A Fast Algebraic Web Verification Service*

M. Alpuente[1], D. Ballis[2], M. Falaschi[3], P. Ojeda[1], and D. Romero[1]

[1] DSIC, Universidad Politécnica de Valencia
Camino de Vera s/n, Apdo. 22012, 46071 Valencia, Spain.
{alpuente, pojeda, dromero}@dsic.upv.es
[2] Dip. Matematica e Informatica
Via delle Scienze 206, 33100 Udine, Italy.
demis@dimi.uniud.it
[3] Dip. di Scienze Matematiche e Informatiche
Pian dei Mantellini 44, 53100 Siena, Italy.
moreno.falaschi@unisi.it

Abstract. In this paper, we present the rewriting-based, Web verification service WebVerdi-M, which is able to recognize forbidden/incorrect patterns and incomplete/missing Web pages. WebVerdi-M relies on a powerful Web verification engine that is written in Maude, which automatically derives the error symptoms. Thanks to the AC pattern matching supported by Maude and its metalevel facilities, WebVerdi-M enjoys much better performance and usability than a previous implementation of the verification framework. By using the XML Benchmarking tool xmlgen, we develop some scalable experiments which demonstrate the usefulness of our approach.

1 Introduction

The automated management of data-intensive Web sites is an area to which rule-based technology has a significant potential to contribute. It is widely accepted today that declarative representations are the best way to specify the structural aspects of Web sites as well as many forms of Web-site content. As an additional advantage, rule-based languages such as Maude [8] offer an extremely powerful, rewriting-based "reasoning engine" where the system transitions are represented/derived by rewrite rules indicating how a configuration is *transformed* into another.

In previous work [2, 4], we proposed a rewriting-based approach to Web-site verification and repair. In a nutshell, our methodology w.r.t. a given formal specification is applied to discover two classes of important, semantic flaws in Web sites. The first class consists of correctness errors (forbidden information

---

* This work has been partially supported by the EU (FEDER) and Spanish MEC TIN-2004-7943-C04-02 project, the Generalitat Valenciana under grant GV06/285, and Integrated Action Hispano-Alemana HA2006-0007. Daniel Romero is also supported by ALFA grant LERNet AML/19.0902/97/0666/II-0472-FA.

that occurs in the Web site), while the second class consist of completeness errors (missing and/or incomplete Web pages). This is done by means of a novel rewriting-based technique, called *partial rewriting*, in which the traditional pattern matching mechanism is replaced by a suitable technique based on the *homeomorphic embedding* relation for recognizing patterns inside semistructured documents. The new prototype WebVerdi-M relies on a strictly more powerful Web verification engine written in Maude [8] which automatically derives the error symptoms of a given Web site. Thanks to the AC pattern matching supported by Maude and its metalevel features, we have significantly improved both the performance and the usability of the original system. By using SOAP messages and other Web-related standards, a Java Web client that interacts with Web verification service has been made publicly available within the implementation.

Although there have been other recent efforts to apply formal techniques to Web site management [10, 12, 14, 18], only few works addressed the semantic verification of Web sites before. The key idea behind WebVerdi-M is that rule-based techniques can support in a natural way not only intuitive, high level Web site specification, but also efficient Web site verification techniques.

VeriWeb [18] explores interactive, dynamic Web sites using a special browser that systematically explores all paths up to a specified depth. The user first specifies some properties by means of *SmartProfiles*, and then the verifier traverses the considered Web site to report the errors as sequences of Web operations that lead to a page which violates a property. Navigation errors and page errors can be signaled, but tests are performed only at the http-level. In [14], a declarative verification algorithm is developed which checks a particular class of integrity constraints concerning the Web site's structure, but not the contents of a given instance of the site. In [10], a methodology to verify some semantic constraints concerning the Web site contents is proposed, which consists of using inference rules and axioms of natural semantics. The framework XLINKIT [12, 19] allows one to check the consistency of distributed, heterogeneous documents as well as to fix the (possibly) inconsistent information. The specification language is a restricted form of first order logic combined with Xpath expressions [23] where no functions are allowed.

