Code Generation from Configuration Specification Languages

For Program Execution Environment Configuration

Seminar aus Programmiersprachen

Martin Kaufleitner

Matr. Nr.: 1027229

MartinKaufleitner@gmx.at

May 6, 2016

1 Abstract

A big problem in software development is running the applications with the correct configuration. Many application errors arise, because the program is not executed in the developers intended way. The two main causes for this error are a different running environment or different program configurations. Specification languages can be used to create a formal, well-defined description of the applications expected or needed configuration, in order to specify the exact Program Execution Environment (PEE). PEE can be defined as *"those components that are used together with the application's code to make the complete system"*, this includes processors, networks, operating systems, call parameters, configuration files and so on. Additionally to the big advantage of making the application a lot less error prone to misconfiguration, a formal and complete description of an application is the ideal starting point for automatic code generation. In this paper we will have a look at common specification languages and how they can be used to automatically generate source code.

2 Introduction

When it comes to errors in computer programs, not always software bugs are responsible for not working applications. In many cases, the application runs perfectly in the developers intended environment, but fails when executed in a different one. This might also be the case, when a program is executed with unexpected parameters or invalid configuration. (Tianyin Xu, 2013) In general the setting, in which a program is executed is the so-called *Program Execution Environment (PEE)*. In Alan Burns (2001) the PEE is defined as *"those components that are used together with the applications code to make the complete system"*, this includes processors, networks, operating systems, call parameters, configuration files and so on. This variety of different components, which have to fit together in order be able to run the program properly can make the execution of software very error prone. A solution, which targets the problem of misconfiguration is the specification of the configuration of the program. In order to be able to create such a specification, specific *Configuration Specification Languages* are used. Such description languages allow to formulate the configuration, behavior and structure of the system. Using an exhausting formal description of the system which is planned to be developed comes with two major advantages. First the configuration is well defined, making the execution of the application less error prone, even on different target systems. In Barroso and Hlzle (2009) it is stated, that every second major service-level failure of Google's main services is caused by misconfiguration. Furthermore a recent study shows, that about 27% of customer support cases are caused by misconfiguration. Preventing this kind of errors using a complete configuration specification might help to save a lot of money and unpredictable problems already in advance.
Furthermore after having this complete and sound formal description, automatically generating the software for the final system is the obvious next step. This paper focuses on how code generation techniques can be used to generate program code from formal system configuration specifications.
Basically the paper is separated in four main parts: After the introduction, specification languages which are used to specify the systems configuration are explained, this will be done using two famous examples (SDL and XVCL). In the next section, two code generation techniques, which use exactly the previously mentioned specifications will be explained. Afterwards an example of the two approaches and evaluation on the usability for PEE will be given. In the last section, a conclusion of the paper and a short evaluation will be given.

3 Specification Languages

In this section we will have a look at specification languages and how they work. Two formal languages which are suited for code generation will be stated and explained in further detail. A comparison of more different architecture description languages can be found e.g. in Medvidovic and Taylor (2000).

3.1 Specification and Description Language (SDL)

A well known example of this specific type of formal languages is the Specification and Description language (SDL). The language is specified in UNION (1999) and is a formal language, which was originally invented to address the design and implementation of real-time, distributed and event driven systems in telecommunication. These are mainly applications, which consist of connections, adapter, signals and so on. The provided structures in SDL are focused on these kind of applications parts. Describing the system is separated in three main parts: **behavior**, **data** and **structure**. The behavior of a system can be described by extended finite state machines. The data description relies on data types and the structure can be formulated by a hierarchical decomposition UNION (1999). The main element in SDL is the finite state machine, which consist of the following tuple: $< S, I, U, d, t, s_0 >$. S is

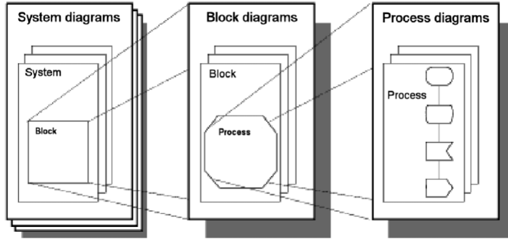

Figure 1: Structure of SDL Description (Blocks and Processes) Becucci et al. (2005)

a finite and non-empty set of states; I is the set of inputs and U the set of outputs; d defines a transition function $d : S \times I \rightarrow S$; t defines the output function $S \times I \rightarrow U$ and s_0 is the initial state. Basically the behavior description of a generic system is a composition of several finite state machines. The SDL description of the system is then a model of blocks and processes (Figure 1). A more detailed description about how modeling with SDL works can be found in the specification UNION (1999).

