A Common Metamodel for Code Generation

Michael PIEFEL

Institut für Informatik, Humboldt-Universität zu Berlin
Unter den Linden 6, 10099 Berlin, Germany
piefel@informatik.hu-berlin.de

ABSTRACT

Models can be used in many stages of many different processes, but in software engineering, the ultimate purpose of modelling is often code generation. While code can be generated from any model, we propose to use an intermediate model that is tailored to code generation instead. In order to be able to easily support different target languages, this model should be general enough; in order to support the whole process, the model has to contain behavioural as well as structural aspects. This paper introduces such a model and the ideas behind it.

Keywords: Metamodel, code generation, MOF, Java, C++

1. INTRODUCTION

Models, in general, do not relate to programs at all. However, in software engineering models do refer to systems and their components, which may be executable code. Ideally, model-driven development (such as embodied in the OMG's Model Driven Architecture (MDA) [9]) eventually leads to code generation.

Our research group has been involved in simulation and modelling for a long time. We developed SITE [17], a compiler and runtime environment for the Specification and Description Language of the ITU-T, SDL [5]. This compiler uses conventional techniques of a hidden representation of the abstract syntax tree and code generation (to C++).

Lately, we proposed an open framework for integrated tools for ITU-T languages that is provisionarily called ULF-Ware [4]. An overview of its architecture can be seen in figure 1 on the following page.

Oversimplifying, ULF-Ware contains a model-based compiler for SDL. The input (in this case, a textual SDL specification) is parsed and a *model* in the SDL repository is generated from it that adheres to the *SDL metamodel*. The next step transforms this to a new model in the Java/C++ repository adhering to the *Java/C++ metamodel*. Finally, code generators turn this model into C++ or Java.

To make this work smoothly, a run-time library is needed to support high-level concepts of the source language, such as signal routing. For a description of how to interface with such a library, see [14].

This paper deals with the Java/C++ metamodel, a metamodel that is applicable to both Java and C++ and that comprises structural as well as behavioural aspects. The requirements for such a metamodel which is geared towards code generation are not obvious. Many decisions may prove to be problematic further on. The metamodel presented here is far from finished.

Similar metamodels exist, each with strengths and weaknesses. At the end of the next section, we will give a short overview of them. Mostly, they are concerned with structural aspects only. There is, however, a need for a new metamodel to cover all aspects of a programming language.

Section 2 will present general choices that had to be made in order to determine the shape of the metamodel. Section 3 presents the metamodel in detail.

2. DESIGN CONSIDERATIONS FOR CEEJAY

The code-generation metamodel is conceived to be useful to generate C++ as well as Java from it. Therefore we decided to call it *CeeJay*.

The meta-metamodel

Metamodels are not a recent development, but rather a new terminology and a different focus. Conventional language definitions are frequently presented in BNF and derivates. There are obvious correspondances between program and model, programming language and metamodel, "programming language definition language" and meta-metamodel. It is advantageous to employ the same meta-metamodel for all steps – this is a novel point of view: A conventional compiler has BNF for the parser, but builds its abstract syntax tree using other means such as Kimwitu++ [13, 7] or in an ad-hoc fashion.

We have chosen to use MOF as the meta-metamodel. MOF is the Meta Object Facility of the OMG [8]. It is used in many OMG standards, most prominently as the meta-metamodel for the UML [12]. MOF is closely tied to UML, in fact there is a number of packages called the UML Infrastructure [10] that are shared between MOF and UML.

MOF, however, is more than just a meta-metamodel. As the name suggests, it provides a metadata management framework. There are a number of mappings from MOF to different platforms, such as a mapping to XML and a mapping to Java. While the former allows model interchange via files, the latter gives interfaces to create, navigate and modify models.

Using MOF and an appropriate tool for it gives a standard way to access models. First, you define a metamodel based on the MOF meta-metamodel. The tool then generates interfaces for models of this metamodel and an implementation to store the models. There are a number of tools that work this way, but the only one adhering to the new MOF 2.0 standard is "A MOF 2.0 for Java" [16].

