MDE Basics with a DSL Focus

Suzana Andova, Mark G.J. van den Brand, Luc J.P. Engelen, and Tom Verhoeff

Eindhoven University of Technology, Den Dolech 2, NL-5612 AZ Eindhoven
{s.andova,m.g.j.v.d.brand,l.j.p.engelen,t.verhoeff}@tue.nl

Abstract. Small languages are gaining popularity in the software engineering community. The development of MOF and EMF has given the Domain Specific Language community a tremendous boost. In this tutorial the basic aspects of model driven engineering in combination with Domain Specific Languages will be discussed. The focus is on textual Domain Specific Languages developed using the language invention pattern. The notion of abstract syntax will be linked to metamodels as well as the definition of concrete syntax. Defining static and dynamic semantics will be discussed. A small but non trivial Domain Specific Language `SLCO` will be used to illustrate our ideas.

1 Introduction

Our society has become completely dependent on software. We do our bank transactions via the internet, we book our holiday trips, order books, etc. on line, and we submit our tax forms electronically. Medical information is exchanged electronically between doctors. Software is no longer running only on "traditional computers", but is incorporated into products that we use in daily life like mobile phones, personal organizers, game computers, personal care equipment, and even cars. A modern car contains about 10 million lines of code; a wafer stepper is run by over 30 million lines of code. Apart from this code explosion, people put more trust in software. In fact, we have become software dependent, *all* modern devices are software based.

The software engineering community is facing two main challenges; how to produce and maintain this huge amount of software written in a broad variety of (programming) languages and how to guarantee the correctness of the resulting software. Increasing the level of abstraction seems to be a logical solution. This is in line with the development of programming languages over time, see Section 1.1. In the last decade (graphical) models have become popular when developing software and new formalisms such as the Unified Modeling Language (UML) have been defined, see Section 1.2. These new formalisms have created a new field within software engineering: *model driven (software) engineering* (MDSE) which may be a solution but it may also be the next "no silver bullet" [24]. The use of high level models offers also the possibility to perform model analysis and verification of properties. This is beneficial for any sort of systems but is crucial for safety critical systems.

M. Bernardo, V. Cortellessa, and A. Pierantonio (Eds.): SFM 2012, LNCS 7320, pp. 21–57, 2012.

1.1 From Low Level to High Level Programming Languages

In the middle of the previous century, computers where introduced to perform calculations for ballistic missiles. Soon after this, the first commercial software applications were developed [27]. In 1951, Grace Hopper wrote the first compiler, A-0, see [94]. She was also involved in the development of the first compiler-based programming languages, including ARITH-MATIC, MATH-MATIC and FLOW-MATIC. A compiler is a program that transforms high-level statements in a programming language into low-level computer instructions. Since the programmer is working at a higher level of abstraction, more can be expressed in fewer lines of code. Programming has evolved enormously since then. The development can be characterized by indeed making programming easier, by introducing, for instance, high-level language constructs, and by increasing the level of abstraction, by introducing procedures and classes. The introduction of high-level control flow constructs, such as, conditionals, loops and exceptions (which replace low-level constructs like GOTOs, see [31]) has improved the quality of the software. Increasing the level of abstraction does not only lead to an improvement in productivity but also to better quality of the code.

Models raise programming to the next level of abstraction. They are in general used to design both hardware and software. Often the models are manually translated into other more detailed design documents and/or code. However, that route offers no guarantee for consistency between model, design and the resulting code and should be improved.

1.2 From Informal to Formal Modeling

Models have become important when designing software. People used to make informal drawings of the (structure of the) software when designing software. However, these informal drawings are not machine processable, and have to be converted into (source) code manually. One can observe more than once that developers take pictures of the black board to recreate the models in Visio or some other modeling tool. These informal models lack any form of semantics, or the semantics is only in the head of the designer.

Apart from these informal drawings, others advocate the use of formal methods as much as possible to describe the (behaviour of) software systems. The advantages of formal methods are their rigorousness and software developed using these methods is usually free of errors. They have well defined semantics and can be automatically processed. A number of formal methods allow the interpretation of the models, or even generation of executable code. However, the learning curve of formal methods is steep, whereas the learning curve for drawing diagrams on the black board is very low.

1.3 From UML to MOF

Quite a number of modeling languages have been developed over the years, each dealing with different aspects of software (development). Some of these modeling

languages are data oriented, e.g., E/R models; some are structure oriented, e.g., class diagrams; some behaviour oriented, e.g., use cases, state machines, sequence diagrams, activity diagrams, and others are architecture oriented, e.g., package diagrams, component diagrams.

The Object Management Group (OMG) took the initiative of unifying a number of these diagrams, among others class diagrams, state machine diagrams, sequence diagrams, into UML1.x [65]. This was done by James Rumbaugh, Grady Booch and Ivar Jacobson (the Three Amigos) [76]. UML combines Rumbaugh's Object-Modeling Technique (OMT), for Object-Oriented Analysis (OOA), and Booch method, for Object-Oriented Design (OOD), with the work of Jacobson, the Object-Oriented Software Engineering (OOSE) method.

UML, via the UML profiles, offers an extensibility mechanism that can be used to develop domain specific modeling languages [81]. The lecture on "MDE Basics with a UML Focus" by Bran Selic discussed this approach in more detail. The development of UML has also lead to the creation of the Meta Object Facility (MOF) [46], also by OMG. MOF is used to define the various modeling formalisms of UML in a uniform way. The Meta Object Facility has a four-layered architecture:

M3. The **Meta-Meta-Model Layer** contains the MOF language, which is used to describe the structure of metadata (and, also, of MOF itself). It provides a meta-metamodel at the top layer.

M2. The **Meta-Model Layer** contains definitions for the structure of metadata. The M3-model is used to build metamodels on level M2. The most prominent example is the UML metamodel, the model that describes the UML itself.

M1. The **Model Layer** contains definitions of data in the information layer. The metamodels of level M2 describe the structure of elements of the M1-layer, for example, models written in UML.

M0. The **Model Layer** contains objects or data in the information layer.

MOF is an alternative for developing domain specific (modeling) languages. Although MOF is the official standard, there are only a few implementations [78], the most popular of which is the Eclipse Modeling Framework (EMF) [25,83]. EMF has become very popular for developing Domain Specific Languages (DSLs). An entire range of Eclipse plugins have been developed to deal with the various aspects of DSL development. In this tutorial, we address the design of DSLs, and EMF among others is used as the main implementation medium, although the presentation will be as tool independent as possible.

1.4 Outline of Tutorial

The tutorial uses a small but non-trivial language called "Simple Language of Communicating Objects" (`SLCO`) as a running example. All aspects of designing and implementing a DSL will be demonstrated using this language. In Section 2, background information on DSLs in general is given. This section gives some

definitions of DSLs and a rough classification of DSLs. In Section 3, the notions of abstract syntax and metamodels are introduced. Section 4 describes the static and dynamics of a DSL. We provide some background information on static semantics and what we understand by it. The second half of this section is devoted to the formalization of the dynamic semantics. Section 5 presents ways of defining the concrete syntax of a DSL. The focus in this section is mainly on the context-free syntax and less on the lexical part. We conclude this tutorial, Section 6, by discussing the way we use `SLCO` in our research. We will also sketch a number of research directions for the DSL community.

2 Domain Specific Languages

It is hard to give a very precise definition of Domain Specific Languages or little languages [29]. In general, a language is a symbolic system for communication. More formally, a language is a collection of sentences or expressions, constructed according to certain grammatical rules, where the language elements refer to real-world entities. In this sense, a DSL is a formal, processable language targeting a specific aspect of an information-processing system or task, for instance building user interfaces, performing database queries, building web pages, exchanging data, generating scanners and parsers. There is no requirement that a DSL should be Turing complete, in contrast to a general purpose language. Its semantics, flexibility and notation are designed to support working with these aspects as efficiently as possible. Yet another definition of DSLs can be found in the annotated bibliography [30]: "A language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain".

2.1 Forms of Domain Specific Languages

Given these definitions we know DSLs are not general purpose languages but rather languages used to address problems in a restricted domain. The next question is what do DSLs look like? Mernik et.al. [62] give a detailed description of the various domain specific design patterns. The classification given in this tutorial is from a user perspective instead of a developer perspective. DSLs differ a lot in their external appearance. It all depends on which viewpoint is taken.

In a number of cases, (built-in) libraries that come with an integrated development environment or a programming language, such as the Swing library of Java, can be considered as a DSL. It contains the vocabulary of concepts that are needed to deal with specific tasks. The programs written to solve these tasks use these libraries. Adapting the DSL reduces to changing the libraries.

Another approaches is to embed a DSL in an existing language. Hagl [93] is an example of such an embedding in Haskell [49]. This way of developing DSLs is very efficient. The embedded language has the flavour of the host language and reuses the underlying parser and type system, at least in case of Haskell.

The difference with the first approach, the library approach, is not very obvious from a user perspective. Both approach can be considered as internal DSLs.

Yet another way is to add extensions to existing languages to increase the expressive power of the language. This can be considered as a partly external DSL. TOM [63] is an example of such a domain specific extension, that adds a pattern matching facility to the host language. The language constructs of the "extension" are translated to the host language Java in case of TOM. Such extensions involve quite a large implementation effort. The grammar of the host language has to be extended and in many cases a separate static analysis phase has to be developed along with finally a translator. The benefit is added expressive power, combined with the general utility of Java.

The last category are DSLs which are not built on top of, or embedded in, an existing general purpose language, but are independent DSLs. This can be considered as a fully external DSL. In Mernik et.al. [62], these are called *language invention*. This type of DSL is the main focus of this tutorial. The design effort for this type of language is comparable to that for other forms of DSLs, see Section 2.3, but the implementation effort is bigger due to the fact that the language has to be built from scratch.

2.2 Effectiveness of Domain Specific Languages

Before continuing, it is important to summarize the advantages and drawbacks of DSLs, independent of their category. The effort to design, implement, and maintain a DSL is huge, even given the fact that they are "little". Designing a good DSL involves not only writing a syntax definition, but also the definition of a proper semantics, tooling and eventually methodology and documentation. The gain in investing this effort must exceed the costs. Note, however, that not using a DSL also has its costs, especially in the longer run.

First, we enumerate the drawbacks of DSLs.

- The cost of a DSL implementation and the training of its users may be high.
- It may be difficult to identify the right scope of domain specific concepts and to find a good balance between these concepts and general-purpose language constructs. In other words: is the resulting language usable and effective?
- Domain specific languages offer solutions for a limited set of problems. They are not generally applicable. In a few cases, a DSL evolves into a general purpose language.

In contrast to these drawbacks, a number of obvious advantages can be identified.

- The possibility of expressing the solution in terms of domain concepts. This may lead to higher productivity when developing software. Furthermore, the models developed may offer the opportunity of use in a different setting, for documentation purposes or verification purposes.
- Besides the gain in productivity, the reliability, maintainability, and portability may increase. Or, to put it differently, not using a DSL may lead to software that is hard or impossible to maintain and port.

– A DSL captures domain knowledge and thus leads to concise and in many cases self-documenting specifications.

It is up to the reader to decide whether a DSL is an appropriate solution to his or her problem.

2.3 Identification of Domain Concepts

As with software development in general [12,14], the first step when designing a DSL is to capture the domain concepts, since without an understanding of the domain concepts it is impossible to design a DSL that will be practical and usable. The domain concepts have to be captured in appropriate language constructs, for instance in a mechanism to exchange messages between state machines. The language constructs should be at the right abstraction level, they should not be geared towards a specific platform or general purpose programming language. Furthermore, they should have a proper specified semantics, both statically and dynamically.

