T.J.L. Wolterink

Operational Semantics Applied to Model Driven Engineering

Thesis *for the degree of*

Master Of Science

(Computer Science, Track Software Engineering)

Supervisors
Dr. I. Kurtev
Dr.Ir. K.G. van den Berg
A. Göknil

Faculty of Electrical Engineering,
Mathematics and Computer Science

Abstract

Model driven engineering (MDE) is an approach to software engineering and is used more and more in both research and practice. It provides a higher level of abstraction and supports domain specific languages (DSLs). There is much research done on the static aspects of MDE and most examples deal with static models. However DSLs are also used more and more to model the behaviour of a specific domain. These DSLs often lack clear and formal semantics.
Some approaches have been taken to define the semantics of modelling languages. These approaches differ in their pragmatic and formal aspects. None of these approaches are built upon the existing formalism of Structural Operational Semantics (SOS). SOS specifies the semantics of a language based on rules. These rules define a transition system in which the states are extended ASTs. This research adapts SOS in such a way that it can be applied on MDE models instead of ASTs. SOS seems to be a good candidate for specifying the semantics of modelling languages because it is based on the structure of programs which are well defined in a metamodel of a modelling language.
The main contribution of this thesis consists of a Semantic Language, *SemLang*, which is based on SOS. The Semantic Language is defined using an MDE approach: *SemLang* is defined as a metamodel. It closely resembles SOS but has some differences that make it suitable for defining the semantics of a DSL based on metamodels. A *SemLang* model is a set of rules that defines a transition system where the states are extended models. *SemLang* has constructs to deal with lists and graph structures. The Semantic Language is formalized by altering the existing formalizations of SOS.
Another result of this research is a tool which supports simulating and debugging of models given their semantics description in *SemLang*. The tool is built upon the Eclipse framework and provides a graphical user interface. As a proof of concept we apply our approach on some simple functional and imperative languages. We also show that this approach can be used on DSLs like Activity Diagrams and Petri Nets.

Acknowledgements

All the work done for this thesis could not be done without the help, encouragement and support of numerous people. Different people supported me in different ways and this combination ensured a pleasant working environment.
At first I want to thank my first supervisor, Ivan Kurtev, for his guidance before and during the project. Without his help I would have had difficulty discovering the interesting world of academic research. I also admire his calmness and informality, even when he had to supervise numerous other graduation students. I would also thank my other supervisors, Arda Göknil and Klaas van den Berg, for taking the time to read and review my thesis.
Secondly I want to thank my family; my brothers and especially my parents for supporting me throughout my years at the university. Even though it is difficult for them to understand the work done in my thesis I am sure they will try to read some parts of this thesis. Especially this paragraph.
Another important ingredient were my fellow students from room 5066 (formerly 5070). The working atmosphere in that room was great on they provided help when needed. The discussions during lunch and during work were both entertaining and occasionally also informative.
At last I want to thank anybody that I forgot to thank. This includes the authors of the interesting papers I have read and of course anybody that showed interest in my work as a master student.

Tjerk Wolterink, August 2009, Enschede

"The sciences do not try to explain, they hardly even try to interpret, they mainly make models. By a model is meant a mathematical construct which, with the addition of certain verbal interpretations, describes observed phenomena. The justification of such a mathematical construct is solely and precisely that it is expected to work."

John Von Neumann (1903 - 1957)

Table of Contents

List of Figures

List of Abbreviations

ASM	Abstract State Machine
AST	Abstract Syntax Tree
BNF	Backus–Naur Form
DSL	Domain Specific Language
DSM	Domain Specific Modelling
EMF	Eclipse Modelling Foundation
EMOF	Essential Meta Object Facility
GME	Generic Modelling Environment
GMF	Graphical Modelling Framework
GPL	General Purpose Language
LTS	Labelled Transition System
LTTS	Labelled Terminal Transition System
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MOF	Meta Object Facility
MSOS	Modular Structural Operational Semantics
OCL	Object Contraint Language
OMG	Object Management Group
QVT	Query View Transformation
RCP	Rich Client Platform
SemLang	Semantic Language
SI	Structural Induction
SOS	Structural Operational Semantics
SWT	Standard Widget Toolkit
TCS	Textual Concrete Syntax
TSS	Transition System Specification
UML	Unified Modelling Language
XML	Extended Markup Language

1 Introduction

1.1 Background

Modelling is used to understand processes and systems in the real world. Different formal and informal modelling languages have been introduced. This includes petri nets, state charts and UML . Programming languages and human languages on the other hand can also be seen as modelling languages. These languages differ in their formality, ambiguity and application domain.
A new approach to software development is Model Driven Engineering (MDE) [Ken02]. MDE raises the level of abstraction by introducing metametamodels [BÓ4]. When defining a language without MDE one first specifies the concepts of the language (either mathematically or by using a syntax specification). Then one defines the semantics of the language in either an informal or formal way.
With MDE one specifies a modelling language by creating a metamodel which describes the structures of the models. A textual or graphical concrete syntax can be created if needed (for example with TCS [JBK06]). However, there is no commonly established way of specifying the semantics of an executable modelling language. A simple approach is to use code generation, however, this approach is rather informal. Some research is done in the area of formal semantics for MDE; different authors apply different formal frameworks to solve this issue [SW08, Hec06, BH02, RRDV07, RV07, CCG+08, VPF+06, CSAJ05].
No research to adapt the Structural Operational Semantics (SOS) [Plo81] formalism to MDE is done in the past. This formalism seems a fruitful approach for MDE because the SOS formalism is built upon the structure of a modelling language. MDE has metamodels which clearly define the structure of a modelling language.

1.2 Problem Statement

Adapting SOS in order to make it useful in an MDE context is not straight forward. The first problem is the incompatibility between SOS and MDE; SOS is applied on trees while MDE models are graph structures. However, we think it is possible to apply SOS, with some changes, to MDE. This leads to the following research question:

Can SOS be adapted and applied successfully on MDE?

The adapted SOS will be called the Semantic Language (abbreviated to *SemLang*). The main research questions can be divided into several sub-questions

that need to be answered. The first sub-question is:

MDE models are graph structures but SOS is based on trees. Can SOS be adapted in order to make it suitable for using it with graph structures?

The main problem is the differences between the programs in SOS and MDE. In SOS the program is represented as a (extended) abstract syntax tree. In MDE, however, the program is represented as a model which is basically a more general graph structure. This difference is a problem that needs to be solved.

SOS is based on the abstract syntax description of languages, how can SOS be adapted to deal with metamodels?

The rules in SOS refer to the abstract syntax of a language in order to transform language structures to new language structures. In MDE metamodels have the same role as the abstract syntax description. This difference requires changes to SOS in order to apply it to MDE.

The semantic domain of SOS is a labelled transition system in which the states are trees. How can we change the semantic domain in order to make it suitable for models?

A semantic domain is a well known mathematical domain in which the semantics is expressed (for more details see section 2.5). The semantic domain of SOS is specified in different papers [Plo81, AFV01, Mos04]. The semantic domain of SOS is not suitable for defining the semantics of MDE modelling languages.
It is of course difficult to test whether the main hypothesis is met in the end of the thesis. Therefore we limit the test to a set of DSLs which have different semantic properties. This includes imperative and functional languages but also graph like languages like petri nets and state charts. Another important result is to point out what the difficulties are when applying SOS to MDE.

1.3 Research Objectives

The research objectives are listed below. A description of each objective is given.

- Adapt SOS in order to make it suitable for MDE

  The objective of this research is to adapt SOS in order to make it useful for MDE. This includes adapting the SOS formalism and specifying how it is related to the metamodel of the modelling language. The goal is to show that MDE and SOS are a good match by proof of concept.

- Introduce a new semantic language which is based on SOS

  The main work in order to reach that objective is to develop a new semantic language based on SOS which can be used to define the semantics of modelling languages. The focus should be on the pragmatic aspects.

- Provide a partial formalization of the new language

  In order to provide some mathematical foundation for our work an important contribution is the partial formalization of the new language. This formalization can be built upon existing formalizations of SOS.

- Implement a tool that support simulating and debugging of models

Another major objective is the development of a tool. The tool is needed in order to make the SOS based language useful. The tool should be able to load and simulate models given their semantics description. The addition of a graphical user interface is also desirable.

- Validate the research by applying the new language on some example DLSs

  The new language will be applied on several example DSLs in order to illustrate that the research goal is met. These example DSLs should cover different language types, like imperative and functional language. Part of this objective is to apply the new language on graph based languages like Activity Diagrams.

1.4 Contributions

Each objective resulted in at least one contribution. The contributions are listed below with a description accompanying each contribution.

- A new *Semantic Language* is introduced

  This work introduces a *Semantic Language* which can be used to specify the dynamic semantics of modelling languages. The Semantic Language builds upon SOS and is therefore greatly influenced by the structure of SOS rules. However, the Semantic Language has features that make it suitable for defining the semantics of modelling language defined using MDE concepts. The Semantic Language itself also follows the MDE principles; the language is defined as a DSL. It comes with a concrete syntax which makes it easy to define the semantics of a DSL.

- A partial formalization of the *Semantic Language* is given

  The known semantics of SOS are adapted in order to provide the mathematical foundations for the Semantic Language. The main differences are in the semantic domain and in the rules. The states in the semantic domain of the *Semantic Language* are basically extended graph structures. The rules consist of terms that refer to metamodel elements instead of AST elements.

- A solution to the problem of applying SOS with graphs

  The main difference between the Semantic Language and plain SOS is that the Semantic Language is suitable for defining the semantics of graph based models. The main changes that were needed was the introduction of a breath-first-search based copy algorithm. This copy algorithm ensures that nodes are not copied more that once, thus preventing infinite recursion.

- An implementation of a graphical tool called *Semantic Engine*

  Another major part of the work done for this thesis is the implementation of a tool called the *Semantic Engine*. This tool is an implementation of the Semantic Language and it provides a complementary graphical user interface in order to simulate models for which the semantics are specified. The tool also provides debugging functions like state inspection, step-by-step simulation and a visualization of transition proof trees.

- The *Semantic Language* is applied to several example DSLs

  Another contribution is the application of the *Semantic Language* to several example DSLs. This is done as a proof of concept. The DSLs cover a range of language types; functional and imperative languages but also graph based languages like Petri Nets and Activity Diagrams.

Different approaches for defining the semantics of DSLs already existed. However, nobody tried adapting SOS in order to make it suitable for defining the semantics of DSLs. This thesis provides a first approach in combining SOS and MDE.

1.5 Outline

Chapter 2 introduces all basic concepts that are used in this thesis. It explains MDE and highlights some approaches in the MDE field. Secondly the use of domain specific languages is explained. The remaining of the chapter focuses on the semantics of languages. The structural operational semantics (SOS) approach to semantics is explained in more depth.
Chapter 2.7 takes a look at the current approaches to specifying the semantics of models in an MDE context. Some current approaches are explained. These approaches must be explained to place this thesis into a proper context, it also provides some comparison material.
Chapter 3 introduces a DSL named *SemLang* which can be used to specify the semantics of other DSLs. The DSL is based on SOS but has some nice features that make it suitable in an MDE context. The chapter also provides some example DSLs on which *SemLang* is applied.
Chapter 4 formalizes the DSL *SemLang* DSL. It builds upon existing formalizations of SOS. The formalization provides a solid mathematical framework on which the *SemLang* DSL is built.
Part of this research was also the implementation of a tool which could simulate DSLs for which the semantics are defined using *SemLang*. The tool requirements, architecture and evaluation can be found in chapter 5.
The evaluation of the work done in this thesis can be found in chapter 6. Our approach is compared to the existing approaches for defining semantics of models. The conclusion of this paper is in the last chapter (chapter 7) in which the answers to the research questions can be found.
This paper can be read in different ways. To accommodate the reader a diagram of the dependencies between the chapters is given in figure 1.1. The *Basic Concepts* chapter may be skipped if the reader is already familiar with MDE and SOS concepts.

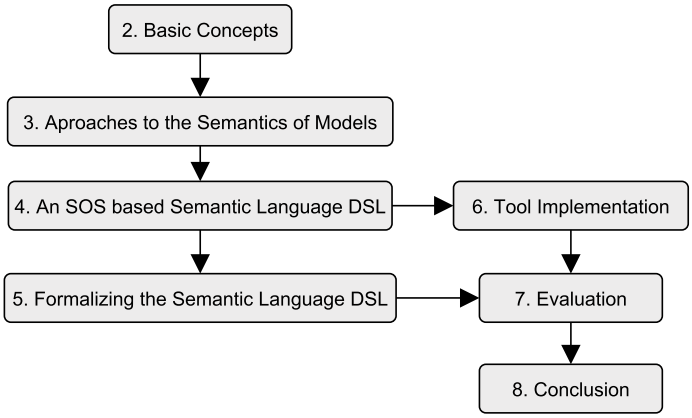


Figure 1.1: Dependencies between chapters

2 Basic Concepts

2.1 Introduction

The main aim of the thesis is to introduce a framework which can be used to define the semantics of modelling languages in a formal way, therefore it is important to investigate the basic concepts in this area. First the main concepts in the Model Driven Engineering (MDE)[Ken02] field are explained. The concepts of *model* and *metamodel* and the relations between them are explained. Subsequently some approaches to MDE are explained and their differences and similarities are discussed. In section 2.4 domain specific languages (DSLs) and their relation to the MDE field are covered. Novices in the MDE field may skip the first two sections.
In the end the current state of the art in the specification of the semantics of programming languages is explored. The knowledge in this area can be used to specify semantics for modelling languages. The next chapter (chapter 2.7) explores how the semantics of DSLs are currently specified.

2.2 Model Driven Engineering

Model Driven Engineering (MDE) provides a higher level of abstraction with respect to software engineering. Analogous to the principle *everything is an object* in object technology, MDE embraces the principle that *everything is a model* [Bó4]. The notion of a model is a powerful unification concept in MDE, therefore it is important to know what a model is and how modelling languages are defined.

2.2.1 Models and Metamodels

A model represents a system (part of the reality), and is expressed in a modelling language. In MDE the structure of the modelling language is given by another model, called its metamodel. This is a generalization of what is common in computer science (e.g., a *Java program* conforms to the *Java grammar*).
Models and metamodels can be placed in layers, by MDE convention the reality is in layer *M0*, models that represent the reality are in layer *M1* and the metamodels of those models are in layer *M2*.
Some informal definitions are given below which are inspired by the definitions analyzed by Kurtev [Kur05]:

Definition 1 (Model) *A model (at M1) is an abstraction of a part of the reality for a specific purpose. A model is expressed in a modelling language.*

Definition 2 (Modelling language) *A modelling language is a well understood (not always formal) language which describes the concepts and their relation of a part of reality.*

Definition 3 (MetaModel) *A metamodel (at M2) is a model of a modelling language.*

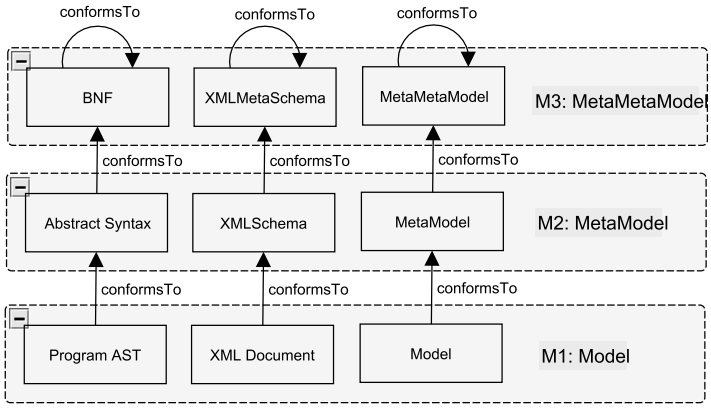


Figure 2.1: Layers in MDE

Because a metamodel is a model it is expressed in a modelling language. The model of this modelling language is the *metametamodel*. This suggests infinite iteration, to limit the iteration of the *conformsTo* relation a metametamodel is said to conform to itself (e.g., the syntax of BNF is defined in itself). The metametamodelling layer is layer *M3*. The relations between the layers (M1, M2 and M3) can be seen in figure 2.1. This figure also shows that existing technologies like BNF and XMLSchema fit into this view.

Definition 4 (MetaMetaModel) *A metametamodel (at M3) is a model of a modelling language which can be used to express metamodels. The model of the modelling language of the metametamodel is expressed by itself.*

In MDE all metamodels ideally conform to the same metametamodel in the *M3* layer. This supports the principle that everything is a model. The metametamodel defines the set of elements and references that are allowed for specifying metamodels. A metamodel is well-formed if it $conformsTo$ the metametamodel. This section introduced some concepts in MDE. However, these concepts are not formally defined. The meaning of the $conformsTo$ relation is not specified (i.e. when does a model conform to a metamodel?). In [JB06] and [TCCG07] attempts have been made to formalize these concepts. These attempts only formalize simple metamodels and models: the metametamodel is kept small too keep the formal framework simple. However, these formal frameworks do give some insight into the formal properties of MDE. The definitions as proposed by Thirioux et al [TCCG07] will be explained below.

2.2.2 Model Definition

Informally a model represents something in the real world and conforms to a MetaModel. The definition proposed by Thirioux et al [TCCG07] defines a model as follows.

Definition 5 (Model) *Let $C \subseteq Classes$ be a set of classes.*
Let $R \subseteq \{\langle c_1, r, c_2\rangle | c_1, c_2 \in C, r \in References\}$ be a set of references among classes such that:

$$\forall c_1 \in C, \forall r \in References, |\{c_2 | \langle c_1, r, c_2\rangle \in R\}| \leq 1$$

We define a model $\langle MV, ME\rangle \in Model(C, R)$ as a multigraph over a finite set MV of typed objects and a finite set ME of labelled edges such that:

$$MV \subseteq \{\langle o, c\rangle | o \in Objects, c \in C\}$$

$$ME \subseteq \{\langle\langle o_1, c_1\rangle, r, \langle o_2, c_2\rangle\rangle | \langle o_1, c_1\rangle, \langle o_2, c_2\rangle \in MV, \langle c_1, r, c_2\rangle \in R\}$$

The definition shows that a model is basically a special kind of graph. The $conformsTo$ relation is not specified in the model definition. As we will see in the next sub-section this is specified in the metamodel.

2.2.3 Metamodel Definition

Now we have a definition of a model we need a definition of a metamodel, again we follow the definition given by Thirioux et al [TCCG07]. A metamodel (also called a reference model in the paper of Thirioux) is defined as follows:

Definition 6 (MetaModel) *A metamodel is a multigraph representing classes and references as well as semantic properties over instantiation of classes and references. It is represented as a pair composed of a multigraph (RV, RE) built over a finite set RV of classes, a finite set as a RE of labelled edges, with a property over models represented as a predicate function.*
We define a reference model as a triple
$\langle (RV, RE), conformsTo\rangle \in metaModel$ such that:

$$MMV \subseteq Classes$$

$$MME \subseteq \{\langle c1, r, c2\rangle | c1, c2 \in MMV, r \in References\}$$

$$conformsTo : Model(MMV, MME) \rightarrow Bool$$

The model is only valid if the following condition holds:

$$\forall c_1 \in MMV, \forall r \in References, |\{c_2 | \langle c_1, r, c_2\rangle \in MME\}| \leq 1$$

where $|A|$ is the cardinality of a set A. Informally this condition says that a specific reference from a class can only go to maximal one other class. The $conformsTo$ function specifies the semantic properties over instantiations of the metamodel. This function is not further specified here. It simply evaluates to true when a model conforms to the metamodel. For an example of the $conformsTo$ relation with respect to EMOF we refer to Thirioux et al [TCCG07].

Again it is important to understand that different MDE approaches use different metametamodels and thus allow different metamodels and models. The metametamodel in the formal framework of Thirioux only contains classes and references. Most MDE approaches also allow attributes, class hierarchies and references with multiplicities.

2.3 MDE Approaches

MDE and its concepts where introduced by the Object Management Group (OMG) [Sol00]. However, other approaches where proposed as well. This section will discuss the main approaches to MDE and their differences and similarities. First the initial approach of the OMG is covered. Then the approach taken by the Microsoft Software Factory Tools (MS/DSL) [GSCK04] is discussed. In the end two similar approaches Ecore [ECO], from the Eclipse Modeling Foundation [EMF] , and KM3 [JB06] are covered.

2.3.1 Object Management Group

The Object Management Group (OMG) introduced the Model Driven Architecture (MDA^{TM}) . They started the development of a metametamodel called the Meta Object Facility (MOF) . This resulted in MOF 1.4 [OMG02] and MOF 2.0 [OMG03a]. This language is designed in such a way that it can be used as the metametamodel for UML [OMG]. However, this comes at a higher complexity. The OMG understood that a simpler metametamodel was needed and therefore defined Essential MOF (EMOF) as a subset of MOF. However, even EMOF is seen as rather big by the MDE research community. As we will see a simpler metamodel facilitates understanding and makes it a better candidate for formalization.
The semantics of MOF is informally described using text. Ambiguous interpretation of MOF can therefore not be prevented and may cause inconsistent tool implementations. The OMG approach is a rather ambitious approach and tries to cover all aspects of MDE, however, in our view it is better to start with a simple pragmatical MDE approach and extend that if needed instead of enforcing a specific full blown approach.

2.3.2 MS Software Factory Tools

The Microsoft tools for Domain Specific Languages (MS/DSL) is a suite of tools to support for creating and using domain specific data for automating the software development process [GSCK04]. MS/DSL takes a pragmatic approach: the main focus is on tool support. The MS/DSL does not have a rigid specification, in fact there is no explicit metametamodel defined [BHJ+05].
MS/DSL is a proprietary platform and its main aim is to generate code. It therefore comes accompanied with a code generator. However, the aim of MDE is much broader than merely code generation. For example there is no model transformation language in MS/DSL. The biggest disadvantage is that it is a closed platform and it does not have a large research community.
An advantage of MS/DSL is that it takes a pragmatical approach to MDE. It comes with a lot of tools that makes the life of the MDE developer easier.

This may be the reason that there is no explicit metamodel defined, this only makes the tool complex. This may be appealing for beginners in the MDE field; however, in the long run an explicit defined metamodel is really beneficial.

2.3.3 Eclipse Modelling Framework

The Eclipse Modelling Framework (EMF) project is a large modelling and code generation framework which can be used in an MDE context [EMF]. The project tries to align itself with the OMGs MDA approach. But an important aim of the project is to be useful in a practical sense. However, this does not undermine the theoretical aspects like we see with the MS Software Factory Tools. In fact the metamodel, named ECORE [ECO] is explicitly defined and is almost identical to EMOF.
EMF core advantage is that it has a rich user community and a lot of tools that can be used. There are code generation frameworks, model2text and model2model transformation engines, and rich editors available. This richness makes EMF the most commonly used MDE platform.
Another advantage is that there are some MDE approaches which are integrated with the EMF Framework. For example the Topcased Toolkit [VPF+06] is such an approach. Also the approach by Bézivin which uses KM3 [JB06] as a metamodel has an implementation which allows models and metamodels to be transformed to models and metamodels that fit in the EMF framework.
The KM3 approach also comes with a technique which can be used to define a concrete syntax for a modelling language. Jouault et al for example provides a way to define the concrete syntax for a specific metamodel using TCS [JBK06].

2.4 Domain Specific Languages

Most programming languages are general purpose languages (GPLs) , they are very rich and can be used in any problem domain. MDE can be used to specify the metamodel of a GPL, however, the real power of MDE is the decoupling of the model from the technical platform.
Programs often solve problems in some problem domain. The program, however, is some programming language that has (often) no relation with the problem domain. The translation of the requirements in the problem domain to the actual program is a difficult and intensive task.
Suppose that the problem domain could actually be modelled in some language specific to that domain. Such a language is called a Domain Specific Language (DSL). Such a language closes the gap between the problem and the implementation. In fact DSLs have been successfully used in some problem domains [vDKV00].
Domain specific languages (DSLs) model the domain of a specific problem and therefore the software model is closer connected to the domain. A DSL uses domain specific notations and constructs and or mostly smaller than GPLs. This increases productivity and enlarges the user base [MHS05, Hud98]. The smaller size of DSL programs compared to GPL programs also make the use of analysis, verification, optimization, parallelisation and transformation more feasible [MHS05].

A problem with DSLs is that they are difficult to design and implement and require higher initial costs [Hud98]. Here is where MDE comes into play: DSLs and MDE are a perfect match. The MDE approach makes the specification of a DSL much easier. Creating a DSL in the MDE context is often called Domain Specific Modelling (DSM) [KT08].

2.4.1 MetaModel

The syntax of textual languages is mostly defined using Bachus-Naur Form (BNF). This is also visualized in figure 2.1. The grammar of a language is the metamodel of the language and BNF itself is a metametamodel, in fact BNF can be described in itself.
When defining a model for a specific domain one uses domain concepts to express the model. These domain concepts must be captured in a metamodel for that domain. The metamodel for a DSL is analog to the grammar for a textual language. Therefore the structure of a DSL can is captured in a MetaModel and the concrete syntax can be captured using different tools within MDE (for example TCS [JBK06]).

2.4.2 Semantics

The metamodel for a specific domain only specifies the structure of the models. However, the semantic properties such as conditions over valid models and the behavioural semantics of a model (if any) are not specified. Semantics are as important as the structure of the language. Some argue that semantics are even more important than the structure [Hud98].
The semantics of a DSL can be specified in different ways. They can be specific informally using text or using model-to-code transformations. The semantics could also be specified in a formal way. More on the general aspects of semantics of languages can be found in section 2.5.

2.5 Semantics of Languages

Language definition deals with defining the *structure* of the language. However, the meaning of those structures must also be defined. This meaning is defined by the *semantics* of the language. The semantics of a language can be defined in different ways. This section explains what the semantics are and we take a look at different approaches for specifying semantics.

2.5.1 General Theory

Any language consists of a structure defined by some syntactic elements. Depending on the type of the language these elements can be words, sentences, boxes, diagrams etc. However, these structures do not have a meaning without semantics. The semantics of a programming language essentially models the computational meaning of each program [Mos06]. In order to define the meaning a mapping is defined from the syntax L of the language to a semantic domain S. This mapping $M : L \rightarrow S$ defines the meaning of the language [HR04].

So in order to define the semantics for a syntax definition on has to choose a semantics domain S and a mapping from the syntax to the semantic domain has to be made. This mapping defines the meaning of the syntactic elements.
The next two subsections will dive into two dimensions of semantics; the formality of the semantics and the dimension of static and dynamic semantics.

