Towards Model-Driven Testing
of a Web Application Generator

Luciano Baresi[1], Piero Fraternali[1], Massimo Tisi[1], and Sandro Morasca[2]

[1] Dipartimento di Elettronica e Informazione
Politecnico di Milano, Milano, Italy
{baresi,fraterna,tisi}@elet.polimi.it
[2] Dipartimento di Scienze della Cultura, Politiche e dell'Informazione
Università dell'Insubria, Como, Italy
sandro.morasca@uninsubria.it

Abstract. Conceptual modelling is a promising approach for Web application development, thanks to innovative CASE tools that can transform high-level specifications into executable code. So far, the impact of conceptual modelling has been evaluated mostly on analysis and design. This paper addresses its influence on testing, one of the most important and effort-consuming phases, by investigating how the traditional notions of testing carry over to the problem of verifying the correctness of Web applications produced by model-driven code generators. The paper examines an industrial case study carried out in a software factory where code generators are produced for a commercial Web CASE tool.

1 Introduction

Web application testing is a challenging but scarcely investigated subject in the Web Engineering community. In the state of the practice, Web application developers still use a "code and fix" approach to software verification, rather than a systematic and tool-supported method. This situation is the combined result of many factors: Web applications are multi-tiered systems and testing requires different procedures for each tier; developers use multiple languages (e.g., SQL, Java, XSLT, HTML), which hampers a unified testing paradigm; the runtime environment (e.g., the browser) cannot be fully controlled and often behaves differently in different products.

In recent years, *conceptual modelling* has been used to tackle such a complexity. The core idea is that applications are specified by using a high-level visual notation and the implementation code is automatically generated from design models. The benefits of conceptual modelling have been extensively studied in the upper phases of the development process. Little is known on the impact of conceptual modelling on the testing phase.

This paper tries to overcome this limitation and investigates the relationship between conceptual modelling and testing. As automatic code generation substitutes manual coding, the focus of testing shifts from verifying individual Web applications to testing the Web code generator; the latter objective lends itself

to systematic treatment and potentially yields far-reaching benefits, because the results of testing the code generator affect the development of multiple applications.

With model-driven development, the activity of testing a specific Web application splits into two sub-tasks: *schema validation* and *code generator validation.* The former assesses whether the application's conceptual schema[1] is correct with respect to the application requirements and adheres to the syntax and semantics of the chosen Web modelling language. The latter aims at evaluating whether the code generator maps all correct conceptual schemas into correct implementations on all platforms[2]. While the former activity must be performed for every individual application, the latter can be done only once for each deployment platform.

Schema validation requires non-trivial human expertise. However, various techniques, like rapid prototyping [1], model verification [2], and usage analysis [3], may alleviate such a task. Furthermore, schema validation is technology-independent and thus can be addressed also by domain experts.

The validation of the code generator is the novel problem addressed in this paper. The intuition behind our work is that if one could ensure that the code generator produces a correct implementation for all legal and meaningful conceptual schemas (i.e., combinations of modelling constructs), then testing Web applications would reduce to the more treatable problem of schema validation.

The contribution of the paper is twofold. On the theoretical side, we propose a novel formalization of the problem of testing Web code generators as an instance of ordinary black-box testing where test data generation is based on the grammar of a graphical conceptual modeling language (WebML). Testing confidence is expressed by a notion of syntactic coverage, and is characterized by three different classes of coverage (rule, edge, and path) with increasing power and complexity.

From the practical standpoint, the theoretical results were applied to a real scenario. We analyzed the testing process of the code generator produced by the WebRatio CASE tool company (http://www.webratio.com), which maps specifications in the WebML language [1] into code for the J2EE architecture. In the WebRatio software factory, each release of the code generator is tested by automatically running 38 test cases constructed manually by developers in the last four years. Such tests are repeated for all the platforms on which WebRatio applications are certified. We analyzed the empirical test cases used by WebRatio, quantified the associated testing confidence, and identified the minimal test set under the three formal notions of coverage. We also considered 46 conceptual schemas of real applications developed by WebRatio and its partners and recomputed coverage measures with respect to the fragment of WebML that is actually used in real-world applications. This re-evaluation reinforced the testing confidence in the empirically developed test cases.

