widely accepted by developers, it is necessary that its principles and techniques be applied not only to the generation of code from Platform Independent Models (PIMs), but more generally to all the phases of the software life-cycle. This paper focuses on the use of PIMs to support automation in the regression testing phase of a system; the proposed framework lets developers record and replay testing sessions and investigate testing failures at the level of their PIMs; this is made possible by traceability links automatically weaved into the generated code, whereby developers can follow application execution at the model level. Such traceability links are created by a modified version of the code generation transformation, automatically produced by means of a Higher-Order Transformation (HOT). A HOT is a transformation that takes as input a transformation (the original code generator) and creates another transformation (the code generator capable of producing traceability links). 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. This approach is particularly helpful in the Web context, where code generation transformations must follow changes in the technology and presentation styles. Using HOTs ensures the automatic evolution of the transformation for traceability links when the code generation transformation changes.

Keywords: Model-Driven Engineering, Model-Driven Architecture, Testing, Regression, Testing Automation
Communicated by: M. Gaedke, M. Grossniklaus, and O. Diaz

1 Introduction

Model-Driven Engineering (MDE) is the branch of Software Engineering that advocates the use of models and model transformations as the primary means for representing a system and its application domain and for turning the system specifications into an executable application [10]. Models incorporate the knowledge about the application domain or software solution at hand. In the Model-Driven Architecture [23], the OMG view on model-driven development, models are classified, according to their degree of abstraction with respect to the implementation technology, into Computation Independent Models (CIMs), which express knowledge about the application domain irrespective of the software system under design, Platform Independent Models (PIMs), which describe a specific system abstracting away from the implementation technology details, and Platform Specific Models (PSMs), which incorporate knowledge about the selected deployment infrastructure and technology [28].

The knowledge embodied in the model is primarily used for forward engineering, that is, the progressive refinement towards the final implementation code, from CIM to PIM to PSM. However, models have a range of application that goes beyond code generation [34]. 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, 25, 29, 31].

In the domain of testing, the use of models mostly concentrates on automating the production and execution of test cases, for example by using the knowledge embedded within models for producing test sets satisfying specific quality criteria (e.g., selected coverage criteria over the application models or execution paths).

Other activities related to testing, like model-based selective regression testing and behavioral result evaluation are less supported [26]. Regression testing refers to the inspection of successive versions of a deployed system, in order to identify and remove *regressions*, which are those situations in which a fault occurs in a program functionality that previously worked, due to a change in the software. Behavioral result evaluation, instead, refers to the activity whereby a developer traces the results of a testing session, typically a set of identified bugs, to the most likely source of malfunctioning in the software [37].

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 supported by code generation transformations, 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. 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.

