

# Evolving XSLT Stylesheets For Document Transformation<sup>\*</sup>

P. Garcia-Sanchez, J.J. Merelo, J.L.J. Laredo, A.M. Mora, and P.A. Castillo

GeNeura Team, Dept. ATC, Universidad de Granada, Spain  
{pgarcia, jmerelo, juanlu, amorag, pedro}@geneura.ugr.es

**Abstract.** This paper presents a new version of an evolutionary algorithm that creates XSLT programs from its intended input and output. XSLT is a general purpose, document-oriented functional language, generally used to transform XML documents (or, in general, solve any problem that can be coded as an XML document). Previously, a solution that solved the problem efficiently was proposed. In this paper, we improve on those results by testing different fitness functions, adding a new operator and changing the type of desired output document that can be obtained. The experiments show that the best results are obtained without considering the XSLT length and including this new operator.

## 1 Introduction

Since the Information Technology industry has settled on different Extensible Markup Language (XML) dialects as information exchange format, there is a business need for programs that transform from one XML set of tags to another, extracting information or combining it in many possible ways; a typical example of this transformation could be the extraction of news headlines from an on-line newspaper that uses XHTML.

XSLT stylesheets (XML Stylesheet Language for Transformations) [1], also called *logicsheets*, are programs designed for this purpose: applied to an XML document, they produce another. There are other possible solutions: programs written in any language that work with text as input and output (using, for instance, regular expressions) or SAX filters [2], that process each tag in a XML document in a different way, and do not need to load into memory the whole XML document. However, these solutions require programming in external languages, while XSLT is a part of the XML set of standards; in fact, XSLT logicsheets are XML documents. This is why XSLT is one of the most common ways of specifying document transformations. XSLT make use of XPath expressions [3] to select nodes from the source document.

The work needed for logicsheet creation scales quadratically with the number of input and output formats: for  $n$  input and  $m$  output formats,  $n \times m$  transformations will be needed. Considering that each conversion is a hand-written program and the initial and final formats can vary with certain frequency, any

---

<sup>\*</sup> Supported by projects TIN2007-68083-C02-01, P06-TIC-02025 and OTRI-1515.

automation of the process means a considerable saving of effort on the part of the programmers<sup>1</sup>.

So, the problem is to find the XSLT logicsheet that, from one input XML document, is able to obtain an output XML document which contains exclusively the information desired from the first one. This information may be sorted in any possible way (possibly in an order different to the input document). In this work, an Evolutionary Algorithm (EA) [4] to resolve this problem is presented. The logicsheet will be evolved using evolutionary operators that will take into account the structure of the program and its components.

Thus, XSLT provides a general mechanism for the association of patterns in the source XML document to the application of format rules to these elements, but in order to simplify the search space for the evolutionary algorithm, only three instructions will be considered in this paper: `template`, which selects the XML fragments that will be included when the element in its `match` attribute is found; `apply-templates`, which is used to select the elements to which the transformation is going to be applied and delegate control to the corresponding `templates`; and `copy-of`, which includes the text representation of the nodes of the input XML into the output file (that is, copy all node contents and tags), so the output file will be a complete XML instead a list of content, as we did in our previous work. Of course, XPath expressions will also be used to select particular elements and sets of them.

In a previous work [5], we published an initial set of XSLT evolution experiments, testing different document structures and operators. In this paper we will try to improve on those results, by using XML output documents with tree structure, instead of plain text-only documents. This means that output documents are composed of several nodes, which makes it easier to compare them with each other. So, the output XML will be a complete XML document with a (possibly sorted in a different way) list of nodes present in the original document.

The rest of the paper is structured as follows: the state of the art is presented in Section 2. Section 3 describes the solution presented in this work, with the novel elements introduced. Experiments with the automatic generation of XSLT stylesheets for different examples are described in Section 4, and finally the conclusions and possible lines of future work are presented in Section 5.

## 2 State of the art

To our knowledge, there are few works related to the application of genetic programming techniques to the automatic generation of XSLT logicsheets; one of them, by Scott Martens [6], presents a technique to find XSLT stylesheets that transform a XML file into HTML by using genetic programming. Martens works on simple XML documents and uses the UNIX diff function as the basis for its fitness function. He concludes that genetic programming is useful to obtain solutions to simple examples of the problem, but it needs unreasonable execution

---

<sup>1</sup> And money by whoever hires them

times for complex examples and might not be a suitable method to solve this kind of problems.