The paper is organized as follows. Section 2 presents some preliminaries, and in Section 3 we briefly recall the rewriting-based, Web-site verification technique of [2]. In Section 4, we discuss the efficient implementation in Maude (by means of AC pattern matching) of one of the key ingredients of our verification engine: the *homeomorphic embedding* relation, which we use to recognize patterns within semi-structured documents. Section 5 briefly describes the service-oriented architecture of our verification prototype WebVerdi-M. Finally, Section 6, we present an experimental evaluation of the system on a set of benchmarks which shows impressive performance (e.g. less than a second for evaluating a tree of some 30,000 nodes). An extended version of this work can be found in [3].

2 Preliminaries

By $\mathcal{V}$ we denote a countably infinite set of variables and Σ denotes a set of *function symbols* (also called *operators*), or *signature*. We consider varyadic signatures as in [9] (i.e., signatures in which symbols do not have a fixed arity).

$\tau(\Sigma, \mathcal{V})$ and $\tau(\Sigma)$ denote the *non-ground term algebra* and the *term algebra* built on $\Sigma \cup \mathcal{V}$ and Σ, respectively. Terms are viewed as labelled trees in the usual way. Given a term t, we say that t is *ground*, if no variable occurs in t. A *substitution* $\sigma \equiv \{X_1/t_1, X_2/t_2, \ldots\}$ is a mapping from the set of variables $\mathcal{V}$ into the set of terms $\tau(\Sigma, \mathcal{V})$ satisfying the following conditions: (i) $X_i \neq X_j$, whenever $i \neq j$, (ii) $X_i\sigma = t_i$, $i = 1, ..n$, and (iii) $X\sigma = X$, for all $X \in \mathcal{V} \setminus \{X_1, \ldots, X_n\}$. An *instance* of a term t is defined as $t\sigma$, where σ is a substitution. By $Var(s)$ we denote the set of variables occurring in the syntactic object s. Syntactic equality between objects is represented by $\equiv$.

3 Rule-based Web site verification

In this section, we briefly recall the formal verification methodology proposed in [2], which allows us to detect forbidden/erroneous information as well as missing information in a Web site. This methodology is able to recognize and exactly locate the source of a possible discrepancy between the Web site and the properties required in the Web specification. An efficient and elegant implementation in Maude of such a methodology is described in Section 4.

We assume a Web page to be a well-formed *XML document* [22], since there are plenty of programs and online services that are able to validate XML syntax and perform link checking (e.g. [24],[21]). Moreover, as XML documents are provided with a tree-like structure, we can straightforwardly model them as ground Herbrand terms of a given term algebra.

The Web specification language. A Web specification is a triple (I_N, I_M, R), where I_N and I_M are a finite set of correctness and completeness rules, and the set R contains the definition of some auxiliary functions.

The set I_N describes constraints for detecting erroneous Web pages (*correctNess rules*). A correctness rule has the following syntax: $l \rightharpoonup error \mid C$ where l is a term, *error* is a reserved constant, and C is a (possibly empty) finite sequence (wich could contains membership tests of the form $X \in rexp$ w.r.t. a given regular language $rexp$;[4] and/or equations/inequalities over terms). When C is empty, we simply write $l \rightharpoonup error$. Informally, the meaning of a correctness rule is the following: whenever (i) a "piece" of a given Web page can be "recognized" to be an instance $l\sigma$ of l, and (ii) the corresponding instantiated condition $C\sigma$ holds, then Web page p is marked as an incorrect page.

The third set of rules I_M specifes some properties for discovering incomplete/missing Web pages (*coMpleteness rules*). A completeness rule is defined as

---

[4] Regular languages are represented by means of the usual Unix-like regular expression syntax.

