X-MuT: A Tool for the Generation of XSLT Mutants

Francesca Lonetti, Eda Marchetti
Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo"
Consiglio Nazionale delle Ricerche
via G. Moruzzi, 1 - 56124 Pisa, Italy
{francesca.lonetti, eda.marchetti}@isti.cnr.it

***Abstract*—Mutation testing has been historically applied to many programming languages as a white box testing technique. In this paper, we propose a set of mutation operator classes for XSLT language, and we implement them into a tool called X-MuT. The tool automatically generates the set of mutants and provides facilities to run a given test suite on the mutants and to compute the test suite effectiveness in terms of mutation score. We provide an example of X-MuT application to evaluate the effectiveness of an existing test suite for a XSLT stylesheet transforming the standard MARCXML to the Dublin Core format.**

***Index Terms*—XSLT; Mutation operators; Fault detection effectiveness; Mutation testing.**

I. Introduction

In every application domain, for achieving interoperability among heterogenous information systems, the key emerging notations are: the XML language, which permits to share structured context-rich data and to personalize them with specific tags; the XML Schema, which is used for establishing the rules to which documents should conform; and the XSLT (eXtensible Stylesheet Language Transformations), which is a language for transforming XML documents into other XML documents. These three notations provide the formation of spontaneous or institutionalized communities where users can express their domain-specific markup languages or vocabularies and share repository and databases.

Just to cite only some examples of different application domains, in the healthcare sector, Health Level Seven (HL7) [1] provides XML Schema based international reference models for clinical and administrative documents. In this context, XSLT technologies are commonly adopted to flexibly transform clinical documents so to be imported and exported in the various heterogeneous hospital databases. Another notable example is the sharing of bibliographic information: the world needs to agree on standard formats for exchanging and interpreting bibliographic data elements, and two main formats endorsed by different community are MARC [2] and Dublin Core [3]. Also in this case, XSLTs are used to transform the standard MARC format to the Dublin Core format and viceversa.

A common observation from the various XSLT application domains is the increased complexity of the exchanged information across different systems. This forces the XSLT documents to be larger and more complex so to capture all the specific peculiarities of the transformation of the input documents into the target ones. Thus, XSLT becomes often the core of documents creation and processing. In such situation, the correctness of XSLT documents is an important factor of software quality.

In literature some existing tools for XSLT stylesheets testing are available. For example, UTF-X [4] is an extension of the JUnit Java unit testing framework. It provides functionality for XSL stylesheet unit testing, strongly encouraging a test-first-design principle. Other tools include: Tennison-Tests [5], which allows users to write unit tests in XML and exercise XSLT from Ant; Alster [6] which is a XSLT unit testing framework supporting the creation and execution of XSL stylesheets containing marked test templates; TAXI tool [7] that applies systematic black-box techniques for generating a set of XML instances to be used as test cases for the XSLT validation.

Despite the good results in terms of XSLT testing shown by the above mentioned proposals, an evaluation of the effectiveness of the obtained test suite in terms of the ability to detect faults is missed.

Usually, the standard approach either for comparing different testing strategies or for evaluating the quality of a test suite is the mutation testing [8][9]. The purpose of mutation testing is to simulate the possible errors that programmers could leave in the code during the implementation. To do this, syntactic changes are opportunely introduced in the original program, and a set of faulty programs, called mutants, each containing a single different fault, is derived. To assess the effectiveness of a test strategy, these mutants are executed against the test cases derived by the test methodology and the number of seeded faults, detected during the tests execution, is counted. The mutation score is the ratio between the number of killed mutants and the total number of mutants. An analysis and a survey of the current existing mutation approaches and tools can be found in [10].

However, in the best of our knowledge, so far any tool is available for the automatic mutants generation from XSLT documents. The purpose of this paper is twofold: implement the automatic generation of the XSLT mutants, and provide the user with some facilities to compute the mutation score,

i.e. the percentage of mutants killed by a given test suite.

We first provide a short overview of the characteristics of the XSLT language and we define possible faults. In particular we identify, as describe in Section II, six classes of mutation operators that seed different kinds of faults into the variables definition, arithmetic expressions and values assignment. For each class we implement the automatic generation of mutants from a given XSLT.

