Modelling SDL, Modelling Languages

Michael PIEFEL

Institut für Informatik, Humboldt-Universität zu Berlin

Unter den Linden 6, 10099 Berlin, Germany

piefel@informatik.hu-berlin.de

and

Markus SCHEIDGEN

Institut für Informatik, Humboldt-Universität zu Berlin

Unter den Linden 6, 10099 Berlin, Germany

scheidge@informatik.hu-berlin.de

ABSTRACT

Today's software systems are too complex to implement them and model them using only one language. As a result, modern software engineering uses different languages for different levels of abstraction and different system aspects. Thus to handle an increasing number of related or integrated languages is the most challenging task in the development of tools.

We use object oriented metamodelling to describe languages. Object orientation allows us to derive abstract reusable concept definitions (concept classes) from existing languages. This language definition technique concentrates on semantic abstractions rather than syntactical peculiarities. We present a set of common concept classes that describe structure, behaviour, and data aspects of high-level modelling languages. Our models contain syntax modelling using the OMG MOF as well as static semantic constraints written in OMG OCL.

We derive metamodels for subsets of SDL and UML from these common concepts, and we show for parts of these languages that they can be modelled and related to each other through the same abstract concepts.

Keywords: Metamodel, MDA, SDL, MOF, OCL, UML

1. INTRODUCTION

Increasing demand for more and more complex software requires that software engineering itself becomes more and more complex. Informal problem analyses, crude design techniques, and ad-hoc program writing cannot fulfil modern software engineering needs. For that reason the building block of today's quality-conscious realizations of extensive software projects have to be formal analyses, complex design models, generated program code, and automated development tasks within standardized software engineering processes.

The mere size of today's software system forces you to break down its complexity. One way (horizontal separation) is to break the complexity down into multiple views on the system. The various aspects of a system are modelled in separate views. A common partitioning is into a structural, a behavioural and a data view. Following this principle, you use different languages that are specialized for the modelling of different system aspects. Often those languages are related, and they build a family of languages, like the UML [14] or the collection of ITU-T languages, called ULF.

Vertical separation is the other way to deal with complexity; a system is modelled by using different levels of abstraction. The

Model Driven Architecture (MDA) [11], a method with increasing popularity, uses models in various levels of abstraction to drive the process of software development from early analysis to a fully implemented and deployed ready-to-use system. In the MDA, the important details that you abstract from in higher level design phases are platform dependency, performance issues, real-time aspects, etc. Of course you have to use different languages for different levels of abstraction; you use languages that are specialized and made for the depth of detail needed at a certain point in the development process.

Therefore complex software systems force you to use many languages because you model different system aspects with different languages and model the system in different levels of abstraction, using different languages. But as matter of fact, the integration between languages is usually poor, and most languages have more commonalities than differences. This provides a big motive for a technique that allows to relate languages and thus promotes a better integrated use of languages and allows the reuse of their language concepts and implementations.

We think that metamodelling is the technique that is needed. Metamodelling allows object-oriented decomposition of languages by finding abstractions. Metamodelling offers different concepts to relate the concepts of languages and reuse via inheritance hierarchies. In this paper we want to exemplify how metamodelling can be used to build a common model for SDL [8] and UML.

Section 2 gives an introduction into metamodelling; it explains the idea of common concepts, the reusable core of our models; it shows how common concepts are identified, modelled and used. In Sect. 3 we present a common structural, behavioural and data model. This model is used in Sect. 4 as the common base for an SDL and an UML model. We demonstrate this common model by applying it to an SDL model and UML model of the same simple example system. Here we show: How the common model relates structural, behavioural and data concepts to each other and how the common concept correlates SDL and UML concepts with each other, and how these concepts are reused in the two different languages. Section 5 will finally draw some conclusions and present ideas for further work.

2. METAMODELLING, OBJECT-ORIENTATION AND COMMON CONCEPTS

Before we present the result of our efforts to build common models, we want to give an introduction about what metamodels are, what concepts are, what makes concepts common, and how does the overall idea of metamodelling for language integration/alignment and language reuse work.

Metamodelling

The building blocks of metamodels are concepts. Concepts are modelled as classifications of language entities of the same kind. Examples for concepts are *variables, data type, procedures etc.* Those classifications (concept classes) can be related to other concepts: they can inherit from more abstract concepts; they can aggregate other concepts; they can associate with other concepts, and finally they can contain other concepts: structural features, like attributes or references to other concepts, and behavioural features, like methods. Meta-models are usually notated using UML class diagrams.