---

[1] From now on, we use the more precise term *schema* to denote the design specification of a particular application encoded in a given Web modelling language

[2] By platform we mean a specific mix of products for running a Web application at all the involved tiers

All the described experiments were conducted using a prototype visual coverage tool developed in Java.

Although applied to WebML, it is important to stress that all the results are independent of the chosen modelling language and technological setting. Any Web modelling language expressible by means of a formal grammar and equipped with a code generator could be used.

2 Background on WebML and Testing

WebML [1] is a conceptual language originally conceived for specifying Web applications developed on top of database content described using the E-R model.

A WebML schema consists of one or more hypertexts (called *site views*), expressing the Web interface used to publish or manipulate the data specified in the underlying E-R schema.

A site view is a graph of *pages* to be presented on the Web. Pages enclose *content units*, representing components for publishing content in the page (e.g., indexes listing items from which the user may select a particular object, details of a single object, entry forms, and so on); content units may have a *selector*, which is a predicate identifying the entity instances to be extracted from the underlying database and displayed by the unit. Pages and units can be connected with *links* to express a variety of navigation effects and to provide the necessary parameters passing from one unit to another one. Figure 1 shows a WebML hypertext specification and its possible rendition in HTML.
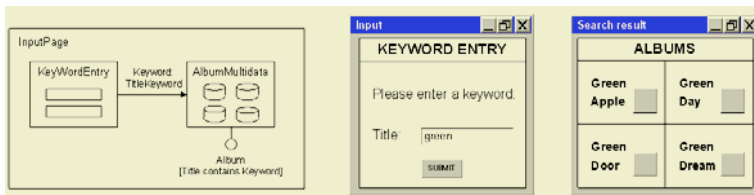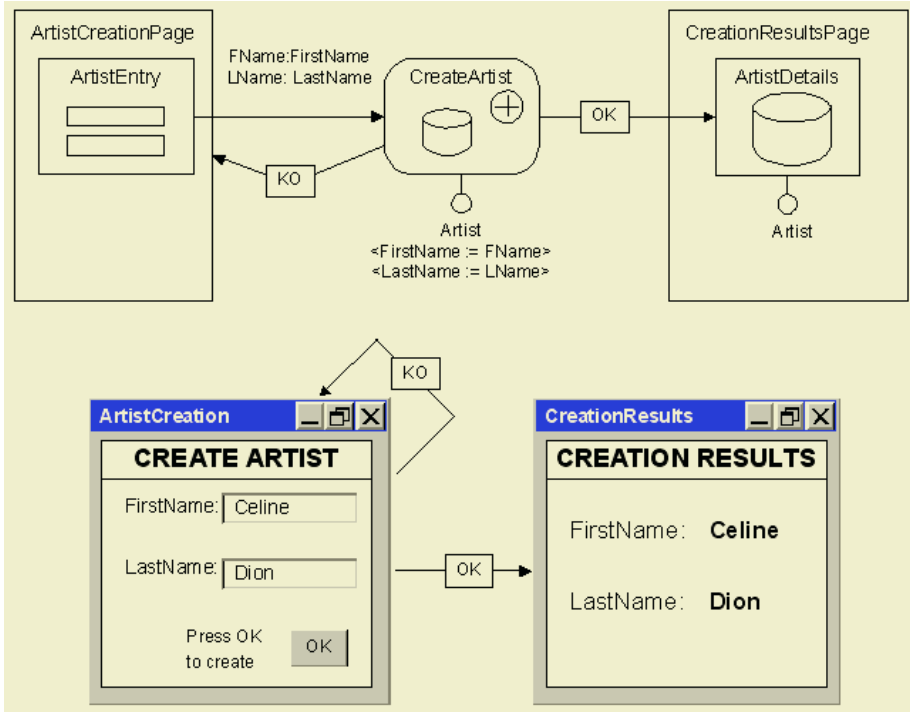

Fig. 1. Example of WebML hypertext and a possible rendition in HTML

The hypertext contains one page (called Input Page), with two units. An entry unit (KeyWordEntry) represents a data entry form and a multidata unit (AlbumMultidata) displays all the instances of entity Album whose titles contain the submitted keyword. The link from the entry unit to the multidata unit is rendered as the submit button, which transports the string inserted by the user as a parameter to be used in the computation of the selector condition (`[Title contains Keyword]`) of the multidata unit.