Schmidt and Waltermann [7] approached the problem taking into account that XSLT is a functional language, and using functional language program generation techniques on it, in what they call *inductive synthesis*. First they create a non-recursive program, and then, by identifying recurrent parts, convert it into a recursive program; this is a generalization of the technique used to generate programs in other programming languages such as LISP [8], and used thoroughly since the eighties [9].

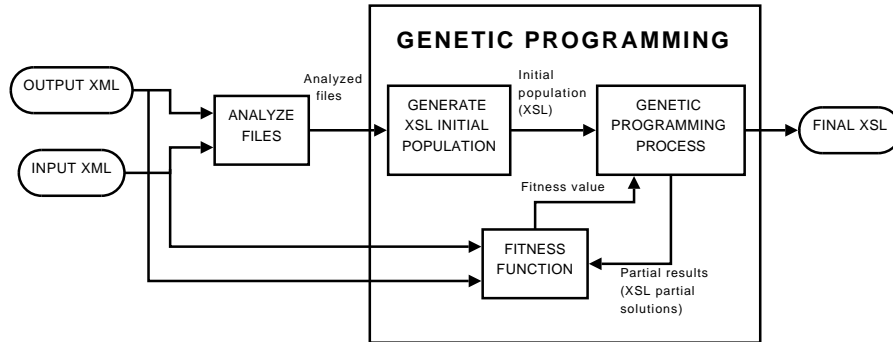
A few other authors have approached the general problem of generating XML document transformations knowing the original and target structure of the documents, as represented by its DTD (Document Type Definition): Leinonen et al. [10, 11] have proposed semi-automatic generation of transformations for XML documents, but user input is needed to define the label association. There are also freeware programs that perform transformations on documents from a XSchema to another one. However, they must know both XSchemata in advance, and are not able to accomplish general transformations on well formed XML documents from examples.

In our previous work [5], we presented an evolutionary algorithm to obtain a XSLT that extracts information represented in a output XML from an input XML. Several XSLT structures and operators were presented and studied. The main inconvenient of that work is the output XML file is a list of text elements instead of XML nodes, which would be much more useful to perform real XML transformations. Additionally, the existing operators apparently led to situations where evolutionary changes were quite difficult, so a new operator is proposed. In this paper, the desired XML output document includes a set of nodes (text and tags) extracted from the input document, which can be additionally processed to change the required output tags.

### 3 Methodology

The EA described here evolves XSLT stylesheets, which are generated using a set of operators and evaluated using a fitness function that is related to the difference between generated XML and output XML associated to the example. The way the algorithm works is shown in Figure 1. The solution has been programmed using JEO [12], an evolutionary algorithm library developed at University of Granada as part of the DREAM project [13], which is available from <http://www.dr-ea-m.org>.

Since the search space of possible stylesheets is exceedingly large, language grammar must be considered in order to restrict it and avoid syntactically wrong stylesheet generation. Due to this, transformations are applied to a predetermined stylesheet structure which was selected among three different ones in previous work [5]. An example of this structure is shown in Figure 2. This type of structure is more constrained than other types; and search is thus easier, since



**Fig. 1.** This figure shows how the algorithm works. Each individual of the population is an XSLT stylesheet whose fitness is computed comparing the XML generated by the stylesheet (using the input XML) with the output XML.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output indent="no" method="xml" />
  <xsl:template match="/">
    <grammar>
      <xsl:apply-templates select="/grammar"/>
    </grammar>
  </xsl:template>
  <xsl:template match="/grammar">
    <xsl:copy-of select="div[3]" />
    <xsl:copy-of select="div[1]/h3[5]" />
    <xsl:copy-of select="h1" />
  </xsl:template>
</xsl:stylesheet>
```

**Fig. 2.** Example of a final XSLT generated by the algorithm. This logic sheet, applied to the input XML document, produces an XML document equals to the desired XML output document.

less stylesheets are generated. Despite the constraints, mutation and crossover are much more disruptive, generating a rougher landscape than before.

The operators may be classified in two different types: the first one consists in operators that modify XPath routes in the attributes of the XSLT instructions (`apply-template` and `copy-of`); and the other are the operators used to modify the XSLT tree structure. In order to ensure the existence of the elements (tags) added to the XPath expressions and XSLT instruction attributes, every time one of them is needed it is randomly selected from the input file. These operators have been described in more detail in our previous work [5], so we refer the interested reader to that paper. A complete list is shown in Table 1 whose names are quite descriptive. When an operator of the second group of the table is selected to modify an individual, another operator is selected randomly from the first group to use both in conjunction.

However, these operators are not enough to perform a smooth search; sometimes the XSLT search converges into a bad solutions when we want to select ordered but alternated items from a node. So it is necessary to add a new operator to increase the diversification of this solution. The new operator proposed, `XSLTreeMutatorSplitTemplate`, expands a random `copy-of` node into a list of complete `copy-of` with all cardinalities (as shown in Figure 3). The result of