Semantically a *concept* is *an idea or a notion that we apply to things, or objects in our awareness. A concept applies or does not apply to an object* [10]. When you transfer this common definition to the metamodelling domain, a concept is an idea or a notion that applies or does not apply to an element of a model or a program.

A good introduction to metamodelling can be obtained from Atkinson [1, 2] or the MOF standard [12].

Concepts

The nature of object-orientation is to compose a larger system from smaller parts or to decompose a larger system into smaller parts. Of course you can do the same with concepts; you compose more expressive concepts from less expressive concepts.

There are two forms of composition; the specialization of concepts using inheritance and structural composition using concept relations like aggregations or associations. This is known as object-oriented (de)composition [3], the (de)composition with abstractions as key; you compose less abstract concepts from more abstract ones.

There are two important metrics that describe the applicability of concepts for further composition. These metrics are cohesion and coupling [4]. A concept has a high cohesion if it describes a single, atomic characteristic. A concept *NamedElement* whose only characteristic is the property *name* is highly coherent. A concept has low coupling if it does not depend on other concepts. The concept *NamedElement* has low coupling; it only depends on the primitive data type *String*.

These metrics describe the capability of concepts to be the basis for composition. A concept with high cohesion and low coupling is easy to use without getting characteristics you do not want (high cohesion) or dependencies you can not handle (low coupling).

Common Concepts

As a matter of fact, languages have more commonalities than differences. The languages that are made to model the same system aspects, but on different levels of abstraction, often share many concepts – because they are meant to model the same things, but just with a different level of detail.

For example eODL [9], SDL, and even implementation languages (like Java) incorporate the concept of generalization, they support structural features like attribute, or they have data typed elements such as variables. Here the shared concepts form a direct alignment between languages and allow easy derivation of translation processes to convert models between the different abstraction levels.

On the other hand, languages that are made for modelling at the same abstraction level, but for the modelling of different aspects, basically use more different concepts. In the end, those languages are used to model only one consistent system, and for that reason

they have to be related to each other. Their concept spaces overlap. The concepts in this intersection are common concepts.

As an example, take a parameterized procedure call. The parameter's type is in the domain of data modelling; the procedure call that uses the parameter falls into the structural view; the procedure's behaviour may access the parameter and uses operations defined by the data type of the parameter. So all of the three views, and their respective languages, must have the same syntactical and semantic idea about what a data type/parameter is; they must share this common concept. Here the shared concepts often form a direct relation between languages that can be utilized for an integrated use of languages.

How does metamodelling allow the shared use of common concepts? Well as laid out earlier metamodels are simply object-oriented models and their entities can be composed (reused) to more concrete concepts. We mentioned two metrics that influence the applicability of a concept for reuse. In summary there are three properties that a concept must fulfil in order to be a common, and more importantly a reusable concept:

1. A common concept must be common; it must be applicable to the entities of different languages.

2. It must have high cohesion in order to be freely and flexible combinable without the ballast of unnecessary or unwanted concept features.

3. It must have low coupling; this is needed to prevent conflicts when the concept is used for composition.

Sources for Common Concepts

The overall nature of a set of common concepts is that this set is evolving. As languages evolve, new concepts and techniques are invented and used, the set of common concepts must adopt and evolve over time. In other words, the common concepts must continuously adopt to their sources. What are these sources, how do we find them? We identified three different sources that can be used to identify concepts that are common to considered languages, and that will probably be common to other languages:

1. Concepts that already exist, are well known. These are concepts of the object-oriented paradigm, concepts taken from abstract type systems, or abstract computational models like state automata, algebraic expressions, etc.

2. Object-oriented decomposition of concepts of existing languages into smaller, more abstract, and potentially common concepts.