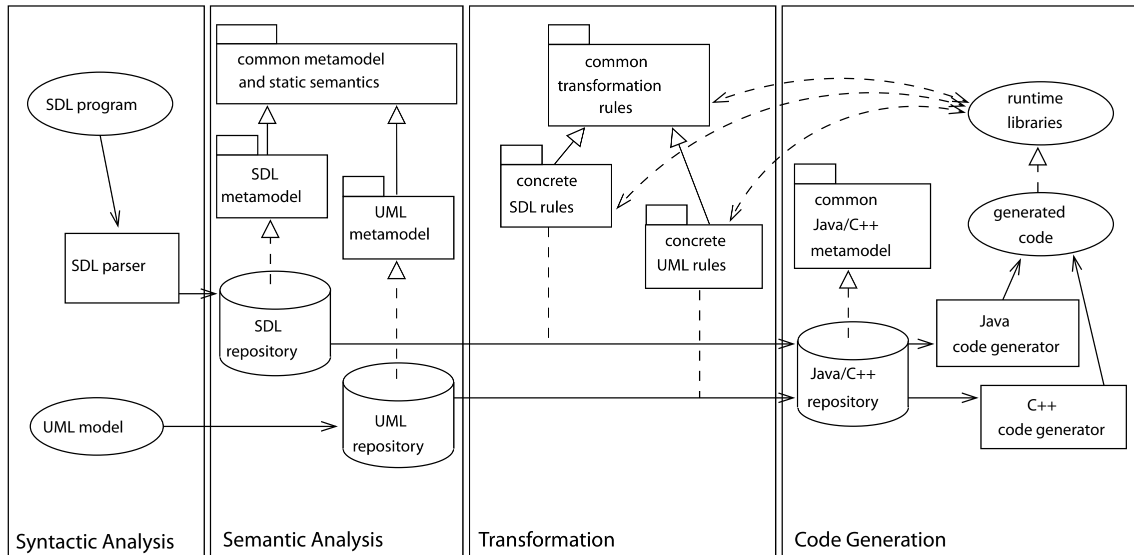

Figure 1: ULF-Ware overview

Generic or specifc

High-level models are quite different from programming languages. They abstract from most of the detail that a programming language exhibits. Once you want to generate real code, all this detail has to be filled in. This makes code generation from those models a difficult task. Moreover, many decisions in this process are similar for different target languages, but it is hard to make use of these commonalities. Finally, the way back, i. e. from program text to high-level models, is very hard. Note that most tools that promise to do this (e. g. reverse engineering of Java to UML) only capture the structural aspects of the language (i. e. they only produce UML class diagrams).

The reverse approach is to use models that are very low-level and close to a specific language. There have been a number of papers such as [1] implementing this. The metamodel obtained this way is close to the original BNF of the language. It is arguable whether such metamodels are desirable; there are several criteria regarding the quality of metamodels. In any case models like this are difficult to obtain. They would be the result of a model transformation from a high-level model. Here, the intelligence would have to lie in the transformations.

Thus the level of detail of the metamodel of a programming language determines whether there will be more work to do in the code generator or the model transformator. We have chosen a level of abstraction that allows true object-oriented models (as opposed to models closely relating to the syntax of a language) while still being close enough to programming to make code generation a straightforward process.

We will add another criterion to the decision as to how close to the target language the model should be: Can we use *one* metamodel for *many* languages?

Commonalities of object-oriented programming languages

Many languages share common concepts, as has been shown and made use of in [15], such as the quite abstract concept of namespace. For programming languages, the similarities go even further.

Many differences in those languages are purely syntactical or for simple static semantics, such as the declaration of variables before use. The most important differences are support for crash-avoidance (which is irrelevant in a theoretical context) and the extent of the available libraries, neither of which affect the metamodel.

Java and C++ in particular are very similar to each other. Still, a complete metamodel would exhibit a number of fine differences such as visibility and the (non-)existence of multiple inheritance. However, we want to use Java and C++ as output languages only. This allows us to build a metamodel that can represent only the intersection of features from Java and C++.

Since Java and C++ have so much in common, the combined metamodel is still expressive enough to allow arbitrarily complex models. In fact, other object-oriented languages share the same concepts in very similar ways; there are always classes, there are variables and loops.