3.2 XML-based Variant Configuration Language (XVCL)

This meta-programming technique is available as open source software which was developed at the National University of Singapore. The goal of XVCL is not only to provide concepts of describing a system but already for the automatic generation of the according source code. The basic concept is the use of "composition with adaption" rules to compose program code from reusable, generic meta-components. The mentioned adaption of the components can be performed at specified variation points, defined by XVCL commands. One component is an XML file, which contains program code and different XVCL commands, provided as XML tags. The components are then composed in a hierarchical structure called x-framework. Basically higher level meta-components are build from lower level ones after possible adaptions. Jarzabek et al. (2003).

4 Code Generation

Now that we have seen a couple of specification languages, which can be used to describe the behavior and functionality as well as the specifications of a software system, we want to have a look at code generation. Nowadays when it comes to software development, the goal is often to provide a single code base, that can be used in different application scenarios by just adapting it. In this context Software Product Lines (SPLs) have to be mentioned. SPLs are used to *"create talor-made software products by managing and composing reusable assets"* Rosenmüller et al. (2008). This can be done either statically before runtime or dynamically during program loading or execution. Although many SPL frameworks force the developer to decide in advance between static or dynamic composition, there are approaches to support both e.g. (Rosenmüller et al., 2008). Well known implementation technique concepts are preprocessor definitions, components, collaboration-based designs, aspect-oriented programming (AOP), feature-oriented programming (FOP) and aspectual modules. Also frames, defined in the already mentioned XVCL (Section 3.2) can be used.

4.1 Code Generation Using Dynamic Frames

SPL helps to increase the software-making-productivity by developing in a manner like industrial production. Simpler parts are build together piece by piece. For this purpose smaller artifacts to compose the software are needed. In Radošević and Magdalenić (2011) code templates and application parameters are used to automatically generate code according to the source code generator's configuration. This approach is very close to XVCL and describes how code can be automatically generated for XVCL defined systems. Furthermore Specification-Configuration-Template (SCT) models are used to describe the code generator. This opens a number of opportunities regarding code generation:

- it is aspect based,
- uses code templates,
- enables graphical and textual representation and
- is independent from the program language.

Using SCT, the source code generator is basically described as a multi-level tree structure which can be composed bottom up. This means, that more complex generators are build by composing more basic generators from the sub-tree. Actually the SCT generator model was developed for the development of web applications, but generally there are no restrictions for the use of this approach. A very common application are for code generation are web applications. The inherit structure of web applications makes them primary candidates for code generation by composing the applications of smaller units since these applications usually already naturally consist of a number of small scripts. Another reason is, that for code generation, a well defined model (using an appropriate description

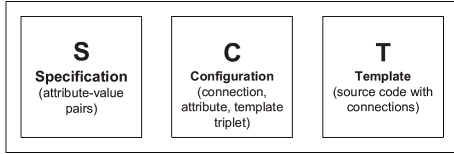

Figure 2: SCT frame Radošević and Magdalenić (2011)

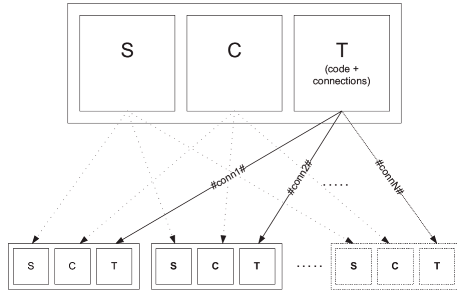

Figure 3: SCT tree Radošević and Magdalenić (2011)

language) is needed. This is mostly the case in web applications. Configurations share this property of comprehensive formal models and therefore the same concepts for code generation can also be used in this area. Therefore such dynamic frames code generation can be applied for generating code from configuration specification files.

The SCT generator model consists of three parts (Figure 2):

- **Specification (S)** specifies the features of the application and is basically a list of attribute-value pairs.
- **Configuration (C)** defines rules for connecting specifications with templates.
- **Template (T)** is a small piece of source code in the target programming language.

