Bruno Fontes Barroca

Mestre em Engenharia Informática

Analysable Software Language Translations

Dissertação para obtenção do Grau de Doutor em
Engenharia Informática

Orientador : Dr. Vasco Miguel Moreira do Amaral,
Prof. Auxiliar, Universidade Nova de Lisboa

Júri:

Presidente:	Prof. Doutor Luis Caires Marques da Costa Caires
Arguentes:	Prof. Doutor Juan de Lara
	Prof. Doutor João Carlos Pascoal Faria
Vogais:	Prof. Doutor Hans Vangheluwe
	Prof. Doutor Alberto Manuel Rodrigues da Silva
	Prof. Doutora Carla Maria Gonçalves Ferreira
	Prof. Doutor Vasco Miguel Moreira do Amaral

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Janeiro, 2013

Analysable Software Language Translations

Acknowledgements

Over the past five years, I have received support and encouragement from a great number of individuals. In particular, Professor Vasco Amaral has been a mentor, and a friend. With him, I have learned the complexity problems of software engineering, and how models and modeling languages can be a powerful solution to solve them. His guidance has made this a thoughtful and rewarding journey. Also, Dr. Levi Lúcio gave me the pleasure to work with him during the most important phase in my PhD research work. With him, I have learned how math and set theory can be used to elegantly describe and study both the problems and solutions we may find in computer science.

I would like to thank the SMV group at the Université de Genève, in particular to Dr. Matteo Risoldi, and Professor Didier Buchs for their teachings on modeling, formalisms and verification techniques. I would also like to thank the members of the Technical Advisory Committee of my PhD thesis for their patience and valuable insight about this research work: Professors Carla Ferreira, and Hans Vangueluwe.

Finally, I would like to thank all the people that worked with me at CITI, and gave valuable contributions to my PhD research work: Vasco Sousa, Roberto Félix, and Cláudio Gomes.

A special thank-you goes to the Centro de Informática e Tecnologias de Informação (CITI), Portugal, the Department of Informatics of the Universidade Nova de Lisboa, and the Fundação para a Ciência e a Tecnologia, Portugal that through the grant SFRH/BD/38123/2007 supported me financially and organizationally over all this time.

Abstract

The most difficult tasks in the Software Language Engineering (SLE) process, are the design of the semantics of a Domain Specific Modeling Language (DSML), its implementation (typically in a form of a compiler), and also its verification and validation. On the one hand, the choice of the appropriate level of abstraction when designing a DSML's semantics, affects directly its usability, and the potential for its analysis. On the other hand, in practice, not only the compiler's implementation, but also its verification and validation are performed manually, while having as reference the DSML's semantic models.

The challenge of this research work is to apply a complete model driven software development approach in the tasks of designing a DSML's semantics, implementing, verifying and validating DSMLs' compilers. This involves the choice of the most appropriate abstraction levels, and the design and development of adequate tools to support SLE practitioners on these tasks.

This thesis reports: *i)* the design and implementation of formal languages (and associated tools) to support the task of DSML's semantics design (i.e., DSLTrans and SOS); *ii)* the automatic generation of DSMLs' compilers based on translation specifications; and *iii)* automated validation of DSMLs' semantic designs based on the analysis of translation specifications. Finally, the approach presented in this thesis is illustrated with the design and implementation of a real life DSML.

Keywords: Translations, Model Transformations, Structured Operational Semantics (SOS), Model Transformation Analysis

Resumo

As tarefas mais dificeis de executar no decorrer do processo de Engenharia de Linguagens de Software (ELS), são o desenho da semântica de uma Linguagem de Modelação de Domínio Específico (LMDE), a sua implementação, e também a sua verificação e validação. Por um lado, a escolha do nível de abstração no desenho da semântica da LMDE, afecta directamente a sua usabilidade, e o seu potencial para ser analisada. Por outro lado, na prática, não só a implementação do seu compilador, mas também a sua verificação e validação são executadas de forma manual, tendo como referência modelos semânticos das LMDEs.

O desafio deste trabalho de investigação é aplicar de forma completa uma abordagem de desenvolvimento de software baseada em modelos, nas tarefas de desenho, implementação, verificação e validação de compiladores de LMDEs. Isto envolve a escolha dos níveis mais apropriados de abstração, e o desenho de desenvolvimento de ferramentas adequadas para suportar a prática da ELS nestas tarefas em particular. Esta tese reporta: *i)* o desenho e implementação de linguagens formais (e ferramentas associadas) para suportar a tarefa de desenho de semânticas de LMDEs (DSLTrans e SOS); *ii)* a geração automática de compiladores de LMDEs com base em especificações de tradução; e *iii)* a validação automática de desenhos de semânticas de LMDEs através da análise de especificações de tradução. Finalmente, a abordagem apresentada nesta tese é validada com o desenho e implementação de uma LMDE usada na vida real.

Palavras-chave: Traduções, Modelos de Tradução, Semântica Operacional Estruturada (SOS), Análise de Modelos de Transformação

Contents

List of Figures

List of Tables

1

Introduction

An increasing number of people, in all professional fields and knowledge areas, rely on software systems to perform their daily routines and responsibilities. The immersion of computer technology in a wide range of domains, leads to a situation where the users' needs become demanding and complex; besides that, the quality of the users' interaction with this kind of technology is becoming of utmost importance. Consequently, the development of successful software systems themselves becomes increasingly more complex.

Software engineers need to cope with the increasing complexity on developing, maintaining and evolving software solutions, which are consequently getting increasingly costly and error prone. If we look carefully to the sources of this complexity, we conclude that there is a class of complex problems from a given domain, which are sometimes very complex to understand and learn, such as the rules and technical jargon found in domains like the Physics Computing, Financial Domain, among others. In other words, the essential complexity from a given domain is definitely unavoidable [Jr.87]. However, besides having to provide solutions that effectively solve a given class of essential problems from a given domain, the Software Engineer has also to deal with the accidental complexity of the used computer technology—e.g., the use of low level abstraction programming languages, while integrating a wide plethora of different tools and libraries.

