Adopting Meta-modelling for ITU-T Languages: Language Tool Prototypes as a by-Product of Language Specifications

Markus Scheidgen

Department of Computer Science, Humboldt Universität zu Berlin
Unter den Linden 6, 10099 Berlin, Germany
{scheidge}@informatik.hu-berlin.de

Abstract. Grammars have been used to describe computer languages since the age of the first programming languages. But the kind of languages that we use in modern engineering has changed drastically over the last decade. Modern computer languages come in a variety of forms and their characteristics demand more from a description technique than grammars can offer. In this paper, we examine two widely different approaches to language descriptions: grammars and object-oriented modelling. We argue about language requirements and that grammars fail where object-oriented modelling shows new efficient ways to not only describe languages, but also prototype language tools. In this paper we briefly compare grammars and meta-modelling based on their advantages and disadvantages and show how meta-modelling can be used to automatically develop language tool prototypes from language specifications.

1 Introduction

As a language description technique, grammars were used for a long time and we know them pretty well. Meta-modelling on the other hand is a new promising technique that still needs a lot of research to exploit its full potential and define its boundaries. In this position paper, we collect arguments contra grammars and pro meta-models based on requirements originating in modern computer languages. One of the more important pros for meta-modelling are the many existing frameworks that can be used to describe most language aspects and generate language tools automatically. We put a special interest in this generative engineering for language tools and use a big part of this paper to illustrate how meta-modelling can be used to achieve automated development of language tools prototypes. The goal of this paper is to provide arguments for meta-modelling as a language description technique for ITU-T languages.

Before we can even begin with an introduction, we need to characterise computer languages and create a framework of definitions that allows us to clearly communicate about the subject. This is especially important, because we discuss the terms of two communities in this paper. With the given definitions, we introduce context-free grammar-based language descriptions, put this traditional

technique into the context of modern computer languages to characterise existing limitations, and finally we introduce object-oriented meta-modelling (OOMM). The main part of the paper will give an overview of OOMM techniques and how they can be used to overcome grammar limitations. We apply OOMM techniques to describe the ITU-T language SDL to show the range of language aspects that can be described and how OOMM supports the development of language tools. We close the paper with related-work and conclusions.

2 Computer Languages

Languages are means to convey information to something. *Computer languages are all those languages that are used to convey information to a computer.* Not surprisingly, this intuitive definition describes computer languages as a means for communication. Because one of the communicating partners is a computer system, each utterance of a computer language has to have formally defined structure and meaning: each computer language utterance must be processable by a finite automaton (the theoretical model of a computer system) and its meaning must be free of ambiguities. As long as there is no confusion with another kind of language, we will refer to computer languages simply as languages. In the following we define what languages and language utterances are; we discuss the structural aspects of languages, the syntax of language utterances; finally, we address the representation and meaning of language utterances.

2.1 Languages

As in formal language theory, a language as a set of language utterances. Because this set forms a class or group of elements with common characteristics, we call these utterances *language instances* and define them as follows:

Definition 11 *A* language instance *is a well defined representation for a piece of information.*

To achieve this well *definedness*, we introduce the concept of *language descriptions*. At this point we do not want to elaborate on how this concept can be realised, we just accept that there are language descriptions that are conform with the following definition:

Definition 12 *A* language description *is a finite system of rules that describes what constitutes the valid instances of the described language. Therefore, a language description is a means to generate all the valid instances of the described language by accepting valid instances.*

Finally, we can define what a language is:

Definition 13 *A* computer language *(or simply* language*) is a set of language instances generated by a language description.*

2.2 Syntax

A language instance is not a monolithic piece, it has a structure and is constructed from smaller parts. We call these parts *language constructs*.

Definition 14 language construct *are the building blocks for language instances. A language comprises several language constructs; language constructs are related to each other.*

Language constructs have to be well defined within a language description. In other words, a language description comprises a collection of *language construct definitions* (it is a system of rules).

Definition 15 *A* language construct definition *defines a language construct. A construct definition also relates the defined construct to other constructs of the same language.*

A language instance is built up from these language constructs. A language instance consists of *construct instances*. Whereby, each of these instances instantiates a construct definition. Construct instances might be connected with each other according to the relations between corresponding construct definitions.

