Toolbased Language Development

Joachim Fischer[1], Eckhardt Holz[2], Andreas Prinz[3], and Markus Scheidgen[1]

[1] Department of Computer Science, Humboldt Universität zu Berlin
Unter den Linden 6, 10099 Berlin, Germany
{fischer,scheidge}@informatik.hu-berlin.de
[2] Hasso-Plattner-Institut für Softwaresystemtechnik GmbH
Prof.-Dr.-Helmert-Str. 2-3, D-14482 Potsdam
Eckhardt.Holz@hpi.uni-potsdam.de
[3] Faculty of Engineering, Agder University College
Grooseveien 36, N-4876 Grimstad, Norway
andreas.prinz@hia.no

Abstract. This papers shows how to tackle the problem of ever larger languages and of the combination of multiple languages. The central idea is the foundation of the language definition by a metamodel driven approach that is supported by language engineering tools. Thus the development and the further evolution of languages will be transformed from the traditional paper-and-pen style into an engineering activity. This allows an early detection of problems within the language definition and increases the reliability of the language and the supporting tools.

1 Introduction

Developing high quality software in a cost effective manner is crucial for the competitiveness of software industry, and continues to be a technological challenge. Today, a diversity of languages and notations is used, depending, e.g., on the development phase and the viewpoints to be expressed. However, the use of several notations in the same project often leads to ambiguities and inconsistencies, jeopardizing the overall success of the project. To overcome this problem and to achieve the full benefit of the languages, they are to be integrated both syntactically and semantically.

An additional problem arises from the complexity of the single languages. The monolithic style of language definition of e.g. SDL-2000 makes it not only hard for the user to fully understand and apply the language but aggravates also the development of supporting tools. Today, five years after the finalization of the Z.100 recommendation there is still not a tool available, that fully supports SDL-2000. A hierarchical language definition consisting of a language core and a series of enhancing building blocks accompanied with a clear list of compliance points to syntactically and semantically integrate the single parts promises a solution.

On the other side the development of SDL-2000 and especially the definition of the formal semantics have shown that it is possible and feasible to use software

engineering tools already throughout the language development. This leads to a lower number of errors and inconsistencies in the standard and results in a prototypical set of tools supporting the language.

Moreover, there is a trend towards languages that are tailored for special problem domains and also towards integration of different languages; in the OMG context this can be done by using the extension mechanisms of UML, definitions of UML profiles, and also definition of new MOF based metamodels [2]. Tools that support the definition and application of this type of languages are largely missing.

It has been shown, that the combination of grammar based languages with languages based on a metamodel is possible [3]. However, in general this requires considerable amount of work compared to the combination of purely metamodel based languages. Where as in the latter case a very generic solution can be achieved. To overcome this problem a transformation of the grammar based definition into an metamodel is necessary. In [6] an approach is given, which combines the automatic generation of an initial metamodel from a grammar with semi-automatic model transformations and manual refinements.

Until today most approaches for the integration of languages are tailored to two concrete languages (e.g. SDL and MSC, SDL and UML, etc.) and are not reusable. If the integration of another language is required, many steps of the previous work have to be repeated. Therefore we propose a general framework for the metamodel based integration of languages.

This general framework is the key to the integration of the building blocks as well as of different languages. It will allow the definition of languages on all modelling and metamodelling levels including a proper exchange format for models. This implies also the definition of languages in a modular fashion including proper interfaces. Such a language core can then be extended in different ways to cover further, more language specific concepts, and is sufficiently flexible to support future language add-ins. Ultimately, the availability of a common semantic middleware with the possibility of modular language description paves the way for a new methodology of language design: languages can be customized from a set of building blocks on a case to case basis, depending, for instance, on the application context and the preferences of the system developer.

We will implement this strategy within the SMILE[4] project. Our SMILE project targets all levels of the OMG four-layer metamodel architecture, using a common basic representation on all levels. This representation is called FORM after its main construct `Form`[5]. This lays the ground for executable UML with "metadata facilities" and makes it possible to specify how two neighbouring levels fit together.