A promising 'divide-and-conquer' idea to break down the increasing complexity in software engineering is the concept of multi-paradigm modeling (MPM).

The base idea of MPM [dLVA04] is to have multiple viewpoints to look at any existing (or intended) object in multiple perspectives simultaneously, without the existence of logical contradictions between those viewpoints. Instead of trying to express every system-related object using a general purpose modeling language (GPML), MPM realizes each perspective (or viewpoint) by means of domain specific modeling languages (DSMLs) [KT08]. DSMLs provide the end-users —i.e., the modelers involved in a domain specific modeling (DSM) activity — with a pragmatic and usable way to understand and write their descriptions. Each DSML is supposed to capture the occurrence of the reusable programming patterns of the given application domain where it is defined, while using a restricted terminology limited to the perspective (or viewpoint) of its application domain. Moreover, DSML solutions provide to the end-users in a given application domain, a pragmatic way to automatically apply these reusable programming patterns in a controlled manner.

However, building a DSML from scratch is far from trivial. One of the most crucial (and complex) steps while building a new software language, is to assign its semantics by mapping its syntax onto a computational semantic domain. This semantic assignment can be done formally in a platform independent way by means of a set of rewrite rules which describes how each construct of the language can be rewritten by an abstract machine. Each rewrite step usually represents a computation step—i.e., the typical computational semantics of a software language. This formal model of the software language's semantics (also called as Operational Semantics) can then be used as a reference to build a compiler for the language. A compiler is a program that basically translates the high level constructs of the language onto constructs of a low level programming language (supported by a particular platform) in such a way that they mimic all of the computation steps described formally in the provided formal Operational Semantics model. The hardest problem for a Language Engineer is then to prove that the implementation of this compiler is indeed correct according to the language's semantics model [GS98]. Typically this proof is complex and requires extensive testing phases, that usually hampers the speed of the language evolution. This is due to the difficulty of finding pertinent tests to test the compiler, since a software language usually produces an infinite amount of possible sentences, and each sentence might produce an infinite amount of computation steps.

Yet, we are witnessing the adoption of DSMLs in the software industry: companies are building languages for their domains, by using Language Workbenches (e.g., MPS [1], MetaEdit+ [2], GMF [3], DSLTools [4], etc.). These Language Workbenches apply the principles of using models (i.e., specifications at an appropriate level of abstraction) during software engineering—also known as Model Driven Development (MDD)— by enabling the specification of both syntactic and semantic models of the languages. Typically, on these Workbenches, the model of a DSML's syntax can be defined based on a meta-modelling language such as MOF [5], and its computational semantics can be defined by means of a model transformation specification which in turn may be expressed in a Model Transformation Language such as QVT [6]. The main advantages of using these Language Workbenches lies in the fact that they enable the automatic generation of any DSML given that we provide both its syntactic and semantic models. For instance, based on the defined syntactic model of the DSML, it is possible to automatically generate a prototype of the DSML's editor. Moreover, if we specify the semantic domain of a DSML by means of model transformations, we can automatically generate an interpreter or a compiler for that DSML. The systematic use of metamodels and model transformations is in the heart of MDD approaches, where software development's complexity is dealt in a systematic way, by its modularization into several levels of abstraction, and the definition of automated translations between these levels, each of them having their own rules and restrictions formalized in a particular DSML.

1.1 Research Question

During the PhD research work, we found that there were still no adequate methodologies (and supporting tools) to ease the development (and maintenance) of DSMLs with a reasonable level of confidence and guarantees that they were correctly implemented. These guarantees are required not only in critical domains (such as avionics or automotive), but also in conventional domains (such as software game development). Moreover, the lack of these guarantees may be hampering further adoption of DSMLs by the industry, and consequently hamper the solution for the increasing complexity in software engineering.

[1] http://www.jetbrains.com/mps/
[2] http://www.metacase.com/mep/
[3] http://www.eclipse.org/modeling/gmp/
[4] http://www.microsoft.com/en-us/download/details.aspx?id=2379
[5] http://www.omg.org/mof/
[6] http://www.omg.org/cgi-bin/doc?ptc/2007-07-07

The fundamental research question in this PhD research work, can be stated as:

is it possible to effectively develop a DSML's compiler (to an arbitrary computational platform), while bringing with it more correctness guarantees than extensive testing?

Following the principles of MDD, we concluded that a DSML's compiler is in fact a translation, and that the translations could be specified using the terminology of a model transformation language and executed by means of a generic model transformation engine. In fact, this generic model transformation engine acts as a meta-compiler, since it is able to produce a software language compiler from an existing translation specification. Each translation specification describes how the syntactic constructs of the source language are mapped into the syntactic constructs of the target language of the translation. Therefore, in principle, it should be possible to also analyse these specifications in order to validate them.

1.2 Challenges

The first challenge we stepped into, while tackling our research question, was to find the appropriate model transformation languages (MTLs) in order to provide the quality properties required for any DSML compiler, such as compiler's termination, confluence, and analysability. For instance, one characteristic of the existing MTLs is that they were designed for other purposes besides building compilers—they also may be used for refining models, synchronize them, or manage their versions. Even in the SLE, these MTLs can be used in different ways such as executing, simulating, verifying, or even animating models, typically by specifying *inplace* [7] model transformations as definitions of the operational semantics of the DSML in which these models are expressed. Moreover, we can also build DSML compilers with these MTLs, by specifying *outplace* model transformations as translations specifications that, when run, are able to automatically translate models expressed in a source language into other models expressed in a target language. This multi-purpose characteristic appears to have a negative impact on the properties that we can analyse in a model transformation expressed

[7] When run, an *inplace* model transformation transforms a given model M and modifies it, whereas an *outplace* model transformation takes as input a given model M and produces another model M'.

in such kind of MTLs. In general, the more expressive an MTL is (e.g., if it can express recursion), the more difficult it is to analyse their transformation specifications in a reasonable way, and in some cases reach the limits of undecidability. In other words, if we take a model expressed in a given modeling language, we may not find a generic function that takes an infinite sized analysis space and reduce it into a finite one—of course this produces a great impact in the properties that we can observe (or prove) in that model, as for instance the termination property [Plu98]. The main reason for this is that this generic reduction function is often associated with the semantic complexity of the modeling language (i.e., the semantics of the provided entities and operations of the modeling language).