3. By direct integration of languages and straightforward comparison of the related concepts in different languages.

Source 2 would be the most desirable. It describes the object oriented method that we use in daily software development and that 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.
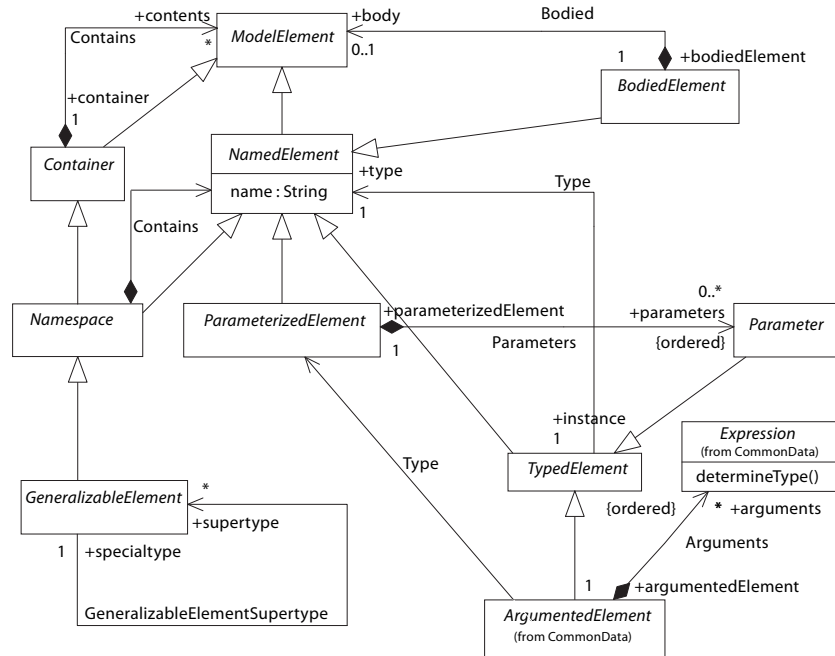
Figure 1: Common Structure

3. THE COMMON MODEL

In this section we present a model of common concepts for structural, behavioural and data modelling. We developed this model by comparing SDL with UML, we looked for abstractions in those languages; we normalized these abstractions to fit into each other; we were also influenced by existing metamodels to UML and our common knowledge about language constructs.

Here are few notes on the notation: The model is notated with UML class diagrams. Most of the concept classes are abstract; so they cannot be instantiated directly; they have to be specialized by a concrete language definition. There is heavy use of specialization throughout the model. Unfortunately, the support for specialization of associations in MOF/UML 1.x is very poor. In order to notate association specialization anyway, we used dependencies with the same name. They should be understood as replacement for the original association, but with the dependency's ends as new specialized association ends.

We omit detailed explanations of the model, but we give a few notes on those parts that we think are not self explanatory.

Common Concepts among Common Concepts
The Common package (Fig. 1) contains the most abstract concepts; these are concepts that themselves are heavily used by the common concepts. ModelElement is the most abstract element; it is the super concept to all other concepts. Its purpose is to model every element as possible contents for the Container concept.

The container-contents relation (also known as composite-component) uses composite aggregation semantics in order to impose a strong part-of-a-whole semantics on the instances of Container and its content. To both participants of this relation there are specializations: Namespace and NamedElement. These four concepts are widely used in structural modelling; classes, components, SDLAgents, packages are only a few examples.

The usefulness of BodiedElement is yet unclear. On the one hand this element can be replaced in a *container-contents* relation. On the other hand, some constructs, like a process in SDL, contain named elements but also contain a nameless body (processes contain a state automaton). Further applications have to show whether BodiedElement is needed or not.

The concept TypedElement characterizes an element that references exactly one *type* of that very element. The name *typed* easily confuses, because you instantly think of *data-type* semantics; but the concept TypedElement is just a syntax property that simply says that there is exactly one *type* to every *typed element*.

The concept Parameter for example is a classical application of TypedElement. But another application of TypedElement, ArgumentedElement, might seem a bit more peculiar. It is best explained through an example: Imagine a *procedure call* (an *element with arguments*) and its *type* the *procedure declaration* (an *element with parameters* – ParameterizedElement).

On first sight GeneralizableElement seems to be incoherent because it unnecessarily specializes Namespace. But wherever generalization/specialization occurs it seems to have inheritance semantics. The reason is that in order to have something to inherit, generalizable elements should be able to contain something. Take the methods and members of a Java class as an example.

Behaviour
The next model package is CommonBehaviour (Fig. 2 on the next page). Because there are a lot of techniques that are used in modelling behaviour, this package, or better the set of behaviour modelling concepts in general, have potential for further development. We only present concepts for the behaviour modelling with state automata and concepts for a simple communication mechanism.