In order to integrate our approach with traditional grammar based language definitions of SDL and other ITU languages, we developed supporting tools to transform them into a metamodel based definition. This is accompanied by a proposal for a hierarchical division of the language into a core language and

[4] Semantic Metamodel-based Integrated Language Environment

[5] We also consider FORM to mean "Fantastic Organized Representation of Models".

building blocks of enhanced features. Such a solution does not only help to manage the complexity of the language itself but does also supports the development of dedicated tools.

Different solutions for the combination of the language core with the enhancement building blocks are shown. These are not only applicable for the definition of a single language but also for the combination with other languages (e.g. SDL with UML).

This paper is structured as follows. Section 2 gives an introduction into metamodelling and MDA as well as into the combination of models. Afterwards we give a short description of the SMILE project as a way to solve the integration problems. We discuss the first steps in Section 4, namely the basic FORM representation and the transformation of a grammar based language definition into a metamodel. In Section 5 we provide a deeper discussion of approaches for the combination and integration of different languages. Finally we give some conclusions in Section 6.

2 Metamodelling

In many industrial sectors, the development of high quality software systems in a cost effective manner requires mixing various specification techniques, reusing existing software artefacts and capitalising existing know-how. This reality is addressed by the Object Management Group with the Model Driven Architecture (MDA). This architecture is an approach to the full lifecycle integration and interoperability of enterprise systems comprised of software, hardware, humans, and business practices.

The MDA puts forward the idea that the growing complexity of software systems and the rapid rate of change in technology can be more effectively tackled by putting models into the centre of system engineering. Modelling abstractions enables the separation of platform-independent concerns from platform-specific ones, so developers can cope better with changes in technology. A significant feature of the MDA paradigm is the support it gives engineers to build application models that can be conveniently and with minimal effort and risk ported to new, emerging technologies - implementation languages, middleware, etc. In order to take full advantage of the model driven development techniques it is important:

- to be able to reason at the model level independently of the modelling languages
- to have interaction parts of the model described using different languages
- to use the model for more than documentation purposes (e.g. in verification, simulation, code generation, testing, etc).

When it comes to really support these items, it is quite clear that there is no single methodology supporting all of them. Moreover, as the primary goal of MDA is working with the model, a successful methodology would have to take into account the existence of several languages for system modelling. There is, however, also no support currently for this. Finally, a real model-driven approach

would also include the possibility to define domain-specific languages with semantic background. Therefore an important prerequisite to get MDA into place is the existence of a real level-independent metamodelling environment.

Several metamodelling environments exist today, which are usually bound to one or two levels. Moreover, all these environments apply a built-in semantics to their constructs, which cannot be adapted by users. What is necessary is an environment that allows arbitrary semantics attached to the metamodel with maximal flexibility. Today the only way to attach semantics is by stating static constraints. This is to be extended to allow also the formulation of other kinds of semantics, especially dynamic semantics. Moreover, the connection between metamodelling and the classical grammar approach can be used to allow language tools based on modular language definitions.

2.1 Model Combinations

Modular language definitions open the door for the combination of modelling languages. Several research projects dedicated to this topic have taken the approach to act at the syntax level of either SDL, or UML, or both languages, to make them compatible. In contrast, we are focusing on a common semantic level as a basis for a smooth integration of different semantic concepts and thus effectively establishing the relationship between the languages in a clear and unambiguous manner. As a consequence no one-to-one language translation is needed and real interoperability between system parts written in various modelling languages is enabled.

A key issue for meaningfully combining several modelling languages relates to their semantics. A precise semantics of each language is a basis for deriving a well defined integration within a coherent and consistent modelling framework. All the above languages have a formal semantics based on the mathematical modelling paradigm of abstract state machines (ASMs). Note however that the various semantic models cannot easily be combined, because the underlying semantic concepts are different and require a tighter integration resolving fundamental issues of interdependency and interoperability, which will be featured by our environment.