For testing purposes we also provide some facilities for detecting the possible seeded faults, i.e. for calculating the number of mutants killed during the execution of a given test suite. However, due to the heterogeneity of the possible applications of the XSLT in the various fields, in this first release of the proposed tool we concentrate on the following situations:

1) The case in which XSLT is used for transforming XML instances into other XML documents that must conform to a specific XML Schema. This feature has been conceived considering in particular the XML-based environment where interpretability and interconnection among repository systems, relie on the intensive use of XML and XML Schema notations. This is, for instance, the typical situation when XML clinical information has to be exchanged among the repositories of several hospitals using different XML Schema formats for data storage. In this specific case, the main purpose of the XSLT is the correct transformation of the input data into output documents compliant to a given XML Schema and not the check of the specific content of documents. Thus, the test cases generation is focused on discovering faults in XSLT stylesheet causing structural modifications of the output documents. Consequently, the verification proposed by our tool has the purpose of detecting the mutants simulating these types of faults.
2) The case in which XSLT is used for transforming documents into other XML-format documents. For instance, this is the case when the XML documents have to be transformed into HTML format documents for being correctly visualized on web sites. In this case, the verification of our tool has the purpose of detecting all differences between the output document of the target XSLT, and the documents derived by its mutants.

In this paper, our intention is to provide automatic facilities that could help in mutation analysis without the pretension to be exhaustive. Improvements of the current proposal and additional facilities for mutants detection are part of our research and future work.

The remaining of the paper is structured as follows: we first describe a set of mutation operator classes for the XSLT language (Section II) and then we present the main structure of a tool called X-MuT (XSLT Mutation Tool) for the automatic implementation of the mutation testing process (Section III). Finally, in Section IV, we present an example of application of X-MuT to some available test suites for a standard XSLT stylesheet, transforming documents from the standard MARCXML format to the Dublin Core format. General conclusions are drawn in Section V.

II. XSLT Mutation Classes

XSLT is the W3C recommendation [11][12] for a XML stylesheet language. It is a powerful language allowing transformation of a XML document into another format such as HTML, XML, or textual. A basic XSLT program is a collection of `template rules`. A template rule has a `pattern` specifying the nodes it matches and a `template` to be instantiated when the pattern is matched. A XSLT processor, transforming a XML document, visits each node of the XML document tree and compares it with the pattern of each template rule in the stylesheet. When the processor finds a node that matches a template rule's pattern, it outputs the rule's template. This template generally includes markups, new data, and some data of the source XML document. XSLT uses XML to describe these rules, templates, and patterns. A full explanation of XSLT is of course beyond the scope of this paper. We refer to the official standard [11][12] for details.

In this section, we describe the XSLT mutation operator classes proposed in this paper. In particular, these operators simulate the actions that are typical sources of errors in the writing of a XSLT stylesheet. Such actions are:

- assignment of variable values;
- typing of arithmetic, logic and relational operators;
- definition of XPath expressions;
- definition of conditional and iterative instructions;
- template definition;
- elements and attribute manipulation.

Mutation operators have been proposed and established for a wide array of languages including Ada, C, Cobol, C#, Fortran, Java, and SQL. The survey in [10] provides an overview of the research that has been conducted on mutation testing including the mutation operators that have been introduced for various languages.

In defining our mutation operators, we considered typical mutation operators designed to modify variables and expressions by replacement, insertion and deletion, and specific operators for the XSLT language. In particular, the typical mutation operators are derived by those for Fortran and Java programs, implemented in *Mothra* [13] and *MuJava* [14] tools respectively. Here, we adapt them to the XSLT language. Specifically, the mutation operators of *Mothra* and *MuJava* that we consider are the following (see Table III of [10]):

- Arithmetic Operator Replacement (AOR), Logical Connector Replacement (LCR), Relational Operator Replacement (ROR) for arithmetic, logical and relational condition management;
- Constant replacement (CRP), Statement Deletion (SDL), Scalar variable replacement (SVR), for modifying variables and statements.