The focus of this paper is to establish a link between the implementation level at which the application is tested and the platform-independent model at which it is designed, so as to ease the production of regression test sets and their behavioral evaluation after a new release of the system is generated and deployed. This goal is pursued by means of a framework for regression testing, which enables Model-Driven Engineers to 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 general architecture of the Higher-Order Transformation framework for the production of model traceability clues in a model-to-code transformation; Section 4 describes in greater detail the rules that compose the HOT; Section 5 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 6 describes the runtime environment for executing the regression test sessions. Section 7 briefly discusses the implementation work; Section 8 compares our contribution to the related work; Section 9 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 [32] and TestGen4Web [33]), 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. Fill the input form with the string ‘WebTest’
3. Press the ‘I’m Feeling Lucky’ button
4. Assert that the string ‘WebTest’ must appear in the returned page
```

Specifying an assertion requires an extension of the browser. Figure 1 shows how step (4) is performed in a popular Record & Play tool.

The test scripts generated by the Navigation Recorder can then be executed, using one of several Web test environments available, e.g., Canoo WebTest [13], Cactus [35], HTMLUnit [20] and JWebUnit [22], which replay the test session and verify the assertions, highlighting failures.

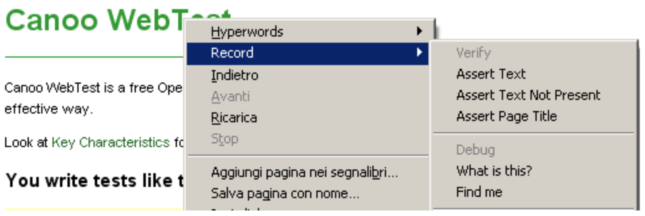


Fig. 1. Creating an Assertion with a Record & Play tool

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 [14]. 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 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 2 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*.

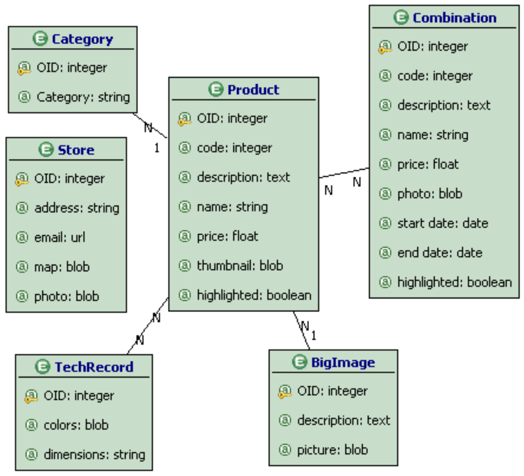


Fig. 2. Data model of the Product Catalog Application

Figure 3 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. Data publishing units are associated to *selectors* to filter which data elements have to be shown. For instance the *Product of the day* data unit shows only the product whose attribute *highlighted* is true. The units have outgoing links, which enable navigation and parameter passing. For example, the *Product of the day* 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.

Figure 4 shows a fragment of the site view for managing the data of the same application. The *login unit* in Figure 3 checks the credentials of the user and redirects him to this administration siteview, if he has sufficient rights. The fragment in Figure 4 has only one page, for deleting products and inserting new ones. This page contains a list of all products and a form for new insertions. The list of products is associated to checkboxes, for selecting the ones that have to be deleted. A clickable link will activate a *Delete unit* that will erase the selected products from the database. For the addition of new products, once the form has been filled, a link will start a chain of operations to implement the requirement of having only one highlighted item in every moment. First a *Switch unit* will check the value of the *highlighted* field in the form. If this value is negative, the new product will be simply created by a *Create unit* that receives the needed data from the form by a transport link. Otherwise, if the administrator wants to create an highlighted item, before the product creation a *Modify unit* will be activated, to reset the *highlighted* flag from any other product.

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

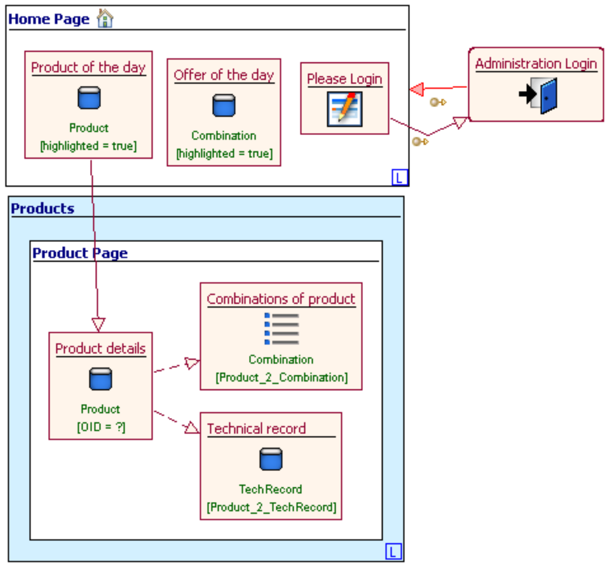


Fig. 3. Publishing site view of the Product Catalog Application

utilize any presentation 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 presentation 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 ‘Product of the day’ data unit displays the ‘Lucid’ item
3. Fill the login form with username ‘manager’, password ‘manager’ and click the ‘Login’
   button
4. Fill the ‘New product’ form inserting as code ‘1234’, as name ‘Aladdin’, as price
   ‘3000’, selecting the ‘highlighted’ checkbox, and clicking on the ‘Save’ button
5. Click on the ‘Logout’ link
6. Check that the ‘Product of the day’ data unit displays the ‘Aladdin’ item
7. Navigate the outgoing link of the ‘Product of the day’ unit
8. Check that the ‘price’ attribute is ‘3000’
```

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 and 5 test the authentication mechanism. Step 4 exercises the chain of operations for the creation of a new highlighted product and Step 6 verifies the result of these operations. Step 7 and 8 test the contextual navigation from the Home Page to another Web page, verifying that the link in the Home Page exists and has proper parameters and 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 (2) requires evaluating an XPath expression over the page markup:

