XSLT Mutation Testing

Internal Report

April 2010

Francesca Lonetti, Eda Marchetti, Nicholas Pacini

Contents

List of Tables

Chapter 1

Mutation testing

Mutation analysis is the most common form of software fault-based testing. Faulty programs are created by seeding faults, that is, by making a small change to the program under test. The patterns for changing program text are called mutation operators, and each variant program is called a mutant. A mutant is valid if it is syntactically correct. A mutant is useful if, in addition to being valid, its behavior differs from the behavior of the original program for no more than a small subset of program test cases [1]. Defining mutation operators that produce valid and useful mutations is a nontrivial task.

Mutation testing is used to assess the effectiveness of a test set in terms of its ability to detect faults. To do this, the 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 [2].

Mutation Testing performs software testing at the unit level, the integration level, and the specification level. It has been applied to many programming languages as Fortran programs, Ada programs, C programs, Java programs, C# programs, SQL code and AspectJ programs. However, in the best of our knowledge, so far any tool is available for the automatic mutants generation of XSLT programs.

In this document, we provide six classes of mutant operators that seed different kinds of faults into the variables definition, arithmetic expressions and values assignment and for each class, we describe its mutant operators.

We implement the automatic generation of mutants of a given XSLT and we also provide facilities for detecting killed mutants. In particular, each document transformed by the original XSLT is compared with those obtained by its mutants set for discovering differences among them. If the two XML-format documents are different, the mutant is classified as killed. Note that, in our assumptions, two XML-format documents are considered equal if, in their tree representation, all the corresponding node names and values are equal and all

the corresponding attribute names and values are equal.

The remaining of the document is structured as follows: we describe a set of mutation operator classes for the XSLT language and for each class we give the corresponding operators (Chapter II). In addition, in Chapter III, we present some experimental results about mutation scores obtained by executing three test suites on the standard XSLT stylesheet, that transforms documents from the standard MARCXML format to the Dublin Core format.

Chapter 2

XSLT Mutation Operators

2.1 XSLT language

XSLT is the W3C recommendation [3][4] for a XML stylesheet language. It is a powerful language allowing transformation of a XML document into another format such as HTML, XML, or anything else. 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, looks 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 [3][4] for details.

2.2 XSLT Mutation classes

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* [5] and *MuJava* [6] 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 [2]):

- 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 sections, we give an overview of the proposed mutant classes and the corresponding operators.

2.2.1 VM - Variable Manipulation

These mutation operators modify the `select` attribute of the XSLT `variable` element, eliminating its initialization or replacing its value. They are:

- vr - variable replacement. This operator replaces the value of the `select` attribute of the XSLT `variable` element, with that of other `select` attributes contained in other XSLT `variable` elements.
- vie - variable initialization elimination. It deletes the `select` attribute of the XSLT `variable` element.

2.2.2 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. In particular, this class has the following operators:

- aor - arithmetic operator replacement. It replaces an arithmetic operator used in an expression with one belonging to $\{\div, \times, +, -, mod\}$.
- lcr - logical connector replacement. It replaces a logical connector used in an expression with one belonging to $\{and, or\}$.
- ror - relational operator replacement. It replaces a relational operator used in an expression with one belonging to $\{=, !=, <, \leq, >, \geq\}$.

2.2.3 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. They are:

- xer - XPath expression replacement. It replaces a XPath expression in the `select` attribute of the `value-of` element with a XPath expression contained into another `select` attribute of the `value-of` element.

- dsi - double slash insertion. It inserts a // at the beginning of a XPath expression.
- ad - attribute deletion. It is applied to the attributes of the XPath expressions. It deletes the @ character before the attribute name.
- dnr - dot notation replacement. In XPath expressions the {.} notation indicates the current node and the {..} notation indicates the parent node. This operator replaces the {.} with {..} and viceversa.
- nrw - node name replacement with wildcard. It replaces an expression with the wildcard special character *.
- arw - attribute name replacement with wildcard. It is applied to expressions containing an attribute, in particular to those starting with @, and it replaces the whole expression with @* that is the wildcard special character for selecting attributes.

2.2.4 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 the `choose` element. They replace the `test` attribute value of the `when` or `if` elements with other `test` attribute values contained in other `when` in the same `choose` element or in the same `if` element respectively. In the same way, they operate on the `select` attribute of the `for each` element. They are:

- cwr - choose when replacement. It is applied to the `choose` element and replaces the `test` attribute value of the `when` element with another `test` attribute value contained into another `when` in the same `choose` element.
- cod - choose otherwise deletion. It deletes the `otherwise` element of the `choose` element.
- ir - if replacement. It is applied to the `if` element and replaces the `test` attribute value with another `test` attribute value contained into another `if` element.
- fer - for each replacement. It is applied to the `for each` element and replaces the `select` attribute value with another `select` attribute value contained into another `for each` element.

