iMatix Corporation sprl

13/15 rue des Ateliers, 1080 Bruxelles, Belgium
Tel.+32-75-235-984, fax.+32-2-414-8404
http://www.imatix.com
info@imatix.com

Technical White Paper

Template-based Code Generation

What is 'template-based code generation', and why is it so important in today's world of Internet applications? We look at the GSL technology developed by iMatix Corporation and compare this to other techniques such as XSL.

This document is aimed at a technical audience with some knowledge of current Internet standards, particularly XML, and developers looking to understand and exploit code-generation techniques.

Copyright

Version Information

Written: 15 November 1999
Revised: 23 January 2000

Disclaimer

The information contained in this document is distributed on an "as-is" basis without any warranty either expressed or implied. The customer is responsible for the use of this information and/or implementation of any techniques mentioned. iMatix Corporation has reviewed the information for accuracy, but there is no guarantee that a customer using the these techniques and/or information will obtain the same or similar results in its own operating environment.

It is possible that this material may contain references to, or information about, iMatix Corporation products or services that have not been announced. Such references or information must not be construed to mean that iMatix intends to announce such products or services.

iMatix Corporation retains the title to the copyright in this paper, as well as title to the copyright in all underlying works. iMatix Corporation retains the right to make derivative works and to republish and distribute this paper to whoever it chooses to.

What Is Template-based Code Generation?

Template-based code generation is a powerful way to reduce the cost and effort of writing code. It's a technique that's underused by most developers, because there is little literature on this subject, and there are few decent code generation tools.

In this paper I'll present a short history of code generation, and then explain how to use and exploit the GSLgen code generator, an Open Source code generator developed by iMatix Corporation that we use heavily in our products and work.

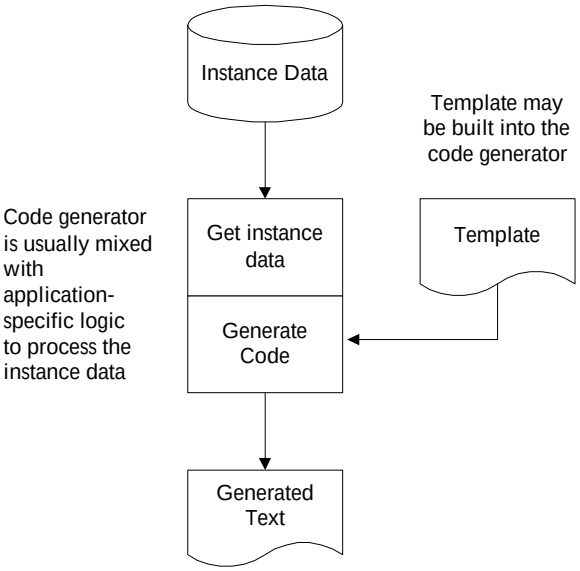
Defining Code Generation

To generate code, we identify the parts of an application's source code that are repetitive, and we then produce these mechanically, rather than manually. By 'source code' I mean all texts that go into the production of an application, including documentation, scripts, definitions, makefiles, and so on. Without code generation tools, developers write all their code by hand. This can be efficient if quality and documentation are kept high, but generated code is cheaper and more reliable when it's well applied.

Code generators work by combining various pieces of information to produce an output file. We define these terms:

- The 'template' is the unchanging part of the output.
- The 'instance data' defines the specifics for each case.
- The 'generated text' is the resulting output file.

The Code Generation Process



These terms make more sense when we look at some examples:

- A mail-merge application mixes names and addresses (instance data) with a standard form letter (template) to produce a series of customised letters (generated text).
- A CGI (web) program mixes data to display (instance data) with a standard HTML file (template) to produce a HTML page ready to be sent to the browser (generated text).

All applications contain repetitive code: one challenge is to define and isolate this. Applications lie on a bell-curve of repetition. At one extreme, very little can be produced mechanically. At the other end, just about everything can be generated. In the middle, the bulk of applications can be profitably built from a mix of generated and hand-written code.

As a programmer, I'm naturally interested in generating programs, but a code generator can just as well act as - for example - a mail-merge application.

Code generation is thus, at least, the process of mechanically producing source code and the other texts that are used in the construction of a software application, and in a wider context, any mechanical production of text files.

A History of Code Generation

In the fifteen years that I've written and worked with code generators, I've come to classify these tools in three distinct classes, or generations, each elaborating a set of techniques. Many developers have discovered these techniques independently, and have just as independently fallen into the same pitfalls.