$l \rightharpoonup r \ \langle \mathtt{q} \rangle$ where l and r are terms and $\mathtt{q} \in \{\mathtt{E}, \mathtt{A}\}$. Completeness rules of a Web specification formalize the requirement that some information must be included in all or some pages of the Web site. We use attributes $\langle \mathtt{A} \rangle$ and $\langle \mathtt{E} \rangle$ to distinguish "universal" from "existential" rules, as explained below. Right-hand sides r of completeness rules can contain functions, which are defined in R. In addiction, some symbols in the right-hand sides of the rules may be marked by means of the symbol $\sharp$. Marking information of a given rule r is used to select the subset of the Web site in order to check the condition formalized by r. Intuitively, the interpretation of a universal rule (respectively, an existential rule) w.r.t. a Web site W is as follows: if (an instance of) l is "recognized" in W, (an instance of) the irreducible form of r must also be "recognized" in *all* (respectively, *some*) of the Web pages that embed (an instance of) the marked structure of r.

Web Verification Methodology. Diagnoses are carried out by running Web specifications on Web sites. The operational mechanism is based on a novel, flexible matching technique [2] that is able to "recognize" the partial structure of a term (Web template) within another and select it by computing *homeomorphic embeddings* (cf. [16]) of Web templates within Web pages.

Homeomorphic embedding relations allow us to verify whether a template is somehow "enclosed" within another one. Our embedding relation $\trianglelefteq$ closely resembles the notion of *simulation* (for the formal definition, see [2]), which has been widely used in a number of works about querying, transformation, and verification of semistructured data (cf. [6, 1, 15, 5]). Let us illustrate the embedding relation $\trianglelefteq$ by means of a rather intuitive example.

Example 1. Consider the following Web templates (called s_1 and s_2, respectively): $hpage(surname(Y), status(prof), name(X), teaching)$ and
$hpage(\ name(mario), surname(rossi), status(prof), teaching(course(logic1),$
$course(logic2)), hobbies(hobby(reading), hobby(gardening)))$
Note that $s_1 \trianglelefteq s_2$, since the structure of s_1 can be recognized inside the structure of s_2, while $s_2 \not\trianglelefteq s_1$.

It is important to have an efficient implementation of *homeomorphic embedding* because it is used repeatedly during the verification process as described in the following.

First, by using the homeomorphic embedding relation $\trianglelefteq$, we check whether the left-hand side l of some Web specification rule is embedded into a given page p of the considered Web site. When the embedding test $l \trianglelefteq p$ succeeds, by extending the proof, we construct the biggest substitution[5] σ for the variables in $Var(l)$, such that $l\sigma \trianglelefteq p$. Then, depending on the nature of the Web specification rule (correction or completeness rule), it is as follows:

(Correction rule) evaluating the condition of the rule (instantiated by σ); a correctness error is signalled in the case when the error condition is fulfilled.

---

[5] The substitution σ is easily obtained by composing the bindings X/t, which can be recursively gathered during the *homeomorphic embedding* test $X \trianglelefteq t$, for $X \in l$ and $t \in p$.

(Completeness rule) by a new homeomorphic embedding test, checking whether the right-hand side of the rule (instantiated by σ) is recognized in some page of the considered Web site. Otherwise, a completeness error is signalled. Moreover, from the incompleteness symptom computed so far, a fixpoint computation is started in order to discover further missing information, which may involve the execution of other completeness rules.

4 Verifying Web sites using Maude

Maude is a high-performance reflective language supporting both equational and rewriting logic programming, which is particularly suitable for developing domain-specific applications [20, 11]. In addiction, the Maude language is not only intended for system prototyping, but it has to be considered as a real programming language with competitive performance. In the rest of the section, we recall some of the most important features of the Maude language which we have conveniently exploited for the optimized implementation of our Web site verification engine.