In addition to content publishing, WebML allows the specification of operations, like Web Service invocation or the update of content, possibly wrapped inside atomic transactions. Basic data update operations are: the creation, modification and deletion of instances of an entity, or the creation and deletion of

Fig. 2. Example of WebML hypertext with operations and a possible rendition in HTML

instances of a relationship. Operations do not display data and are placed outside the pages; user-defined operations can be specified, such as sending an e-mail, logging in and out, e-paying for something, and so on. Figure 2 shows a WebML hypertext specification including operation units and its possible rendition.

The hypertext contains one page (ArtistCreation) with an entry unit (ArtistEntry), whereby the user can enter the details of a new artist. Navigating the output link of the entry unit triggers a create operation (CreateArtist) unit, which inserts a new artist into the database. If the operation succeeds, its output link OK is followed, which displays a page (CreationResults) with the details of the new artist. Otherwise, page ArtistCreation is redisplayed.

In addition to the visual notation, WebML has a formal syntax, encoded in XML. As an example, the following fragment of the WebML DTD shows the syntactical structure of a site view construct.

```
<!ELEMENT SITEVIEW (OPERATIONUNITS, TRANSACTION*, AREA*, PAGE*,
GLOBALPARAMETER*, PROPERTY*, COMMENT?)>
<!ATTLIST SITEVIEW
        id                      ID         #REQUIRED
        name                    CDATA      #IMPLIED
        homePage                IDREF      #IMPLIED
```

```
protected                    (yes|no)    "no"
secure                       (yes|no)    "no"
localize                     (yes|no)    "no"
presentation:style-sheet     CDATA       #IMPLIED
presentation:page-layout     CDATA       #IMPLIED
graphmetadata:go             IDREF       #IMPLIED
>
```

The expressive power of WebML stems primarily from its capability of combining a few elementary concepts in multiple ways to obtain a variety of effects. As we will see, assessing the coverage of testing with respect to all the "meaningful" combinations of concepts is an essential goal of our work.

WebML is implemented in WebRatio, a commercial CASE tool for designing data-centric Web applications.

The architecture of WebRatio (shown in Figure 3) consists of two layers: a WebML Design Layer, providing functions for the visual editing of specifications, and a Runtime Support Layer, implementing the basic services for executing WebML units on top of a standard Web application framework.
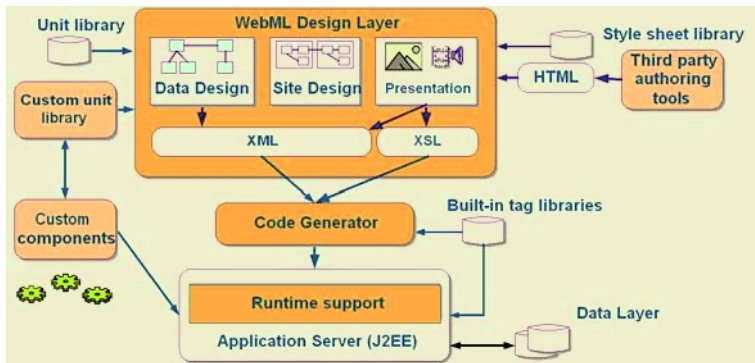

Fig. 3. Architecture of WebRatio

The design layer includes graphical user interfaces for data and hypertext design, which produce an internal representation in XML of the WebML schemas; a second module (called Data Mapping Module) maps the entities and relationships of the conceptual data schema to one or more physical data sources, which can be either created by the tool or pre-existing. A third module for Presentation Design allows the designer to create XSL style sheets from XHTML mockups, associate XSL styles with WebML pages, and organize page layout by arranging the relative positions of content units in each page.

The architecture is completed by the **WebRatio Code Generator**, which exploits XSL transformations to translate the XML specifications visually edited in the design layer into application code executable on top of any platform conforming to the J2EE specifications.

2.1   Software Testing

Testing a program [4] entails running it with a number of input data and checking whether it behaves as expected. Formally, if P denotes a program under test and D its input domain (i.e., the set of all data that can be supplied to P), given a particular $d \in D$, P is *correct* for d if its corresponding output, $P(d)$, satisfies P's specifications. Each element $d \in D$ is said to be a *test case*[3] and a *test set* is a finite set of test data. In general, the only way to achieve absolute certainty about the correctness of P is *exhaustive testing*, that is testing $\forall d \in D$. Obviously, such testing is almost never possible in practice, due to the unwieldy (if not infinite) number of possible inputs, and thus the selection of the actual test set becomes fundamental to assess the correctness of the program. To this end, one can envisage different *testing criteria*, for generating the test set $S \subseteq D$ and probing the correctness of P, and *coverage measures*, for evaluating how thoroughly S can test P. Coverage can be taken as a quantitative measure of testing confidence: the broader the coverage, the more extensive the testing process and thus the confidence on the tested program.