The identification of domain concepts is closely related to the field of requirements engineering. So, elicitation techniques used there can also be applied to design a domain specific language, see traditional software engineering course books on this topic [92]. However, before starting the elicitation phase, it is important to identify the problem domain: "A problem domain is defined by consensus, and its essence is the shared understanding of some community" [8]. In the design of a DSL for financial products [9], the world of financial transactions has to be well defined and understood, for instance what is the notion of a customer, bank account, interest period? Given the problem domain, the next step is to identify the problem space, for instance the creation of new interest products which may take too long or the software developed to implement the interest products contains too many bugs. The next step is to identify the language concepts. This can be done by studying the existing informal description of the interest products and their implementations. From this it is, for instance, possible to identify the relevant library components. This information can now be used to identify and define the relevant DSL concepts [60]. This process is clearly an iterative and continuing process. It may even be the case that during the implementation phase of the language some steps have to be redone.

Capturing DSL concepts can be done in multiple ways depending on the underlying implementation pattern [62]. The concepts can be implemented in a library, as extension, embedding or as a complete new language. In the latter case, more general purpose language constructs must be added, such as control flow, procedural abstraction or modularity, of course depending on the need for expressiveness.

2.4 Examples of Domain Specific Languages

It is impossible to give an exhaustive list of all domain specific language. In 1966, Landin [57] already predicted an explosion of programming languages.

DSLs have contributed to the growth of languages considerably. Nevertheless it makes sense to mention a few languages which can be seen as DSLs. HyperText Markup Language (HTML) the language for developing web pages is an example of a non-executable DSL. In the area of web pages there is whole range of DSLs. A recent development is WebDSL [40], which is a DSL which captures the concepts of designing web pages but shields of the underlying implementation details. SQL is a very well known and popular language for relational database queries which has evolved into a general purpose language PL/SQL [37]. YACC [47] is a tool, but the corresponding grammar definition formalism is a non-executable DSL for creating parsers. LEX [59] is the language for defining regular expressions for specifying lexers. In this area SDF (Syntax Definition Language) [44] is a DSL to describe grammars in a declarative and modular way. GraphViz [39] is a software package used for graph layout and DOT is the corresponding language to describe the graphs. BOX [23] is a small non-executable language to describe the formatting of computer programs. LaTeX [56] is a language for formatting texts.

We shall now study a small DSL in detail.

2.5 Simple Language of Communicating Objects

The *Simple Language of Communicating Objects* (`SLCO`) provides constructs for specifying systems consisting of objects that operate in parallel and communicate with each other. `SLCO` has been used to describe the software that controls conveyor belts, even though there are no conveyor belt related concepts or concepts like motors and sensor in the language. Instead, each of these concepts are simply represented by objects that communicate over channels. In Section 3.2 we will present the metamodels of `SLCO`, but in this section we describe the language and motivate a number of the design decisions.

An `SLCO` model consists of a number of classes, objects, and channels, see Listing 1.2. Objects are instances of classes and communicate with each other via channels, which are either bidirectional or unidirectional, see Figure 9 for the metamodel of the channels. `SLCO` offers three types of channels: synchronous channels, asynchronous lossy channels, and asynchronous lossless channels. An example of two objects connected by three channels is shown in Figure 1. The objects p and q, which are instances of classes P and Q, can communicate over channels $p1\_q1$, $q2\_p2$, and $p3\_q3$. The arrows at the ends of the channels denote the direction of communication. Synchronous channels are denoted by plain lines (e.g. $p1\_q1$), asynchronous lossless channels are denoted by dashed lines (e.g. $p3\_q3$), and asynchronous lossy channels are denoted by dotted lines (e.g. $q2\_p2$). A channel can only be used to send and receive signals with a certain signature, indicated by a number of argument types listed between brackets after the name of the channel. Channel $p1\_q1$, for instance, can only be used to send and receive signals with a boolean argument, and channel $q2\_p2$ only allows signals without any arguments.

A class describes the structure and behaviour of its instances, see Figure 6 for the metamodel of classes. A class has ports and variables that define the structure

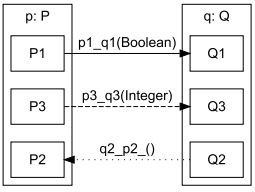

Fig. 1. Objects, ports and channels in SLCO

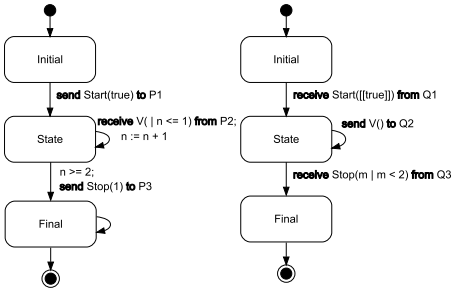

Fig. 2. Two SLCO state machines

of its instances, and state machines that describe their behaviour. Variables may be initialized. If no initial value is specified, integer variables are initialized to 0, boolean variables are initialized to *true*, and string variables are initialized to the empty string. Ports are used to connect channels to objects. Figure 1 shows that object p has ports *P1*, *P2*, and *P3*, connecting it to channels $p1\_q1$, $q2\_p2$, and $p3\_q3$, and that object q has ports *Q1*, *Q2*, and *Q3*, connecting it to the same channels.

A state machine consists of variables, states, and transitions, see Figure 6 for the metamodel of state machines. SLCO allows for two special types of state: initial states and final states. Each state machine has exactly one initial state, and can contain any number of ordinary and final states. Figure 2 shows an example of an SLCO model consisting of two state machines, whose initial states, `Initial`, are indicated by a black dot-and-arrow, and whose final states, `Final`, are indicated by an outgoing arrow to a circled black dot. As explained below, the left state machine specifies the behaviour of object p and the right state machine specifies the behaviour of object q, both already introduced in Figure 1.

A transition has a source and a target state, and a finite number of statements. There are multiple types of statements. Expressions denote statements that must evaluate to true to enable the transition from the source to the target state to be taken. The expression `n >= 2` that is part of the transition with the source `State` and the final state as the target state in the state machine of p is an example of such a statement. A transition with a delay statement is enabled after a specified amount of time has passed since entering its source state. Note that our running example does not have this type of statement. A transition with a signal reception statement is enabled if a signal is received via the port indicated by the statement. When a signal reception statement has a condition, naturally, the condition must hold for the transition to be enabled. It is allowable for the condition to refer to arguments of the signal just being received. Take for instance the transition in q from `State` to the final state, with signal reception `receive Stop(`m | m `< 2) from` $Q3$. It is only taken if the value of the argument sent with the signal `Stop` is smaller than 2. Additionally, another form of conditional signal reception is offered. Expressions given as arguments of a

signal reception specify that only signals whose argument values are equal to the corresponding expressions are accepted. Thus, q in state `Initial` accepts only signals whose argument equals *true*. `SLCO` also offers statements for assigning values to variables and for sending signals over channels. The state machines in Figure 2 specify the following communication between p and q, assuming that the variable n is initialized to 0. The two objects first communicate synchronously over channel $p1\_q1$, after which q repeatedly sends signals to p over the lossy channel $q2\_p2$. As soon as p receives 2 of the signals sent by q, it sends a signal over channel $p3\_q3$ and terminates. After receiving this signal, q terminates as well.

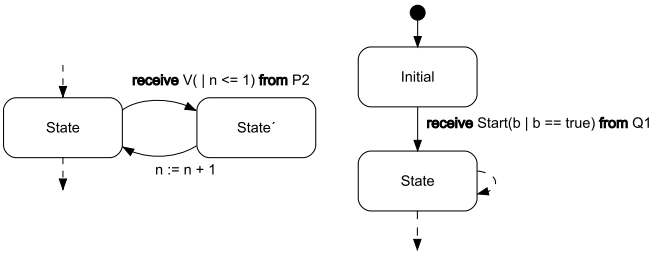

Fig. 3. Parts of an `SLCO` model without syntactic sugar

The example in Figure 2 contains two transitions that both contain two statements, whereas the metamodel in Figure 7 allows at most one statement per transition. Although multiple statements per transition are not allowed according to the metamodel, we consider models containing such transitions valid `SLCO` models and regard these models as the syntactically sugared versions of equivalent models that adhere to the stricter metamodel. The leftmost part of Figure 3 shows the unsugared version of one these transitions, that features an auxiliary intermediate state. The rightmost part of this figure shows that expressions given as arguments of signal receptions can also be regarded as a form of syntactic sugar. The conditional signal reception on the transition from `Initial` to `State` in this figure is equivalent to the conditional signal reception on the corresponding transition in Figure 2, assuming that variable b is an auxiliary boolean variable.

In addition to the graphical concrete syntax shown above, `SLCO` has a textual concrete syntax. Listing 1.1 shows a part of the textual equivalent of the model described above.

3 Defining the Structure of a Domain Specific Language

In the rest of this tutorial we will concentrate on the design of DSLs based on the invention pattern. We assume that the first step in the design of a new DSL, the identification of the domain concepts, has been performed as described in Section 2.3.

```
model M {
  classes
    P {
      variables Integer n
      ports P1 P2 P3
      state machines
        P {
          initial Initial state State final Final
          transitions
            Receive from State to State {
              receive V( | n <= 1) from P2;
              n := n + 1 }
            ...
        }
    }
  ...
  objects p:P q:Q
  channels p1_q1(Boolean) sync from p.P1 to q.Q1
           ...
}
```

Listing 1.1. Part of a textual SLCO model

The design and implementation of an effective DSL is more than just writing a context-free grammar in Xtext [32] and a code generator for some back-end, such as Java. The proper steps are the design of an abstract syntax, semantics and concrete syntax. One can argue about the order of the steps. Kleppe [52] proposes to design the concrete syntax after defining the abstract syntax. Whilst defining the semantics it may turn out that the abstact syntax is not optimal and has to be adapted. This may lead to a modification of the concrete syntax. However, having a concrete syntax may facilitate experimentation and interaction with the users of the language and provide usable feedback. It is obvious these steps must be performed iteratively and continuously, leading to evolutionary design of a DSL [4].

In this section we will explain why a proper abstract syntax is needed as the basis for a DSL. We shall discuss the material on an abstract level and give examples of metamodels in EMF [83], but this tutorial is not an introduction into a specific technology or tool.

In Section 3.1 we will start with introducing the basic concepts of defining abstract syntax. In Section 3.2 we will make this concrete in terms of metamodeling based on EMF.

3.1 Abstract Syntax

The abstract syntax, signature or abstract data type of a language describes the basic structure (skeleton) of a language. It can serve as the starting point for defining a concrete syntax (both textual and graphical), semantics (static and dynamic), and is the basis for tool development. It abstracts from specific details on the concrete level, such as the keywords, priorities between operators, associativities of binary operators, etc. In its basic form an abstract syntax definition is a collection of constructors. Starting with the definition of a language

in an abstract syntax notation has the advantage of having a concise overview of the underlying structure of the language.

Unfortunately there is no ISO standard for defining abstract syntax. We will use a signature-like notation to describe the abstract syntax of a language. A signature can be defined as follows:

- A collection of constructors which define sorts and operators.
- A sort represents a nonempty set of terms.
- A term is the application of a k-ary operator to k terms of the appropriate sort.
- A k-ary operator is a constructor function mapping k terms to a term.
- The argument of a k-ary operator may represent zero or more (`*`) or one or more (`+`) terms of the same sort.
- A sort can be considered a nonterminal in the abstract syntax.

