Home-grown CASE tools with XML and XSLT

Rick van Rein   Richard Brinkman
University of Twente, the Netherlands
{vanrein,brinkman}@cs.utwente.nl

March 13, 2000

Abstract

This paper demonstrates an approach to software generation where XML representations of models are transformed to implementations by XSLT stylesheets. Although XSLT was not primarily intended for this use, it serves quite well. There are only few problems in this approach, and we identify these based on our examples.

Keywords: Software generation, CASE tools, XML, XSLT.

Introduction

With the increasing popularity of the Internet, the HTML language has grown out of proportions, and the XML language is intended to structure its content better. Aside from its implications on the web, XML also receives considerate attention from software developers as a standard representation for data storage, which is mainly due to the intention of XML to model *content* rather than *layout* as is custom practice with HTML. To map this content to a layout, the XSLT language was devised; this is a lanuage that recognises patterns in XML documents and transforms them according to a standard specification.
In our work, we have used the XSLT language for more than just transformation of content to layout, we increasingly *generate software* with it. Our experiences are reported in this paper. We start to discuss the advantages of the XML family of languages and tools in section 1, and continue to discuss its applicability to software generation in section 2. In section 3 we give several examples, and finally we present our conclusions.

1 Why XML is Useful for Modelling

An XML document contains *elements*, each of which may have a number of *attributes*, each of which may encapsulate sub-elements. Since XML documents also have a single root element, the result is a tree of (what you could call) objects.
This general format has a standardised representation, and the web consortium seems to have collected the critical mass to make it into a popular standard. This means that many tools are available for XML.

One class of tool provides parsing and writing libraries for a variety of programming languages. These libraries can take care of many recurring compiler tasks, including lexical analysis and name space management, because the representations of these aspects is the same for all XML documents. The fact that much of the standard compiler issues resolved before you design your own XML data format means that a lot of overhead is taken out of the use of such a format; this saves considerable time in application development, and CASE tool development is not an exception to that rule.

Another class of tools allows editing of XML documents, XSLT specifications and even document syntaxes. These tools rely on the self-descriptive (and thus reflective) nature of XML. Every valid XML document refers to an explicit representation of its syntax. This information can be used in editors to ensure that only correctly formatted information is input, and even to automatically generate a graphical interface that presents the content such that it is browsable and editable. A particularly interesting development is XML SCHEMA, which captures this reflective information in XML, which means that all the advantages of XML can be applied at reflective levels too. This drastically increases the achievable complexity and fun of XML development.

2 Why XSLT is Useful for Software Generation

An XSLT stylesheet (which we shall call XSLT *transformers* in this paper) recognises patterns in an XML document, and transforms these to another form. This recognition of patterns is quite instrumental in describing compilers. The possibility to specify 'queries' that search the input replaces searching in self-made, compiler-internal storage structures.

The input of an XSLT transformer is a tree which is traversed as required to generate a tree-structured output. This implies that any tree structure can be generated from XSLT. Among the tree structured file formats are not only HTML and XML, but also *programs* in most modern programming languages. This means that it is possible to generate programs with XSLT transformers.

One additional property of XML makes it particularly suitable for software generation through XSLT transformations, namely the possibility to mix plain text (in our case, program text) with XML elements to be resolved by another XSLT transformer. Because this involves the use of multiple XSLT transformers applied in a sequence, it is possible to separate concerns according to software engineering principles; notably, to implement different design decisions in different XSLT transformers.

In our experiments with XSLT as a software generation language we have worked with XSLT transformers that make decisions like 'web enable through cgi-bin scripting' or 'use SQL for queries' or 'exploit Motif'. We believe that this grain-size is quite usable in practice.

The dependency structure between XSLT transformers seems to vary between software generation projects, although not between source documents. Furthermore, the dependency structure may mix in reflective constructs. This means that the transformations can get complicated and project-specific, but not beyond the reach of make. Since the result of this project-specific tweaking usually is a self-built sort of CASE tool, we think the effort of describing the dependencies that way is more than worthwhile.