Formal versus Informal Semantics

Programming language tutorials often explain the meaning the language by explaining them with text and code examples. Those tutorials explain the semantics in an informal manner which is easier for the reader.
However, the disadvantages of informal semantics are that they can be ambiguous and imprecise. The semantic domain and the mapping from syntax to the domain are not explicitly specified.
Formal semantics on the other hand have an explicit semantic domain, which is often a well known mathematical domain like natural numbers, graphs or labelled transition systems. The mapping to that domain is also explicitly given. Formal semantics are precise and computer readable. They can also be combined with rigorous mathematical techniques like validation, proofs, simulation etc.

Static and Dynamic Semantics

The semantics of a language can be divided into static and dynamic semantics. The static semantics deals with compile-time semantics like type checking and well-formedness constraints. These semantics are called static because these semantics can only be checked before running the program.
The dynamic semantics on the other hand models the runtime behaviour of the program. All observable behaviour of a running program is defined by the dynamic semantics.

2.5.2 Approaches

There are several approaches to specifying dynamic semantics. Some of these approaches are denotational semantics, axiomatic semantics, Abstract State Machine semantics and action semantics [Mos06]. The axiomatic and denotational approaches are discussed briefly below. However, the focus of this thesis is on another approach named structural operational semantics (SOS). This approach is explained in a separate section.

Axiomatic Semantics

Axiomatic semantics [Hoa69] is based on mathematical logic. The semantics of a language are specified by making general assertions (or axioms) about syntactic structures in the language. These assertions are known as *Hoare triples* written as $\{P\}S\{Q\}$. They consist of a precondition P, a syntactic structure S and a postcondition Q. An example assertion is for example:

$$\{x = A\}x := x + 1\{x = A + 1\}$$

This assertion states that if x equals A, then after execution of the statement $x := x+1$ the variable x will equal $A+1$. The semantics of a language is defined

by a set of axioms which define the behaviour of all syntactic structures in the language.

Denotational Semantics

Denotational semantics was develop by Scott and Strachey at Oxford [SS71, Sch86]. In denotational semantics the meaning of a language is defined by a set of functions, or *denotations*. Each denotation of a construct has a set of arguments that represent the information before its execution and a result that represents the information available after its execution.
Denotational semantics has been used to define the semantics of functional programming languages. However, attempts to give semantics to larger programming languages have been less successful [Mos06].

2.6 Structural Operational Semantics

Structural Operational Semantics (SOS) was first introduced by Plotkin in 1981 [Plo81] and has been researched further by other researchers [Hen90], notably Mosses [Mos02, Mos04]. SOS has been used to define the semantics of process algebras [Mil90] and programming languages [IPW01]. SOS is a compositional: the semantics of a phrase is specified by the semantics of the subphrases. Compositional rules are used to specify the semantics of a specific language structure.

2.6.1 Semantic Domain

The semantic domain of SOS is a labelled transition system (LTS) (see definition 7), which is a well-studied mathematical object.

Definition 7 (Labelled Transition System) *A LTS is a triple* $(Q, A, \rightarrow)$ *with* Q *as set of states, a set* A *of labels* α*, a relation* $\rightarrow \subseteq Q \times A \times Q$ *of labelled transitions (*(s, α, s') *is written as* $s \xrightarrow{\alpha} s'$*).*

The mapping from the syntactic domain of a language to the semantic domain is done by defining a SOS specification. A SOS specification consists of a definition of the states and a set of rules which define the transition relation in the LTS. The states are often sentences of the language with some extensions.

2.6.2 Rules

SOS rules are based on the abstract syntax of a language. A rule consist of assertion of transitions $t \xrightarrow{\alpha} t'$ where the terms t, t' are syntactic constructs of the language which contain meta-variables and α is the label of the assertion. A rule is written as follows:

$$\frac{c_1, \cdots, c_n}{c} \tag{2.1}$$

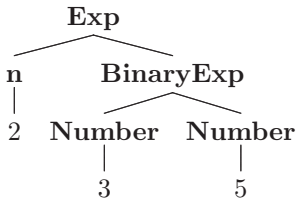
Where $c, c_1, \cdots, c_n$ are transition assertions in which $c_1, \cdots, c_n$ are the conditions and c is the conclusion of the rule.

2.6.3 Example

In order to understand the rules the abstract syntax and SOS rules for a simple expression language will be given. Consider the abstract syntax of a simple expression language as defined below:

$\langle \text{Exp} \rangle \rightarrow \text{n} \mid \langle \text{BinaryExp} \rangle$
$\langle \text{BinaryExp} \rangle \rightarrow \langle \text{Exp} \rangle \ \langle \text{Exp} \rangle$

This is the abstract syntax for a simple language in which the sentences are numbers n or binary expressions. In order to keep the language simple we did not add any operators for the binary expressions. In this case a binary expression is simply an expression which has two sub expressions. A simple sentence of this language is visualised as an abstract syntax tree below:



This simple language will be used to illustrate how SOS is used to define the semantics of a language. In order to specify the semantics for this simple language we must know what the intended behaviour of the language is. In this case we simply evaluate a binary expression to the sum of its sub-expressions. Therefore a sentence in this language will always evaluate to a number.
The SOS rules that specify this behaviour can be seen below. The rules contain meta-variables x, e_i and n_i. The meta-variable x matches any syntactic construct, the meta-variable e_i only matches binary expressions and the meta-variable n_i only matches numbers.

$$\frac{e_1 \rightarrow e_1'}{\textbf{BinaryExp}\,(e_1, x) \rightarrow \textbf{BinaryExp}\,(e_1', x)} \tag{2.2}$$

Rule 2.2 consist of a condition $e_1 \rightarrow e_1'$ and a conclusion. The condition can be interpreted as: *there is a rule which transforms the syntactic construct bound to* e_1 *to a new syntactic construct* e_1'. The conclusion of the rule simply states that in order to evaluate a binary expression one must first evaluate the first child expression.

$$\frac{e_2 \rightarrow e_2'}{\textbf{BinaryExp}\,(n_1, e_2) \rightarrow \textbf{BinaryExp}\,(n_1, e_2')} \tag{2.3}$$

Rule 2.3 matches binary expressions in which the first child is already evaluated to a number (which is bound to meta-variable n_1). The rule states that if the second child, bound to e_2, of a binary expression evaluates to e_2' then the binary expression will evaluate to a new binary expression in which e_2 is substituted by e_2'.

$$\frac{n = n_1 + n_2}{\textbf{BinaryExp}\,(n_1, n_2) \rightarrow n} \tag{2.4}$$

The last rule, rule 2.4, evaluates a binary expression, with two numbers n_1 and n_2 as children, to the sum of those numbers. The condition $n = n_1 + n_2$ is not a transition condition but simply an assertion.
Note that the three rules presented here do not strictly follow a specific rule format. They are just presented here to give you an intuitive understanding of SOS. To get a more in depth understanding of SOS it is recommended to read some papers on SOS, for example the paper by Plotkin [Plo81].

2.6.4 Transition System Specification

The semantic domain of SOS is a LTS. The states of the LTS are extended syntactic constructs of the language. The transitions between the states are specified inductively using the rules. Thus in fact an SOS specification specifies a LTS and is therefore called a Transition System Specification (TSS) . For a formal definition of TSSs and SOS we refer to papers by Mousavi et al [MRG07]. In this introduction we will just explain how a transition in the LTS can be proved by a set of rules. Such a proof is called a proof-tree. We define a proof tree in a similar way as Peter Mosses does in his paper about Modular Structural Operational Semantics (MSOS) [Mos04].

Definition 8 (Transition Proof Tree) *Given a set of rules, a triple* (s, α, s') *is in the transition relation of the LTS if and only if a* ***finite upwardly branching tree*** *(the* ***proof tree****) can be formed that follows satisfies the following conditions:*

1. *All nodes are labelled by elements of* $Q \times A \times Q$
2. *the root nodes is labelled by* (s, α, s')
3. *for each node with n child nodes there is a rule* $\frac{c_1, \cdots, c_n}{c}$ *and an interpretation of the meta-variables such that*
   - *the label of the node is the interpretation of c*
   - *the labels of the child nodes are the interpretations of* $c_1, \cdots, c_n$.

The paper of Plotkin [Plo81] gives some examples of how to create a proof tree. Chapter 3 of this thesis shows how to create a proof tree using the proposed *Semantic Language*.

2.6.5 SOS Styles

Different authors use SOS in different application domains. However, this resulted in a range of SOS styles [MRG07]. In order to choose an appropriate rule format for the Semantic Language we must be aware of the current rule styles and their features.
One important distinction in SOS approaches is the *small-step* approach and the *big-step* approach. In the *small-step* SOS each transition generally corresponds to an indivisible item of information processing. In *big-step* SOS (also called *Natural Semantics*) a computation is a single transition that leads directly to a terminal configuration. For intuitive examples of both *big-step* and *small-step* rules we refer to a paper of Mosses [Mos04].

In general the *small-step* style will require a greater number of rules than the same specification in *big-step* style. However, the *small-step* style rules tend to be simpler. Another advantage of the *small-step* style is that it facilitates the description of interleaving. The *big-step* style can become ambiguous with respect to interleaving because the *big-step* style rules do not enforce a specific order.
Other important characteristics of different SOS styles are described in the paper by Mousavi et al [MRG07]. We will not list the SOS formats described in that paper (there are more than 17 formats). However, an important conclusion of the paper is that negative and infinite conditions are a complicating factor in SOS frameworks.

2.7 Approaches to defining the Semantics of Modelling Languages

2.7.1 Introduction

The problem of defining the semantics of a modelling language is not new. Some considerable amount of research has already been done, this has led to a number of approaches for defining the semantics. This chapter discusses existing approaches and gives advantages and disadvantages of the different approaches. This chapter also provides an initial bridge between the MDE and the semantics of languages in general as explained in the Basic Concepts chapter. It also provides comparison material which is used in the evaluation of this thesis.
The sections in this chapter will cover the existing approaches like model transformations, graph transformations, Maude, the Topcased Toolkit and semantic anchoring.

2.7.2 Model Transformations

A powerful technique in MDE is model transformations. The unifying concept in MDE "*Everything is a model*" makes the description of model to model transformation a relatively easy process. One standard for model transformations is the QVT (Query/View/Transformation) [OMG08] language as proposed by the Object Management Group (OMG).
The model transformation technique is used by Sadelik and Wachsmuth [SW08] to specify the semantics of a modelling language. The approach uses a LTS as the semantic domain. The states (or *configurations* as called by Sadelik and Wachsmuth) are models that conform to the *configuration metamodel*. The transitions in the LTS are specified by a QVT model-to-model transformation. An *initialisation transformation* is also needed which transforms a model to the first initial state model.
The authors built an Eclipse plugin, named *EProvide*, which relies existing technologies like EMF [EMF], MOF [OMG02, OMG03a] and QVT. *EProvide* uses Graphical Modelling Framework [GMF] to visualize the intermediate states during simulation.
Scheiden and Fischer [SF07] use a similar approach as Sadelik and Wachsmuth. The main difference is in the way the model transformations are defined. Scheiden and Fischer use an action language based on UML activities and OCL

[OMG03b] to define the model-to-model transformation. The transformation is defined graphically which is an advantage according to the authors.

2.7.3 Graph Transformations

Graph transformations [Hec06, BH02]is similar to model transformations however as the name suggests, it is based on graphs as first class entities. To use graph transformations in MDE one has to create a *type-graph* from the language metamodel. The semantic domain is also a LTS. The states in the LTS are graph with a structure-preserving mapping to the type graph. Therefore it is required to transform a model to a graph before using it with graph transformations.
Similar to SOS, the semantics are specified using a set of rules. A rule basically consists of two graphs L and R. The first graph L represents the pre-conditions of the rule and the second graph R represents the post-conditions. Intuitively a rule matches some part of the state graph and deletes/adds edges and nodes in order to create a new state. A rule is often created with a visual graph editor which makes graph transformations inherently a visual approach to specifying semantics.
Because graph transformations are built upon graph theory it has a sound mathematical base. The semantic domain, the type-graphs and the rules are all formally defined. However a disadvantage of graph transformations is that the rules tend to become big; there is always a one-to-one relation between a transition in the LTS and a rule.

2.7.4 MDE with Maude

Maude is a high level language and an efficient rewriting engine that integrates functional programming with rewriting logic and provides metalanguage capabilities. Because of the facilities and capabilities of Maude it is used as a notation and semantic framework for specifying semantics of models and metamodels by Rivera et al [RRDV07, RV07, RGdLV08].
In the approach with Maude they first transform a language metamodel to Maude objects. Models are also represented as Maude objects and the Maude type system is used to check the validity of a model given its metamodel. The mapping from the MDE domain to the Maude domain is needed in order to use the Maude rewrite system for specifying the semantics of the modelling language.
The behavioural semantics of a modelling language are specified in Maude in terms of rewrite rules. These rules are added to the specification of the metamodel and model in Maude objects. The Maude rewrite rules are similar to the graph transformation rules but are based on *rewrite-theory* and are specified in a textual form.
The Maude team is currently working on more model operations and on better tool integration with tools like MOF, EMF and KM3 [OMG03a, EMF, JB06].

2.7.5 Simulation in the Topcased Toolkit

The Topcased Toolkit [CCG+08, VPF+06] is built upon the EMF Framework [EMF] framework takes a pragmatic approach toward the semantics of DSLs.

In order to use Topcased for a given modelling language one has to create different metamodels. The initial metamodel of the language is called the *static metamodel* within Topcased. In order to define the semantics one has to create an *event metamodel* which models all possible events that occur during simulation. A *dynamic metamodel* which relies on the *static metamodel* is used during the simulation. This *dynamic metamodel* may be used to specify some runtime elements that are needed to execute the model. Finally a *trace metamodel* is needed. This metamodel is used to define simulation scenarios as a sequence of events.
The actual semantics are defined in a pragmatic way: a component must be defined which implements the execution semantics using the programming language Java. EMF is used as a framework to update the structure of the model [EMF].

2.7.6 Semantic Anchoring

Semantic Anchoring [CSAJ05] uses yet another different approach. Essentially semantic anchoring uses model-to-model transformations to transform a model, for which the metamodel has no clear semantics, to a model for which the semantics is formally defined. One first defines a minimal modelling language L_i for a well known mathematical domain (for example Abstract Sate Machines (ASM)). The abstract syntax and semantics of this language must be precisely specified. In order to specify the semantics of a modelling language L a mapping (or model transformation) much be made between the abstract syntax of L to the abstract syntax of L_i. This mapping M_A is called the semantic anchoring of L to L_i.
The GME [GME] tool suite is used for defining the abstract syntax of DSLs. The GReAT [GRe] tool suite is used to define the semantic anchoring. In the paper [CSAJ05] abstract state machines are used for the minimal modelling language L_i, the semantics of L_i is specified as an AsmL specification.
An advantage of Semantic Anchoring is that one does not have to define the formal semantics of a DSL L, only a model transformation from that DSL to a DSL with formal semantics L_i is needed. However, this is not always possible because a DSL may have semantic properties that are very different from L_i. Another disadvantage is that during simulation one does not simulate the model that conforms to L but a model that conforms to L_i. This may complicate things and tracing between the two models must be needed in order to find bugs.

2.8 Conclusion

This chapter introduced all the basic concepts and contains references to related work. The MDE approach is explained and definitions for models and metamodels are given. Different MDE approaches are covered and compared. We also showed the use of Domain Specific Languages and their relation with MDE.
The concepts used in language definition are explained and different approaches for specifying semantics are listed. A section covered one approach in more detail: Structural Operational Semantics (SOS). SOS is compositional which makes it pragmatical; each syntactical structure of the language has one or

more rules, the rules in the *small-step* approach are generally small and the rules only refer to directly accessible elements.
This chapter also covered different approaches for defining the semantics of modelling languages. In all approaches there is a mapping from the modelling language to a semantic domain. However, this mapping is done in different ways. Most approaches specialized LTS as the semantic domain, however the transition relation in the LTS is specified in different manners.
The approaches differ in some aspects. Some approaches are have a rigorous formal basis however other approaches use a more pragmatical approach. None of the existing approaches use SOS as a framework for specifying semantics. The next chapter will explain how SOS can be used in an MDE context.

3

An SOS based Semantic Language DSL

3.1 Introduction

Semantics of models can be expressed in different ways. This chapter takes a MDE approach to specifying dynamic semantics. We will create a DSL, named *SemLang* (an abbreviation for *Semantic Language*), which can be used to describe the semantic of other DSLs. *SemLang* will be built upon existing knowledge about SOS and MSOS.
This chapter explains *SemLang* informally with the aid of examples. A formal definition will be given in chapter 4. The examples start with a definition of a DSL, sequentially the semantics of the DSL is defined using *SemLang*. Each section adds more complexity to *SemLang*. After this chapter the reader should have a complete but informal understanding of *SemLang* and the differences between the well known SOS (Plotkin style) and *SemLang*.
In this chapter we will use the term DSL for the language for which the semantics is described. The difference between the *SemLang* and the DSL for which the semantics is described is crucial. This is visualized in figure 3.1.

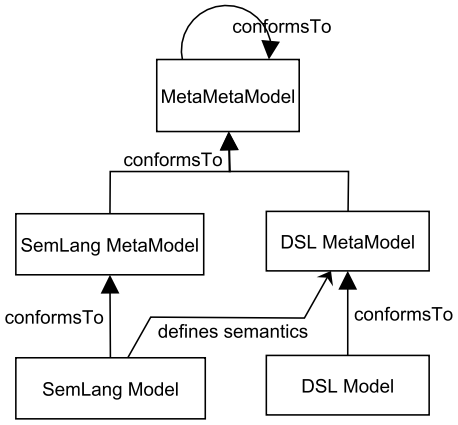


Figure 3.1: SemLang describes the semantics of a DSL

It is important that the semantics of our *SemLang* is also expressed formally, this prevents ambiguity and provides consistency, completeness and correct-

ness. This chapter, however, does not deal with formalization and is meant to be relatively easy reading. This is done to facilitate the understanding of the formalization in the next chapter.
Note that the metamodels and models defined in this chapter are built using the Eclipse Modelling Framework [EMF]. EMF is chosen because it has explicitly defined metametamodels and the tool support is well developed. For further reasons see section 2.3.3. Therefore the metamodels in this chapter conform to the ECORE metametamodel [ECO]. Because *SemLang* is a DSL itself it must also have a metamodel. The metamodel is too big to include here, but it is included in appendix A on the disc.

3.2 SOS for Expressions

In this section we will define a simple expression language by giving its metamodel. A small sample model is also given. The semantics is explained both informally as well as formally using *SemLang*. In the end of this section the sample model will be simulated by applying the semantics.

3.2.1 Metamodel

First the semantics of a simple expression language will be defined. The expression language only consists of the concepts **Exp**ression, **Number** and **Binary-Exp**ression. The metamodel is given in figure 3.2. Models of the metamodels are either numbers or expressions. Also note that the models are always trees, this is because the *lhs* and *rhs* relations of a binary expression are containment relations (as specified by metametamodel ECORE [ECO]).

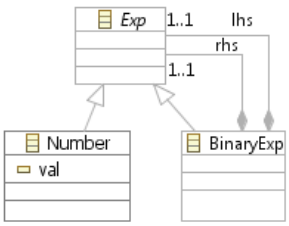


Figure 3.2: Expression language metamodel

An example model that conforms to the expression language metamodel is given in figure 3.3. The meaning of the nodes is explained in the list below:

- **Black nodes** represent objects, the label of the node is the name of the class that it conforms to.
  - The outgoing edges of an object node are either references or attributes
  - Black outgoing edges are containment references
  - Green outgoing edges are attributes
  - The name of an edges refers to the name of the reference or attribute

- **Green nodes** represent attribute values

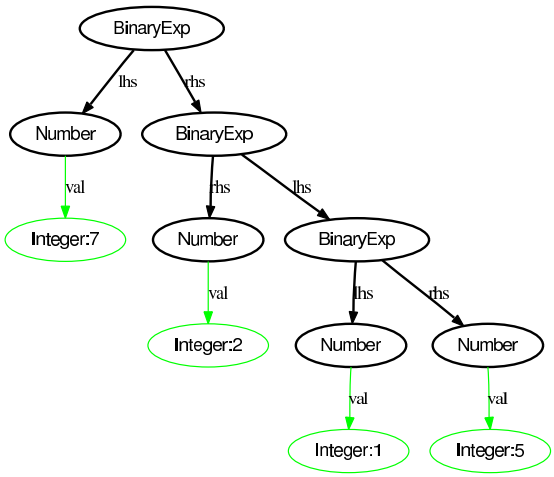


Figure 3.3: Expression language sample model

The example model is needed to show how the semantics is applied to a specific model. Before the model can be executed the semantics of the expression language must be defined.

3.2.2 Semantics

The semantics of the expression language is very simple: a binary expression evaluates to the sum of its operands. The result of a full computation is therefore the sum of all the numbers in the model. This description is of course incomplete and informal.
The semantics must be described formally using a flavour of SOS suited for DSLs. As explained in section 2.6 an SOS specification consists of a number of rules. These rules specify the transition relation between states when executing a model. A transition is only valid if a tree of rules can be constructed which explain the transition (for full details see [Mos04]).
Because of the simplicity of the expression language we can create SOS rules with minimal modification to the format of the rules. Plotkin style SOS [Plo81] uses the syntax of the language in the rules. As explained in section 2.2.1 a metamodel plays the same role in a DSL as the syntax does in a textual language. Therefore we change the format of the rules in such a way that we refer to the metamodel instead of the syntax. The rules for the expression language are given below.
The first equation is not a rule, it declares which states are end states (computed states). This is needed for every SOS specification to indicate when an end state is reached. The end state specification is rather simple: it declares that objects that conform to the **Number** class are end states.
The three rules themselves consist of *conditions*, a *pattern* and a *result* as explained in section 2.6. The *pattern* consists of a name of a class. Only objects that conform to that class are matched. The *pattern* also consists of zero or more variable bindings. These variables are bound to either attribute values or referenced objects. All the rules have a pattern like **BinaryExp** (**lhs** = L, **rhs** = R)

$$Computes = \{\mathbf{Number}\} \tag{3.1}$$

$$\frac{\mathrm{L} \rightarrow \mathrm{NL}}{\mathbf{BinaryExp}\,(\mathbf{lhs} = \mathrm{L}, \mathbf{rhs} = \mathrm{R}) \rightarrow \mathbf{BinaryExp}\,(\mathbf{lhs} = \mathrm{NL}, \mathbf{rhs} = \mathrm{R})} \tag{3.2}$$

$$\frac{\mathrm{R} \rightarrow \mathrm{NR}}{\mathbf{BinaryExp}\,(\mathbf{lhs} = L, \mathbf{rhs} = \mathrm{R}) \rightarrow \mathbf{BinaryExp}\,(\mathbf{lhs} = L, \mathbf{rhs} = \mathrm{NR})} \tag{3.3}$$

$$\mathbf{BinaryExp}\,(\mathbf{lhs} = L, \mathbf{rhs} = R) \rightarrow \mathbf{Number}\,(\mathbf{val} = (L.val + R.val)) \tag{3.4}$$

Figure 3.4: Semantic SOS rules for the expression language

which matches objects that conform to the **BinaryExp** class, the variables L and R are bound to the value of *lhs* and *rhs* respectively.
There is one subtle difference between the patterns in the rules; the variable L is italic in the second rule, and both L and R are italic in the last rule. Italic variables are called computed variables. Computed variables can only bind to a computed value, as defined by the equation $Computes = \{\mathbf{Number}\}$. If a computed variable cannot be bound then the rule does not match. This is also a subtle difference with respect to the Plotkin rule format. Plotkin uses different symbols for computed values (for example m and n for numbers). Note that non italic variables can also match computed values: italic variables match a subset of what non-italic variables match.
Rule 3.2 and 3.3 come with *conditions* like L → NL. The variable at the left side of the arrow (L) in the condition must be bound to an object in the pattern of the rule. The condition holds if there is a rule x that matches the object to which L is bound to. If there is a rule found then NL is bound to the result of the application of rule x. If such a rule cannot be found then the condition does not hold. If a condition does not hold then the rule itself does not match. The variable at the right side of the arrow (NL) can be used in the *result* of the rule.
The *result* of a rule looks similar to the pattern. However, it is more complex and has a different purpose: it constructs new objects. For example the rule 3.2 has the result $\mathbf{BinaryExp}\,(\mathbf{lhs} = \mathrm{NL}, \mathbf{rhs} = \mathrm{R})$. This constructs a new object that conforms to the **Expression** class and sets *lhs* to a **copy** of the object to which the variable NL is bound to. The **rhs** reference is bound to the **copy** of the object to which the variable L is bound to.
The copy algorithm can be kept simple in this case. This is because the meta-model of the expression language does not have cross-references; the model will always be a tree. Therefore the copy algorithm only has to deal with trees. The copy algorithm simply copies the current object and its complete tree. The last DSL for which we describe the semantics (in section 3.5) does contain cross references. The complete copy algorithm is explained there.
The result in rule 3.4 $\mathbf{Number}\,(\mathbf{val} = (L.val + R.val))$ constructs a **Number** object. The attribute *val* is assigned the sum of the *val* attribute of R and L. This shows how the result of a rule can contain simple mathematical expressions.

Plotkin could not embed these expressions directly in the rules because they could interfere with the syntax of the language for which the semantics were described. Because the *SemLang* rule format does not refer to the syntax of the DSL we can simply embed these calculations in the rules.