A StateAutomaton is a composite element that contains States and Transitions. Transitions connect two states. NextState directly relates a transition with its successor state. This is straightforward.
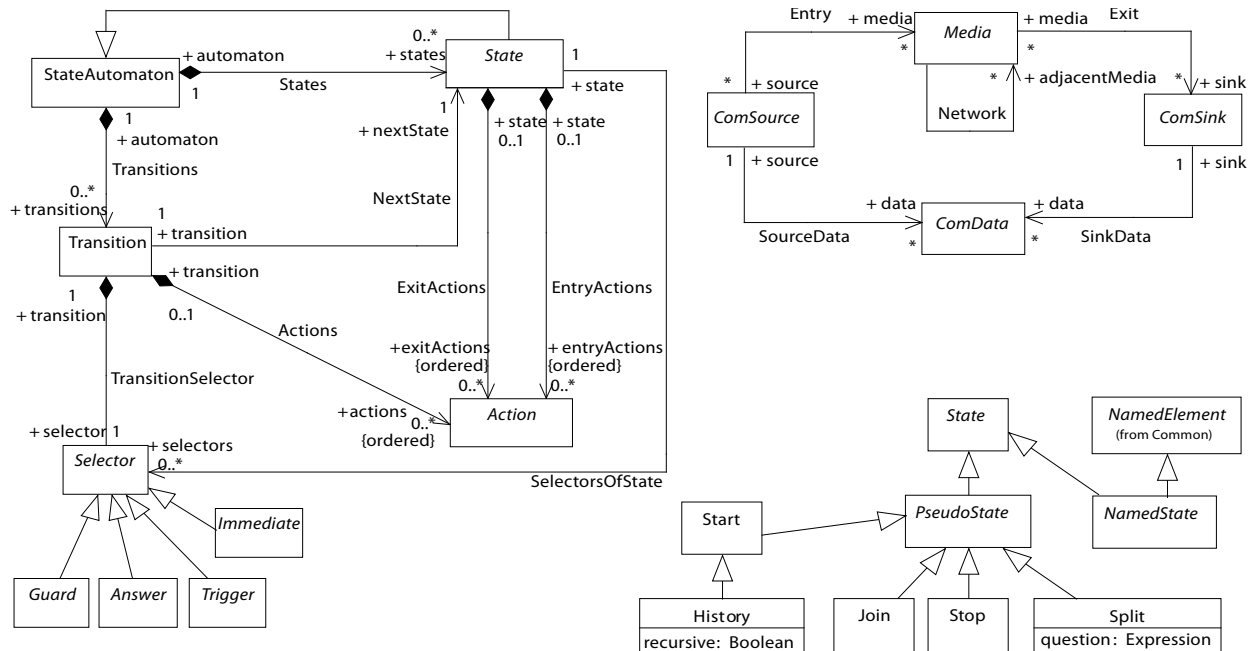
Figure 2: Common Behaviour – State Automata and Communication

Unfortunately, the relation from the predecessor state to an transition is not straightforward at all. For one thing, there has to be a concept that *selects* the transition that should be executed; for another we have to decide where to place the Selector. The current solution (as presented in the figure) is that Selector is part of Transition and can be reached via the SelectorsOfState association from the predecessor state.

As many decisions this one yet has to prove sensible. Other possibilities are that the *selectors* are part of the states and simply reference the *transitions* that they *select*; or the *selected* transition is part of the *selector* (it is done that way in the Z.100 (SDL) specification). We chose this solution because we think it meets best with the scholar's idea of a state automaton.

Transitions can contain Actions. Often a transition shall not simply execute a chain of actions, but branch on conditions or join with another transition. To enable this, we introduce PseudoState; states that that a automaton cannot reside in over a period of time, but only can instantly fire a transition.

The states Split and Join are such pseudo states. A Transition cannot perform splits or joins on its own; but it can connect join and split states. A split state can only be predecessor to transitions that have Answer selectors. Those answers select the path to choose, according to the question expression of the split. A join state, can only be predecessor to exactly one transition with Immediate selector.

Communication concepts are modelled in a simple way; a ComSource sends ComData over a Network of Media to an ComSink. These concepts are explained in detail in Sect. 4, where they are used.

Data
The data concepts modelled in the CommonData (Fig. 3 on the next page) describe an operative interface to actual data types, expressions and their application, and the definition variables.