Listing 1.2 shows a part the abstract syntax of the `SLCO` language as a signature.

```
"Synchronous"()          -> ChannelType
"AsynchronousLossless"() -> ChannelType
"AsynchronousLossy"()    -> ChannelType

"Integer"()    -> PrimitiveType
"Boolean"()    -> PrimitiveType
"String"()     -> PrimitiveType

"model"(Class*,Object*,Channel*)                                -> Model
"class"(Name,Port*,StateMachine*,Variable*)                     -> Class
"object"(Name,Class)                                            -> Object
"channel"(Name,ChannelType,ArgumentType*)                       -> Channel
"port"(Name)                                                    -> Port
"statemachine"(Name,Variable*,Vertex*,Transition*)              -> StateMachine
"variable"(Name,PrimitiveType)                                  -> Variable
"argumenttype"(PrimitiveType)                                   -> ArgumentType
"vertex"(Name,Transition*,Transition*)                          -> Vertex
"transition"(Name,Vertex,Vertex)                                -> Transition
"transition"(Name,Vertex,Vertex,Statement)                      -> Transition
```

Listing 1.2. Signature specification of a part of the abstract syntax of `SLCO`

Given a signature definition it is possible to generate Application Programming Interfaces (APIs). GOM [73] generates an API for accessing the underlying abstract syntax when developing TOM specifications [63] and ApiGen [50,20] generates, given a signature definition, a type API to access terms in the ATerm libary [19].

3.2 Metamodeling

The next step is to represent the abstract syntax as a metamodel. A metamodel describes the model elements that are available for developing a class of models as well as their attributes and interrelations. A model describes the elements

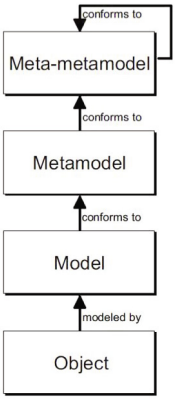

Fig. 4. Four-layer metamodeling architecture

of a real-world object as well as their attributes and the way they interrelate. Therefore, a metamodel can be considered as a model of a class of models [80], or a model of a modeling language [55]. Since a metamodel is itself a model, the concepts and relations that can be used to define them need to be described as well. The metamodel used for this purpose is called a meta-metamodel. A meta-metamodel is typically a reflexive metamodel. This means that it is expressed using the concepts and relations it defines itself. This four-layer metamodeling architecture [66] is schematically depicted in Figure 4.

EMF provides a metamodel (also referred to as Ecore) which is a general model of models from which any model can be defined. It can be used to model classes, attributes, relationships, data types, etc. Figure 5 (taken from [83]) shows a simplified Ecore metamodel:

EClass models classes that
- are identified by a name,
- contain zero or more attributes, and
- contain zero or more references.

EAttribute models attributes that
- are identified by a name, and
- have a type.

EDataType represents basic types.

EReference models associations between classes and
- are identified by a name,
- have a type which must be an EClass, and
- a containment attribute indicating whether the EReference is used as "whole-part" relation.

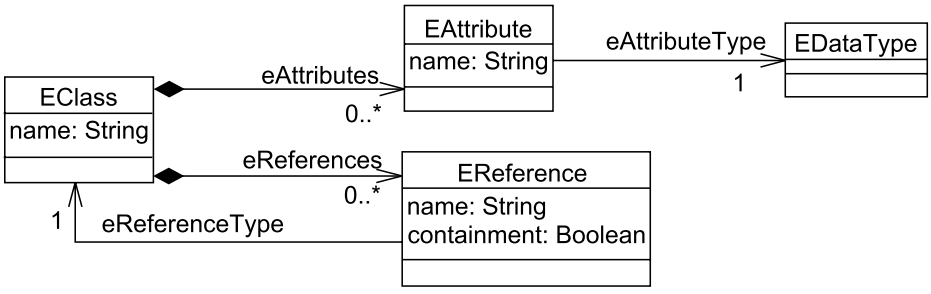

Fig. 5. Simplified Ecore metamodel

In Section 2.5 the SLCO language has been described in great detail, we will now only present the metamodels. Figure 6 shows the main metaclasses of the Ecore metamodel of SLCO, which corresponds to the signature presented in Listing 1.2. Figure 7 gives the details of statements. Figure 8 defines the expressions of SLCO and Figure 9 shows the details of the channels.

4 Semantics

While syntax is concerned with the form of a valid model or program, semantics is about its meaning. Kleppe [52] introduces the term *mogram* in order not to make the distinction between a program and a model. We will consistently use the term *model*, because a program is also a model. We shall consider two different views on defining the semantics of a DSL. The static semantics of a language defines the structural properties of valid models (Section 4.1). The dynamic semantics is concerned with the execution and operation of models (Section 4.2).

4.1 Static Semantics

Static semantics defines properties of models that can be determined without considering either input or execution. Because of this there is always a debate on the status of static semantics. Static semantics can be considered to be part of syntax analysis, because the models need not be executed. We consider static semantics to be a separate phase. We shall distinguish three separate aspects:

- identifier resolution;
- scope resolution; and
- type resolution.

In many languages these aspects are intertwined. If the language supports overloading, type resolution is needed when performing identifier resolution.

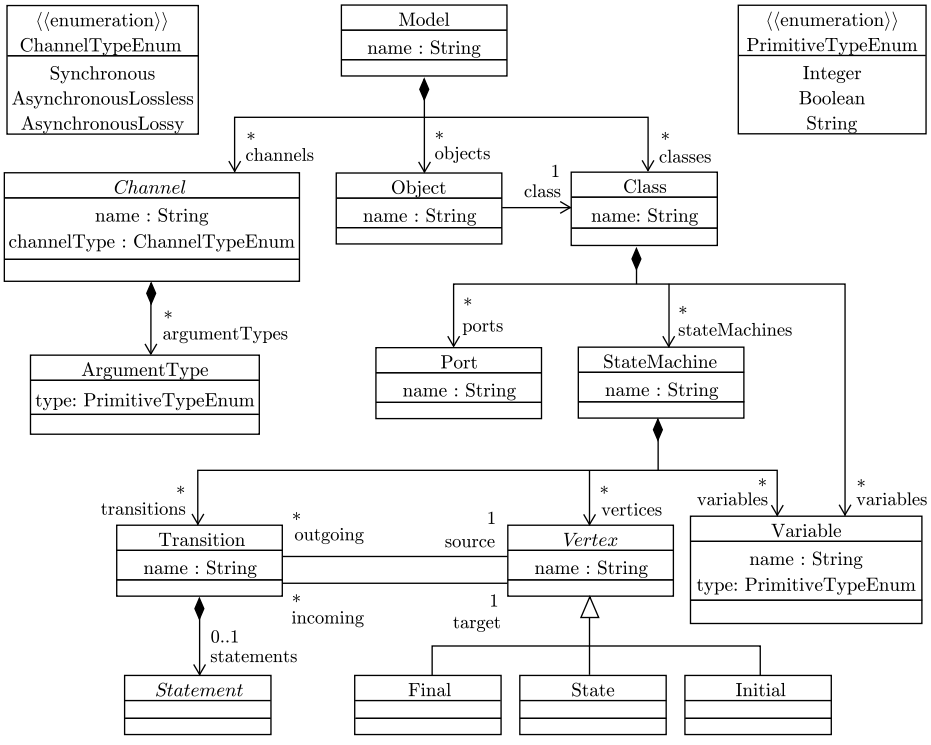

Fig. 6. Main concepts of the SLCO metamodel

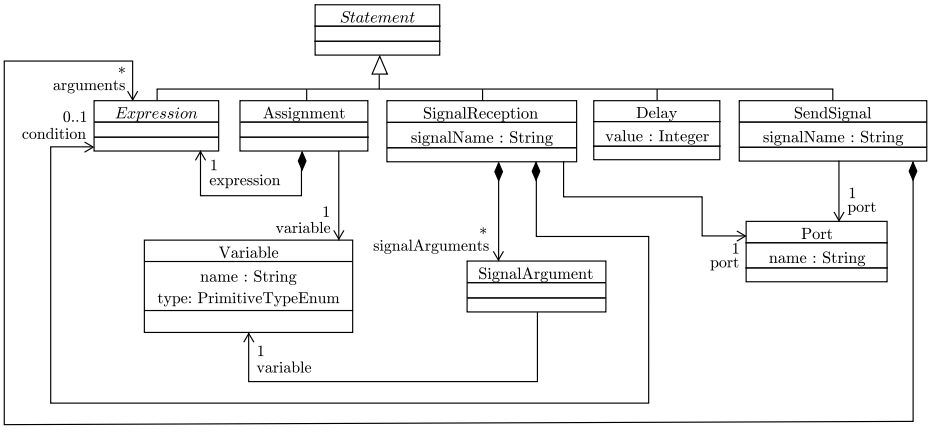

Fig. 7. Statements in the SLCO metamodel

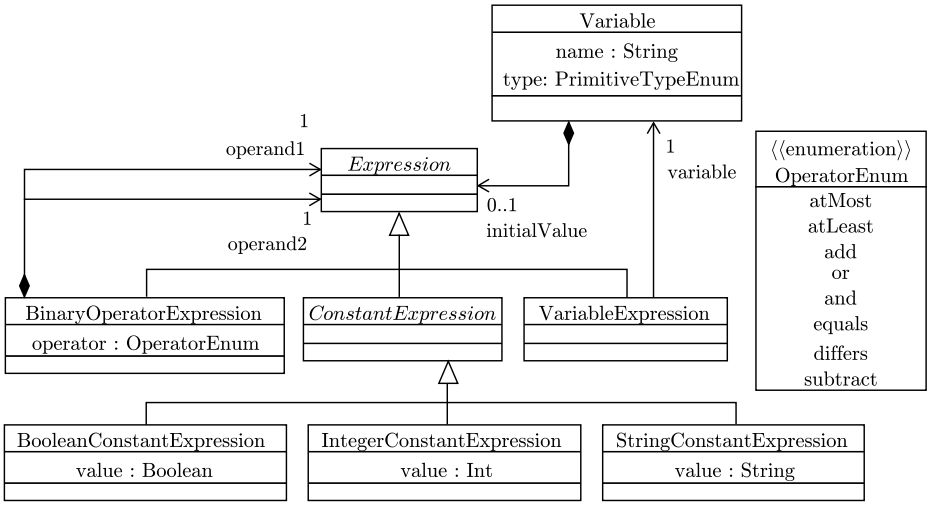

Fig. 8. Expressions in the SLCO metamodel

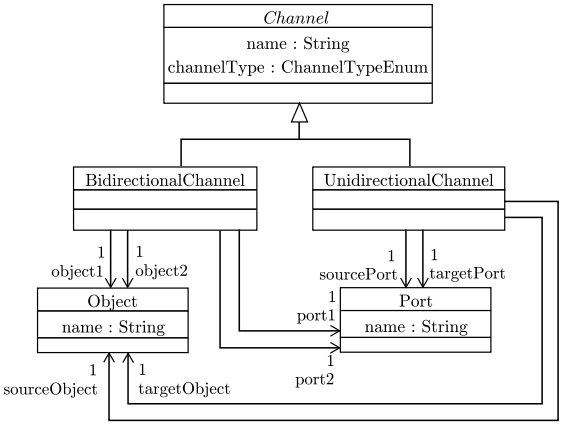

Fig. 9. Channels in the SLCO metamodel

Identifier Resolution. Identifiers are an important notion in (programming) languages. They may represent for instance values, variables, functions, etc. In order to be valid, in many (programming) languages an identifier has to be explicitly declared. In many cases the declaration of an identifier is accompanied by a description of its type, for instance for a value or variable. If the identifier represents a function, procedure or method, the declaration captures the operational behaviour in the body. Furthermore, the parameters (together with their types) and the result parameter have to be specified. A declaration establishes a *binding* of an identifier I to some entity X. An *applied occurrence* is an occurrence of I where X is made available through I. Note that if the language

supports overloading, the same identifier is bound to two or more different bodies where each body has discriminating parameters and/or result types, type resolution is needed when resolving the binding.