3.2.3 Example Simulation

The semantics of the expression language is specified: the end state is defined and the rules are given and their format is explained (informally). The sample model in figure 3.3 can now be executed. The semantics of the expression language is modelled by a *transition system*. A transition system consists of states and transitions. Execution in the sense of SOS is a path in a transition system. The path consists of states and transitions from state to state. The last state is a valid end state. Each transition is proved by a tree of rules.

$$ExecutionPath = State_1 \rightarrow State_2 \rightarrow \cdots \rightarrow State_{n-1} \rightarrow State_n = EndState$$

The transitions can be labelled, but for now they will not be labelled. The transitions are specified by the rules. However, we must clearly define the states of the transition system. The result part of every rule creates a new object that conforms to the metamodel. There are no elements created that are not defined by the metamodel. Therefore in this stage the states can be defined as models that conform to the metamodel of the DSL (the expression language).
When executing the sample model the first state is identical to the sample model itself. This state is not an end state because the root object does not conform to a **Number**. Therefore all rules that match will be applied to create the new state. Only rule 3.3 matches.
Rule 3.2 does not match because the transition condition L → NL does not hold: the L was bound to value of the *lhs*, which is a **Number** object. The transition condition did not hold because there was no rule found which matches the L **Number** object.
Rule 3.3 does match because the computed variable L was successfully bound to the *lhs* object. Variable R was bound to the *rhs* object, which conforms to a **BinaryExp**. The transition condition R → NR holds because a rule could be found which matches the R object: This is rule 3.2.
At this stage the variable L will be bound the *lhs* which is a **BinaryExp**. The variable R is bound to the *rhs* which is a **Number** (where attribute *val* equals 2). Note that rule 3.3 does not match because the *lhs* was not a computed value. The transition condition L → NL of also holds because the last rule 3.4 which transforms a binary expression to a number can be applied.
The complete proof for the transition can be given by expanding the transition conditions. The first rule that is matched is rule 3.3:

$$\frac{\mathrm{R}_1 \rightarrow \mathrm{NR}_1}{\mathbf{BinaryExp}\,(\mathbf{lhs} = L_1, \mathbf{rhs} = \mathrm{R}_1) \rightarrow \mathbf{BinaryExp}\,(\mathbf{lhs} = L_1, \mathbf{rhs} = \mathrm{NR}_1)}$$

The transition condition $\mathrm{R}_1 \rightarrow \mathrm{NR}_1$ can be expanded by applying rule 3.3:

$$\frac{\mathrm{L}_2 \rightarrow \mathrm{NL}_2}{\mathrm{R}_1 = \mathbf{BinaryExp}\,(\mathbf{lhs} = \mathrm{L}_2, \mathbf{rhs} = \mathrm{R}_2) \rightarrow \mathrm{NR}_1 = \mathbf{BinaryExp}\,(\mathbf{lhs} = \mathrm{NL}_2, \mathbf{rhs} = \mathrm{R}_2)}{\mathbf{BinaryExp}\,(\mathbf{lhs} = L_1, \mathbf{rhs} = \mathrm{R}_1) \rightarrow \mathbf{BinaryExp}\,(\mathbf{lhs} = L_1, \mathbf{rhs} = \mathrm{NR}_1)}$$

The transition condition $\mathrm{L}_2 \rightarrow \mathrm{NL}_2$ can be expanded by applying rule 3.4:

$$\frac{\mathrm{L}_2 = \mathbf{BinaryExp}\,(\mathbf{lhs} = L_3, \mathbf{rhs} = R_3) \rightarrow \mathrm{NL}_2 = \mathbf{Number}\,(\mathbf{val} = (L_3.val + R_3.val))}{\mathrm{R}_1 = \mathbf{BinaryExp}\,(\mathbf{lhs} = \mathrm{L}_2, \mathbf{rhs} = \mathrm{R}_2) \rightarrow \mathrm{NR}_1 = \mathbf{BinaryExp}\,(\mathbf{lhs} = \mathrm{NL}_2, \mathbf{rhs} = \mathrm{R}_2)}{\mathbf{BinaryExp}\,(\mathbf{lhs} = L_1, \mathbf{rhs} = \mathrm{R}_1) \rightarrow \mathbf{BinaryExp}\,(\mathbf{lhs} = L_1, \mathbf{rhs} = \mathrm{NR}_1)}$$

This gives a complete proof for a transition condition to a new state. The proof can now be used to construct the new state. This is done by substituting variables with the constructed objects in each rule:

$$\frac{\mathrm{L}_2 = \mathbf{BinaryExp}\,(\mathbf{lhs} = L_3, \mathbf{rhs} = R_3) \rightarrow \mathrm{NL}_2 = \mathbf{Number}\,(\mathbf{val} = (L_3.val + R_3.val))}{\mathrm{R}_1 = \mathbf{BinaryExp}\,(\mathbf{lhs} = \mathrm{L}_2, \mathbf{rhs} = \mathrm{R}_2) \rightarrow \mathrm{NR}_1 = \mathbf{BinaryExp}\,(\mathbf{lhs} = \mathrm{NL}_2, \mathbf{rhs} = \mathrm{R}_2)}{\mathbf{BinaryExp}\,(\mathbf{lhs} = L_1, \mathbf{rhs} = \mathrm{R}_1) \rightarrow \mathbf{BinaryExp}\,(\mathbf{lhs} = L_1, \mathbf{rhs} = \mathrm{NR}_1)}$$

$$\frac{\mathrm{NL}_2 = \mathbf{Number}\,(\mathbf{val} = (1 + 5))}{\mathrm{R}_1 = \mathbf{BinaryExp}\,(\mathbf{lhs} = \mathrm{L}_2, \mathbf{rhs} = \mathrm{R}_2) \rightarrow \mathrm{NR}_1 = \mathbf{BinaryExp}\,(\mathbf{lhs} = \mathrm{NL}_2, \mathbf{rhs} = \mathrm{R}_2)}{\mathbf{BinaryExp}\,(\mathbf{lhs} = L_1, \mathbf{rhs} = \mathrm{R}_1) \rightarrow \mathbf{BinaryExp}\,(\mathbf{lhs} = L_1, \mathbf{rhs} = \mathrm{NR}_1)}$$

$$\frac{\mathrm{NR}_1 = \mathbf{BinaryExp}\,(\mathbf{lhs} = \mathbf{Number}\,(\mathbf{val} = 6), \mathbf{rhs} = \mathrm{R}_2)}{\mathbf{BinaryExp}\,(\mathbf{lhs} = L_1, \mathbf{rhs} = \mathrm{R}_1) \rightarrow \mathbf{BinaryExp}\,(\mathbf{lhs} = L_1, \mathbf{rhs} = \mathrm{NR}_1)}$$

Therefore the new state can be constructed using the following constructor. Note that the free variables in this constructor are bounded to objects.

$$\mathbf{BinaryExp}\,(\mathbf{lhs} = L_1, \mathbf{rhs} = \mathbf{BinaryExp}\,(\mathbf{lhs} = \mathbf{Number}\,(\mathbf{val} = 6), \mathbf{rhs} = \mathrm{R}_2))$$

For each state we can find all possible transition by finding proofs for the transitions. If the DSL is deterministic the only one proof can be found. The new state can be constructed by using the proof for the transition.

A complete execution path is given in figure 3.5. The proof given is the proof from the first state (most left in the figure) to the second state. The execution path consists of four states and three transitions. The proof for the other two transitions is given as an exercise to the reader.

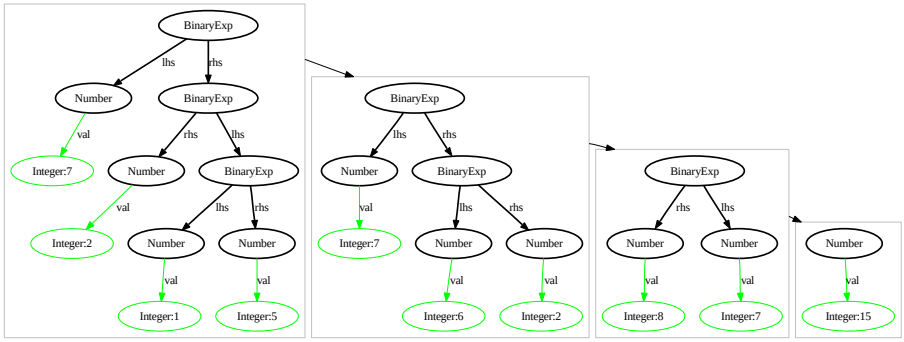


Figure 3.5: Execution states after applying the rules to the example model

3.3 Introducing the Store

This section introduces a simple imperative language. First the language will be explained by defining the metamodel and giving a small sample model. Then the semantics will be defined using *SemLang*. This requires that *SemLang* must be extended to deal with a store. In the end the semantics will be applied to the sample model.

3.3.1 MetaModel

The imperative language consists of binary expressions as explained in the previous section. However, a binary expression also has an operator. In this language we only have a **Plus** and a *Minus* operator.

The simple imperative language given in this section also allows occurrences of variables; a value can be assigned to them and their value can be retrieved. Therefore a new expression is added: the **Var** expression. **Var** evaluates to the value of the variable that it is referring to.

The main difference is that the imperative language given here allows commands to be executed sequentially. A command is either a **Seq** command or an **Assign** command. A complete metamodel is given in figure 3.6. The given language is almost identical to the language given by Plotkin in chapter two of his paper [Plo81]. The main difference is again that our language is a DSL defined as a metamodel.

Notice we do not use collections (reference multiplicities greater than one) in our metamodel. This is done to keep the metamodel close to the language as defined by Plotkin. The rules must allow list-matching and list-construction when using lists in the metamodel. This is introduced in the last section in this chapter.

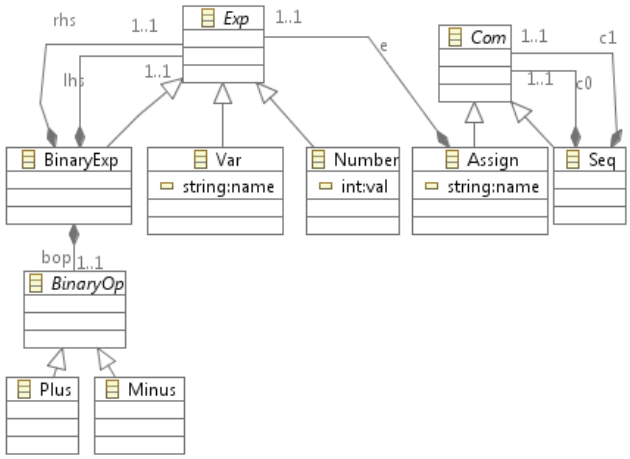


Figure 3.6: Simple imperative language metamodel

Representing an example model of this language as a graph tends to become large. Therefore the example model will be given in textual form to accommodate easier understanding. However, this requires that we give a syntax definition for the metamodel. This is done by using the Textual Concrete Syntax (TCS) technology as defined by Jouault, Bézivin and Kurtev [JBK06].

Listing 3.1: TCS for the simple imperative DSL

```
1  template Exp main abstract;
2  template Number: val;
3  template Var: name;
4  template BinaryOp abstract;
5  template Plus: "+";
6  template Minus: "-";
7  template BinaryExp: "(" lhs bop rhs ")";
8  template Com main abstract;
9  template Assign: name ":=" e;
10 template Seq: "(" c0 ";" c1 ")" ;
```

This TCS specification can be used to represent a model in textual form. An example model is given below.

Listing 3.2: Textual sample model

```
1 ( a:=10;
2         ( b:=(10+a);
3           a:=(b-3)
4         )
5 )
```

To be complete this model is also given as a graph in figure 3.7. This model is retrieved by parsing the sample code using the TCS syntax definition. For full details we refer to the TCS paper [JBK06].

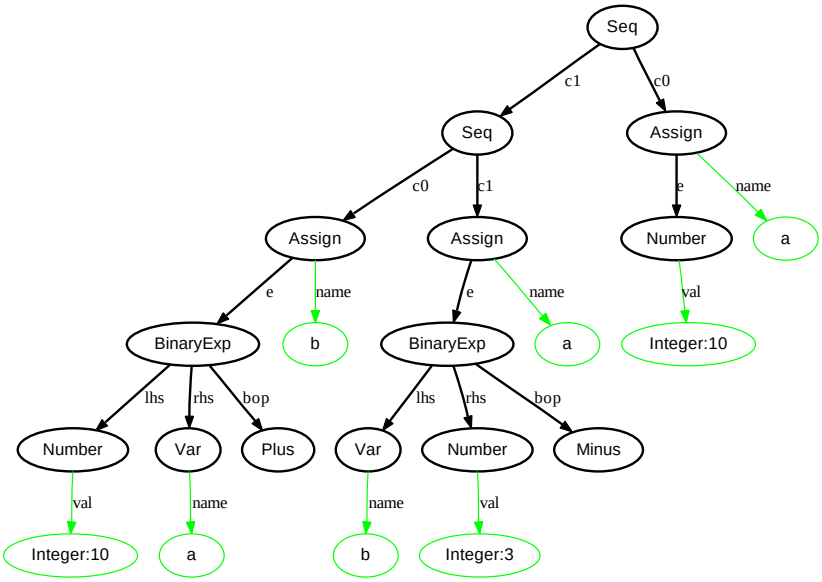


Figure 3.7: Sample model as a graph

3.3.2 Semantics

The semantics of the simple imperative language is first explained informally. Expressions are evaluated analogous to the semantics of the expression language (as explained in the previous section). The only difference is that the calculation that must be performed is specified by the operator, either **Min** for subtraction or **Plus** for addition. The **Var** expression is evaluated by getting the value of the referring variable from the store (internal memory).

A **Seq** is executed by first executing *c0* and then executing *c1*. An **Assign**ment is executed by first executing the expression and then by updating the store by binding the calculated value to the name of the variable. Executing a command does not yield a result.

The informal description of the imperative language includes the need of memory in which variables can be created, read, updated (and deleted). Therefore we add the concept of store, which is an abstraction of memory, to our *SemLang* just like Plotkin did in chapter 2 of his paper [Plo81]. In the approach of Plotkin a store is a function from names to values. To keep our approach flexible we define a store as a function from (ECORE) objects to other objects: $Store : Object \rightarrow Object$.

Each object has its own identity. When an object gets created it is assigned an identity. If it is copied then the copy gets a new identity. An object equals another object if it has the same identity. However, primitive objects like strings and integers have a different identity; they have the same identity if their values are equal.

Plotkin added the store to the states of the transition system that is specified by a set of rules. A more modular approach MSOS (Modular SOS) was introduced by Mosses [Mos99, Mos02, Mos04]. In MSOS the store operations are specified in the labels. *SemLang* will adopt MSOS: the operations will be specified in

the labels. However, internally the store will be added to a state. This is more readable and avoids using the use of category theory that MSOS requires.

The *SemLang* rules for the imperative language are explained step by step. First some simple rules are explained and then more complex rules that deal with the store are explained. The new constructs that are added to *SemLang* will also be explained.

The rules for **BinaryExp** are almost identical to the rules for the expression language. These are presented in figure 3.8. An additional construct in the rules is a type-check condition. For example the condition $\mathrm{O} \in \mathbf{Plus}$ holds if the object bounded to O conforms to the **Plus** class in the metamodel.

$$Computes = \{\mathbf{Nil}, \mathbf{Number}\}$$

$$\frac{\mathrm{L} \rightarrow \mathrm{NL}}{\mathbf{BinExp}\,(\mathbf{lhs} = \mathrm{L}, \mathbf{bop} = \mathrm{O}, \mathbf{rhs} = \mathrm{R}) \rightarrow \mathbf{BinExp}\,(\mathbf{lhs} = \mathrm{NL}, \mathbf{bop} = \mathrm{O}, \mathbf{rhs} = \mathrm{R})} \tag{3.5}$$

$$\frac{\mathrm{R} \rightarrow \mathrm{NR}}{\mathbf{BinExp}\,(\mathbf{lhs} = L, \mathbf{bop} = \mathrm{O}, \mathbf{rhs} = \mathrm{R}) \rightarrow \mathbf{BinExp}\,(\mathbf{lhs} = L, \mathbf{bop} = \mathrm{O}, \mathbf{rhs} = \mathrm{NR})} \tag{3.6}$$

$$\frac{\mathrm{O} \in \mathbf{Plus}}{\mathbf{BinExp}\,(\mathbf{lhs} = L, \mathbf{bop} = \mathrm{O}, \mathbf{rhs} = R) \rightarrow \mathbf{Number}\,(\mathbf{val} = (L.val + R.val))} \tag{3.7}$$

$$\frac{\mathrm{O} \in \mathbf{Minus}}{\mathbf{BinExp}\,(\mathbf{lhs} = L, \mathbf{bop} = \mathrm{O}, \mathbf{rhs} = R) \rightarrow \mathbf{Number}\,(\mathbf{val} = (L.val - R.val))} \tag{3.8}$$

Figure 3.8: SOS rules for binary expressions

Notice that the set of computed values $\{\mathbf{Nil}, \mathbf{Number}\}$ includes **Nil**. **Nil** refers to a pseudo-object that represents no value, or no object. $Computes = \{\mathbf{Nil}, \mathbf{Number}\}$ defines therefore that the **Nil** pseudo-object or a **Number** object is a valid end state.

The introduction of **Nil** means that the states are not pure models that conform to the DSL metamodel. A state now must conform to a different metamodel that is an extension of the DSL metamodel. The only extension is that any reference may also point to a **Nil** object.

Next the rules that deal with the **Var** expression and the commands (**Seq** and **Assign**) are introduced. Two rules deal with the evaluation of a **Seq** command. Rule 3.9 makes sure that first the object assigned to *c0* is evaluated. Rule 3.10 states that a **Seq** command evaluates to the object assigned to *c1* if the object assigned to *c0* is a computed value. Together they define the order in which commands are executed.

$$\frac{\mathrm{C0} \rightarrow \mathrm{NC}}{\mathbf{Seq}\,(\mathbf{c0} = \mathrm{C0}, \mathbf{c1} = \mathrm{C1}) \rightarrow \mathbf{Seq}\,(\mathbf{c0} = \mathrm{NC}, \mathbf{c1} = \mathrm{C1})} \tag{3.9}$$

$$\mathbf{Seq}\,(\mathbf{c0} = \mathit{C0}, \mathbf{c1} = \mathrm{C1}) \rightarrow \mathrm{C1} \tag{3.10}$$

Figure 3.9: Rules for the **Seq** command

In order to evaluate both a **Var** and an **Assign**ment we need a store. The store will represented using a function:

$$\mathit{Store} : \mathit{Object} \rightarrow \mathit{Object}$$

A function consists of zero or more tuples, a tuple with a key k and a value v is written as $k \mapsto v$. A concrete function with n tuples can be written as:

$$\{k_1 \mapsto v_1, k_2 \mapsto v_2, \ldots, k_{n-1} \mapsto v_{n-1}, k_n \mapsto v_n\}$$

There are some operations that can be performed on a function. Suppose we have a function named a, then $a(k)$ result in the value that is bound to k, or **Nil** if the key is not found. More formally:
Suppose we have another function named b, then the expression $a\,[b]$ results in a new function with tuples $a \cup b$, however if there is a key clash then the tuple of b is used. This operation is called overriding. More formally if $c = a\,[b]$ then
SemLang is extended by allowing the definition of functions (which can be used as a store). Each function has a name, domain and range. A function is defined by as follows:

$$\mathit{functionName} : \mathbf{DomainClass} \rightarrow \mathbf{RangeClass}$$

where *functionName* indicates the name of the function. **DomainClass** and **RangeClass** both refer to class names in the DSL metamodel. Only objects that conform the **DomainClass** can be used as keys, objects that conform to the **RangeClass** can be used as values.
The DSL only needs a function that maps variables to number, therefore equation 3.11 defines a function named *store* that maps variable names to numbers. The function definition and the final rules are presented in figure 3.10.
Note that most programming languages do not bind variables to values directly. The notion of *location* is used to decouple the variables from the values. A location can be seen as a *memory address*. A variable binds to a location, and a location binds to a value. This can be achieved by defining two functions. First a function that binds variables to locations: *store* : **String** → **Integer**. And a function that maps locations to values: *memory* : **Integer** → **Number**.
Rule 3.12 evaluates a **Var** expression with the name bound to N to the value *store* (N). This is value v if $N \mapsto v \in \mathit{store}$. Informally: a **Var** evaluates to its current value in the *store* function.
Two rules deal with the evaluation of an assignment. Rule 3.13 makes sure that the expression bound to **e** is first evaluated to a computed value. Rule 3.14 is more complicated. It evaluates to **Nil** but it updates the *store* function.

$$store : \mathbf{String} \rightarrow \mathbf{Number} \tag{3.11}$$

$$\mathbf{Var}\,(\mathbf{name} = \mathrm{N}) \rightarrow store\,(\mathrm{N}) \tag{3.12}$$

$$\frac{\mathrm{E} \rightarrow \mathrm{NE}}{\mathbf{Assign}\,(\mathbf{name} = \mathrm{N}, \mathbf{e} = \mathrm{E}) \rightarrow \mathbf{Assign}\,(\mathbf{name} = \mathrm{N}, \mathbf{e} = \mathrm{NE})} \tag{3.13}$$

$$\mathbf{Assign}\,(\mathbf{name} = \mathrm{N}, \mathbf{e} = E) \xrightarrow{store' = store[\{\mathrm{N} \mapsto E\}]} \mathbf{Nil} \tag{3.14}$$

Figure 3.10: Store definition and rules for **Assign** and **Var**

Rule 3.14 introduces some new notation. The new notation is a label on the arrow. The label in rule 3.14 is:

$$store' = store\,[\{\mathrm{N} \mapsto E\}] \tag{3.15}$$

A label is used to update a function. The format of the label is:

$$functionName' = functionExpression$$

It means that the function named *functionName* is set to the value of *functionExpression* in the next state. A *functionExpression* is an expression that results in a new function. This expression can be a new function by using the format $\{k_1 \mapsto v_1, \ldots, k_n \mapsto v_n\}$. The expression can also be a reference to a function (by using its name). Another possibility is the usage of the override expression. Label 3.15 updates the function named *store* to the expression $store\,[\{\mathrm{N} \mapsto E\}]$ in the next state. The expression indicates that the current value of *store* must be overridden with $\{\mathrm{N} \mapsto E\}$. This value is a new function with one tuple $\mathrm{N} \mapsto E$. Both variables N and E were bound to an object in the *pattern* part of the rule: $\mathbf{Assign}\,(\mathbf{name} = \mathrm{N}, \mathbf{e} = E)$. The total result of the expression is a new function where $\mathrm{N} \mapsto E$ is added to the existing *store* function (if a tuple with key N already existed it is overridden). This new function is then bounded to the function named *store* in the new state.

3.3.3 Example Simulation

In this section the simulation of the sample model will be explained. Most rules that describe the semantics of the simple imperative language do not introduce new concepts compared to the rules in the simple expression language. Therefore the explanation of the simulation will be limited to the new constructs that use functions.

Some aspects of the first transition, from the first state to the second state, will be highlighted. This is done in order to explain the effects of having a function during simulation. The first transition is visualised in figure 3.11.

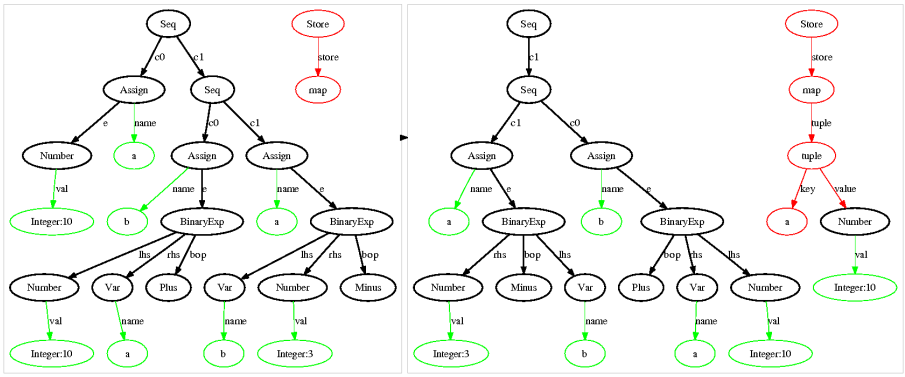


Figure 3.11: First transition, from first state (left) to second state (right) of the sample model

The first state is almost identical to the example model itself. However, two new red nodes named *Store* and *map* are added. The node *Store* simply represents a set of functions. The value of a function is represented as a *map* node which has zero or more tuples. The label of an edge from *Store* to a *map* indicated the name of the function. The *map* does not have any tuples in the first state. However, in the second state the map does have a tuple as indicated by a *tuple* node. A tuple has a key and a value. In the second state this key is the string *a* and the value is an object that conforms to **Number**. This tuple is the result of the application of assignment rule 3.14.

The transition from the first state to the second state can be proven by the following rules: the root node matches the first **Seq** rule (rule 3.9) and the transition condition is satisfied by the **Assign**ment rule (rule 3.14). The assignment rule evaluates to **Nil**, therefore we do not see a *c0* reference from the root node **Seq** (this is just a visual representation choice, another visual representation was to have an edge labelled *c0* from to root node to a **Nil** node).

The label of the assignment rule (3.14) indicates that the new *store* function must have the value $store\,[\{\mathrm{N} \mapsto E\}]$. N is bound to the string object "a" and E is bound to the **Number** object with attribute *val* equal to 10. Therefore the map of function *store* will be overridden with the new tuple $\mathrm{N} \mapsto E$. This results in the map in the second state.

The transition from the second-last state to the last state is visualised in figure 3.12. The explanation for this transition is analogous to the first transition. However, there are a few things to notice.

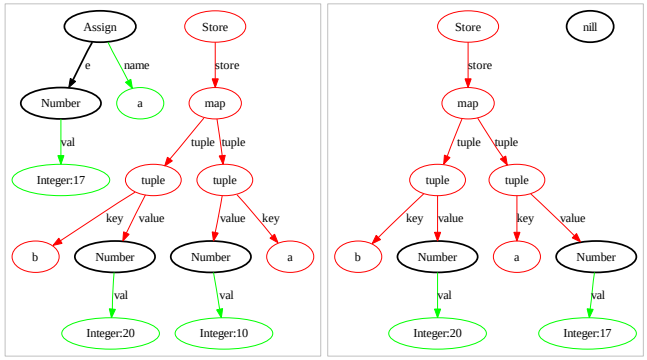


Figure 3.12: Last transition from state (left) to end state (right) of the sample model

First the value of variable *a* in the store is overridden from number 10 to number 17. Secondly notice that the end state is a **Nil** object and the *store*. So the result of a simulation of an imperative language is (often) only a store.
The full proof of the last transition is left as an exercise to the reader.

3.4 Introducing the Environment

Functional languages do not have a state. However, functional languages (and also imperative languages) do have a concept of scope. Imperative languages often have a scope in which variables are bound to locations, and a store (or memory) in which locations are bound to values. This decoupling of variables and values by the usage of locations was already introduced briefly in the previous chapter.
We will use *environment* as a more general term for scopes. Constructs in an environment aware language are executed within a certain environment. The difference between the concept of *store* is that an *environment* has a certain context. Both environments and stores can be represented using functions as explained in the previous section.
This section introduces a functional language that requires the need of an environment. The language is almost identical to the language defined by Plotkin in chapter 3 of his article [Plo81]. Again we will take an MDE approach.

3.4.1 MetaModel

The metamodel (presented in figure 3.13) of the functional language is an extension to the simple expression language. The most important class is the abstract class **Exp**ression. Like the simple expression language there are primitive values, **Number** for integers and **Bool** for booleans. The **Var** class has the same purpose as the **Var** class in the imperative language, i.e. it references variables. There is also a binary expression **BinaryExp** which has an operator **BinaryOperator**. The figure does not include the subclasses of the **BinaryOperation**; the concrete operators like **Plus**, **Minus**, **Multiply**, **And**, **Or**, **Eq**uals and **Neq** (Not equals). These are not included in the figure to keep the figure small.

