ULF-Ware – An Open Framework for Integrated Tools for ITU-T Languages

Joachim Fischer, Andreas Kunert, Michael Piefel, and Markus Scheidgen

Humboldt-Universität zu Berlin
Institut für Informatik
Unter den Linden 6
10099 Berlin, Germany
`{fischer|kunert|piefel|scheidge}@informatik.hu-berlin.de`

Abstract. Model driven engineering is a popular attempt to deal with the complexity of modern software systems. For the telecommunication sector a model driven approach means that you have to handle half a dozen ITU-T modelling languages in a single process to cover all aspects of telecommunication system development. Unfortunately, this is a difficult task, because the ITU-T languages are hard to use together. That is why the ITU-T started the Unified Language Family (ULF) initiative with the goal to unify the ITU-T language definitions and allow an easier alignment and integrated use of these languages.
We present a tooling framework for those ULF languages; it is called ULF-ware. Our framework uses metamodelling and a shared use of common language concepts for a tight language integration. Around these language models it incorporates a set of tools to cover the various responsibilities of development environments such as program parsing, model checking, model transformation, and code generation.
This paper shows work in progress. We demonstrate our ideas on a tool chain for a subset of SDL. But the overall goal is an open framework that is extendable with other languages, even beyond ULF, and with tools for other software engineering tasks, like model simulation or software deployment.

1 Introduction

Over the past decades the ITU-T developed a series of modelling languages, each to cover a special aspect of telecommunication system specification. These languages are by name: *eODL*, used for high-level component description; *SDL* [1] and *MSC* define different approaches of behaviour modelling; *ASN.1* is used to define data; and *TTCN* allows to write test cases. So you virtually have a modelling technique for every need, but in reality this is meaningless when you can not use your languages together.

Different methodologies used in language development and definition make it hard to align and relate languages with each other, and thus integration is not trivial – it is barely possible. Of course this is no news, and various calls have been made. The ITU-T proposed the idea of an *Unified Language Family* (ULF), a consistent, uniform foundation for all ITU-T languages. But which is the method of choice to accomplish the described task?

Two rivals have emerged: the field-tested and well-founded context free grammars, versus the new (incarnation of an old idea) metamodelling that has proven itself by building the base for ULF's antagonist UML [2]. Omitting all political arguments, metamodelling is the more promising, and therefore scientifically more interesting approach. This paper proposes an approach to language tool development that uses metamodelling; it is named after its overall goal: ULF-Ware.

ULF-Ware is about utilizing metamodelling potential for faster tool and language development cycles, reuse of language concepts, and language integration. The metamodelling method gives us two advantages: First, you can define the abstract syntax of many languages as a combined model. Second, it allows to separate the development of the various tools that are needed to use a language properly.

The first point is founded in the independence from concrete notation and metamodelling's ability to form reusable object-oriented structures. It is the independence from concrete syntax that allows to model language concepts abstractly, independent from syntax details, and it is object-orientation that allows to reuse and specialize the common, abstract concepts in concrete languages. This way the separate ULF languages will become the ULFamily.

And for the second point: Metamodels are data models that specify (and can even standardize) all interfaces that are needed between different language tools. The use of abstract, coherent concepts in the metamodel further looses the coupling between concept implementations and allows its reuse.

With ULF-Ware we propose a metamodel-based, extendable framework for the implementation of ULF in the spirit of the OMG's MDA [3]. Section 2 explains the overall idea and philosophy of ULF-Ware, and we introduce a first piece of ULF-Ware that we are implementing right now – an SDL/UML compiler tool chain. In section 3 we present our current work in progress; this section gives interesting insights into the various aspects of metamodel-based compiler construction. The concluding section discusses the future of ULF.

2 ULF-Ware

The label ULF-Ware denotes all our tools around the Unified Language Family. We constituted all ULF-Ware components around a conceptual model based architecture, the ULF-Ware philosophy. We have begun to implement combined SDL and UML compiler tools. These first ULF-Ware bits have to prove the applicability of the ULF-Ware philosophy.

2.1 Philosophy behind ULF-Ware

ULF-Ware uses a centralistic architecture; it has orbits placed around a core. Figure 1 gives an overview on ULF-Ware.