Some of the differences between languages are evened out because we want to use the models only for code generation. In Python, for instance, variables do not have a type, or rather, the type of variables and parameters remains undeclared. While generating code from a model containing type information, it is easy to just suppress generation of type names. Assuming the model was correct in the first place, Python's interpreter will infer just the right types.

Related Work

There are a number of metamodels for different languages around. However, the public availability of these metamodels is limited. Further, the focus of the metamodels can be quite different, as mentioned above.

There is a project called jnome [2]. The metamodel developed therein is tailored to generating documentation for Java files. It lacks support for the implementation of methods and is as such not suitable for complete code generation. The metamodel is available as a set of Java interfaces.

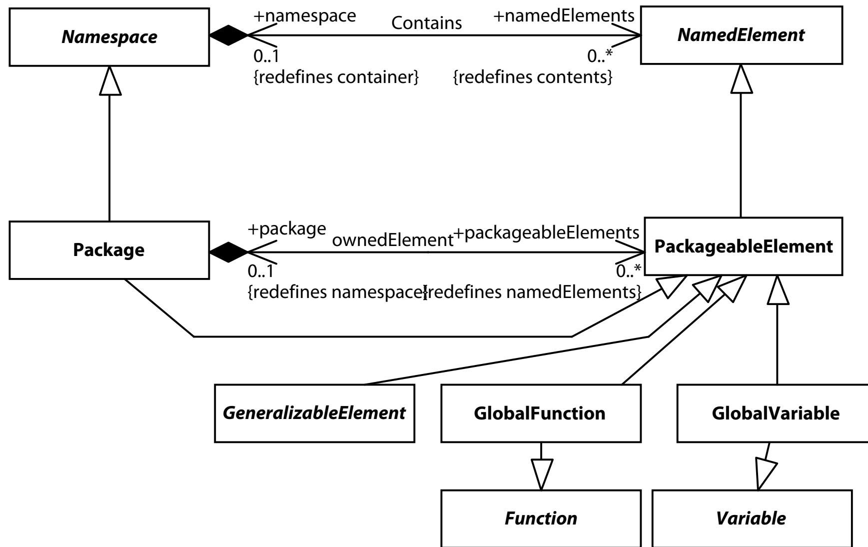

Figure 2: Package diagram

Both the Netbeans IDE and Eclipse seem to use a Java metamodel internally. Both IDEs bring their own repository functionality: MDR [6], a MOF 1.4 repository, and EMF [3], a repository and metamodel that is similar to MOF. Both metamodels are not MOF 2.0 metamodels.

The Object Management Group has a specification containing a Java metamodel [11]. It only contains structural aspects and is based on MOF 1.3. This metamodel seems to have been abandoned in an early stage of development.

Numerous other works are concerned with automatically generating metamodels from grammars. The results are usually close to the grammar and, naturally, specific to the language they are based on. They are not suitable for a more general approach.

A metamodel that captures some common features of Java and Smalltalk is presented in [18]. This metamodel is concerned with common refactoring operations in those two languages.

3. THE CEEJAY METAMODEL

Common metamodelling elements

As has been outlined in [4], we use a package with common metamodelling building blocks for all our metamodelling activities. This makes it easier to speak about common concepts under the same name, much as Design Patterns in software engineering help programmers to talk about common programming concepts.

Figure 3 shows a minimal extract of common elements; more concepts are introduced below. Those concepts are well known. We have chosen names that are similar (or equal) to name of corresponding concepts in the UML Infrastructure [10].

At the root of the inheritance tree there is the abstract ModelElement, and each element in any model will be an instance of a subclass of it. NamedElement adds the ability for a model element to hold a name. Similar elements, each introducing a single new attribute, can be found further down. Also, there is the abstract concept of containment. In the diagram you can see a redefinition of this concept: Namespaces only contain named elements.