Tool builders already started implementing the combined use of these languages, which is possible because the concepts of the three languages are similar. However, the interfaces between the languages are not standardized and are only defined ad-hoc. Moreover, the tools developed using these techniques are not able to exchange their models.

2.2 Language Standards

The standardization bodies managing UML and SDL reacted to their respective language community demands for combining the advantages of both modelling languages. ITU introduced into SDL concepts and techniques inspired by UML in the SDL-2000 release. SDL was enriched with concepts such as interface, association, composite state machines, etc, and a new graphical syntax (like UML)

was defined as an alternative notation to the already existing one. With these changes, it is now possible to make an SDL model looking very much like a UML model. Related to this is an ITU initiative to define SDL as an UML profile [Z.109]. Z.109 describes all SDL concepts in terms of UML, often using UML extension mechanisms. The result is an "SDL profile" for UML. OMG felt the same need for rapprochement between UML and SDL, at least in terms of taking advantage of the experience gained after tens of years of using the language in system modelling. Therefore, in 2000 the action semantics proposal was finished, taking the SDL algorithmic language as a means to express details of behaviour in UML. The UML 2.0 definition represented another good opportunity to use SDL know-how in the UML language definition. This is exactly what happened, as the core team that worked at the definition of UML 2.0 contains main contributors at the SDL definition. The result was indeed the "export" of some concepts that proved successful in SDL (like the static hierarchical structure, the communication mechanisms and primitives, etc). Although beneficial for the UML community, these efforts do not help the common use of UML and SDL (they are at the same level as the SDL extensions inspired by UML).

3 The SMILE project

The central objective of the SMILE project is to create a semantic metamodelling framework to allow the definition and integration of computer languages. Included in this project are also concrete integrated language definitions for system design languages as UML, SDL, and SystemC. Another area of language definition will be knowledge representation and reasoning in a web context; one example is Topic Maps but also other languages might be defined and installed as language plug- ins. More specifically, the following goals are to be achieved by the project:

- Conception and definition of a common semantic middleware (CORE)
- Definition of ways to attach semantics to the core
- Establishment of hierarchical system modelling language definitions
- Combined application of modelling languages
- Definition and application of Topic Maps
- Generation of public domain tools for the basic modelling activities.

Before these goals are described in more detail, we present the four conceptual domains of SMILE, visualized through the four quadrants of Fig. 1. The first quadrant we focus on SMILE as a *metamodel* based approch. A metamodel for a language descibes the potential structure of models in that language similar to an abstract syntax. Metamodels allow the modelling of abstract languages. These abstract languages can be specialised to describe concrete languages, like UML and SDL in the figure. A second SMILE domain offers the opportunity to extend a language description by adding additional *meaning*, i.e. semantic information. That could be a set of static semantic constraints, language mappings, a concrete textual syntax, etc. The third quadrant shows the *model* domain;