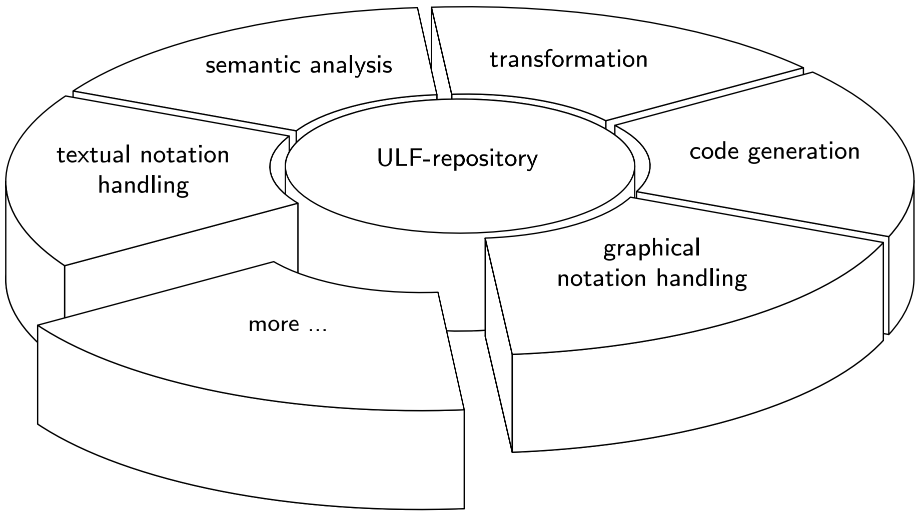


Fig. 1. An Overview of the ULF-Ware.

The core's responsibility is to handle all models: these are M1-models like specifications and programs as well as M2-models, the metamodels. It is a model-centred architecture. It is responsible for model storage and representation; the core also facilitates functionality for model exchange. This way it connects the various language tools; it offers all functionality needed to integrate the orbiting tools. It can be understood as a provider for all common functionality and shared data that the language tools need. We realize the core by using a MOF-compliant repository. The *Model Object Facility* (MOF) is the standardized metamodelling architecture of the OMG [4].

The orbits settled around the core use, import and modify the models in the core; they use the core's operational interface. The distinct orbits act independent of each other, except that their behaviour is based on the shared data provided by the core repository. Because all orbits are independent of each other, the architecture is not fixed to the initial given orbits and is easily extensible.

When we step back from this structural view point and look at core and orbits in terms of languages, then the core handles abstract syntax and the orbits handle semantics, where the concrete notations are considered a part of semantics.

The core provides the languages; it handles the metamodels and it provides a repository for actual language instances (specifications or programs). The language instances are realized by the *extent* concept. An extent is a conceptual space, where the lifecycle of model elements takes place. An extent is automatically generated from the metamodel for which it provides an instance.

The orbits add meaning to the syntax stored in the core. Examples for those semantics are: static semantics, the check of models for static correctness; model transformation, as a possible representation of dynamic semantics; code generation, the question of how a model can be represented by implementation code; textual and graphical representations, which relate graphical or textual tokens to abstract model entities. There are many other possible semantics, like simulation, deployment, etc.

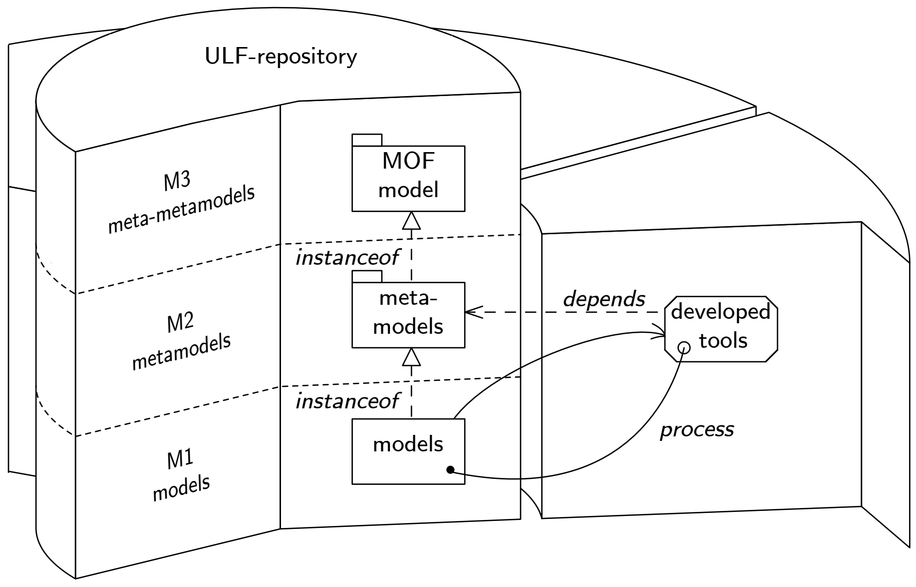


Fig. 2. The philosophy of the distinct ULF-Ware orbits.

Figure 2 provides an closer view on the ULF-Ware philosophy. The core is a realization of a 4-layered metamodelling architecture: The models of every layer are described by a more abstract (*more 'meta'*) model in the layer above. An example: the M1-layer represents SDL specifications; the M2-layer contains the language description, the SDL metamodel; and the third layer defines the language used to write metamodels, in case of a MOF-repository this is the MOF-model.[1]

The semantics are realized by tools. In the first development state these might be hand-written tools that depend on the languages that they are written for. This dependency shows itself in the fact that the tools rely on the metamodel, they rely on the syntax. Tools can express semantics by modifying, creating and using models in the repository – they process models.