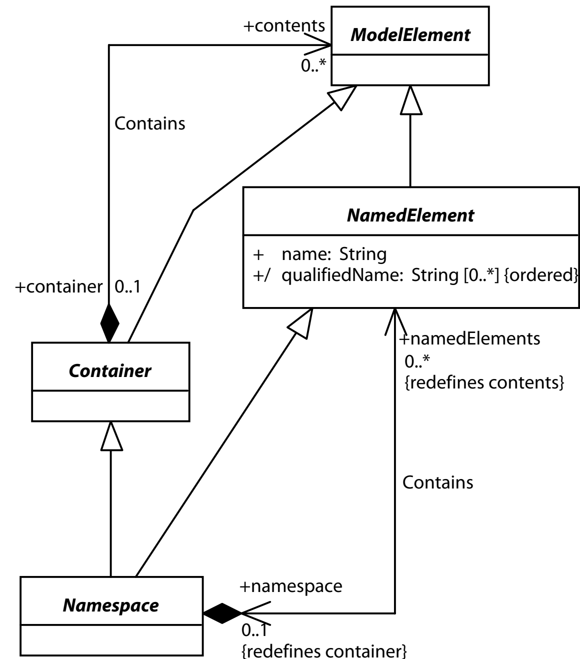

Figure 3: Common diagram

In fact, the basic building blocks can be viewed as a stripped-down version of the Infrastructure. It remains to be seen whether it is advantageous to have a seperate representation which will aid in understanding the concepts because it is simple, or whether we should rather use the Infrastructure itself.

Packages

Figure 2 shows the representation of packages in CeeJay.

The *package* is an important structural concept of Java. In C++, there is the concept *namespace*. The two concepts are equivalent: They form a space where elements such as classes can reside in

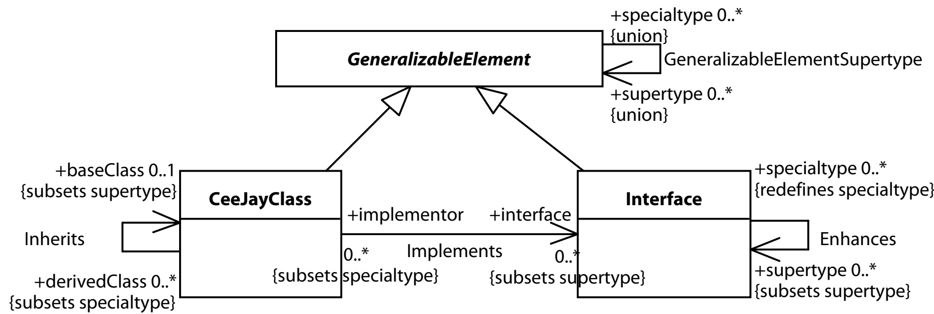

Figure 4: Class diagram

without colliding with elements of the same name in other packages.

Theit usage is slightly different: Java packages have to reside in respectively named directories and they can be imported in one step as a whole. C++ namespaces, however, can be arbitrarily distributed into compilation units, and the inclusion of a header file declares only a specific subset.

These differences, however, are of no concern to the metamodel of the languages. While generating code, it is easy to take care that every generated C++ namespace has an associated header file declaring everything, such that the inclusion of it is equivalent to the import of a package in Java. The distribution of components into files or directories is an implementation artifact.

Consequently, a single metaclass Package suffices for both Java packages and C++ namespaces. It is a specialization of Namespace that is limited to contain only PackageableElements. Each element that can be contained directly within a package inherits from the latter. Most notably, Package itself; this allows nesting of packages.

Note that there is no constraint in the metamodel limiting classes (or other model elements, more specifically, other NamedElements) to be contained in packages. This conforms to both Java and C++, where classes are allowed to exist outside of packages.[1]

Class

Figure 4 shows the class diagram defining CeeJayClass and Interface.

Java does not allow multiple inheritance. The least common denominator here is to allow only single inheritance in CeeJay as well. Java does, however, have interfaces, that are also generalizable elements. A class may implement any number of interfaces. An interface may extend any number of existing interfaces.

While C++ does not have interfaces, they can be represented by abstract classes.[2] Thus, the metamodel is applicable to C++ as well.

A language without multiple inheritance and without interfaces may be difficult to map into from CeeJay models.

Types

The Types diagram in figure 5 on the following page shows the different types of CeeJay and their usage: They are referenced by a TypedElement. There are three main groups of types: Primitive types, classes and collections.