In test data selection, *white-box testing* uses information about the internal structure of the program for selecting test cases, whereas *black-box testing* (also called *functional testing*) only considers the program's functionality and tries to exercise it. In this latter case, test data generation can be facilitated, and even automated, if it is possible to represent the program's behavior – or at least its inputs – in a formal way. This is the case of *syntax-driven testing* [5], which uses the grammar that describes the input domain D to select the test data. A complete coverage of the program input is reached if the test data cover all the grammar's productions, i.e., if the creation of the test data through the grammar requires that each production be applied at least once.

The number of *used* productions is captured by the concept of *rule coverage* [6], which defines the *percentage of grammar productions (rules) applied for deriving the test data*. However, rule coverage is insufficient to characterize the quality of the selected test set, because all test data that contain a given type of input elements are considered equivalent, irrespective of the *context* in which the element appears. Lämmel [6] overcomes this limitations by introducing a subtler coverage measure, called *context-dependent rule coverage*, which takes into account the *context* in which a rule is covered. Intuitively, two input data are *not* considered equivalent if they exercise the same production, but in different combinations with the other rules of the grammar.

Besides selecting "smart" test data, testing also needs a means to evaluate the correctness of program executions. The so-called *oracle* defines a mechanism for verifying that the outputs obtained by executing the program with the test data actually comply with the program specification. Implementing the oracle is often one of the most demanding tasks of the whole testing process.

---

[3] In this paper, we do not distinguish between test case and test datum and we use them as synonyms

3   Testing the Web Application Generator

In this section, we bind the concepts of testing theory to their counterparts in the realm of testing Web code generators, using the WebRatio code generator as running case. We propose a black-box approach based on the formal grammar of WebML. We did not start with a white-box approach because of its intrinsic costs and the need for suitable tools to instrument the different parts of the generator. White-box techniques are the natural complement to our approach and are part of our future work.

The program under test P corresponds to the WebRatio code generator and its input domain D is the set of all possible application schemas, i.e., the set of all valid sentences in the WebML syntax exemplified in Section 2.

An input datum d is a specific WebML schema (an XML file, that comes from the translation of the graphical representation of the model). For example, the following fragment of XML code is a simplified version of the ArtistCreation test case and shows the typical structure of our input data.

```
<SITEVIEW id="sv1" [...]>
  <PAGE id="page1" name="ArtistCreationPage">
    <ENTRYUNIT id="enu1" name="ArtistEntry">
      <LINK id="ln1" to="cru1">
        <LINKPARAMETER id="par1" source="fld1" target="cru1.att2"/>
        <LINKPARAMETER id="par2" source="fld2" target="cru1.att3"/>
      </LINK>
      <FIELD id="fld1" name="FName"/> <FIELD id="fld2" name="LName"/>
    </ENTRYUNIT>
  </PAGE>
  <OPERATIONUNITS>
    <CREATEUNIT entity="ent1" id="cru1" name="CreateArtist">
      <KO-LINK id="kln1" to="page1"/> <OK-LINK id="oln1" to="dau1"/>
    </CREATEUNIT>
  </OPERATIONUNITS>
  <PAGE id="page2" name="CreationResultsPage">
    <DATAUNIT entity="ent1" id="dau1" name="ArtistDetail"/>
  </PAGE>
</SITEVIEW>
```

A test set S is a set of such schemas. In the experimentation, the test set comprises the 38 WebML schemas used in the WebRatio factory. To give an idea, the first group of test schemas contains cases developed to verify the core features of WebML. Other test schemas derive from the addition of new features to the tool or to the language. Finally, the verification of some bug introduced a number of ad-hoc test cases.