A SDL model checker, for example, is a tool that implements rules like: *every agent of process kind must only contain other processes.* Such a rule depends

[1] Please refer to [5] for an introduction into metamodelling architectures.

on the SDL metamodel; it uses the metamodel elements *agent, agent-kind, process, containment*. The model checker applies this rule to an SDL metamodel instances, to SDL specifications.

Another example is model transformation. A transformer simply implements rules for metamodel entities: *every agent can be realized with a Java class.* It depends on the metamodels for SDL and Java; it uses the elements *agent (SDL) and class (Java).* The transformer uses SDL instances; it reads SDL specifications. Based on this model data, it fills an Java extend; it successively creates all Java elements required according to the transformation rules.

Both semantic orbits depend on the same language model (SDL), and they both access the same language instance (the SDL specification). They share the functionality of modifying SDL extents and they exchange SDL models. The orbits are connected through the core. In the context of a compiler tool chain, the model checker proves a model's correctness, the checked model is passed on and is used by the transformer as model transformation source.

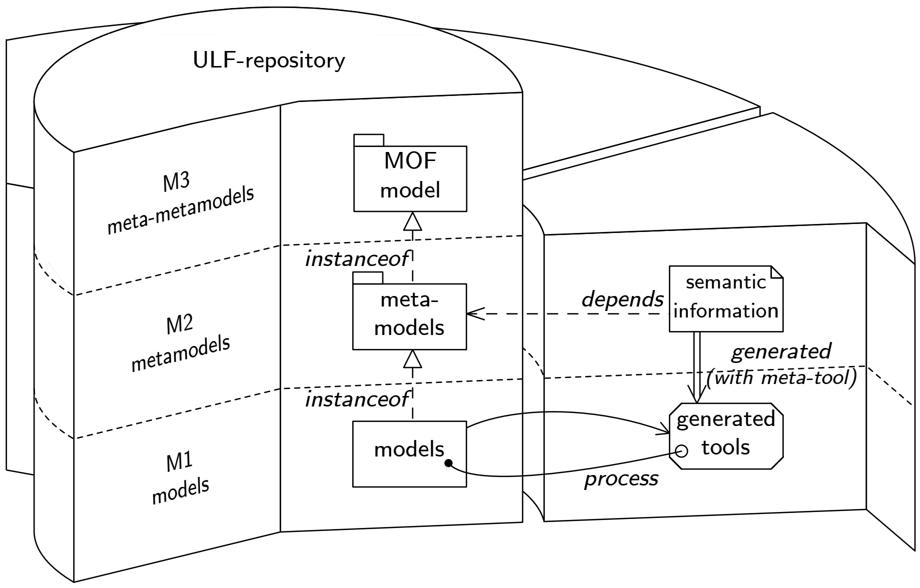


Fig. 3. Realization of an ULF-Ware orbit with a Meta-tool.

Beside the hand-written-tools approach a more sophisticated realization of semantic orbits exists. We call those tools meta-tools (figure 3). Using meta-tools, we detach the language-dependent part of a semantics implementation. As an example, compare two model checkers for two different languages: they basically do the same thing; they apply static semantic rules to models. The only difference and the only language dependency lays in the rules.

Thus meta-tools take semantic descriptions as input, and they realize the described semantic by creating a generated tool. For example such a meta-tool might take a set of static semantic rules from a file of specified format and generate a checker from these rules. This checker can then instantly be applied. An actually existing example are OCL implementations, programs that allow you to write rules for arbitrary metamodels and allow to apply these rules to instances of these metamodels.