Intuitively, the simpler the language is, the easiest it is to analyse and assure its quality. An example of such MTLs is EMF Tiger [BET08]. While using EMF Tiger, the language engineer can rely on certain properties proved by construction—such as termination or confluence—but only for a particular kind of model transformation patterns—in the general use of EMF Tiger, those properties are not guaranteed.

During the PhD research work, we found that there are many ways of validating model transformation specifications. Most of them rely on testing—i.e., executing a given model transformation specification with a concrete source model, and then evaluate the results. Again, this implies extensive testing and its coverage is not complete [BDtmM+06]. Therefore, we explored alternative approaches to validate model transformation specifications. One of such ways is to symbolically execute the transformation specification, and then evaluate the correspondences between symbolic patterns from source language, with symbolic patterns from the target language.

However, we soon realized that the high expressiveness in the existing MTLs, was hampering the exploration of the proposed approach. In fact, if we analyze a transformation specification expressed in one of the existing MTLs by means of a symbolic execution, we might stumble into an infinite symbolic space, which may be impossible to lead into a valid conclusion. Therefore, we concluded that we had to first design and build a completely new MTL with less expressive power than the others, but still able to be used to specify and build (or prototype) a DSML's compiler.

The main challenge was then to design a syntax-to-syntax model transformation language (MTL) to specify translations that could be automatically validated. Of course, we had to ensure that the resulting MTL is still usefull in in an SLE context—i.e., it is expressive enough to automatically derive compilers for any

(realistic) DSML, from those translation specifications. Notice that these translation specifications are expressed as syntax-to-syntax model transformations between two software languages: the source DSML, and the target language, which is usually a programming language. Not restricted to compilations that are particularly intended for execution, these translations can also enable the reuse of the capabilities offered by different kinds of computational platforms, such as efficient analysis algorithms and data structures, simulation and visualization capabilities, and so on.

1.3 Research Topics

In order to address the above referred challenges, we had to focus in three main research topics. The first, was the existing research on **modeling language's design**, in what matters to the expressiveness of DSMLs. Here, we focused not only in the cognitive aspects of the syntax of DSMLs, which produces a significant impact in their usability, but also on the semantic aspects, which produces a significant impact in the tractability of the existing analysis algorithms (i.e., DSML's analysability). The second, can be considered as derived from the first one, was the existing research on **formal models and formal languages to express language's semantics**. Here, we focused not only on languages to express syntax-to-syntax translations (i.e., perhaps the most intuitive use of model transformation languages), but also on languages to express operational (small step) semantic definitions. Finally, the third one, was the existing research on **verification of model transformations**. Here we focused on which kind of verification techniques that could be used in order to validate model transformations specifications, and verify compiler implementations based on those specifications.

1.4 Contribution Overview

During this PhD research work, we designed and built **a new language called DSLTrans** [BLA+10] and its associated tools (editors and execution engine). In order to avoid possible infinite symbolic spaces, we had to make sure that all of the expressible translations in DSLTrans were **terminating and confluent**: we achieved this by restricting DSLTrans' expressiveness (such as avoiding recursion), while assuring that DSLTrans was still useful in an SLE context. These properties were determinant to provide to the DSLTrans' translations the **ability to be analyzed** due to the finite size of the resulting translation's symbolic

execution space.

The resulting symbolic space for each translation specification, can be used to search for intended (or non-intended) correspondences between symbolic patterns from both source and target languages. This analysis can determine what are all the possible relations between syntactic structures expressed in the source DSML, and what are their translated versions expressed on the target language. Therefore, in [LBA10], we developed a **verification method** (and its associated checker) so that we are able to check, on any given model transformation expressed in DSLTrans, that a given **correspondence between syntactic structures** of a source and a target language holds, or not. Furthermore, this verification method involved the design of a small language to express these correspondences, and pass them to the checker.

In order to automate the validation process of a language translation, we had then to build up a way to generate an oracle that could automatically determine the validity of every existing correspondence in the whole symbolic execution space of that translation. In other words, when generated this oracle is supposed to be able to automatically decide if both the source and target models on an existing correspondence have the same meaning. In order to formally describe this decision procedure, we had to provide a definition of a **notion of semantic equivalence between arbitrary source and target languages**, namely the **Bisimulation Equivalence** [Par81]. This notion uses the operational semantic definitions of both the source and target DSMLs in order to be able to conclude that two sentences expressed in each of the languages have the same computational meaning. In order to specify DSML's operational semantic definitions, we designed and implemented a **new language called SOS**, which enables the language engineer to specify a DSML's operational semantics by means of an **algebraic semantic domain**, and a set of step rules that are able to mimic the computation steps of evaluating a DSML's sentence, in a platform independent fashion.

Based on the notion of Bisimulation Equivalence, we then developed a **verification method (and associated checker)** that is able to use the above mentioned SOS specifications of both source and target languages in order to automatically validate any given translation expressed in DSLTrans. This is done by automatically verifying the Bisimulation Equivalence relation between the source and target patterns on every existing correspondences found in the whole symbolic execution space of DSLTrans translation under validation.

For the sake of clarity, we provide **mathematical formalizations** of all of the developed languages and verification methods based on graph theory and set

theory. We used these formalizations to define how the symbolic execution space of an arbitrary translation can be explored in order to validate it in w.r.t. the defined SOS semantic definitions of both source and target languages involved in that translation. In other words, a given translation might be semantically wrong if a model in the source language do not have exactly the same meaning after being translated to the target language — intuitively, a translation is valid if it preserves the semantics of all models expressible in the source language of the translation. Therefore, we provide a formal definition of what is **semantic preservation**, and an algorithm that is able to check this preservation on a given translation.