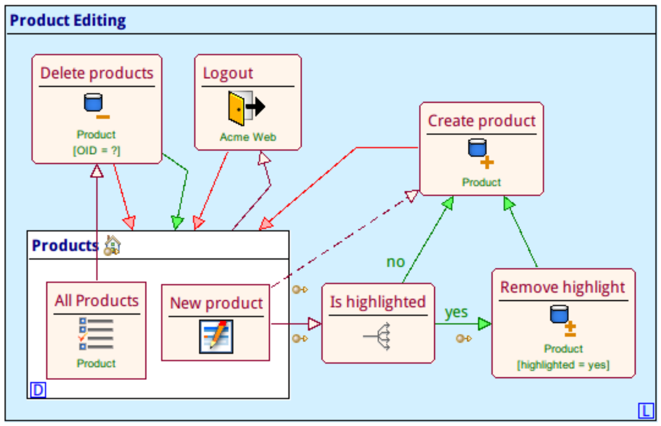


Fig. 4. Administration site view of the Product Catalog Application

the evaluation of some XPath expressions may change if the page layout is updated (even if functionality does not change).

3 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.

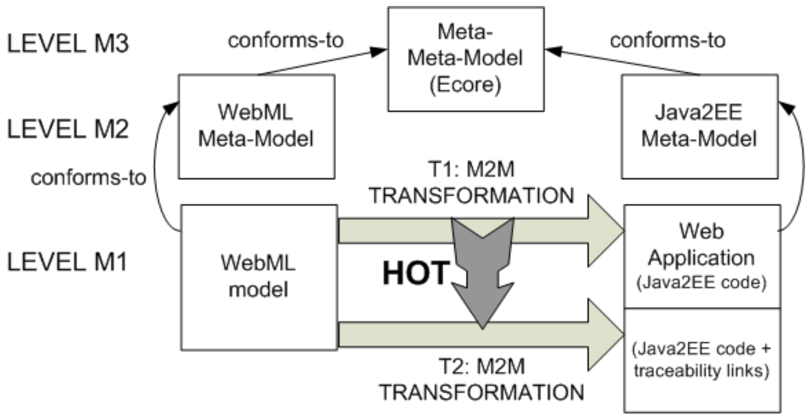


Fig. 5. Using HOT to weave traceability links into the code generator transformation

Figure 5 pictorially illustrates the HOT framework: the code generation process can be seen as a model-to-model transformation (T1 in Figure 5) 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 5 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 6.

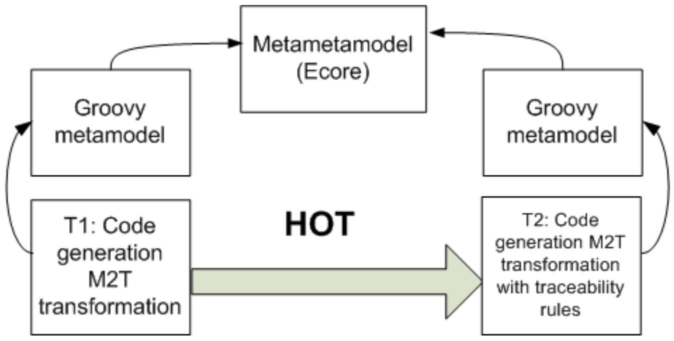

Metametamodel (Ecore)
Groovy metamodel
Groovy metamodel
T1: Code generation M2T transformation
HOT
T2: Code generation M2T transformation with traceability rules

Fig. 6. Input and Output Models of the HOT

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) for producing the traceability links in the implementation code.

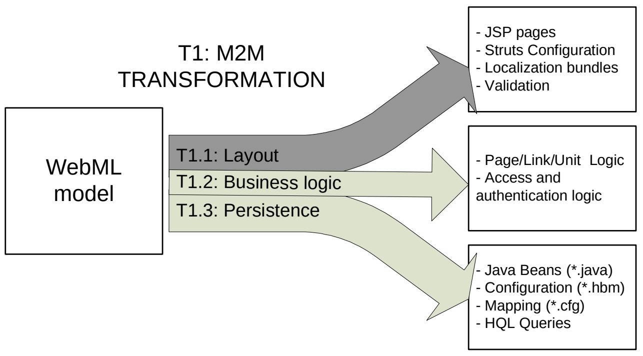


Fig. 7. Structure of T1 transformation

Figure 7 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.