The concept ValueDataType is capable of modelling an interface to any algebraic data type, since it uses Literal to define a *set* and it defines Operations that work on that set.

The Expression concept allows arbitrary inductive expressions over data using LiteralExpression, VariableAccess and AnyExpression as base step and Operation as inductive step.

4. APPLICATION OF THE COMMON MODEL

The model for SDL was derived from the SDL abstract grammar using the techniques described in [7, 13]. Basically this means that a preliminary model was generated from the SDL grammar, using an automated mapping from BNF to the MOF-model (the language that we write our metamodels with [12]), and then the elements of this model were related (marked as specializations) to the elements from the common model. For the UML model, concrete concepts were taken from the UML standard and related to our common model.

Unfortunately this is a paper of limited size, so it is not possible to provide you with the actual metamodels for SDL and UML that we derived from the common model. Instead we would like to show an SDL and UML model of an simple example system, and we want to show with this example, where which common concept is used in which concrete language concept. The example system is a very primitive constructed system but it shows some major concepts of the modelling of structure, communication, and state automata.

Figure 4 shows the structure of the system. In both models there is multiple use of the communication concept Media. In SDL, gates and channels are media, used to create a communication link between agent instances. UML defines associations between classes to define links as media between class instances. Media is conceived as a unidirectional communication link; for that reason both the bi-directional SDL channel as well as the bi-navigable UML association must be aggregations of two media. Further the UML association is also an aggregation of its association ends (each of them is a media too); the SDL gates instead are part of the agent instead.
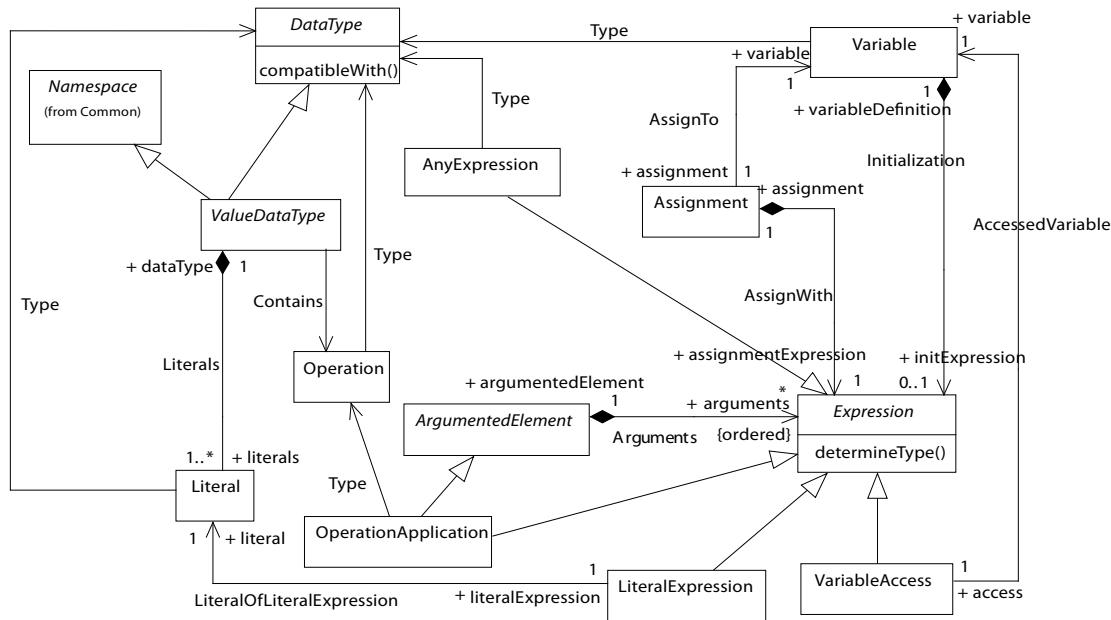
Figure 3: Common Data

It might seem strange to apply communication semantics to UML association. The semantics of UML is for the bigger part open to individual interpretation. We apply the following semantics: There must be a navigable link from object *a* to object *b* in order to allow *a* to call a method (to send a message) on *b*.

This is a bit difficult because methods have synchronous communication semantics. This means that there must be an additional implicit ComItem characterising the *return*. Regardless of these semantic differences both SDL signals and UML methods comply to the ComItem concept.