Definition 16 *A* language construct instance *is a single occurrence of a language construct within in a language instance. In accordance to the corresponding language construct definition, construct instances can be connected.*

Construct instances and their connections form the structures that constitute language instances. We call such a structure the *syntax* of a language instance.

Definition 17 *The* syntax *of a language instance is a representation of this instance that reflects its structure.*

The concepts syntax and language instance are closely related; sometimes both terms describe the very same thing. As we will see later, OOMM uses graph-structures as language instances. In this case, the language instance and its syntax (the graph-structure itself) are the same thing. When we look at textual languages and grammars on the other hand, a language instance is a piece of text and its syntax is a tree that describes how the text was constructed from grammar rules. Here language instance and syntax are different things.

2.3 Representation and Semantics

A language instance must have more than just plain syntax; it has to have meaning to convey information. Now we can look at *meaning* from two sides: a language instance can mean something, or it can be the meaning of something. We defined a language instance as a representative of a piece of information. To this end a language instances means something: this something is the information it represents. On the other hand, a language instance itself is a piece

of information, and hence can be the meaning of another language instance (of another language). We distinguish between *semantics* and *representation*. Semantics is the meaning of a language instance, and *representation* is another language instance that represents the language instance. These concepts only differ in the point of view and we use the same means to realise these concepts.

Definition 18 *A language instance is only a representation of something (e.g. a statement, expression, command, program, software model, formula, etc.). The meaning of a language instance is the something it represents. This something is the* semantics *of a language instance.*

Definition 19 *A language instance (a piece of information) can be represented by another language instance (of another language). This other language instance is a* representation *for the language instance. A language with instances that are representations for the instances of another (the notated) language is called* notation *for this other language. The semantics of notation instances are the instances of the notated language.*

The semantics of a language or notation is defined by a semantic mapping and a semantic domain. Often the semantic domain is just another language.

Definition 110 *A* semantic domain *is a set. A* semantic mapping *is a relation that associates language instances of a language (range) with elements of a semantic domain (domain). A* semantics description *is a finite description; it comprises a description for the semantic mapping and either a description for the semantic domain or a reference towards an existing semantic domain.*

As explained, a language instance conveys information, therefore it is necessary that is has well defined semantics. But does a language instance always needs a representation? To answer this question, we have to distinguish between concrete language instances and abstract language instances.

Definition 111 *A* concrete language instance *is a language instance that has a concrete, physical, tangible form. An* abstract language instance *is a language instance that does not have a concrete, physical, tangible form. It can be represented by many concrete entities (e.g. concrete language instances); an abstract language instance represents all the possible representing concrete language instances.*

A language instance is used to convey information by exchanging language instances (as representatives of information) between two communicating entities. An abstract language instance can not be exchanged because it is not a tangible object. But, it is not necessary to transfer a possibly abstract language instance itself, we can always use a concrete representation instead. Conclusively, if we want to use a language that consists of abstract language instances, we need a notation consisting of concrete representations to use this language.

3 Comparing Grammars and Meta-Modelling

3.1 Grammar-based Language Descriptions

The most common form of language descriptions are grammar-based language descriptions (part of the disciple called *formal language theory*). Grammars were established as a computer language description technique with the first programming languages and extensively used throughout the last decades. Grammars are a specific form of language descriptions for a specific type of computer language, namely textual programming languages. Textual (programming) languages are addressed in formal language theory:

Definition 112 *A* textual language instance *is a language instance that comprises of a sequence of symbols from some set of symbols (alphabet). A* textual language *is a language that consists of textual language instances constructed over a single set of symbols (alphabet).*

Grammars are a specific form of language descriptions for textual languages:

Definition 113 *A* grammar *is a specific form of language description used to describe textual languages. A grammar is a system of rules over a set of symbols (alphabet) that describe which possible sequences of symbols constitute valid textual language instances.*

A grammar can be used in two ways and we can derive two types of tools or theoretical tool models (automata) form a grammar. First, a grammar describes which language instances are members of a language. This allows to construct a recogniser for the described language. A *recogniser* is a finite automaton that consumes potential textual language instances and either accepts or rejects them. Second, a grammar describes how to analyse the structure of language instances. This allows to construct a parser. A *parser* is a finite automaton that consumes language instances and produces the language instances' syntax. A language instance can have several syntaxes based on the same grammar, because grammars can be ambiguous.