1. 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.

2. 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.

3. 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 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 and name of the content units appearing within the pages of the WebML model;
  - the id, name and type of each value included in the content units;
  - the id, name and parameters of navigable links;
  - the id of form fields.
- 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 3 can be represented by the following fragment of the input model:[a]

```
<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 “Lucid”:

```
<table>
 <tr>  <td>Lucid</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">
         Lucid
            </div></td> </tr>
 <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 rendering-neutral DIV elements, link the output model (e.g., an XHTML table cell containing the string ‘Lucid’) to the input model (e.g., the *name* attribute published by the *ProductOfTheDay* data unit).

4 Higher-order transformation rules

The HOT for producing traceability links must match the input model (transformation T1) and produce an output model (transformation T2), by suitably extending T1. Therefore, the HOT will contain rules that detect the portions of T1 that produce relevant presentation code and augment them with extra logics for creating the traceability links, without altering the semantics of T1 code generation rules.

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"])) %>
```

[a]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.

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 creates 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:

```
1  rule UnitLink {
2    from
3      matched : GroovyMM!Scriptlet (
4        matched.recursiveStatements()->exists(s |
5          s.oclIsKindOf('GroovyMM!MethodInvocation') and
6          s.name='printRaw' and
7          s.arguments->exists(e |
8            e.oclIsKindOf('GroovyMM!MethodInvocation') and
9            e.name='executeTemplate' and
10           e.arguments->at(2).oclIsKindOf('GroovyMM!Map') and
11           e.arguments->at(2).elements->exists(t |
12             t.key='templateType' and
13             t.value.oclIsKindOf('GroovyMM!String') and
14             t.value.value='unit'
15     ))))
16   to
17     div : GroovyMM!Tag (
18       name <- 'DIV',
19       attributes <- Sequence{id},
20       children <- Sequence{matched}
21     ),
22     id : GroovyMM!TagAttribute (
23       name <- 'id',
24       value <-'testUnit_id:<%=unitId%>_name:<%=unitName%>_'
25     )
26 }
```

The HOT rule matches any Groovy scriptlet in the generator. To check that the scriptlet triggers the expansion of the unit content, the rule calls the *recursiveStatements()* helper, which returns the sequence of statements contained into the scriptlet. To have a match, one of these statement has to print to the output (by the *printRaw* method) the result of an *executeTemplate* call to a unit template. Hence the filter checks for a nested call to *executeTemplate* with a parameter *templateType = 'unit'*. This ensures that only the portion of T1 that deals with the transformation of a WebML content unit into some JSP code fragment of the View is actually detected. Notice that the matching part of the rule is designed to be robust to code-generation changes, as long as they do not affect the syntax of the call to the content unit template that generates the JSP code, which is an unlikely occurrence.

The output pattern of the rule is a *Tag* named *DIV* containing a *TagAttribute* named *id* representing an encoding of the traceability link. The encoding exploits some environment variables of the code generation platform, namely `<%=unitId%>` and `<%=unitName%>`, which are computed at code-generation time by the Groovy interpreter. The matched scriptlet is finally copied as a child of this Tag.

Similar HOT rules have been developed to address the other kinds of traceability links, for tracing attribute values, input fields and links. The rules have the same structure as the previous one, and they differ only for the type of call they match (e.g., `t.value.value = 'attribute'}` at line 14) and the kind of id they generate (e.g., `value <- testAttribute_id:<%`

= `attr["id"]%>_name:<%= attr["name"]%>_type:<%=attr["type"]%>_unitName:<%unitName%>`).

5 Test Session Recording

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 *Navigation Recorder* has been designed, by extending the TestGen4Web Firefox add-on [33], so to recognize the trace links in the page presentation and save them in the final test script automatically, without any user's intervention.

With respect to the vanilla TestGen4Web tool, the script produced by the Navigation Recorder contains, besides the usual Canoo tags, additional information coming from the trace links. The Navigation Recorder:

- implements all the standard steps (actions and assertions) relying uniquely on the stylesheet-independent traceability ids for identifying the platform independent page elements; concretely, element identification exploits only XPath expressions that do not depend on presentation markup;
- introduces a *testInfo* tag that memorizes useful information about a test step, e.g., the traceability link pointing towards the related model element (denoted by the *trace* attribute) or a readable identifier for recalling the user choices that have been used to compose the test session, e.g., the selection of an item from an index (denoted by the *input* attribute);
- uses the *echo* tag to print a sequence of human-readable descriptions of steps as the test result, which provides a human-intelligible trace of the performed test session.

All this information is both recorded in the script and communicated back to the modeling environment during the test replay.

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.org/page1.do"/>
<wrInvoke url="http://www.acme.org/page1.do"/>

/*step 2*/
<testInfo type="trace" info="dau16"/>
<echo message="Verify the xpath of the attribute:'name' of the unit 'Product of the day
    '"/>
<verifyXPath description="check the xpath" regex="true" text=".*Lucid.*" 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="username"/>
<testInfo type="trace" info="enu2"/>
<echo message="Fill the field 'username' with the value 'manager'"/>
<setInputField description="set input field fld3" name="fld3" value="manager"/>

<testInfo type="input" info="password"/>
<testInfo type="trace" info="enu2"/>
<echo message="Fill the field 'password' with the value 'manager'"/>
<setInputField description="set input field fld4" name="fld4" value="manager"/>

<testInfo type="trace" info="ln17"/>
<echo message="Click the button with the label 'Enter'"/>
<clickButton description="click button" label="Enter"/>
```

```

/*step 4*/
<testInfo type="input" info="name"/>
<testInfo type="trace" info="enu10"/>
<echo message="Fill the field 'name' with the value 'Aladdin'"/>
<setInputField description="set input field fld38" name="fld38" value="Aladdin"/>

<testInfo type="input" info="price"/>
<testInfo type="trace" info="enu10"/>
<echo message="Fill the field 'price' with the value '3000'"/>
<setInputField description="set input field fld40" name="fld40" value="3000"/>

<testInfo type="input" info="highlighted"/>
<testInfo type="trace" info="enu10"/>
<echo message="Set the radio button 'highlighted' to 'yes'"/>
<setRadioButton description="set radio button fld12" name="fld12" value="yes"/>

<testInfo type="trace" info="ln86"/>
<echo message="Click the button with the label 'Create'"/>
<clickButton description="click button" label="Create"/>

/*step 5*/
<testInfo type="trace" info="ln33"/>
<echo message="Click on the link with label Logout"/>
<wrClicklink description="Click the link with the label Logout" label="Logout" exactmatch
    ="true"/>

/*step 6*/
<testInfo type="trace" info="dau16"/>
<echo message="Verify the xpath of the attribute:'name' of the unit 'Product of the day
    '"/>
<verifyXPath description="check the xpath" regex="true" 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 7*/
<testInfo type="input" info="Aladdin"/>
<testInfo type="trace" info="ln30"/>
<echo message="Click on the label More... of Aladdin"/>
<wrClicklink fieldIndex="0" label="More..." exactmatch="true" description="Click on the
    label More... of Aladdin"/>

/*step 8*/
<testInfo type="trace" info="dau1"/>
<echo message="Verify the xpath of the attribute:'price' of the unit 'Product details'"/>
<verifyXPath description="check the xpath" regex="true" text=".*3000.*" xpath="//div[@id='
    testUnit_id:dau1_name:Product details_']//div[@id='testAttribute_id:att11_name:
    price_type:float_unitName:Product details_']"/>
```

In the following paragraphs, we describe the example script reported above, addressing each feature separately.

Trace links. Each step of the script is automatically annotated by the ID of the model element it refers to (e.g., as in `<testInfo type="trace" info="page1"/>` in Step 1). The elements mentioned in the test step can be: *pages*, for direct page access, *links*, for hypertext navigation (e.g., in Step 5), or WebML *units*, for assertions on the displayed content or actions performed on the unit (e.g., filling an input field of an entry unit, as in Step 3 and 4).

Human-readable descriptions. For each step a message is echoed to a log file (e.g., as in `<echo message="Fill the field 'username' with the value 'manager'"/>` in Step 3). This human-readable description, automatically generated by the Navigation Recorder, is used in our toolsuite for two purposes:

- to obtain a log-file composed by a sequence of these descriptions, as a human-readable trace of the test (this is particularly useful in case of errors, to clearly identify the steps the lead to it, even without looking at the model);

- to show the description during the test replay, as an additional information for the model debugging.