Two new expressions are included in the functional language. First the **If-Exp**ression which is a simple if-then-else expression. The most important new expression is the **LetExp**ression. The let expression defines an environment using a **Def**inition **d** and executes the expression **e** within that environment.

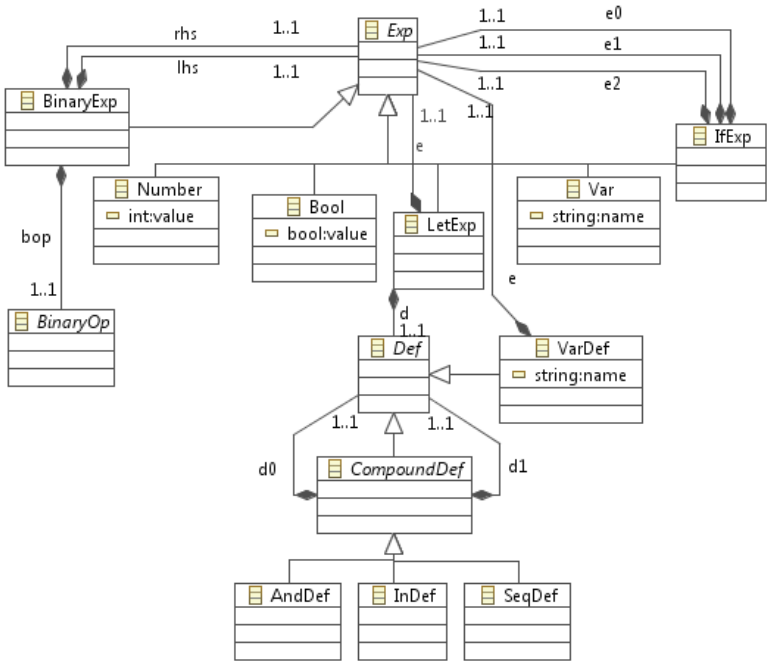


Figure 3.13: Functional language metamodel

A definition is either a **VarDef**inition or a so called **CompoundDef**inition. A variable definition defines a variable. A compound definition is a combination of two definitions. There are three types of compound definitions. Each with different semantics. The semantics will be explained after this subsection.
Again a textual concrete syntax is represent in figure 3.3 to accommodate easier understanding. The syntax of the **Number**, **Var**, **BinaryExp**, **Min** and **Plus** will not be given, they are similar to the syntax in the simple imperative language.

Listing 3.3: TCS for the functional language

```
1  template Bool: (value ? "true" : "false");
2  template Multiply: "*";
3  template Eq: "=";
4  template And: "and";
5  template Or: "or";
6  template Neq: "<>";
7  template LetExp: "let" d "in" e;
8  template IfExp: "if" e0 "then" e1 "else" e2;
9  template Def abstract;
10 template VarDef: name "=" e;
```

```
11 template CompoundDef abstract;
12 template SeqDef: "(" d0 ";" d1 ")";
13 template AndDef: "(" d0 "and" d1 ")";
14 template InDef: "(" d0 "in" d1 ")";
```

This syntax definition can now be used to define a sample model in textual form. The sample model is given in listing 3.4. The texts in green are comments. It shows how the definitions are evaluated. The next subsection will explain the semantics of the functional language in depth.

Listing 3.4: Textual sample model

```
1  let
2  ( (x=4;y=8);                          -- env = {x=4, y=8}
3          (
4                  (x=6 in y=2*x)        -- env = {y=12}
5                  and
6                  x=y                   -- env = {x=8}
7          )
8  )                                     -- env = {x=8, y=12}
9  in
10   if (y<>x) then (y+x) else x         -- 12+8 because 12<>8
```

Again to be complete and to avoid confusion the model is also given as a graph in figure 3.14. The comments in listing 3.4 are not part of the model.

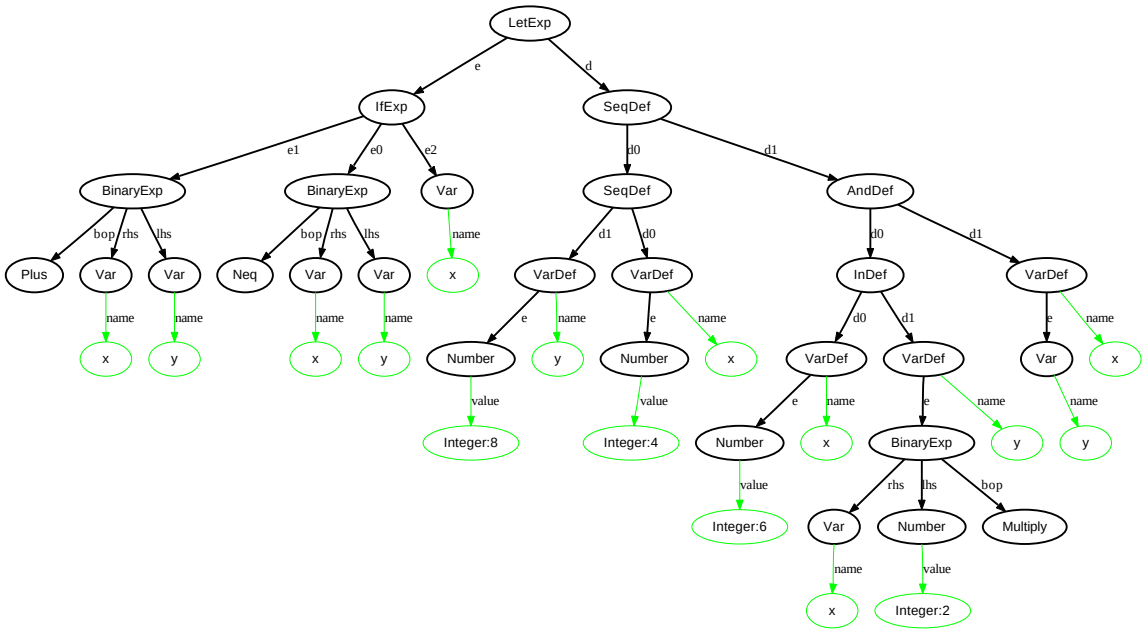


Figure 3.14: Sample model as a graph

3.4.2 Semantics

Like the previous section only the semantics of new elements will be explained. These elements are the **IfExp**ression, the **LetExp**ression and all subclasses of a **Def**inition. The semantics of the **Var** class is different but similar to the semantics of the class in the simple imperative language.
The semantics of the **IfExp**ression is straightforward. If **e0** evaluates to true then the **IfExp**ression evaluates to **e1**. The **IfExp**ression evaluates to **e2** otherwise. The *SemLang* rules are given below.
The first rule shows that first expression **e0** must be evaluated. **e0** acts as the condition for the **IfExp**ression.

$$\frac{\mathrm{E0} \rightarrow \mathrm{NE}}{\mathbf{IfExp}\,(\mathbf{e0} = \mathrm{E0}, \mathbf{e1} = \mathrm{E1}, \mathbf{e2} = \mathrm{E2}) \rightarrow \mathbf{IfExp}\,(\mathbf{e0} = \mathrm{NE}, \mathbf{e1} = \mathrm{E1}, \mathbf{e2} = \mathrm{E2})} \tag{3.16}$$

The last two rules define which one of the expression **e1** or **e2** must be evaluated if **e0** is fully evaluated to a computed value *E0*. The computed value will always be a **Number**. Therefore *E0.value* refers to the attribute **value** of that **Number**.
If that value is true rule 3.17 will match. This will mean that the evaluation will result in expression bound to E1. Otherwise, if the value is false (as indicated by the condition $\neg$*E0.value*) rule 3.18 will fire. This will evaluate the **IfExp**ression to the expression bound to E2.

$$\frac{\mathit{E0.value}}{\mathbf{IfExp}\,(\mathbf{e0} = \mathit{E0}, \mathbf{e1} = \mathrm{E1}, \mathbf{e2} = \mathrm{E2}) \rightarrow \mathrm{E1}} \tag{3.17}$$

$$\frac{\neg\mathit{E0.value}}{\mathbf{IfExp}\,(\mathbf{e0} = \mathit{E0}, \mathbf{e1} = \mathrm{E1}, \mathbf{e2} = \mathrm{E2}) \rightarrow \mathrm{E2}} \tag{3.18}$$

In order to explain the rules for the **LetExp**ression we must first explain the rules for the **Def**initions. Definitions are evaluated differently from all previous structures. They do not evaluate to another object or to a **Nil**; definitions evaluate to an environment. This makes sense because a definition defines an evaluation context. An environment has the same structure as a store: they are both functions.
The evaluation of the definition in the sample model can be explained informally. Each definition imports an environment. Then the definition is evaluated within that environment. Finally the definition exports an environment. The different definitions are evaluated (informally) as follows:

- A **VarDef**inition first evaluates the expression e and then evaluates to an environment with one tuple.
- A **SeqDef**inition evaluates its sub-definitions in sequence. The second definition *d1* imports the environment that was exported by the first definition *d0*. The **SeqDef**inition exports both environments that were exported by the sub-definitions *d0* and *d1*.
- The **AndDef**inition evaluates its sub-definitions in parallel. The environments of both sub-definitions are exported.

- Finally, the **InDef**inition evaluates its sub-definitions in sequence. However, only the environment of the second sub-definition *d1* is exported. Therefore the environment exported by *d0* is private (only visible to *d1*).

The complete evaluation of the definition in the sample model is visualized in figure 3.15. It shows how definitions export (the lines that end with an arrow) and import (the lines that begin with a diamond) environments. For example the first **SeqDef**inition ($x = 4; y = 8$) is evaluated to the environment $\{x \mapsto 4, y \mapsto 8\}$. It also shows how the **InDef**inition does not export the environment of its first sub-expression $x = 6$.

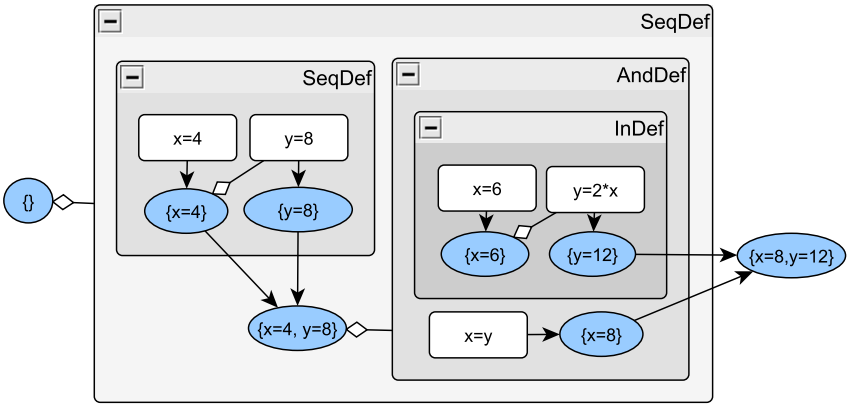


Figure 3.15: Evaluation of the definitions. Rounded boxes are definitions. Blue ellipses are environments.

The informal explanation of the **Def**initions is used to define the semantics formally using *SemLang*. First the rules for the evaluation of the **VarDef**inition are given.

$$\frac{\mathrm{E} \rightarrow \mathrm{NE}}{\mathbf{VarDef}\,(\mathbf{name} = \mathrm{N}, \mathbf{e} = \mathrm{E}) \rightarrow \mathbf{VarDef}\,(\mathbf{name} = \mathrm{N}, \mathbf{e} = \mathrm{NE})} \tag{3.19}$$

$$\mathbf{VarDef}\,(\mathbf{name} = \mathrm{N}, \mathbf{e} = E) \rightarrow \{\mathrm{N} \mapsto E\} \tag{3.20}$$

Rule 3.19 does not include any new structure and its meaning should be clear. Rule 3.20 on the other hand shows how a **VarDef** is actually evaluated to a function with one tuple $\{\mathrm{N} \mapsto E\}$.
Rule 3.20 is only valid if *SemLang* is extended to support functions as objects in the states. The extension is rather simple: a reference of an object is allowed to point to a function. The definition of a function was explained in the previous section.
The rules for the compound definitions are more complex. They require the definition of an environment function, much like a store was defined for the simple imperative language. The environment required for the functional language will be named env_ρ. The definition of an environment function is similar to the definition of a store function:

$$env_\rho : String \rightarrow Object$$

The only difference is that the name of the function is labelled with a ρ symbol. This symbol indicates that the function should only be used as an environment and not as a store. The equation defines an environment function named env_ρ that maps strings to objects. Environment functions differ from store function in that they can only be used in the labels of transition conditions, the next rules illustrate this fact.
The usage of environment functions becomes clearer when the rules for the compound definitions are explained. The rules for the **SeqDef**inition are given below. Rule 3.21 is simple: it states that first **d0** should be evaluated. Note that **d0** will evaluate to an environment function.

$$\frac{\mathrm{D0} \rightarrow \mathrm{ND0}}{\mathbf{SeqDef}\,(\mathbf{d0} = \mathrm{D0}, \mathbf{d1} = \mathrm{D1}) \rightarrow \mathbf{SeqDef}\,(\mathbf{d0} = \mathrm{ND0}, \mathbf{d1} = \mathrm{D1})} \tag{3.21}$$

Rule 3.22 is more complex. The pattern part $\mathbf{SeqDef}\,(\mathbf{d0} = d0_\rho, \mathbf{d1} = \mathrm{D1})$ matches a **SeqDef** object where reference **d0** points to a function. This is because variable $d0_\rho$ only binds to environment functions, not to other objects. Variable D1 will bind to a non computed **Def**inition object.
The transition condition also introduces new constructs: $\mathrm{D1} \xrightarrow{env_\rho = env_\rho[d0_\rho]} \mathrm{ND1}$. It means that D1 must be evaluated under the environment $env_\rho\,[d0_\rho]$. In other words: D1 will be evaluated in an environment that imports the definitions as defined by **d0**.

$$\frac{\mathrm{D1} \xrightarrow{env = env_\rho[d0_\rho]} \mathrm{ND1}}{\mathbf{SeqDef}\,(\mathbf{d0} = d0_\rho, \mathbf{d1} = \mathrm{D1}) \rightarrow \mathbf{SeqDef}\,(\mathbf{d0} = d0_\rho, \mathbf{d1} = \mathrm{ND1})} \tag{3.22}$$

The last rule (rule 3.23) shows how a **SeqDef** is eventually evaluated to an environment function. It also shows that the rhs of a rule may be a function expression. In this case it is a function override expression $d0_\rho\,[d1_\rho]$. Thus the final result of a **SeqDef**inition is the environment defined by **d0** overridden by the environment defined by **d1**.

$$\mathbf{SeqDef}\,(\mathbf{d0} = d0_\rho, \mathbf{d1} = d1_\rho) \rightarrow d0_\rho\,[d1_\rho] \tag{3.23}$$

The rules for the **AndDef**inition are almost identical to the rules of the **SeqDef**inition. Rule 3.24 and 3.26 are actually completely similar to rules 3.21 and 3.23 respectively.

$$\frac{\mathrm{D0} \rightarrow \mathrm{ND0}}{\mathbf{AndDef}\,(\mathbf{d0} = \mathrm{D0}, \mathbf{d1} = \mathrm{D1}) \rightarrow \mathbf{AndDef}\,(\mathbf{d0} = \mathrm{ND0}, \mathbf{d1} = \mathrm{D1})} \tag{3.24}$$

$$\frac{\mathrm{D1} \rightarrow \mathrm{ND1}}{\mathbf{AndDef}\,(\mathbf{d0} = d0_\rho, \mathbf{d1} = \mathrm{D1}) \rightarrow \mathbf{AndDef}\,(\mathbf{d0} = d0_\rho, \mathbf{d1} = \mathrm{ND1})} \tag{3.25}$$

The only difference is in rule 3.25. The sub-definitions in an **AndDef**inition are evaluated in the same environment; definition **d1** is not aware of the environment function $d0_\rho$. This characterizes the parallel behaviour of the **AndDef**inition.

$$\mathbf{AndDef}\,(\mathbf{d0} = d0_\rho, \mathbf{d1} = d1_\rho) \rightarrow d0_\rho\,[d1_\rho] \tag{3.26}$$

The rules for the **InDef**inition are similar to the rules of the **SeqDef**inition. The only difference is that the environment of the first sub-definition $d0_\rho$ is not exported. This is specified by rule 3.29.

$$\frac{\mathrm{D0} \rightarrow \mathrm{ND0}}{\mathbf{InDef}\,(\mathbf{d0} = \mathrm{D0}, \mathbf{d1} = \mathrm{D1}) \rightarrow \mathbf{InDef}\,(\mathbf{d0} = \mathrm{ND0}, \mathbf{d1} = \mathrm{D1})} \tag{3.27}$$

$$\frac{\mathrm{D1} \xrightarrow{env = env_\rho[d0_\rho]} \mathrm{ND1}}{\mathbf{InDef}\,(\mathbf{d0} = d0_\rho, \mathbf{d1} = \mathrm{D1}) \rightarrow \mathbf{InDef}\,(\mathbf{d0} = d0_\rho, \mathbf{d1} = \mathrm{ND1})} \tag{3.28}$$

$$\mathbf{InDef}\,(\mathbf{d0} = d0_\rho, \mathbf{d1} = d1_\rho) \rightarrow d1_\rho \tag{3.29}$$

The rules for all types of definitions are explained at this point. Definitions are only allowed in **LetExp**ressions. The rules for **LetExp**ressions are not very difficult.
A **LetExp**ression first evaluates its definitions to an environment. This is defined by rule 3.30

$$\frac{\mathrm{D} \rightarrow \mathrm{ND}}{\mathbf{LetExp}\,(\mathbf{d} = \mathrm{D}, \mathbf{e} = \mathrm{E}) \rightarrow \mathbf{LetExp}\,(\mathbf{d} = \mathrm{ND}, \mathbf{e} = \mathrm{E})} \tag{3.30}$$

If the definitions are fully evaluated to an environment p_ρ (represented as a function) then the expression **e** is evaluated within the environment $env_\rho\,[p_\rho]$. This is the environment in which the **LetExp**ression is evaluated env_ρ overridden by the environment p_ρ defined by the definition in the **LetExp**ression. This is shown in rule 3.31
This basically means that the variables defined by the definition **d** are only visible within expression **e** of the **LetExp**ression. This is also known as the scope of the definition.

$$\frac{\mathrm{E} \xrightarrow{env_\rho = env_\rho[p_\rho]} \mathrm{NE}}{\mathbf{LetExp}\,(\mathbf{d} = p_\rho, \mathbf{e} = \mathrm{E}) \rightarrow \mathbf{LetExp}\,(\mathbf{d} = p_\rho, \mathbf{e} = \mathrm{NE})} \tag{3.31}$$

Finally when the expression **e** is evaluated to a computed value E (a **Number**) then the complete **LetExp**ression is evaluated to that value. The environment p_ρ is dropped. So the scope of p_ρ has ended. This behaviour is defined by rule 3.32.

$$\mathbf{LetExp}\,(\mathbf{d} = p_\rho, \mathbf{e} = E) \rightarrow E \tag{3.32}$$

The rules show how expressions are evaluated in a certain environment. However, the only expression that actually needs the environment is the **Var** expression. A **Var** expression has an attribute **name** that refers to a name of a variable. The result of an evaluation of a **Var** object is the value of the variable with name **name** in the current environment.
Rule 3.33 shows how the **Var** object is evaluated $env_\rho\,(\mathrm{N})$, which is equals to the value V if $\mathrm{N} \mapsto \mathrm{V} \in env_\rho$.

$$\mathbf{Var}\,(\mathbf{name} = \mathrm{N}) \rightarrow env_\rho\,(\mathrm{N}) \tag{3.33}$$

3.4.3 Example Simulation

This section will briefly explain how the example model is simulated. The emphasis is on the simulation of the parts that involve the environment. The **LetExp**ression is evaluated by first evaluating the definitions and then by evaluating the expression.

Definition Evaluation

The simulation of the definitions in the example model is already explained informally using figure 3.15. This figure does not show how the actual state looks after evaluating definitions to environments. Figure 3.16 explains this better. The figure shows how the **SeqDef**inition *"x=4;y=8;"* is evaluated. It shows a branch of the first four states of the simulation.

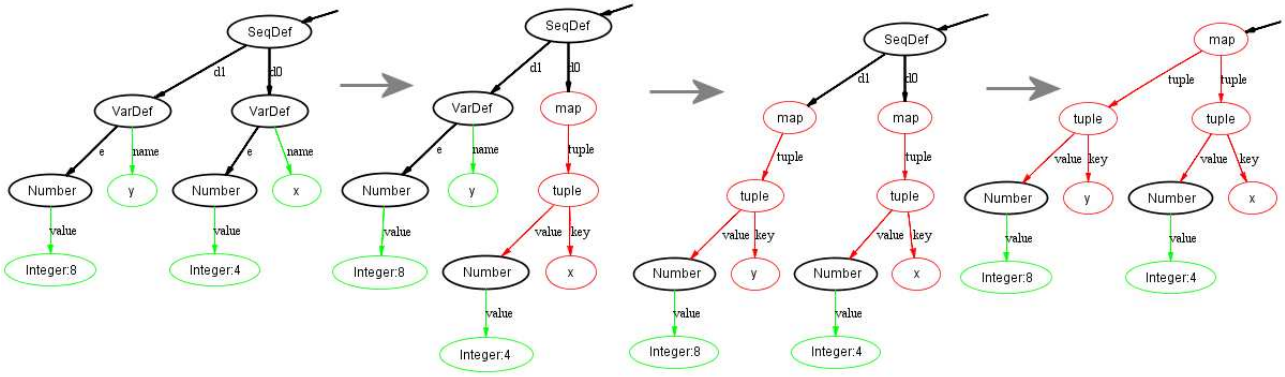


Figure 3.16: Evaluation of the SeqDef to an environment

The first transition evaluates the **VarDef**inition *"x=4"* by applying rule 3.20 . The result is a function with one tuple. The function is represented by the *map* node. The *map* contains one *tuple* node which contains the key and value of the tuple. This second state is no longer conformant to the language metamodel: the **d0** reference of the **SeqDef**inition points to a function. This clearly shows that the state conforms to an extension of the language metamodel as already explained.
The second transitions evaluates *"y=8"* to a function analogous to the first transition. The last transition merges the two environment functions to a new environment function by applying rule 3.23. Environment bound to **d0** is overridden by the environment bound to **d1** resulting in a new environment as shown in the last state.
The full evaluation of the other definitions (**InDef**, **AndDef**) is not given here, this is done because the other definitions do not introduce new rule constructs. The tool can be used if one wishes to inspect all the states found during simulation. The tool visualizes the full states and allows easy simulation.

Variable Evaluation

Now that the evaluation of definitions to function is explained we can explain how these functions are actually used as environments in which expressions are

evaluated. This is explained using figure 3.17.

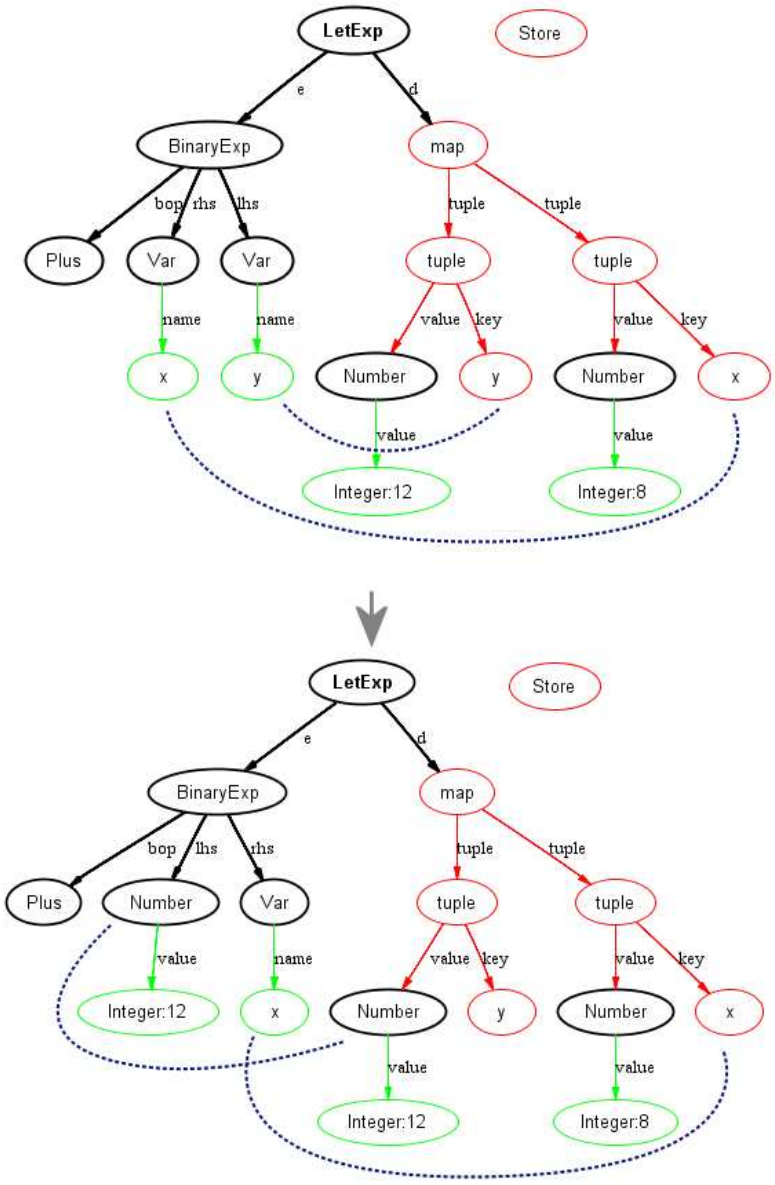


Figure 3.17: Reading a variable from the environment

The top state in the figure shows a **LetExp**ression for which the definition is fully evaluated to a function with two tuples "x" $\mapsto$ **Number** (**value** $= 8$) and "y" $\mapsto$ **Number** (**value** $= 12$) (represented by the *map* node). The expression $x + y$ must be evaluated within that environment.