The First Generation

A first generation code generator (*g1cg*) is usually produced by a developer who has identified repetitive source code, and is tired of producing it by hand. A g1cg has these characteristics:

1. It produces one type of generated text.
2. The template is implicitly hard-coded within the code generator as 'print' instructions. Changing the template means changing these instructions within the code generator.
3. Therefore, changing the template is expensive. End-users cannot change the template.
4. The generated text is usually low quality, since it's expensive to change and thus improve the template.

Using a g1cg is usually better than writing code by hand, but large projects will eventually produce many g1cg's, each for different needs. The main problem with a g1cg is that the template is expensive to modify, and the tool is limited to producing one small set of output files.

The Second Generation

In a second-generation code generator (*g2cg*), the designer moves the template to an external file. The g2cg is then a parser for this file. A g2cg has these characteristics:

1. It still generates code for a specific case.
2. The template is an external file.
3. The tool user can modify the template, a good thing for everyone.
4. The generated code can be of very high quality, since the template easy to modify, customise, and tune. A well-tuned template can produce code that is much better than hand-written code.

Issues that are costly to solve in a first-generation code generator become trivial to solve e.g. changing the template to generate SQL for one database in place of another. The mail-merge and CGI applications described above are examples of g2cg's.

The disadvantage of using a second-generation approach is that each code generator ends-up defining its own template language. Few developers are good at writing parsers, so the ad-hoc template languages they develop can be cryptic to the tool user. Since templates are meant to be changed and maintained by the tool user, this can eventually become a problem.

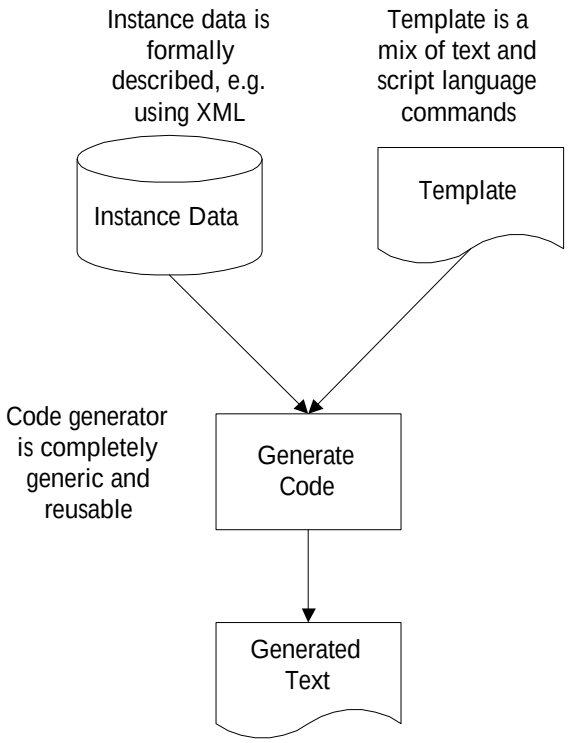
The Third Generation

In a third-generation code generator (*g3cg*), the code generator becomes a parser capable of interpreting a general-purpose template language. A g2cg has these characteristics:

1. It can generate code for an unlimited range of problems.
2. The template file starts to looks like a program or script.
3. The instance data becomes an explicit part of the code generator API.

In a g3cg, problem-specific processing moves from the code generator to the template, and the code generator is reduced to a fast, general-purpose engine capable of doing the basic task of mixing instance data with template data. The competence needed to write a code generation engine is separated from the competence needed to write a code generation application.

3rd-Generation Code Generator



The GSLgen Code Generator

At iMatix Corporation we make Open Source software (OSS) tools, mainly for use in our own products and projects. I started designing a general-purpose code generator as long ago as 1990 when I was working on a COBOL CASE tool called ETK[1]. I was getting a little tired of writing, maintaining, and using the numerous g2cg's we had developed for ETK.

Starting in 1996, Jonathan Schultz and myself built a number of prototype code generators in Perl. We defined a general-purpose template language, GSL, that looks a little like a scripting language, and is based on several g2cg's we wrote and used.

We also realised that an ideal form for the instance data was some kind of hierarchical format, rather than the flat-file tabular format used in g2cg's like a mail-merge program. We played with several alternatives, and then discovered XML (extensible mark-up language) [2,3]. An XML data file is easy to parse and load into memory as a tree. GSL lets the template writer play with this tree, and use it to drive the code generation process.

We built a prototype GSL interpreter in Perl, using the XML::Parser [4] written by Larry Wall and Clark Cooper, an interface to James Clark's XML parser, expat. The resulting program (GSLperl), is simple. Thanks to XML::Parser, loading an XML file is a one-line operation. GSLperl also depends on Perl's interpreter to handle expressions, rather than parsing these itself.