Dynamic Information. Trace links are also enhanced with dynamic information about the objects appearing in the navigated page. For instance, step (7) 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).

Use of external names for objects. While the Navigation Recorder tool is able to automatically detect the values passed as parameters during the navigation, these values are most of the times internal identifiers (e.g., database OIDs); in many cases, it would be more convenient to use in the test trace object external names, to provide a more descriptive denotation of the object associated with the parameter. For instance, Step (7) could pass a numerical identifier for the product to show and this information would probably be less useful in the testing session. In our example the tester would need to look at the database to verify that the numerical id actually corresponds to the 'Aladdin' product. To support the use of external names, the implemented recording tool allows the tester to manually override the default dynamic information stored into the `testInfo` tag. The overriding is performed directly in the browser by highlighting a label on the page and inserting it into the trace by means of a new ad hoc command added to the Canoo contextual menu (shown in Figure 1), called *Set input*.

Assertions Creation and Usability of the Navigation Recorder. 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.

The Navigation Recorder currently enables the verification of equality predicates over the strings contained within the generated code of a WebML element. Such assertions can be expressed at two levels of granularity:

- The basic level corresponds to testing a single value (e.g., a *unit attribute*, in the WebML jargon) in the page. For instance Step 2 and 6 check that the *name* attribute of the *Product of the day* WebML unit has been updated from 'Lucid' to 'Aladdin'. The control is performed by retrieving the element code, using an XPath expression, and checking that it contains the required string. The XPath expression matches DIV elements with the specific id (contained into the right unit). The verification is performed by comparing the content of the DIV under test with a regular expression obtained by stripping the original DIV from any tag (e.g., `".*Aladdin.*"`).

- A second level of granularity is provided for testing entire units with one command. When a WebML unit is selected in an assertion step, a unit-level assertion can be declared. Such a command produces a textual assertion over the unit's name and analyzes the unit's structure to identify any internal attribute; for each detected unit attribute, an attribute assertion is automatically created.

The creation of assertions during the test recording session can be simplified by providing an easy way to select directly in the browser the WebML elements to test. To make the Navigation Recorder more usable, we have extended the HOT illustrated in Section 4, so to produce not only traceability clues but also recording usability links. The modified code generator produced by the HOT creates, for each WebML element, special-purpose usability links with a dedicated id at the beginning of the DIV element. For example, the augmented presentation code of the *Product of the day* unit is:

```
<div id="testUnit id:dau16 name:Product of the day"><a id="WRTest" href="#">Test</a>
 <table>
 <tr>   <td><div id="testAttribute id:att10 name:name
        type:string unitName:Product of the day"><a id="WRTest" href="#">Test</a>
         Lucid
            </div></td> </tr>
 <tr>   <td><div id="testAttribute id:att11 name:price
        type:float unitName:Product of the day"><a id="WRTest" href="#">Test</a>
         1500
            </div></td> </tr>
 <tr>   <td><div id="testAttribute id:att12 name:thumbnail
         type:blob  unitName:Product of the day"><a id="WRTest" href="#">Test</a>
         <img src="images/small_table.jpg" />
            </div></td> </tr>
 </table>
</div>
```

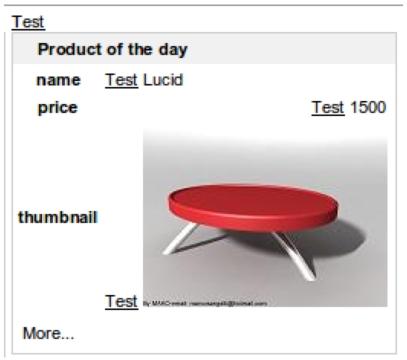


Fig. 8. Links for recording assertions.

A click on any of these recording usability links is caught by the Navigation Recorder and triggers the addition of an appropriate assertion step for the associated WebML element.

6 Test Session Execution

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 9.