Identifier resolution establishes so-called *use-def* relations. Every applied occurrence should correspond to exactly one binding occurrence. There are a few (older) programming languages, for instance Basic and Fortran, where identifiers were not explicitly defined, but the first usage of the identifier was actually the definition. In dynamic languages, such as Python [89], there is no obligation to define identifiers explicitly. This is makes programs concise and more error prone.

Scope Resolution. The introduction of language constructs like functions or modules leads to the notion of blocks. Blocks are syntactic constructs that delimit the visibility of declarations. This is called scoping. The scope of a declaration is the part of the model where the declaration is effective. Blocks and their scopes may overlap.

Although for every applied occurrence there should be exactly one binding occurrence, it is possible that an identifier may be defined in multiple blocks. In the case of nested blocks, some outer block may contain a declaration of identifier *I*. If the inner block contains a declaration of *I*, then this declaration hides the declaration of *I* in the outer block.

There are a few (older) programming languages, for instance again Basic, which do not offer blocks and thus have a single scope. This means that (recursive) functions are unsupported language constructs. This leads to a very restricted way of programming.

When performing identifier resolution we have to take scopes into consideration.

Type Resolution. The main principles of identifier and scope resolution can be explained in a few paragraphs but for type resolution or type checking this is impossible. The definition of proper type check rules for a language depends heavily on the expressive power of the language. If a language contains overloading, polymorphism, inheritance, etc. the definition of a type checker is far from trivial, see the book on type checking by Pierce [71] for an elaborate discussion of type checking.

A type system is an important component of the static semantics of a language. Language constructs, for instance expressions and functions, may produce results and some cases these results respresent values. Types represent abstractions of these results. For instance, the evaluation of an expression may yield the values `true` or `false`, the corresponding type is then `boolean`. Expressions use operators to calculate the results and these operators are only valid for a restricted set of types, in many cases only one type. The purpose of a type system is to prevent illegal operations, like multiplication of strings by booleans, which should result in a type error, disambiguation in case of (operator or function) overloading. The process of assigning types is referred to as *typing*.

Until recently not much work has been done on type checking of DSLs developed using EMF. Xtext offers a primitive mechanism to deal with identifier resolution and OCL [67] is used to write simple constraints to do a basic form of type checking. [11] and [22] describe approaches to tackle type checking of DSLs in a more structured way. [26] uses attribute grammar JastAdd [33] to add the semantics to metamodels.

Static Semantics of `SLCO`. A number of static semantic rules can be defined for the `SLCO` language. Here are a few of the rules:

- Classes should have a unique name within a model.
- Objects should have a unique name within a model.
- Only classes that have been defined can be instantiated as objects.
- Variables should be declared before they can be used.
- Variables should have a unique name within a class.
- State machines should have a unique name within a class.
- The source and target state of a transition should exist.
- States should have a unique name within a state machine.
- Transitions should have a unique name within a state machine.
- The expressions should be type correct.

Listing 1.3 presents a simplified version for checking the uniques of class names using a rewriting based specification formalism.

4.2 Dynamic Semantics

Defining a DSL by means of its abstract and concrete syntax allows the rapid development of languages and some associated tools, such as editors. We have already seen that static semantics is another important ingredient of a DSL definition. Static semantics defines aspects such as the well-formedness and typing of concrete models in order to make the DSL more usable and the development of models more robust. However, none of these aspects of the DSL definition help to understand the behaviour described by models nor help us to inspect whether the behaviour specified is exactly as intended nor help us to create a proper execution model.

Dynamic semantics (also known as execution semantics) covers these language aspects: it defines the model of computation for the execution behaviour of models. There are many ways of defining dynamic semantics, natural language being one of them. In practice, the dynamic semantics of DSLs are implicitly and informally defined through, either, a software constructor that generates compilable source code, or by an engine/interpreter that directly processes DSL models. In these cases, the quality of the design of a system, due to the lack of a formal definition of the DSL semantics, is usually assessed by manually studying and exploring the created models, and by testing the software before actual delivery. Given the increasing complexity of current systems, this has become an error-prone, time-consuming and costly process, which often results

```
module SLCO-typecheck

signature
  "checkModel"(Model)          -> {Error ","}*
  "checkClasses"(Class*, TNPT) -> {Error ","}* # TNPT
  "checkClass"(Name, TNPT)     -> {Error ","}* # TNPT

signature
  Table[[Name,PrimitiveType]] -> TNPT

variables
  "vChannel*"      -> Channel*
  "vClass*"        -> Class*
  "vError*"[12]*   -> {Error ","}*
  "vName"          -> Name
  "vObject*"       -> Object*
  "vPort*"         -> Port*
  "vStatemachine*" -> Statemachine*
  "vTNPT"[123]*    -> TNPT
  "vVariable*"     -> Variable*

equations
[cMdl1] checkModel(model(vClass*, vObject*, vChannel*)) =
          when vError* # vTNPT := checkClasses(vClass*, new-table)

[Clss1] checkClasses(, vTNPT) = # vTNPT
[Clss2] checkClasses(vClass vClass*, vTNPT1) = vError*1, vError*2 # vTNPT3
          when vError*1 # vTNPT2 := checkClasses(vClass*, vTNPT1),
               vError*2 # vTNPT3 := checkCLass(vClass, vTNPT2)

[cCls1] checkCLass(class(vName,  vPort*, vStatemachine*, vVariable*), vTNPT)=
          error("Class redefined: " ++ vName) # vTNPT
          when lookup(vTNPT, vName) != not-in-table

[cCls2] checkCLass(class(vName,  vPort*, vStatemachine*, vVariable*), vTNPT)=
          # insert(vName, vTNPT)
          when lookup(vTNPT, vName) == not-in-table
```

Listing 1.3. Part of specification of the static semantics of SLCO in a rewrite rule style

in software design faults being uncovered only after the system is fully developed and installed.

However, as a result of a significant amount of academic research, different approaches and techniques used to express dynamic semantics of languages in a formal manner have been defined. Furthermore, supporting (semi-)automated tools have been developed for model analysis, which can process models described in a particular formal modeling language. The question of making these formal frameworks available to domain engineers to be used for unambiguous specification and analysis of domain models has been receiving much attention in the last decade, from both the industrial and academic community. In order to make the results useful for industry, it is essential to have precise understanding of the DSL semantics. The lack of well-understood DSL semantics may easily lead to semantic ambiguities or a semantic mismatch between a DSL (the developed models) and modeling languages of analysis tools.

There are various ways to define the dynamic semantics of a language formally. The most common techniques are operational, translational, and denotational

semantics. *Operational semantics* specifies the computation a language construct induces when it is executed, thus describing not only the effect of the computation, but also how the computation is produced. Another technique to define the dynamic semantics of a DSL is *translational semantics* which maps language constructs from the initial domain to another language with an already defined formal semantics. Thus, the semantics of the original DSL is defined by the syntactic mapping. *Denotational semantics* is given by a mathematical function which maps the syntax of the language to semantic values – denotations. In that sense denotational semantics corresponds to translational semantics, where the target language is a mathematical formalism.

Both, the operational and translational approach, have advantages and drawbacks. A number of papers report on explicit definitions of the corresponding operational semantics of individual or a family of DSLs and the way they have been directly expressed in other existing environments (see for instance [28,82,16,36,15,75,77]). In this way DSL models can be executed. However, operational semantics is not easy to implement in general, and so far there is no semantic framework that supports an automated and efficient implementation of the operational language semantics allowing for direct DSL model execution. Nevertheless, it is beneficial to have the operational semantics of the DSL explicitly defined, first of all, because it enables rather easy detection of possible inconsistencies between or redundancies of (the semantics of) language constructs. Furthermore, the investment of defining the operational semantics of a DSL eases the effort of anchoring the DSL to different target languages for different purposes. It allows also (due to its modularity) for low-effort adjustments of the semantics needed in case of language evolution or language changes.

Translational semantics saves the effort of defining the semantics of the DSL explicitly. This approach is very appropriate when the initial language and target language are semantically closely correlated, or when semantic reasoning only for particular instances of the DSL models is required. However, establishing syntactic mappings between the two languages requires implicit semantic knowledge of the DSL and an extensive in-depth knowledge of the (semantics of the) target language. Therefore if, due to a lack of a single sufficiently expressive underlying target framework, multiple frameworks are needed, this approach is obviously not appropriate. Furthermore, as the mappings to different target languages and platforms are defined only at the syntactic level, there is no guarantee that the final translation models still adhere to the semantics of the original model. For example, consider a mapping that transforms the source model to executable code, whereas another mapping translates the source model to a model for formal analysis. It is natural to ask whether the model that has been analysed is the one that will be actually executed.

As a last technique to define dynamic semantics of languages we mention *Action Semantics* [64]. Action semantics is based on operational semantics and allows a modular way of defining the semantics of language constructs. The actions defining the semantics of a language can be interpreted and there is an

$$\frac{\langle \texttt{Class}^*, \texttt{Object}^*, \texttt{Channel}^*, \texttt{S}_{\texttt{OSMS}}, \texttt{V}_{\texttt{OS}}, \texttt{V}_{\texttt{OSMS}}, \texttt{B}\rangle \xrightarrow{\texttt{l}}_{\texttt{OBJECTS}} \langle \texttt{S}'_{\texttt{OSMS}}, \texttt{V}'_{\texttt{OS}}, \texttt{V}'_{\texttt{OSMS}}, \texttt{B}'\rangle}{\begin{array}{c}\langle\ \texttt{model}(\texttt{Class}^*, \texttt{Object}^*, \texttt{Channel}^*), \texttt{S}_{\texttt{OSMS}}, \texttt{V}_{\texttt{OS}}, \texttt{V}_{\texttt{OSMS}}, \texttt{B}\ \rangle \\ \xrightarrow{\texttt{l}}_{\texttt{MODEL}} \langle\ \texttt{model}(\texttt{Class}^*, \texttt{Object}^*, \texttt{Channel}^*), \texttt{S}'_{\texttt{OSMS}}, \texttt{V}'_{\texttt{OS}}, \texttt{V}'_{\texttt{OSMS}}, \texttt{B}'\ \rangle\end{array}}$$

Fig. 10. Deduction rule for models

execution environment [18] for composing semantic definitions, which makes this approach one of the exceptions. In [84] action semantics has been applied in the context of model driven engineering and the semantics of a small DSL is defined. This work is very preliminary.

Dynamic Semantics of SLCO. The semantics of SLCO has been formalized in the form of Structural Operational Semantics (SOS) [72]. Using SOS rules, the behaviour of an SLCO model is described in terms of the behaviour of its parts. The behaviour of these parts is in turn specified in terms of the behaviour of their parts, and so on. Figure 10 shows the SOS rule that describes the behaviour of models. A state of a model is referred to as a configuration. A model configuration is determined by: the current states of the state machines of the objects (function $\texttt{S}_{\texttt{OSMS}}$), the current values of the all local and global variables that appear in the model (functions $\texttt{V}_{\texttt{OS}}$ and $\texttt{V}_{\texttt{OSMS}}$), and the current contents of the buffers associated to the asynchronous channels (function B). In the rule in Figure 10, one configuration is represented by the functions $\texttt{S}_{\texttt{OSMS}}$, $\texttt{V}_{\texttt{OS}}$, $\texttt{V}_{\texttt{OSMS}}$, and B, and another by the functions $\texttt{S}'_{\texttt{OSMS}}$, $\texttt{V}'_{\texttt{OS}}$, $\texttt{V}'_{\texttt{OSMS}}$, and $\texttt{B}'$ (which essentially represent corresponding updates of the previous functions). The rule specifies that a model can take a step labeled l from one configuration to another configuration, if the list of classes ($\texttt{Class}^*$), objects ($\texttt{Object}^*$) and channels ($\texttt{Channel}^*$) that are part of the model can take the same step labeled l. In short, the behaviour of a model is defined in terms of the behaviour of its objects, classes, and channels.