The output of the tested program P is a Web application $a = P(d)$, and the oracle used to verify the output is any program capable of deciding whether the application a conforms to P, that is, whether it implements the schema d correctly. In the experimentation, the output is a Web application for the J2EE platform automatically produced by the WebRatio code generator from a WebML schema; for testing purposes, such an application is associated with

a fixed-content data source. The oracle is a program that runs an input script representing a significant user navigation of the generated application; the oracle checks XPath logical expressions on the HTML code returned by the Web server. If any XPath expression evaluates to false, the test succeeds, otherwise the code generator is considered to behave correctly with respect to the supplied test schema[4]. Intuitively, each oracle navigates the Web application and verifies whether the displayed pages comprise the expected content. The investigation of the oracle problem is well beyond the scope of the paper.

3.1   Setting Up the Experimentation

In the ideal world, testing the code generator would require inventing all possible conceptual schemas, generating the corresponding applications, and checking them with the oracle. Since an exhaustive testing is inherently infeasible, the testing problem has been reformulated as that of quantitatively assessing the degree of confidence with respect to a given test set S by following a syntax-driven approach. In this scenario, the testing problem can be summarized by the following questions:

- What fraction of the WebML language is covered by the test set?
- What is the minimal subset of the test set sufficient for achieving the same coverage as the whole set?
- How does the coverage change if one considers only the fragment of the WebML language "used in practice"?

Syntax-driven testing applied to Web code generation requires that each non-terminal symbol in the WebML grammar (i.e. each possible WebML primitive) be used at least once by some test in the test set. The WebML grammar is context-free and each rule (production) models a single WebML construct (e.g., page, unit, link). The grammar is rendered graphically by means of a *Direct Occurrence Graph (DOG)* to highlight the dependencies between rules (explained later) and the usages by the different test cases.

The DOG is a graph built by representing grammar productions as nodes and their relations as edges. For example, two productions $p_1 := A \rightarrow B$ and $p_2 := B \rightarrow C$ are connected by a directed edge from p_1 to p_2 because the right-hand side of p_1 contains an occurrence of the non terminal symbol expanded by p_2. Direct occurrence relationships generate possibly cyclic graphs.

When applied to the WebML grammar, the DOG shows all the possible uses of each WebML construct with respect to the other primitives of the language. For example, Figure 4 shows the representation of the SITEVIEW element, displayed in the Java tool developed for supporting coverage analysis. We report, as example, the grammar production for element SITEVIEW (generated from element SITEVIEW of the DTD fragment shown above).

---

[4] In testing theory, a test succeeds *if it reveals a failure*

```
SITEVIEW -> OPERATIONUNITS, TRANSACTION*, AREA*, PAGE*,
 GLOBALPARAMETER*, PROPERTY*, COMMENT?, <SITEVIEW@id>, <SITEVIEW@name>?,
 <SITEVIEW@homePage>?, <SITEVIEW@protected>, <SITEVIEW@secure>,
 <SITEVIEW@localize>, <SITEVIEW@presentation:style-sheet>?,
 <SITEVIEW@presentation:page-layout>?, <SITEVIEW@graphmetadata:go>?
```

The upper part of the graph shows two *incoming* edges: the SITEVIEW can be contained in a NAVIGATION element or be referenced by the siteView attribute of a LOGOUTUNIT. The lower part of the graph shows outgoing edges, i.e., the elements "used" by the SITEVIEW construct. Proceeding counterclockwise, we encounter the DTD elements nested inside the SITEVIEW element (in white), the homePage attribute (referencing a PAGE element), the textual attributes (in grey), and the enumeration attributes (in light gray). The DOG visualization tool is used to visually present the coverage of the WebML grammar provided by a given test set by decorating and annotating the nodes and edges of the graph. The three kinds of coverage measures that we are going to introduce (and that our tool calculates) mirror the concepts of statement, branch and path coverage, usually used in testing source code.
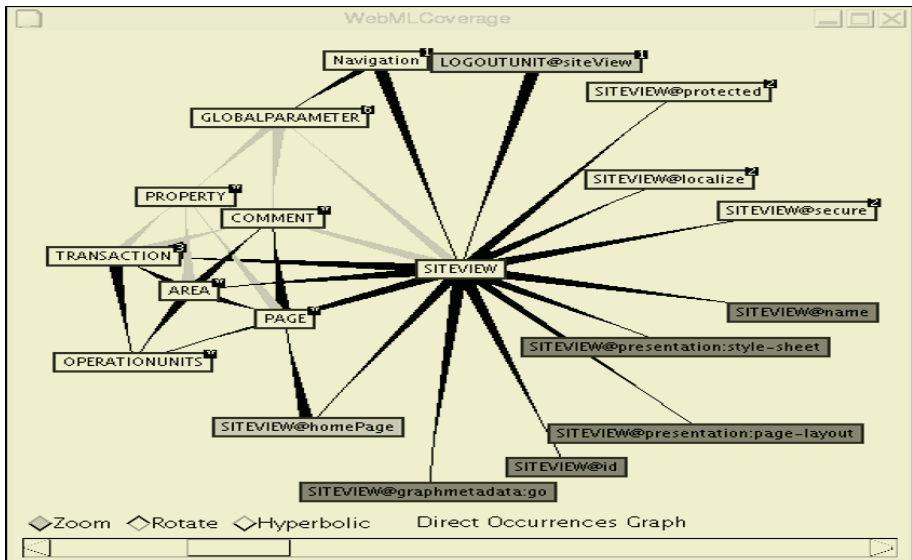