2.2 Utilizing the ULF-Ware philosophy – An SDL/UML Compiler

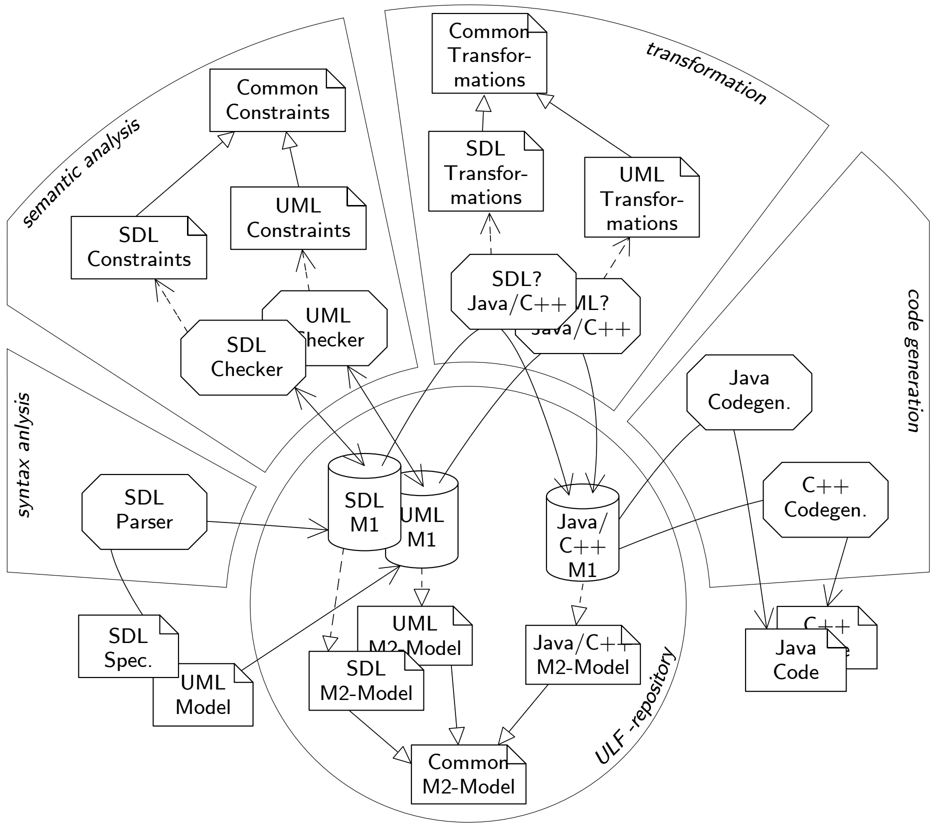


Fig. 4. The Compiler Architecture.

Around the previously described, more philosophical, conceptual ideas we implement actual language tools. Our first goal is a compiler tool chain for an integrated SDL/UML language. These tools shall translate according models into Java or C++ code and finally allow the execution of the generated code in different run-time environments.

Figure 4 shows the compiler's architecture: The core entities (extents for the various languages – the syntax) and the different semantic descriptions that are used to build these tools, as well as the model flow between them. According to the ULF-Ware philosophy, the four boxes represent the semantic orbits *syntax analysis, semantic analysis, transformation* and *code generation.*

2.3 What we gain

The overall ULF idea is to unify the definition of languages. We use MOF metamodelling to describe the syntax of SDL and UML with the same method. We use this unified definition to relate and align those languages with each other. This unified definition, a common metamodel of descriptions for common modelling concepts, is an evolving product. We hope that it will be further filled with concepts contributed from the other ULF languages.

In addition to the reuse of shared abstract syntax (common metamodel), we hope that we can utilize further reuse in the tools and descriptions that describe semantics. The idea is that when a common concept is shared, then the semantics depending on this concept are shared as well.

3 Realization

We begin to implement a compiler for a very simple language, and we plan to successively extend this language. This course of action allows us to focus the development of methods and techniques with out the hassle of very complex languages. The language SDL- is a small subset of SDL-2000; it is a small feature set that allows the specification of executable systems. The language is barely usable in real application, and its only purpose is to give us a research playground. The language is described using an simplified version of the SDL-2000 abstract syntax grammar.

We used the technique described in [6] to develop a metamodel for SDL-based on its grammar. It is part of the used technique to derive and use a set of common concepts. With this metamodel and the corresponding repository, the SDL- compiler core is established. It is planned to successively extend the definition of SDL- (and the used metamodel) with other concepts of the SDL-2000 standard. Later on UML will be integrated, sharing the same concepts and implementations that were written for SDL. This concrete ULF-Ware is planned as an evolving product.

3.1 Common Concepts in the Metamodel Space

Common concepts are modelled with abstract MOF classes which are reused and specialized in different concrete language definitions. Thus the definitions of the various languages are simple specialization of a common metamodel. Due to the shared core, all languages use the same common concepts. If you want to learn more about the idea of common concepts, refer to [7]; the publications [8, 9] are standard material for the object-oriented method and terminology.

There are several ways to obtain those common concepts: There are well-known concepts from different domains such as the object-oriented paradigm or state automata; there are the results of decomposing the concepts of existing languages into smaller, more abstract and potentially common concepts; and it possible to directly integrate languages and compare related concepts.

Common concept blur the boundaries between the modelled languages; this is clearly a positive point. There are two important properties that we would like to emphasize: A common concept is polymorphic, and represents concrete concepts in (different) languages, and a common concept relates concrete concepts in (different) languages. Where the second simply helps to align languages to each other and helps to integrate languages, the first enables to reuse implementations for languages. A transformation or a static condition can be built at a more abstract level, for a polymorphic concept class. Then it can be reused for concrete concepts that act in place of the polymorphic concept.

We understand meta-models as a collection of packages with well-defined relations. They describe language concepts on different abstraction levels, with different levels of detail, and with a maximum of reuse between the distinct packages. This is the same method that is used to define the UML, where the different diagram sorts (each of them is a language of its own) are described by one large model that consists of a diversity of packages and is commonly known as the UML meta-model.