Figure 11 shows that a list of classes, objects and channels can take a step labeled $\texttt{bvn} := \texttt{bc}$ if one of the objects, $\texttt{object}(\texttt{oname}, \texttt{cname})$, is an instance of a class, $\texttt{class}(\texttt{cname}, \texttt{Port}^*, \texttt{StateMachine}^*, \texttt{Variable}^*)$, that can take the same step labeled $\texttt{bvn} := \texttt{bc}$. This is one of a number of rules that specify the behaviour of lists of objects, classes, and channels in terms of their parts. The other rules define communication between objects over the various types of channels.

$$\frac{\begin{array}{c}\texttt{Class}^* \equiv \texttt{Class}_1^*\ \texttt{class}(\texttt{cname}, \texttt{Port}^*, \texttt{StateMachine}^*, \texttt{Variable}^*)\ \texttt{Class}_2^*, \\ \texttt{Object}^* \equiv \texttt{Object}_1^*\ \texttt{object}(\texttt{oname}, \texttt{cname})\ \texttt{Object}_2^*, \\ \langle\ \texttt{class}(\texttt{cname}, \texttt{Port}^*, \texttt{StateMachine}^*, \texttt{Variable}^*), \\ \texttt{S}_{\texttt{OSMS}}(\texttt{oname}), \texttt{V}_{\texttt{OS}}(\texttt{oname}), \texttt{V}_{\texttt{OSMS}}(\texttt{oname})\ \rangle \xrightarrow{\texttt{bvn} := \texttt{bc}}_{\texttt{CLASS}} \langle\ \texttt{S}_{\texttt{SMS}}, \texttt{V}_{\texttt{O}}, \texttt{V}_{\texttt{SMS}}\ \rangle, \\ \texttt{S}'_{\texttt{OSMS}} = \texttt{S}_{\texttt{OSMS}}[\texttt{S}_{\texttt{SMS}}/\texttt{oname}], \quad \texttt{V}'_{\texttt{OS}} = \texttt{V}_{\texttt{OS}}[\texttt{V}_{\texttt{O}}/\texttt{oname}], \quad \texttt{V}'_{\texttt{OSMS}} = \texttt{V}_{\texttt{OSMS}}[\texttt{V}_{\texttt{SMS}}/\texttt{oname}]\end{array}}{\begin{array}{c}\langle\ \texttt{Class}^*, \texttt{Object}^*, \texttt{Channel}^*, \texttt{S}_{\texttt{OSMS}}, \texttt{V}_{\texttt{OS}}, \texttt{V}_{\texttt{OSMS}}, \texttt{B}\ \rangle \\ \xrightarrow{\texttt{bvn} := \texttt{bc}}_{\texttt{OBJECTS}} \langle\ \texttt{S}'_{\texttt{OSMS}}, \texttt{V}'_{\texttt{OS}}, \texttt{V}'_{\texttt{OSMS}}, \texttt{B}\ \rangle\end{array}}$$

Fig. 11. Deduction rule for lists of objects concerning assignments

$$\frac{\texttt{StateMachine}^* \equiv \texttt{StateMachine}_1^*\ \texttt{statemachine}\ \texttt{StateMachine}_2^*,\quad \langle\ \texttt{statemachine}, \texttt{S}_{\texttt{SMS}}, \texttt{V}_0, \texttt{V}_{\texttt{SMS}}\ \rangle \xrightarrow{\texttt{l}}_{\texttt{SM}} \langle\ \texttt{S}'_{\texttt{SMS}}, \texttt{V}'_0, \texttt{V}'_{\texttt{SMS}}\ \rangle}{\langle\ \texttt{class}(\texttt{cname}, \texttt{Port}^*, \texttt{StateMachine}^*, \texttt{Variable}^*), \texttt{S}_{\texttt{SMS}}, \texttt{V}_0, \texttt{V}_{\texttt{SMS}}\ \rangle \xrightarrow{\texttt{l}}_{\texttt{CLASS}} \langle \texttt{S}'_{\texttt{SMS}}, \texttt{V}'_0, \texttt{V}'_{\texttt{SMS}}\rangle}$$

Fig. 12. Deduction rule for classes

The behaviour of a class is defined in terms of the behaviour of the state machines of that class. Figure 12 shows that a class can take a step labeled `l` if one (`statemachine`) of the state machines (`StateMachine`) that are a part of that class can take that same step.

$$\frac{\begin{array}{c}\texttt{Transition}^* \equiv \texttt{Transition}_1^*\ \texttt{transition}\ \texttt{Transition}_2^*,\\ \langle \texttt{transition}, \texttt{S}_{\texttt{SMS}}(\texttt{smname}), \texttt{V}_0, \texttt{V}_{\texttt{SMS}}(\texttt{smname})\rangle \xrightarrow{\texttt{l}}_{\texttt{TRANS}} \langle \texttt{vertex}', \texttt{V}'_0, \texttt{V}_{\texttt{SM}}\rangle,\\ \texttt{S}'_{\texttt{SMS}} = \texttt{S}_{\texttt{SMS}}[\texttt{vertex}'/\texttt{smname}],\quad \texttt{V}'_{\texttt{SMS}} = \texttt{V}_{\texttt{SMS}}[\texttt{V}_{\texttt{SM}}/\texttt{smname}]\end{array}}{\begin{array}{c}\langle \texttt{statemachine}(\texttt{smname}, \texttt{Variable}^*, \texttt{Vertex}^*, \texttt{Transition}^*), \texttt{S}_{\texttt{SMS}}, \texttt{V}_0, \texttt{V}_{\texttt{SMS}}\rangle\\ \xrightarrow{\texttt{l}}_{\texttt{SM}} \langle \texttt{S}'_{\texttt{SMS}}, \texttt{V}'_0, \texttt{V}'_{\texttt{SMS}}\rangle\end{array}}$$

Fig. 13. Deduction rule for state machines

Figure 13 shows the SOS rule that specifies that the behaviour of a state machine is deduced from the behaviour of the transitions, `Transition`*, that are a part of that state machine. If a transition `transition` from the list of transitions can take a step labeled `l`, then the state machine can also take a step labeled `l`.

$$\frac{\langle \texttt{AssignmentStatement}, \texttt{V}_0, \texttt{V}_{\texttt{SM}}\rangle \overset{\texttt{l}}{\Rightarrow}_{\texttt{ASSIGN}} \langle \texttt{V}'_0, \texttt{V}'_{\texttt{SM}}\rangle}{\begin{array}{c}\langle\ \texttt{transition}(\texttt{tname}, \texttt{vertex}, \texttt{vertex}', \texttt{AssignmentStatement}), \texttt{vertex}, \texttt{V}_0, \texttt{V}_{\texttt{SM}}\rangle\\ \xrightarrow{\texttt{l}}_{\texttt{TRANS}} \langle \texttt{vertex}', \texttt{V}'_0, \texttt{V}'_{\texttt{SM}}\rangle\end{array}}$$

Fig. 14. Deduction rule for transitions

Figure 14 shows that a transition from `vertex` to `vertex`$'$ leads to a step labeled `l` if the assignment `AssignmentStatement` leads to a step labeled `l` given valuation functions $\texttt{V}_0$ and $\texttt{V}_{\texttt{SM}}$. (Here $\langle \texttt{AssignmentStatement}, \texttt{V}_0, \texttt{V}_{\texttt{SM}}\rangle \overset{\texttt{l}}{\Rightarrow}_{\texttt{ASSIGN}} \langle \texttt{V}'_0, \texttt{V}'_{\texttt{SM}}\rangle$ simply means an update of the valuation functions $\texttt{V}_0$ and $\texttt{V}_{\texttt{SM}}$

according to the assignment statement `AssignmentStatement`, which is also defined by a set of rules.) These valuation functions are part of the configuration mentioned above, and are deduced from the model valuation functions $\mathtt{V_{OS}}$ and $\mathtt{V_{OSMS}}$ as follows. Given an object named `on` and a variable named `vn`, $\mathtt{V_{OS}(on)(vn)}$ represents the value of this variable. A valuation function $\mathtt{V_O}$ that maps the global variables of a particular object named `on` to their values is obtained by applying the function $\mathtt{V_{OS}}$ to the name `on`. The model valuation function $\mathtt{V_{OSMS}}$ has a similar purpose and valuation functions for state machines such as $\mathtt{V_{SM}}$ are obtained similarly.

There are rules for transitions with (conditional) signal receptions, expressions, and send signal statements similar to one described.

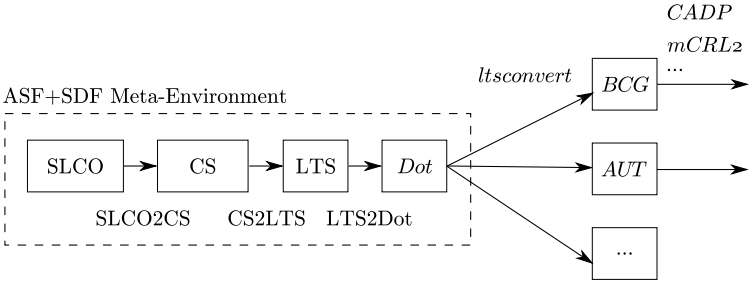

Fig. 15. Languages involved in the executable prototype of the semantics of `SLCO`

To test various alternative semantics for `SLCO`, we first implemented an executable prototype of the semantics before formally defining the semantics on paper [7]. The prototype consists of a number of transformations that transform `SLCO` models to Labeled Transition Systems (LTSs) [72] represented as Dot graphs [39] via a number of intermediate languages. Figure 15 shows the languages and transformations that are involved in this prototype. The boxes in Figure 15 represent (intermediate) languages. The names of existing languages are shown in italics and the names of newly created languages are shown using a plain typeface. The arrows in the figure represent transformations. The names of transformations implemented by existing tools are again shown in italics and the names of newly created transformations are shown using a plain typeface. The transformation *SLCO2CS* transforms an `SLCO` model into a list of configurations and a list of steps, where each step is a pair: configuration and label. The implementation of this transformation uses conditional rewrite rules, which closely resemble the SOS rules described above and the notation used to specify the static semantics. This simplifies the process of formalizing the semantics after the prototype reached a stable state.

Figure 16 shows the state space of the example model of Section 2.5. It is obtained by transforming the `SLCO` model to an LTS represented as a directed

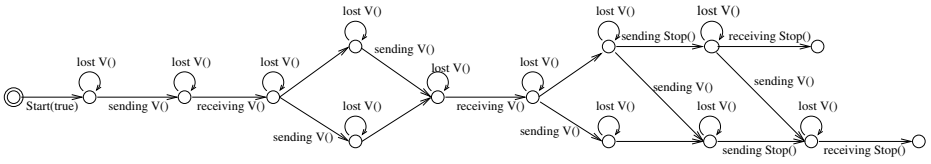

Fig. 16. State space of the example model of Section 2.5

graph in Dot format. The behaviour represented by the LTS matches the informal description of the behaviour in the final paragraph of Section 2.5.

5 Concrete Syntax

Having defined the abstract syntax of a DSL, the next step is the definition of a concrete representation of the DSL. Although it is possible to define a concrete graphical representation of a DSL using the Graphical Modeling Framework (GMF), see for instance Chapter 4 of [41], we will restrict ourselves to the development of textual based DSLs.

In Figure 4, the four-layered architecture of the metamodeling world is presented. For the grammar world, a similar four-layer architecture exists [2,54].

M3. (E)BNF grammar defines structure of the (E)BNF in (E)BNF.
M2. Programming language grammar defines the structure of programming language in (E)BNF.
M1. Program describes the manipulation (algorithm) of data in the data layer.
M0. Data layer where the data we wish to manipulate resides.

Although both worlds seem to be far apart, given the development of Xtext, EMFtext, etc., they are getting closer together [17].