```

<xsl:template match="book">
  <xsl:copy-of select="chapter"/>
</xsl:template>

```

```

<xsl:template match="book">
  <xsl:copy-of select="chapter[1]"/>
  <xsl:copy-of select="chapter[2]"/>
  <xsl:copy-of select="chapter[3]"/>
  <xsl:copy-of select="chapter[4]"/>
</xsl:template>

```

**Fig. 3.** The left template is transformed into right template applying the split mutator. The number of `chapter`s in the input XML were 4 (this is known by the algorithm when it process the input XML at the beginning of execution).

applying the new XSLT and the previous is the same, but it is easier for genetic operators to modify the list of `copy-of` than the generic one (modifying, adding or removing XPath and/or tags).

Since in this paper output is a fully formed XML document, fitness has been changed to be the XML difference between the desired and the obtained output, that is, the difference in nodes between the desired T and the actual document X. This difference breaks down in insertions (nodes in X but not in T) and deletions (nodes in T but not in X). We will leverage this vectorial structure of fitness so that evolution can profit from it: instead of using a single aggregative function, as we did in previous papers [5], fitness is now a vector that includes the number of node deletions and additions needed to obtain the target output from the obtained output, and the resulting XSLT stylesheet length. The XSLT stylesheet is correct only if the number of deletions and additions is 0;

and minimizing length helps removing useless statements from it. So, fitness is minimized by comparing individuals as follows: An individual is considered better than another

- if the number of deletions is smaller,
- if the number of additions is smaller, being the number of deletions the same,
- if the length is smaller, being the number of deletions/additions the same.

Separating and prioritizing the number of deletions helps guide evolution, by trying to find first a stylesheet that includes all elements in the target document, then eliminating unneeded elements, while, at the same time, reducing length. However, this last element introduces selective pressure towards small stylesheets, which might hinder discovering the correct one, so we have also tested in this paper whether we should consider length or not as a part of the fitness.

## 4 Experiments and results

To test the algorithm we have performed several experiments with 7 different XML input and output files. The algorithm has been executed 30 times for each input XML. Every experiment took 200.76 seconds in average to finish. The same input file was used for several experiments: a RSS feed from a weblog (<http://geneura.wordpress.com>) and an XHTML file. All input and output files and programs used in this experiment are available from our Subversion repository: <http://tinyurl.com/6nxv8c>.

The computer used to perform the experiments is a Centrino Core Duo at 1.83 GHz, 2 GB RAM, and the Java Runtime Environment 1.6.0.01. The population size was 128 individuals for all runs, generated using the input XML as information source. The termination criteria was set to 300 generations or until a solution was found, and selection was performed via a 5-Tournament; 30 experiments were run, with different random seeds, for each input document. The XML and XSLT processors were the default ones included in the JRE standard library. The operator rates used in the experiments, which were tuned heuristically, are shown in Table 1. The crossover and mutation probability have been set to 0.25 and 0.5, after several experimental runs.

Due to the use of the new mutation operator we have performed the experiments using 3 different configurations. The first is the algorithm without the new operator (`XSLTreeMutatorSplitTemplate`), the second one, using the operator and the third without considering the length of the stylesheet in the fitness function. This helps to keep the solutions which have the same number of deletions and insertions but larger size caused by the use of the operator (expanding the copy-of tags). The breakdown of results per input file is shown in Table 2.

The fitness function, in general, yielded better results than previously. The algorithm was able to find an adequate XSLT stylesheet within the pre-assigned number of generations in most cases.

**Table 1.** Operator priorities (used for the roulette wheel that randomly selects the operator to apply) used in the experiments.

Operator	Priority