The transition from the top state to the bottom state is proved by rule 3.34:

$$\frac{\mathrm{E} \xrightarrow{env_\rho = env_\rho[p_\rho]} \mathrm{NE}}{\mathbf{LetExp}\,(\mathbf{d} = p_\rho, \mathbf{e} = \mathrm{E}) \rightarrow \mathbf{LetExp}\,(\mathbf{d} = p_\rho, \mathbf{e} = \mathrm{NE})} \tag{3.34}$$

The variables p_ρ is bound to the environment:

$$p_\rho = \{"x" \mapsto \mathbf{Number}\,(\mathbf{value} = 8)\,, "y" \mapsto \mathbf{Number}\,(\mathbf{value} = 12)\} \tag{3.35}$$

For simplicity we assume that the current environment env_ρ is empty. Because it is empty we can conclude that $env_\rho\,[p_\rho] = p_\rho$. Variable E is bound the **BinaryExp**ression and its child nodes as indicated by $\mathbf{BinExp}\,(\cdots)$ expression. The variable E is now substituted in the rule.

$$\frac{\mathbf{BinExp}\,(\cdots) \xrightarrow{env_\rho = p_\rho} \mathrm{NE}}{\mathbf{LetExp}\,(\mathbf{d} = p_\rho, \mathbf{e} = \mathbf{BinExp}\,(\cdots)) \rightarrow \mathbf{LetExp}\,(\mathbf{d} = p_\rho, \mathbf{e} = \mathrm{NE})} \tag{3.36}$$

The transition condition can be met by applying the rule for evaluating the **lhs** of a **BinExp**. This rule is not given for this language but it is identical to rule 3.5 for the simple imperative language.

$$\frac{\mathrm{L} \rightarrow \mathrm{NL}}{\dfrac{\mathbf{BinExp}\,(\mathbf{lhs} = \mathrm{L}, \mathbf{bop} = \mathrm{O}, \mathbf{rhs} = \mathrm{R}) \xrightarrow{env_\rho = p_\rho} \mathbf{BinExp}\,(\mathbf{lhs} = \mathrm{NL}, \mathbf{bop} = \mathrm{O}, \mathbf{rhs} = \mathrm{R})}{\mathbf{LetExp}\,(\mathbf{d} = p_\rho, \mathbf{e} = \mathbf{BinExp}\,(\cdots)) \rightarrow \mathbf{LetExp}\,(\mathbf{d} = p_\rho, \mathbf{e} = \mathrm{NE})}} \tag{3.37}$$

The variable L, O and R are not filled in to keep the proof readable. L is bound to $\mathbf{Var}(\mathbf{name} = "y")$. The transition condition $\mathrm{L} \rightarrow \mathrm{NL}$ can be expanded by applying rule 3.33. Also note that the new environment propagates to the transition condition.

$$\frac{\mathbf{Var}(\mathbf{name} = "y") \xrightarrow{env_\rho = p_\rho} env_\rho\,("\mathrm{y}")}{\dfrac{\mathbf{BinExp}\,(\mathbf{lhs} = \mathrm{L}, \mathbf{bop} = \mathrm{O}, \mathbf{rhs} = \mathrm{R}) \xrightarrow{env_\rho = p_\rho} \mathbf{BinExp}\,(\mathbf{lhs} = \mathrm{NL}, \mathbf{bop} = \mathrm{O}, \mathbf{rhs} = \mathrm{R})}{\mathbf{LetExp}\,(\mathbf{d} = p_\rho, \mathbf{e} = \mathbf{BinExp}\,(\cdots)) \rightarrow \mathbf{LetExp}\,(\mathbf{d} = p_\rho, \mathbf{e} = \mathrm{NE})}} \tag{3.38}$$

This concludes the proof. The result of $env_\rho\,("\mathrm{y}")$ is $\mathbf{Number}\,(\mathbf{value} = 12)$ because $"y" \mapsto \mathbf{Number}\,(\mathbf{value} = 12) \in env_\rho$.
All the meta-variables that were changed can now be filled in to construct the new state. The dots $(\cdots)$ indicate that nothing is changed with respect to the proof 3.38.

$$\frac{\dfrac{\mathbf{Var}(\mathbf{name} = "y") \xrightarrow{env_\rho = p_\rho} \mathbf{Number}\,(\mathbf{value} = 12)}{\mathbf{BinExp}\,(\cdots) \xrightarrow{env_\rho = p_\rho} \mathbf{BinExp}\,(\mathbf{lhs} = \mathbf{Number}\,(\mathbf{value} = 12)\,, \cdots)}}{\mathbf{LetExp}\,(\cdots) \rightarrow \mathbf{LetExp}\,(\mathbf{d} = p_\rho, \mathbf{e} = \mathbf{BinExp}\,(\mathbf{lhs} = \mathbf{Number}\,(\mathbf{value} = 12)\,, \cdots))} \tag{3.39}$$

In summary this says in the new state the **lhs** of the **BinExp**ression is replaced by $\mathbf{Number}\,(\mathbf{value} = 12)$. This is exactly what has happened in the new state in figure 3.17.

3.5 Supporting Graph Structures: Activity Diagrams

Until now we only showed that *SemLang* is able to define the semantics of languages similar to the languages for which Plotkin described the semantics. The models of these languages are always trees. This is of course a logical consequence when using abstract syntax trees (ASTs). The metamodel of the languages we introduced so far always use containment references. This prevents cross references in the model and it ensures that the models are always trees.
However, in MDE metamodels are also allowed to contain cross references. Therefore the models in MDE are generally graph structures. Plotkin style SOS, however, is based on EBNF and therefore the structures are always ASTs. The rules in standard SOS therefore only deal with tree structures.
However, *SemLang* must be able to deal with graph structures. The way *SemLang* deals with graph structures is illustrated by an example language for which the models are graphs. This example language is the Activity Diagram Language.

3.5.1 MetaModel

The Activity Diagram language is a simple diagram language that can be used to visualise the control flow of programs. This is similar to the Activity Diagrams found in UML. An example diagram is given in figure 3.18, it represents a simple count-to-five program. The visual syntax is taken from UML Activity Diagrams.

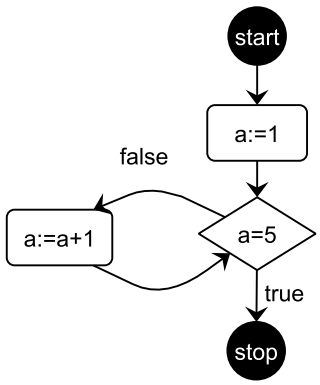


Figure 3.18: Concrete visual model of an Activity Diagram

There are different types of nodes in the example program. There is a *start* and *stop* node. After the *start* node there is an *assignment* node in which variable a is assigned the value 1. After the *assignment* node there is *test* node in which the expression $a = 4$ acts as an condition. Depending on the value of the condition either the *stop* node is visited or another *assignment* $a := a + 1$ node is visited.
The metamodel of the Activity Diagram language is given in figure 3.19. An Activity **Diagram** consists of **Node**s. Each node has a **name**, this is done for the concrete textual syntax of the language, which will not be explained here.
A node is either a **StopNode** or a **SeqNode**. A **SeqNode** is a node which has reference to a **next** node. A **SeqNode** is either a **StartNode**, **Test** node

or an **Assign**ment node. Both the **Test** and **Assign**ment nodes consist of an **Exp**ression. For the **Test** node this expression acts as an condition and for the **Assign**ment node this acts as the value that is bound to the variable.
It is important to see that the references **next** of **SeqNode** and **alternative** of the **Test** node are cross references. This is indicated by the open arrow. The lines that have a filled diamonds represent containment references.

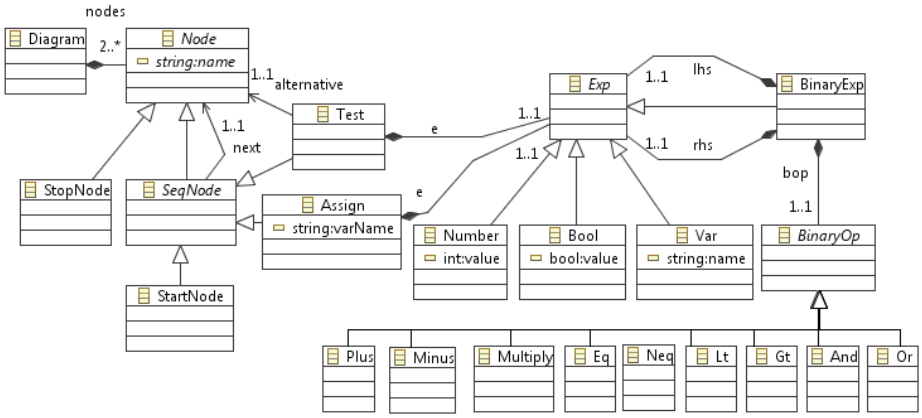


Figure 3.19: Activity Diagram language metamodel

The example diagram in figure 3.18 can also be represented as an **Diagram** object that conforms to the metamodel. This model is presented in figure 3.20. The new things in this model are the blue arrows. Blue arrows represent cross-references. The thick black arrows represent containment references. This is important to know. Also note that the black-arrows always form a tree, this is a natural result of the use of containment references.

3.5.2 Semantics

Informal Semantics

The semantics for the Activity Diagram Language is not very complex. The semantics of the expressions will not be explained. They are already explained for the different languages in the previous sections. The focus is on the semantics of the nodes.
An Activity Diagram should have one **StartNode** and at least one **StopNode**. A diagram is evaluated by moving through the nodes and by evaluating each node. The evaluation starts at the **StartNode**. Then we move to the next node. Evaluation of a diagram is complete if a **StopNode** is reached.
An **Assign**ment node is evaluated by first evaluating the expression **e**, and then by binding the new value to the variable and moving to the next state.
A **Test** node is evaluated by first evaluating the condition expression **e**. If the condition is true the **next** node is taken. The **alternative** node is taken otherwise.
This concludes an informal explanation of the semantics. However, it is important to know that a node can be evaluated multiple times; for example when

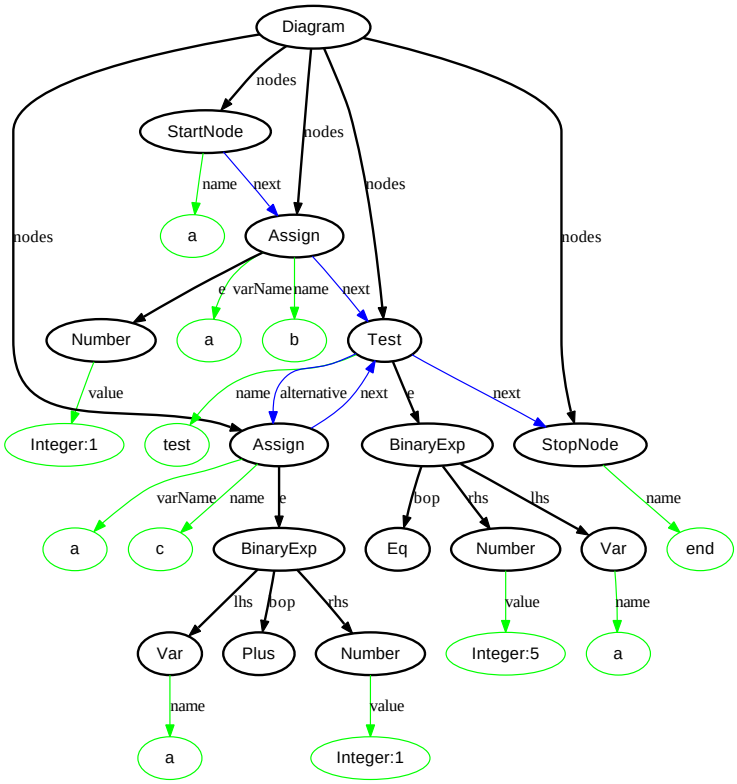


Figure 3.20: Object model of an Activity Diagram

there is a loop in the activity diagram. This requires a special copy algorithm that copies the node and rewires all references. SOS cannot be used because it does only deals with trees in which loops are never encountered.
The Semantic Language has to be extended to deal with the graph structure. The copy algorithm must also be clearly explained. This is done by first explaining the rules of the Activity Diagram Language. The copy algorithm will be explained afterwards.

Formal Semantics

In this section the *SemLang* rules will be explained. There is a special emphasis on the fact that we *copy* certain objects. Note that this was not done in the explanation of all the previous rules. However, objects were always copied until now, there was no need to differentiate between a *copy* and a *reference*.
We first specify the valid end states. Both **Number**s and **Bool**eans are computed values, these are used in expressions. A diagram is calculated if it is reduced to a **StopNode**. The valid end states are defined as follows:

$$Computes = \{\mathbf{StopNode}, \mathbf{Number}, \mathbf{Bool}\} \tag{3.40}$$

The Activity Diagram language also needs a store in which variables can be stored and updated. This is similar to the store in the simple imperative language:

$$store : String \rightarrow Object \tag{3.41}$$

Now the rules can be specified. The first rule matches the root **Diagram** object and evaluates it to a *copy* of a **StartNode** (see rule 3.42). The pattern of this rule has a new construct: NS $\langle \exists$n $\in$ **StartNode**$\rangle$. This means that the rule only matches if there exists an object n in NS that conforms to a **StartNode**. This is called existential quantification.

$$\textbf{Diagram}\,(\textbf{nodes} = \text{NS}\,\langle \exists \text{n} \in \textbf{StartNode} \rangle) \rightarrow \text{n} \tag{3.42}$$

If there are multiple **StartNode**s then this rule matches all of them. Therefore multiple **StartNode**s in a single **Diagram** introduces nondeterminism for the first transition.
The next rule, rule 3.43, evaluates a **StartNode** to a *copy* of the **next** node.

$$\textbf{StartNode}\,(\textbf{next} = \text{N}) \rightarrow \text{N} \tag{3.43}$$

An **Assign**ment node is evaluated by first evaluating the expression **e**. Rule 3.44 defines this behaviour. It is important to note that the rhs of the rule, **Assign** $(\cdots)$, is a constructor: it constructs a new object that conforms to the **Assign** class. All attributes and references are *copied*:

- **varName** is assigned to a *copy* of VN
- **e** is assigned to a *copy* of NE
- **next** is assigned to a *copy* of N

This is an important detail: after the copy algorithm is explained one will see that this copying preserves the old **Assign** object if that object is in the subgraph of the object N.

$$\frac{\text{E} \rightarrow \text{NE}}{\textbf{Assign}\,(\textbf{varName} = \text{VN}, \textbf{e} = \text{E}, \textbf{next} = \text{N}) \rightarrow \textbf{Assign}\,(\textbf{varName} = \text{VN}, \textbf{e} = \text{NE}, \textbf{next} = \text{N})} \tag{3.44}$$

The next **Assign**ment rule (rule 3.45) simply evaluates to a *copy* of the **next** node. However, a new tuple is added to the store. This is identical to the store updates in the simple imperative language. Therefore this will not be explained further.

$$\textbf{Assign}\,(\textbf{varName} = \text{VN}, \textbf{e} = E, \textbf{next} = \text{N}) \xrightarrow{store' = store[\{\text{VN} \mapsto E\}]} \text{N} \tag{3.45}$$

The first **Test** rule simply evaluates the condition **e** first. This is defined by rule 3.46. The rule looks like the first **Assign**ment rule (rule 3.44). Again it is important to see that the rhs of the rule constructs a new **Test** object and binds copies of NC, N, and A to **e**, **next** and **alternative** respectively.

$$\frac{\text{C} \rightarrow \text{NC}}{\textbf{Test}\,(\textbf{e} = \text{C}, \textbf{next} = \text{N}, \textbf{alternative} = \text{A}) \rightarrow \textbf{Test}\,(\textbf{e} = \text{NC}, \textbf{next} = \text{N}, \textbf{alternative} = \text{A})} \tag{3.46}$$

The last two **Test** rules simply evaluate to either a *copy* of the **next** node or a *copy* of the **alternative** node, depending on the value of the condition C. This is similar to the rules for the **IfExp** in the functional language.

$$\frac{C.value}{\mathbf{Test}\,(\mathbf{e} = C, \mathbf{next} = \mathrm{N}, \mathbf{alternative} = \mathrm{A}) \rightarrow \mathrm{N}} \tag{3.47}$$

$$\frac{\neg C.value}{\mathbf{Test}\,(\mathbf{e} = C, \mathbf{next} = \mathrm{N}, \mathbf{alternative} = \mathrm{A}) \rightarrow \mathrm{A}} \tag{3.48}$$

The Copy Algorithm

Application of the *SemLang* rules often involves copying parts of the model. This is performed by the copy algorithm. As already explained this copy algorithm is not that important for DSLs in which the models are always trees. However, the copy algorithm is crucial for languages in which the models are graphs.

The copy algorithm is basically a breath-first-search algorithm. All objects are copied by exploring the graph in a breath first fashion. The copy algorithm ensures that objects are only copied once. It treats *containment* references in a special way: they are always copies. This ensures that the *containment* edges never form a cycle.

```
COPY-ALGORITHM(object)
 1  ▷ a map from old object to copied objects
 2  copies ← Object → Object
 3  ▷ a queue of object that must be copied
 4  queue ← [object]
 5  ▷ a queue of objects that must be rewired
 6  updateQueue ← []
 7  ▷ perform a breath-first-copy
 8  while queue ≠ ∅
 9        do object ← queue.dequeue()
10           if object isA function
11             then copy ← COPY-FUNCTION(object)
12                  copies.add(object, copy)
13             else copy ← new object that conforms object.class
14                  copies.add(object, copy)
15                  ▷ add all attributes
16                  for att ∈ object.attributes
17                        do copy.set(att, object.get(att))
18                  ▷ enqueue all referenced objects
19                  for ref ∈ object.references
20                        do if object ∉ updateQueue
21                              then updateQueue.add(object)
22                           node ← object.get(ref)
23                           ▷ enqueue the node if it is a containment
24                           ▷ or if it is not copied before
25                           if node ∉ queue ∧ (ref.isContainment ∨ node ∉ copies.keys)
26                              then queue.add(referred)
27 ▷ rewiring: makes sure all references are updated
28 while updateQueue ≠ ∅
29       do object ← updateQueue.dequeue()
30          copy ← copies.getCopyOf(object)
31          for ref ∈ object.references
32                do node ← object.get(ref)
33                   ▷ make sure it refers to the copied object
34                   nodeCopy ← copies.getCopyOf(node)
35                   copy.set(ref, nodeCopy)
```

The copy algorithm has a same time complexity as the breath-first-search algorithm and is therefore $O(n + e)$, where n is the number of nodes in the model and e is the number of edges (references).

```
COPY-FUNCTION(f)
1  copy ← newfunction
2  ▷ copy each tuple
3  for key ↦ value ∈ f
4        do f ← f ∪ {COPY-ALGORITHM(key) ↦ COPY-ALGORITHM(value)}
5  return copy
```

3.5.3 Example Simulation

To understand the copy algorithm fully an example will be given in this section. Only one transition will be explained. This is the transition in which an **Test** object is evaluated to a new *Test* node. This is visualized in figure 3.21.

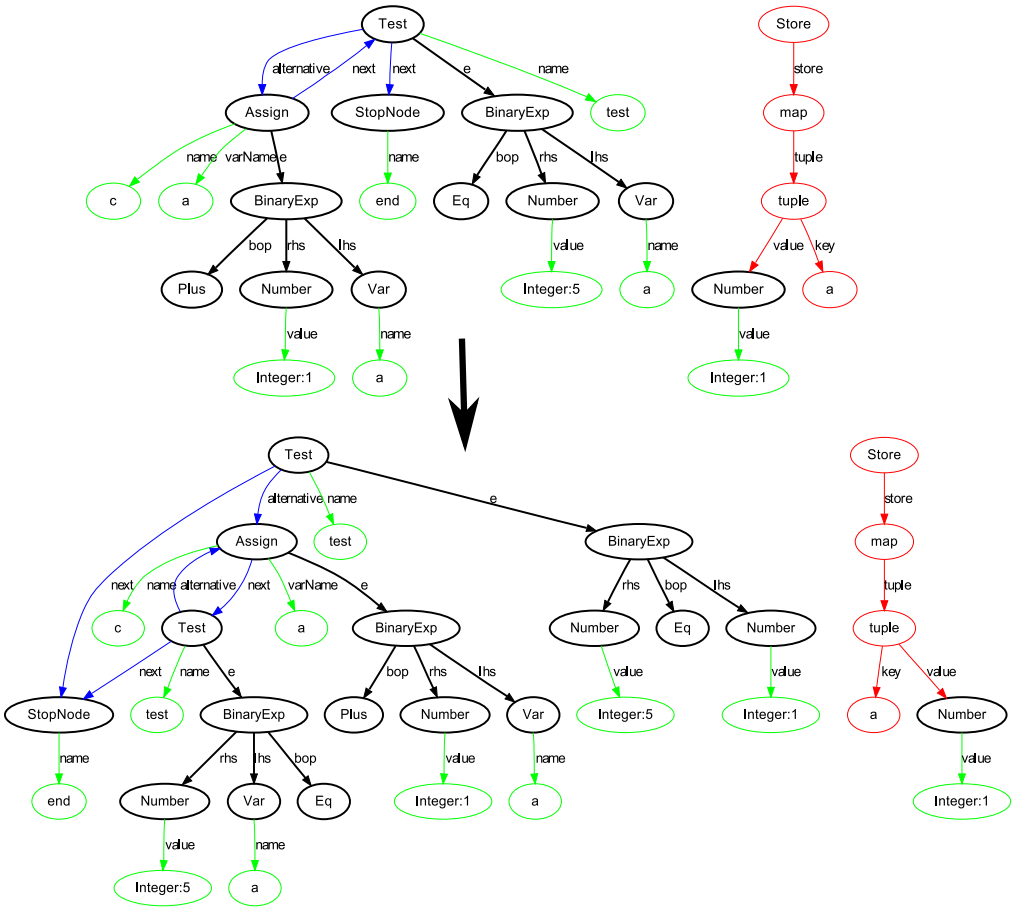


Figure 3.21: Example state transition for the example Activity Diagram

The rule that is proves this transition is rule 3.46. This rule repeated below. The rule shows how a **Test** node is evaluated to a newly created **Test** node. The new **Test** node is created by the constructor in the rhs of the rule: **Test** (**e** = NC, **next** = N, **alternative** = A).
This means that a new **Test** objects is created for which the references **e**, **next** and **alternative** are bound to *copies* of NC, N and A respectively.

$$\frac{\mathrm{C} \rightarrow \mathrm{NC}}{\textbf{Test}\,(\textbf{e} = \mathrm{C}, \textbf{next} = \mathrm{N}, \textbf{alternative} = \mathrm{A}) \rightarrow \textbf{Test}\,(\textbf{e} = \mathrm{NC}, \textbf{next} = \mathrm{N}, \textbf{alternative} = \mathrm{A})} \quad (3.49)$$

This explains why there are two *Test* nodes in the second state. The first **Test** node is the newly created node. The second **Test** node is the one copied due to the copy algorithm.

Another detail is that the **StopNode** is not copied twice. This is what one would expect until now. Because both the **alternative** and **next** objects of the **Test** node are copied. However, all objects that are copied within the a constructor are remembered. The copy algorithm keeps track of what is copied. Whenever a cross reference is encountered it is only copied when it is has not been copied. Otherwise it is just rewired to the copied object. This explains why The copied **Test** node in the second state does not copy the **StopNode** again: it is simply rewired to the copied **StopNode**.

3.6 Supporting Graph Structures: Petri Nets

The copy algorithm explains how the rules are applied to non tree models. However, the expressiveness of the rules is limited with respect to the semantics of graph based languages. It is for example impossible to define the semantics for Petri Nets.
Therefore the Semantic Language *SemLang* is again extended with new features to support more graph based languages. These extensions are explained by explaining another graph based semantic language: Petri Nets.
Again we first introduce the metamodel of Petri Nets. Then the semantics is explained informally and by the use of *SemLang*. New constructs are explained when they are introduced. In the end the example model will be simulated using the given *SemLang* rules.

3.6.1 MetaModel

A Petri Net is a graph in which the nodes are either *places* or *transitions*. *Places* can hold *tokens* and are often visualized as a circle. *Transitions* are visualized as black horizontal bars. Directed edges (also known *arcs*) connect *places* with *transitions* and vice versa. Arcs that connect *places* with *places* or *transitions* with *transitions* are not allowed.
A very simple Petri Net is given in figure 3.22. The Petri Net consists of three places (*a*, *b* and *c*) and two transitions (*t1* and *t2*). Six arcs connect the nodes with each other.

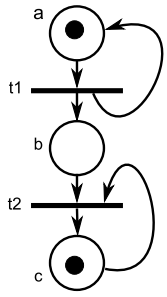


Figure 3.22: Concrete visual model of an example Petri Net

The metamodel of the Petri Net language is defined in figure 3.23. The static semantics that is needed to limit the set of valid models is not given. The focus of this thesis is on the dynamic semantics.

Petri Net diagrams are very simple compared to the other languages in this chapter. This is indicated by size of the metamodel. A Petri Net consists of **Node**s (with a **name**) and **Arc**s. A **Node** is either a **Place** or a **Transition**. The number of tokens at a **Place** is stored in the **count** attribute.

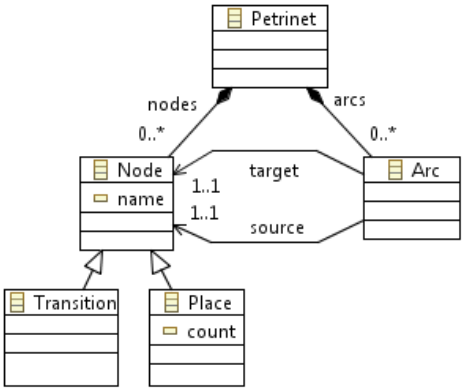


Figure 3.23: Petri Net language metamodel

Note that the metamodel given is not the only metamodel that is possible for Petri Nets. Different metamodels are possible but their semantic rules would also differ. For some metamodels it is difficult (or even impossible) to define the semantics using *SemLang*. The expressive power of *SemLang* is currently not known an is seen as a future research subject.

The Petri Net in figure 3.22 can now be visualised as a MDE model by using the metamodel. The model is given in figure 3.24. The model shows how the arrows are represented as **Arc** objects. The **Arc** object either has a **Transition** or a **Place** as its target/source. Also note that the *target* and *source* references are cross references.