GSLperl was too slow for production use, and our main effort went into writing a fast GSL interpreter in C. The result, GSLgen, is portable and efficient and useful, and we've already used it in many projects. GSLgen is based on our OSS SFL library [5] (which includes XML i/o functions), and we built it using Libero [6], an OSS program construction tool.

Using XML

GSLgen expects its instance data to come as a well-formatted but non-validated XML file. That is, it does not use a formal Document Type Definition (DTD) to validate the XML file. We chose XML because it is, frankly, an excellent way to represent structured data.

---

1 http://www.etk.com - The ETK CASE tool is available as Open Source Software for various platforms

2 http://www.xml.org - Portal for XML information.

3 http://www.w3c.org - Home of the WWW Consortium, which designed XML.

4 http://www.cpan.org - The Comprehensive Perl Archive Network. Search for “xml::parser”.

5 http://www.imatix.com/html/sfl/ - The SFL home page.

6 http://www.imatix.com/html/libero/ - Libero is also described in Dr Dobb's Sourcebook July/August 1996.

It's also simple, robust, portable, widely supported, and standard. This is how I might represent a simple list of currencies as an XML file:

```
<CURRENCY_LIST>
  <CURRENCY NAME="BEF" DESCRITPION="Belgian Franc" />
  <CURRENCY NAME="HKD" DESCRITPION="Hong Kong Dollar" />
  <CURRENCY NAME="USD" DESCRITPION="US Dollar" />
</CURRENCY_LIST>
```

The instance data is fully self-describing, and the code generator reads it with minimal checking beyond the basic check that it is a well-formatted XML file. If attributes are missing, they can get default values. If there's extra, unexpected instance data, the code generator just ignores it.

Using a template language like GSL, and XML to represent the instance data, we can generate any kind of text - HTML, source code, PostScript, SQL, junk e-mails, and so on.

An indirect advantage of using XML for the instance data is that the code generator becomes a more generally-useful tool. XML is used for many applications in its own right, and a GSL template can do more than just generate code. It can validate the XML, using intelligence that is hard to build into classic XML-processing technology such as DTDs and XSL style sheets.

In fact, it's possible to write whole XML-processing applications in GSL. Code generation is just one slant on this, even just a special case (although it remains GSL's focus).

The Generator Script Language, GSL

A GSL template is a code generation application. The template decides what to generate, when, and how to do it. It does this by issuing GSL commands like '.output', which creates a new output file, and '.for', which starts a loop. The template mixes GSL commands with the text it wants to output. GSL commands start with a dot at the start of the line. Let's look at a small example that generates some HTML:

```
.output "itemlist.htm"
<HTML><BODY><H1>Generating HTML</H1><UL>
.for currency
<LI>$(name:) - $(description:)
.endfor
</UL></BODY></HTML>
.close
```

Here, the instance data describes some 'currency' that has two properties, 'name' and 'description'. The template implements a loop that creates a bulleted list for the items. The code generator works through the template line by line, and looks for template commands like '.output', '.for', and '.endfor'. It copies anything else to the file it's creating, here "itemlist.htm". GSLgen interprets the template commands, and inserts the instance data as required.

GSLgen also works in a non-template mode, looking more like a conventional scripting language. In this mode, output lines start with '>':

```
output "itemlist.htm"
><HTML><BODY><H1>Generating HTML</H1><UL>
for currency
><LI>$(name:) - $(description:)
endfor
></UL></BODY></HTML>
close
```

GSL evolved over several years and is based on other code generators we've written and used, mainly Libero, and the htmlpp HTML preprocessor [7] we use for our websites.

The requirements for such a language are modest compared to full-blown programming languages. It does not need complex data types, since the instance data is the principle data object. Rather, it needs fine control over the alignment and formatting of the output text. We also designed GSL around XML's capacity for structure and repetition.

In the current release – GSL/2.000 – we are adding object-orientation, extensibility, and in general turning GSL into a language capable of operating in many situations. Its principal strength as a code generator remains.

Basically, a GSL code generator like GSLgen loads one or more XML files into memory, then executes the template from start to end. These are some of GSL's principal commands:

```
.output <filename>
```

Start creating a new output file

```
.include <filename>
```

Include another template file

```
.interpret <expression>
```

Interpret expression as GSL

```
.echo <expression>
```

Echo expression to standard output

```
.abort <expression>
```

Display an error message, and abort

```
.if <expression>
```

Start a conditional block

```
.while <expression>
```

[7] See http://www.htmlpp.org.