In our example, SDL, UML and even Java as well as C++ use the same basis. Even though the languages are melted together at an abstract level, it does not mean that the individual languages get lost. When you imagine the model as a tree-like abstraction hierarchy of language concepts, then the leaves of that hierarchy represent the concrete concepts, and these concrete features of one language can be clearly distinct from those of another language.

3.2 Tools for Static Language Aspects

Figure 5 shows the model flow from SDL specification until the model is passed for transformation.

The first tool in the chain is the parser. On the input side there is nothing special; it takes a textual SDL specification as input and analyzes it with context free grammar based techniques. We used *JavaCC*, a tool that allows lexical analysis and syntactical analysis with LL-1 grammars.

The result of the syntax analysis is a filled repository. Therefore the parser creates an extent of a special variant of the SDL metamodel, the SDL M2 WCSE. The actions triggered by the various grammar rules then simple create proper elements in that extent.

The SDL metamodel with the affix WCSE (*with concrete syntax extensions*) is basically an extension to the SDL metamodel. The reason for this extra package is that the SDL model is rather abstract; it omits syntactical details that

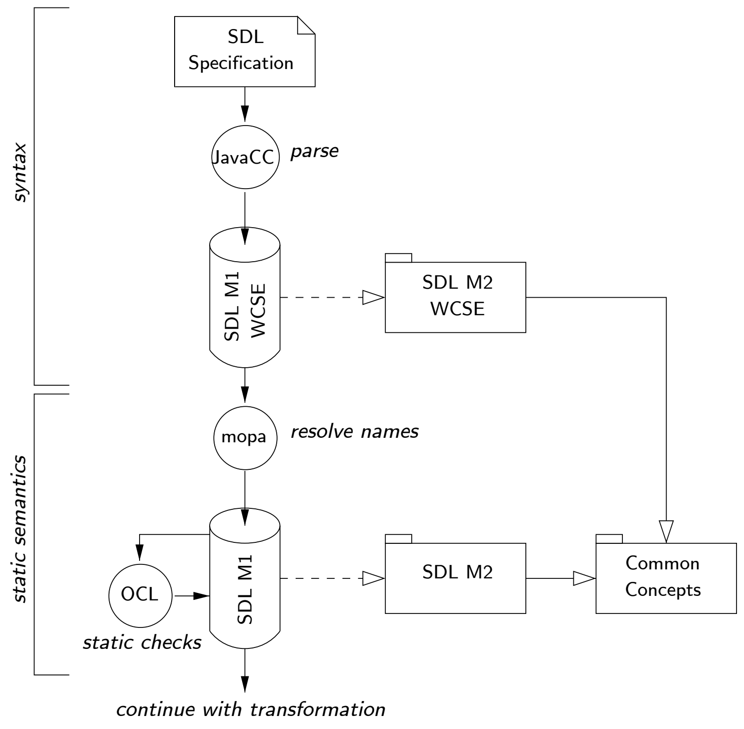


Fig. 5. Model flow from Specification to Model Transformation.

cannot be resolved by the parser on its own. The idea is to use a model transformation from a model that still contains syntax details, to a model in which these details are resolved.

Mainly these syntax details are string references (names, identifier) in the specification text. Take a variable definition as example. A variable definition assigns a type to a variable. In the textual syntax the type is specified, using an identifier. Later in the SDL model this identifier is replaced with a link between the variable element and the type element.

Figure 6 shows a few of these concrete syntax extensions and how they relate to the given example. The syntax extensions contain two kinds of elements. First elements like*ConcreteSyntaxExtension, StringReference, PathItem, Qualifier, Identifier*, they describe concepts that are exclusive to the concrete syntax. Second, the placeholder elements, these elements are specializations of concepts in the normal SDL metamodel. The instances of a placeholder are temporary representatives of the elements that are yet to be resolved.

The variable example again: the parser reads a variable definition; it creates a variable element (a specialization of `TypedElement`) in the repository; it does not know which `type` it shall assign to the created variable, because the parser does