Formal language theory distinguishes between four nested classes of textual languages with four corresponding types of grammars (known as Chomsky hierarchy). The according grammars can be realised by corresponding automata, which themselves fall into four classes. The most complex language class that still has a corresponding automaton type simple enough to be practically implementable for today's computers is the class of context-free languages. Therefore, the context-free grammar formalism (or formalisms based on context-free grammars) is almost exclusively used to define grammars for textual languages.

At this point the details about context-free grammars are not important, but their implications on language description expressiveness, general form of corresponding syntaxes, and implications on depending semantics descriptions are. Context-free grammars and the languages that can be generated have the following limitations.

- Grammar-based language definitions describe tangible language instances. These language instances are directly used to notate the language. And grammars are not suited to distinguish between the language and different notations.[1]
- Context-free grammars only allow syntaxes that have term structure. This is too weak, even for the textual programming languages that grammars are used for. Especially references in programs (e.g. the reference from a variable usage to an according variable declaration) can not be represented in a language instance. As a conclusion, grammar-based language definition almost always have to be augmented with further descriptions, e.g. name-tables or other static semantics concepts.
- Context-free grammars use simple rules to define the constructs of the language. These rules cannot be explicitly defined as generalisations of each other, and it is hard to express abstractions within a language definition. As a result, inheritance or other forms of reuse between more abstract and more concrete language construct definitions are not possible. Furthermore, grammars do not provide modularisation concepts, and grammars do not allow to define auxiliary constructs or construct properties that could be used for related descriptions (e.g. notations and semantics).

3.2 Language Description Requirements for Modern Computer Languages

How much do the limitation of grammars effect the description of modern computer languages? Modern software engineering does not solely rely on textual programming languages anymore. Methodologies like *Model Driven Development*, *Product Line Development*, or *Domain Specific Languages* rely on language with far more diverse characteristics. Furthermore, we use large number of languages within a single software development project: we use different languages to describe the problem domain or software product in different the project phases and on different abstraction levels. As a result, modern computer languages have the following characteristics:

- They are not all textual languages. We have languages that are notated in text, tables, or diagrams. Thereby, a single language instance is often represented by a number of texts, tables, or diagrams.
- Languages might use several notations. This can mean that a whole language instance can be represented in different notations, or that different parts of a language instance are represented in different notations.
- Several instances of different languages describe the same thing on different abstraction levels or from different views. As a result, the constructs of different languages are logically related to each other.

[1] Anyhow, grammar-based language definitions distinguish between concrete and abstract syntax, but the corresponding grammars are very similar and basically enforce the same structures with in the concrete and abstract syntax of a language instance. As a conclusion, grammar-based language definitions are only feasible for languages with exactly one textual notation.

- More general languages are used in more specific contexts and have to be specialised for this purpose. This requires means to specialise language constructs or create language profiles to alter language representation and semantics.
- Very specialised languages with a narrow set of constructs that are only used within one domain (Domain Specific Languages) or even for just a single project.

Not only the character of computer languages has changed, but also the habits of using them and how these languages are realised in language tools. Where in the past a plain text editor and compiler was satisfactory, we now demand integrated development environments (IDE) that combine the tools of all used languages. We switch between different views on our software, between different abstraction levels. Changes made to one part of a software system description automatically changes others. Characteristics of modern computer language tools are:

- The efficiency and quality of software development depends on the quality of language tools as much as it depends on the quality of the used languages.
- Highly specialised languages, such as DSLs, require efficient development of language tools to be economically sustainable.
- Specialised languages also require to change a language often. When the domain changes, the DSLs has to change as well.
- Tools need to be integrated; language instances are exchanged between tools of different vendors; changes in one artefact inflict automatic changes in others.

From those characteristics of computer languages and their usage, we can derive a set of requirements for language descriptions.