Start a repeated block

```
.define <name> = <expression>
```

Define a new attribute at some level

GSL uses the concept of 'scope' to address different levels within the XML data. The .for command opens a new scope, usually a child of the current item:

```
    .for [<scope>.] <name>
        [as <alias>]
        [where <expression>]
        [by <expression>]
    .endfor
```

The .for command acts like a combined select, sort, and loop. It's the essential GSL building block. Using the .for command, the template iterates through a list of items, and generates code as required. The 'where' clause selects a subset of items depending on some logical condition. The 'by' clause orders the items.

As a GSL template can become quite complex, GSL lets you define subroutines, called macros. The user has full control over the look and feel of the generated code, including alignment and spacing. GSLgen can generate any programming language, including COBOL in 80-column format with line numbering.

Using GSLgen

The main reason to use GSLgen is that it becomes cheap to apply code generation techniques to any problem. All one needs to do is:

1. Produce the XML instance data.

2. Write the templates.

3. Integrate GSLgen into the application toolkit.

I'll demonstrate this with some concrete examples.

Generating SQL

In one project, we needed to produce large SQL scripts to manage replication between copies of a database . The vendor's own replication tool (a classic g1cg) produced SQL that was incomplete. We had the choice of patching this, or generating our own. We wrote a program to extract the database catalogue definitions for a table as XML. We wrote this program using C and embedded SQL, but could easily have done it using Visual Basic and ODBC, Java and JDBC, etc. Then we took the vendor-generated replication code, and turned this into a GSL template. Finally we wrote a Perl script to wrap it all together with some simple menus. Our end-user, the database administrator, was happy to modify and tune the template, without technical knowledge of how the code generator worked, and a minimal explanation of GSL.

We've used the same approach to generate many kinds of SQL, including scripts to create databases, mirror data, compare database table contents, and so on.

GSLgen in a Web Server

In our Xitami web server [8], we use GSLgen in a number of places, including:

- To allow server-side XML processing. Using a simple GSL script, it's easy to transform XML data files into HTML pages.
- To produce directory listings. These are HTML pages showing the contents of a directory. Here the instance data is the list of files, their size, type, and permissions. The template can show icons for each file type, can add links, and so on.
- To produce HTML pages for HTTP errors like '404 not found'. The instance data holds the cause of the error, plus all the information that the web server has about the user. This template can make intelligent decisions about the cause of the error, even the type of browser, and so produce appropriate output.

[8] See www.xitami.com - The Xitami web site.

- To analyse log files. Xitami creates XML log files, and it's useful to analyse them using GSLgen.

These applications for GSLgen are modest but offer a great deal of flexibility at a low cost. Previous releases of Xitami acted like g1cg's, with hard-coded output. It took us only a few hours to add GSL capability to Xitami.

A Large-scale Code Generation Application

A more heavyweight GSL application is our Studio Workbench, a web-based application generator that iMatix Corporation is developing . This is basically a set of GSL templates that produce all the components of web-based application programs from a XML-based data dictionary. It's this work that is pushing the development of GSL and GSLgen. For example: one of our targets is that the code generator should work at least as fast as a compiler. This is easy for simple applications, but the Workbench template makes multiple passes in order to generate code. A naïve approach to GSL interpretation is too slow, so we spent quite a lot of time optimising the way GSLgen works.

Makefile and build-script generation

Our software runs on many platforms, and for each package, we need to provide makefiles, build scripts, and install scripts. For instance, the Xitami package includes build scripts for Unix, OS/2 and Windows, makefiles for these systems, and several install scripts for Windows. Currently we manage these half-manually and half using various ad-hoc code generators. We're looking at using GSLgen to generate these various files from a single master XML file.

Web Sites

Up to now we've used the **htmlpp** HTML pre-processor to build our websites. This tool does make it easy to manage the 700-800 pages on imatix.com and xitami.com, but we're planning to use GSLgen instead. In this way we could describe our web site formally as XML data, with our own mark-up language. A GSL template would then turn this into HTML. The difference between this approach and a 'stylesheet' approach like XSL is that GSL can, like htmlpp, create tables-of-contents, indexes, and multiple output pages from a single source file.

Embedding GSLgen in Other Applications

Any application can produce XML instance data and then call the GSLgen code generator, and finally frees the XML tree. This kind of embedding is easy if you're writing C or C++ programs, or working in a language that can call C functions. As OSS, GSLgen comes with full source code. For other applications, it can be easier to write the XML data to a sequential file, and then start GSLgen as a command-line process.