XSLTTreeMutatorXPathSetSelf	0.1
XSLTTreeMutatorXPathRemoveBranch	0.17
XSLTTreeMutatorXPathAddFilter	0.18
XSLTTreeMutatorXPathMutateFilter	0.18
XSLTTreeMutatorXPathRemoveFilter	0.2
XSLTTreeMutatorXPathAddBranch	0.16
XSLTTreeMutatorAddTemplate	0.2
XSLTTreeMutatorMutateTemplate	0.10
XSLTTreeMutatorRemoveTemplate	0.12
XSLTTreeAddApply	0.1
XSLTTreeMutateApply1	0.1
XSLTTreeMutateApply2	0.14
XSLTTreeRemoveApply	0.1
XSLTreeMutatorSplitTemplate	0.05
Probability of crossover	0.25
Probability of mutation	0.5

Examples 1, 2, 3 and 6 are complete and ordered lists of elements of one or several nodes, whose solution is simple, since the algorithm can easily create a logicsheet that extracts all the childrens of a specific node. Example 7 takes specific and repeated elements from distinct nodes, which makes it more difficult, because different expressions are needed to extract each one of them and the generated logicsheet is more complex. Finally, examples 4 and 5 focus into portions of ordered and unordered fragments of an XML section (namely the 3rd and 6th chapters of a book), so the population converges into solutions with all the elements of a node (selecting all chapters of a book) due to the way the fitness works. Obtaining solutions for these examples is quite difficult without using the new operator (just two times for example 5 and none for 4), which is fixed when we use it; instead of selecting all chapters of a book (`book/chapter`), it selects all chapters using XPath location (`book/chapter[1]`, `book/chapter[2]...`), so it is easier for the algorithm to evolve this partial solution into a better one.

On the other hand, overruling the XSLT length comparison in fitness gives different solutions the same chances to evolve, so the algorithm maintains more diversity and finds solutions in less time than the cases comparing that length (see Table 3). However, the generated XSLT may contain useless statements, that could produce incorrect XMLs in a a production environment.

When a solution was found, the number of generations and time used to find it also varies, as shown in Table 3. In general, the exploration/exploitation balance seems to be biased towards exploration. Being such a vast and rough search space makes that, after a few initial generations that create stylesheets with a small difference from the target, mutations are the main operator at work.

**Table 2.** Number of times, out of 30 experiments, a solution is found within the predefined number of generations without the split mutator, using the split mutator with normal fitness, and using split mutator without considering of the length of the generated stylesheet.

Input file	Without Split	With Split	With Split w/o length
1	26	25	25
2	30	30	30
3	30	30	30
4	0	24	24
5	2	30	29
6	30	30	30
7	10	9	13

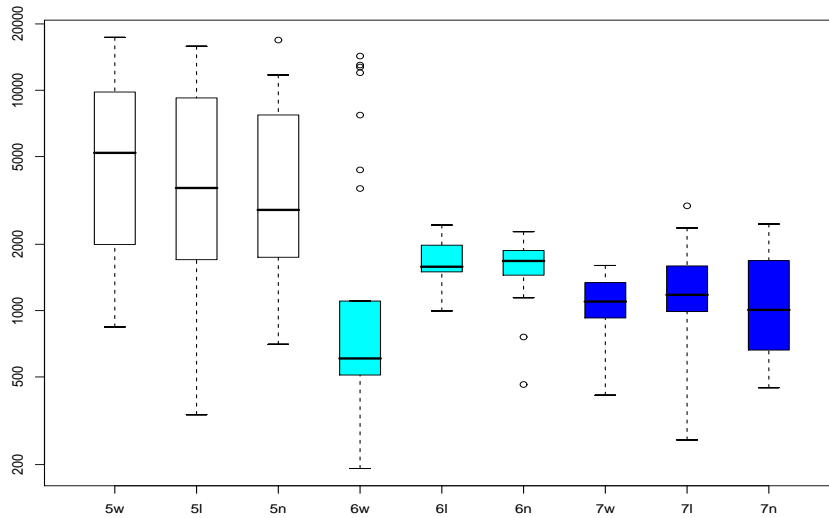
**Table 3.** Average generations/standard deviation to find an optimal solution (in less of 300 generations), without the split mutator, using the split mutator with normal fitness, and using split mutator without considering of the length of the generated stylesheet.

Input file	W/o Split	With Split	With Split w/o length
1	62.86 ± 102.03	83.33 ± 112.81	66.33 ± 106.85
2	1.5 ± 1.25	1.6 ± 1.30	1.1 ± 1.29
3	4.13 ± 2.06	3.13 ± 1.94	3.83 ± 2.90
4	-	81.03 ± 112.25	71.68 ± 107.24
5	289.9 ± 45.09	26.83 ± 5.35	36.03 ± 50.19
6	15.43 ± 5.74	20.43 ± 9.88	19.0 ± 11.12
7	232.17 ± 105.48	245.46 ± 96.92	214.0 ± 112.90

## 5 Conclusions, discussion, and future work

In this paper we present the results of an evolutionary algorithm designed to search the XSLT logicsheets that is able to make a particular transformation from an input XML document into a desired output one; one of the advantages of this application is that resulting logicsheets can be used directly in a production environment, without the interaction of a human operator. It tackles a real-world problem found in many organizations and it is open source software, available from <http://tinyurl.com/5lwjcn>.

The experiments have shown that the search space is particularly rough, with mutations in general leading to huge changes in fitness. The hierarchical fitness used is probably the cause of having a big loss of diversity at the beginning of the evolutionary search, leading to the need of a higher level of explorations later during the algorithm run. This problem will have to be approached via explicit diversity-preservation mechanisms, or by using a multiobjective evolutionary algorithm, instead of the one used now. A deeper understanding of how different operator rates affect the result will also help; for the time being, operator rate tuning has been very shallow, and geared towards obtaining the result. In addition, results showed in this paper can be used as a baseline for future ver-