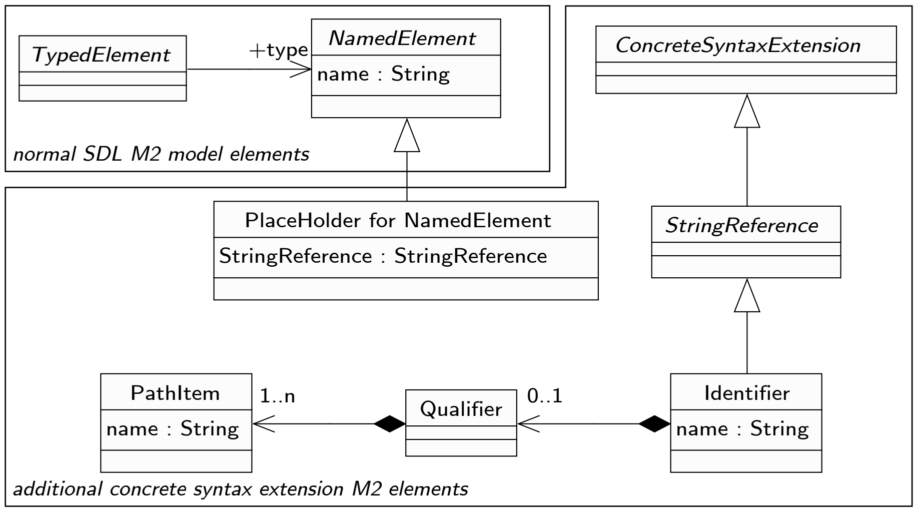


Fig. 6. Concrete Syntax Extension Example.

not know how to resolve the identifier that is used in the variable definition. Thus the only thing the parser can do is create a `PlaceHolder for NamedElement` and to save the identifier into that place holder.

The syntax extensions are partially hand-written; the place holder elements could be generated automatically.

The analysis of static semantics is done in two steps. The first step is to resolve all concrete elements from the *wcse* model and to create a real instance of the SDL metamodel. To do that, the wcse model is traversed for place holder elements. If a place holder is recognized, the reference in it is resolved, and the place holder is replaced by the referenced element.

After all concrete syntax elements are resolved, the SDL specification is a true instance of the SDL metamodel (all WCSEs have been removed), and semantic rules can be applied to check the model's static correctness.

We use the *Object Constraint Language* [10] to implement semantic rules. These OCL constraints are basically predicate logic expressions that use elements from the M2 level and are evaluated against M1 models. The example OCL constraint in figure 7 expresses *An SDL agent of system kind must not*

```
context SDLAgent
inv: this.kind = SYSTEM implies this.container->isEmpty()
```

Fig. 7. Example OCL constraint.

be contained in another agent. The OCL constraints themselves are part of the metamodel. They are attached to the model classes that they constrain. Evaluation of static semantic rules means: For every class all attached constrains are evaluated against every instance of this class.

3.3 Transformations and Code Generation

The repository is now filled with an SDL specification. For the tools further down the chain, it is immaterial *how* the repository was filled. It could also have been the working repository of a metamodel based, graphical SDL tool. While tools like this do not exist for SDL, they may exist for other languages, such as UML, where metamodel based tools are not rare.

In short, there are two easy steps. Step one is to perform a *model transformation* from the SDL M 1 model that we use as input to the combined Java/C++ M 1 model. Step two is to generate Java or C++, which should not be more than simple pretty-printing. These steps are shown in Fig. 8.

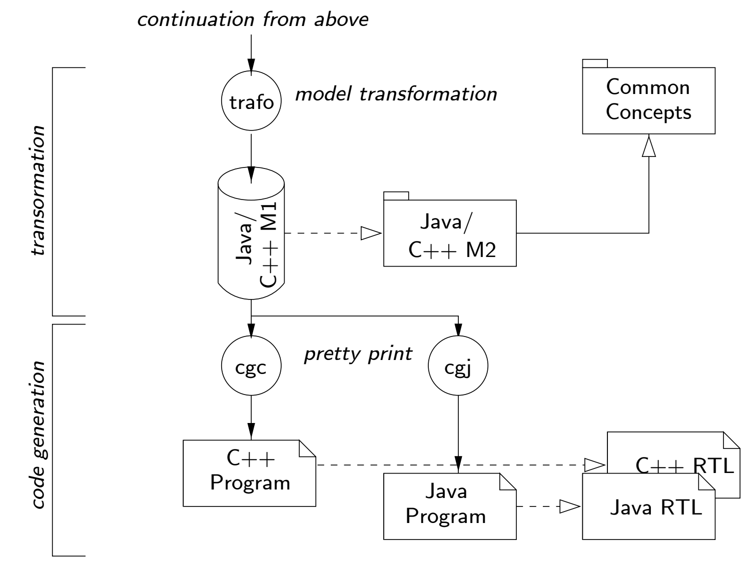


Fig. 8. Transforming the models (continuation of Fig. 5)

The challenging part is, of course, the model transformation. To understand the elements of the transformation, it is necessary to look deeper into the target model. Therefore, we first discuss the target of the transformation before we turn to the transformation itself.

The combined Java/C++ model One of the key strengths of ULF-Ware is the relative ease with which source or target language can be replaced by something else. As long as the metamodel of the new language is similar to that of the old one, only few transformation rules will have to be adapted. This is partly due to the inheritance of transformation rules as explained in Sect. 3.3.