- Language instance and representation have to be two separate things. Language descriptions must allow the definition of several notations for a language. Different forms of notations have to be combined.
- Language description techniques, include techniques for the description of language, notations, and semantics must provide the means to express abstractions within language constructs. Language construct definitions should form specialisation hierarchies, including the reuse/inheritance of construct characteristics and related descriptions for representation and semantics.
- Language description must allow the development of language tools. Therefore, it is critical that language description techniques are well align with predominant programming paradigms, especially object-orientation.
- Language descriptions must allow the efficient development of language tools and prototypes.
- It must be possible to combine different language description. The description techniques must allow to relate language constructs of different languages. Language descriptions must facilitate the description of mappings between different languages.

3.3 Object-oriented Language Description Techniques

There are several approaches that issue the limitations of context-free grammars. Examples are graph-grammars and attributed grammars. It is not the goal of this paper to evaluate all these approaches to find the ultimate replacement technology for context-free grammars. In this paper, we look at one specific technology, and examine to what extend this technology can fulfil the listed requirements and what the future potential of this technology is.

This technology can be called *object-oriented meta-modelling* (OOMM); we will call is *MOF-like meta-modelling* based on the MOF OMG recommendation. OOMM uses object-oriented structure models to describe what possible language instances are. In the same way a context-free grammar (in the following just grammar) uses rules and symbols, a MOF-like meta-model (in the following just meta-model) uses classes and properties to define structures (language instances). But MOF-like meta-modelling promises the following advantages over context-free grammar based language descriptions:

- A meta-model describes language instances independent from their notation. The described language instances are graph-structures: an instance and the instance's syntax are the same thing. This allows to define several independent notations and all forms of notations for a language.
- Meta-models describe graphs and not trees. This makes meta-models more powerful: for example, references in textual languages can be modelled or languages that are notated by diagrams (graphs themselves).
- Meta-models are object-oriented and abstractions between constructs can be expressed. Classes can form specialisation hierarchies and properties can be inherited. As a conclusion abstract language constructs can be reused. This is not only imported for the meta-models themselves, but also for the language tool design based on this models. Furthermore, meta-models can be modularised.

4 Generative Engineering of Language Tool Prototypes

The most important aspect of language engineering is the development of language tools, because the tools are prerequisite to use a language. Therefore, a language specification needs to address efficient tool development. State of the art language description techniques (and OOMM in particular) can be used for the generative engineering of language tools.

4.1 Generative Engineering

In this paper, we focus on the generation of language tool prototypes from language specifications. In this section we want to look at generative engineering[2]

[2] This is also often referred too as Domain Specific Modeling[1] or Model-Driven Software Engineering[2].

in general. Instead of manually programming similar programs again and again, generative engineering uses specialised description languages to generate the instances of a class of similar computer programs. This raises the level of abstraction and reduces development efforts because you describe the things that are specific for a concrete computer program and neglect the things that are common for the whole class of similar programs.

Description languages are intended to describe a specific class of things. A description language for textual notations is used to describe the concrete syntax of textual computer languages, or a description languages for operational semantics is used to describe the execution of instances of a programming language. Descriptions in general can be used to generate computer programs. This requires a generator based on a specific description language. The generator can create a specific class of computer programs based on the specific class of things that the description language can describe. Diagram editors for graphical computer languages, for example, are very similar. They provide a drawing canvas and a tool box. Users can select tools and create graphical items. They can select objects and move or delete them. All these features are included in all graphical editors. The only thing that is language specific is the used notation: the used symbols and possible connections. With generative engineering a language developer only describes the notation and generates the editor with all its common characteristics from that description.

A description can be unambiguous or ambiguous: it can either describe exactly one thing or it describes, deliberately or not, two similar but different things. A description language can have unambiguous or ambiguous semantics: all instances of a unambiguous description language are unambiguous; some instances of an ambiguous description language are ambiguous. Only unambiguous description languages are reasonable for a generator framework, because only those guarantee that the generated computer program reflects the intended meaning.

A description language can use constructs that are either similar or completely different to the computational concepts of a target platform. This influences whether or not a generator framework for this description language and this target platform is feasible or not. For example, a description language that relies on abstract mathematical constructs that are hard to realise on a computer platform makes it hard to create a generator for that language. This usually corresponds to the level of abstraction that a description language facilitates. The more abstract a description is, the more details a generator has to create by itself. Since a generator is written for a description language and not specific for each description, the abstract to concrete mapping solutions that a generator provides are generic for all descriptions. This either makes it impossible to develop a generator, because creating such details requires more intelligence than one can put into such a generator, or the generator creates computer programs that do not perform well enough. As a general rule of thumb, the more abstract a description language is relative to the target platform, the more intelligence has

to be put into the generator and the more generic and therefore less performing are the generated computer programs.