Debunking Some Myths about Code Generation

Code generators are often seen as a technological burden, rather than as useful tools. I suspect that this is because g1cg's and g2cg's fail to deliver their full potential. Some of the common myths about code generation are:

- "Code generators only work for simplistic cases". This is often true, but only because most code generators are simplistic. A template-based code generator like GSLgen is extraordinarily flexible, and can generate highly complex texts and source code.
- "Generated code is unreadable", and "Generated code is low quality". This is often true for g1cg's, because their designers focus on the application-specific problem, rather than on making the template easy to modify and improve. In a g2cg and g3cg, the generated code can be as good as, and often is much better, than hand-written code.
- "Code generators are expensive". Again, this is typical of g1cg's, where the slightest change to the template means modifying, compiling, linking, and distributing a new release of the code generator. The cost of building and using a g2cg is much more reasonable. And you can get g3cg's for free.
- "Code generators are complex". Any program that solves multiple problems is going to be complex. A g1cg and g2cg handles both application-specific issues and code-generation issues, and is often a large and complex program. A g3cg does just one job, and can be simple. A g3cg written in a text-processing language like Perl can be just a few pages long.
- "Code generators are too much effort". Many tools require a serious learning curve. This says more about tool designers than about the problems that the tools solve. One things I like about GSL is that the language is simple enough to give to non-expert technicians, yet it's rich enough to build really complex toolkits.

Designing For Code Generation

To get the most from a code-generation approach, you need to design the application around this. It's like building houses. If every house is different, you need new plans each time. If houses are based on a common model, you can design a new house simply by defining the variations from the common model. In the same way, code generation lets you build programs by making variations on a common model. The application has to be designed so that such programs are what it needs. It is not economical to generate special cases. Ergo, the application is built mostly from common cases. Even if you can't generate everything, you can often generate specific types of source code.

A simplistic approach to code generation can lead to applications in which the user-interface is fragmented into too many ‘simple’ steps. This is not necessary. A code generation approach can be sophisticated enough to build complex components, for instance mixing customised code with the generated code.

Comparative Products

Scripting and Template Languages

Several web programming languages allow template-based code generation, as a by-product of their principal duty as HTML generators:

- Microsoft's VBScript/ASP combination,
- PHP,
- PerlScript, and so on.

These tools have the same disadvantages for application as general-purpose code generators:

- They are not designed to work with abstracted instance data, but with a small data set, usually coming from the form.
- You cannot easily embed these in other applications to do general-purpose code generation.

These tools are oriented towards HTML production, and may be unsuitable for more general-purpose tasks. However, it would probably be possible to extend any of these languages to work in a GSLgen-like fashion.

Other Code Generators

Few software engineers have studied the problem of code generation in detail, and there are few tools that pretend to do this. There are a few notable exceptions.

In 'Advanced Perl Programming' [9], Sriram Srinivasan writes lucidly about the advantages of template-driven code generation, and presents a g3cg, 'Jeeves', written in Perl. The Jeeves template language has several similarities to GSL, showing that there are strong evolutionary forces at work. As far as we know, both tools evolved independently. To supply Jeeves with data, you write a small parser for your specific instance data language. Jeeves is a remarkable illustration of the power potential of a small g3cg, but does not support XML directly.

Several CASE tools include code generation abilities. These are almost always based on a specific design methodology, usually object-orientation. I do not know whether such tools generally use templates, but in my experience to date, they do not. This makes them of limited use even for the applications they are intended to produce, and of no use at all for general-purpose work.

---

[9] http://www.oreilly.com/catalog/advperl/ - Chapter 17, "Template-Driven Code Generation".

The XSLT Language

The XSL (extensible stylesheet language) is driving a set of standards that promise template-based transformation of XML that starts to look like the type of work we can do with GSL. We will look at one of these, XSLT[10], a language primarily intended for XML transformations (i.e. production of one XML file from another) but which can be used for more general-purpose code generation.

XSLT is very significant because it marks the first Internet standard (produced by the world-wide-web consortium, the W3C) directly or indirectly aimed at template-based code generation. As we see, it has many parallels with GSL, but still falls short in places.

XSLT Example

This is the XML file we will use as an example – it is taken from the W3C XSLT documentation edited by James Clark:

```
<sales>
    <division id="North">
        <revenue>10</revenue>
        <growth>9</growth>
        <bonus>7</bonus>
    </division>
    <division id="South">
        <revenue>4</revenue>
        <growth>3</growth>
        <bonus>4</bonus>
    </division>
    <division id="West">
        <revenue>6</revenue>
        <growth>-1.5</growth>
        <bonus>2</bonus>
    </division>
</sales>
```