Finally, for the sake of soundness, we **illustrate our approach** with the analysis of two translations expressed in DSLTrans. In order to ease the comprehension of the approach, we selected Petri Nets to be the target language of both the translations. We define the operational semantics of the two source languages (i.e., State Machines Language and the Role Playing Games Language) and also of the target language (i.e., the Petri Nets Language) with respect to **reachability** properties. This means that once these two translations are proved to be correct (using our approach), then we know that at least all the possible reachability properties will be preserved during the translation. This analysis capability on meta-compilers establishes an **analysable bridge**, hence **promoting interoperability** between arbitrary software languages, where quality properties (such as reachability, safety, etc.) can be effectively analysed using adequate languages and their respective engines.

1.5 Structure of this Thesis

In Chapter 2, we describe the context of this research work, and present what are the main research trends and challenges related to the work of this thesis. In Chapter 3, we give a theoretical overview of the approach while introducing an illustrating example that will be used in the following two Chapters. In Chapter 4 we present the formal definitions of models, languages and also their syntactic and semantic models. In Chapter 5, we show how to mechanically validate translations by either checking properties or using operational semantic definitions as oracles. In Chapter 6, we present a real life case study and discuss the results of this approach. Finally, in Chapter 7 we conclude and present future evolution on the research on this subject, and possible impact of the contributions presented on this thesis.

2

Background

In this chapter, we introduce the basic notions and concepts that will be used throughout this thesis. In order to fully understand the work in this thesis, one must first understand the notions of models, and the notion of languages to express those models. We will also introduce concepts that are involved when we study and design any kind of language, namely its syntax and its semantics. Moreover, whenever we talk about software languages in this thesis, it will be important to understand the systematic approach to engineer software languages. Therefore, we discuss the most suitable language development processes, and what are the language models involved in those processes. Finally, we drill down into the notion of model transformations, that we will use as a language model, in particular to *(i)* design and specify a language semantics; *(ii)* implement its execution engine; and *(iii)* validate this implementation w.r.t. the designed language model.

2.1 Models and Languages

Despite the fact that the use of models in software engineering is starting to gain momentum as a valuable solution to deal with its complexity, the notion of model and its use in engineering (not only in software engineering) is indeed very old [Fav04]. However, it might be difficult to agree a common definition of model with other fields of study (e.g., business, mathematicians, etc.), since in

software engineering the word model usually refers to an artifact, formulated in a modeling language such as the Unified Modeling Language (UML), which describes a system and preferably its environment. Moreover, as shown by Thomas Kühne in his paper 'What is a Model' [Küh04] there exists a lot of work trying to capture the essential features of models—i.e., what features an artifact needs to possess in order to be considered a model in every sense of the word.

For now, let us just follow the standard definition of model. One of the definitions of model that is generally accepted can be found in Oxford Dictionaries [1]: *'a simplified description, especially a mathematical one, of a system or process, to assist calculations and predictions'*. It then adds an example: *'a statistical model used for predicting the survival rates of endangered species'*.

2.1.1 Descriptions and Prescriptions

The above definition says that a model is a sentence that **describes** a real object (which might be a system or a process). This description is always a result of an interpretation of the observed phenomena on the described object. However, that object might not yet exist—a model for an object that is intended to exist in the future is called **prescription**, a **recipe**, or a **specification**. An example of a specification can be a design, sketch or plan of a bridge before its construction. If we build an object out of a specification, and make a new description for that object, then we can say that the object is **complying with the specification** if the description is not contradictory with that specification.

2.1.2 Levels of Detail (Abstractions)

The dictionary's definition also says that a model is a *'simplified'* description, which implies some notion of abstraction. For example, instead of describing a physical object, we may be interested to describe a particular aspect in a class of physical objects. In this abstract example, the effort of completely describing the physical object is somehow *'simplified'*, and the resulting model does not only refers to that object in particular, but instead to a set of objects that fit the studied aspect. This kind of models are called theories—i.e., they are descriptions which use some universal quantifiers such as 'for all', or 'always' when referring to the studied aspects in the objects; or directly referring to the studied class of objects—as in the dictionary's definition *'endangered species'*. Typically, scientists take advantage of using high levels of abstraction during a modeling activity, so that the

[1] http://oxforddictionaries.com/

resulting models (theories) that can then be used (as devices for prediction) to (as the dictionary's definition also say) *'assist on calculations and predictions'* about the described object.

Choosing an adequate level of abstraction gives the ability to the modeler to cope with the complexity of the object's representation. The more simplified a model is, the more easy it is to be understood, and analysed. In fact, the modeling activity always involves to choose what is the most suitable level of detail to describe a given object. This choice determines in a model what will be explicit, and what will remain implicit (or hidden in a rather undefined interpretation context). Moreover, while dealing with the increasing complexity of modern software systems (e.g., avionics), the modeler is forced to multiply these modeling activities through several orthogonal aspects of the intended software system. One example of such kind of orthogonal aspects, are the security requirements versus the functional requirements of a software system. Therefore, it is usual that while engaging on these modeling activities, the modeler is forced to know and use several different formalisms. This idea of having multiple formalisms (i.e., languages) during system's modeling, in what is called multi-paradigm modeling (MPM), was firstly introduced in [Van00] and explored in [PJM04]. Indeed, years before [Mil93], Milner also rejected the idea that there can be a unique formalism for describing all aspects of something as large and complex as concurrent systems modelling. Instead, modelers naturally need many orthogonal levels of description, different theories, and languages to express them.

