PROGRAM GENERATORS AND CONTROL SYSTEM SOFTWARE DEVELOPMENT

Klemen Zagar[*], Anze Vodovnik[**], J. Stefan Institute and Cosylab Limited, Slovenia

Abstract

Development of a control system typically requires that many software artefacts are kept in sync with one another. For example, if a hardware device is replaced with a newer model, which exposes some additional functionality, this might require changes or additions in device driver code, in networking protocol, the operator's user interface, and the configuration database. Such modifications are often trivial: addition of functions, graphical user interface elements, and database columns, all based on established templates. Traditionally, software engineers would be assigned the task of affecting these changes, which is error prone – e.g., forgetting to make a modification in the configuration database – and time consuming – performing regression tests to check that nothing has been broken by the widespread modification. To ameliorate these issues, we propose describing devices using XML files with a well-defined schema and generating all the software artefacts from these descriptions using templates. We have developed a specialized language called XPGL (*Extensible Program Generator Language*) for the purposes of defining the templates, which is presented in this article.

1 INTRODUCTION

The complexity of a system is measured in the number of components that it encompasses, as well as the number of interactions among them. Thus, control systems are fairly non-trivial, since every piece of hardware that needs to be controlled requires:

- A driver, which allows communication of the control system with the hardware, and some times also autonomous real-time control of the hardware.
- A graphical user interface (GUI) for the human operators, through which the hardware can be monitored and controlled.
- A configuration database entry, which describes the piece of hardware in terms of addressability and initial configuration.
- A networking protocol that allows for the above to interact.
- Some kind of a *business logic*, which models the concept of a physical device, defines its behaviour, is knowledgeable about interactions with other devices, and is capable of a higher-level control that need not be real-time.

Although there can be hundreds of devices in a control system, there are not many types of them. This means that each and every device must still be manufactured, installed, and properly configured, but it needs to be designed only once. Also, the software for the device needs to be written only once.

The work can be greatly assisted through the use of program generators, because the code is very repetitive. The repetitiveness becomes even more obvious when adhering to the principles of the object-oriented programming and decoupling code using multi-tier architecture [1]. Patterns of repetition by themselves cannot be expressed explicitly in terms of code, but they can be conveyed in a form of code templates.

Our [2] work was focused towards defining a good code template definition language, which we call the *Extensible Program Generator Language* (XPGL). The name was coined to emphasize the following:

- It is an artificial computer language with a well-defined syntax.
- It is extensible in a sense that new features can be added to the language easily.
- Its primary purpose is generating program source code.

2 PATTERNS

If there is a pattern in the code, this implies that most of the code is the same regardless of particular instantiation of the pattern, and that only a small part of the code is different on per-instance basis. For example, when writing a Java [3] class, its fields are typically not exposed publicly, and accessor/mutator method pairs are provided instead of them:

```
class SomeClass {
  private int myVariable;
  public int getMyVariable() {
    return myVariable;
  }
  public void setMyVariable(int value) {
    myVariable = value;
  }
};
```

(The code in **bold** are Java keywords, and *italics* are the parts of the code that differ from instance to instance.) In this example, the pattern is clearly visible (the non-italic text), and the placeholders for volatile code can also be inferred (the data type, ***int***, and the name of the field, *myVariable*).

3 DESCRIBING THE SOFTWARE

Apart from identifying the patterns, one also has to describe the content – the software that is to be built. In

[*]klemen.zagar@cosylab.com
[**]anze.vodovnik@cosylab.com

the example above, one has to specify that there exists a class named “some class” and that it contains an integer variable called “my variable”.
The description of the software has to satisfy several requirements:

- Agnostic of the target programming language. This will make it possible to use the same description for producing artefacts in several different programming languages.
- Easy and concise to create. If the complexity for providing the description exceeded the complexity for writing the code, use of generators would not be economical.
- Well-defined and unambiguous.
- Highly structured and flexible due to the nature of program code.

3.1 Extensible Markup Language (XML)

We found it beneficial to use the Extensible Markup Language (XML) standard [4] for describing the software. The main features of XML are:

- The data is stored in comprehensive, human-readable text files.
- XML files can be assigned a *schema*, which specifies the exact syntax of XML files. This allows validation of XML files and reduces the possibility of an error.
- High-quality parsers and validators of XML files are widely available.

The schema of XML files suitable as input to a program generator depends on the kind of the system that is described. For example, the XML files for control systems [6] would be describing devices and their control/monitor points, whereas files for banking would focus on types of accounts and their specific attributes.

3.2 Assuring Programming Language Neutrality

If there is only one target programming language, then the tokens appearing in the XML program description can be treated verbatim and simply inserted at the placeholders.
The most frequent points of difference between programming languages, as far as placeholder representations are concerned, are:

- Primitive data types: `int` in C, C++ and Java is called `xs:int` in XML schema definition language (XSD) and `Integer` in Pascal.
- Naming convention: in some languages, camel-case is frequently used (`myVariable`), whereas in others all words are capitalized (`MyVariable`). The naming convention also depends on context, for example in Java, the `myVariable` field would be accessed through function `getMyVariable`.

To circumvent these issues, we recommend:

- Always use Java types in the XML description. Convert to language-specific types in the templates.
- Use proper capitalization of words in naming tokens, and separate words with either underscores or spaces. When inserting placeholders in templates, take care of proper capitalization, which takes the context into account.

4 THE TEMPLATE

When the pattern is identified, it must be somehow expressed as a template understandable by the program generator.