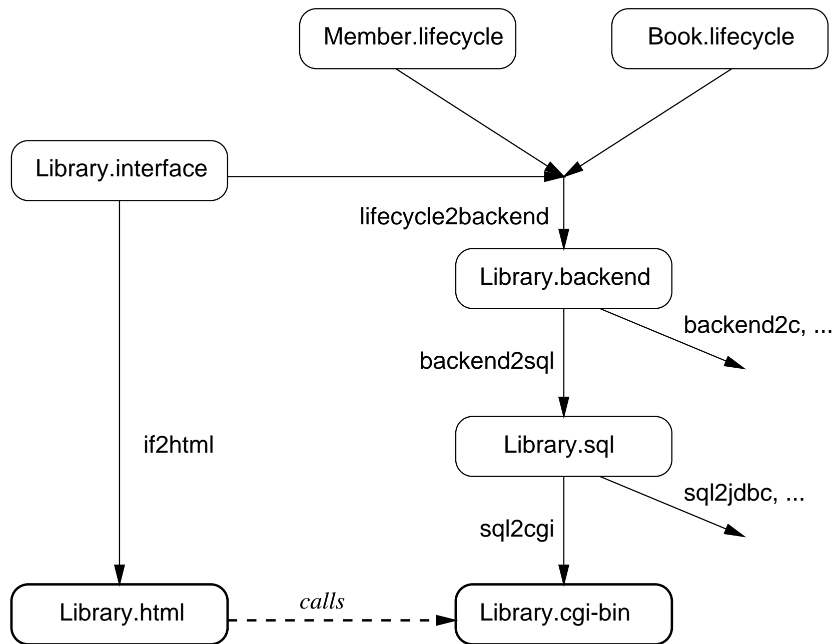

Figure 1: Translating Life Cycles to a web application.

3 Examples Software Generation Projects

This paper is a report of practical experience from several projects. In this section, we describe some situations where we benefitted from our (ab)use of XSLT for software generation.

3.1 Mapping Processes to SQL

In my research, I specify *life cycles*, informally explained as communicating state diagrams. We use the CASE tool TCM to input diagrams, and transform its native storage format into a suitable XML description of life cycles.
Figure 1 shows how these XML files are translated to a web application. Each of the $x2y$ labels refers to a transforming step from format x to format y. The lifecycle format describes the processes in an application, in this case a Member and a Book in a library case study. The interface format describes externally visible events. The backend format integrates knowledge from the input life cycles, based on the interface description, and is the starting point for both software generation and our planned formal verification. The sql format contains the information from the backend, implemented in SQL with such XML annotations that it may be embedded in many 'host' languages, like Perl, Java and PHP3.
Some transformation steps could be encoded in XSLT, but not all (at least not without hacking around the intentions of XSLT).
A fragment of the library case study on such a life cycle description is the following, describing how a Book can go from Available to Out state when a

checkout event occurs with that book's identity as the first parameter.

```
⟨lifecycle name="Book" startstate="START" endstate="END"⟩
  ...
  ⟨transit from="Available" to="Out"⟩
    ⟨event name="checkout"⟩
      ⟨param name="this" type="Book" use="out"/⟩
      ⟨param name="m" type="Member" use="in"/⟩
    ⟨/event⟩
  ⟨/transit⟩
  ...
⟨/lifecycle⟩
```

The lifecycle2backend transformer combines a few of those life cycles, and holds them against an interface, which is an XML document describing externally visible events. We regret that XSLT is constrained to a single input file, even though we could *hack* our way around it; this is why lifecycle2backend is not implemented in XSLT.

Our preference would have been to construct a binary operator on life cycle descriptions, used to unite two life cycles to one life cycle. Such an approach would greatly benefit from the ability to perform algebraic analysis on the transformer, in this case to prove commutativity and associativity. Such checks help to ensure the correctness of the transformers. We believe such checks can be performed using structure induction based on the input/output file structure, which is well defined since they are XML files.

The backend format can be transformed to, say, SQL code. This is a decision which deserves its own transformer because it defines a decision which has alternatives; such alternatives can be specified in other transformers on the backend format. Since SQL is a language which may be embedded in practically everything, it is fruitful to defer the decision of the programming language to the next XSLT transformer.