In addition, we define specific operators for the manipulation of XSLT elements, such as XPath Expressions and `template rules`.

In the following, we give a brief overview of the proposed mutant classes:

- VM - Variable Manipulation. These mutation operators modify the `select` attribute of the XSLT `variable` element, eliminating its initialization or replacing its value.
- ALROM - Arithmetic, Logic and Relational Operator Manipulation. These operators modify the arithmetic, logic and relational operations contained in the XSLT program, for example in the `test` attribute of the `if` and `when` elements.
- XPEM - XPath Expression Manipulation. These operators are related to XPath expressions contained in the `select` attribute of the `value-of` element of the XSLT document. They replace a XPath expression with other XPath expressions used in the document or modify a XPath expression replacing the contained node or attribute name.
- CIM - Condition, Iteration Manipulation. The operators of this class manipulate the XSLT elements representing the conditional instructions (`choose` and `if` elements) and the iterative ones (`for each` element). In particular, they delete the `otherwise` element of each `choose` element, generating a mutant for each `choose` element. They replace the `test` attribute value of each `when` or `if` element with another `test` attribute value contained into another `when` in the same `choose` element, or into another `if` element respectively. In the same way, these operators replace the value of the `select` attribute of the `for each` element, with each value of the `select` attribute of the other `for each` elements.
- TM - Template Manipulation. These operators are applied to `template`, `apply-templates` and `call-template` elements. They operate changing the attributes values of these elements.
- EAM - Element, Attribute Manipulation. These operators focus on `element` and `attribute` nodes of XSLT. They replace the `name` attribute value with each value of the same attribute that is in the other `element` and `attribute` nodes.

For a detailed description of mutation operators of each class you can see [15].

III. X-MuT TOOL

In this section, we depict the main architecture of X-MuT. As shown in Figure 2 the tool has five main components:

- **GUI**: which is the interface for the user interaction. As we can see in Figure 1 the user can: select the classes of mutants to be generated; execute the mutants themselves against a test suite; and verify which mutants have been killed by the application of the `Validate` or the `Diff` features provided by X-MuT. Further details will be provide in the rest of this section.
- **MutantGenerator**: This component generates the mutants for a given XSLT. The generation involves only the classes of mutants selected by the user.
- **Executor**: This executes one by one the test cases on the original XSLT and the generated set of mutants.

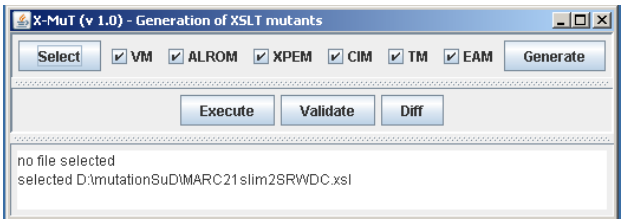

Fig. 1. Main interface of X-MuT tool

- **Validator**: This executes the validation of the transformed XML documents against a specific XML Schema. The validation is executed as follows: the XML instances generated by the `Executor` component are validated against the XML Schema by using an integrated Java XML Schema Validator. Specifically the current version of X-MuT includes the Java API for XML Processing (JAXP) embedded in the Java Platform, Standard Edition version 6.0 [16]. If the transformed XML instance is not compliant with the XML Schema, the corresponding mutant is classified as killed. The list of killed mutants is then provided in a log file.
  In this case, since the purpose is to discover faults causing the structural modifications of the output documents with respect to the output Schema, there is the possibility that some mutants remain undetected. For instance, if a mutant changes only the values in the transformed XML instance but the transformed instance is still compliant to the required output XML Schema, the mutant remains alive.
- **Difference**: This component works as follows: each document transformed by the original XSLT is compared with those obtained by its mutants set for discovering differences among them. If the two documents are different, the corresponding mutant is classified as killed.
  In particular, the current version of X-MuT implements a component performing differences between two XML-format documents. It considers equal two documents if, in their tree representation, all the corresponding node names and values are equal and all the corresponding attribute names and values are equal in the same order.