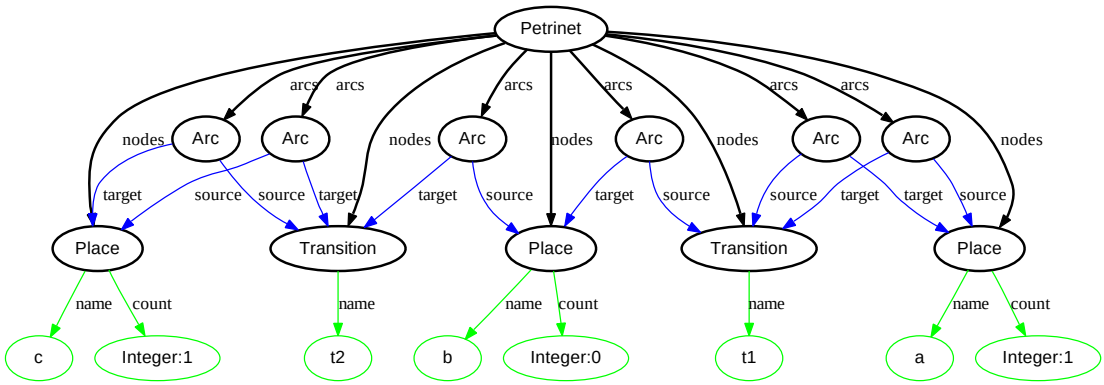


Figure 3.24: Example Petri Net model

3.6.2 Semantics

Informal Semantics

The **Transition** nodes in a Petri Net can *fire* if they are enabled. A **Transition** is enabled if each **Place** for which there is an **Arc** pointing to the **Transition** has at least one token. The firing of a **Transition** is a two-step process. First it *consumes* one token in each **Place** that points to the **Transition**. Then it *produces* one token for each **Place** if there is an **Arc** pointing from the **Transition** to the **Place**.
Transitions do not fire simultaneously. However, multiple **Transition**s can be enabled. This makes Petri Nets non-deterministic.
An example simulation of the example model is shown in figure 3.25. There are five states (labelled *S1* to *S5*). First **Transition** *t1* fires: it first consumes one token in **Place** *a* (at state *S2*). Then it produces a token in both *a* and *b* (at state *S3*). Note that the firing of a **Transition** involves two states.
Sequentially **Transition** *t2* fires. It consumes a token in both *b* and *c* (state *S4*) and then produces a token in **Place** *c* (state *S5*).

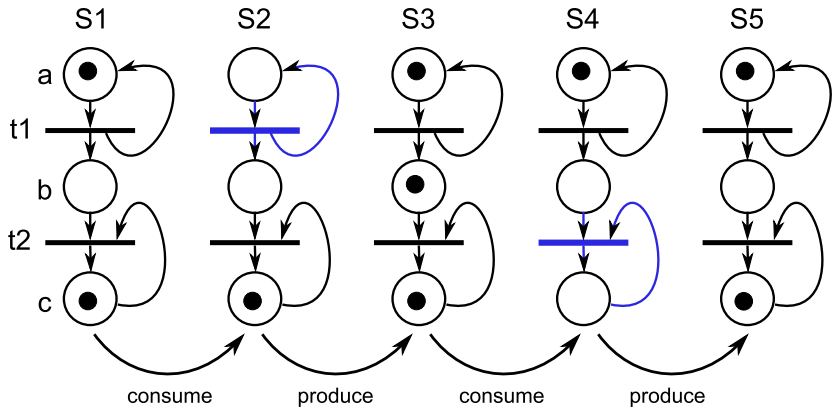


Figure 3.25: Petri Net example simulation

Note that the firing of a **Transition** is broken into two steps. The semantics could also be defined in a way that the firing of a **Transition** only involves one state transition in which the consuming and producing executes simultaneously. We choose to break it down into two steps to keep the rules smaller.

Formal Semantics

The *SemLang* semantics for Petri Nets only consists of two rules. One rule for the consuming of tokens and one rule for the producing of tokens. However, these two rules are rather complex. Both rules also use a store named *firing*. This store is a singleton: it only stores one object. The *firing* singleton stores the current **Transition** that is firing. It is defined as follows:

$$firing : Transition$$

This is a new addition to *SemLang*: functions (stores and environments) can also be singletons.

The rules contain both existential quantification (explained in the previous section) and universal quantification. These new constructs are explained first to understand the *SemLang* rules for Petri Nets.

- **Existential quantification**
  - $\exists var \mid condition$
    Holds if there exists at least one item for which the *condition* holds
  - $\exists var \in \mathbf{ClassName}$
    Holds if there exists at least one item that conforms to the class named **ClassName**
  - $\exists var \in \mathbf{ClassName} \mid condition$
    Holds if there exists at least one item that conforms to the class named **ClassName** and for which the *condition* holds.
- **Universal quantification**
  - $\forall var \mid condition$
    Holds if for all items the *condition* holds
  - $\forall var \bullet assignment$
    Executed the *assignment* for each item. This form of quantification can only be used in the constructor part of a rule, because it changes the value of an attribute
  - $\forall var \mid condition \bullet assignment$
    Executes the assignment for each *item* for which the *condition* holds. This form of quantification can also only be used in the constructor part of the rule.

The first rule (rule 3.50) is the consume rule. It has a number of extensions to *SemLang*. The rule matches a **Petrinet** if there exists a node that conforms to a **Transition**. This is defined by the $\langle \exists t \in \mathbf{Transition} \rangle$ construct. This construct was explained in the previous section.
The condition $firing\,() = \mathbf{Nil}$ is true if the singleton *firing* does not point to an object. This condition makes sure that the two rules do not fire simultaneous. The consume rule only fires if there is no current **Transition** firing.

$$\frac{firing\,() = \mathbf{Nil}}{\mathbf{Petrinet}\left(\begin{array}{l}\mathbf{nodes} = \mathrm{N}\,\langle \exists t \in \mathbf{Transition} \rangle\,, \\ \mathbf{arcs} = \mathrm{A}\,\langle \forall \mathrm{a} |\, (\mathrm{a.target} = \mathrm{t}) \rightarrow (\mathrm{a.source.count} > 0) \rangle\end{array}\right) \xrightarrow{firing' = \hookrightarrow \mathrm{t}} \mathbf{Petrinet}\left(\begin{array}{l}\mathbf{nodes} = \hookrightarrow \mathrm{N}, \\ \mathbf{arcs} = \mathrm{A}\,\langle \forall \mathrm{a} \mid \mathrm{a.target} = \mathrm{t} \bullet \mathrm{a.source.count} := \mathrm{a.source.count} - 1 \rangle\end{array}\right) \tag{3.50}$$

The arcs are bound to the meta-variable **A**. There is also a condition that must be met for the arcs:

$$\langle \forall \mathrm{a} \mid\ (\mathrm{a.target} = \mathrm{t}) \rightarrow (\mathrm{a.source.count} > 0) \rangle$$

This is a form of universal quantification. It means that the rule only matches if for all **Arc**s a the condition $(a.target = t) \rightarrow (a.source.count > 0)$ holds. The

$\rightarrow$ symbol in the condition is the logical implication operator. Intuitively this means that the rule only matches if all **Place**s that point to the **Transition** t have at least one token.
The constructor also has a universal quantification construct. However, this construct also has an assignment in it. The meaning of the universal quantification is that for each **Arc** a for which the condition $a.target = t$ holds the assignment $a.source.count := a.source.count - 1$ is executed.

$$\langle \forall \text{a} \mid \text{a.target} = \text{t} \bullet \text{a.source.count} := \text{a.source.count} - 1 \rangle$$

The effect of this is that for each source **Place** the **count** attribute is decremented by one. This is exactly what should happen when the tokens are consumed.
Another new construct is the use of the $\hookrightarrow$ symbol. This symbol can be placed before meta-variables and indicates that an object should not be copied. A variable that is preceded by the $\hookrightarrow$ is called a *referenced variable*. For example $\hookrightarrow$ N indicates that the object(s) in N should be referenced and not be copied. The copy-algorithm is aware of all objects that are referenced within a constructor. Whenever it performs a copy it will remember all referenced objects and rewire all references to make sure they point to the referenced objects. An example will make this easier to understand.
Suppose we have a constructor that looks like **Petrinet** (**nodes** $=\hookrightarrow$ N, **arcs** $=$ A). When the **Arc**s in A are copied it will not copy references to **Node** objects if the **Node** is in N. Instead it will make sure that the reference will point to the **Node** in N.
The consume rule also updates the store. This is defined by $firing' =\hookrightarrow$ t. It makes the *firing* singleton point to a *reference* of the **Transition** object t.
The produce rule (rule 3.51) should produce one token in all target **Place**s. It only matches if the condition $\neg(firing\,() = \textbf{Nil})$ holds: i.e. it only matches when there is a **Transition** firing.

$$\frac{\neg(firing\,() = \textbf{Nil})}{\begin{array}{l}\textbf{Petrinet}\,(\textbf{nodes} = \text{N}, \textbf{arcs} = \text{A}) \\ \xrightarrow{firing'=\textbf{Nil}} \\ \textbf{Petrinet}\left(\begin{array}{l}\textbf{nodes} =\hookrightarrow \text{N}, \\ \textbf{arcs} = \text{A}\,\langle \forall \text{a} \mid \text{a.source} = firing\,() \bullet \text{a.target.count} := \text{a.target.count} + 1\rangle\end{array}\right)\end{array}} \quad (3.51)$$

The only new thing in the produce-rule is the universal quantification with an assignment in the constructor of the rule:

$$\langle \forall \text{a} \mid \text{a.source} = firing\,() \bullet \text{a.target.count} := \text{a.target.count} + 1\rangle$$

This makes sure that for all target **Place**s the **count** attribute is incremented by one. This is exactly what should happen: it produces one token in the target **Place**s.

3.6.3 Example Simulation

The meaning of the new constructs is easier to understand by giving an example simulation of the model. The simulation of the Petri Net given in figure 3.25 is

now represented in figure 3.26. The figure shows the first three states and two transitions of the simulation in figure 3.25.

The first state shows the initial model. Note that **Transition** named *t1* has a thicker border, this is because this transition is the one that will fire.

The second state shows how the application of the consume rules alters the model. The **count** attribute of the **Place** named *a* is decremented by one, resulting in a **count** value of zero. The thick border around the value of the **count** attribute emphasizes this.

The singleton store named *firing* is updated to point to a *reference* of the **Transition** *t1* in the second state. This is why the *firing* singleton points to the **Transition** *t1* within the **Petrinet**. No copy is made of the **Transition** object. That would happen if the consume rule updated the *firing* singleton like $firing' = \mathrm{t}$ instead of updating it to a *reference* of the variable: $firing' =\hookrightarrow \mathrm{t}$.

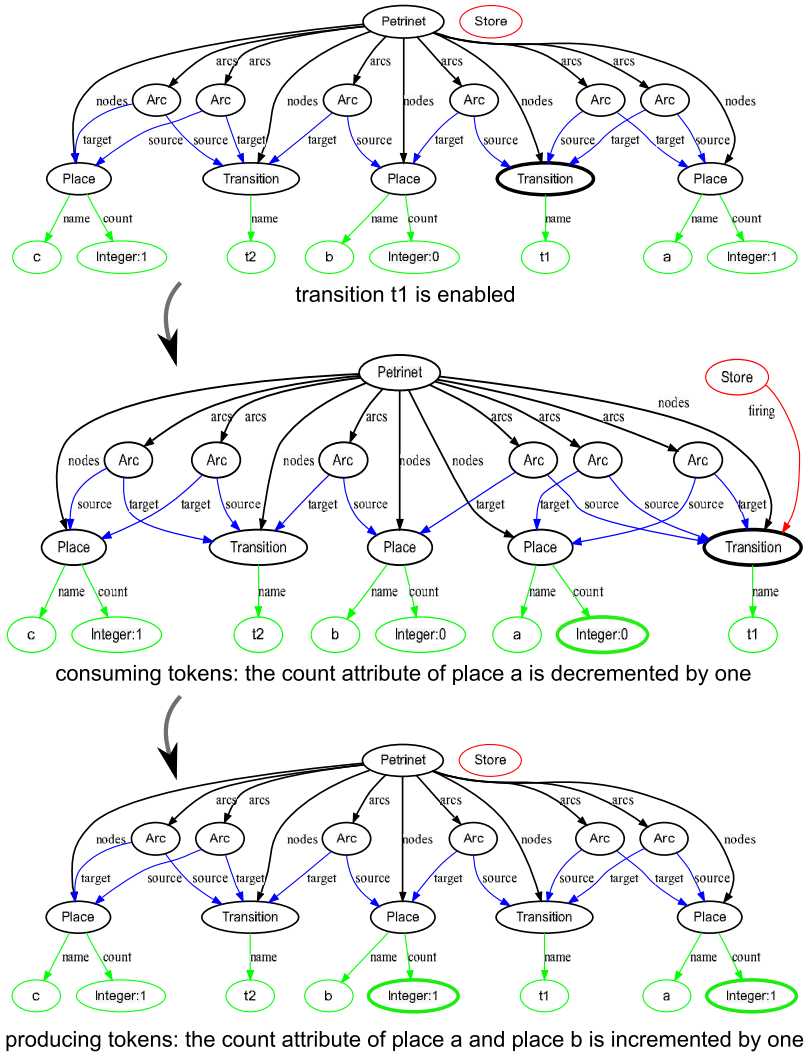


Figure 3.26: Petri Net example model simulation

The third state shows the result of the produce rule. The singleton *firing* points to nothing (internally it points to **Nil**). The places named a and b are updated: their count attribute is incremented by one.
Another important thing to notice is that the effect of the reference symbol $\hookrightarrow$ in the constructor of both rules. The **Arc** children of the **Petrinet** object are not copied but *referenced*. If they were copied it would mean that all **Node** objects will be copied twice; firstly when all **Arc**s are copied, and secondly when all **Node** children of the **Petrinet** are copied. This is clearly not desirable for Petri Nets; the graph must be kept intact. This is exactly why the reference symbol $\hookrightarrow$ is introduced. It allows the specifier of the rule to specify when objects must be *copied* or *referenced*

3.7 Conclusion

This chapter showed in an informal but detailed way how SOS can be adapted in order to make it useful in an model driven context. The first few sections showed how the semantic language *SemLang* supports the specification of the semantics of tree based languages.
We showed how *SemLang* could be used to specify the semantics for imperative and functional languages. A broad range of programming constructs are supported. It showed how *SemLang* can be used to specify the semantics for expressions, stores and declarations.
SemLang can also be used for the definitions of functions and (recursive) function invocation. Different parameter passing techniques like pass *by reference*, *by value* and *by name* are also supported. These techniques are not explained in this chapter but they do not introduce new constructs in the rules of *SemLang*. For more details we refer to the disc in the appendix A. The disc has a collection of example DSLs with their semantics specified in *SemLang*.
This chapter also showed how *SemLang* supports the specification of the semantics of certain graph based languages. This is an extension to the Plotkin style SOS, as it only supports tree based states. The following additions to *SOS* were needed to support the graph based languages.

- An explanation of the copy algorithm for graphs
- Support for referencing instead of copying
- Support for existential and universal quantification

These additions increased the expressiveness of *SemLang* and makes *SemLang* applicable to more languages (tree based and certain graph based languages). However, it is important to understand that *SemLang* cannot be used for any graph based languages. The expressiveness of *SemLang* compared to other approaches is not researched and therefore still an open issue. More discussing on the the expressiveness can be found in chapter 6.

4

Formalizing the Semantic Language DSL

4.1 Introduction

This chapter provides a mathematical foundation for the Semantic Language presented in the previous chapter. The formalization is based on formalization of SOS as described in [MRG07, AFV01]. Graph theory is also used to formalize models and states.
First a formalization of the semantic domain is given. The semantic domain of SOS in general is a labelled transition system (LTS). The states and structure of the LTS is given in the first section.
Sequentially we formalize the structure of *SemLang* models. The last section describes how a *SemLang* model specifies a LTS. This is based on the fact that an SOS description can be seen as a Transition System Specification (TSS). A TSS specifies the valid set of LTS for a specific language.

4.2 The Transition System

The semantic domain for SOS and therefore also for *SemLang* is a terminal LTS, also called a Labelled Terminal Transition System (LTTS).

Definition 9 (Labelled Terminal Transition System) *A LTTS is a quadruple* $(Q, T, A, \rightarrow)$ *with* Q *as set of states, a set* A *of labels* α*, a relation* $\rightarrow \subseteq Q \times A \times Q$ *of labelled transitions (*(s, α, s') *is written as* $s \xrightarrow{\alpha} s'$*), and a set* $T \subseteq Q$ *of terminal configurations such that* $s \xrightarrow{\alpha} s'$ *implies* $s \notin T$

The set of states Q plays an important role. As explained in the previous chapter the states are extended models. In the previous chapter the states were visualised as graphs, however a formal definition for a state was not given.
The LTTS also includes a set of labels A. Labels are used in LTTSs are useful for concurrency and interactivity. However in this thesis we did not focus on those aspects. Therefore an empty set will be used for the labels $A = \emptyset$. A transition therefore has no label and is written as $s \rightarrow s'$.

4.2.1 Models as graphs

To formalize the states we must first formalize the MDE models. This is because states are essentially extended models. The visual representation of the example models in the previous chapters are graphs. In fact they are a special kind of

graph named *directed multigraphs* (this follows the definition of a model in section 2.2.2) A directed multigraph is basically a graph which allows self-loops and duplicated edges.

Definition 10 (Directed MultiGraph) *A directed multigraph or multidigraph G is a pair (V, E), where V is a set of vertices, and E is a multiset of ordered pairs $\{u, v\}$ such that $u, v \in V$ (the edges of G).*

A model however is not exactly a directed multigraph. The edges in models have a name and are called references. There are also two categories of references: *containment* references and *cross* references.
Containment references are references which model a containment relationship between two objects. They are not allowed to contain cycles: a model with only containment references is always a tree or a set of trees.
Cross references are the references that are not containment references. Cross references do not have the constraint for cycles.
In section 2.2.2 a formal definitions for a model was given. A new definition for a model is given below. This new definition is useful for the definition of the states Q of the LTTS in the semantic domain. So this formal definition is meant to be used in the formalization of $SemLang$. It does not capture the relationship between a model and its metamodel formally.

Definition 11 (Model) *A model M is a quadruple (O, o_1, A, R) where*

- *O is a set of objects or nodes which also contains a set of primitives $P \subset O$*
- *$o_1 \in O$ is the main node*
- *A is a set of labels*
- *R is the set of references which is a relation $R \subseteq O \times A \times O$ of ordered triples $\{o, \alpha, o'\}$. R is partitioned into a set R_a of attributes, a set R_c of containment references and a set R_r of cross references ($R = R_a \cup R_c \cup R_r$ and they are also pairwise disjoint $R_a \cap R_c = \emptyset$, $R_a \cap R_r = \emptyset$ and $R_c \cap R_r = \emptyset$)*

We require the following conditions on a model M.

- *$o \in P \rightarrow \neg\exists\{o, \alpha, o'\} \in R$, i.e all primitives are leaf nodes*
- *$\{o, \alpha, o'\} \in R_a \rightarrow o' \in P$, i.e all attribute references refer to primitives*
- *The graph representation of a model is a connected graph*
- *The graph representation of a model in which the R_r edges are removed is called a tree (a connected graph without cycles)*

Sequentially the definition of the model of an object o is given. The model of an object o is basically a model which contains all the objects that can be reached from object o. This definition is given because it is used in the next sections of this chapter.

Definition 12 (Model of an Object) *Given a model (O, o_1, A, R). The model of an object $o \in O$ denoted* MODEL(o) *is a model $(O_{model}, o, A_{model}, R_{model})$ in which:*

- $O_{model} = \{x \in O \mid \textsc{Path}(x, o)\}$
- $R_{model} = \{(o_1, \alpha o_2) \in R \mid o_1 \in O_{model} \wedge o_2 \in O_{model}$
- $A_{model} = \{\alpha \in A \mid (x, \alpha, y) \in R_{model}\}$

The definition shows that a model always has a single entry point o_1, called the main object. Another important aspect of models is that the subgraph which only contains containment references R_c is always a tree. This is a logical consequence from the fact that *containment* references model a containment relationship.
Note that the formal definition is a simplification of a model in the sense of EMF and ECORE [EMF, ECO]. The ECORE metametamodel includes more properties on references and model elements. It also supports lists. These features are not formalized to keep things to the essential core of models and to omit trivial details.
The definition does not capture the relation between a model and its metamodel. However, each object node $o \in O$ implicitly refers to its metaclass. This is implicitly done by giving the object a name that equals the name of its metaclass.
The last two conditions in the definition of a model refer to the *graph representation* of a model. This is because models are essentially a special type of directed graphs. This allows the use of all definitions and theories found in graph theory. The graph representation of a model is given below.

Definition 13 (MultiDiGraph representation of a Model) *A model $M = (O, o_1, A, R)$ can be represented as a multidigraph $G = (V, E)$ by the following mapping*

- *$O \rightarrow V$, i.e. all objects are vertices*
- *$\{o, \alpha, o'\} \in R \rightarrow \{o, o`\} \in E$, i.e. all references are edges*

4.2.2 State representation

The previous chapter informally explained that states are essentially extended models. This is analogous to the Plotkin style SOS in which states are essentially sentences of a value-added syntax. The main extension to models is the use of *stores* and *environment* which are represented as functions.

Definition 14 (Function) *A function E is a injective function $E : O \rightarrow O$ from object to objects.*

To support imperative languages a store is needed in which values can be stored, this was explained in section 3.3 in the previous chapter and can also be found in [Plo81]. This requires that a state must have some sort of construct to support stores. This is done by adding a set of functions that is separate from the model. Functional languages and imperative languages that support scope on the other hand need an environment. The previous chapter informally explained that these environments are functions that can be part of the extended model.
States are models with these two extensions. Formally they can be defined as follows.

Definition 15 (State) *A state* S *is a tuple* (E, S, O, M) *in which* E *and* S *are sets of environments,* O *is a set of objects,* M *is a model* (O_M, o_1, A, R) *for which* $O_M \subseteq O \cup E$ *and* $E \subset O_M$

- E *is called the set of environment functions* $env : O \to O$
- S *is called the set of store functions* $store : O \to O$
- O *is called the set of objects*

A model can directly be represented as a state. This transformation is needed to transform the initial model to the initial state. It is also useful for some other definitions.

Definition 16 (State representation of a Model) *Given a model* $M = (O_M, o_1, A, R)$. *The state of this model, denoted* Model2State(M), *is* (E, S, O, M) *in which* $E = S = \emptyset$ *and* $O = O_M$.

Using the definition of a model as an object we can give a definition of the state representation of an object. This transformation from object to state is needed for describing the semantic language.

Definition 17 (State representation of an Object) *Given an object* o *the state of this object, denoted* Object2State(o), *equals* Model2State(Model(o))

4.3 The Transition System Specification

A *SemLang* model specifies the semantics of a certain DSL. The semantic domain of *SemLang* is an LTTS. Therefore a *SemLang* model is essentially a specification for a LTTS. This is why *SemLang* models are called *Transition System Specifications* (TSS). TSSs were introduced in section 2.6 about SOS, however in order to formalize how *SemLang* specifies a LTTS we must first formalize the *SemLang* models.

4.3.1 *SemLang* Models

A *SemLang* specification is a model itself and therefore the general definition for a model can be used. However, this general definition is not useful for defining how a *SemLang* model specifies a LTTS. Therefore we use a different definition. A *SemLang* model consists of a set of predicates over *terminal states*, a set of *functions* that can act like stores or environments, and a set of *rules*.

Definition 18 (SemLang Model) *A SemLang model is a tuple* $(E, F_{env}, F_{store}, R)$ *in which* E *is a set of predicates over end states,* F_{env} *a set of functions (called environment functions),* F_{store} *a set of functions (called store functions) and* R *a set of rules.*

For each predicate $P \in E$ and state $S \in Q$ then S is an end state if and only if $P(S)$. The functions in a *SemLang* model are partitioned in *environment* and *store* functions. This is done because each type of function is used differently as explained in the previous chapter.

In order to define the rules we must first define *terms* that are used in the rules. Terms are simple patterns which match an object or construct an object. A term always refers to a metaclass and may also contain meta-variables which are in the set Var.

Definition 19 (Meta Variables) *The set of meta-variables Var is partitioned set of normal variables $\mathrm{x}, \mathrm{y}, \mathrm{z} \cdots$ and a set of computed variables $x, y, z \cdots$*

The distinction between normal and computed variables is needed because some meta-variables should only bind to computed objects. A computed object is an object for which the state of that object is an end state, i.e. an object o for which $P(\textsc{Object2State}(o))$ holds. Computed variables only match computed objects. Computed variables can be recognized by their italic typesetting.
The terms defined here are analogous to the term algebras as defined by Aceto et al [AFV01]

Definition 20 (Term) *The set of terms $\Pi(MM)$ over a metamodel MM and a set of meta-variables Var is the least set such that:*

- *each $x \in Var$ is a term*
- **ClassName**$(\mathbf{r_1} = \mathrm{T}_1, \cdots, \mathbf{r_n} = \mathrm{T}_\mathrm{n})$ *is a term, if* **ClassName** *is the name of a metaclass m in MM, $\mathbf{r_1}, \cdots, \mathbf{r_n}$ are a subset of the attributes and references of the metaclass m or of a superclass of m and $\mathrm{T}_1, \cdots, \mathrm{T}_\mathrm{n}$ are terms*

Definition 21 (1-Level Deep Term) *A term* **ClassName**$(\mathbf{r_1} = \mathrm{T}_1, \cdots, \mathbf{r_n} = \mathrm{T}_\mathrm{n})$ *is 1-level deep if $\mathrm{T}_1, \cdots, \mathrm{T}_\mathrm{n} \in Var$.*

Definition 22 (Variables in a Term) *A variable $x \in Var$ is in a term t (written as $x \in t$) if*

- *either $x = t$*
- *or $t =$* **ClassName**$(\mathbf{r_1} = \mathrm{T}_1, \cdots, \mathbf{r_n} = \mathrm{T}_\mathrm{n})$ *and there is an $i \leq n$ for which $x \in \mathrm{T}_\mathrm{i}$*

Definition 23 (Closed Term) *A term t is called a* ***closed term*** *if there are no variables $x \in Var$ for which $x \in t$ holds. Otherwise the term is called an* ***open term.***

Notice that $\mathbf{r_1}, \cdots, \mathbf{r_n}$ are not required to contain all the attributes and references of the metaclass m: some attributes or references may be skipped. Also note that $\mathbf{r_1}, \cdots, \mathbf{r_n}$ may also be attributes or references of a superclass of m: terms are aware of the inheritance in the metamodel MM.
Terms were already used in the rules in the previous chapter. Some sample terms are listed below:

- **BinExp** $(\mathbf{lhs} = l, \mathbf{bop} = \mathrm{o}, \mathbf{rhs} = \mathrm{r})$
  - A term for metaclass named **BinExp** with meta-variables l, o and r
- **IfExp** $(\mathbf{e0} = \mathrm{e}_0, \mathbf{e1} = \mathrm{e}_1, \mathbf{e2} = \mathrm{e}_2)$

– A term for metaclass named **IfExp** with meta-variables e_0, e_1 and e_2

The definition of a the set of terms $\Pi(MM)$ for a metamodel MM can now be used to give the definition of a rule for a specific metamodel.