Equational attributes. Let us describe how we model (part of) the internal representation of XML documents in our system. The chosen representation slightly modifies the data structure provided by the Haskell HXML Library [13] by adding commutativity to the standard XML tree-like data representation. In other words, in our setting, the order of the children of a tree node is not relevant: e.g., $f(a, b)$ is "equivalent" to $f(b, a)$.

```
fmod TREE-XML is
sort XMLNode .
op RTNode : -> XMLNode .                    -- Root (doc) information item
op ELNode _ _ : String AttList -> XMLNode . -- Element information item
op TXNode _ : String -> XMLNode .           -- Text information item
--- ...  definitions of the other XMLNode types omitted ...
sorts XMLTreeList XMLTreeSeq XMLTree .
op Tree (_) _ : XMLNode XMLTreeList - > XMLTree .
subsort XMLTree < XMLTreeSeq .
op _,_ : XMLTreeSeq XMLTreeSeq -> XMLTreeSeq    [comm assoc id:null] .
op null : -> XMLTreeSeq .
op [_] : XMLTreeSeq -> XMLTreeList .
op [] : -> XMLTreeList .
endfm
```

In the previous module, the XMLTreeSeq constructor `_,_` is given the equational attributes `comm assoc id:null`, which allow us to get rid of parentheses and disregard the ordering among `XML` nodes within the list. The significance of this optimization will be clear when we consider rewriting XML trees with AC pattern matching.

AC pattern matching. The evaluation mechanism of Maude is based on rewriting modulo an equational theory E (i.e. a set of equational axioms), which is accomplished by performing *pattern matching modulo* the equational theory E. More precisely, given an equational theory E, a term t and a term u, we say that t *matches* u *modulo* E (or that t E*-matches* u) if there is a substitution σ such that $\sigma(t) =_E u$, that is, $\sigma(t)$ and u are equal modulo the equational theory E. When E contains axioms for associativity and commutativity of operators, we talk about *AC pattern matching*. AC pattern matching is a powerful matching mechanism, which we employ to inspect and extract the partial structure of a term. That is, we use it directly to implement the notion of homeomorphic embedding of Section 3.

Metaprogramming. Maude is based on rewriting logic [17], which is reflective in a precise mathematical way. In other words, there is a finitely presented rewrite theory $\mathcal{U}$ that is universal in the sense that we can represent in $\mathcal{U}$ (as a data) any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself), and then mimick in $\mathcal{U}$ the behavior of $\mathcal{R}$. We have used the metaprogramming capabilities of Maude to implement the semantics of correctness as well as completeness rules (e.g. implementing the homeomorphic embedding algorithm, evaluating conditions of conditional rules, etc.). Namely, during the partial rewriting process, functional modules are dynamically created and run by using the meta-reduction facilities of the language.

Now we are ready to explain how we implemented the homeomorphic embedding relation of Section 3, by exploiting the aforementioned Maude high-level features.

Homeomorphic embedding implementation. Let us consider two XML document templates l and p. The critical point of our methodology is to (i) discover whether $l \trianglelefteq p$ (i.e. l is embedded into p); (ii) find the substitution σ such that $l\sigma$ is the instance of l recognized inside p, whenever $l \trianglelefteq p$.

Given l and p, our proposed solution can be summarized as follows. By using Maude metalevel feactures, we first dynamically build a module `M` that contains a single rule of the form

$$\mathtt{eq\ l} = \mathtt{sub}(''\mathtt{X_1}''/\mathtt{X_1}), \ldots, \mathtt{sub}(''\mathtt{X_n}''/\mathtt{X_n}), \qquad \mathtt{X_i} \in \mathtt{Var(l)}, \mathtt{i} = \mathtt{1}, \ldots \mathtt{n},$$

where `sub` is an associative operator used to record the substitution σ that we want to compute. Next, we try to reduce the XML template p by using such a rule. Since l and p are internally represented by means of the binary constructor `_,_` that is given the equational attributes `comm assoc id:null` (see Section 4), the execution of module `M` on p essentially boils down to computing an AC-matcher between l and p. Moreover, since AC pattern matching directly implements the homeomorphic embedding relation. The execution of `M` corresponds to finding all the homeomorphic embeddings of l into p (recall that the set of AC matchers of two compatible terms is not generally a singleton). Additionally, as a side effect of the execution of `M`, we obtain the computed substitution σ for free as the sequence of bindings for the variables $\mathtt{X_i}$, $\mathtt{i} = \mathtt{1}, \ldots, \mathtt{n}$ which occur in the instantiated rhs

$$\mathtt{sub}(''\mathtt{X_1}''/\mathtt{X_1})\sigma, \ldots, \mathtt{sub}(''\mathtt{X_n}''/\mathtt{X_n})\sigma, \qquad \mathtt{X_i} \in \mathtt{Var(l)}, \mathtt{i} = \mathtt{1}, \ldots \mathtt{n},$$

of the dynamic rule after the partial rewriting step.