The concrete textual syntax of a (programming or domain specific) language can be described using regular expressions (for the lexical tokens of the language) and context-free grammars (for the (tree) structure of the language). Scanning and parsing have been extensively studied in relation to compilers and interactive development environments. Over the years a broad range of algorithms, see [42], and tools [68] have been developed. The aforementioned tools, like LEX [59] and YACC [47] are results of this research. The formalisms underlying LEX and YACC have been considered as the standard for defining lexical and context-free grammars. In that sense, they are DSLs themselves. Unfortunately there is no ISO standard for defining context-free grammars; every tool comes with their own formalism.

We will first present some context-free grammar terminology and discuss recent developments with respect to modular grammar specification formalisms, which is relevant because of the modularity of metamodels. Then ANTLR and

EMF based derivations will be discussed and finally a context-free grammar for SLCO will be presented.

5.1 (E)BNF

The fact that there is no standard notation for grammars makes it ackward to present concepts and ideas. Formally, a context-free grammar is a 4-tuple $G = (N, \Sigma, P, S)$ where:

- N is the set of nonterminals;
- Σ is the set of terminals (disjoint from N);
- P is a subset of $N \times (N \cup \Sigma)^*$, where an element $(A, \alpha) \in P$ is called a production, usually written as A ::= α;
- $S \in N$ is the start symbol; and
- sets N, Σ, and P are finite.

A context-free grammar can be considered a simple rewrite system: $\alpha A \beta \Rightarrow \alpha\gamma\beta$ if A ::= $\gamma \in P$ where α, β, $\gamma \in (N \cup \Sigma)^*$, $A \in N$.

A full example of a context-free grammar is:

$N = \{$E$\}$
$\Sigma = \{$+, *, (,), -, a$\}$
$S =$ E
$P = \{$E ::= E + E, E ::= E * E, E ::= (E), E ::= - E, E ::= a$\}$

When writing grammar definitions, the nonterminals and terminals are usually implicitly defined and only the start symbol and the production rules are defined explicitly. If the production rules have the form as presented in the example, the grammar is in Backus Normal Form (BNF). If the grammar definition formalism offers operators like *, +, and ?, the grammar is in Extended BNF (EBNF). Xtext [32] and EMFtext use variations of EBNF.

The language $L(G)$ generated/recognized by the context-free grammar $G = (N, \Sigma, P, S)$ is defined by $L(G) = \{w \in \Sigma^* | S \Rightarrow^* w\}$. A sentential form α is a string of terminals and nonterminals which can be derived from S: $S \Rightarrow^* \alpha$ with $\alpha \in (N \cup \Sigma)^*$. A sentence in $L(G)$ is a sentential form in which no nonterminals occur.

There a number of aspects one has to take care of when writing context-free grammars:

1. *The grammar may be left recursive.*
   A grammar G is left recursive if one or more of the production rules has the form A ::= $A\alpha$, so called direct left recursion. This means that after one or more steps in a derivation an occurrence of A reduces again to an occurrence of A without recognizing anything in the input sentence. Or there is a collection of production rules, for instance A ::= $B\alpha$ and B ::= $A\beta$, such

that $A \Rightarrow^* A\gamma$, so called indirect left recursion. It is relatively easy to remove (in)direct left recursion from a grammar. However, removing left recursion introduces new nonterminals and changes the structure of the grammar. In a conventional top-down parser or recursive descent parser, left recursion leads to non-termination. However, Frost et.al. [38] have developed a recursive descent parser using parser combinators, that can deal with left recursion. The GLL algorithm [79] can also handle it without modifying the grammar.

2. *The grammar may be non-left factored.*
   A grammar G is non-left factored if one or more of the production rules has at least 2 alternatives that derive strings with the same prefix that is non-empty, for instance A `::=` $\alpha\beta_1|\alpha\beta_2|\ldots$. This case is relatively easy to left factor, but again new nonterminals are introduced and the structure of the grammar is changed. However, A `::=` $X\beta_1|Y\beta_2|\ldots$, where X `::=` $'x'$ and Y `::=` $'x'$, requires a *First* calculation; see [1] for details. In a top-down parser or recursive descent parser, arbitrary look ahead may be needed to decide which of these alternatives to choose.
3. *The grammar may contain cycles.*
   A grammar G is cyclic if in one or more derivation steps A produces A without recognizing any token from the input: $A \Rightarrow^* A$. This can be considered a bug in the grammar specification. Both top-down (LL) and bottom-up ((LA)LR) parsers will have trouble with grammars containing cycles; both types of parsers will not terminate.
4. *The grammar may be ambiguous.*
   A grammar G is *ambigous* if a word $w \in L(G)$ has two or more derivations. This may happen if there are production rules of the form `E ::= E O E` without an explicit definition of priorities between binary operators or associativity of the binary operators. The sentence `a + a * a` can be recognized in two different ways. If a language is inherently ambiguous, no refactorings of production rules will help to solve the problem.

5.2 SDF

The Syntax Definition Formalism (SDF) [44,91] is a modular declarative grammar definition formalism. SDF allows modularization of context-free grammars in order to enable reuse and clarity. Associativity and priorities of binary operators can be defined in a declarative way. The definition of lexical and context-free syntax rules are fully integrated which contributes also to the declarative way of writing grammar definitions. Restricted classes of context-free grammars, such as the classes of LL and LR grammars, are not closed under union, only the largest class of general context-free grammars is closed under union. The underlying implementation of SDF is a generalized LR parsing algorithm (S)GLR [74,90] which can handle this general class. General context-free grammars may be ambiguous; SDF offers multiple various mechanisms to deal with ambiguities [21]. Johnstone et.al. [48] describes the fundamentals of modularity of grammar

formalisms. Spoofax/SDF [51] is an Eclipse plugin for developing SDF modular grammar definitions. Rascal [53] offers also a modular grammar definition formalism, it can be considered as a follow up of SDF, Rascal is available as an Eclipse plugin as well.

5.3 ANTLR, Xtext and EMFtext

Recent developments (over the last 10 years) have resulted in a renewed interested in parsing technology. ANTLR [69], GLR [86,74], SGLR [90] and GLL [79] are recent implementations of newly developed algorithms.

ANTLR. ANTLR is the basis of the popular Xtext [32] and EMFtext[88] implementations available for EMF. ANTLR is very popular because of its availability on multiple platforms. ANTLR is based on LL(*) parsing [70] and this is in many cases a serious drawback since it prohibits left recursion in the grammar. ANTLR, in contrast to SDF, does not support grammar modularity.

Xtext. Xtext [32] is popular for defining the concrete syntax of DSLs [34]. It is available as an Eclipse plugin and is well integrated with EMF. Given a context-free syntax definition, a metamodel can be derived automatically. The nonterminal in the left hand side of a production rule is transformed into an object of that type, and nonterminals in the right hand side are transformed into attributes of this object. You can enforce the creation of a class of a specific type using the `returns` action to create of an object of that specific type. In addition, Xtext offers a mechanism to create cross references. A link can be automatically established to an earlier created object. Actually, this feature breaks the context-freeness of Xtext grammars.

A number of common lexemes are predefined, such as `ID`, `INT`, `STRING`, `WS`, `ML_COMMENT`, `SL_COMMENT`, and `ANY_OTHER`.

Xtext is based on ANTLR, so it has the same characteristics. Xtext is very suited to define the concrete syntax of new DSLs, however for the definition of languages for which an abstract or concrete syntax already exists, Xtext is more tedious.

Grammar rules for `Model` and `CLass` of the Xtext specification of `SLCO` is given in Listing 1.4. Both grammar rules show the use of the Xtext `+:=` operator to concatenate lists. Listing 1.5 shows the grammar rules for `Expression` after removing left recursion. Note, that the binary operators have all become right associative and have the same priority. This listing shows the use of the Xtext `enum` construct when defining `Operator`. In the grammar rules for `TerminalExpression` and `BracketExpression`, the Xtext `returns` operator is used.

EMFtext. EMFtext [88] is tightly integrated with EMF. It enables the definition of a textual concrete syntax for Ecore based metamodels. Similarly to

```
Model :
  'model' name = ID '{'
    ('classes'
       (classes += Class)*
    )?
    ('objects'
      (objects += Object)*
    )?
    ('channels'
      (channels += Channel)*
    )?
  '}';

Class :
  name = ID '{'
    ('variables'
      (variables += Variable)*
    )?
    ('ports'
      (ports += Port)*
    )?
    ('state machines'
      (stateMachines += StateMachine)*
    )?
  '}';
```

Listing 1.4. The `Model` and `Class` definition in Xtext

Xtext, it uses ANTLR as its underlying parsing technology. It differs from Xtext in that it assumes an existing set of metamodels for which a concrete syntax has to be defined. EMFtext offers some sort of modularity; it offers an import mechanisms for various metamodels and modularization of the concrete syntax specifications.

EMFtext offers predefined lexical tokens, as well as the option to define the lexical syntax yourself.

EMFtext offers a special annotation (`@Operator`) to define the operator precedence and associativity of unary and binary operators. This annotation can be used when defining the expression syntax which would otherwise be defined using left recursive rules.

6 Outlook

6.1 Development and Usage of SLCO

The creation of the SLCO language has been, to some extent, motivated by the Falcon project [35,43]. The overall challenge of the project was developing a fully integrated and automated logistics warehouse of the future. One part of the project activities has been centered on investigating the applicability of MDE techniques for modeling of composite system components and for their proper integration with advanced hardware components (like grippers) [6]. The underlying concept was that:

```
Expression :
  TerminalExpression ({BinaryOperatorExpression.operand1 = current}
                      operator = Operator operand2 = Expression)?;

TerminalExpression returns Expression :
  BooleanConstantExpression |
  IntegerConstantExpression |
  StringConstantExpression |
  VariableExpression |
  BracketExpression;

enum Operator :
  atLeast = '>=' | atMost = '<=' | add = '+' | and = '&&' |
  or = '||' | equals = '==' | differs = '!=' | subtract = '-';

BooleanConstantExpression :
  value = BOOLEAN;

IntegerConstantExpression :
  value = INT;

StringConstantExpression :
  value = STRING;

VariableReference :
  name = ID;

VariableExpression :
  variable=VariableReference;

BracketExpression returns Expression:
  "(" Expression ")";
```

Listing 1.5. The Xtext specification of SLCO Expressions

- Warehouse control software is usually a collection of interacting components, which have the same (or very similar) functionality.
- All components' activities are triggered by the reception of messages/signals from other components.
- All system components have the same interfaces.
- The overall system behaviour is determined by the communication of the system components.

Having these aspects in mind as the main guidelines and taking a rather abstract and simplified view at the warehouse software control design issue, SLCO has been developed to describe the software that controls conveyor belts. In order to do simple experiments we have built a simple conveyor belt system using Lego Mindstorms [58]. By shifting the problem domain towards the communication between components, we could abstract even further away from conveyor belt related concepts and concepts like motors and sensors. Besides developing a language, the driving idea was to develop an environment which, among other things, allows automated generation of various refined models for different purposes from a *single* abstract SLCO model:

- automated generation of models for model (simulation) analysis and
- variants of executable code for different platforms.

Before we go in some more detail on the SLCO development process, we stress once again that these two aspects are very important for a DSL to be properly used in practice. First, using one single (abstract) DSL source model, from which various (refined) models are generated, guarantees that the requirements modeled in the original model and validated by simulation or checked by formal analysis are also preserved in the code to be executed, under the assumption of the correctness of the used model transformations. Second, the DSL modeling should be independent from the platform on which the system is going to be implemented. In the simple case of SLCO and control software for Lego Mindstorms conveyor belts, an abstract SLCO model capturing communication between controllers, should not contain details about the number of controllers in the implemented system. These details are very likely irrelevant also for simulation or verification of the abstract model, and they can easily increase the model complexity, and thus make the analysis more difficult. Therefore, such platform details should be added at some later stage, when the control software execution code is generated: starting from a single abstract model, various well-defined model transformations shall generate platform-dependent code for the various platforms used.