As a conclusion, description languages for generator frameworks have to be reasonably concrete. This presents a trade-off: on one hand a description should be as abstract and as small as possible, on the other hand a description has to allow efficient (automatic at best) development of performing computer programs. Therefore, the description languages of today's generator frameworks present a compromise. They provide constructs on a fairly high abstraction level. This allows to realise a small amount of the most frequently appearing use cases, but does not cover the very special and therefore seldom details. The argument behind this strategy is that one can describe the bigger part of what one needs to describe and for all the specialities one has to leave the realm of the description language and use a different technique. Most generator frameworks therefore allow to augment the instances of specific description languages with pieces of code written in multi-purpose programming languages. This renders the development of corresponding computer programs at least partially automatic. As a result, generator frameworks are very popular for the development of prototypes that do not necessarily need to contain all the specific details that are required for the final software product.

4.2 Generative Engineering for Language Tools

What does one need to use a language? Two things. First, a language specification that allows potential language users to learn and use the language. Second, language tools: editors that allow language users to create language instances, analysers, code-generators, or interpreters that allow to process and execute language instances. Besides describing a language to humans, generative engineering allows to use a language description to automate the creation of language tools. Where a language tool is a tool that processes language instances or representations of language instances.

Language tools fall into several categories and so do descriptions for languages. A language description determines what the instances of a language are. Besides describing the language itself, we also need to describe possible notations and semantics. We distinguish between the *language aspects* language, notation, and semantics. We already showed that the aspects notation and semantics require the description of other languages (notation or semantic domain) and the description of mappings between notation/semantic domain and language. For each different language aspect we use different description languages and we generate different kind of tools.

A description itself, no matter if it is a language description or mapping description, is just a piece of information. A way to represent this information is in a language specialised for this kind of information. This becomes confusing, if we neglect to strictly distinguish between different levels of description and described object. Therefore we define:

Definition 114 *A* meta-language *is a computer language used to describe other computer languages.*

It is not necessary to write a description in a computer language, but, as we will see later, describing a language in a computer language allows us to process language descriptions. Specific computer programs allow to generate language tools that in turn allow us to process instances of the described language. We need to make the same distinction between language tools, and programs that generate language tools (meta-language tools):

Definition 115 *A* meta-tool *is a computer software that generates language tools based on a description written in a corresponding meta-language.*

The relationship between meta-language and meta-tool is the same as between language and language tool. While language tools that realise semantics might simulate, analyse, or compile language instances, meta-tools generate language tools from meta-language instances. The only difference between a common computer language and meta-language (the same holds for tools and meta-tools) is its purpose. A meta-language is just a computer language with the specific purpose to describe other computer languages. Meta-language and corresponding meta-tools form generator frameworks in the sense of generative engineering.

5 OOMM and Generative Engineering of Language Tools for SDL

In this section, we want to illustrate the usage of OOMM language descriptions and generative engineering for the specification of SDL and the development of prototypical SDL tools.

5.1 A meta-model for the SDL language

A meta-model is just an object-oriented model of SDL's language constructs. We derived a SDL meta-model from SDL's abstract syntax grammar in [3]. Existing OOMM meta-modelling languages are all very similar, so the SDL meta-model is available in the important MOF dialekts: MOF 1.x, CMOF, EMOF (the two meta-modelling languages of the MOF 2.x), and eclipse's ecore.

The SDL meta-model is the basis for the description of other language aspects, like SDL's notations and its operational semantics. The SDL meta-model is a visual specification of SDL's concepts and structure. The SDL meta-model thereby specifies a set of valid SDL models. It allows to store and exchange SDL specifications based on XMI.