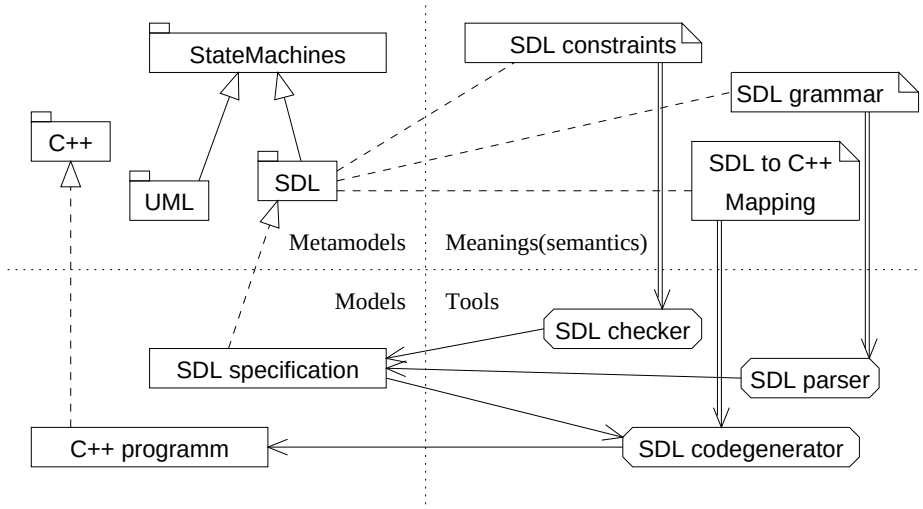

Fig. 1. SMILE overview

SMILE architecture handles models the same way it handles metamodels; they are stored, manipulated and written equally. Models are, of course, instances of metamodels. Thus the meanings defined for a metamodel can be applied to the instances of that metamodel. This enables the forth SMILE domain to automatically generate modelling *tools* from meanings. This is exemplified with the transformation of an SDL specification to C++ code in the figure.

3.1 Conception and definition of the SMILE-CORE

A common semantic middleware is the key to the integration of different languages. The SMILE-CORE will allow the definition of languages on all modelling and metamodelling levels including a proper exchange format for models. This will imply also the definition of languages in a modular fashion including proper interfaces. This core can then be extended in different ways to cover further, more language specific concepts, and is sufficiently flexible to support future language add-ins.

Ultimately, the availability of a common semantic middleware with the possibility of modular language description paves the way for a new methodology of language design: languages can be customized from a set of building blocks on a case to case basis, depending, for instance, on the application context and the preferences of the system developer. Composition of language building blocks will be defined not only on a purely syntactical level, but also on a semantic one. Furthermore, modular structuring of modelling languages also supports the development of customized tools, for instance, for the generation of efficient code, which may require certain language restrictions.

An essential part of the core is the definition of a common exchange format that is shared not only among tool providers, but also among different languages. This may lead to an entirely new technology of aspect-oriented tool building, i.e., the provision of tools that focus on specific aspects of a system specification, such as test case generation or consistency checks.

3.2 Definition of ways to attach semantics to the core

The central idea of this proposal is the explicit handling of semantic information. Almost all current metamodelling tools have the semantics of the language and the definition of the metalanguage built into their tools. We will design our framework in a manner that allows explicit ways to attach semantics to the language entities. This is achieved by considering the language entities as objects according to the object-oriented paradigm and adding semantic attributes to them. This way we will define several different sorts of semantic information, e.g. static semantics, dynamic semantics, textual and graphical representation as well as exchange formats. The ways to define semantics will be defined within the framework itself. This is possible since we allow the definition of models and metamodels on arbitrary levels of the hierarchy.

3.3 Creation of modular modelling language definitions

The first area of application of our framework will be the definition of system modelling languages. In order to be widely applicable, system modelling languages tend to be expressive, extensive and complex. This, however, may inhibit their use in domains such as embedded systems, where more streamlined, special-purpose languages would be preferable. Therefore, the objective is to establish a methodology of modular language definition much in the spirit of UML2.0. Then we will decompose UML, SDL, and SystemC into a core language and a set of building blocks that may then be selected and composed yielding reduced, customized language versions, and that may even be extended by adding special-purpose language elements. Furthermore, the core language as well as the language building blocks will be enriched with a semantics enabling the automatic generation of basic tools.

3.4 Combined application of modelling languages

The availability of the modular language definitions defined in the section above enables the combined application of different modelling languages. This is possible because all the languages share a common semantic base description. Once this integration has been provided, it will be possible to specify parts of a system using different, compatible modelling languages, and to combine them into a single model. This finally gives a precise meaning to the conception of a model driven architecture, as promoted in the context of UML. Furthermore, the integrated model can be used for verification, simulation, testing, and code generation.

3.5 Definition and application of Topic Maps

In order to have a wide spectrum of applications of our framework we will also define languages that are intended to give basic meaning to web services. Specifically we will define a language description for Topic Maps and enrich this with the possibility to attach constraints, which is similar to giving static semantics constraints.