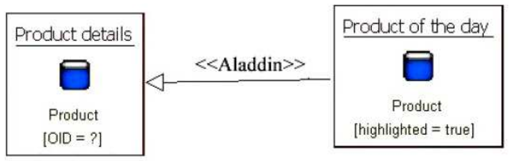


Fig. 9. Visual replay of the testing session with dynamic information overlaid on the WebML model

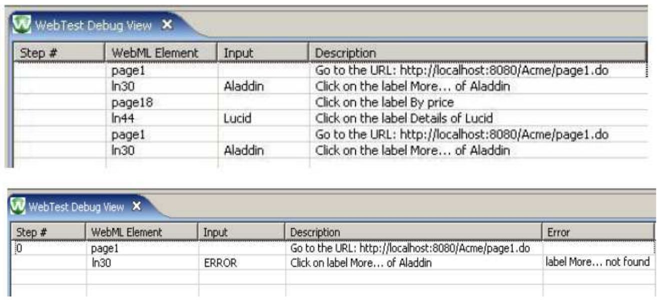
The replay of a testing session from within the WebRatio IDE is achieved by a client/server connection between the WebRatio Regression Testing Plugin and the Canoo test environment.

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 communication between the test execution platform and WebRatio is performed without an ad-hoc modification of the Canoo WebTest environment. Since Canoo supports the insertion of arbitrary code inside a test in the form of Groovy tags, our Navigation Recorder can easily instrument the test using tags like the following:

```
<groovy>
  def s = new Socket("localhost", Integer.parseInt
           (step.project.properties.eventSocketPort))
  s &lt;&lt; "trace|page1_input| _
           message|Go to the URL: http://www.acme.com/page1.do"
  s.close()
</groovy>
```

The WebRatio plug-in presents the regression test results in a tabular pane (see Figure 10), where each row displays the identifiers of the WebML elements, their input and a description in natural language of the current step



Step #	WebML Element	Input	Description
	page1		Go to the URL: http://localhost:8080/Acme/page1.do
	ln30	Aladdin	Click on the label More... of Aladdin
	page18		Click on the label By price
	ln44	Lucid	Click on the label Details of Lucid
	page1		Go to the URL: http://localhost:8080/Acme/page1.do
	ln30	Aladdin	Click on the label More... of Aladdin

Step #	WebML Element	Input	Description	Error
0	page1		Go to the URL: http://localhost:8080/Acme/page1.do	
	ln30	ERROR	Click on label More... of Aladdin	label More... not found

Fig. 10. Tabular representation of a test: success (top) and failure (bottom)

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 environment, and reports the cause of the errors in the debugging pane (as shown in the bottom part of Figure 10).

7 Implementation

The HOT has been implemented using the ATL language and the AmmA [11] framework. To integrate the Groovy language in the transformation framework, a minimal Groovy metamodel has been developed extending the JavaAbstractSyntax metamodel provided by the MoDisco project [1].

The Navigation 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 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.

8 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 [30], 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 [16]. 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 [12],[27], while other approaches represent traceability relationships explicitly, e.g., [19]. 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 [15], either by providing dedicated support for traceability (e.g., Tefkat [24], QVT [2]), or by encoding traceability as any other link between the input and output models (e.g., VIATRA [36], GreAT [6]). Traceability links may be encoded manually in the transformation rules (e.g., [24]), 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 [21], 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 [18], the traceability links are represented by an ad-hoc extension of a standard metamodel for modeling correspondences, the Atlas Weaving Metamodel [17]. 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, an alternative way to inject traceability in the code generator could rely on typical Aspect Oriented Development techniques. HOTs give to our approach an higher expressing power and flexibility, allowing the definition of complex manipulation rules.

9 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.

A validation of the framework is ongoing. A preliminary version of the tool has been used in the WebRatio team to create regression tests for the WebRatio toolsuite itself. We are in the process of involving industrial users of WebRatio in managing the regression testing of their Web applications by our tool.