The validations implemented by the `Validator` and `Difference` components can be both applied if their applicability requirements are satisfied. They represent two level of validation with different targets: the former is focused on the killing of mutants causing structural modifications of the output documents; the latter is conceived for discovering all the differences in terms of nodes name and value between XML-format documents. Note the, the mutation score obtained by `Validator` is included in that of `Difference`.

In Figure 2 we show how the different components interact each others and with the user. First, the user selects in the `GUI` the mutant classes that have to be applied, by ticking the corresponding boxes in the interface, and then, it provides the URL address of the considered XSLT (messages 1 and 2 of

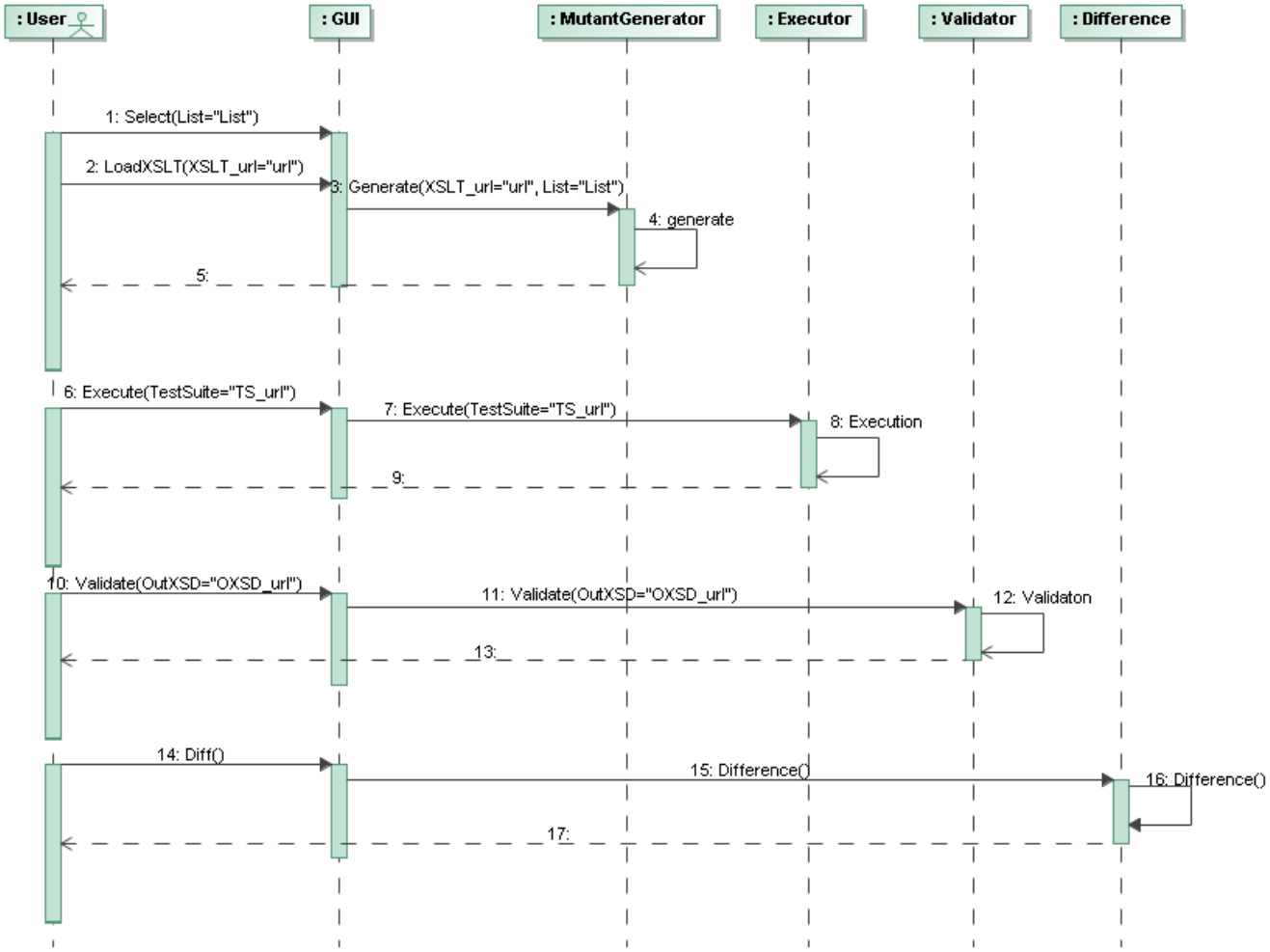

Fig. 2. Sequence diagram of X-MuT tool

Figure 2 respectively).