3.6 Public domain tools for basic modelling activities

Given a formal language definition, it is feasible to automatically generate a set of tools for basic modelling activities, such as syntax and static semantic analysis, simulation, reference code generation, and translation to a common exchange format. Experience with SDL-2000 has shown that the required information can be extracted from a properly written standardization document, and that the generation of the tool chain can be completely automatic. This means that as soon as the language definition is complete, tools for basic modelling activities are instantly available. This even holds for language subsets in case of hierarchical language definitions, and provides extreme flexibility. With the generated tools made public domain, commercial tool providers as well as research bodies will be in a position to focus on advanced, special-purpose tools, instead of wasting effort on basic tools and producing incompatible tool chains.

3.7 Methods and Success Factors

The research project described here is of considerable size and can only be successful, because parts of the approach have already been proven in practice.

1. The modular definition of languages has already been given for UML2.0, although not completely related to the semantics.
2. The use of a common base for integrating several languages has been shown in the joint simulation of SDL, VHDL and MSC specifications with the Humboldt-University simulation environment.
3. The generation of a complete tool set (parser, semantics checker and simulation) from a formal language description has already been shown within the SDL-2000 formal semantics project.

4 First Steps

In this section we want to explain the first achievements we have taken in the SMILE project. We will first describe our basic FORM representation which can be used on all metamodelling levels. Afterwards, we describe how to transform a grammar-based language description into a metamodel based one.

4.1 The FORM Representation

As our aim is to represent models the same way as metamodels, we describe in this section a basic representation for models called FORM, which can be used independently of the level of the models. See also [5] for a more in-depth introduction. Our proposed representation is based on the following observation: *Every part of a level in the metamodel structure is an instance of something at the level above.*

The metamodelling environment can be restricted to objects; in a sense "object" is the lowest common denominator of all levels. Since our metamodelling approach is object-oriented, we need to represent the following information:

- Object instances.
- Slots owned by objects.
- Slot values and their type.
- Links between objects.
- Every object knows its class.
- Every slot knows its attribute.
- Every link knows its association.

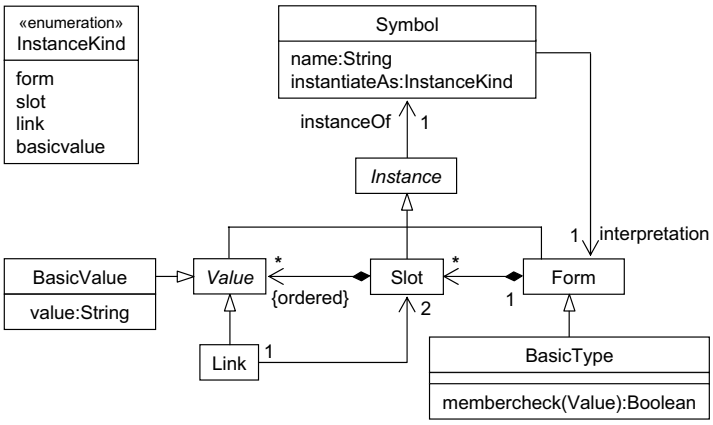

Fig. 2. The FORM definition

All these parts are covered in FORM as can be seen in Fig.2, where we define physical building blocks that can be put together to form complex structures, much like how atoms can be put together to form molecules.

The metaphor above used the term physical which in this context can be seen as the same as concrete, one can choose to see the computer objects as existing physical objects [4]. "Abstract information" must be coded in concrete representation; the coded information is forming a pattern (structure) that might

be interpreted by humans or machines. A computer can interpret in the sense that it can transform and operate on the structure.

It is of course problematic to talk about abstract representations and even to handle them. Therefore we also need a concrete (physical) representation.