The templates also contain connections from the configuration which are some kind of placeholders, which can be replaced by other variable code parts (templates). In this way a tree of SCT frames can be generated and also graphically represented (Figure 3).

This can also be represented in a textual XML format. Figure 4 shows how the textual representation looks like. The specification part is a list of attribute-value pairs and the configuration defines, which templates are connected as shown in figure 5. In the template section connections to other frames can be formulated. The frames can be separated in different levels, where higher level, more complex

```
<sct>
  <specification>
    <s attribute="..." value="..." role="..."/>
    <s attribute="..." value="..." role="..."/>
  </specification>
  <configuration>
    <c connection="..." source="..." template="..."/>
    <c connection="..." source="..." template="..."/>
  </configuration>
  <template>
  . . .
  </template>
</sct>
```

application specification

generator configuration

code template

Figure 4: SCT frame in XML Radošević and Magdalenić (2011)

Level 1

```
<sct>
      <specification>
      . . .
      </specification>
      <configuration>
      . . .
      </configuration>
      <template>
      . . .
          #conn1#
      . . .
          #conn2#
      . . .
      </template>
</sct>
```

Level 2

```
<sct>
    <specification>
    . . .
    </specification>
    <configuration>
    . . .
    </configuration>
    <template>
    . . .
    </template>
</sct>
<sct>
    <specification>
    . . .
    </specification>
    <configuration>
    . . .
    </configuration>
    <template>
    . . .
    </template>
</sct>
```

Figure 5: SCT frame connections using XML Radošević and Magdalenić (2011)

frames are composed from lower ones. A more detailed description of the approach can be found in Radošević and Magdalenić (2011).

4.2 Code Generation Using Declarative Mappings

Declarative Mappings can be used, to automatically generate code from specifications made with the already mentioned description language SDL. The idea of the approach proposed in Mansurov and Ragozin (1999) is, that automatic code generation is a mapping from a formal specification language, called the source language (like SDL) onto an imperative target language (like C++). The mapping than basically has to define, how a construct from the source language can be represented in the target language. The traditional way of doing this are direct mappings, following this scheme:

- Executable constructs of the source language, which have equivalents in the target language like loops or conditional operators are represented directly by those equivalents.
- If there is no direct executable equivalent, the

construct is modeled by a group of target structural and executable constructs.

- Structural constructs of the source language, which have equivalents in the target language like objects, procedures or packages are represented directly by those equivalents.
- If there is no direct structural equivalent, the construct is either transformed during translation or again modeled by a combination of executable and structural constructs of the target language.

The key idea of declarative or indirect mappings is to use so-called executable statements as a uniform representation for semantically distant constructs. This executables are used to build the internal run-time structures for the specified system. It is assumed, that the execution of the generated program can be separated in a start-up and a run-time part. During start-up the generated executables are performed to build the internal representation of the system. During run-time this representation can be used by the support system for executing the program. Therefore generated code from declarative mappings consists of parts which are only performed once at start-up and code for executable statements of the source language. In Mansurov and Ragozin (1999) it is stated, that generated code from declarative mappings needs less so-called glue code. Glue code is needed, to "stick" the separately generated code parts together to make them run. This is achieved by using so-called executable constructs instead of simply translating one part of the origin language to one of the source language. Those executable constructs satisfy syntactic and semantic constraints of the target language and therefore don't need additional code to make it run.

5 Evaluation

We have seen two different kinds of automatically creating source code from description languages, as well as two different description languages. In this section we want to evaluate the explained concept in the sense of usability for PEE configuration specification. Until now, the approaches and technologies were explained rather general. We want to have a look, how they can be specifically adapted to the context of Program Execution Environment. For this purpose, XVCL will be explained in combination with dynamic frames and the SDL description language together with dynamic mappings. The detailed description of the code generation is very complex and would exceed this paper's specace, it can be found in the according papers Radošević and Magdalenić (2011) for dynamic frames and Mansurov and Ragozin (1999) for declarative mappings. Nevertheless we want to have a look at a possible implementations to get an idea how the technologies could be used for PEE.