The list of the classes of mutants selected and the URL address of the XSLT are passed to the `MutantGenerator` that generates the mutants and memorizes them in an internal database (messages from 3 to 5).

The user successively provides the test suite and the `Executor` component executes the test cases on each generated mutant and provides the transformed documents (messages from 6 to 9). At this point, the user can select which kind of verification has to be executed. If the output of the XSLT transformation has to be compliant with a given XML schema, the validation can be executed. In this case the user provides the XML Schema and the component `Validator` executes the validation of the transformed documents with the given XML Schema. The list of killed mutants is then provided to the user in a log file (messages from 10 to 13).

The other possibility is that the user executes the analysis of the transformed documents for detecting differences. In this case the `Difference` component evaluates if the tree representation of the documents obtained by the original XSLT and that of the documents obtained by the mutants are equal. The user receives in a log file the list of the killed mutants (messages from 14 to 17). Note that the `Validator` and the `Difference` components do not work in mutual exclusion and both can be executed.

IV. Using X-MuT: an example

In this section, we show the usage of X-MuT considering a XSLT which transforms MARCXML [2] format to the Dublin Core format [3]. We also evaluate the effectiveness of some of the available test suites by means of the X-MuT facilities. In particular, in Section IV-A we briefly describe the MARC and Dublin Core environments, in Section IV-B we provide some insights into how mutants model the real faults and finally in Section IV-C we discuss about the fault detection effectiveness of the selected test suites.

A. MARC and Dublin Core

MARC means *MAchine-Readable Cataloging*: a MARC record is a machine-readable cataloging record, formatted

according to MARCXML format [2]. It includes a set of standards for representing and exchanging disparate data such as authority, bibliographic, classification, community information, and holdings them in a machine-readable form. MARCXML is a framework for working with MARC data in a XML environment, which consists of many components, including schemas, stylesheets and software tools.

The Dublin Core metadata standard defines a set of elements used for describing a wide range of networked resources. The schemas of the Dublin Core define the structure and syntax of metadata specifications in a formal way.

Along with the usage of MARC records, exchanging information from MARC-based systems to other systems is often required, so MARC Standards Office produces a set of XSLT stylesheets to do conversion from MARC records to other formats, such as MODS (Metadata Object Description Schema) and Dublin Core, as well as the stylesheets for opposite transformation. In particular, there are three available stylesheets transforming metadata from MARCXML to Dublin Core.

For this experiment, we considered the XSLT stylesheet named `MARC21slim2SRWDC.xsl` [2] that transforms the XML documents into instances compliant to the `dc-xsd` XML schema[1].

As test cases we used the sample XML instances available on the web site of MARCXML [2].

B. Mutants Analysis

We used X-MuT to generate the possible mutants for the selected `MARC21slim2SRWDC.xsl` stylesheet. After the removal of the equivalent mutants, for each of the mutation operators classes described in Section II, we get the number of mutants, shown in the second column of Table I and Table II. As can be noticed, X-MuT was not able to derive mutants for the `EAM` class because the elements and attributes of the `MARC21slim2SRWDC.xsl` stylesheet do not satisfy the applicability conditions required by this class. All the other typical faults considered in Section II were represented in the set of mutants obtained by the X-MuT tool. This was a positive and important signal for evaluating how the X-MuT could simulate real faults in the XSLT stylesheet development. However, from the obtained results, it is evident that the number of mutants derived for each class has a big variability: the class `CIM` has an extremely high number of mutants (637), while `TM` has the lowest one (1).

Note that, the number of generated mutants for each class is strictly related to the content of the stylesheet, and thus general conclusions can not be achieved. However, this preliminary analysis evidences the necessity of increasing the mutation operators for the classes specifically conceived for the XSLT language. For instance, `TM` and `XPEM` require some potential new mutation operators in addition to those already implemented in X-MuT.