An instance of class `Symbol` (typically the name of what it represents), will have a link to the `Form`-instance it symbolizes. The mentioned `Form`-instance, together with connected slots and possibly some other linked `Form`-instance, can be seen as an interpretation of the symbol. Instead of letting an instance be linked directly to what it is an instance of, we have an interface of Symbols between levels. This allows us to handle levels separately from their adjoining levels and only bringing them together when necessary. It is easy to relate the model to a metalevel border: the interpretation of a symbol is on the upper side of the border and the "instances of the symbol" are on the lower side of the border. The symbols of the instantiable forms of one metalevel constitute the interface towards the level below. This way levels can be seen as separate components.

Element	Concrete Visual Representation
Symbol instance	
Form instance	
Slot instance	
Link instance	
Value instance	
Connection (Reference)	

Fig. 3. The visual symbols of the FORM-notation.

In order to be able to visualize the (abstract) FORM representation, we need some (physical) notation. Of course it would be possible to use some kind of UML notation, because UML includes all aspects that are necessary here. However, as our notation is used for representing models and metamodels, it usually leads to confusion if we use a known notation. Therefore we use the symbols shown in Fig. 3 to visualize the elements of our abstract representation . We visualize a reference as a line connecting the two involved entities. The λ inside `Symbol` instance is to be replaced with the actual name of the `Symbol`; the λ inside `Value` instance is to be replaced with the value (e.g. `5`).

In Fig. 4 our mechanism is used with a small example. It can be clearly seen that our representation is capable of representing all levels in a uniform way using only the few notational elements given in Fig. 3.

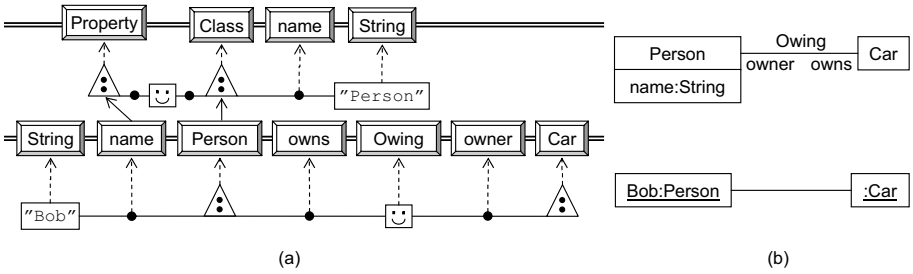

Fig. 4. An instance of FORM (left) representing a part of the UML model (right).

Please note the special handling of primitive data. A primitive type is special since it is implemented or built-in by the underlying infrastructure. Its semantics cannot be found on a level in the metamodel structure - it has no *"relevant substructure (i.e. it has no parts)"* as they state in [1]. Assume that `5` is used on level `M1`; `5` could then be seen as an instance of the type `Integer` which resides on level `M2`; `Integer` could then be seen as an instance of `PrimitiveType` which resides on level `M3`, and finally `PrimitiveType` could be seen as an instance of `Class`.

However, a basic value has already a defined semantics, which can be related to the definition of its (proto)type on the level above. The same is not true for the other instances, they all have an explicit structure as their semantics.

In our approach, however, we do not have the possibility to give special semantics to names – all semantics should be given explicitly or be built-in. Therefore we introduce a special kind of `Value` for the basic values called `BasicValue`. We also introduce a special kind of `Form` for the basic types called `BasicType`. Both basic types and basic values are not characterized by their internal structure, but they carry their semantics already with them. In a way, they have an "external structure"[6]. Handling of external structure is only possible when built-in, which is what we do here.

The difference to UML is that we are giving the semantics bottom-up instead of top-down as within UML. In other words, it is completely arbitrary which class the basic types are collected in because we do not attach meaning to names.

[6] As the research on abstract data types has shown it is possible to turn every external structure into an internal structure. However, as was also shown by the ADT community, this is in general very error-prone and should be avoided if there are other possibilities.