Definition 24 (Rule) *Let MM be a metamodel, let t, t' range over $\Pi(MM)$, let x and x' range over Var in which $x \in t$ and $x' \in t'$. A rule is a tuple (H, C) where H is a set of premises $x \xrightarrow{\alpha} x'$ and predicates $P(x)$. C is the conclusion which is of the form $t \xrightarrow{\beta} t'$ in which:*

- *the lhs term of the conclusion is 1-level deep and is called the **pattern** of the rule*
- *the rhs term of the conclusion is called the **constructor** of the rule*

A rule is written as

$$\frac{H}{C}$$

Not all rules in the previous chapter fit in this definition. However the essential parts of a rule are captured in this definition. We do not aim at full formalization of the SOS language. The aim of the formalization is to give a sound mathematical basis that captures the essential features of *SemLang*.
The constructs that were used in the previous chapter but which are not formalized include: lists and universal and existential quantification over lists, functions, singleton functions and variable references.
The labels α and β were used in the previous chapter to update functions. The α label in a premise of a rule was used to update an environment functions. The β label in the conclusion was used to update store functions. This is similar to the Modular SOS (MSOS) approach taken by Mosses [Mos99, Mos02, Mos04, Mos06]. The next section, in which we will prove transitions in the LTTS, does not take these labels into account. Again this is done to keep the formalization small and understandable.

4.3.2 Proving Transitions

A *SemLang* model specifies a LTTS and is therefore a TSS. Until now we formalized the states Q of the LTTS and the *SemLang* models. A *SemLang* model specifies the transitions in the LTTS. A transition $s \xrightarrow{\alpha} s'$ in the LTTS is proved by a set of rules in the *SemLang* model. This proof is called the proof tree.
The proof trees in *SemLang* are essentially the same as the proof trees used in SOS. The main difference between SOS and *SemLang* is the fact that the states in *SemLang* are special kinds of graphs (explained in section 4.2.2). The difference in state representation has an impact on the TSS. A *SemLang* TSS differs from a SOS TSS. This also changes the way a proof tree is constructed.
Before we define a proof tree we must first define how the *pattern* term of a rule matches a state, and how the *constructor* term of a rule constructs a state. Both the pattern term and constructor term may be open terms (which contain variables). These variables are used in a so called *binding* σ. A binding essentially binds variables to objects.

Definition 25 (Binding) *Given an model (O, o_1, A, R). A binding for this model is a mapping $\sigma : Var \rightarrow O$.*

Definition 26 (Fully Bounded Term) *Given a term t and a binding σ. The term t is fully bound in σ if for all variables $x \in t$ there exists a tuple $(x, o) \in \sigma$*

The *pattern* term of a rule matches some objects. If an object o is matched a binding is created that binds all free variables in the *pattern* term to the direct neighbour objects of o.

Definition 27 (Term Model Matching) *Given a term t and a model $m = (O, o_1, A, R)$.* $\textsc{Match}(t, m) = \sigma$ *holds if the term $t =$* **ClassName**$(\mathbf{r_1} = x_1, \cdots, \mathbf{r_n} = x_n)$ *matches the model m if o_1 is an instance of the metaclass named* **ClassName**. *The result of the matching is an binding σ such that if r_i point to an object o_i (i.e. $(o_1, r_i, o_i) \in R$) then $(x_i, o_i) \in \sigma$.*

The *constructor* term of a rule is a term which constructs a new model. When a model is constructed the copy algorithm as explained in section 5.4 is used.

Definition 28 (Term Model Construction) *Given a term t which is fully bound in a binding σ. The construction of the term is an model $(O, o_1, A, R) =$* $\textsc{Construct}(t)$ *such that:*

- *if $t \in Var$ then the construction is a model which is a* ***copy of*** *the model representation of $\sigma(t)$*
- *If $t =$* **ClassName**$(\mathbf{r_1} = \mathrm{T}_1, \cdots, \mathbf{r_n} = \mathrm{T}_n)$ *then the construction is a model in which object o_1 conforms to the metaclass named* **ClassName**. *Each reference r_i of object o_1 is bound to the construction of term T_i. The sets O, A and R only contain elements which are connected to o_1*

The definitions for matching and construction of states are needed to setup a proof tree. The definition for a proof tree looks like the proof trees in SOS explained by [AFV01, MRG07, Mos04]. The proof tree for a transition is called a *finite upwardly branching tree.*

Definition 29 (Transition Proof Tree) *Given a metamodel MM, a set of all possible states Q_{MM} for MM , t, t' which range over $\Pi(MM)$, a SemLang model for metamodel MM with a set of rules R which specifies an LTTS $(Q, T, A, \rightarrow)$. A transition is $s \xrightarrow{\alpha} s$‘ is only in $\rightarrow$ if and only if a finite upwardly branching tree can be formed satisfying the following conditions:*

1. *all nodes are labelled by elements of $Q_{MM} \times A \times Q_{MM}$*
2. *the root node is labelled by $s \xrightarrow{\alpha} s$‘*
3. *for each n-ary node labelled $s_1 \xrightarrow{\alpha} s_2$ there is a rule $\frac{H}{t \xrightarrow{\beta} t'}$ and an interpretation of the meta-variables that occur in it such that* $\textsc{Match}(t, s1) = \sigma$ *and* $\textsc{Construct}(t') = s2$. *For each premise $x \xrightarrow{\alpha} x' \in H$ there is a labelled branch $s_b \xrightarrow{\alpha} s'_b$ such that $\sigma(x) = s_b$ and a rule with constructor term t'_b such that $\sigma(x') =$* $\textsc{Construct}(t'_b)$. *Each predicate $P(x) \in H$ must hold.*

4.4 Conclusion

This chapter provides a mathematical foundation for *SemLang*. It shows that *SemLang* relies on graph theory and on the foundations of SOS given by different authors [AFV01, MRG07, Mos04].
The semantic domain of *SemLang* are labelled terminal transition systems LTTS. We showed that the states are special kind of graph in which the objects and edges have special properties. The most important structures of *SemLang* models were also formalized.
It also became clear that *SemLang* models are essentially Transition System Specifications (TSSs). The mapping from a *SemLang* model to a LTTS was also given by defining the states in the LTTS. The mapping from the TSS to the transitions in the LTTS can be done by setting up prove trees. Again this approach is similar to the approach in SOS.
An important difference between SOS and *SemLang* is that the terms have a special task in *SemLang*. The terms in *SemLang* either match models or construct new models. By setting up a prove tree in the current state s_1 by matching models with the *pattern* terms in the rules, one can construct the new state by construction new models by using the *constructor* terms in the rules.
This chapter does not formalize all constructs and features of *SemLang*. The formalization is kept to the essential parts to keep it understandable. The main goal of the formalization is to give a formal basis for *SemLang*. The focus of this thesis is on the pragmatical aspects of *SemLang*. However this chapter also shows that there is a formal basis on which *SemLang* depends on.

5

Semantic Engine: Tool for the Semantic Language

5.1 Introduction

This chapter presents the implementation of the tool. The tool is called *Semantic Engine* and consists of an engine which is able to simulate models. An user interface is built upon the engine and provides better usability and supports visualization of the states.
This chapter begins with a list of requirements for the tool in section 5.2. Section 5.3 explains the architecture of the tool. Both a high level architecture as well as a more detailed level architecture description is given.
Section 5.4 is devoted to the quality attributes of the tool and is divided into runtime and non-runtime quality attributes. The last section concludes this chapter.

5.2 Requirements

Software systems are created to solve a problem in a problem domain. The problem can be decomposed into sub-problems from which requirements can be derived. The system solves the problem if it implements all the requirements.
The tool can be used to simulate models given a semantic specification. There is a list of requirements for the tool. However, these requirements are less strict because the tool is a proof of concept. Therefore we will not give a detailed list of requirements; only some explicit high level requirements will be given.
The main functional requirement of the tool is that it must be able to simulate model given a semantic specification. This, in turn, requires that the semantics for *SemLang* are implemented. Other, less important, functional requirements are that the tool must support state inspection and proof tree construction. Another feature that can be implemented is step-by-step simulation (debugging).
Non-functional requirements do not deal with the function of the tool. These requirements are less strict because the tool is just a proof of concept. However, this does not mean that there are no non-functional requirements. The performance of the tool is for example important because the tool is useless if the simulation is very slow. The documentation of the tool is also important, because this facilitates future extensions and future research. Usability and extensibility are also non-functional requirements are important.

5.3 Architecture

The architecture is first explained at a high level. Sequentially we explain how the *Semantic Language* is specified. Then the two main components of the tool are covered in more detail. These two components are the *Semantic Engine* and the *User Interface*. We also show which *design patterns* [GHJV00] are used in the architecture.

5.3.1 High Level Architecture

A high level view of the architecture is visualised in a component diagram in figure 5.1. The diagram shows the high level components of the tool, their interfaces and connections to other components. This component diagram is based on the component diagrams as described by UML [OMG].
The main component is the *Semantic Engine* (abbreviated to *SemEngine*) component. This component is responsible for simulating a model given a *SemLang* model that specifies the semantics for that model. Therefore there is a relation between the *SemEngine* and the *DSL Model* and *SemLang Model* components. A *SemLang Model* conforms to the Semantic Language Metamodel. The Semantic Language itself defined as a modelling language complete with a concrete syntax specification. To create a *SemLang* model one has to create a text file that conforms to this syntax.
The *SemEngine* component reads the models into memory and these model are represented as **ECore** [ECO] objects. Therefore there is a relation between *SemEngine* and the *EMF Ecore* component.

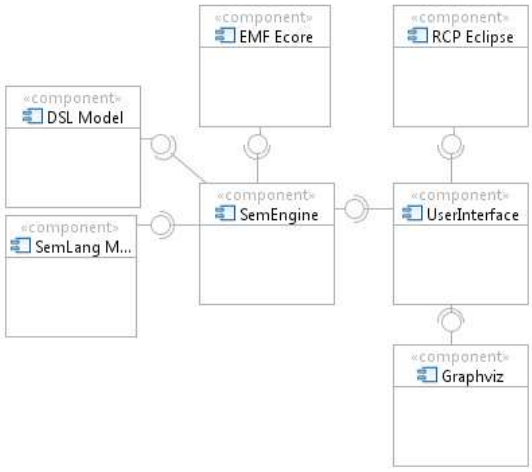


Figure 5.1: Component Diagram of the Tool

The three components on the right in the diagram are the components that are part of the graphical user interface. The tool can still be used without these components if no graphical user interface is needed.
The *User Interface* component contains all the visual elements that form the user interface window of the tool. The *User Interface* is built upon the Eclipse

Rich Client Platform [RCP] which is represented by the *RCP Eclipse* component. RCP is basically a rich library for creating user interfaces which is based upon SWT [SWT] and JFace [JFa].
The *Graphviz* component represents a software library [gra]. Graphviz is used to layout and render graph representations of the states.

5.3.2 The Semantic Language

A semantic specification is written in *SemLang*. *SemLang* itself is defined as a modelling language. The KM3 [JB06] and Textual Concrete Syntax [JBK06] framework is used to create the Semantic Language. KM3 is used as a metametamodel. The metamodel of *SemLang* is written in a text-file that conforms to the concrete syntax of KM3. The specification of the metamodel can be found on the disc accompanying this thesis.
A concrete syntax for *SemLang* was specified to make it easier to create *SemLang* models. TCS was used to create this concrete syntax specification. The TCS specification for *SemLang* can also be founds on the disc. We also recommend to study example *SemLang* models to become familiar with the concrete syntax of *SemLang*.

5.3.3 Architecture of the Semantic Engine

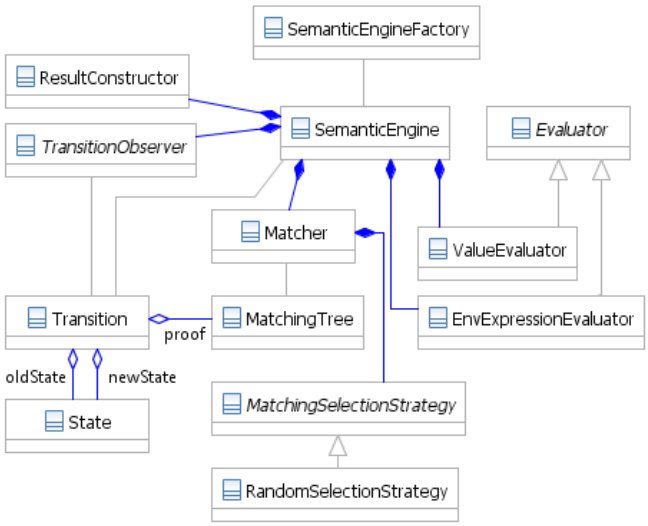


Figure 5.2: Class Diagram of the Semantic Engine

The *Semantic Engine* component is visualized as a class diagram in figure 5.2. To understand this diagram a description for each class is given.
The **SemanticEngine** class is the most important class and acts as a facade for other systems that interact with the engine. Therefore it follows the *facade design pattern*. It also fits the *composite design pattern* nicely because it is composed of a number of objects. This class is also responsible for managing simulations and keeping track of states and transitions.

The **SemanticEngineFactory** is a simple class that follows the *factory design pattern*. It is used to create instances of the **SemanticEngine** class.

The **Evaluator** is an abstract class which defines an interface for evaluating certain objects. There are two types of evaluators. The **ValueEvaluator** simply evaluates simple calculations that may occur within a rule. The **EnvExpressionEvaluator** on the other hand evaluates expressions that result in a new *function* which may act as a store or environment. It implements for example the *function override expression* as encountered in section 3.3.2.

The **Matcher** class is responsible for object matching. The proof tree is represented by the **MatchingTree** class. Whenever there are multiple possible matches (non-determinism) it consults a **MatchingSelectionStrategy** object to choose a matching. The **MatchingSelectionStrategy** abstract class follows the *strategy design pattern*.

A **ResultConstructor** is responsible for building the result object for a given matching (i.e. a **MatchingTree** object). It constructs new objects and copies or references existing objects. It also has an implementation of the copy-algorithm described in section 5.4.

An object that implements a **TransitionObserver** can be hooked into a **SemanticEngine**. It observes all transitions made and can be used to update the user interface. This ensures a clean separation between the model and the view as described in the *model-view-controller* design pattern. The controller can be found in the user interface component.

5.3.4 Architecture of the User Interface

The user interface is built upon the JFace library, which results in a number of subclasses of classes in the JFace library. The architecture of the user-interface is therefore influenced by the architecture of JFace itself. We recommend to study JFace in order to understand the architecture description of the user interface.

The class diagram is figure 5.3 shows the structure of the user interface. The grey filled classes are from the JFace library. The **AbstractUIPlugin** is an abstract class which must be subclassed to create an eclipse plugin, the **Activator** is this subclass, it *activates* the plugin. The user interface consists of a *perspective*. A perspective is a set of *views* an optionally one or more *editors*. The **Perspective** class represents the *perspective* and has five views and one editor.

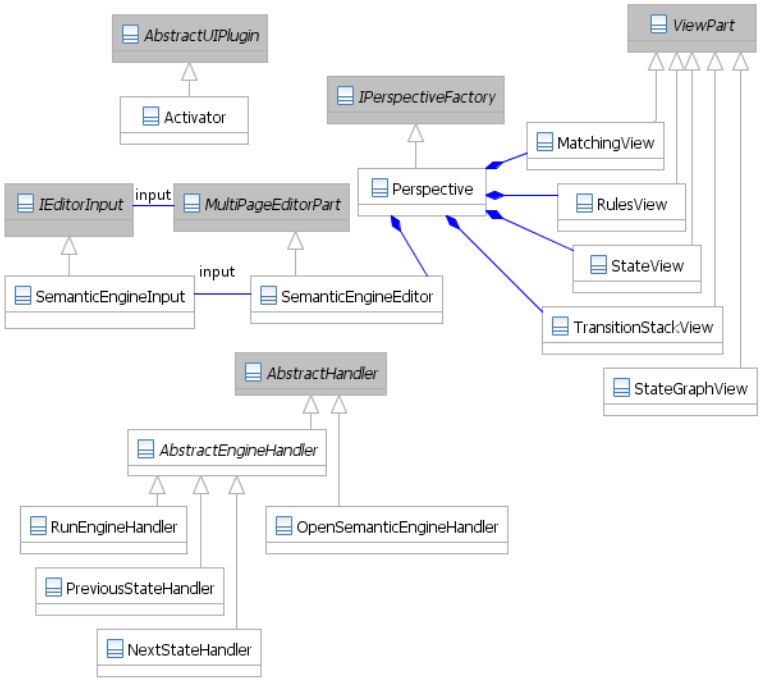


Figure 5.3: Class Diagram of the UI

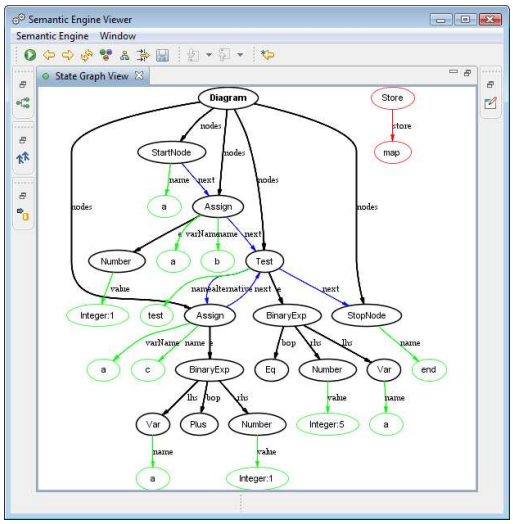


Figure 5.4: The state graph view

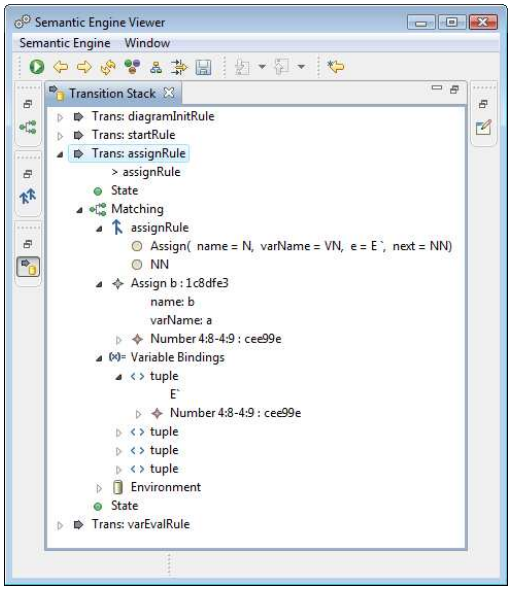


Figure 5.5: The transition stack view

The **MatchingView** visualizes the proof for a transition, the **RulesView** shows a list of rules and the **TransitionStack** view shows the sequence of transitions during simulation (see figure 5.5). The state is visualized as a graph in the **StateGraphView**, the Graphviz library is used to visualize the state (see figure 5.4). Figure 5.6 shows a overview screenshot and the decomposition of the *perspective* into different views.

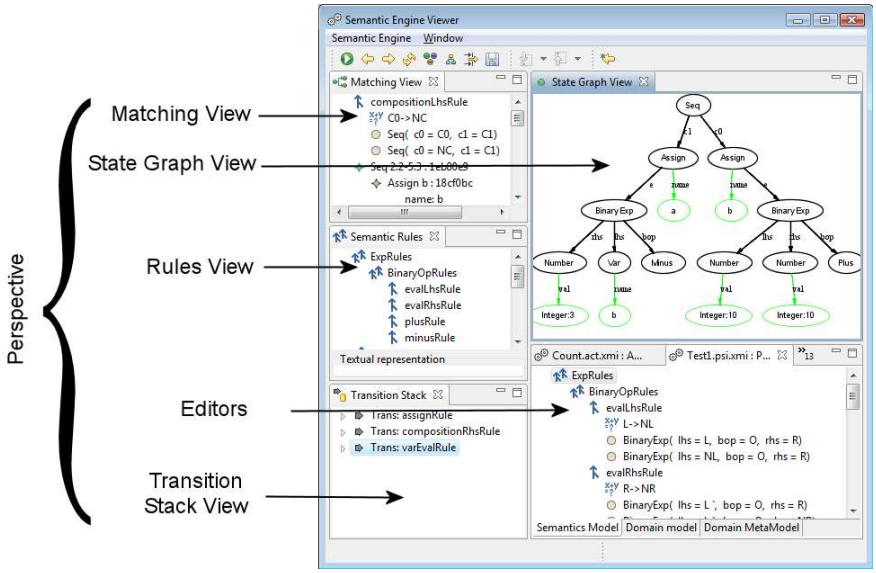


Figure 5.6: The views in the user interface

User events (occur when a user clicks a menu-item or toolbar-item) are handled

by subclasses of **AbstractHandler**. The **AbstractEngineHandler** class is a convenience class that has some useful methods for accessing the semantic engine.
The user interface has one editor, the **SemanticEngineEditor**. Multiple editors can be opened. Each editor relates to one **SemanticEngine**, which is encapsulated by an **SemanticEngineInput** object. The editor consists of multiple pages (that is why it is a subclass of **MultiPageEditorPart**), one page shows the current DSL, one page the current model and another page the current *SemLang* specification. Note that the data in the editor cannot be changed.

5.4 Quality attributes

This section discusses the quality attributes of the tool briefly. Quality attributes can be categorized in two parts, runtime quality attributes and non-runtime quality attributes. Most quality attributes that are discussed here are not measured by any means, mainly because the tool is a proof of concept. However, this section gives some insight into the quality aspects of the tool.
Runtime quality attributes can be measured when the tool is executed. It consists of sub qualities like performance, functionality and usability. The performance of the tool was not of high priority and there are several optimizations possible that decreases the memory usage of the tool and increases it performance. However, to get some insight into the performance of the tool a small performance test was performed. In this test we simulated several models of different modelling languages. During this simulation the number of states s was counted and the total elapsed time t was measured. These two numbers could be used to calculate the average time per state transition $(t/(s-1))$. The results are in table 5.1. To perform these tests one has to open the referred model file, language metamodel file and semantic description in the tool. See the disc for more info (appendix A)

Table 5.1: Performance measures

Model File	Language	#States	Time	Time / State
Factorial.ad	Activity Diagram	46	56.84 s	1.24 s
RecursiveFactorial.pkf	Plotkin Functional	94	74.66 s	0.79 s
Test1.pkf	Plotkin Functional	23	18.43 s	0.80 s
FibImperative.pim	Plotkin Imperative	50	38.42 s	0.77 s

The state transition takes a considerable amount of time because it also includes the visualization of the state. The visualization of the state involves applying a graph layout algorithm and a rendering process performed by Graphviz. The layout algorithm used is the one introduced by Sugiyama et al [Sug] and has a time complexity of $O(n)$. The transition algorithm itself uses the copy algorithm as explained in section . The copy algorithm itself is basically a breath-first-search algorithm and has therefore a worst case time complexity of $O(n + e)$ where n is the number of nodes and e is the number of edges in the graphs. The

time complexity of a single transition is not investigated, this is seen as future work.
The requirements of the tool ensured that the tool had some basic functionality. The functionality of the tool is therefore sufficient, however more functions can be added in the future. Again we remind you that this tool is a proof of concept. The usability of the tool is probably rather good because the tool is built upon the Eclipse framework and because we use Graphviz [gra] for state visualization. State visualization greatly increases the usability.
The non-runtime qualities deal with qualities that can be measured when the system does not execute. These qualities include modifiability, portability and testability. The modifiability of the tool is of course hard to measure. The tool has extensive *JavaDoc* documentation and this chapter gives a high level overview of the software. Therefore we ensured good modifiability.
The portability of the tool to other MDE frameworks is rather low; the tool is tightly bound to the EMF [EMF] framework and does not provide a framework abstraction layer. However, with model transformations one can transform a model to an EMF based model and still apply our approach.
There are no unit-tests or other explicit tests added to the source code. The tests were mainly performed by simulating a model and checking the output. It is difficult and time intensive to write unit tests as you have to check that a new state is exactly what is expected. Manual tests are easier because one can use the state view to check whether a new state is correct.

5.5 Conclusion

An implementation of the Semantic Language was needed in order to show a proof of concept. A tool was build to provide simulation and debugging functionality. This chapter explained the requirements of the tool. A high level architecture of both the *Semantic Engine* and the *User Interface* was given in order to understand the composition of the software.
The software architecture follows good practices in software engineering; design patterns are used and explained intensively. Object oriented code practices like encapsulation, abstraction, modularity and inheritance are heavily used. The clear separation between *User Interface* and *Semantic Engine* ensures high cohesion but low coupling.
The quality attributes of the tool are also briefly discussed in this chapter. A better measurement of those attributes is desirable. However, some critical remarks regarding the quality attributes are made. The tool is mainly a proof of concept and therefore there was no focus on runtime quality attributes.

6 Evaluation

6.1 Introduction

This chapter gives an evaluation of our approach and compares it to existing approaches. Firstly we cover the expressiveness of our approach; can *SemLang* be used to define the semantics for *any* DSL? Secondly we compare our approach to existing approaches which were covered in chapter 2.7. The advantages and disadvantages of the approaches are discussed.

6.2 Expressiveness

The expressiveness of a semantic framework in MDE says something about the power of that framework. A semantic framework is fully expressive if the semantics of any DSL can be described using that framework. A limited expressive semantic framework is a framework which can only be used for a small set of DSLs. It is of course difficult to measure the expressiveness of a framework because there are an infinite number of DSLs.
However, insight into the expressiveness of an approach can be gathered by means of a set of different DSLs. In chapter 3 we introduced different DSLs and extended *SemLang* with features which were needed to specify the semantics of each DSL. In the end of that chapter we added features to *SemLang* which makes *SemLang* useful for graph based DSLs like *Activity Diagrams* and *Petri Nets*.
Using these example DSLs we showed that *SemLang* is capable of defining the semantics for a broad range of DSLs. However, this does not prove that *SemLang* can be used for any DSL. In fact it is easy to create a DSL for which it is difficult to define the semantics using *SemLang*. Such a DSL is for example the *Production System* language defined by Rivera et al [RGdLV08]. The semantics for that language are defined using graph transformations and Maude in the same paper for comparison purposes.
We tried to use *SemLang* to define the semantics of the *Production System* language. However, this was not a straightforward process because *SemLang* cannot be easily used to define the semantics of that language. *SemLang* rules cannot contain complex graph patterns; they can only match one object and the direct neighbours of that object. The semantics of the *Production System* requires complex object matching.
We can conclude that *SemLang* is less expressive than graph transformations

or MDE with Maude. However, we could extend *SemLang* with more features to increase the expressiveness. This work is seen as future research.