Example 2. Consider again the XML document templates s_1 and s_2 of Example 1. We build the dynamic module `M` containing the rule

$$\mathtt{eq\ hpage(surname(Y), status(prof), name(X), teaching)} = \mathtt{sub}(''\mathtt{Y}''/\mathtt{Y}), \mathtt{sub}(''\mathtt{X}''/\mathtt{X})\ .$$

Since $s_1 \trianglelefteq s_2$, there exists an AC-match between s_1 and s_2 and, hence, the result of executing `M` against the (ground) XML document template s_2 is the computed substitution: $\mathtt{sub}(\text{"}\mathtt{Y}\text{"}/\mathtt{rossi}), \mathtt{sub}(\text{"}\mathtt{X}\text{"}/\mathtt{mario})$.

5 Prototype implementation

The verification methodology presented so far has been implemented in the prototype WebVerdi-M (Web Verification and Rewriting for Debugging Internet sites with Maude). In developing and deploying the system, we fixed the following requirements: 1) define a system architecture as simple as possible, 2) make the Web verification service available to every Internet requestor, and 3) hide the technical details from the user. In order to fulfill the above requirements, we developed the Web verification system WebVerdi-M as a Web service.

5.1 WebVerdi-M Architecture

WebVerdi-M is a service-oriented architecture that allows one to access the core verification engine Verdi-M as a reusable entity. WebVerdi-M can be divided into two layers: *front-end* and *back-end*. The *back-end* layer provides web services to support the *front-end* layer. This architecture allow clients on the network to invoke the Web service functionality through the available interfaces.

Figure 1 illustrates the overall architecture of the system. For the reader interested in more detail, the types of messages and the specific message exchange patterns that are involved in interacting with WebVerdi-M can be found in [3].

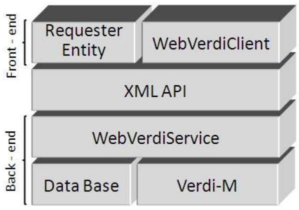

Fig. 1. Components of WebVerdi-M

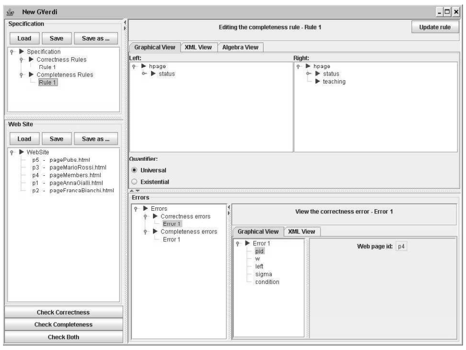

Fig. 2. WebVerdiClient Snapshot

WebVerdiService. Our web service exports six operations that are network-accessible through standardized XML messaging. The Web service acts as a single access point to the core engine Verdi-M. Following the standards, the architecture is also platform and language independent so as to be accessible via scripting environment as well as via client applications across multiple platforms.

XML API. In order for successful communications to occur, both the WebVerdiService and WebVerdiClient (or any user) must agree to a common format for the messages being delivered so that they can be properly interpreted at each end. The WebVerdiService Web service is developed by defining an API that encompasses the executable library of the core engine. This is achieved by making use of Oracle JDeveloper, including the generation of WSDL for making the API available. The OC4J Server (the web server integrated in Oracle JDeveloper) handles all procedures common to Web service development. Synthesized error symptoms are also encoded as XML documents in order to be transferred from the WebVerdiService Web service to client applications as an XML response by means of the SOAP protocol.

Verdi-M. Verdi-M is the most important part of the tool. Here is where the verification methodology is implemented. This component is implemented in Maude language and is independent of the other system components.