2.1.3 Model's Quality

We just defined above some notion of quality between models, namely the compliance relation between a specification of an object and a description of the same object. But what can we say about the quality of models w.r.t. the described objects? What makes a good model? To answer to these questions we must look into two fundamental aspects: *correctness* and *adequacy*.

Regarding to the first aspect, we usually assert the correctness of a model that describes a given object (or class of objects), according to its **soundness** and **completeness** w.r.t. that object. A model is said to be **sound**, if all of its predictions are indeed observed in the real object (i.e., there are facts that were observed in the object that support and confirm the model). Notice that the guarantees of this soundness will obviously depend on both the level of abstraction used in that model, and on the precision of the measurement instruments used to confirm or otherwise disprove it. Conversely, we say that a model of an object is

complete, if and only if all of the observed phenomena on that object confirms or do not contradict what is predicted by that model. Again the guarantees of this completeness will directly depend on both the level of abstraction used in that model, and on the precision of the measurement instruments used to confirm or otherwise disprove it.

Regarding to the second aspect, we usually assert the adequacy of a model by evaluating if that model is expressed in a suitable formalism with the appropriate level of detail to be easily understood, and/or analysed. In other words, we must measure, in a given model, the impact of the choice of terms to be explicit or remain implicit, in the reader's ability to read it, analyse it, and use it for his/her calculations. Models are intended (by definition) to be *'simplified descriptions'* of objects in reality. And *'simplified'* means that a good model is able to cope with the complexity of the object under study, by just focusing in the essential terms and ignoring the irrelevant ones. The cognitive aspects of models and consequently the cognitive aspects of modeling languages are emerging research topics where both the DSL's evaluation techniques and best practices in domain specific modeling activities are being studied [BAGB11a, BAGB11c, BAGB12].

To summarize, to assert a model's quality, we must study *(i)* its **relation with the object** that is being described by it; and *(ii)* its **relation with the person** (or entity) that is reading it (and/or analyzing it).

2.1.4 Models expressed in a Language

When the definition from the dictionary says *'especially a mathematical one'*, it refers to the fact that most of these theories are expressed using the same set of patterns used by mathematicians to describe their theories. Moreover, it is common to use other mathematical theories (e.g., the set theory) to describe a new one. This says that usually, instead of a natural language, models are rather expressed in some kind of an artificial 'formal' language which usually has its own particular notation, such as mathematics. We then select the most adequate language to express models according to the level of abstraction that we desire for them (i.e., we choose what will be explicit terms, and what will remain implicit terms in that model). Even inside mathematics, we can find a huge plethora of different languages founded in mathematical theories, which give their users particular notations and operations that can be very useful to study and develop new models out of reality. An intuitive evidence of this is the expressiveness and power of calculus given by matrices to both express and solve equation systems. One of such notations are the Feynman diagrams, which were developed and

used by the Nobel Prize-winning American physicist Richard Feynman in 1948, in order to describe several rather complex models of sub-atomic particle interactions that are usually expressed by means of extensive physics equations. In fact, not only due to their cognitive capabilities, but also due to their extraordinary rigor and precision, they are still used today to describe this kind of models (see Figure 2.1).

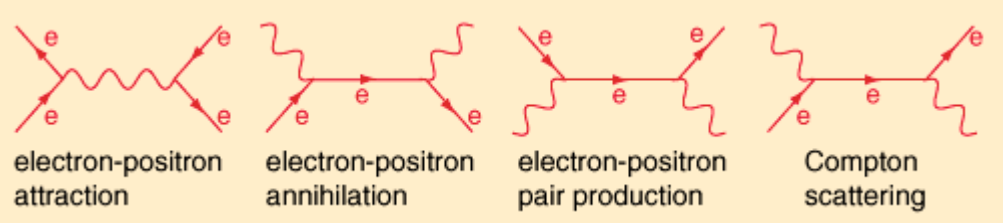

Figure 2.1: Particle interactions described using Feynman diagrams.

Languages are vehicles to express models as their sentences. Notice however that the same model can be expressed in many languages. We have seen before that there are two kinds of relations that we must consider in models: their relations with the users, and with the described objects. Essentially, a language is a means for communication between peers. For instance, two persons can communicate with each other by exchanging sentences. These sentences are composed by signs in a particular order. According to the context of a conversation, these sentences can have different interpretations. If the context is not clear, we call these different interpretations as being *ambiguous*.

Semiotics, as being the study of signs and communication, is divided into three parts: Pragmatics, Semantics and Syntax. These impact directly with the ability of the models expressed in a given language to perform the above mentioned relations. The **Pragmatics** of a language deals with the impact that the Signs used in every sentence of a given language have in the People (humans) that use them. For instance, the above referred cognitive capabilities of the Feynman diagrams are properties that are included in the Pragmatics of that language. With proper evaluation according to the target group of users, we can compare which language is most adequate to express a given model, considering several dimensions such as productivity and usability (which includes asserting all of the expressible model's readability, its learning curve, etc.). The **Semantics** of a language deals with the relations between the Terms (or Concepts) used in every model of a given language, and the described objects. Moreover, each one of these relations describes (all, or part of) the meaning of each Concept by saying how

they can be interpreted (i.e., mapped) into real objects. In general, with proper evaluation, we can assert if (all, or part of) the expressible models in a given language are sound and/or complete w.r.t. the respective described objects. Finally, the **Syntax** of a language deals with the relations among Terms (or Concepts) used in every model of a given language. In particular, *(i)* the **Abstract Syntax** of a language defines its set of Concepts and their inter-relationships; and *(ii)* the **Concrete Syntax** of a language (already mentioned as 'notation') defines its set of Signs (these are symbolic representations of the Concepts for the language's users) and their relationship with the defined language Concepts. Also note that the choice of the set of Signs on a Concrete Syntax of a language can directly impact both the capability of human interpretation of the sentences expressed in that language (i.e., its Pragmatics), and the capability of machine analysis of the sentences expressed in that language (i.e., its Semantics).