There is one common concept needed here that we did not cover in the common model; that is an interface concept. An interface is a communication concept that constrains possible communication; an interface says which ComItems an entity can receive or send. This way an interface is a concept that is purely part of static semantics; it is only used for model checking and has no influence on the actual system behaviour.

We did not introduce an interface concept because we are not sure whether it is really needed. All elements of an interface (signals, methods, etc.) are already covered by other concepts (gates, class, etc.); but for a common implementation of static semantics an additional common interface concept might be very useful. The concept interface becomes even more important when specialization and polymorphism complicates the implicit interfaces of UML classes of SDL agents.

In both languages there is a instantiable structure type concept; agent type in SDL and classes in UML. Both type concepts are GeneralizableElements and Namespaces. They might also comply to the BodiedElement, like their relation to a state automaton that describes the behaviour of a type. Both concepts are instantiable; but SDL allows the definition of whole instance sets. Due to the more detailed and more formal semantics, the SDL agent type is a bit more sophisticated; it differentiates various kinds of agents with different characteristics. The UML class is a more common broader concept. But both share the mentioned concepts.

As mentioned SDL agent and UML class are namespaces; they are namespaces to distinguish features of those structure types. A UML class may contain attributes and methods; an SDL agent is namespace to various definitions including variables and procedures.

Those features might have a specific visibility. In UML they follow the known *private, protected, public* scheme. In SDL visibility is hidden behind the possibility to export variables and to make procedures remotely accessible. As interfaces visibility has purely constraining semantics; it is tightly coupled to the interface concept. It is omitted from our common model for the same reasons.

Both languages are almost identical about the State concept. Beside the *normal* named state, there are pseudo states. These are states in that the behaving entity cannot reside but only immediately trigger a transition or completely stop any further behaviour. Such states are for example start or stop states. States can, in both languages, contain inner states that are computed in parallel.

For the modelling of transitions between such composite states SDL defines entry and exit points to derive semantics that is detailed enough to allow an unambigiuous execution of the automaton. Such *points* comply to the pseudo state concept (not part of the example).

Transitions are in both languages executed when selected by a trigger, a guard, or similar concepts; selectors can also be combinations of such concepts. Transitions can contain actions such as tasks or outputs in SDL or method calls in UML. The exit from a state (the execution of a transition from this state) as well as the entry to a state (the execution of a transition to this state) can be accompanied by actions. All exit, transition, and entry actions are executed in a specific order.

In SDL it is possible to split one transition into reusable parts using labels. Because this is purely syntactical, there is no concept for that in the common model.

Another selector is the `Trigger`, for example inputs in SDL and method call events in UML. Both concepts also comply with the `ComSink` concept and are the far side of a communication. This means that a signal and its actual arguments (the `ComItem`) are received in SDL and the method call message and the actual arguments (the `ComItem`) are received in UML. The opposite is the output in SDL and the method call in UML. These elements comply to both `ComSource` and `Actions`. `ComSource` because they are the local side of a communication and `Actions` because they are executed as part of a transition.

The example system shall transit from *B* to *A* or *B* depending on the value of *varA*. The behaviour has been realized differently in SDL and UML to exemplify both approaches; both are possible in SDL and UML. In UML the transitions selector is a combination of a `Trigger` (the `methodB` call event) and a `Guard` selector; there are two different selectors for two different transitions. In the SDL model, a first transition, leading to a `Split` pseudo state, is selected as usual by the input trigger. From that split state the next transition is immediately selected by one of the two `Answer` selectors; the answers finally make the difference.

5. CONCLUSION

We use metamodelling's object-oriented characteristics to build a model of concepts that are common in high-level modelling languages. This set of common concepts can be the basis for the definition and implementation of languages.

There are three factors that are crucial for the success of the common concept idea:

- Common concepts are subject to frequent changes, and the set of common concept has to be constantly evolved to adopt to new trends in the modelling world. The problem is: How flexible and adoptable is metamodelling really?

- Can common concepts be the base for reusable tool implementations? Is it possible to implement a shared concept once and reuse it in tools for different languages? How easy is it to develop tools for new languages on the basis of already existent tool implementations.

- When common concepts evolve, can implementations based on common concepts evolve with them? Is metamodelling flexible enough to evolve common concepts without constantly crippling and deprecating already written tools that are based on them?

All of these question can only be answered when metamodelling and the common concept idea is practically used. From the theoretical view point, the only point of view that we can look from for now: due to the success of object-oriented software development that metamodelling is based upon, all these question have to be answered with a *yes*.