The SQL code representing the checkout (book,member) concrete event occurrence (where book and member symbolically represent the actual parameters) is generated by backend2sql, and it consists of two parts. The first part is a precondition to implement blocking behaviour, essentially expressing that a Book that is checked out cannot be checked out again:

```
⟨precondition maximum="0" message="..."⟩
  select count(*)
  from Book
  where state='Out' and this=⟨actual param="book"/⟩
⟨/precondition⟩
```

Note how useful the mixture of 'plain text' and XML elements is here: Instead of an embedding-dependent notation ? for a parameter, we used an XML element to represent our intention more accurately. Also note how the implementation of a precondition element may differ between host languages, and how that is hidden from this representation by use of an XML element. These techniques realise the separation of concerns between transformers, by selectively marking things as work to be done in a following transformer step.

The second part of this transaction is the actual update for this transition:

```
⟨transit lifecycle="Book" message="..."⟩
  update Book
  set m=⟨actual param="member"/⟩, state='Out'
  where this=⟨actual param="book"/⟩ and state='Available'
⟨/transit⟩
```

Combined with the precondition, and possibly other effects of checking out a book in other life cycles, a transaction may be formed some SQL embedding interface such as JDBC or DBI. This can now be done with a next transformation step: Another transformer for another design decision. The last stages typically require tailoring with options, such as the host, username and database name for database access. Since XSLT does not incorporate options in its translation process, we decided to use another transformer language for this last stage.
More information on life cycles can be found on the web, http://www.cs.utwente.nl/~vanrein/research.lc.html.

3.2 User Interface Generation

Modern computing has given us means of portability across platforms, of which Java and XML are just two examples. User interfaces are not similarly easy to port to different platforms, because a user interface mostly depends heavily on the functionality of a particular widget library. For instance, Windows offers native support for showing tree structures, which lacks in Tcl/Tk. We therefore did a project to describe user interfaces in a general XML format. We use XSLT transformers for the translation of this generic user interface format to specific program code in a particular programming language. This makes it possible to describe a user interface once and translate it to any development environment for which an XSLT transformer exists. Such environments currently include Java/Swing, Tcl/Tk, Appbuilder/Motif and Delphi.

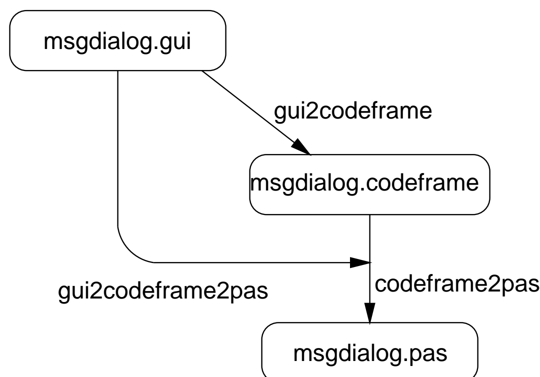

Figure 2: Generating an XSLT transformer to allow editable codeframes.

Figure 2 shows how the general user interface XML files are translated to the development environment. A message dialog, described in the msgdialog.gui file, is the starting point of the XSLT traject.

Such a file looks like:

```
⟨window id="msgdialog" text="...window title..."⟩
  ...
  ⟨button id="okbutton" text="Show message" onclick="true"/⟩
  ...
⟨/window⟩
```

This is a window with a button named okbutton, which responds when it is clicked on by invoking a bit of code.
Each XML file describing the user interface of msgdialog will be transformed to a codeframe. A codeframe is a framework in which the developer types his language-specific program code. A codeframe XSLT transformer creates an empty ⟨code⟩ tag for each event in the window description. A codeframe looks like:

```
⟨codeframe⟩
  ...
  ⟨code id="okbutton" event="onclick"⟩
    begin
      ShowMessage ("Hello, world!");
    end;
  ⟨/code⟩
  ...
⟨/codeframe⟩
```