Simultaneously with the development of the SLCO, a number of model transformations to other formalisms has been defined and implemented: one for simulation, one for execution, and one for formal verification. These model transformations were developed consecutively. First, a model transformation was implemented to enable simulation of the models using POOSL [85]. In this way, models developed using an intuitive, graphical syntax can be simulated without the need for modelers to learn the syntax and semantics of a formalism for simulation. Second, a model transformation was implemented to generate NQC [10] models for execution on the Lego Mindstorms platform [58]. Executing the code generated from a model revealed bugs in the model that originated from unforseen interleavings of concurrent objects. These bugs were not encountered during simulation. To detect these kinds of problems, a third model transformation was implemented to the Promela formalism for formal verification using the model checker SPIN [45]. All three of the aforementioned formalisms have semantic properties that are different from the semantic properties of SLCO. To enable model transformations from SLCO to each of these platforms, several semantic gaps needed to be bridged [5].

Each of these gaps is bridged by one or more *endogenous* model transformations that transform a given SLCO model to different but equivalent models, also specified in SLCO. The resulting model is semantically better aligned with the target platform. Endogenous model transformations are model transformations where the input and output model adhere to the same metamodel [61].

To be able to use endogenous transformations for this purpose, we needed to extend SLCO with constructs to specify systems on a lower level of abstraction too. In other words, the transformations add implementation details to the original SLCO model, resulting in a refined SLCO model that is closer to one of the target formalisms. This approach has as advantage that the *exogenous*

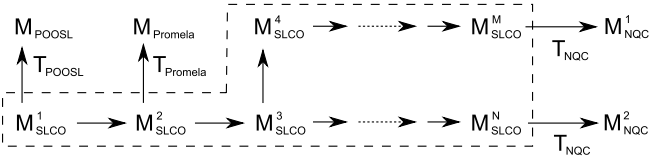

Fig. 17. Sequences of fine-grained model transformations for three target formalisms

transformations to the different platforms are simplified to purely syntactic transformations. Exogenous model transformations are model transformations where the input and output model adhere to the different metamodels [61].

Figure 17 depicts a number of composed model transformations that transform an SLCO model to the three target formalisms. The arrows inside the dashed shape depict endogenous model transformations that transform SLCO models into more refined SLCO models. The arrows across the border of the dashed shape depict exogenous model transformations. Because the semantic gaps between SLCO and the target formalisms are bridged completely by the endogenous model transformations, these exogenous transformations are straightforward translations of SLCO constructs into equivalent constructs in the target formalisms.

Using the prototype semantics we have been able to inspect the relation between original and transformed SLCO models, for a number of model instances. For each of the endogenous transformations, we conjectured that applying such a transformation leads to a model with observationally equivalent behaviour. Although experiments supported the established conjecture for a given number of input models, a more generic approach has been required to establish a relation between an arbitrary input and the output models of an SLCO model transformation.

For this purpose, we needed a general formal framework for the SLCO language allowing us to reason about and to compare model behaviours. For this purpose, we used the SOS definition of SLCO semantics. It generates a labeled transition system (LTS) representation of the dynamics of an SLCO model. In this way, the relation between SLCO models boils down to establishing an appropriate (behavioural) equivalence relation between LTSs.

Here, additional benefits of fine-grained transformations (see [3]) are evident, since they allow for rather straightforward proofs. For the constraints that were required on the input models for some of the transformations, which we detected earlier during our experimental work, it can now be formally shown that these are necessary for the correctness of the transformations as well. Thus, we formally proved that the sequences of transformations used to generate code are well composed.

6.2 How to Get Started

One way to develop experience in model driven software engineering is to dive in and apply it to your own problem domain. Other ways are to explore

examples and case studies, such as SLCO, or to do your own case studies on familiar domains. We briefly describe a few of such domains here. The Eclipse Modelling Framework can be downloaded for free, and provides all the tools you need to get started.

Puzzles and games make for interesting case studies, since their domains are small and well defined, and their semantics are often not trivial but still well contained. More specifically, the reader can try to model sliding block puzzles, like *Rush Hour* [95]. For Rush Hour, the metamodel (abstract syntax) will define the puzzle elements, in particular, the board, the various types of cars and trucks, and how they can be positioned on the board, without overlapping. A concrete syntax provides a language in which one can express specific instances of the Rush Hour puzzle. The dynamic (behavioural) semantics defines the concept of a solution, in particular, the allowed moves and the puzzle's objective, viz. 'liberating' the red car. Finally, model transformations can be used to generate code to visualize and interactively simulate Rush Hour puzzles, or to generate input for a tool (such as a state space explorer) to solve these puzzles. A bigger challenge is to generalise the DSL so as to cover a larger range of sliding block puzzles.

In [60], the domain of traffic light control is used as a case study in DSL design. The main concepts to be modelled in the abstract syntax are: time, traffic participants, junctions consisting of multiple, possibly intersecting, traffic flows, traffic lights, and sensors. The dynamic semantics concerns *state changes* in the traffic flows (as detected by sensors) and in the traffic lights, and the notions of *safety* and *fairness*. Of course, one can start with simplified situations first, e.g., two intersecting traffic flows with a sensor for one flow.

An extensive modelling effort for the railway domain is presented in [13,87]. Again, one can start small, e.g., with railway infrastructure, consisting of railway lines and stations, where railway lines are built from units such as linear segments, switches, simple crossovers, and switchable crossovers. Later, one can add light signals and secured railroad crossings. The DSL can describe specific railway nets. By adding dynamic semantics, one can address the configuration of routes, by appropriate switch and crossover settings. The models can be used to generate code for animation, or to generate input for analysis tools.

Finally, elevator control is an accessible domain. The metamodel defines concepts such as passengers, elevator shafts, elevator cages, floors, doors (in the cage and on the floors), request buttons with controllable lights, and optionally sensors. The elevator control DSL can describe specific elevator systems. The dynamic semantics concerns the movement of passengers, doors, and cages, and the making and handling of passenger requests. The models can be used in ways similar to those for railway nets.

6.3 Future Developments

Model driven software engineering is a promising development with the potential to raise software development to a higher level and to disclose formal methods in disguise to the (embedded) software industry. The formalization of static and

dynamic semantics of DSLs is a must, especially if proofs of correctness are required.

The fact that Eclipse is used as the default implementation platform has led and will lead to bottom-up standardization of formalisms for defining metamodels and model transformations. Xtext and EMFtext are promising tools, but more powerful and declarative grammar formalisms in the context of EMF are still needed. Also, metamodel refactoring with model co-evolution and (meta)-model modularity need better support for large-scale industrial application. Furthermore, a boost for the development of more standardized formalisms to describe static and dynamic semantics is still needed. Some (promising) initial work is done, but more work is needed.

There are some risks; the Eclipse framework may become too heavy and transform into a technological "Tower of Babel". Metamodels, concrete grammars, model transformations, etc. have become mature software artifacts that have to be versioned, maintained and analyzed. Some preliminary work in this area has been performed and will be presented at SFM-2012, but more research needs still to be performed.

The development of DSLs may become easier thanks to better tool support, but the intellectual challenge will remain. The tutorial "DSL Design for Dummies" is still to be written.

Acknowledgment. We would like to thank Arjan van der Meer for proof reading the section on static semantics. We would to thank Adrian Johnstone and Elizabeth Scott of Royal Holloway University London for reviewing this tutorial several times. The second author was on sabbatical in their group and wrote this tutorial during this period and had very fruitful discussions on the various topics addressed in this tutorial.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools. Addison-Wesley Longman Publishing Co., Boston (1986)
2. Alanen, M., Porres, I.: A Relation between Context-Free Grammars and Meta Object Facility Metamodels. Technical Report 606, TUCS (2004)
3. van Amstel, M.F., van den Brand, M.G.J., Engelen, L.: Using a DSL and Fine-Grained Model Transformations to Explore the Boundaries of Model Verification. In: Proc. ICSTW 2011, pp. 63–66. IEEE Computer Society (2011)
4. van Amstel, M.F., van den Brand, M.G.J., Engelen, L.J.P.: An Exercise in Iterative Domain-Specific Language Design. In: Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE), Antwerp, Belgium, pp. 48–57. ACM Press (September 2010)
5. van Amstel, M.F., van den Brand, M.G.J., Protić, Z., Verhoeff, T.: Transforming Process Algebra Models into UML State Machines: Bridging a Semantic Gap? In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 61–75. Springer, Heidelberg (2008)

6. van Amstel, M.F., van den Brand, M.G.J., Protić, Z., Verhoeff, T.: Model-driven software engineering. In: Hamberg, R., Verriet, J. (eds.) Automation in Warehouse Development, pp. 45–58. Springer, London (2011)
7. Andova, S., van den Brand, M.G.J., Engelen, L.: Prototyping the Semantics of a DSL using ASF+SDF: Link to Formal Verification of DSL Models. In: Proceedings of the Second International Workshop on Algebraic Methods in Model-based Software Engineering, AMMSE 2011 (2011)
8. Arango, G.: Domain analysis: from art form to engineering discipline. SIGSOFT Softw. Eng. Notes 14, 152–159 (1989)
9. Arnold, B.R.T., van Deursen, A., Res, M.: An algebraic specification of a language for describing financial products. In: Wirsing, M. (ed.) ICSE-17 Workshop on Formal Methods Application in Software Engineering, pp. 6–13. IEEE (April 1995)
10. Baum, D.: NQC Programmer's Guide (2003)
11. Bettini, L.: A DSL for writing type systems for Xtext languages. In: Proceedings of the 9th International Conference on Principles and Practice of Programming in Java, PPPJ 2011, pp. 31–40. ACM, New York (2011)
12. Bjørner, D.: Rôle of Domain Engineering in Software Development—Why Current Requirements Engineering Is Flawed ! In: Pnueli, A., Virbitskaite, I., Voronkov, A. (eds.) PSI 2009. LNCS, vol. 5947, pp. 2–34. Springer, Heidelberg (2010)
13. Bjørner, D.: Train: The Railway Domain. In: Jacquart, R. (ed.) Building the Information Society. IFIP, vol. 156, pp. 607–611. Springer, Boston (2004)
14. Bjørner, D.: Domain Engineering. In: Boca, P., Bowen, J.P., Siddiqi, J. (eds.) Formal Methods: State of the Art and New Directions, pp. 1–41. Springer, London (2010), doi:10.1007/978-1-84882-736-3_1
15. Bodeveix, J.-P., Filali, M., Lawall, J., Muller, G.: Formal Methods Meet Domain Specific Languages. In: Romijn, J., Smith, G., van de Pol, J. (eds.) IFM 2005. LNCS, vol. 3771, pp. 187–206. Springer, Heidelberg (2005)
16. Bozzano, M., Cimatti, A., Katoen, J.-P., Nguyen, V., Noll, T., Roveri, M.: Safety, Dependability and Performance Analysis of Extended AADL Models. Comput. J. 54(5), 754–775 (2011)
17. van den Brand, M.G.J.: Model-Driven Engineering Meets Generic Language Technology. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 8–15. Springer, Heidelberg (2009)
18. van den Brand, M.G.J., Iversen, J., Mosses, P.D.: An Action Environment. Science of Computer Programming 61(3), 245–264 (2006)
19. van den Brand, M.G.J., de Jong, H.A., Klint, P., Olivier, P.A.: Efficient annotated terms. Software: Practice & Experience 30(3), 259–291 (2000)
20. van den Brand, M.G.J., Moreau, P.E., Vinju, J.J.: A generator of efficient strongly typed abstract syntax trees in Java. IEE Proceedings Software 152(2), 70–78 (2005)
21. den van Brand, M.G.J., Scheerder, J., Vinju, J.J., Visser, E.: Disambiguation Filters for Scannerless Generalized LR Parsers. In: CC 2002. LNCS, vol. 2304, pp. 143–158. Springer, Heidelberg (2002)
22. van den Brand, M.G.J., van der Meer, A.P., Serebrenik, A., Hofkamp, A.T.: Formally specified type checkers for domain specific languages: experience report. In: Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications, LDTA 2010, pp. 12:1–12:7. ACM, New York (2010)
23. van den Brand, M.G.J., Visser, E.: Generation of formatters for context-free languages. ACM Transactions on Software Engineering and Methodology 5(1), 1–41 (1996)