6.3 Comparison to Existing Approaches

Some existing approaches to defining the semantics of models were covered in chapter 2.7. Each approach will be compared to our approach, disadvantages and advantages of the approaches will be discussed.

6.3.1 Model Transformations

When specifying the semantics of a modelling language with model transformations one develops a model transformation which is used to transform the model from state to state, thereby executing the model. An advantage of this approach is that it already builds upon existing technologies in MDE like QVT [OMG08].
A disadvantage of this approach is that it is difficult to use the model transformation specification for formal methods like model checking. It depends on the transformation language used whether rigorous mathematical techniques can be used on the specification. Currently, the most commonly used transformation languages lack formal semantics.
Another disadvantage is the lack of modularity when using model transformations. Firstly there is a one-on-one relationship between a transition in the LTS (in the semantic domain) and the transformation specification. With SOS there are always one or more rules responsible for a transition in the LTS. We are aware that a model transformation specification may consist of rule like structures, however, that depends on the transformation language used.
In fact a *SemLang* specification can also be seen as a model transformation specification, because it consists of some rules which are used to transform a model into a new model. However, the aim of *SemLang* was to define the semantics of models and is therefore a better candidate for specifying semantics than a transformation language which aims at pure model-to-model transformation.

6.3.2 Graph Transformations

Graph transformation is based on graph theory and has therefore a sound mathematical base. A graph transformation specification consists, just like SOS, of a set of rules which define the transitions in the LTS. Apart from these similarities between Graph Transformations and SOS there are also a lot of differences between the two approaches.
The first difference is between the rules. A transition in the LTS is proved by a set of rules in *SemLang*. With graph transformation there is a one-on-one correspondence between the rules and the transitions. This tends to make the rules with Graph Transformation large because each rule must describe more. However, with *SemLang* we tend to have more smaller rules.
An advantage of the smaller rules is that they are more modular. Some rules can be re-used by other rules. It may even be possible to use certain rules for multiple DSL's. This thesis was not focused on providing modularity. However, the *SemLang* rules are built upon the MSOS approach as explained by Mosses

[Mos99, Mos02, Mos04]. MSOS is an extended version of SOS with features that accommodate modularity.
Another advantage of *SemLang* is that there is a guiding principle for creating the rule. The guiding principle is basically: each structure should have one or more rules; there should be a transition condition for each sub-structure of that structure. This guiding principle is derived from linguistics: the semantics of a phrase is a composition of the semantics of the sub-phrases. We are not aware of any guiding principle for creating rules for graph transformations.
A related advantage is that SOS is related to the *interpreter design pattern* [GHJV00]. When designing an interpreter one creates an *interpret* function for each language structure. This is similar to the approach in SOS where each structure has one or more rules.
Maybe one of the most important disadvantages of graph transformation is its performance: rule matching is NP-complete. In SOS rule matching is easy because there is always a current node that has to be matched.
With graph transformations one is required to transform a model to a type graph before graph transformation can be used. *SemLang*, however, works directly with the metamodel. This can be seen as an advantage because the intermediate transformation to a type graph is not needed. However, this introduces high coupling between the metamodel and the semantic specification.
Graph transformations also has advantages over *SemLang*. It is very expressive and it has a sound theoretic basis. The biggest advantage is the graph awareness. SOS has difficulties when it is used with graphs, with *SemLang* we provide a partial solution. But this solution is not as expressive as the graph transformation approach. The comparison is summarized in table 6.1.

Table 6.1: Comparison of SemLang and Graph Transformations

Feature	**Graph Transformations**	**SemLang**
Rules	Single rule for a transition	Multiple rules for a transition
Modularity	May be problematic	MSOS aimed at modularity
Guiding Principle	No guiding principle to create rules	Based on principle of composition
Interpreter	Not easy to map to interpreters	Related to *interpreter design pattern*
Complexity	Rule matching hard: NP-complete	Rule matching easy
Conversion	Needs conversion to type graph	Works directly with the metamodel
Theory	Sound theoretic basis	Theoretic basis work in progress
Expressiveness	Expressive	Limited (future research)
Graph awareness	Good	Locally tree oriented

6.3.3 MDE with Maude

Maude is a very powerful rewriting engine and MDE with Maude seems therefore a fruitful approach. The biggest disadvantage of Maude is that it has its own model framework. It is therefore required that models and metamodels are transformed to Maude objects.

Another possible disadvantage of Maude is its complexity. It is required to learn Maude before one tries to specify the semantics of a modelling language. However, the team that uses MDE with Maude is currently working on better tool integration which will increase the usability.

6.3.4 Simulation in the Topcased Toolkit

The Topcased [CCG+08, VPF+06] approach is pragmatic and very efficient in terms of memory usage and runtime. However, the semantics is defined in the Java programming language which makes it less useful for applying rigorous mathematical techniques. Also the code that specifies the semantics is very hard to reuse for any other modelling language.

6.3.5 Semantics Anchoring

Some general disadvantages of Semantic Anchoring were already covered in section 2.7.6. However, no comparison with *SemLang* was made.
Again, a disadvantage of this approach is that it does not work directly with the metamodel of the DSL for which the semantics is defined. In fact this approach needs an additional *minimal* modelling language for which the semantics is formally defined. A model of a DSL is then transformed to a model of this minimal modelling language.

6.4 Conclusion

This chapter provided an evaluation of the *SemLang* language. The first conclusion that can be made is that the expressiveness of *SemLang* has its limits. It can be extended by adding more features. However, research is needed to find out which features are needed.
Another conclusion is that *SemLang* has several advantages over the other semantic approaches. *SemLang* is based on SOS and has therefore a good mathematical basis. The close coupling between the DSL metamodel and a *SemLang* specification is, in our eyes, a advantage because this coupling is unavoidable and even desirable. This close coupling also prevents to use of any intermediate language like we see with most other approaches.
Apart from having a theoretical basis, *SemLang* is also pragmatical. There is a guidance for creating the rules and the rules are generally small. However, we understand that knowledge of SOS and its compositional aspect is needed in order to define the semantics of a DSL.

7 Conclusions

7.1 Introduction

This last chapter concludes the thesis. The first section gives a summary of the thesis. The next section gives an answer to the main research question and all sub-questions. The last section discusses possible directions for future research.

7.2 Summary

A lot of research is done in the Model Driven Engineering (MDE) community. However, the focus is mostly on the static aspects of MDE. MDE is often used for defining Domain Specific Languages (DSLs). However, the specification of a DSL also requires that the semantics of the DSL are described in some (formal) way.
A semantic approach that seems suitable for defining the semantics of a DSL is Structural Operational Semantics (SOS) [Plo81]. SOS is a semantic framework in which the semantic domain is a Labelled Transition System (LTS). The semantics of a language is described using a set of rules. A single transition in the LTS can be proven by a so called upwardly-branching-tree which is built using the rules. However, there are several problems with applying SOS directly on a DSL. Therefore the main research question of this thesis is: *Can SOS be adapted and applied successfully on MDE?*.
Some other approaches for defining the semantics of modelling languages already existed. In all of these approaches there is a mapping from the modelling language to a semantic domain. This mapping is specified in different ways. Some approaches are very formal while other approaches are more pragmatical. None of these approaches use SOS as a framework for specifying semantics.
Chapter 3 proposes some changes to SOS in order to make it applicable in an MDE context. We propose a new language for defining semantics, called *SemLang*, and is itself specified as a DSL. *SemLang* is used to define the semantics for some DSLs which are similar to the languages in the paper by Plotkin [Plo81]. However, *SemLang* also supports the specification of the semantics of certain graph-based languages. This is an extension to the Plotkin style SOS, as it only supports tree-based states.
Chapter 4 provides a formal basis for *SemLang* based on formalizations of SOS [AFV01, MRG07, Mos04]. The semantic domain of *SemLang* is a labelled terminal transition system (LTTS) in which the states are extended models of the

DSL. *SemLang* models are essentially Transition System Specifications (TSSs) in which there is a mapping from a *SemLang* model to a LTTS (the semantic domain). The transitions in the LTTS can be proven by the construction of proof trees. The main difference between SOS and *SemLang* is that the terms in a rule have a different task. They either *match* or *construct* model objects.
The chapter does not formalize all constructs and features of *SemLang*. The formalization is kept to the essential parts to keep it understandable. The main goal of the formalization is to give a formal basis for *SemLang*. The focus of this thesis is on the pragmatical aspects of *SemLang*.
A tool capable of executing *SemLang* specifications is discussed in chapter 5. It provides simulation and debugging functionality. The requirements of the tool and a high level architecture are both explained. The tool consists of two loosely coupled modules, the *Semantic Engine* and the *User Interface*. The *Semantic Engine* is an implementation of the Semantic Language and a simulation framework. The *User Interface* is a layer on top of the *Semantic Engine*. It provides a graphical user interface which increases the usability of the tool.
Our approach is evaluated in chapter 6. *SemLang* is applied to several DSLs to demonstrate its expressiveness. However, the expressiveness of *SemLang* has its limits. It could, for example, not be applied to the *Production System* language defined by Rivera et al [RGdLV08]. The expressiveness can be increased by adding new features to *SemLang*. Our approach is also compared to the other approaches to specifying the semantics of modelling languages (as discussed in section 2.7). *SemLang* is based on SOS and had therefore a good mathematical basis and is also relatively more pragmatical.

7.3 Answers to the Research Questions

The main problem that needed to be solved was the incompatibility between MDE and SOS. Initially it was not clear how SOS could be adapted to make it useful for defining the semantics of DSLs. The main research question of the thesis was therefore:

Can SOS be adapted and applied successfully to MDE?

The answer to the main research question is that SOS in fact can be adapted to make it useful for defining the semantics of DSLs. The research question was divided into different sub-questions, which are listed and answered below.

MDE models are graph structures but SOS is based on trees. Can SOS be adapted in order to make it suitable for using it with graph structures?

The problem with graphs is that it introduces loops in the model. These loops may cause infinity copying of nodes. This was solved by introducing a copy algorithm which makes sure that nodes are not copied twice. The copy algorithm is basically a breath-first-search based copy algorithm for graphs.
Another feature that was added to deal with this problem is the use of reference meta-variables. The copy algorithm deals with those variables differently. This feature is not strictly needed but it increases the expressiveness of *SemLang*.
Another feature that was added is support for quantification over lists. Both existential and universal quantification is supported. This feature makes *SemLang* more useful for defining the semantics of graph based languages.

SOS is based on the abstract syntax description of languages, how can SOS be

adapted to deal with metamodels?

This question was not difficult to answer because metamodels and abstract syntax descriptions have a lot in common. The rules were changed in order to match and create objects instead of AST nodes. So called *terms* are used to match objects and create them.

Terms in rules The terms in the rules are changed. In SOS they consist of phrases of the DSL. However, a DSL in MDE does not require the syntax to be defined. The main elements of an MDE model are objects instead of sentences. The terms are changed to object patterns in order to match objects in a model, or construct new objects.

The use of the metamodel instead of the abstract syntax description actually enhances SOS because it supports polymorphism; rules can be created which match abstract classes.

The semantic domain of SOS is a labelled transition system in which the states are trees. How can we change the semantic domain in order to make it suitable for models?

In chapter 4 we choose for terminal labelled transition systems (LTTSs) as the semantic domain for *SemLang*. The states in the semantic domain of SOS are sentences of the abstract syntax of the DSL and therefore always tree structures. The states in the semantic domain *SemLang*, however, are extended models. These extended models can be represented as graph structures. This solves the incompatibility.
The answer to the main research question is that SOS can indeed by applied to MDE, this is shown by proof of concept and partial formalization. The proof of concept consists of the *SemLang* language and a tool which can simulate models using a *SemLang* specification. The tool was written using the Eclipse EMF [EMF] and RCP frameworks [RCP]. The architecture documentation can be read in chapter 5. The complete source-code can be found on the disc that comes with this thesis (seen appendix A).

7.3.1 Limitations

There are a number of limitations that can be found in our approach. However, some limitations can be overcome and can therefore be seen as future work. A list of limitations is given below.

- Expressiveness is limited

  As explained in chapter 6 the expressiveness of *SemLang* is limited. This can be increased by adding new features to the language. But it is not know which features must be added and how many; too many features would make *SemLang* complex and impractical.

- The language and tool is highly coupled to the Eclipse Modelling Framework

  It is difficult to create a modelling language without choosing one MDE approach. However, by choosing the EMF [EMF] framework we could only use *SemLang* to define the semantics of modelling language created with EMF. The tool itself is also highly coupled to the EMF framework.

- Modularity is an open issue

  Defining semantics of languages in a cross language modular fashion could enable reuse between different languages. *SemLang* however has no functionality to deal with this kind of modularity. A *SemLang* specification is always a single file and no other specifications can be included.

- No good insight into the efficiency and performance quality attributes

  The quality attributes of the tool were explained in chapter 5. The performance of the tool was only roughly measured. The measurements did not take the effect of the visualisation into account. The layout and visualization of the model is done in each transition and is therefore a major contributor to the runtime.

7.4 Future Work

This thesis provides an initial language based on SOS which can be used to define the semantics of a DSL. The *SemLang* language is a proof of concept and the tool is created to simulate models.
However, there are still open issues and future research directions. A list of possible future research directions is given in this section. First some possible extensions to *SemLang* are given. Secondly we look at some possible applications fields for *SemLang*.

7.4.1 Concurrency & Interactivity

SOS is really powerful in specifying the semantics for concurrent systems [AFV01, MRG07]. The *Semlang* language does not support concurrency and synchronization constructs. This can be achieved by introducing labels in the LTTS of the semantic domain.
The labels can also be used to support interactivity (as encountered in user-interface applications). The labels represent events that occur in the user interface. Together with this extension *SemLang* can be used to define the behaviour of user interfaces.

7.4.2 Rule Extensions

The current rules in *SemLang* consist of terms which either *match* or *create* objects. More constructs can be added to increase the expressive power of *SemLang*.
One way to improve the expressive power of *SemLang* is to add more list matching techniques. A good candidate extension can be found in the paper of Igarashi et al [IPW01]. In their paper they introduce a language *Featherweight Java*, which is a subset of Java. Special list matching techniques are used in the rules in the SOS specification of Featherweight Java.
SemLang already has support for simple object expressions similar like those found in OCL. However, a possible extension is to adopt OCL [OMG03b] as an expression language in *SemLang*. OCL could be used to extend the object matching and object creation capabilities by using OCL path expressions.

OCL was currently not used because it does not allow binding objects to meta-variables, therefore decreasing the object matching power of *SemLang*.

7.4.3 Potential Applications

There are a number of potential applications for *SemLang*. These application domains are similar to the application domains of SOS. A limited list of applications is listed below.

Simulation

The *SemLang* semantics of a DSL can be used to simulate models of that DSL. The purpose of simulation is to gain understanding of a system without manipulating the real system. Therefore simulation is always performed on a model of the system. This is how Combemale et al [CCG+08] define simulation. They also subdivide simulation in three steps: *Workload generation (trace building)*, *Model execution* and *Result analysis*. It is possible to combine two of these steps, or even three of them.
Simulation is, in fact, already supported by the tool written for this research. The tool allows simulating a model for which the semantics is defined. The *workload (execution trace)* is generated on the fly; the user can choose a transition for any non-deterministic step. The *result* is also generated on the fly; the states can be inspected and the proof for the transitions can also be inspected while simulating the model. However, the tool is only a proof of concept. The tool could be integrated with other tools. The usability could be improved, for example by adding a *SemLang* editor.

Proofs and Reasoning

The semantic domain of *SemLang* is formally specified in chapter 4. A big advantage of a well known semantic domain is that rigorous mathematical techniques can be used. In SOS, for example, one can use Structural Induction (SI) to prove certain statements over a program or over the SOS rules themselves. SI fits SOS particularly well because the states in SOS are always trees; this is because SI is a recursive method which works well on recursively defined structures like lists and trees. In order to make SI useful for *SemLang* one must define the states of *SemLang* in a recursive manner. A possible solution to this problem is to use induction over the size of the graph. This work is seen as future research.

Generation of Interpreters

Interpreters (and compilers) must follow the semantics of a language. An interpreter has a bug if it does not follow the behaviour as specified in the (formal) semantics. It looks promising to generate interpreters from the semantics specification of a language. In this way bugs can be avoided and the time to implement an interpreter can be decreased dramatically. In fact this topic is investigated by different researchers [Mic93, Poe94, Bru99].
SOS is a good candidate for generating interpreters because it is related to the Interpreter Design Pattern [GHJV00, pages 243–255]. In the Interpreter Design Pattern an *Interpret* operation is specified for each abstract syntax class. The

code in an *InterpretX* operation for an abstract syntax structure X defines the semantics for that structure. An *Interpret* operation may call other *Interpret* operations in order the interpret the sub-sentences. This technique is similar to the way the rules in SOS are specified; a rule is defined for each metaclass and a transition condition is added for each sub-structure.

References

[AFV01] Luca Aceto, Willem Jan (Wan) Fokkink, and Chris Verhoef. Structural operational semantics. In Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, editors, *Handbook of Process Algebra, Chapter 3*, pages 197–292. Elsevier Science, Dordrecht, The Netherlands, 2001.

[BÓ4] Jean Bézivin. In search of a basic principle for model-driven engineering. *CEPIS, UPGRADE, The European Journal for the Informatics Professional V 2*, pages 21–24, 2004.

[BH02] Luciano Baresi and Reiko Heckel. Tutorial introduction to graph transformation: A software engineering perspective. In *Proc. Graph Transformation - First International Conf.*, Barcelona, Spain, 2002.

[BHJ+05] Jean Bézivin Guillaume Hillairet, Frédéric Jouault, Ivan Kurtev, and William Piers. Bridging the ms/dsl tools and the eclipse modeling framework. *Proceedings of the International Workshop on Software Factories at OOPSLA, San Diego*, 2005.

[Bru99] Kim B. Bruce. Formal semantics and interpreters in a principles of programming languages course. In Daniel Joyce, editor, *Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*, volume 31.1 of *SIGCSE Bulletin*, pages 331–335, N. Y., March 24–28 1999. ACM Press.

[CCG+08] Benoît Combemale, Xavier Crégut, Jean-Pierre Giacometti, Pierre Michel, and Marc Pantel. Introducing simulation and model animation in the MDE topcased toolkit, 2008.

[CSAJ05] Kai Chen, Janos Sztipanovits, Sherif Abdelwalhed, and Ethan Jackson. Semantic anchoring with model transformations. *Springer LNCS*, 2005.

[ECO] *Eclipse Ecore*. `http://www.eclipse.org/modeling/emf/`.

[EMF] *Eclipse Modeling Framework (EMF)*. `http://www.eclipse.org/modeling/emf`.

[GHJV00] Gamma, Helm, Johnson, and Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 2000.

[GME] *Generic Modeling Environment (GME)*. `http://www.isis.vanderbilt.edu/Projects/gme`.

[GMF] *Eclipse Graphical Modeling Framework (GMF)*. `http://www.eclipse.org/gmf`.

[gra] *Graphviz - Graph Visualization Software*. `http://www.graphviz.org/`.

[GRe] *Graph Rewriting and Transformation (GReAT)*. `http://www.isis.vanderbilt.edu/Projects/mobies`.

[GSCK04] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories*. Wiley, 2004.

[Hec06] Reiko Heckel. Graph transformation in a nutshell. *Electr. Notes Theor. Comput. Sci*, 148(1):187–198, 2006.

[Hen90] Matthew Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*. Wiley, University of Sussex, United Kingdom, 1990.

[Hoa69] Hoare. An axiomatic basis for computer programming. *CACM: Communications of the ACM*, 12, 1969.

[HR04] David Harel and Bernhard Rumpe. Meaningful modeling: What's the semantic of "semantics"? *Springer LNCS*, 2004.

[Hud98] Paul Hudak. Modular domain specific languages and tools. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[IPW01] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *TOPLAS*, 23(3):396–459, May 2001.

[JB06] Frédéric Jouault and Jean Bézivin. Km3: a dsl for metamodel specification. *Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (LNCS 4037)*, pages 171–185, 2006.

[JBK06] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. Tcs: a dsl for the specification of textual concrete syntaxes in model engineering. *Proceedings of the fifth international conference on Generative programming and Component Engineering*, 2006.

[JFa] *JFace*. `http://wiki.eclipse.org/index.php/JFace`.

[Ken02] Stuart Kent. Model driven engineering. *Proceedings of ACME/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (Modules/UML '06)*, pages 286–298, 2002.

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling, Enabling Full Code Generation*. John Wiley & Sons, Inc., 2008.

[Kur05] Ivan Kurtev. *Adaptability of Model Transformations*. University of Twente, Netherlands, 2005.

[MHS05] Mernik, Heering, and Sloane. When and how to develop domain-specific languages. *CSURV: Computing Surveys*, 37, 2005.

[Mic93] Greg Michaelson. *Interpreter Prototypes from Formal Language Definitions*. PhD thesis, Heriot-Watt University, Department of Computing & Electrical Engineering, Riccarton, Scotland, EH14 4AS, 1993.

[Mil90] Robin Milner. Operational and algebraic semantics of concurrent processes. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 1201–1242. MIT Press/Elsevier, 1990.

[Mos99] Peter D. Mosses. Foundations of modular sos. Technical report, University of Aarhus, Denmark, 1999.

[Mos02] Peter D. Mosses. *Pragmatics of Modular SOS*. Springer LNCS Vol 2422, University of Aarhus, Denmark, 2002.

[Mos04] Peter D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming 60-61*, pages 195–228, 2004.

[Mos06] Peter D. Mosses. Formal semantics of programming languages, an overview. *Electronic Notes in Theoretical Computer Science 148*, pages 41–73, 2006.

[MRG07] MohammadReza Mousavi, Michel A. Reniers, and Jan Friso Groote. Sos formats and meta-theory: 20 years after. *TCS: Theoretical Computer Science*, 373(3):238 – 272, 2007. Structural Operational Semantics.

[OMG] OMG. *Unified Modeling Language^{TM}(UML®)*. `http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML`.

[OMG02] OMG. Meta object facility (mof) specification, version 1.4. *OMG Document*, 2002.

[OMG03a] OMG. Meta object facility (mof) 2.0 core specication. *OMG Document*, 2003.

[OMG03b] OMG. Uml ocl 2.0 specification. *OMG Document*, 2003.

[OMG08] OMG. *Meta Object Facility (MOF) 2.0, Query/View/Transformation specification, V1.0*. `http://www.omg.org/docs/formal/08-04-03.pdf`, 2008.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical report, University of Aarhus, Denmark, 1981.

[Poe94] A. Poetzsch-Heffter. Developing efficient interpreters based on formal language specifications. In P. Fritzson, editor, *Compiler Construction*, volume 786 of *LNCS*, pages 233–247. Springer, 1994.

[RCP] *Eclipse Rich Client Platform (RCP)*. `http://wiki.eclipse.org/index.php/Rich_Client_Platform`.

[RGdLV08] José Eduardo Rivera, Esther Guerra, Juan de Lara, and Antonio Vallecillo. Analyzing rule-based behavioral semantics of visual modeling languages with maude. In Dragan Gasevic, Ralf Lämmel, and Eric Van Wyk, editors, *SLE*, volume 5452 of *Lecture Notes in Computer Science*, pages 54–73. Springer, 2008.

[RRDV07] R. Romero, J.E. Rivera, F. Duran, and A. Vallecillo. Formal and tool support for model driven engineering with maude. *Journal of Object Technology*, 2007.

[RV07] J.E. Rivera and A. Vallecillo. Adding behavioral semantics to models. *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*, 2007.

[Sch86] David A. Schmidt. *Denotational Semantics – A Methodology for Language Development*. Wm. C. Brown Publishers, 1986.

[SF07] Markus Scheidgen and Joachim Fischer. Human comprehensible and machine processable specifications of operational semantics. *ECMADA-FA, LNCS 4530*, 2007.

[Sol00] R. Soley. Model driven architecture. *OMG Document*, 2000.

[SS71] Dana Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. pages 19–46. April 1971.

[Sug]

[SW08] Daniel A. Sadiley and Huido Wachmuth. Prototyping visual interpreters and debuggers for domain-specific modeling languages. *ECMADA-FA, LNCS 5095*, 2008.

[SWT] *SWT: The Standard Widget Toolkit*. `http://www.eclipse.org/swt`.

[TCCG07] Xavier Thirioux, Benoît Combemale, Xavier Crégut, and Pierre-Lo "ic Garoche. A framework to formalise the mde foundations. *International Workshop on Towers of Models (TOWERS)*, 2007.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain specific languages: An annotated bibliography. *ACM SIGPLAN Notices, V 25, issue 6*, pages 26–36, 2000.

[VPF+06] F. Vernadat, Christian Percebois, Patrick Farail, R. Vingerhoeds, Alain Rossignol, Jean-Pierre Talpin, and David Chemouil. The TOPCASED project - A toolkit in OPen-source for critical applications and system development. In *Data Systems In Aerospace (DASIA), Berlin, Germany*, page (electronic medium), `http://www.esa.int/publications`, May 2006. European Space Agency (ESA Publications).

A

Appendix A: Compact Disc

This thesis is accompanied with a compact disc that contains software, source code, documentation and other files:

1. **Thesis in Digital Form**

   Path /thesis

   Description This thesis in digital pdf format. The thesis includes hyperlinks that allow easy navigation between chapters.

   Path /thesis/latex/

   Description Contains the latex source code of the thesis

2. **Presentation in Digital Form**

   Path /presentation/

   Description Contains the presentation(s) done for this thesis in pdf form

3. **Tool Binaries and Source Code**

   Path /tool

   Description Contains the binaries of the tool for the Windows platform

   Path /code/tool

   Description Contains the sourcecode of the tool. Open this folder in eclipse as an eclipse workspace

   Path /code/tool/SemanticLanguage

   Description Contains the metamodel and concrete syntax of the semantic language defined in km3

   Path /code/tool/SemanticEngine

   Description Contains the sourcecode for the semantic engine

   Path /code/tool/SemanticEnginePlugin

   Description Contains the sourcecode for the user interface of the semantic engine

4. **Example DSLs and their Semantics**

   Path /code/dsls

   Description Contains a list of DSLs for which the semantics are defined.

   Path /code/dsls/*/Metamodel

   Description Contains the metamodel of a dsl

   Path /code/dsls/*/Samples

   Description Contains one or more sample models of the dsl

   Path /code/dsls/*/Semantics

   Description Contains the semantics specification in the Semantic Language of the dsl