4.1 Extensible Stylesheet Language Transforms

Many tools and standards exist that add further value to XML. One of them is the XSL/T, which allows transformation of XML documents to other forms, such as other XML documents, HTML documents, or text files. Syntactically, XSL/T files themselves conform to the XML standard.
In particular, XSL/T can be used for describing templates that produce program code. We used this approach with great success for generation of user-interface integration classes (*Abeans plugs*, [7]) from XML-based descriptions of the devices.
However, it was a pain to change and debug XSL/T templates and the generated code, therefore we have decided to write a dedicated generator, which is optimized for generating source code and not just any text.

4.2 Extensible Program Generator Language

We have developed XPGL [8] because XSL/T is somewhat clumsy to work with. Most of this clumsiness stems from the fact that XSL/T is actually XML, which is by nature very strict, and that consequently makes XSL/T templates difficult to read, write and maintain.
Also, some XSL/T constructs that are otherwise infrequently used, become very common with program generators. These constructs are then very cumbersome to type over and over again.
Finally, XSL/T does not have some features expected from a program generator, such as an ability to preserve the code that the user has modified manually.
The design of XPGL strived towards the following goals:

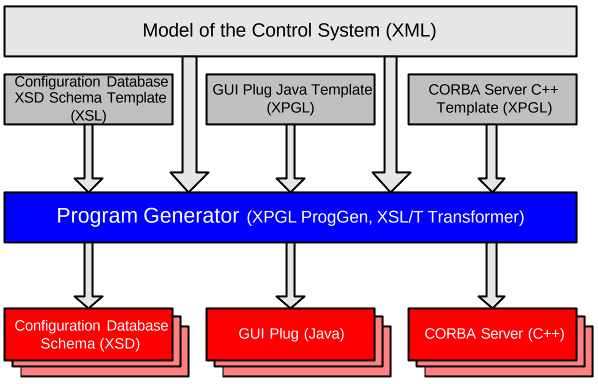
- Use as much of XSL/T and XML standards as possible to leverage existing technologies, tools and knowledge.
- Make frequently used constructs more compact and more readable.
- Pay special attention to indentation. The indentation of the generated source code should be visually very similar to indentation of the template. Also, it should be possible to generate source code comments that are visually appealing.

```
<?xpgl version="1.0"?>
class <"xpgl:naming('UU', /class/@name)"> {
  <for-each "/class/field">
    private <"@type"> <"xpgl:naming('LU', @name)">;
    public <"@type"> get<"xpgl:naming('UU', @name)">() {
      return <"xpgl:naming('LU', @name)">;
    }
    public void set<"xpgl:naming('UU', @name)">(<"@type"> value) {
      <"xpgl:naming('LU', @name)"> = value;
    }
  </for-each>
};
```

Figure 1: XPGL template for generating Java accessor and mutator methods with the associated private field that actually stores the value. XPGL tags are in bold. The values in quotes are XPath expressions [5]. The function `xpgl:naming` takes care of appropriate capitalization of a name (e.g., UU – first word upper, other words upper – produces `MyVariable`, whereas LU returns `myVariable`). The `for-each` XPGL tag instantiates its contents for every sub-element `field` of root element `class` in the input XML.

- There should be support for retaining the modification the user had made manually to the generated source.

To avoid going into too much details regarding the XPGL, the reader is invited to take a look at the example (section 6 below) and the XPGL specification [8].



Model of the Control System (XML)
Configuration Database XSD Schema Template (XSL)
GUI Plug Java Template (XPGL)
CORBA Server C++ Template (XPGL)
Program Generator (XPGL ProgGen, XSL/T Transformer)
Configuration Database Schema (XSD)
GUI Plug (Java)
CORBA Server (C++)

Figure 2: Illustration of the program generation process. Given the description of the software (e.g., the model of the control system) and the templates, the program generator produces the software artefacts (e.g., source files that are then subjected to compilation).

5 THE PROGRAM GENERATOR

If XSL/T is used as the language for the templates, the program generators are XSL/T transformation tools, which are readily available (for example, Xalan [9]). However, for XPGL templates, we had to develop a special transformation tool, which we call *ProgGen*. ProgGen is written in Java, and thus portable to most platforms. It was designed with maintainability and extensibility in mind. Thus, adding new building blocks to XPGL does not influence the rest of ProgGen.

6 AN XPGL EXAMPLE

As an example, let's take a look at how the template for accessor/mutator method pairs shown above looks like in XPGL (Figure 1). If the template was applied using the following XML:

```
<?xml version="1.0"?>
<class name="some_class">
  <field type="int" name="my_variable"/>
</class>
```

the program generator's output would exactly match the definition of class above. Adding an additional field is as simple as adding another `<field>` element to the XML: worrying about supplying the accessor and mutator is program generator's job.

7 CONCLUSION

Currently, XPGL language is fully specified, and ProgGen implements most of it. We are in the process of replacing our existing XSL/T code generation transformations with XPGL ones, especially due to their greater maintainability.

8 REFERENCES

[1] M. Plesko et al., "ACS – the Advanced Control System", PCaPAC 2002, Frascati, Oct 2002

[2] Cosylab Ltd., http://www.cosylab.com

[3] Sun Microsystems, The Java Programming Language, http://java.sun.com

[4] The World Wide Web Consortium, "Extensible Markup Language (XML) 1.0 (Second Edition)", October 2000, http://www.w3.org/XML

[5] The World Wide Web Consortium, "XML Path Language (XPath), Version 1.0", November 1999, http://www.w3.org/TR/xpath

[6] K. Zagar et al., "The Control System Modelling Language", ICALEPCS 2002, San Jose, CA, Nov. 2001

[7] G. Tkacik et al, "Java Beans of Accelerator Devices for Rapid Application Development", PCaPAC99 workshop, KEK, Tskukuba, January 1999

[8] Cosylab Ltd., "Extensible Program Generator Language", http://xpgl.cosylab.com/

[9] Apache, "Xalan XSL/T Transformation Library", http://xml.apache.org/