5.2 SDL's notations

SDL has two different notations that one can use to represent SDL specifications. These are a graphical notation and a textual notation, whereby the graphical SDL notation is highly interspersed with elements of the textual notation. In [4],

we used GMF[5] to describe SDL's graphical notation. With the GMF generator framework we can automatically generate a graphical SDL editor that can be used to create instances of the SDL meta-model. While GMF allows to define most of SDL's graphical notation out of the box, there are detailed features that require manual (programmed) augmentations to the notation description. This emphasises that GMF is definitely a technology capable of generating useful editors, but if you have a concrete graphical notation that uses elements not common among other graphical languages, you need to manually augment the generated editors. An example SDL feature hard to realise with GMF is the distribution of an SDL state automaton specification over multiple diagram pages, or to add gates to diagram pages (i.e. have connections with an open end).

SDL's textual syntax can be realised with generator frameworks like TEF [6] or xText [2]. In these frameworks, a notation description relates a regular context-free grammar that describes the textual notation with the SDL meta-model. From such descriptions one can generate text editors with features like syntax highlighting, code-completion, and error-annotations. TEF's ability to combine graphical GMF-based editors with a TEF generated text editors [7] can be used to integrate textual editing for the extensive textual parts of SDL's graphical notation.

5.3 SDL's operational semantics

SDL's operational semantics is already formally defined based on Abstract State Machines. This semantics can be adopted to work on meta-model-based SDL specifications as well. In fact, the structure of operational semantics descriptions is always the same. One needs a description language to describe a set of configuration. Here, both grammars and meta-models are suitable description languages, because both describe a set of elements. The grammar or meta-model for SDL's syntax merely has to be augmented with structures that can store the additional runtime-information that is necessary to hold the state of a running SDL specification.

Once all possible configurations are specified, transitions between configurations can be described more or less independent of the configuration description technique. Here, we can reuse the existing ASMs, which basically describe updates on the actual configuration (transition) based on conditions within this actual configuration. There are several languages that can be facilitated to describe transitions between meta-model-based configurations. In [8], we used UML activities and OCL to describe operatinal semantics. In [9] the authors provide a platform that allows to choose from scheme, prolog, QVT, or ASMs to define the operational semantics for a meta-model. In [4] we already described how to use the existing SDL ASMs on the basis of an SDL meta-model.

6 From generated prototypes to industry strength language tools

We used the eclipse platform and its OOMM technology EMF to create prototypical SDL tools based on a SDL meta-model. Due to the limitations of the used generator frameworks (GMF for SDL's graphical notation, TEF for SDL's textual notation, MAS (semantics described with UML activities and OCL) for SDL's semantics), these prototypical SDL tools can only be prototypes and lack the quality that we are used from hand crafted, industry strength SDL tools. For example, the generated graphical editors only reflect a rough approximation of SDL's actual notation; the generated simulator for SDL specifications does not nearly show the performance of generated code, because the simulation is driven by the interpretation of UML activities and OCL expressions. Furthermore, some aspects of SDL already had to be implemented by hand: static analysis features like identifier resolution for example could not be described with the used generator frameworks, and already required a considerable amount of programming work.

However, most of the used generator frameworks and existing generator frameworks allow to deal with this limitations by implementing troublesome language features manually. This combination of language description on a high-level of abstraction and implementations in a multi-purpose programming language allows to create industry strength tools with lower development efforts. The generator frameworks hereby provide a series of advantages: one can has a running product very early in the development process (this can be used to implement an agile development strategy); generator frameworks provide a reasonable tool design and thereby reuse this design for all tools of the same kind; generator frameworks are enhanced all the time and each enhancement of the framework automatically enhances all tools described in this framework.

7 Related Work

There are several *language workbenches* that combine generative engineering frameworks for languages based on OOMM that cover multiple language aspects like notation, analysis, transformations, or operational semantics. These workbenches are GME [10], XMF [11] (originated in the MMF approach [12]), AToM3 [13] and meta-programming facilities like MPS [14], kermeta [15], AMMA [16] MetaEdit+ [17]. Some of these frameworks define semantics through general purpose programming languages (MPS, MetaEdit+), others provide specialised languages to define semantics (XMF, kermeta, AToM3). Two different approaches to semantics can be identified: GME and AToM3 use model transformations into a different language or formalism (semantic domain). AMMA, Kermeta, XMF, and MPS use an action language to define operational semantics.