2.2.5 TM - Template Manipulation

These operators are applied to `template`, `apply-templates` and `call-template` elements. They operate changing attributes values of these elements and are called:

- asr - apply-templates selection replacement. It replaces the `select` attribute value of the `apply-templates` element with the `match` attribute value of the `template` element.
- ctr - call-template name replacement. It replaces the `name` attribute value of the `call-template` element with the `match` attribute value of the `template` element.
- tnr - template name replacement. It replaces the `name` attribute value of the `template` element with another `name` attribute value of another `template` element.
- tmr - template match replacement. It replaces the `match` attribute value of the `template` element with another `match` attribute value of another `template` element.

2.2.6 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 in the other `element` and `attribute` nodes. They are:

- anr - attribute name replacement. It is applied to the `attribute` nodes. It replaces the `name` attribute value with another `name` attribute value of another `attribute` node.
- enr - element name replacement. It is applied to the `element` nodes. It replaces the `name` attribute value with another `name` attribute value of another `element` node.

Chapter 3

Experimental Results

We applied the proposed approach for evaluating the fault detection effectiveness of a given test suite. We used the set of mutants derived for the `MARC21slim2SRWDC.xsl` stylesheet [7], and we developed a three 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` [7].

After detecting the equivalent mutants, we executed the test suite on the derived mutants and we obtained the mutation score of Table 3.1.

In the second round of the experiment, we increased the test suite TS1 by adding one XML document named `collection.xml` [7]. We called the increased test suite TS2. We repeated the evaluation using TS2 obtaining the results of Table 3.2.

Finally, we increased the test suite TS2 by adding `sandburg.xml` [7]. We called the increased test suite TS3. We repeated the evaluation using TS3 obtaining the results of Table 3.3.

For detecting mutants, we considered differences between the document transformed by the original XSLT and those obtained by its mutants set. If the two XML-format documents are different, the mutant is classified as killed.

However, as a final consideration of the experiment, the number of mutants killed by the application of TS1 is quite small, 189, i.e. 25.5%. 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 TS3 is 66.8%.

Table 3.1: Mutant-kill ratios achieved by TS1

Mut_Class	Mut_Op	#Mut	#Mut_Kill	%Mut_Kill
VM	vr	10	3	30
	vie	4	1	25
Tot_VM		14	4	28.6
ALROM	lcr	7	0	0
	ror	45	0	0
Tot_ALROM		52	0	0
XPEM	xer	24	8	33.3
	dsi	5	1	20
	dnr	1	0	0
	nrw	6	1	16.7
Tot_XPEM		36	10	27.8
CIM	cwr	72	0	0
	ir	36	8	22.2
	fer	529	166	31.4
Tot_CIM		637	174	27.3
TM	asr	1	1	100
Tot_TM		1	1	100
Tot		740	189	25.5

Table 3.2: Mutant-kill ratios achieved by TS2

Mut_Class	Mut_Op	#Mut	#Mut_Kill	%Mut_Kill
VM	vr	10	7	70
	vie	4	3	75
Tot_VM		14	10	71.4
ALROM	lcr	7	2	28.6
	ror	45	16	35.6
Tot_ALROM		52	18	34.6
XPEM	xer	24	14	58.3
	dsi	5	2	40
	dnr	1	1	100
	nrw	6	2	33.3
Tot_XPEM		36	19	52.8
CIM	cwr	72	25	34.7
	ir	36	10	27.8
	fer	529	360	68.1
Tot_CIM		637	395	62
TM	asr	1	1	100
Tot_TM		1	1	100
Tot		740	443	59.9

Table 3.3: Mutant-kill ratios achieved by TS3

Mut_Class	Mut_Op	#Mut	#Mut_Kill	%Mut_Kill
VM	vr	10	7	70
	vie	4	3	75
Tot_VM		14	10	71.4
ALROM	lcr	7	3	42.9
	ror	45	20	44.4
Tot_ALROM		52	23	44.2
XPEM	xer	24	16	66.7
	dsi	5	3	60
	dnr	1	1	100
	nrw	6	2	33.3
Tot_XPEM		36	22	61.1
CIM	cwr	72	31	43.1
	ir	36	10	27.8
	fer	529	397	75
Tot_CIM		637	438	68.8
TM	asr	1	1	100
Tot_TM		1	1	100
Tot		740	494	66.8

Bibliography

[1] *Software Testing and Analysis: Process, Principles and Techniques*.

[2] 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.

[3] W3C Recommendation, "XSL Transformations (XSLT) Version 1.0 November 1999." http://www.w3.org/TR/xslt, accessed April 2010.

[4] W3C Recommendation, "XSL Transformations (XSLT) Version 2.0 January 2007." http://www.w3.org/TR/xslt20/, accessed April 2010.

[5] 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*, pp. 142–151, July 1988.

[6] 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.

[7] MARCXML. http://www.loc.gov/standards/marcxml/, accessed April 2010.