CodeX: An Approach for Debugging XSLT Transformations

Eric Bae and James Bailey
Department of Computer Science and Software Engineering
University of Melbourne
{kheb,jbailey}@cs.mu.oz.au

Abstract

XML is now a dominant standard for storing and exchanging information. One very important activity is the transformation of XML documents into other formats, via the transformation language XSLT. XSLT provides a powerful way to perform document conversion and exchange, avoiding reliance on application specific solutions. However, XSLT is a complex language and the current level of support for debugging tools is poor. Many tools that do exist are mainly an extension of conventional techniques for imperative programs and not well-suited to the task. In this paper, we present CodeX, a debugger for XSLT and propose three debugging techniques which are particularly suited to the language We aim to offer XSLT users a tool which is beneficial in finding errors, as well as facilitating a better understanding of the XML transformation process.

Introduction: XML is now a dominant standard for storing and exchanging information. To support its use, a number of accompanying standards exist, a primary one being XSLT (Extensible Stylesheet Language Transformations).

Specifically designed for XML transformations, XSLT is a high-level, declarative programming language offering programmers a range of built-in features. However, due to its complexity, both seasoned programmers and beginners alike find the language difficult to learn. Moreover, being a young language, there are not yet many techniques and tools for writing error-free programs[1], compared to general purpose programming languages, such as C and Java.

In this paper, we discuss the principles behind CodeX, a debugger for XSLT. The development of this tool is motivated by the high level of complexity involved when creating XSLT stylesheets and the lack of research on methods for debugging XSLT.

Our main contribution is the identification of three core techniques important for XSLT debugging: i) visual step tracing, ii) backwards debugging and iii) template slicing. The aim is to help programmers build stylesheets with fewer errors and also facilitate a better understanding of the XML transformation process.

Background: XSLT is a language with two main applications. The first of these deals with the presentation of XML data, which involves displaying XML documents in a sensible layout. The stylesheet stores a set of rules that can be used for determining how the content of elements in an XML document should be displayed in a browser.

The second main application of XSLT is for data exchange and conversion of an XML document into another XML document having a different structure. For example, in the publishing industry, it is common to need multiple versions of the same document. Traditionally, a document was duplicated into all the required media, such as a newspaper article, Web page and magazine. However, by applying transformation rules in XSLT stylesheets, a single XML document containing the contents of the material can be straightforwardly transformed into a variety of formats.

A typical XSLT stylesheet consists of a collection of template rules. Each rule can be applied by pattern matching the condition of the rule against the input XML document and then possibly generating some output. These template rules may be applied in any order. Furthermore, XSLT programs disallow variable assignments, thus eliminating side-effects. Characteristics such as lazy evaluation and purity make XSLT similar to functional languages. Indeed our proposed debugging techniques are modelled on some of the methods used for these languages.

Debugging Techniques: Among the many general program debugging methods, the following ones stand out:
Step-tracing is a control mechanism acting on the syntactic order of execution of the program. A programmer executes each statement of the program and evaluates the effects of that particular statement in regard to the overall program, looking for irregular patterns in the expected behaviour.

[1] XSLT applications or programs are commonly called *XSLT stylesheets*. We use this terminology to refer to XSLT programs throughout this paper.

Breakpoints are annotations inserted into programs that interrupt execution. This allows users to halt the program at desired points and check the state.
Call-stack provides a history of the actions performed by the user. It allows the user to check the sequences of actions taken to achieve the current state of the program.
Variable watching involves displaying the changes in variable values during program execution.

A step-tracing method is more suited to imperative languages, where the sequence of program execution is explicit in the syntactic structure of the program and this structure also closely emulates the user's understanding of how the application works. However, the execution in functional languages is not necessarily related to the order of code written and this restriction also applies to XSLT.

Breakpoints are only effective if users are familiar with the program's execution behaviour. While they can help users to analyse the program's state, by allowing pauses during execution, they are often inadequate for isolating specific faults, especially in a large and unfamiliar program.

The purity of XSLT disallows variable assignments, making variable watching meaningless. Call-stack functionality acts as historical data on actions taken by the user and requires further debugging tasks such as backward traversal and a manual static analysis to identify defects.

Debugging techniques for XSLT: An ad-hoc method of debugging XSLT is to transform an XML document and check its output for errors. This might suit a small and simple XSLT transformation, but is difficult for large and complex stylesheets.

Many of today's available XSLT debuggers, however, do not introduce any innovative features to correct stylesheets. They principally implement the traditional methods discussed in the previous section. In CodeX, we aim to identify techniques that are particularly suitable for XSLT. The first of the debugging techniques is a visual step-tracing feature that enhances the user's experience when analysing the program. We extend the traditional step-tracing method in CodeX and this technique dynamically displays how each step *builds* the overall output.

The second technique executes the debugging in reverse order. It has been researched extensively for other languages, but is almost unknown in XSLT. We identify it as being a better replacement for breakpoints.

Lastly, we define a new debugging technique for XSLT called *template slicing*, which is similar to a popular debugging and testing technique called *program slicing* in software engineering. This provides users with the option to choose a set of templates to debug, without having to consider the entire stylesheet.