24. Brooks Jr., F.P.: No silver bullet essence and accidents of software engineering. Computer 20, 10–19 (1987)
25. Budinsky, F., Brodsky, S.A., Merks, E.: Eclipse Modeling Framework. Pearson Education (2003)
26. Bürger, C., Karol, S., Wende, C.: Applying attribute grammars for metamodel semantics. In: Proceedings of the International Workshop on Formalization of Modeling Languages, FML 2010, pp. 1:1–1:5. ACM, New York (2010)
27. Campbell-Kelly, M.: From airline reservations to Sonic the Hedgehog: a history of the software industry. History of computing. MIT Press (2003)
28. Combemale, B., Crégut, X., Garoche, P.-L., Thirioux, X.: Essay on semantics definition in MDE - an instrumented approach for model verification. JSW 4(9), 943–958 (2009)
29. van Deursen, A., Klint, P.: Little languages: little maintenance. Journal of Software Maintenance 10, 75–92 (1998)
30. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. SIGPLAN Not. 35, 26–36 (2000)
31. Dijkstra, E.W.: Letters to the editor: go to statement considered harmful. Commun. ACM 11, 147–148 (1968)
32. Eclipse. Xtext (2012), `http://www.eclipse.org/Xtext` (accessed February 20, 2012)
33. Ekman, T., Hedin, G.: The JastAdd extensible Java compiler. ACM SIGPLAN Notices 42(10), 1–18 (2007)
34. Eysholdt, M., Behrens, H.: Xtext: implement your language faster than the quick and dirty way. In: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, SPLASH 2010, pp. 307–309. ACM, New York (2010)
35. FALCON. Falcon project – "System-of-systems" performance and reliability in logistics (2012), `http://www.esi.nl/research/applied-research/current-projects/falcon/index.dot` (accessed February 21, 2012)
36. Farail, P., Gaufillet, P., Canals, A., Camus, C.L., Sciamma, D., Michel, P., Crégut, X., Pantel, M.: The TOPCASED project: a Toolkit in OPen source for Critical Aeronautic SystEms Design. In: Embedded Real Time Software – ERTS 2006, SIA, SEE, AAAF (2006)
37. Feuerstein, S., Pribyl, B.: Oracle PL/SQL Programming, 4th edn. O'Reilly Media, Inc. (2005)
38. Frost, R.A., Hafiz, R., Callaghan, P.: Parser Combinators for Ambiguous Left-Recursive Grammars. In: Hudak, P., Warren, D.S. (eds.) PADL 2008. LNCS, vol. 4902, pp. 167–181. Springer, Heidelberg (2008)
39. Gansner, E.R., North, S.C.: An open graph visualization system and its applications to software engineering. Software: Practice & Experience 30(11), 1203–1233 (2000)
40. Groenewegen, D.M., Hemel, Z., Kats, L.C.L., Visser, E.: WebDSL: A domain-specific language for dynamic web applications. In: Mielke, N., Zimmermann, O. (eds.) Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2008), pp. 779–780. ACM, New York (2008) (poster)
41. Gronback, R.C.: Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit, 1st edn. Addison-Wesley Professional (2009)
42. Grune, D.: Parsing Techniques: A Practical Guide, 2nd edn. Springer Publishing Company, Incorporated (2010)

43. Hamberg, R., Verriet, J.: Automation in Warehouse Development. Springer (2011)
44. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The Syntax Definition Formalism SDF — reference manual. ACM SIGPLAN Notices 24, 43–75 (1989)
45. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley (2003)
46. ISO. ISO/IEC 19502:2005 information technology – Meta Object Facility (MOF) (2005)
47. Johnson, S.C.: YACC—yet another compiler-compiler. Technical Report CS-32, AT & T Bell Laboratories, Murray Hill, N.J. (1975)
48. Johnstone, A., Scott, E., van den Brand, M.G.J.: LDT: a language definition technique. In: Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications, LDTA 2011, pp. 9:1–9:8. ACM, New York (2011)
49. Jones, S.P. (ed.): Haskell 98 Language and Libraries: The Revised Report (September 2002), `http://haskell.org/`
50. de Jong, H.A., Olivier, P.A.: Generation of abstract programming interfaces from syntax definitions. Journal of Logic and Algebraic Programming 59(1-2), 35–61 (2004)
51. Kats, L.C., Visser, E.: The spoofax language workbench: rules for declarative specification of languages and ides. SIGPLAN Not. 45, 444–463 (2010)
52. Kleppe, A.: Software Language Engineering: Creating Domain-specific Languages Using Metamodels. Addison-Wesley (2009)
53. Klint, P., van der Storm, T., Vinju, J.: EASY Meta-programming with Rascal. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2009. LNCS, vol. 6491, pp. 222–289. Springer, Heidelberg (2011)
54. Kunert, A.: Semi-automatic generation of metamodels and models from grammars and programs. Electron. Notes Theor. Comput. Sci. 211, 111–119 (2008)
55. Kurtev, I.: Adaptability of Model Transformations. PhD thesis, University of Twente, Enschede, The Netherlands (2005)
56. Lamport, L.: Latex: a document preparation system. Addison-Wesley Longman Publishing Co., Inc., Boston (1986)
57. Landin, P.J.: The next 700 programming languages. Commun. ACM 9, 157–166 (1966)
58. LEGO. Lego Mindstorms (2012), `http://www.lego.com/eng/education/mindstorms/` (accessed February 21, 2012)
59. Lesk, M.E., Schmidt, E.: Lex–a lexical analyzer generator, pp. 375–387. W.B. Saunders Company, Philadelphia (1990)
60. Mauw, S., Wiersma, W., Willemse, T.: Language-driven system design. International Journal of Software Engineering and Knowledge Engineering 14(6), 625–664 (2002)
61. Mens, T., Van Gorp, P.: A taxonomy of model transformation. Electron. Notes Theor. Comput. Sci. 152, 125–142 (2006)
62. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Computing Surveys 37(4), 316–344 (2005)
63. Moreau, P.-E., Ringeissen, C., Vittek, M.: A Pattern Matching Compiler for Multiple Target Languages. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 61–76. Springer, Heidelberg (2003)
64. Mosses, P.D.: Action semantics. Cambridge University Press, New York (1992)
65. OMG. Unified Modeling Language specification, version 1.3 (2001), `http://www.omg.org/spec/UML/1.3/PDF/index.htm` (accessed February 21, 2012)
66. OMG. Meta Object Facility specification. Technical Report 2002-04-03, Object Management Group (2004)

67. OMG. OCL (2012), http://en.wikipedia.org/wiki/Object_Constraint_Language (accessed February 22, 2012)
68. Open Directory Project. Links for lexer and parser generators, http://www.dmoz.org/Computers/Programming/Compilers/Lexer_and_Parser_Generators/ (accessed on February 22, 2012)
69. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Bookshelf (2007)
70. Parr, T., Fisher, K.: LL(*): the foundation of the ANTLR parser generator. SIGPLAN Not. 46, 425–436 (2011)
71. Pierce, B.C.: Types and programming languages. MIT Press, Cambridge (2002)
72. Plotkin, G.: A Structual Approach to Operational Semantics. Journal of Logic and Algebraic Programming (2004)
73. Reilles, A.: Canonical Abstract Syntax Trees. In: 6th International Workshop on Rewriting Logic and Applications, WRLA 2006, Vienna, Autriche. Carolyn Talcott and Grit Denker (2006)
74. Rekers, J.: Parser Generation for Interactive Environments. PhD thesis, University of Amsterdam, Amsterdam, The Netherlands (January 1992)
75. Rivera, J., Durán, F., Vallecillo, A.: Formal specification and analysis of domain specific models using Maude. Simulation 85(11-12), 778–792 (2009)
76. Rumbaugh, J., Jacobson, I., Booch, G. (eds.): The Unified Modeling Language reference manual. Addison-Wesley Longman Ltd., Essex (1999)
77. Rusu, V., Lucanu, D.: A $\mathbb{K}$-Based Formal Framework for Domain-Specific Modelling Languages. In: Proc. of 2nd International Conference on Formal Verification of Object-Oriented Systems (FoVeOOS 2011), Torino, Italy, pp. 306–323. Springer (2011)
78. Scheidgen, M.: CMOF-model semantics and language mapping for MOF 2.0 implementations. In: Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software, pp. 84–93. IEEE Computer Society, Washington, DC (2006)
79. Scott, E., Johnstone, A.: GLL Parsing. Electronic Notes in Theoretical Computer Science 253(7), 177–189 (2010)
80. Seidewitz, E.: What Models Mean. IEEE Software 20(5), 26–32 (2003)
81. Selic, B.: A systematic approach to domain-specific language design using uml. In: IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, pp. 2–9 (2007)
82. Stappers, F.P.M., Weber, S., Reniers, M.A., Andova, S., Nagy, I.: Formalizing a Domain Specific Language Using SOS: An Industrial Case Study. In: Aßmann, U. (ed.) SLE 2011. LNCS, vol. 6940, pp. 223–242. Springer, Heidelberg (2012)
83. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0, 2nd edn. Addison-Wesley Professional (2009)
84. Stuurman, G.: Action Semantics applied to Model Driven Engineering. Master's thesis, University of Twente, The Netherlands (2010)
85. Theelen, B.D., Florescu, O., Geilen, M.C.W., Huang, J., van der Putten, P.H.A., Voeten, J.P.M.: Software/hardware engineering with the parallel object-oriented specification language. In: Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE 2007, pp. 139–148. IEEE Computer Society, Washington, DC (2007)

86. Tomita, M.: An efficient context-free parsing algorithm for natural languages. In: Proceedings of the 9th International Joint Conference on Artificial Intelligence, vol. 2, pp. 756–764. Morgan Kaufmann Publishers Inc., San Francisco (1985)
87. TRain. Train – The Railway Domain (2012), http://www.railwaydomain.org/ (accessed February 25, 2012)
88. TUDresden. EMFtext (2012), http://www.emftext.org/ (accessed February 20, 2012)
89. van Rossum, G.: An Introduction to Python for Unix/C Programmers. In: Proc. of the NLUUG Najaarsconferentie. Dutch UNIX users group (1993)
90. Visser, E.: Scannerless Generalized-LR Parsing. Technical Report P9707, Programming Research Group, University of Amsterdam (July 1997)
91. Visser, E.: Syntax Definition for Language Prototyping. PhD thesis, University of Amsterdam (1997)
92. van Vliet, H.: Software Engineering: Principles and Practice, 3rd edn. Wiley Publishing (2008)
93. Walkingshaw, E., Erwig, M.: A domain-specific language for experimental game theory. J. Funct. Program. 19, 645–661 (2009)
94. Wikipedia. Grace hopper — Wikipedia, the free encyclopedia (2012), http://en.wikipedia.org/wiki/Grace_Hopper (accessed February 18, 2012)
95. Wikipedia. Rush hour — Wikipedia, the free encyclopedia (2012), http://en.wikipedia.org/wiki/Rush_Hour_board_game (accessed February 25, 2012)