[10] http://www.w3.org/TR/xsl

Transforming The XML Data Into HTML

This is an XSLT stylesheet that transforms the XML data into HTML:

```
<html xsl:version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      lang="en">
<head><title>Sales Results By Division</title></head>
<body>
    <table border="1">
        <tr>
            <th>Division</th>
            <th>Revenue</th>
            <th>Growth</th>
            <th>Bonus</th>
        </tr>
        <xsl:for-each select="sales/division">
            <!-- order the result by revenue -->
            <xsl:sort select="revenue"
                      data-type="number"
                      order="descending"/>
            <tr>
                <td><em><xsl:value-of select="@id"/></em></td>
                <td><xsl:value-of select="revenue"/></td>
                <td><!-- highlight negative growth in red -->
                    <xsl:if test="growth &lt; 0">
                        <xsl:attribute name="style">
                            <xsl:text>color:red</xsl:text>
                        </xsl:attribute>
                    </xsl:if>
                    <xsl:value-of select="growth"/></td>
                <td><xsl:value-of select="bonus"/></td>
            </tr>
        </xsl:for-each>
    </table>
</body>
</html>
```

This is the equivalent GSL template (note that $(id) gets the value of the attribute called 'id', and $(->revenue) gets the value of the child called 'revenue':

```
<html lang="en">
<head><title>Sales Results By Division</title></head>
<body>
    <table border="1">
        <tr>
            <th>Division</th>
            <th>Revenue</th>
            <th>Growth</th>
            <th>Bonus</th>
        </tr>
.for division
.   define division.revenue = ->revenue
.endfor
.for sales.division by revenue
            <tr>
                <td><em>$(id:)</em></td>
                <td>$(->revenue:)</td>
.- highlight negative growth in red
.if ->growth < 0
                <td style="color:red">
.else
                <td>
.endif
                    $(->growth:)</td>
                <td>$(->bonus:)</td>
            </tr>
.endfor
    </table>
</body>
</html>
```

Transforming The XML Data Into SVG

This is an XSLT stylesheet that transforms the XML data into SVG:

```
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns="http://www.w3.org/Graphics/SVG/SVG-19990812.dtd">
<xsl:output method="xml" indent="yes" media-type="image/svg"/>
<xsl:template match="/">
<svg width = "3in" height="3in">
    <g style = "stroke: #000000">
        <!-- draw the axes -->
        <line x1="0" x2="150" y1="150" y2="150"/>
        <line x1="0" x2="0" y1="0" y2="150"/>
        <text x="0" y="10">Revenue</text>
        <text x="150" y="165">Division</text>
        <xsl:for-each select="sales/division">
            <!-- define some useful variables -->
            <!-- the bar's x position -->
            <xsl:variable name="pos" select="(position()*40)-30"/>
            <!-- the bar's height -->
            <xsl:variable name="height" select="revenue*10"/>
            <!-- the rectangle -->
            <rect x="{$pos}" y="{150-$height}"
                  width="20" height="{$height}"/>
            <!-- the text label -->
            <text x="{$pos}" y="165">
                <xsl:value-of select="@id"/>
            </text>
            <!-- the bar value -->
            <text x="{$pos}" y="{145-$height}">
                <xsl:value-of select="revenue"/>
            </text>
        </xsl:for-each>
    </g>
</svg>
</xsl:template>
</xsl:stylesheet>
```

This is the equivalent GSL template:

```
<svg width = "3in" height="3in">
    <g style = "stroke: #000000">
.- draw the axes
        <line x1="0" x2="150" y1="150" y2="150"/>
        <line x1="0" x2="0" y1="0" y2="150"/>
        <text x="0" y="10">Revenue</text>
        <text x="150" y="165">Division</text>
.for division
.- define some useful variables
.- the bar's x position
.  define pos = index () * 40 - 30
.- the bar's height and y position
.  define height  = ->revenue * 10
.  define rect_y  = 150 - height
.  define value_y = 145 - height
.- the rectangle
        <rect x="$(pos)" y="$(rect_y)"
              width="20" height="$(height)"/>
        <!-- the text label -->
        <text x="$(pos)" y="165">$(id:)</text>
        <!-- the bar value -->
        <text x="{$pos}" y="$(value_y)">$(->revenue)</text>
.endfor
    </g>
</svg>
```

These examples show how GSL and XSLT compare. Both are template-based code generators that can manipulate the XML data using loops, tests, and so on.

In these examples we note that GSL and XSLT appear quite similar. But these examples are taken from the XSLT documentation, and represent trivial cases. When we look at larger-scale code generation, XSLT starts to look weaker.