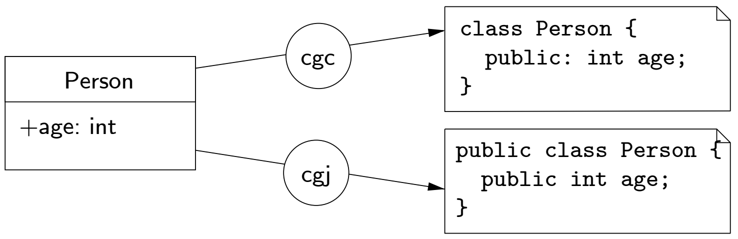
In our ULF-Ware prototype SDL- compiler we use yet another approach for the target model: A combined M 2 model for both Java and C++. Many languages share common concepts, as has been shown and made use of in [11], such as the quite abstract concept of namespace. For programming languages, the similarities 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. Both do not 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-)existance of multiple inheritance. However, we want to use Java and C++ as output languages only.[2] 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. It also inherits from the package `Common Concepts`, which will make the transformation simpler.

To generate source code from the model in the repository is a straight forward unparsing process. In Fig. 8 and 9 these unparsers are preliminarily called cgc and cgj. The results of the pretty-printing are sometimes almost identical, as in the figure.[3]



```
class Person {
  public: int age;
}
```

```
public class Person {
  public int age;
}
```

Fig. 9. Pretty printing for C++ and Java

[2] Although the idea of complete "roundtrippability" is very tempting, it seems almost impossible to build up an SDL specification for an arbitrary Java or C++ program in general. Note that the existing tools that generate UML from Java only cover the structural aspects.

[3] Note that in the figure the left-hand side is in UML syntax for easier recognition.

Run-time libraries When generating target source-code, it is usually convenient not to overburden the code generator with too much intelligence, but to put as much functionality as possible into a common run-time library. As an example, when generating code for an SDL `output`, you *could* generate the code that looks for the correct route, puts the signal in the corresponding queue etc. in place, thereby letting the code generator do all the work. Alternatively, the code generator only produces a function call; the function will be defined in the run-time library.

We have used this technique in our SITE toolchain [12]; an explanation of the library can be found in [13]. The main benefit there was that it is possible to exchange the library to make the generated code behave differently; eg. generate statistics for a simulation run versus fast execution or exchange of signals to the environment via a selectable method.

In the context of ULF-Ware this separation has another advantage: It allows us level the differences in the target languages by abstraction, eg. different data types used for the signal queues etc. This will make it easier to write the code generators, at the expense of having to write the run-time libraries.

Model Transformations In SITE, we have gathered experience with the transformation of abstract syntax trees. We have used Kimwitu++ [14] for the the definition of the abstract grammar and the pattern matching. The transformation of models in repositories is similar to this; the main difference is that the number and order of children is usually not fixed, but expressed through relations between objects. To this end we will use a newly-written pattern matcher for models, called MOPA, already used in earlier stages of the ULF-Ware process.

Just as the source metamodels share common concepts, expressed as a package which both the SDL and the UML metamodel inherit from, the transformations for these languages share common transformations as well. The concrete transformations will inherit the common transformations and complete them with the rules specific for the source metamodel.

Some of the transformations will be trivial: The target metamodel also inherits from the `Commmon Concepts` package. Consequently, some transformations will comprise of merely a copy of the source model element into the target model.

3.4 Meta-tools

The tools discussed so far are sufficient to implement the described metamodel-based SDL- compiler. However, it is very hard to extend our compiler to support additional input languages (in compiler construction terms: to add additional frontends) due to a couple of reasons.

The first one is that usual (textual) programming languages do not have a metamodel. Most programming languages are defined by a grammar that describes the syntactical structure and English text explaining the semantic behaviour. As shown in [11] it is not possible to automatically generate a good metamodel from a given grammar. The metamodels automatically generated are as a matter of principle rather representations of the grammar than of the programming language. However you can use such metamodels as a base and create good metamodels by refinement, but this includes a lot of handwork.

The second problem is that even if you have a grammar and a corresponding metamodel you still do not have a parser for the language nor you have a model-generator. In the SDL- compiler this part has to be hand-coded as well.

The solution to both problems mentioned are meta-tools. One of our planned meta-tools is a program which reads grammars from the desired (input-)language specification and generates a corresponding metamodel. To avoid the mentioned handwritten metamodel refinement we plan to make annotations to the grammar description. These annotations shall be used by the meta-tool to directly generate a good metamodel.

Another meta-tool (or an addition to the first one) is planned that deals with the automatical generation of frontends. These frontends shall be able to parse languages according to a given grammar and generate a corresponding model according to a given metamodel in the repository.