4.2 Transformation of Grammars to Metamodels

In [6] a two-phased methodology was defined to transform a grammar which defines a language into a corresponding metamodel. The first phase is the automatic transformation of the grammar into a primitive metamodel. The initial step of than phase is to introduce for each symbol in the grammar a class in the metamodel carrying the name of the symbol. Afterwards the grammar rules are modelled either by associations between these classes or in case of alternatives by generalization relationships between the classes. Multiplicities and repetition expressions in the grammar are reflected accordingly by multiplicities at the association ends. Additional rules for the introduction and naming of metamodel elements may be applied in case of sub-expressions and complex grammar rules. Altogether the complete transformation set comprises ten rules and has been implemented by a term processor. The tool operates over the syntax tree of the grammar of the original language and produces the metamodel as an XMI-based document.

Due to the derivation of the primitive metamodel from the grammar it still shows characteristics of the grammar which unfortunately limit its applicability. Especially the low degree of reuse of modelling elements and the shallow inheritance hierarchy are unusual for a metamodel of good quality. Therefore in the second phase of the methodology the primitive metamodel is transformed semi-automatically to increase its quality. The main approach is the application of abstraction mechanisms to organize the set of metamodel elements into larger inheritance hierarchies. This takes into account semantic information about the language besides the syntactical information and results in additional abstract metamodel elements. Subsequently, the associations between the classes have to be adapted accordingly and OCL rules are added to ensure consistency. A hierarchy of packages is used to keep the metamodel manageable and comprehensible for the human user. Even though the second phase of the transformation requires a large amount of human input, tools are provided to support the transformation. The rules are implemented by Java applets which operate over a repository containing the metamodel.

The feasibility of this methodology has been proven by the transformation of the SDL grammar into an SDL metamodel.

5 Combination of Modelling Languages

In [3] three general types of model combination and combination of modelling languages within the software engineering process have been identified:

- Process Driven Combination: Models of one design phase or activity serve as starting point for the next phase or activity and different modelling languages are used for the activities or phases. The models are ordered chronologically, i.e. they are not used at the same time.
- Abstraction Driven Combination: The single models describe the system within the same design phase using different abstractions (views). These

views usually deploy different languages. An overall model can be obtained by a superimposition of the sub-models.
- Structural or Architectural Combination: The single models describe different components or sub-systems of the overall system. Different domain or task specific languages may be applied for the sub-models. The overall model can be obtained by a composition/aggregation of the sub-models.

The first two approaches serve a smooth transition between the different phases in the software design and a dedicated capture of design decisions. However, if code generation or prototype production is the goal, the emphasis lies on structural combination and abstraction driven combination. The three forms of model combination do result in different requirements on the languages and may result in varying technical solutions. The structural combination e.g. requires the modelling languages to be able to:

- specify an overall model as a composition of interconnected subsystem and component models,
- specify interfaces between the components and subsystems, and
- specify the components and subsystems as open models (i.e. they must be able to communicate with their environment).

Due to the independence of the single submodels structural combination enables the autonomous verification, validation and evolution of the submodels. However, the interfaces specified for the interconnection must not be changed. The classical approaches to support the combination of languages are the integration of the languages into a single languages or the translation between the languages. However, both ways have shown their limitations for the practical application. Therefore we propose to base the combination on a set of well defined concepts, that form the concept space or metamodel for the method. Mappings are defined, that relate a concept or a set of concepts to elements of a concrete target language (e.g. UML or SDL). Adhering to this idea, the combination comprises three parts, which are

- the metamodel, that defines a language- and tool-independent core terminology for the specification and design of applications,
- a set of rules relating the core concept space with the concept spaces of the actual languages, and
- a single set or multiple sets of mapping rules, which enable a smooth transition from the design to the implementation by concrete tools.

Starting from the informal definition of the common concept space, we apply our SMILE-framework to specify a metamodel that captures the concepts of our concept space. Of course, in order to determine the set of concepts that forms the common concept core, it has to be decided which form of model combination shall be supported. There are three ways to connect the common concept space to the metamodels of the languages which shall be combined:

- use of the common concept space (core language) as foundation for the single metamodels,
- definition of the common concept space as specialization of the metamodels,
- introduction of additional references to the common core in the metamodels.