There are several approaches using a specific meta-language for the definition of operational semantics. In [18] Engels et al present a graphical modelling

approach for UML semantics based on collaboration diagrams and graph transformations. This approach provides strong mathematical foundations, but results in very verbose semantic rules, which are hard to read and execute. In [16] Abstract State Machines (ASM) are integrated into the DSL framework AMMA to support specification of execution semantics for DSLs, using ASMs as just another DSL. Muller et al [19] use a textual action language in combination with OCL for high level semantics descriptions. This action language is executable and provides the foundation for the DSL framework kermeta [15]. A similar approach is used in Mosaic [11] which uses an OCL version extended with actions to define language semantics. We recycled the idea of using OCL for expressive model navigation in our approach. In [20] Gerson Sunyé et al explore the possibility of UML action semantics [21] to create executable UML models and already suggest the use of activities with action semantics for meta-modelling. We use this idea and reduce the set of actions to those necessary to describe operational semantics based on model changes.

Along with structural operational semantics [22], semantics are traditionally defined based on grammars for abstract syntax. A formalism, like term re-writing, is used to describe manipulation of abstract syntax trees (AST are instances of grammars). This describes interpretation of an input program represented by an AST. The formal SDL semantics definition [23] uses Abstract State Machines (ASMs) to realise a similar approach: it defines abstract syntax and runtime states with grammars and represents corresponding ASTs as evolving algebras manipulated by ASMs. Our approach replaces grammars/signatures with meta-models, ASTs/algebras with models, and re-writing/ASMs with our combination of activities, OCL, and actions.

The idea to create textual notations based on meta-models is as old as meta-modelling itself. Work based on meta-modelling with MOF: Alanen et al [24], Scheidgen et al [3], Wimmer et al [25]. This basic research was later utilised in frameworks for textual model editors. These are either based on existing meta-models (TCS [26], TCSSL [27], MontiCore [28]), or they generate meta-models or other parse-tree representations generated from the notations (xText [2], Safari [29]). Besides using context-free grammars and parsing, textual modelling can also be conducted using the model-view controller pattern. The same pattern is used to build graphical model editors. MVC for textual notations is used in intentional programming [30] and the meta programming system [31].

The GMF framework itself, provides some very simple means to describe structured text. It allows to create simple templates that assign different portions of a text to different object features. These simple templates allow less than regular languages, and are therefore inadequate for many textual language constructs. Further attempts to describe the relations between graphical notations and textual notations have been made by Tveit et al [4].

8 Conclusions

Grammars have been successfully used for decades, but recent trends in computer languages (e.g. graphical languages) and language tool development (e.g. generative engineering) give reason to reconsider. OOMM in particular provides interesting properties. OOMM is well aligned with object-oriented programming platforms that are usually used to realise language tools and therefore allow for easier generative engineering. OOMM's independence of language instances and their representation render it more suitable for the specification of graphical languages. Other advantages over grammars are better (quasi-) standardisation and conclusively better tool support and higher interoperability.

OOMM and generative engineering are successfully used for domain specific languages in many instances, and recent case-studies for more complex languages like SDL and UML show OOMM's potential as a specification technique for ITU-T languages.

References

1. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling. Wiley-IEEE Computer Society Press (2008)
2. Homepage: openArchitectureWare See `http://www.openarchitectureware.org`.
3. Fischer, J., Piefel, M., Scheidgen, M.: A metamodel for sdl-2000 in the context of metamodelling ulf. In Amyot, D., Williams, A.W., eds.: SAM. Volume 3319 of Lecture Notes in Computer Science., Springer (2004) 208–223
4. Prinz, A., Scheidgen, M., Tveit, M.S.: A model-based standard for sdl. In Gaudin, E., Najm, E., Reed, R., eds.: SDL Forum. Volume 4745 of Lecture Notes in Computer Science., Springer (2007) 1–18
5. Homepage: Graphical Modelling Framework (GMF) See `http://www.eclipse.org/gmf/`.
6. Homepage: Textual Editing Framework (TEF) See `http://www.informatik.hu-berlin.de/sam/meta-tools/tef`.
7. : Textual modelling embedded into graphical modelling
8. Scheidgen, M., Fischer, J.: Human comprehensible and machine processable specifications of operational semantics. In Akehurst, D.H., Vogel, R., Paige, R.F., eds.: ECMDA-FA. Volume 4530 of Lecture Notes in Computer Science., Springer (2007) 157–171
9. Sadilek, D.A., Wachsmuth, G.: EProvide 2.0: an Extensible Framework for Describing Operational Semantics
10. Agrawal, A., Karsai, G., Ledeczi, A.: An End-to-End Domain-Driven Software Development Framework. In: OOPSLA '03: Companion of the 18th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press (2003)
11. Clark, T., Evans, A., Sammut, P., Willans, J.: Applied Metamodeling, A Foundation for Language Driven Development. Xactium (2004) http://www.xactium.com.
12. Clark, T., Evans, A., Kent, S., Sammut, P.: The MMF Approach to Engineering Object-Oriented Design Languages. In: Workshop on Language Descriptions, Tools and Applications. (April 2001)