5.1 XVCL and Dynamic Frames

The approach of using XVCL frames clearly has its advantages when it comes to web applications. Since a lot of code snippets are already available, the implementation of the code generator should be quite straight forward. As already mentioned, the availability of good models of this kind of applications comes very handy additionally. We want to see now how this can be used for PEE configuration.
Imagine a webapplication which consists of different parts of the website. We use the SCT model to specify the websites structure and content, this is the typical content of a PEE specification configuration. Using this SCT model, the proposed code generator from Radošević and Magdalenić (2011) is able to produce the applications code. We will not explain all part, for a whole description and the complete SCT model, please have a look at (Radošević and Magdalenić, 2011).

5.1.1 Specification

First we want to describe the outputs of the webapplications:

```
<s attribute="OUTPUT"
 value="index"/>
<s attribute="OUTPUT"
 value="output_cgi"/>
 <s attribute="OUTPUT"
 value="output_html"/>
<s attribute="OUTPUT"
 value="questionnaire"/>
```

The index page is then defined as

```
<s attribute="index"
 value="output/index.html"/>
<s attribute="applications"
 value="Database Content Management/>
```

Later as specifiedin the *Configuration* "index" will replace the according parts in the *Template* section. The same is done with "application".

5.1.2 Configuration

Next we have to define the connections between the specification of the application and the actual code templates. This is done in the Configuration part of the SCT model.

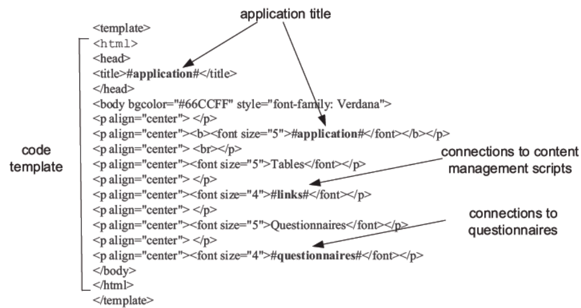

Figure 6: Example code template Radošević and Magdalenić (2011)

```
<c connection="#1"
  template="index.template"/>
<c connection="#2"
  template="script.template"/>
<c connection="#3"
  template="form.template"/>
<c connection="#4"
  template="questionnaire.template"/>
```

Now we see, that for each specification entry a new connection is created and the according template is linked.

5.1.3 Template

The actual code templates are not part of the later PEE configuration specification. They are just common re-occurring code fragments of webapplications which have to be put together. In the essential parts where applications specific things are defined, we find references to the *Specification* using *Configuration*. The template is shown in Figure 6.

5.2 SDL and Dynamic Mappings

As already mentioned, SDL comes from the telecommunication industry and provides therefore according structures, which can be used for other types of applications as well. Imagine, the application has some kind of connections between different endpoints and also the signal route and the processes output is defined. Again, these definitions are typical things that should be stated in the configuration specification, since they describe the environment including the datatypes. An example SDL specification could look like the following Mansurov and Ragozin (1999)

```
SYSTEM TYPE S:
/* BLOCK B1 : B1TYPE */
CHANNEL C1 from B2 to B1 via G1
  with S1, S2;

BLOCK B2:
SIGNALROUTE R from P to env
  with S1, S2;

BLOCK B2:
CONNECT C1 and R;

PROCESS P:
OUTPUT S1 via C1;
```

Now lets have a look at the according $C++$ code parts, which were produced using declarative mapping:

```
sytS::sytS:
chnCl = new SDLChannel("Chl ",
  iblkB2,((bltB2TYPE*)iblkB2)
   -> gatNoGate,
  iblkB1, ((bltBITYPE*)iblkB1)
   -> gatG1,
  newSDLSignallist(sigSl::ld,
                   sigSig2::ld,0));

BltB2::bltB2:
SgrR=new SDLSignalroute("R",
  iprcP, ((prcP *)iprcP)
   -> gatNoGate,
  iprcENV, ((prcENV*)iprcENV)
   -> gatNoGate,
  newSDLSignallist(sigSl::ld,
                   sigSig2::ld, 0));

BitB2::bltB2:
SgrR -> Connect(&(((sytS*)SystemAddr)
 ->chnC1));

CurrentGraph->Output(new sigSl(Self()),
  ((sytS*)SystemAddr)->chnC1);
```

One can see, that after connections and datatypes are defined, the basic structure of this kind of applications is basically fixed and therefore, a mapping from specifications structures to target source code can be done quite straight forward. As already mentioned, this kind of PEE specification and code generation should be used, if the applications consists of parts, which can be represented with SDL structures, otherwise it doesn't make any sense.