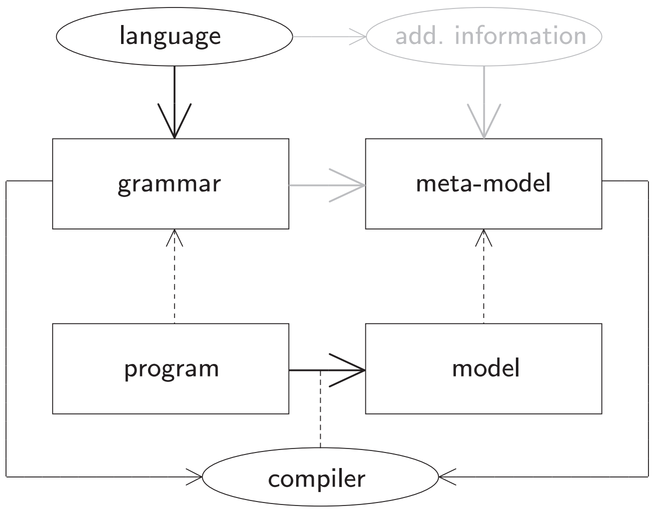


Fig. 10. Connections between grammar-based languages and the metamodelling tools

Figure 10 tries to graphically clarify the mentioned problems and propose solutions. You can see a language and its grammar on the upper-left part of the picture. The language's metamodel shall be derived directly from the grammar and some additional information. The additional information has to be written first, of course.

Once you have a grammar and a corresponding metamodel you can automatically create the compiler which parses programs and creates the appropriate model (the compiler is shown near the lower border in Fig. 10).

Similar but not identical problems exist at the backend of our compiler. In the previous section we described code generators for Java and C++. However, if we want to extend our SDL- compiler to cope with additional target languages we have to implement the according code generators by hand. To avoid this work we plan to implement meta-tools which use the metamodel and the grammar of a specific language and automatically create code generators.

4 Conclusions

With ULF-Ware we propose an open framework, a methodology to build language tools based on a shared core repository with integrated languages based on common concepts. We started to build first example ULF-Ware pieces to prove our concept.

If successful, ULF-Ware allows reuse among languages and among implementation of tools, independent tool development, and tools for integrated languages. This seems promising, but there are a few risky points: The idea of reusing and integrating via a common concept set is yet lacking any practical proof; the integration of languages on tool-level is useless, when there are no proper editors.

However, it is a promising and thus interesting field of research. We plan to continue to develop techniques for ULF-Ware based tools. We will prove the common concept idea with implementations to an SDL/UML integration. Reasonable languages to continue with are eODL, ASN.1, or TTCN. It is a long way to go, but ULF-Ware is a reasonable approach to unify the ITU-T language, and to provide new possibilities and means to model the telecommunication systems of tomorrow.

Even if ULF-Ware is in the first place intended to unify the ITU-T languages, its philosophy (and more importantly all research it results in) is applicable to all language development.

References

1. ITU-T Z.100: Specification and Description Language (SDL). International Telecommunication Union (2002)
2. UML: Unified Modeling Language, Version 1.5. Object Management Group (2003) formal/2003-03-01.
3. MDA: Model Driven Architecture Guide, Version 1.0.1. Object Management Group (2003) omg/03-06-01.
4. MOF: Meta Object Facility, Version 1.4. Object Management Group (2003) formal/2002-04-03.
5. Atkinson, C.: Meta-Modeling for Distributed Object Environments. In: 1st International Enterprise Distributed Object Computing Conference. (1997)

6. Fischer, J., Piefel, M., Scheidgen, M.: A metamodel for SDL-2000 in the context of metamodelling ULF. In: System Analysis and Modeling: 4th International SDL and MSC Workshop. Lecture Notes in Computer Science, Springer-Verlag GmbH (2004)
7. Fischer, J., Holz, E., Prinz, A., Scheidgen, M.: Tool-based Language Development. In: Workshop on Integrated-reliability with Telecommunications and UML Languages. (2004)
8. Martin, J., Odell, J.J.: Object-Oriented Methods: A Foundation. Prentice Hall PTR (1995) 2nd edition (1997).
9. Coad, P., Yourdon, E.: Object-Oriented Design. Yourdon Press (1991)
10. OCL: Object Constraint Language Specification (OCL). Object Management Group (1997) ad/1997-08-08.
11. Scheidgen, M.: Metamodelle für Sprachen mit formaler Syntaxdefinition, am Beispiel von SDL-2000. Humboldt-Universität zu Berlin (2004) master thesis.
12. Schröder, R., Böhme, H., von Löwis, M.: SDL Integrated Tool Environment. Web site, Humboldt-Universität zu Berlin (1997-2003) http://www.informatik.hu-berlin.de/SITE/.
13. Neumann, T.: SDL Code Generation for Open Systems. In: Twelfth SDL Forum. (2005)
14. Neumann, T., Piefel, M.: Kimwitu++. Web site and manual, Humboldt-Universität zu Berlin (2000-2004) http://site.informatik.hu-berlin.de/kimwitu++.