[1]available at http://dublincore.org/schemas/xmls/qdc/2003/04/02/dc.xsd

C. Analysis of the Fault Detection Effectiveness

In this section, we provide an example of how to use X-MuT for evaluating the fault detection effectiveness of a given test suite. We used the set of mutants derived by X-MuT for the `MARC21slim2SRWDC.xsl` stylesheet, analyzed in the previous section, and we developed a two rounds experiment.

In the first round we defined a test suite, called TS1, considering three of the XML available documents. Specifically we considered: `clasmrc.xml`, `namemrc.xml`, `subjmrc.xml` [2]. We used X-MuT for executing the test suite on the derived mutants and we obtained the mutation score. In the particular case of MARCXML to Dublin Core format transformation, a specific constraint is that the final XML documents must be compliant to the `dc-xsd` XML Schema. Due to this peculiarity, both the validation facilities provided by the X-MuT tool can be used for measuring the fault detection effectiveness of the test suite. We show the obtained results in Table I. In particular, in the third column there is the number of mutants killed by the application of the validation against the `dc-xsd` schema, while in the fourth column there is that relative to the analysis of the differences. In the last column we show the overall number of killed mutants. As we can see, in our experiment, all the mutants killed by the validation against the schema are also killed by the analysis of differences, thus the value reported in the last column is equal to that of the forth one.

However, as a final consideration of this experiment, the number of mutants killed by the application of TS1 is quite small, as reported in the last row of the table. The validation again the schema kills 48 mutants, which is equal to the 6.5% of the total; the `Diff` feature of the tool kills 189 mutants, i.e. 25.5% which is also the overall amount of the mutants killed by TS1.

Because the mutation score of TS1 was under our expectation, a second round of the experiment was planned. Thus, we increased the test suite TS1 by adding two further documents: `collection.xml` and `sandburg.xml` [2]. We called the increased test suite TS2. We repeated the evaluation using TS2 as described above obtaining the results of Table II.

The additional test cases increase the number of killed mutants for the classes `VM`, `ALROM`, `XPEM` and `CIM` so that the final mutation score for TS2 is 66.7%. That is a better performance with respect to that of TS1, even if it is not completely satisfactory. For increasing the percentage of killed mutants, as no further documents were available in the MARCXML web site, the subsequent steps should have been to manually construct ad hoc test cases and add them to TS2 test suite. It is out of the scope of this paper to provide a test suite reaching the highest mutation score. The purpose of this example was to show a typical application of the X-MuT tool for the evaluation of the fault detection effectiveness of a test suite.

For aim of completeness, it is important to evidence that the validation and the `Diff` feature implemented in the current release of X-MuT are not a complete oracle.

TABLE I
MUTANT-KILL RATIOS ACHIEVED BY TS1

Mut_Class	#Mut	#Mut_Kill_Val	#Mut_Kill_Diff	%Tot_Mut_Kill
VM	14	0	4	4
ALROM	52	0	0	0
XPEM	36	0	10	10
CIM	637	48	174	174
TM	1	0	1	1
EAM	-	-	-	-
Total	740	48	189	189

TABLE II
MUTANT-KILL RATIOS ACHIEVED BY TS2

Mut Class	#Mut	#Mut_Kill_Val	#Mut_Kill_Diff	%Tot_Mut_Kill
VM	14	0	10	10
ALROM	52	0	23	23
XPEM	36	0	22	22
CIM	637	48	438	438
TM	1	0	1	1
EAM	-	-	-	-
Total	740	48	494	494

It is possible that the documents transformed by a mutant are not exactly equal to that ones transformed by the original XSLT, but both the verification approaches are not able to detect the differences. The purpose of X-MuT is to provide an help in the analysis of mutants so to discover those that can be considered as killed. Is part of our future work to refine the validation features provided by X-MuT so to provide more accurate and precise measures. However, due to the complexity of the validation problem it is in any case out of scope of X-MuT to completely substitute the manual analysis of the testing results.