The above description of Semiotics holds for any kind of communication between cognitive entities—i.e., entities that are able to use language and symbols in order to communicate. This, of course, may involve the communication between humans, or between machines, or even between humans and machines. The first ones are called natural languages, and the second ones are usually called protocols. However in this thesis, we are mostly interested in the communication between humans and machines, since it is where our definition of DSMLs best fits: a language to provide bidirectional communication (i.e., interaction) between humans and computers. Therefore, in this sense: *(i)* DSML's Pragmatics usually refers to the cognitive capabilities of a DSML in w.r.t. the capability of human cognitive interpretation of that DSML's sentences; and *(ii)* DSML's Semantics usually refers to the meaning of that DSML's sentences in a computer system.

2.1.5 (Syntactic) Model of a Language

Both the Abstract Syntax and the Concrete Syntax of a language can be defined by means of a **grammar**, which is a mathematical device of some sort that can be used for either producing or recognizing the sentences of the language under analysis. Grammars are used to model and formalize the syntax of languages. Also, we can expect that a given language might have several different grammars that recognize it. In this case, all of the defined grammars will recognize the same input, but however produce different parse trees. Parse trees are trees that contain the Abstract Syntax Concepts as they were recognized from the input. Depending on the expressiveness of each language, we can find different types of

grammars. In fact, Chomsky defined a hierarchy of languages based on the nature of their respective grammars [Cho56, CS63]. For instance Type-3, Type-2 and Type-1 are three categories of grammars that belong to this hierarchy. This means that languages that are recognized by Type-3 grammars are also recognized by Type-2 grammars, and that languages that are recognized by Type-2 grammars are also recognized by Type-1 grammars, but not the opposite. Table 2.1 shows the different types of grammars identified by Chomsky, and their respective expressiveness which is directly related with the power to recognize languages in each language family. The most expressive and hence complex family of languages is the Recursively Enumerable languages, which only Type-0 grammars are able to express. These grammars are totally unrestricted, which means that there is not any general form to express them. Moreover, in order to recognize this kind of languages one needs to have the expressive power of a Turing Machine. The least expressive of all the languages are the ones that can be recognized using a Type-3 grammar, and they are called Regular Languages. According to Chomsky, in order to recognize this kind of languages one needs to have the expressive power of a finite state machine (FSM).

Grammar	**Language**	**Recognizer**
Type-0	Recursively Enumerable	Unbounded Turing Machine
Type-1	Context-Sensitive	Linear Bounded Automaton (LBA)
Type-2	Context-Free	Pushdown Automaton (PDA)
Type-3	Regular	Deterministic Finite Automaton (DFA)

Table 2.1: Chomsky Grammars and their recognizers.

Intuitively, the more expressive power a language has, the more difficult it is to guarantee the correctness of their parsers. For instance, we know from the Table 2.1 that the parsing procedure of a sentence of any Finite Language (i.e., a Regular Language containing only a finite number of words) will always terminate—given that the parsing procedure is equivalent to an FSM. However this property is, in general, not decidable for other kinds of Languages. Similar properties can be decided for Context-Free Languages expressed in Type-2 grammars—e.g., it is not possible to decide, in general, if a given Type-1 grammar generates any terminal strings at all, although it is decidable for Type-2 grammars expressed for instance in the Backus Normal Form (BNF).

Despite the fact that these syntactic models were originally developed to model and formalize the syntax of natural languages (such as English, or Italian), the above presented forms are mostly used to describe the syntactic models of Software Languages such as programming languages.

2.1.6 Software Languages

As mentioned above, in our research, we particularly focus in languages that are used as communication interfaces between humans and computers—i.e., User Interfaces (UIs). Examples of UIs range from compilers to command-shells and graphical applications. In each of those examples we can deduce the language that is being used to perform the communication between humans and computers: in compilers we may have a programming language; in a command-shell we may have a scripting language supported by the underlying Operating System (OS) in order to perform OS related tasks; and in a graphical application we may have an application specific diagrammatic language, and so on. Moreover, we argue that any UI is actually a realization of a language [BAGB11a], where in this context a language—we call them Software Languages from now on—is considered to be a theoretical object which rules what are the allowed terms, and how they can be composed into the sentences involved in a particular human-computer interaction. Notice, that languages can be deduced in two directions—human-to-computer and computer-to-human—since the feedback from the computer has to be given in such a way that it can be correctly interpreted by the humans.

The first Software Languages were the ones that could be used for the humans to interact with the first computers. These are called Programming Languages, and their users are called Programmers. There exist several Programming Languages using several programming paradigms. In the **Imperative Paradigm**, programming languages such as Assembly, Basic, C, or Object-Oriented (OO) Programming languages such as C++ and Java, describes the computation steps (or set of instructions) that the computer machine has to perform in a pre-determined order. When these instructions are executed, they typically change the memory of the computer machine, and usually it is not easy to track back (or get control of) all of the changes and side-effects in the internal memory resulting from all of the possible evaluations of those instructions considering any state that the computer machine might have at some point in time. In programming languages that use the **Functional Paradigm**, such as Lisp, or CaML, the computation steps are described by means of functions that when evaluated call other functions or data values in their arguments—i.e., each function called as a parameter of other function is evaluated and rewritten as a constant value which is then passed as an argument in order to evaluate the other function. This avoids the side-effects in the internal memory by only allowing the representation of the computer memory by means of function arguments. In programming languages that use the

Declarative Paradigm, such as Prolog, the computation steps are described by means of clauses (i.e, rewrite rules or simply facts), that the evaluation engine must satisfy. In this case the program can be considered as a set of equations or constraints, and the evaluation engine can be seen as a constraint solver, that searches for the solutions on those equations. Again there are no side-effects on the internal memory of the computer machine, and here there is not even any predetermined order from which the clauses are to be evaluated—i.e., it is said that the evaluation order is non-deterministic. Even so, these languages that use this **Declarative Paradigm** are still considered to be programming languages, because they can be generally used to program a computer.