WebVerdiClient. WebVerdiClient is a Web client that interacts with the Web service to use the capabilities of Verdi-M. Our main goal was to provide an *intuitive* and *friendly* interface for the user. WebVerdiClient is provided with a versatile, new graphical interface that offers three complementary views for both the specification rules and the pages of the considered Web site: the first one is based on the typical idea of accessing contents by using folders trees; the second one is based on XML, and the third one is based on term algebra syntax. A snapshot of WebVerdiClient is shown in Figure 2.

DB. The WebVerdiService Web service needs to transmit abundant XML data over the Web to and from client applications. In order to avoid overhead and to provide better performance to the user, we use a local *MySql* data base where the Web site and Web errors are temporarily stored at the server side.

6 Experimental evaluation

In order to evaluate the usefulness of our approach in a realistic scenario (that is, for sites whose data volume exceeds toy sizes), we have benchmarked our system by using several correctness as well as completeness rules of different complexity for a number of XML documents randomly generated by using the XML documents generator xmlgen available within the XMark project [7]. The tool xmlgen is able to produce a set of XML data, each of which is intended to challenge a particular primitive of XML processors or storage engines by using different scale factors.

Table 1 shows some of the results we obtained for the simulation of three different Web specifications $WS1$, $WS2$ and $WS3$ in five different, randomly generated XML documents. Specifically, we tuned the generator for scaling factors from 0.01 to 0.1 to match an XML document whose size ranges from 1Mb –corresponding to an XML tree of about 31000 nodes– to 10Mb –corresponding to an XML tree of about 302000 nodes. An exhaustive evaluation, including comparison with related systems, can be found in

`http://www.dsic.upv.es/users/elp/webverdi-m/`.

Both Web specifications $WS1$ and $WS2$ aim at checking the verification power of our tool regarding data correctness, and thus include only correctness rules. The specification rules of $WS2$ contain more complex and more demanding constraints than the ones fomalized in $WS1$, with involved error patterns to match, and conditional rules with a number of membership tests and functions evaluation. The Web specification $WS3$ aims at checking the completeness of the randomly generated XML documents. In this case, some critical completeness rules have been formalized which recognize a significant amount of missing information.

Size	**Nodes**	**Scale factor**	**Time**		
			$WS1$	$WS2$	$WS3$
1 Mb	30, 985	0.01	0.930 s	0.969 s	165.578 s
3 Mb	90, 528	0.03	12.604 s	2.842 s	1768.747 s
5 Mb	150, 528	0.05	5.975 s	5.949 s	4712.157 s
8 Mb	241, 824	0.08	8.608 s	9.422 s	12503.454 s
10 Mb	301, 656	0.10	12.458 s	12.642 s	21208.494 s

Table 1. Verdi-M Benchmarks

The results shown in Table 1 were obtained on a personal computer equipped with 1Gb of RAM memory, 40Gb hard disk and a Pentium Centrino CPU clocked at 1.75 GHz running Ubuntu Linux 5.10.

Let us briefly comment our results. Regarding the verification of correctness, the implementation is extremely time efficient, with elapsed times scaling lineary. Table 1 shows that the execution times are small even for very large documents (e.g. running the correctness rules of Web specification $WS1$ over a 10Mb XML document with 302000 nodes takes less than 13 seconds). Concerning the completeness verification, the fixpoint computation which is involved in the evaluation of the completeness rules typically burdens the expected performance (see [2]), and we are currently able to process efficiently XML documents whose size is not bigger than 1Mb (running the completeness rules of Web specification $WS3$ over a 1Mb XML document with 31000 nodes takes less than 3 minutes).

Finally, we want to point out that the current Maude implementation of the verification system supersedes and greatly improves our preliminary system, called GVerdi[2, 4], that was only able to manage correctness for small XML repositories (of about 1Mb) within a reasonable time. We are currently working on further improving the performance of our system.