The 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 [26]: 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. Silvia Abrahao and Oscar Pastor. Measuring the functional size of web applications. *Int. J. Web Eng. Technol.*, 1(1):5–16, 2003.
5. Silvia Mara Abrahão, Emilia Mendes, Jaime Gómez, and Emilio Insfrán. A model-driven measurement procedure for sizing web applications: Design, automation and validation. In *MoDELS*, pages 467–481, 2007.
6. Aditya Agrawal, Gabor Karsai, and Feng Shi. Graph transformations on domain-specific models. Technical report, ISIS, November 2003.
7. Stefan Baerisch. 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*, pages 587–590, New York, NY, USA, 2007. ACM.
8. M. Barbero, M. D. Del Fabro, and J. Bézivin. Traceability and provenance issues in global model management. In *3rd ECMDA-Traceability Workshop*, 2007.
9. Luciano Baresi, Piero Fraternali, Massimo Tisi, and Sandro Morasca. Towards model-driven testing of a web application generator. In *ICWE*, pages 75–86, 2005.
10. J. Bézivin. On the unification power of models. *Software and Systems Modeling*, 4(2):171188, May 2005.
11. J. Bézivin, F. Jouault, and D. Touzet. An introduction to the ATLAS model management architecture. *Research Report LINA,(05-01)*, 2005.
12. L. Briand, Y. Labiche, and G. Soccar. Automating impact analysis and regression test selection based on uml designs. *Software Maintenance, IEEE International Conference on*, 0:0252, 2002.
13. Canoo. Canoo Web Test. http://webtest.canoo.com, 2008.
14. S. Ceri, P. Fraternali, A. Bongio, M. Brambilla, S. Comai, and M. Matera. *Designing Data-Intensive Web Applications*. Morgan Kaufmann, USA, 2002.
15. K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *OOPSLA03 Workshop on Generative Techniques in the Context of MDA*, 2003.
16. Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from Model-Based specifications. In *Proceedings of the First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, pages 268–284. Springer-Verlag, 1993.
17. M. D. Del Fabro, J. Bézivin, F. Jouault, E. Breton, and G. Gueltas. Amw: a generic model weaver. In *1re Journe sur l'Ingnierie Dirige par les Modles (IDM05)*, 2005.
18. GMT Project. Amw use case - traceability. http://www.eclipse.org/gmt/amw/usecases/traceability, Feb. 2008.
19. A. Hartman and K. Nagin. The AGEDIS tools for model based testing. *SIGSOFT Softw. Eng. Notes*, 29(4):129–132, 2004.
20. HTMLUnit Team. HTMLUnit. http://htmlunit.sourceforge.net/, 2008.
21. Frdric Jouault. Loosely coupled traceability for atl. In *European Conference on Model Driven Architecture (ECMDA) , workshop on traceability*, 2005.
22. JWebUnit Team. JWebUnit. http://jwebunit.sourceforge.net/, 2008.
23. Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
24. Michael Lawley and Jim Steel. Practical declarative model transformation with tefkat. In *Satellite Events at the MoDELS 2005 Conference*, pages 139–150, 2006.
25. Nuo Li, Qin-qin Ma, Ji Wu, Mao-zhong Jin, and Chao Liu. A framework of model-driven web application testing. In *COMPSAC '06*, pages 157–162, Washington, DC, USA, 2006. IEEE Computer Society.
26. L. Naslavsky and D. J. Richardson. Using traceability to support model-based regression testing. In *ASE '07*, pages 567–570, New York, USA, 2007. ACM.
27. C Nebut, F Fleurey, Y Le Traon, and J Jezequel. Automatic test generation: A use case driven approach. *IEEE Transactions on SE*, 32(3):155, 140, 2006.
28. Oscar Pastor and Juan Carlos Molina. *Model-Driven Architecture in Practice: A Software Pro-*

duction Environment Based on Conceptual Modeling. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.
29. A. Pretschner. Model-based testing in practice. In *FM*, pages 537–541, 2005.
30. G. Rothermel and M.J. Harrold. Analyzing regression test selection techniques. *Software Engineering, IEEE Transactions on*, 22(8):529–551, 1996.
31. Mohammad Al Saad, Nicolai Kamenzky, and Jochen Schiller. Visual scatterunit: A visual model-driven testing framework of wireless sensor networks applications. In *MoDELS '08*, pages 751–765, Berlin, Heidelberg, 2008. Springer-Verlag.
32. Selenium Project. Seleniumhq. http://seleniumhq.org/, 2008.
33. Vinay Srini. Testgen4web. http://developer.spikesource.com/ blogs/vsrini/2008/06/testgen4web_update_10_1.html, 2008.
34. Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons, 2006.
35. The Apache Jakarta Project. Cactus. http://jakarta.apache.org/cactus, 2008.
36. Dániel Varró, Gergely Varró, and András Pataricza. Designing the automatic transformation of visual languages. *Sci. Comput. Program.*, 44(2):205–227, 2002.
37. L. J. White. Software testing and verification. *Advances in computers*, 26:335–391, 1987.