Primitive types are the predefined types of Java and C++. As a start, only integers and booleans are used. The third primitive type here is void, which is used in functions (see 3) as the return value. This type may be problematic and it may disappear in a future version of the metamodel. The alternative is to have a typed element that does not, in actuality, have a type. This is also allowed in this metamodel; this is how it is done in UML.

There is a constraint on the primitive types that is not expressed formally: There may be only one instance of the metamodel element PrimitiveType for each PrimitiveTypeKind, and there always has to be exactly one. This means that there will have to be a number of model elements in CeeJay models that are always supposed to be there, just as the package Predefined in SDL [5].

The second variety of types is GeneralizableElement. In both Java and C++, each class is also a type, and interfaces as well.

Collections are mentioned explicitly in this metamodel. This is not strictly necessary, since they are normal classes in most languages. However, in order to use the built-in collections of C++ or Java although they have different names (and different semantics and usage), there has to be a specific representation. A Collection has the usual attributes for order and uniqueness. It has an association with a type; this is the type of the elements of the collection.

Functions and variables

As can be seen in figure 5 on the following page, both variables and functions are model elements that have a name and a type. Additionally, functions also have parameters (they extend ParameterizedElement, see on the next page) and a body.

It is not yet completely clear how to represent the body – is it a nested namespace containing, among other things, the local variables, or is it only an ordered sequence of statements? Also, the requirement that each function needs a type leads to the need for a type void, which has to be explicitly disallowed for variables. The respective constraint is not shown here.

Variables are adorned with an attribute isConstant. This covers both Java and C++, the languages we are mainly concerned with,

[1] Rather, they are assumed to exist within the default package or the default namespace.

[2] Rather, a class without any method implementations: An interface in Java does not need to contain any method declarations, it can be empty or contain constants only. Such a class would not be abstract in C++, as an abstract class is a class with an abstract method.

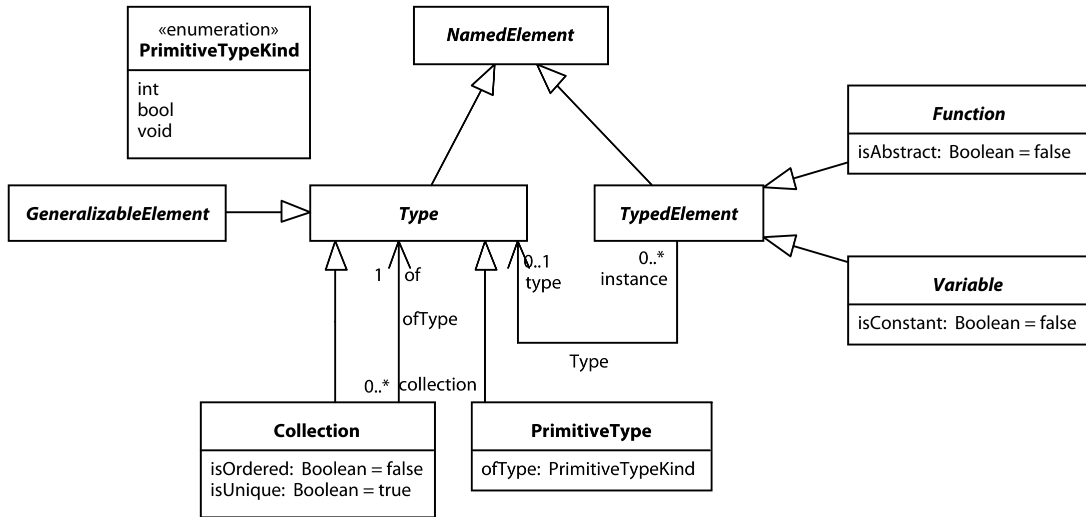

Figure 5: Types diagram

but not Python. Of course, when generating Python, the isConstant specifier could just be ignored, and assuming the model was correct in the first place (i. e. the constant is not modified anywhere) the semantics are not changed. On the other hand, constants need initial values, and those can imply an order for the variables, which is not desired.

Note that there is no constraint in the metamodel limiting functions and variables to be contained in classes. In the package diagram (figure 2), there are explicit metaclasses GlobalFunction and GlobalVariable.