Here, the onclick behaviour of the previously seen button okbutton is described in a codeframe entry; the three lines of Pascal are a specific implementation of an event response for Delphi. Note that the separation of code fragments is advantageous for future extensions, including the embedding of code fragments of one language (like SQL) into another; this can be done with an XSLT transformer that operates on codeframes.
The generated codeframe is originally empty, but on regeneration of the codeframe, already previously entered program code is retained to make it possible to keep the user interface and program code integrated while the application evolves. It is the developers' task to fill in the contents of the ⟨code⟩ tags, but combining them remains the responsibility of this XML based mini CASE tool.
The second step in our parse traject is to generate the program code for the selected development environment. Because it is currently not possible to jointly parse two XML-files like the window description and the codeframe with a single XSLT transformer, we have chosen to split this step into two sub-steps. First, we transform the window description with an XSLT transformer into another XSLT transformer. This XSLT transformer contains all the language-specific code necessary for displaying the user interface. At places where the ⟨code⟩ tags from the codeframe are to be inserted, an XSLT command like

```
⟨xsl:value-of select="codeframe/code[@id='okbutton' and @event='onclick']"/⟩
```

is inserted. This selects the Pascal code snippet given above. The second and last step evaluates this *generated* XSLT transformer to insert the codeframe in the rest of the program code. The result can be compiled with a traditional compiler.
More information and code of this interface generation project can be found on http://www.cs.utwente.nl/~vanrein/PLUG.

3.3 Database Schema Generation

A last practical example shows the use of XML SCHEMA, which are XML representations of XML document structure. The reflective information contained in an XML SCHEMA makes it possible to generate reflective code, such as the SQL statements to create a set of database tables that can contain the XML data instantiated from the XML SCHEMA.

We used this technique on a project, the Linux hardware Support Database. This database includes a mirrorring scheme where (database schema independent) representations of the database content are transferred in an XML document over secure SSH network connections. For this mirroring process, we also *generated* (from the XML SCHEMA) a program that reads XML documents from the secure input stream and puts its contents in the database which has the database schema that was also generated.

The benefit of this approach is, once again, the separation of concerns. The mirrorring process passess around a standardised format, and several database schemas can read/write that format. This is all fairly standard. What the above scheme adds, is the automatic generation of all access to the database schema; another database schema can be selected by changing to another set of XSLT transformers. This makes it possible to employ different schema optimisation techniques on different databases, but still support the communication for mirrorring.

An interesting problem that comes up in this situation is that XML documents are represented as trees of objects, whereas (relational) database schemas contain a more general network of tuples. Furthermore, order is not implied by database tables, whereas XML implicitly *always* assumes order to be information. This is due to the single focus on *parsing* XML files based on schema definitions, and insufficient attention to *writing* them.

It should be noted that IBM has launched a similar project named XLE. In this project, they worked from a DTD rather than an XML SCHEMA, and not surprisingly, they need annotations to address the differences between a relataional database schema and XML representation.

Conclusions

We experienced XML as a fruitful data representation, and well worth the effort of mapping a native format to. The XSLT language has been similarly fruitful in transforming conceptual descriptions to an implementation. It seems to be a good idea to construct the transformation as a number of sequential steps, where each XSLT stylesheet represents a design decision; this provides a placeholder for alternative decisions.
We stretched XSLT a bit beyond the current treatment that it receives from the web consortium, which led to a number of problems:

1. An XSLT transformer takes only one input, and generates only one output file. This means that transformations like the life cycle compiler cannot be (directly) implemented in XSLT.
2. It is not possible to make small modifications of the behaviour of an XSLT transformer by supplying options and arguments to it.
3. We would be quite pleased to see tools to verify an XSLT transformer by treating it as an operator, and use structural induction to verify algebraic properties such as transitivity, idempotence, monotonicity, associativity and commutativity.
4. The XML SCHEMA language specifies sufficient information for parsing an XML document; however, subtle matters like whether or not the data is ordered is lacking, which makes writing out XML documents based on their schema non-deterministic.

Since we believe that our use of XSLT extends the line of thinking of the web consortium, we hope and expect that these matters can be resolved in future versions of the XSLT specification.

Acknowledgements. We wish to thank the students Sander Evers, Kristiaan Breuker and Jeroen van Nieuwenhuizen for their work on the graphical interface generator, and Roelof van Zwol and Maurice van Keulen for helping to tame them.