References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web. From Relations to Semistructured Data and XML.* Morgan Kaufmann, 2000.
2. M. Alpuente, D. Ballis, and M. Falaschi. Automated Verification of Web Sites Using Partial Rewriting. *Software Tools for Technology Transfer*, 8:565–585, 2006.
3. M. Alpuente, D. Ballis, M. Falaschi, P. Ojeda, and D. Romero. The Web Verification Service WebVerdi-M. Technical Report DSIC-II/08/07, DSIC-UPV, 2007.
4. D. Ballis and J. García Vivó. A Rule-based System for Web Site Verification. In *Proc. of 1st Int'l Workshop on Automated Specification and Verification of Web Sites (WWV'05)*, volume 157(2). ENTCS, Elsevier, 2005.
5. E. Bertino, M. Mesiti, and G. Guerrin. A Matching Algorithm for Measuring the Structural Similarity between an XML Document and a DTD and its Applications. *Information Systems*, 29(1):23–46, 2004.

6. F. Bry and S. Schaffert. Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. In *Proc. of the Int'l Conference on Logic Programming (ICLP'02)*, volume 2401 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
7. Centrum voor Wiskunde en Informatica. XMark – an XML Benchmark Project, 2001. Available at: `http://monetdb.cwi.nl/xml/`.
8. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. The maude 2.0 system. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in Lecture Notes in Computer Science, pages 76–87. Springer-Verlag, 2003.
9. N. Dershowitz and D. Plaisted. Rewriting. *Handbook of Automated Reasoning*, 1:535–610, 2001.
10. T. Despeyroux and B. Trousse. Semantic Verification of Web Sites Using Natural Semantics. In *Proc. of 6th Conference on Content-Based Multimedia Information Access (RIAO'00)*, 2000.
11. S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker and its implementation. In *Model Checking Software: Proc. 10 th Intl. SPIN Workshop*, volume 2648 of *LNCS*, pages 230–234. Springer, 2003.
12. E. Ellmer, W. Emmerich, A. Finkelstein, and C. Nentwich. Flexible Consistency Checking. *ACM Transaction on Software Engineering*, 12(1):28–63, 2003.
13. Joe English. The HXML Haskell Library, 2002. Available at: `http://www.flightlab.com/ joe/hxml/`.
14. M. Fernandez, D. Florescu, A. Levy, and D. Suciu. Verifying Integrity Constraints on Web Sites. In *Proc. of Sixteenth International Joint Conference on Artificial Intelligence (IJCAI'99)*, volume 2, pages 614–619. Morgan Kaufmann, 1999.
15. M. F. Fernandez and D. Suciu. Optimizing Regular Path Expressions Using Graph Schemas. In *Proc. of Int'l Conf on Data Engineering (ICDE'98)*, pages 14–23, 1998.
16. M. Leuschel. Homeomorphic Embedding for Online Termination of Symbolic Methods. In T. Æ. Mogensen, D. A. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation*, volume 2566 of *LNCS*, pages 379–403. Springer, 2002.
17. N. Martí-Oliet and J. Meseguer. Rewriting Logic: Roadmap and Bibliography. *Theoretical Computer Science*, 285(2):121–154, 2002.
18. B. Michael, F. Juliana, and G. Patrice. Veriweb: automatically testing dynamic web sites. In *Proc. of 11th Int'l WWW Conference*. ENTCS, Elsevier, 2002.
19. C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency Management with Repair Actions. In *Proc. of the 25th International Conference on Software Engineering (ICSE'03)*. IEEE Computer Society, 2003.
20. J. Meseguer S. Escobar, C. Meadows. A Rewriting-Based Inference System for the NRL Protocol Analyzer and its Meta-Logical Properties. *Theoretical Computer Science*, 367(1-2):162–202, 2006.
21. Typke und Wicke GbR. Validate/Check XML. Available at: `http://www.xmlvalidation.com/`.
22. World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.0, second edition, 1999. Available at: `http://www.w3.org`.
23. World Wide Web Consortium (W3C). XML Path Language (XPath), 1999. Available at: `http://www.w3.org`.
24. World Wide Web Consortium (W3C). Markup Validation Service, 2005. Available at: `http://validator.w3.org/`.