Fig. 4. An example of Direct Occurrence Graph

Rule Coverage. The simplest measure of the confidence provided by a test set can be obtained by assessing the percentage of the WebML grammar rules covered by the WebML schemas in the test cases. We call this measure *rule coverage*. Despite its simplicity, rule coverage analysis can already return important information, i.e., the set of elements of the conceptual model that are never used during the testing session, thus giving the first guidelines for improving the

completeness of the test set. The processing of the WebML syntax led to a DOG with 528 nodes. The WebRatio test set induced a node coverage of 89.2%, thus highlighting the presence of 58 untested nodes (6 constructs and 52 attribute values). By manually inspecting the uncovered nodes with the DOG navigator, the WebRatio staff recognized the absence of test cases for a few, scarcely used, features of the WebML language. Minimality analysis revealed that a subset composed of 9 out of 38 test cases would provide the same rule coverage percentage. This result conflicted with the empirical evidence of the usefulness of the remaining 29 test cases, which prompted us to define more precise coverage measures.

Edge Coverage. To make coverage more precise, we took into account the context in which a WebML construct is used, by considering not only the nodes of the DOG, but also its edges. Intuitively, an edge from construct A to construct B represents the usage of B in the context of A. Therefore, a different edge (say $C \rightarrow B$) may represent a different usage of the same construct B in another context. For instance, the WebML construct PAGE can be used in a variety of ways: nested in a site view, in an area, or as a sub-page of a page. All these situations correspond to different DOG edges. This notion of context is crucial in any real modelling language, where the same primitive can be combined in multiple ways. We call the percentage of covered edges in the DOG *edge coverage*. Covering each edge means building a test set that uses each pair of language constructs in all legal combinations (i.e. combinations permitted by the WebML DTD specification). This criterion is nearly equivalent to Lämmel's context-dependent rule coverage. The DOG used in our experiment had 984 edges. The complete test set induces an edge coverage of 76.93%, thus showing the intrinsic significance of the empirical test set, which was designed without any systematic method for quantitatively assessing confidence. Minimality analysis showed that there are 4 equivalent test sets that provide the same 76.93% coverage measure. Each minimal set contains 16 Web applications. Thus, regardless of the specific minimal set chosen, edge coverage is not sufficient to justify the remaining 22 test applications, which prompted us for further investigation. Edge coverage analysis also pointed out a significant overlap of test cases because several test schemas, taken alone, cover about 500 edges out of 984. Furthermore, the results of the analysis supported the systematic addition of new test cases. The edge coverage percentage was easily improved by designing a few test cases for the uncovered edges, also reducing the overlaps among test schemas.

Path Coverage. As a final evaluation, we considered an additional coverage measure to investigate not only relations between pairs of concepts, but also possible combinations of groups of WebML constructs, because the designers' experience suggested that the subtlest errors originate from unexpected complex interactions of multiple language concepts. The *path coverage* measure addresses this issue. It is defined as the percentage of *paths* covered in the DOG. Covering all paths intuitively means that the test set exercises each construct in all its

legal combinations with all the other constructs. 100% path coverage is clearly an infeasible requirement in most practical cases, even after breaking cycles. Nevertheless we discovered interesting information on the effectiveness of the test set. In the experimentation, our tool calculated more than 12 million legal paths, of which the test set exercises less than 1%. Minimality analysis was more revealing. With the help of our software tool, we identified one redundant test schema, which could be removed from the test set safely, and 6 test schemas that contribute a very low number of new paths to the coverage. Although theoretically speaking the latter test cases cannot be removed from the test set in a totally safe way, their contribution is marginal, as they are almost totally overlapping with other cases. This is an example of the optimization guidelines provided by the path coverage analysis to the developers of the testing set. It is much more important to evaluate the inadequacy of a given test case, rather than demonstrating its adequacy.