13. The Modelling, Simulation and Design lab (MSDL), School of Computer Science of McGill University Montreal, Quebec, Canada: AToM3 A Tool for Multi-Formalism Meta-Modelling. http://atom3.cs.mcgill.ca/index.html.
14. Dmitriev, S.: Language Oriented Programming: The Next Programming Paradigm. onBoard, electronic monthly magazin (November 2004)
15. Team, T.: Triskell Meta-Modelling Kernel. IRISA, INRIA. www.kermeta.org.
16. Ruscio, D.D., Jounault, F., Kurtev, I., Bézivin, J., Pierantonio, A.: Extending AMMA for Supporting Dynamic Semantics Specifications of SDLs (2006) technical report.
17. Case, M.: MetaEdit+. http://www.metacase.com.
18. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In: UML 2000 - The Unified Modeling Language. Advancing the Standard.
19. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving Executability into Object-Oriented Meta-languages. In: Model Driven Engineering Languages and Systems: 8th International Conference. LNCS, Springer (2005)
20. Sunyé, G., Pennaneac'h, F., Ho, W.M., Guennec, A.L., Jézéquel, J.M.: Using UML Action Semantics for Executable Modeling and Beyond. In: 13th International Conference on Advanced Information Systems Engineering. LNCS, Springer (2001)
21. OMG: Action Semantics for the UML. Object Management Group (2001) ad/2001-08-04.
22. Plotkin, G.D.: A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus (1981)
23. ITU-T: SDL formal definition: Dynamic semantics. In: Specification and Description Language (SDL). International Telecommunication Union (November 2000) Z.100 Annex F3.
24. Alanen, M., Porres, I.: A Relation between Context-Free Grammars and Meta Object Facility Metamodels. Technical report, TUCS (2004)
25. Wimmer, M., Kramler, G.: Bridging grammarware and modelware. In: Satellite Events at the MoDELS 2005 Conference. (2006) 159–168
26. Jouault, F., Bézivin, J., Kurtev, I.: Tcs:: a dsl for the specification of textual concrete syntaxes in model engineering. In: GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering, New York, NY, USA, ACM Press (2006) 249–254
27. Muller, P.A., Fleurey, F., Fondement, F., Hassenforder, M., Schneckenburger, R., Gérard, S., Jézéquel, J.M.: Model-Driven Analysis and Synthesis of Concrete Syntax. In: Proceedings of the 9th International Conference, MoDELS 2006. (2006) pp. 98–110
28. Krahn, H., Rumpe, B., Völkel, S.: Integrated definition of abstract and concrete syntax for textual languages. In Engels, G., Opdyke, B., Schmidt, D.C., Weil, F., eds.: MoDELS. Volume 4735 of Lecture Notes in Computer Science., Springer (2007) 286–300
29. Charles, P., Dolby, J., Fuhrer, R.M., Stanley M. Sutton, J., Vaziri, M.: Safari: a meta-tooling framework for generating language-specific ide's. In: OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, New York, NY, USA, ACM (2006) 722–723
30. Simonyi, C.: The death of computer languages, the birth of Intentional Programming. Technical report, Microsoft Research (1995)
31. Dmitriev, S.: Language Oriented Programming: The Next Programming Paradigm. onBoard (1) (November 2004)