This violates Java rules, where there are no top-level functions and variables. They are, however, allowed in C++ (and other languages that are often derogatively called multi-paradigm languages). Moreover, they are even needed in C++ to model the entrance point of the program (the main function). The alternative is to have a special metamodel element for the entry point, which might even be preferable, as the parameters of this main function and the Main method are determined by the target language anyway and not subject to modelling.

Often, it is not necessary for functions to live within a class, as they do not alter or access the state of an object or a class. An example of this are functions in the mathematical meaning, having no state at all, for instance the trigonometric functions. Java does not allow this, but it can easily be simulated by wrapping those functions in a class and making them static, as is done in Java's java.lang.Math. A Namespace that contains *only* functions and variables, but no classes, can be mapped to a Java class of the same name instead of to a package.

Parameters

Some language constructs require a list of nameless parameters, for instance signals in SDL. In the UML, the name of a NamedElement is in fact optional. Instead of relying on this, the CeeJay metamodel contains a metaclass SignaturedElement which inherits from NamedElement and Container. In Java and C++, we use parameters for functions, where they always have an associated name. Therefore CeeJay's ParameterizedElement additionally inherits from Namespace, as can be seen in figure 6.

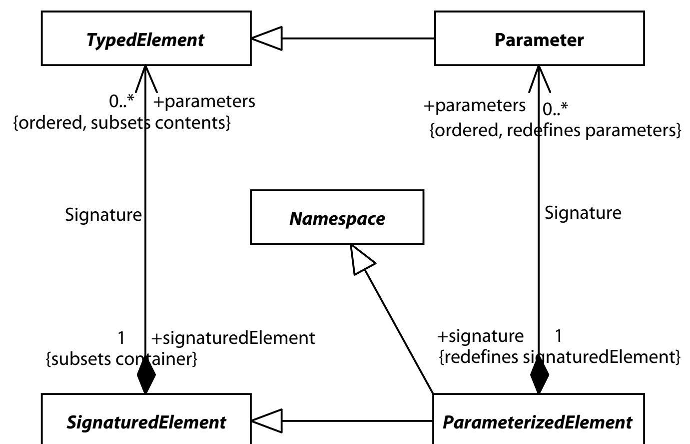

Figure 6: Parameters diagram

Members

The member diagram in figure 7 on the next page shows the members of Java and C++ classes.

Members have a visibility, which can be either public, protected or private. Java allows a fourth visibility of package visibility that has no direct equivalent in C++. Note that there are differences between *visibility* and *accessibility*. This is not reflected in the CeeJay metamodel.

A MemberFunction is a Function that is also a Member, and a MemberVariable is a Variable that is also a Member.

Any GeneralizableElement, which is a Namespace, can contain members, and nothing else. A member can only be contained within a GeneralizableElement. Therefore, the composition ClassMembers redefines both its association ends. The multiplicity has changed, as a member can be contained in only one class (and must be contained). A member has access to the class it is in, therefore the navigability of the composition has been changed to reflect this as well.

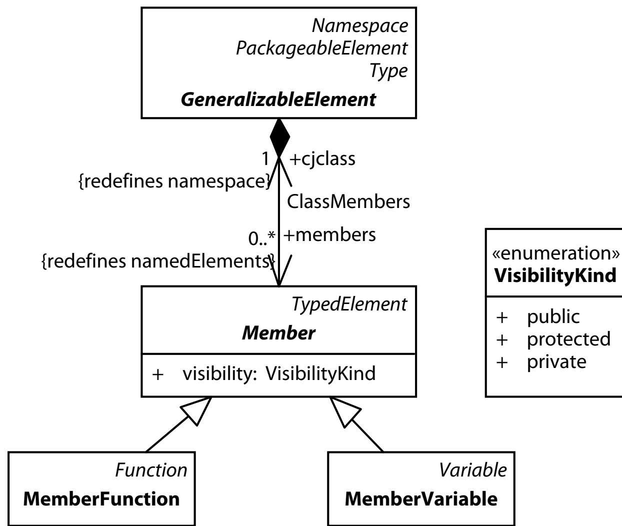

Figure 7: Member diagram

4. CONCLUSION

The last step in a complete model driven software engineering process is the generation of implementation artifacts, usually in the form of source code. While code can be generated directly from high-level models such as UML, this puts too much work on the code generator. Instead, a stepwise refinement of the model into a model geared towards code generation is preferable.