Visualisation: Visualising the execution of a program, is a result-oriented technique, moving programmer attention away from internal computations and allowing them to focus directly on the output. It has achieved great successes in the field of debugging.

In CodeX, we implement a visual step-tracing technique which displays, at each step, the result of transformation from the beginning of the stylesheet, up to the current statement being stepped over. i.e. Each step consists of the application of all statements in the stylesheet that precede the current one. In contrast to traditional step-tracing, where only the result of a statement at each step is shown, CodeX incrementally *builds* the final output at each step, with respect to the evaluation order of the stylesheet.

An important implementation aspect is correct handling of recursive template applications. If a given template needs to be applied to two nodes in the document, then we display the output produced as each node is processed, rather than the output after both nodes have been processed. This requires dynamic manipulation of subsets of the input XML.

The benefits are clear. Since the programmer can visually experience the stages of transformation, the response to any irregular behaviour can be immediate. Moreover, in traditional text-based step-tracing, the programmer may frequently overlook errors. Consistent visual feedback therefore provides a means to examine the program's behaviour and keep the user's interest in finding errors. In general, the technique allows users to better learn the overall transformation process of their stylesheet. It reduces the effort required to manually compute the behaviour of the program and allows focus to be more on suitable corrective action.

In addition, since many XSLT transformations yield HTML output, we provide an internal browser to directly show how HTML transformation is *built*. We aim to involve the users with the output directly, without the need to analyse the resultant HTML source code. Thus creating a result-oriented debugging approach.

Backwards Debugging: The manifestation point of a bug, which is the first time a user detects the defect, may actually be several statements away from the source of the problem. In a simple program, static analysis might be able to identify the defect. However, the task becomes extremely complex, if not infeasible, in a large program.

The common method used here involves breakpoints. Inserting breakpoints allows users to interrupt the execution to check the state of the program. However, this technique is effective only when programmers are well-aware of the program's behaviour and are able to think ahead to position the breakpoints at appropriate locations.

A logical solution to overcome the above situation is to think backwards; deduce the conditions under which the program produces the incorrect output. This technique, known as *backward debugging*, has been implemented for a number of languages. Backward debugging has significant advantages over breakpoints. Firstly, the debugging cost is reduced, since the nature of debugging is linear. Moreover, in the previous method, if the programmer overlooks the actual location of the bug, a new breakpoint and a re-execution are required. Backward debugging eliminates these extra efforts by permitting direct transitions between a forward and backward stepping.

The technique, however, is difficult to implement in a compiled language, since the debugger must note all memory accesses and save states prior to any changes. All instructions must also be decoded for their reversible state and even so, there are instructions that are irreversible such as I/O operations or calls to library routines. In functional languages, however, backwards debugging can be better adopted than in procedural languages and there are tools which prove this argument [2, 7]. Given XSLT's functional nature, backwards debugging is then a natural idea.

We implement backwards debugging in CodeX by storing the output history generated from visual step tracing. e.g. Suppose the user has performed several step tracing activities. If they now wish to apply backwards debugging, then the current statement is marked to later resume visual step tracing. The user is then free to move back and forth in the code and CodeX will display the correct output (derived from the history) relevant to the current statement being examined. Once the user reaches the marked statement, visual step tracing then resumes as before.

Program Slicing: Weiser [8] describes an instinctive mental model which programmers employ when debugging unfamiliar programs. It involves breaking large program code into many coherent pieces, where statements in each piece are related by their flow of data. The algorithm developed in his paper is a technique called *program slicing*, which finds all statements in a program that directly or indirectly affect the value of a given variable occurrence in a given point of the program. These pieces are called *slices* of the program. The algorithm generates these slices as different executable output. The goal is to execute these slices and achieve the exactly same impact on the variable, compared to executing the entire program.

There are various types of slicing. Backward slicing consists of all statements that *affect* a given point of the program, whereas forward slicing deals with statements that *are affected* by a given point of the program. Dynamic slicing uses both program input and particular test cases.

The benefits of this technique are significant. In a typical program, there are only a certain set of statements that may influence the value of a variable. This makes the rest of the code irrelevant to the final value of the variable. Observing only the influential statements through this syntactic simplification technique reduces the debugging time. Additionally program slicing resembles the mental model of debugging in a programmer's mind and can support them in correctly identifying the origin of the bug. The technique can also enhance the backwards debugging method described in the previous section. Since users frequently think backwards during fault location, applying the backward debugging in each slice will permit users to avoid stepping through unrelated statements as well as keeping them to focus on a particular problem at a time.

XSLT Slicing: The original definition of program slicing proposed in [8] was primarily for imperative languages and based on two assumptions: i) Programs contain variables and statements and ii) Slices consists solely of statements.

Even though the proposal was still targeted to imperative languages, it raises the possibility of utilising in the area of functional languages. Functional languages are, as mentioned earlier, based on the expressions and definitions where the order of evaluation is driven by dependencies between the functions. For this reason, slicing with respect to the program source is less suitable for debugging XSLT. Penney [7] suggests that the natural way to utilise program slicing in functional languages is to slice with respect to a particular sub-expression. In XSLT, we consider a sub-expression as each of the template rules available. Therefore, we adopt *template slicing* in CodeX, where the stylesheet is sliced with the respect to the template rules.