The first case provides us a tight integration of the original languages, however, it does also require fundamental modifications of the original metamodels. For that reason mostly the two other cases will be applied. They do introduce reference points into the languages which are directly based on the set of concepts common to both languages and specify the interface between the subsystems.

Finally one has to build this common concept space. There are different sources that can be used to identify concepts that are common to the considered languages, and that will probably be common to new languages too:

1. concepts that are already defined and well known, like concepts of the object oriented paradigm or concepts taken from abstract type systems or abstract computational models like state automata, algebraic expressions, etc.
2. decomposition of concepts of existing languages into smaller, more abstract, and potentially common concepts
3. by direct integration of languages and straightforward comparison of related concepts of different languages.

Source (2) would be the most desirable. It describes object oriented method that we use in daily software development and is known to lead to flexible and reusable artefacts. However, it is also the most problematic source, because it is unlikely that two abstractions gained independently from two languages will be the same. Therefore a direct comparison of languages is inevitable and the integrated use of source (3) and (2) would be the method of choice. Of course also the ideas taken from source (1) must be taken into account. Those known concepts reflect the common knowledge about languages and only this knowledge makes a set of common concepts applicable for future languages.

Obviously the common concept space is a moving target and it can only be modelled in a delicate, evolutionary process, adding features and concepts with every new concrete language that is included to the development process. Thus SMILE will not only profit from common concept spaces, but it also has to provide the means for a controlled verified derivation of common concepts from concrete language definitions.

6 Conclusions and Further Work

This article shows the feasibility of an engineering approach to language development. In the same way as software, languages too can be developed in a stepwise and iterative process, where the current state can be checked using appropriate tools. Most of the tools necessary for this are already in place for grammar-based language definitions.

We propose to transfer this expertise to the metamodel world and to keep the same level of executability and tool support. In addition, we combine them with the much better modelling facilities of the metamodel approach. The article has shown that already a considerable amount of knowledge is present to make MDA work not only on programming level but also on language level. However, several parts are still missing. The SMILE project aims at tackling exactly these missing parts and of collecting the existing bits and pieces.

The work within SMILE is still at the beginning. Although we have a clear idea on what to do, we still have to make it happen. The parts already done encourage us to proceed with this approach.

To fulfil the SMILE vision and to handle its rather large scope we are gathering a consortium of universities, tool builders and end users to join forces in an EU project. The work on the central SMILE structures, presented in section 4.1 is progressing and we started to integrate grammars and metamodels as indicated in chapter 4.2. The analysis of languages, especially SDL and UML has started, metamodels are developed and first common concepts are put to work. The IST project EIAO (`http://www.eiao.net/`) already applies SMILE technology. We expect the first running applications in our pilot project within one year.

References

1. OMG Editor. *OMG Unified Modeling Language Specification, Version 2.0.* OMG Document. OMG, http://www.omg.org, 2003.
2. OMG Editor. *Revised Submission to OMG RFP ad/2003-04-07: Meta Object Facility (MOF) 2.0 Core Proposal.* OMG Document. OMG, http://www.omg.org, April 2003.
3. Eckhardt Holz. *Kombination von Modellierungstechniken für den Softwareentwurf.* Der Andere Verlag, 2004.
4. Ole Lehrmann Madsen, Birger Møller Pedersen, and Kristen Nygård. *Object-Oriented Programming in the Beta Programming Language.* ACM Press, Addison-Wesley, 1993.
5. J.P. Nytun and A. Prinz. Representation of levels and instantiation in a metamodelling environment. In *2nd Nordic Workshop on the Unified Modeling Language (NWUML 2004)*, 2004.
6. Markus Scheidgen. *Metamodelle für Sprachen mit formaler Syntaxdefinition am Beispiel von SDL-2000.* Humboldt Universität zu Berlin, 2004.