However, there exist languages, such as Modeling Languages [Küh06], that no longer can be used to directly program a computer. Instead, these were built in order to be able to express the models with adequate notations. There exist Modeling Languages built and used for different purposes and reasons. Some, General Purpose, cover a wide spectrum of applications, such as the Unified Modeling Language (UML) or SysML, both proposed by the Object Management Group (OMG [2]), where the object of its descriptions is essentially software systems. Given the abstract level of these descriptions, and their general scope, it is generally not possible to use these languages in order to automatically synthesize a complete software system: in order to do so, one has to be able to include platform information, which is done by means of a programming language. Therefore, these descriptions can only be used as specification from a software architect or designer to a programmer that will implement the software system using a programming language (i.e., it can only be used as a means for human-to-human communication).

Domain-Specific Modeling Languages (DSMLs), are Software Languages specially suited for the needs of a particular domain that needs specialized computational tool support— typically in order to increase the productivity in that domain, not only by means of a formal language—i.e., also as a means for human-to-human communication, but also by means of their specialized analysers, checkers, simulators, automated code generators, etc.—i.e, as a means for human-to-machine communication. This kind of languages emerged in order to cope with the growing of both *essential* and *accidental* complexity in software engineering [Jr.75]. Besides having to provide solutions to solve a class of *essential* problems from a given domain (which are sometimes very complex to learn, such as

[2] http://www.omg.org

the rules and technical jargon found in domains like the Physics Computing, Financial Domain, among others), the Programmer has also to deal with the *accidental* complexity of the used computer technology—e.g., the use of programming languages (i.e., using a low level of abstraction in their sentences), while having to integrate and manage a wide plethora of different tools and libraries. These languages help realizing the MPM [PJM04] approach which follows the principles of User-Centered Design (UCD) [JIMK03]. This promising *divide-and-conquer* idea tries to break down the increasing complexity in software engineering by having multiple view points to look at any existing (or intended) object/artifact in multiple perspectives simultaneously, while avoiding the existence of any logical contradictions between those viewpoints. These different viewpoints can then be realized by means of several DSMLs, hence providing the end-users—i.e., the modelers involved in a DSM activity—with a pragmatic and usable way to understand their descriptions w.r.t. their particular viewpoints. Each DSML captures the occurrence of reusable patterns in a given domain, while describing an artifact in a particular perspective (i.e., limited to a given viewpoint or domain focus); and provides to the end-users with a pragmatic way to apply these reusable patterns in a controlled manner.

2.2 Software Language Engineering

A good DSML is hard to build since it requires both domain knowledge and language development expertise, and few people have both[MHS05]. The activity of DSML's validation is not a trivial task, and it can be both expensive and time consuming (mostly because it involves humans). Software Language Engineering (SLE) is the application of a systematic, disciplined and quantifiable approach to the development, usage, and maintenance of software languages. According to [MHS05] the Language life cycle consists of a set of phases, namely: Decision; Domain Analysis; Design; Implementation, Verification and Validation. In [HVV08] adds Deployment; and Maintenance to this process. SLE as a systematic approach to the construction of DSMLs is becoming a mature activity, building upon the collective experience of a growing community, and the increasing availability of supporting tools [Kle09]. A typical SLE process starts with the Decision and Domain Engineering phases, in order to elicit the domain concepts. The following step is to design the language, capturing the referred concepts and their relationships. Then, the software language is implemented which involves

implementing the editors of the language, the execution engines, debugging systems, type checkers, and so on. Then, the language is validated and verified, and finally, the language is documented and deployed. Furthermore, as any software engineering process, the language's life cycle also includes the maintenance and evolution, and retirement phases.

2.2.1 Decision and Domain Analysis of a Software Language

In the **Decision phase**, we also elicit the requirements of the to be built Software Language in order to answer questions such as: *Do we really need a new Software Language?* This does not differ so much from deciding if it is reasonable to build a new product line, since we have to take into account the reuse factor of existing products that can now (with a new software language) be automatically generated and verified, versus the whole cost of building a completely new software language. The **Domain Analysis phase** involves a thorough research in the terminology used in the domain, by looking into existing documentation such as Problem and Solution Descriptions. The rules of the domain under study may be either implicitly or explicitly defined, therefore we usually also need to perform informal interviews with the experts of the domain, and a survey of the existing tools such as: general purpose editors, simulators, compilers/interpreters, and general purpose execution engines (e.g., Java Virtual Machine). This particular kind of analysis, also called **Co-Domain Analysis**, involves the analysis of the variability at the level of the implementation (target) platforms. Here, we study the target platform variability, the same way we do for Software Product Lines. This variability model can be expressed by means of Feature Models using a Feature Model language [CHE04].

Based on these kinds of analysis, we have different methods of designing a DSML [Kle09]. That is, if we perform an analysis on the names of reusable components (in reusable infrastructures), and the reusable data structures and methods from existing APIs, and figure out all of the possible ways to combine them in a meaningful way, then we can infer our DSML from those reusable infrastructures. This is called the **bottom-up** method of designing DSMLs. This method may however generate languages that lack generality in the capability of solving any other class of problems from that domain. A **top-down** method lies in completing the domain analysis phase that is behind the existing reusable infrastructure, by discarding any existing implementation and focusing only on the complete description and categorization of the class of problems from which the users will use the DSML under design. In some domains this can be hard, since

the domain of the problem might not be fully bounded (categorized)—i.e., there may exist combinations of sentences which has no agreed meaning, or still under research. The effectiveness of these design methods will therefore depend on the domain under analysis. In practice, it is more usual that a DSML is designed using some sort of a combination of both bottom-up and top-down approaches.