**Fig. 4.** Logarithmic boxplot of the number of evaluations to find the best individual in examples #5, #6 and #7 without split mutator (w), with split mutator considering length (l) and without this feature (n).

sions of the algorithm, or other algorithms for the same problem. At any rate, unlike what was mentioned in the pioneering paper [6], solutions can be found effectively and efficiently.

However, there are some questions and issues that will have to be addressed in future papers:

- Using the DTD (associated to a XML file) as a source of information for conversions between XML documents and for restrictions of the possible variations.
- Adding different labels in the XSLT to allow the building of different kinds of documents such as HTML or WML.
- Testing evolution with other kind of tools, such as a chain of SAX filters.
- Obviously, testing different kinds and increasingly complex set of documents, and using several input and desired output documents at the same time, to test the generalization capability of the procedure.
- Using the identity transform [14] as another frame for evolution, as an alternative to the structure shown here. The identity transform puts every element found in the input document in the output document; elements can then be selectively eliminated via the addition of single statements.
- Tackle difficult problems from the point of view of a human operator. In general, the XSLT stylesheets found here could have been programmed by

a knowledgeable person in around an hour, but in some cases, input/output mapping would not be so obvious at first sight. This will mean, in general, increase also the XSLT statements used in the stylesheet, and also in general, adding new types of operators.

## References

1. Clark, J.: XSL transformations (XSLT), version 1.0, W3C recommendation 16 november 1999. Available from <http://www.w3.org/TR/xslt.html> (1999)
2. Wikipedia: Simple API for XML — Wikipedia, the free encyclopedia (2007) [Online; accessed 21-March-2007].
3. Clark, J., DeRose, S., et al.: XML Path Language (XPath) Version 1.0. W3C Recommendation **16** (1999) 1999
4. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, Boston, Massachusetts, EE. UU. (1989)
5. Garcia-Sanchez, P., Laredo, J.L.J., Sevilla, J.P., Castillo, P., Merelo, J.J.: Improved evolutionary generation of XSLT stylesheets. ArXiv database: <http://arxiv.org/abs/0803.1926> (1999)
6. Martens, S.: Automatic creation of XML document conversion scripts by genetic programming. In: Genetic Algorithms and Genetic Programming at Stanford. (2000) 269 ff.
7. Schmid, U., Waltermann, J.: Automatic synthesis of XSL-transformations from example documents. In Hamza, M., ed.: IASTED International Conference on Artificial Intelligence and Applications. (2004) 252–257
8. Biermann, A.: The inference of regular LISP programs from examples. IEEE Transactions on Systems, Man and Cybernetics **8**(8) (1978) 585–600
9. Biermann, A.W., Guiho, G., eds.: Computer Program Synthesis Methodologies. Reidel, Dordrecht (1983)
10. Leinonen, P.: Automating XML document structure transformations. In: Proceedings of the 2003 ACM Symposium on Document Engineering. (2003) 26–28
11. Kuikka, E., Leinonen, P., Penttonen, M.: Towards automating of document structure transformations. In: Proceedings of the 2002 ACM Symposium on Document Engineering. (2002) 103–110
12. Arenas, M.G., Dolin, B., Merelo-Guervós, J.J., Castillo, P.A., de Viana, I.F., Schoenauer, M.: JEO: Java Evolving Objects. In: Proceedings of the Genetic and Evolutionary Computation Conference. (2002) 991
13. Arenas, M., Collet, P., Eiben, A., Jelasity, M., Merelo, J.J., Paechter, B., Preuß, M., Schoenauer, M.: A framework for distributed evolutionary algorithms. In: Proceedings of the Seventh Conference on Parallel Problem Solving from Nature, volume 2439 of Lecture Notes in Computer Science. (2002) 665–675
14. Wikipedia: Identity transform — Wikipedia, The Free Encyclopedia (2007) [Online; accessed 24-January-2008]: [http://en.wikipedia.org/wiki/Identity\\_transform](http://en.wikipedia.org/wiki/Identity_transform).