To this end we have prepared a metamodel that is reasonably close to the target languages Java and C++, while still being general enough to not only cover these two languages, but other object-oriented languages as well.

This is different from existing metamodels that have been published (mainly for Java), which are close to the grammar of Java. Thus, they can often hardly be called metamodels, as they are no more than a MOF representation of the abstract grammar.

The CeeJay metamodel will be used in a framework where C++ is generated from SDL specifications. The aim of this open framework is to extend it for other output languages, such as Java or Python, and other input languages, such as UML or domain specific languages.

The metamodel presented in this paper is not finished yet, it is work in progress. Different approaches for the behavioural aspects have been considered, but they are not yet in publishable form. We lack experience of how well the code generation will work for larger projects; for the time being, we only have generated code from small models.

5. REFERENCES

[1] M. Alanen and I. Porres. A relation between context-free grammars and meta object facility metamodels. Tucs technical report no 606, Turku Centre for Computer Science, 2003.

[2] J. Dockx *et al.* jnome: A Java meta model in detail. Report cw 323, Department of Computer Science KULeuven, 2001.

[3] Eclipse Project. *Eclipse Modeling Framework*, 2006. URL http://www.eclipse.org/emf/. Last checked: February 27, 2006.

[4] J. Fischer *et al.* ULF-Ware – an open framework for integrated tools for ITU-T languages. In A. Prinz *et al.*, eds., *SDL 2005: Model Driven: 12th International SDL Forum*, vol. 3530 of *Lecture Notes in Computer Science*, p. 1. Springer-Verlag GmbH, 2005.

[5] ITU-T. *ITU-T Recommendation Z.100: Specification and Description Language (SDL)*. International Telecommunication Union, 2002.

[6] Netbeans. *Metadata Repository (MDR)*, 2006. URL http://mdr.netbeans.org/. Last checked: February 27, 2006.

[7] T. Neumann and M. Piefel. *Kimwitu++ – A Term Processor*, 2002. URL http://www.informatik.hu-berlin.de/~tneumann/manual.html.

[8] OMG. *Meta Object Facility (MOF) 2.0 Core Specification*. Object Management Group, 2003. ptc/03-10-04.

[9] OMG. *Model Driven Architecture Guide, Version 1.0.1*. Object Management Group, 2003. omg/03-06-01.

[10] OMG. *UML 2.0 Infrastructure Specification*. Object Management Group, 2003. ptc/03-09-15.

[11] OMG. *Metamodel and UML Profile for Java and EJB Specification*. Object Management Group, 2004. formal/04-02-02.

[12] OMG. *UML 2.0 Superstructure Specification*. Object Management Group, 2004. ptc/04-10-02.

[13] M. Piefel and M. von Löwis. The term processor Kimwitu++. In N. Callaos *et al.*, eds., *Proceedings of SCI 2002*, vol. V, pp. 182–186. I I I S, Orlando, USA, 2002.

[14] M. Piefel and T. Neumann. A code generation metamodel for ULF-Ware: Generating code for SDL and interfacing with the runtime library. In *System Analysis and Modeling: 5th International SDL and MSC Workshop*, Lecture Notes in Computer Science. Springer-Verlag GmbH, 2006.

[15] M. Scheidgen. *Metamodelle für Sprachen mit formaler Syntaxdefinition, am Beispiel von SDL-2000*. Dissertation, Humboldt-Universität zu Berlin, 2004.

[16] M. Scheidgen. *A MOF 2.0 for Java*, 2005. URL http://www.informatik.hu-berlin.de/sam/meta-tools/aMOF2.0forJava. Last checked: February 8, 2006.

[17] R. Schröder *et al.* *SDL Integrated Tool Environment*, 1997. URL http://www.informatik.hu-berlin.de/SITE/. Last checked: February 27, 2006.

[18] S. Tichelaar *et al.* A meta-model for language-independent refactoring. In *Proceedings ISPSE 2000*, pp. 157–167. IEEE, 2000.