But despite its success, object-orientation could not fulfil every promise made, and it cannot be the final answer to all software engineering problems. For that reason the theoretical (hypothetical)

yes might seem a bit faint and questionable to the practitioner's eye.

We started to use the common concept idea, and the models presented in this paper, in methodologies, frameworks and actual implementations for language tools. In [5] a conceptual architecture for language development and tool implementation is proposed, and we use this *tool-based language development* to implement tools for ITU-T languages; our work in progress is presented in [6].

6. REFERENCES

[1] Colin Atkinson. Meta-Modeling for Distributed Object Environments. In *1st International Enterprise Distributed Object Computing Conference*, October 1997.

[2] Colin Atkinson and Thomas Kühne. The essence of multi-level metamodeling. In *4th International Conference of the Unified Modeling Language*, October 2001.

[3] Grady Booch. *Object-Oriented Design with Applications*. Addison Wesley Professional, 1991. 2nd edition (1993).

[4] Peter Coad and Edward Yourdon. *Object-Oriented Design*. Yourdon Press, 1991.

[5] Joachim Fischer, Eckhardt Holz, Andreas Prinz, and Markus Scheidgen. Tool-based Language Development. In *Workshop on Integrated-reliability with Telecommunications and UML Languages (WITUL)*, November 2004.

[6] Joachim Fischer, Andreas Kunert, Michael Piefel, and Markus Scheidgen. ULF-Ware – An Open Framework for Integrated Tools for ITU-T Languages. In *SDL 2005*, LNCS. Springer-Verlag, 2005.

[7] Joachim Fischer, Michael Piefel, and Markus Scheidgen. A metamodel for SDL-2000 in the context of metamodelling ULF. In *SAM 2004*, LNCS. Springer-Verlag, 2005.

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

[9] ITU-T Z.130. *Extended Object Definition Language (eODL)*. International Telecommunication Union, July 2003.

[10] James Martin and James J. Odell. *Object-Oriented Methods: A Foundation*. Prentice Hall PTR, 1995. 2nd edition (1997).

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

[12] MOF. *Meta Object Facility, Version 1.4*. Object Management Group, March 2003. formal/2002-04-03.

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

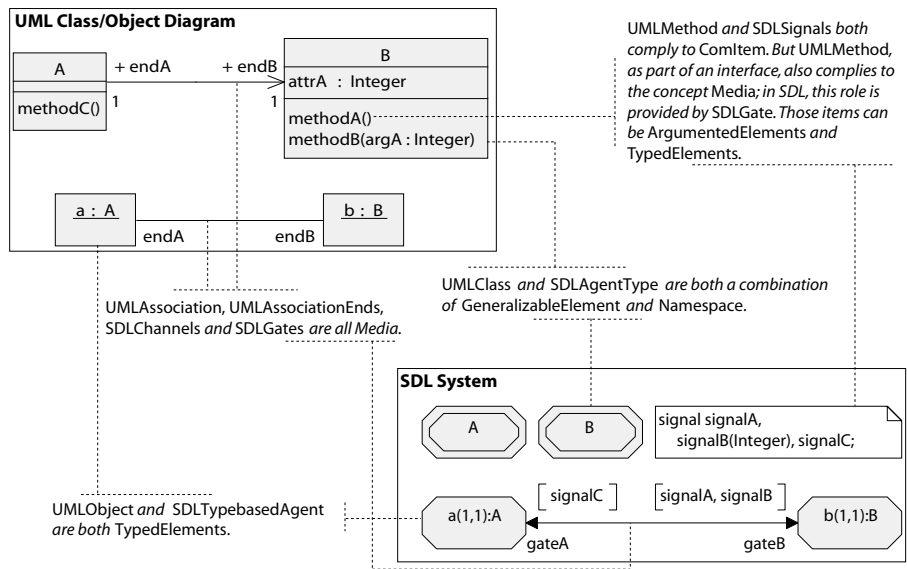[14] UML. *Unified Modeling Language, Version 1.5*. Object Management Group, March 2003. formal/2003-03-01.