Comparison of GSL and XSLT

In many ways, you can use XSLT and GSL interchangeably. But there are differences:

1. GSLgen has its origins in code generation, while XSLT has its origins in XML transformation. These are not the same, and GSLgen has many syntactic refinements that are intended to allow code generation for different programming languages and environments – for example the manipulation of symbol values to suit different programming languages' requirements for variable names.
2. Since XSLT is an XML tag language, it is fairly verbose. GSL is more compact, and probably easier to work with (the GSL commands are visually distinct from the rest of the source code.
3. XSLT is still a proposal (version 1.0 dates from 8 October 1999).
4. XSL tools in general have suffered from several problems: they are mostly very complex to install and use, often non-portable, and often include proprietary extensions. When XSLT becomes a standard, it is unclear that good free portable XSLT processors will be available.

5. XSLT is an inherently complicated language, and the investment needed to create good XSLT tools will be significant (in our opinion, the same problems have lead to a lack of good XSL tools).

Some of the abilities that GSL provides that are lacking in XSLT are:

- Some kind of subroutine capability to encapsulate code used several times. GSL allows macros and functions that can be invoked throughout the template.
- Manipulation of values to adapt to different external requirements. For instance, GSL lets you output a value like “this is a title” in several ways: “This Is A Title”, “THIS IS A TITLE”, “this_is_a_title”, and so on.
- Control of what is generated. A GSL template can produce several files at once, by specifying multiple .output commands.

These issues will most likely be cleared-up and improved over time, and we expect XSLT to become a standard for template-based code generation within a timeframe of 3 to 5 years.

Worked Example

In this section we present a moderately complex case where GSLgen proves to be an invaluable assistance in the code-generation process.

Our worked example implements a task-based workflow object. Workflow objects are used in more sophisticated business environments, where data objects go through a life-cycle of proposal, approval, execution, and so on.

In most workflow applications, this life-cycle is built-in to the code, an obvious way of doing it, but one with several drawbacks:

- The life-cycle is not explicitly defined anywhere, so errors are possible, even if the documentation is correct.
- It's hard (expensive) to change and improve the life-cycle.
- It's very hard (very expensive) to add multiple life-cycles to the same application.

In our example, we describe the life-cycle for a very simple task-based object – the *to-do list item*. A to-do list item has a very simple life-cycle: it's created, it stays pending until it is deleted or closed.

We can describe the life-cycle formally as a *finite-state machine*. This simple means that we note the states in which the object can exist, and for each state we note the events that can occur. Events generally map to end-user actions ('close item'). Each event pushes the item's life-cycle to a possibly new next state, and performs some actions on the way.

This a description of the life-cycle of a to-do item as an XML file:

```
<fsm name = "todo" script = "workflow.sch" >
<state name = "initial" >
    <event name = "ok"     nextstate = "pending" />
    <event name = "error"  nextstate = "final"   />
</state>

<state name = "pending" >
    <event name = "close"  nextstate = "closed" >
        <action name = "close" domain = "data" />
    </event>
    <event name = "change" nextstate = "modified" >
        <action name = "update" domain = "screen" />
    </event>
    <event name = "delete" nextstate = "final" >
        <action name = "delete" domain = "data" />
    </event>
</state>

<state name = "closed" >
    <event name = "purge"  nextstate = "final" >
        <action name = "delete" domain = "data" />
    </event>
    <event name = "renew"  nextstate = "modified" >
        <action name = "update" domain = "screen" />
    </event>
</state>
```

```
<state name = "modified" >
    <event name = "ok"     nextstate = "pending" >
        <action name = "update" domain = "data" />
    </event>
    <event name = "cancel" nextstate = "pending" >
    </event>
    <event name = "delete" nextstate = "final" >
        <action name = "delete" domain = "data" />
    </event>
</state>
</fsm>
```

Having described the life-cycle, we then write a GSL template that implements it in some way. There are many ways to implement a FSM. The simplest is a set of 'CASE' statements, selecting the next state and action depending on the current state and event.

Let's see how this would be implemented in a simple language like Microsoft's VBScript:

```
'   This subroutine implements the workflow state-machine for the
'   $(name) object.  The workflow subroutine accepts a state and
'   event as input and returns a state, action, and action arguments
'   as output.  This implementation allows one action per transition.
'
'   Input:  cur_state   Name of current state
'           cur_event   Name of event that was invoked
'
'   Output: cur_state   New state after transition
'           action      Action to execute on object
'           domain      Action domain
'
'   The function return code is 0 if the state transition was valid,
'   -1 if the cur_event is illegal in this state, and -2 if
'   the cur_state is not a valid state.

function wf_$(fsm.name)_step (cur_state, cur_event, action, domain)
    action = ""
    domain = ""
    select case cur_state
.for state
        case "$(name)"
            select case cur_event
.   for event
                case "$(name)"
                    cur_state = "$(nextstate)"
.       for action
                    action    = "$(name)"
                    domain    = "$(domain)"
.       endfor
.   endfor
                case else
                    wf_$(fsm.name)_step = -2
            end select

.endfor
        case else
            wf_$(fsm.name)_step = -2
    end select
end function
```