V. CONCLUSIONS

Mutation testing is a widespread technique for evaluating the fault detection effectiveness of a given test suite. Current proposals for mutation testing do not consider the XSLT language, which is a common means for transforming XML documents into other XML documents.

We proposed here a tool called X-MuT for generating mutants for a XSLT stylesheet. X-MuT implements a set of classes of mutation operators and provides some facilities for evaluating the mutation score obtained by a test suite execution.

Considering the XSLT stylesheet transforming the standard MARCXML to the Dublin Core format, we have shown the usage of X-Mut for mutants generation and evaluated the fault detection effectiveness of some existing test suites. The obtained results confirm that X-MuT correctly simulates typical faults of the XSLT language. X-MuT can also be a valid help for the automatic detection of the mutants killed by a test suite execution.

Of course such conclusions must be taken in light of the threats to validity of the performed experiment. They must be mitigated against the limited scope of the experiment: here we only apply X-MuT to the simple, even if real, example of the `MARC21slim2SRWDC.xsl` stylesheet. We need to make larger experiments to generalize the statements, so in the future we plan to study larger case studies focusing on the performance analysis of X-MuT.

Moreover, in the current release of X-MuT we only implemented classes of mutation operators for the widespread elements of the XSLT language. It is our intention to complete the mutation operator classes definition, addressing all the possible elements of the XSLT language, and to implement these classes into the next release of the of X-MuT tool.

Finally, as future work we plan to use X-MuT for validating the fault detection effectiveness of greater available XSLT test suites.

ACKNOWLEDGMENT

Authors would like to thank Nicholas Pacini for the work done on the implementation of the X-MuT tool.

This work has been partially funded by EC FP7 under Grant Agreement N. 216287 (TAS^3 - Trusted Architecture for Securely Shared Services) and by PRIN 2007 D-ASAP (Dependable Adaptable Software Architectures for Pervasive Computing) project.

REFERENCES

[1] Health Level Seven, http://www.hl7.org/, accessed April 2010.
[2] MARCXML, http://www.loc.gov/standards/marcxml/, accessed April 2010.
[3] The Dublin Core Metadata Initiative, http://dublincore.org/, accessed April 2010.
[4] Unit Testing Framework - XSLT, http://utf-x.sourceforge.net/, accessed April 2010.
[5] Tennison Tests, http://tennison-tests.sourceforge.net/index.html, accessed April 2010.
[6] Alster-XSLT Unit Testing Framework, http://alster.sourceforge.net/, accessed April 2010.
[7] A. Bertolino, J. Gao, E. Marchetti, and A. Polini, "TAXI–A Tool for XML-Based Testing," in *ICSE COMPANION '07: Companion to the proceedings of the 29th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 53–54.
[8] A. Mathur, *Foundations of software testing: Fundamental Algorithms and Techniques*. Pearson Education, 2008.
[9] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge Univ Pr, 2008.
[10] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," available at http://www.dcs.kcl.ac.uk/pg/jiayue/repository/TR-09-06.pdf, 2009, Techical Report TR-09-06. accessed April, 2010.
[11] W3C Recommendation, "XSL Transformations (XSLT) Version 1.0 November 1999," http://www.w3.org/TR/xslt, accessed April 2010.
[12] ——, "XSL Transformations (XSLT) Version 2.0 January 2007," http://www.w3.org/TR/xslt20/, accessed April 2010.
[13] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt, "An Extended Overview of the Mothra Software Testing Environment," in *Proc. of the 2nd Workshop on Software Testing, Verification, and Analysis*, July 1988, pp. 142–151.
[14] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: an automated class mutation system: Research articles," *Softw. Test. Verif. Reliab.*, vol. 15, no. 2, pp. 97–133, 2005.
[15] F. Lonetti, E. Marchetti, and N. Pacini, "XSLT Mutation Testing," http://labse.isti.cnr.it/tools/xmut, Internal Report. April 2010.
[16] "JavaTM API for XML Processing (JAXP)," http://java.sun.com/javase/6/docs/technotes/guides/xml/jaxp/index.html, accessed April 2010.