5.2.1 Comparison

The two described approaches are quite different. Among being the most popular ones, this was one of the reasons for choosing them as an example in this paper.
If an application can be more or less easily separated in smaller code parts, which maybe even already exist, the SCT model provides a powerful tool to specify the applications configuration and automatic code generation support.

If the applications is not that easily separable, but contains a lot of structures which can be found in the very comprehensive SDL specifications, than the declarative mapping approach might be the better choice.

6 Conclusion

We have seen two main approaches to use a programs configuration specification for automatic application code generation. The first one uses a SCT model for defining the applications structure, providing already existing code templates. The second one maps small parts of the specification to according source code parts.

One problem using this techniques might be, that thinking about the whole applications structure in advance might be not so easy, since details can easily be forgotten. In my opinion here comes the big advantage of this type of code generation, since it is very flexible. Adding, changing or removing some of the code snippets is an easy way to change the applications behavior. This is way harder in SDL, where the behavior is modeled with complex structures. On the other hand, there might be applications which are very process oriented, where using XVCL would end up in the need of writing a lot of so-called glue code to make the application run. In this case, the SDL specification language, which provides structures for modeling processes and communication and using a declarative mapping afterwards might suit better. Therefore one can not just decide on one specification language and code generation approach. It is very important to have a close look at the individual application in order to be able to decide on the best fitting approach.

There are additional problems one might have to deal with in order to be able to implement this approaches. A lot of effort has to be put in the proper creation of the configuration specification files. For example in XVCL, the code generator needs all code snippets before it is able to generate parts of an application or even the whole program. Depending on the application, many of them might be re-used from other applications or are very generic, but others might be more complex. Another problem might also be to convince developer in spending most of their time in formulating the system with an relatively abstract formal modeling language rather than programming it right away.

More work has to be done in the flexibility of the applications description using specification languages and still providing the automatic code generation. This is always some kind of trade-off, since a weaker model makes the code generation harder.

References

Alan Burns, A. W. (2001). *Real-Time Systems and Programming Languages*. Pearson Education Limited, England.

Barroso, L. A. and Hlzle, U. (2009). *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines.*

Becucci, M., Fantechi, A., Giromini, M., and Spinicci, E. (2005). A comparison between handwritten and automatic generation of c code and sdl using static analysis; softw.-pract. exp. *Software-Practice & Experience, 2005 Nov 25, Vol.35(14), pp.1317-1347.*

Hervieu, A., Baudry, B., and Gotlieb, A. (2012). *Testing Software and Systems: 24th IFIP WG 6.1 International Conference, ICTSS 2012, Aalborg, Denmark, November 19-21, 2012. Proceedings*, chapter Managing Execution Environment Variability during Software Testing: An Industrial Experience, pages 24–38. Springer Berlin Heidelberg, Berlin, Heidelberg.

Jarzabek, S., Bassett, P., Hongyu Zhang, P., and Weishan Zhang, P. (2003). Xvcl: Xml-based variant configuration language.

Mansurov, N. and Ragozin, A. (1999). Using declarative mappings for automatic code generation from {SDL} and asn.1. In Lahav, R. D. v. B., editor, *{SDL} '99*, pages 275 – 290. Elsevier Science B.V., Amsterdam.

Medvidovic, N. and Taylor, R. (2000). A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering, Jan. 2000, Vol.26(1), pp.70-93.*

Radošević, D. and Magdalenić, I. (2011). Source code generator based on dynamic frames. *Journal of Information and Organizational Sciences, 2011.*

Rosenmüller, M., Siegmund, N., Saake, G., and Apel, S. (2008). Code generation to support static and dynamic composition of software product lines. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, GPCE '08, pages 3–12, New York, NY, USA. ACM.

Tianyin Xu, Jiaqi Zhang, P. H. J. Z. T. S. D. Y. Y. Z. S. P. (2013). Do not blame users for miscongurations. *University of California, San Diego.*

UNION, I. T. (1999). Specification and description language (sdl).