3.2   Baseline Definition with Real Web Applications

To improve the coverage figures, we evaluated the hypothesis that only a subset of all possible WebML paths is exercised by real developers. We focused on 46 real-world Web applications generated with WebRatio and performed the analysis on a reduced DOG comprising only the paths actually included in such applications. This path coverage measurement over the reduced DOG better assesses the impact of testing on real-world Web development. Path coverage jumped to a reasonable 33.43% on the reduced DOG, which strongly increases the confidence on the test set manually crafted by WebRatio developers, who probably know well which parts of the language are actually used in practice. We also determined that each real-world Web application was covered by the test cases for an average of 50.08%, with values ranging from 41.51% to 54.74%.

Real-world path coverage can also be used for assessing the necessity of each test case. We found that 3 of the 6 test cases with low path coverage contribution do not exercise any new path in the real-world subset, so they appear superfluous also with respect to the real-world usage of WebML.

3.3   Limits of Grammar-Based Systematic Testing

The experimentation also revealed some interesting limits of grammar-based testing. The evaluation cannot be applied to lexical errors, e.g., those errors that depend on particular formats of string values. For example, grammar-based testing does not uncover bugs caused by character encoding, multi-word fields, or the interpretation of special characters. Such lexical errors must be addressed by a specific group of tests.

4   Conclusions and Future Work

In this paper we have addressed the problem of systematically testing Web applications in the model-driven scenario, where testing splits into schema verification

and code generator verification. We have provided both the formal underpinning of code generator testing and the results of an in-depth experimentation.

To the best of our knowledge, our study is the first formal investigation of testing in a *model&generate* Web development environment. Systematic testing has already been applied to schema validation by Ricca and Tonella [7] by representing the structure of the single Web application with an ad-hoc UML schema. Others [8] follow the same approach with different representations. Further experimental work is ongoing, to reach complete path coverage for the core paths of WebML and for the paths statistically more significant in real-world applications. Slight extensions of the implemented tools will enable the quantitative assessment of the testing confidence of individual Web applications, starting from their conceptual model and test sets defined during requirements analysis.

From the theoretical viewpoint, a promising research direction is the quantitative evaluation of the relation between the generator's correctness measures and other Web application quality metrics (e.g., usability). Finally, a further direction involves the connection of grammar-based testing with white-box testing in the field of Web applications, a study that could bridge two of the most promising approaches in software testing.

References

1. S. Ceri, P. Fraternali, et al.: Designing Data-Intensive Web Application. Morgan Kauffman (2003)
2. Lanzi, P., Matera, M., Maurino, A.: A framework for exploiting conceptual modeling in the evaluation of web application quality. In: Proc. ICWE 2004. Volume 3140 of LNCS., Springer Verlag (2004) 50–54
3. Masand, B.M., Spiliopoulou, M., eds.: Web Usage Analysis and User Profiling, WEBKDD'99, San Diego, CA, August 15, 1999. In Masand, B.M., Spiliopoulou, M., eds.: WEBKDD. Volume 1836 of LNCS., Springer (2000)
4. Myers, G.J., Sandler, C.: The Art of Software Testing. John Wiley & Sons (2004)
5. Beizer, B.: Black-box Testing: Techniques for Functional Testing of Software and Systems. John Wiley & Sons, Inc. (1995)
6. Lämmel, R.: Grammar testing. In: FASE '01: Proc. 4th Int. Conf. on Fundamental Approaches to Software Engineering. (2001) 201–216
7. Ricca, F., Tonella, P.: Analysis and testing of web applications. In: ICSE '01: Proceedings of the 23rd International Conference on Software Engineering, IEEE Computer Society (2001) 25–34
8. Liu, C.H., Kung, D.C., Hsia, P., Hsu, C.T.: Structural testing of web applications. In: ISSRE '00: Proc. 11th Int. Symp. on Software Reliability Engineering (ISSRE'00). (2000) 84