Through template slicing, users have the option to debug only the necessary template rules individually and check whether their transformations are performed correctly. This is particularly useful in the context of software maintenance, where extra functionality may be added. In such situations, the programmer does not want to step through the entire stylesheet, being presented with template rules that have previously been verified through earlier debugging sessions. Template slicing serves as a tool which users can employ to only observe the chosen template rules by dividing the stylesheet into its coherent slices.

Visual step tracing is implemented in Codex by providing users with a window displaying each template. They may then simply select a number of templates to construct the slice. Once a slice has been chosen, visual step tracing and backwards debugging techniques may then be applied.

Implementation: Our debugger, CodeX incorporates the three methods we have discussed. It is written in Java and uses Xerces 2 for parsing XML and Xalan 2 for processing XSLT. It is a prototype implementation intended to demonstrate the potential of the techniques and does not currently debug the full XSLT language. The main restrictions are i) It does not handle stylesheets with conditional and iteration statements, instead requiring the order of evaluation to be purely defined by template reference elements. ii) Importation of external stylesheets is disallowed and so only one stylesheet at a time can be debugged.

Related Work: At the time of writing, we know of no published research undertaken on XSLT debugging. Instead, work in this area seems to be solely represented by commercial product implementations. This lag in academic research is perhaps explained by the pressure of market demands and the rapidly increasing uptake of XSLT.

However, debugging techniques implemented in CodeX, namely *software visualisation*, *backward debugging* and *program slicing* are of course found in other areas. Visual techniques for debugging have a long history, going back to the the use of flowcharts. See [4, 3] for recent examples.

With regard to backward debugging, one well-known system was by Lieberman and Fry [5], called ZStep 95 which, via visualisation, provided the reversible capability to debug backwards. A debugger built by Penney [7] called FIT (Functional Interactive Tracer) introduced *augmented trace-based* debugging for use in lazy functional languages.

The term *program slicing* was first introduced by Weiser [8], who proposed to break programs into several *slices* which contains statements related by data flow. *Dynamic slicing* was introduced in [1]. A little research has been done in applying program slicing in functional languages [6, 7].

Commercial XSLT Debuggers: There has been a proliferation of tools for XSLT in recent times, reflecting the growing popularity of the language. The majority of these are developed by major companies and include the XSLT debugging feature within a much bigger tool, namely an IDE(Integrated Development Environment), that promises complete development solutions for not only XSLT, but XML and other scripting languages. These tools typically have highly usable interfaces, but the main focus is in the development environment for various languages, rather than specifically debugging XSLT.

The accompanying table considers the functionality of currently available debuggers. We see that CodeX is the only debugger that allows template slicing and full visual step tracing (3 others provide a partial visual step tracing capability that displays the output generated so far in markup form, but not interpreted via a browser). It is also interesting to see that nearly all tools also make breakpoints available, though, as we have argued, they are not particularly appropriate for XSLT.

Tools	step trace	call stack	break points	visual step trace	back ward debug-ging	temp-late slic-ing
CodeX	x	x		x	x	x
Komodo 2.3	x	x	x	part		
Stylus Studio 3.0	x	x	x	part		
XML Origin 1.0	x	x	x			
Xselerator 2.5	x	x	x	part	x	
xsldbg	x		x			
XSLTracer	x		x			

Summary and Future Work: This is the first paper we are aware of to examine debugging XSLT. We have discussed the principles of three main techniques we have implemented for debugging XSLT - visual step tracing, backward debugging and template slicing. None of the commercial debuggers currently implements all these techniques. In the future, we plan to extend CodeX by adding support for more XSLT syntax and also improve its efficiency.

References

[1] Agrawal, DeMillo, and Spafford. Debugging with dynamic slicing and backtracking. *Software Practice and Experience*, 23(6), 1993.

[2] S. P. Booth and S. B. Jones. Walk backwards to happiness - debugging by time travel. In *Proceedings of the Third International Workshop on Automated and Algorithmic Debugginga*, pages 171–183, 1997.

[3] J. Eagan, M. J. Harrold, J. A. Jones, and John T. Stasko. Technical note: Visually encoding program test information to find faults in software. In *IEEE Symposium on Information Visualization*, pages 33–36, 2001.

[4] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program execution. In *Proceedings of the 19th International Conference on Software Engineering*, pages 360–370, 1997.

[5] H. Lieberman and C. Fry. Zstep 95: A reversible, animated source code stepper. In *Software Visualization, Programming as a Multimedia Experience*. The MIT Press, 1997.

[6] H. Nilsson and P. Fritzson. Algorithmic debugging for lazy functional languages. In *Journal of Functional Programming*, pages 337–370, 1994.

[7] A. Penney. *Augmenting Trace-based Functional Debugging*. PhD thesis, University of Bristol, 1999.

[8] M. Weiser. Programmers use slices when debugging. In *Communications of ACM*, volume 25, pages 446–452, 1982.