UML Class/Object Diagram

A
+ endA   + endB
methodC()
1

B
attrA : Integer
methodA()
methodB(argA : Integer)
1

a : A   endA   endB   b : B

UMLMethod *and* SDLSignals *both comply to* ComItem. *But* UMLMethod, *as part of an interface, also complies to the concept* Media; *in SDL, this role is provided by* SDLGate. *Those items can be* ArgumentedElements *and* TypedElements.

UMLClass *and* SDLAgentType *are both a combination of* GeneralizableElement *and* Namespace.

UMLAssociation, UMLAssociationEnds, SDLChannels *and* SDLGates *are all Media.*

SDL System

A   B

signal signalA,
    signalB(Integer), signalC;

a(1,1):A   gateA

signalC   signalA, signalB

b(1,1):B   gateB

UMLObject *and* SDLTypebasedAgent *are both* TypedElements.

Figure 4: Common Concepts among SDL's and UML's Concepts

Statechart for Class B

A

methodB (attrA) [attrA!=1]

State, Start, Stop *are the same in both languages.*

methodB / endA.methodC()   methodA

This Selector *is a combination of an* UMLMethedCallEvent *and a* Guard .

B

methodB (attrA) [attrA==1]

SDLVariable *as well as* UMLAttribute *are* TypedElements *and part of the surrounding* Namespace

The UMLMethodCall *and the* SDLOutput *are* Action *and also* ComSource.
The UMLMethodCalledEvent *and the* SDLInput *are* Trigger *and also* ComSink.

Process Type B

dcl varA Integer;

A

B

signalB(varA)

signalB   signalA   varA

signalC via gateB   B   else   1   A   B

signalA, signalB   signalC

gateB

The SDLDecision *is a* Split, *a* PseudoState *that uses* Answer *as* Selector *for the two successor transitions.*
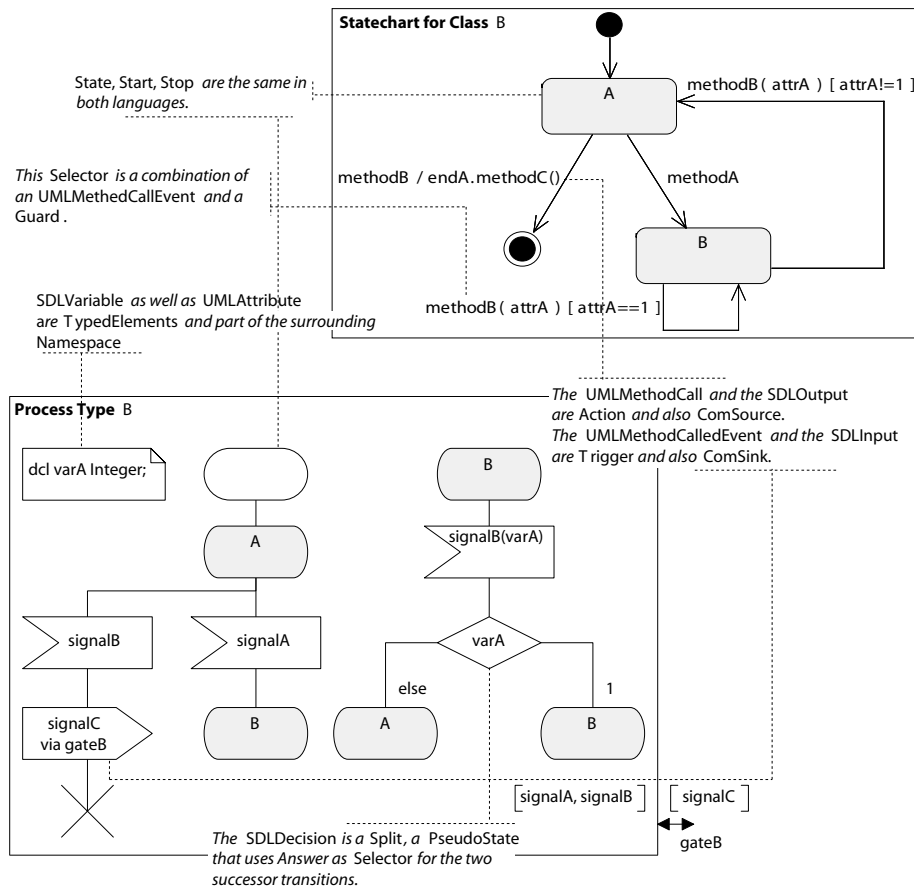
Figure 5: Common Concepts among SDL's and UML's Statechart Concepts