It's useful to be able to get the initial state automatically too. Let's generate that code:

```
'   Set initial state for object

function wf_$(fsm.name)_initial_state
.   for state where index() = 1
    wf_$(fsm.name)_initial_state = "$(name)"
.   endfor
end function
```

Finally, let's generate a routine that returns us the events valid in any given state:

```
function wf_$(fsm.name)_methods (cur_state)
    select case cur_state
.for state
        case "$(name)"
.   define events = ""
.   for event
.       define events = "$(events:)" + "$(Name)" + " "
.   endfor
            wf_$(fsm.name)_methods = "$(events:)"
.endfor
        case else
            wf_$(fsm.name)_methods = ""
    end select
end function
```

This is the full code generated for the to-do item:

```
'
'   wf_todo.asp - Workflow layer for todo
'
'   Generated: 1999/11/30 from workflow.sch
'   Script written by iMatix Corporation <pieter.hintjens@imatix.com>
'

'   Set initial state for object

function wf_todo_initial_state
    wf_todo_initial_state = "initial"
end function

'
'   This subroutine implements the workflow state-machine for the
'   todo object.  The workflow subroutine accepts a state and
'   event as input and returns a state, action, and action arguments
'   as output.  This implementation allows one action per transition.
'
'   Input:  cur_state   Name of current state
'           cur_event   Name of event that was invoked
'
'   Output: cur_state   New state after transition
'           action      Action to execute on object
'           domain      Action domain
'
'   The function return code is 0 if the state transition was valid,
'   -1 if the cur_event is illegal in this state, and -2 if
‘   the cur_state is not a valid state.

function wf_todo_step (cur_state, cur_event, action, domain)
```

```
action = ""
domain = ""
select case cur_state
    case "initial"
        select case cur_event
            case "ok"
                cur_state = "pending"
            case "error"
                cur_state = "final"
            case else
                wf_todo_step = -2
        end select

    case "pending"
        select case cur_event
            case "close"
                cur_state = "closed"
                action    = "close"
                domain    = "data"
            case "change"
                cur_state = "modified"
                action    = "update"
                domain    = "screen"
            case "delete"
                cur_state = "final"
                action    = "delete"
                domain    = "data"
            case else
                wf_todo_step = -2
        end select

    case "closed"
        select case cur_event
            case "purge"
                cur_state = "final"
                action    = "delete"
                domain    = "data"
            case "renew"
                cur_state = "modified"
                action    = "update"
                domain    = "screen"
            case else
                wf_todo_step = -2
        end select

    case "modified"
        select case cur_event
            case "ok"
                cur_state = "pending"
                action    = "update"
                domain    = "data"
            case "cancel"
                cur_state = "pending"
            case "delete"
                cur_state = "final"
                action    = "delete"
                domain    = "data"
            case else
                wf_todo_step = -2
        end select

    case else
```

```
            wf_todo_step = -2
    end select
end function

function wf_todo_methods (cur_state)
    select case cur_state
        case "initial"
            wf_todo_methods = "Ok Error "
        case "pending"
            wf_todo_methods = "Close Change Delete "
        case "closed"
            wf_todo_methods = "Purge Renew "
        case "modified"
            wf_todo_methods = "Ok Cancel Delete "
        case else
            wf_todo_methods = ""
    end select
end function
```

Of course, it's now easy to generate a corresponding program for much more complex life-cycles, such as purchase orders, project propositions, formal documents, and so on.

Conclusions

GSLgen represents a best-of-breed product that is ahead of the industry by several years. It is unique in providing a reusable, portable, Open Source XML-based template-based code generator that is unique in its power and scope.

When xmlsoftware.com added GSLgen to their library, they had no category for it, and finally placed it in 'Converters'. The concept of code generation as a tool category is poorly understood by the software industry.

GSLgen is not simply a code generator. It is a tool for constructing code generators. These range from the trivial to the extremely sophisticated. Any model that can be abstracted as XML metadata files and a template becomes a candidate for code generation.

We expect GSLgen as it stands today to be overtaken by developments, especially those surrounding XSL and XSLT, within three to five years. Future development of GSLgen is aiming towards the inclusion of a full Open Source scripting language like JavaScript. We intend to replace our second-generation code generations (especially Libero) with GSLgen during 2000.

GSL represents an excellent technology for constructing small, medium, and large-scale code generation tools based on arbitrary XML metadata and arbitrarily-complex requirements.