2.2.2 Design Models of a Software Language

Regardless of the used method, in the **Design phase**, the language engineer specifies both the syntax and the semantics of his/her language, given the previous output resulting from both the Domain and Co-Domain Analysis. In what matters to the **syntactic models** of a Software Language, there exist several notations and languages that enable the specification of the syntax of a language. In particular, the already mentioned Backus-Naur Form (BNF) is a formalism that allows the specification of the structural shape of the language's sentences by means of a set of rewrite rules where we define the terminal and non-terminal symbols of the language. These language symbols are manipulated only according to what is allowed and specified in those rules, hence defining the syntax of the language under design. Other notations such as the MOF-based metamodels also allows the language engineer to specify the syntactic structure of his/her language. In this case, the space of all possible instances of a given language—i.e., the space of all of its valid sentences—is defined by means of an UML class-diagram like model called metamodel. Since these metamodels only deal with instances, typically they are used to only define the abstract syntax of the language under design. The metamodels can however be annotated with the required concrete syntax symbols, which can be either textual or diagrammatic. Moreover, it is said that the very syntax of the MOF [3] language which allows the specification of these metamodels is also defined by means of a metamodel. There are several language workbenches that use these notions of metamodels and conforming models. Perhaps the most popular is the Eclipse Modeling Framework, which uses a variant of these MOF-based metamodels called ECore[4]. In this tool, in particular, both models and metamodels are expressed in the XML Metadata Interchange (XMI) format, which is a standard also proposed by the OMG. The use of this format enables the interoperability between different software applications, as happens in integrated development environments (IDEs) such as software language editors. Other important advantage of having explicit models (persisted in XMI or XML

[3] Meta-Object Facility is an OMG standard.
[4] http://www.eclipse.org/modeling/emf/?project=emf

format) of software applications, is that it turns out to be more easy to inspect, perform changes or manage the application's data (i.e., its parameters) either manually or by other applications. Moreover, these tools manage models as being graphs, while using either relational theory as in relational databases [Dat04], or graph theory [Roz97]. Finally, from these syntactic definitions, these language workbenches are able to automatically prototype either graphical or textual editors for the DSMLs under development, which gives to the SLE a nice validation step during the design phase of a DSML.

In what matters to the **semantic models** of a Software Language, the Software Language engineer has to perform several choices on his/her language design. First of all, one has to choose what are the most appropriate computation models in order to define the meaning of the language under design's sentences, and for these there exists several models of computation [Fer09] such as sequential computation, several concurrency models, etc. The choice of what is the most appropriate computation model will depend on what is the ultimate purpose of our DSML. We can build a language to serve multiple purposes such as execution (which involves choosing from several possible execution platforms and their respective programming languages), optimisation, simulation, and analysis (which involves static analysis such as type checking, or dynamic analysis such as model checking). For instance, if the intention is to design the execution semantics of a language, then the chosen model of computation will have enough detail to precisely explain what is the meaning of each sentence of the language under design by means of computation steps. If otherwise, the intention is to design a semantics for providing a sound analysis of a language, then we can have courser models such as: *(i)* non-deterministic computation models, where some computation choices are left underspecified, allowing for the analysis algorithms to automatically explore all of the possible choices while searching for possible inconsistencies or errors in the specified sentences; or *(ii)* stochastic computation models, where we assign probability values to each possible choice in our language, so that we are then able to automatically compute what are the most probable outcomes of the computation of any given sentence expressed in our language under design. Depending on the chosen computation model, the software language engineer then chooses the most appropriate notation to give semantics to his/her language. Again, there are several ways to assign semantics to a software language. Perhaps the most intuitive way to describe the semantics of a software language, is by presenting pertinent examples of the language, and then explaining its meaning in an informal way (i.e., what is the computational

effect of that particular example)—this is also called natural semantics. A more precise way to specify language's semantics is to formally describe by means of a set of inference rules, the individual computation steps that the interpretation of a given construct of the language will produce in a symbolic representation of the current state of a virtual (abstract) computer—this is usually called the **structural operational semantics** [Plo04], SOS, or also small-step semantics. Another formal way to specify language's semantics is called **denotational semantics**, where each syntactic construct of the language is mapped into mathematical values by means of a set of equations. This mapping is again conditioned on a symbolic representation of the current state of a virtual (abstract) computation system. Although formal and precise, these specifications are indeed models. In fact they are abstract enough to provide some degree of platform independence, which means that with this kind of specifications, we do not compromise the meaning of the language under design with the programming language used in the underlying platform that will ultimately interpret, execute or analyse our language's sentences. However, there is a special version of denotational semantics, where we map each syntactic construct of our language into syntactic constructs of another language, instead of mathematics—this version is called **source-to-source translations**.

2.2.3 Implementation of a Software Language

After the **Design phase**, the language engineer uses both of the defined syntactic and semantic models to implement his/her Software Language. This is done in what is called the **Implementation phase**. Usually, the implementation of a DSML can be divided into the implementation of the language's editor, and the implementation of the language's interpreter engine or a language's compiler. The language's editor is a software program that allows the users to specify DSML's sentences, which can be textual, visual/diagrammatic, or both. Moreover, the language's editor implements the parser procedure that builds and stores in memory an abstract syntactic tree (AST) that internally represents the input sentence from the user. The language's execution engine, is a program that is able to dynamically process the internally stored AST and interpret it. Alternatively, the language's compiler can take this internally stored AST and generate source code expressed in a programming language that can be later on executed. Typically both of the language's interpreter and the compiler is programmed by the software language engineer using a programming pattern called the *visitor pattern*. The idea here is to 'visit' each part of the internally stored AST and call

(at each node of the tree) some procedures according to the defined language's semantic definitions. Typically, with these procedures a compiler directly implements **source-to-source translation** definitions of a DSML, and the interpreter directly implements the **structural operational semantics** definitions of a DSML. In the end of the implementation, the language developer has to interpret these semantic models and add additional details into it. These details are purely platform dependent information, but also essential so that the interpreter (or the result of the compilation) is able to correctly execute the DSML's sentences as defined in its semantics definition. Therefore, depending on how abstract and underspecified are these semantic models, there will be several degrees of freedom in the language developer's interpretation of these models.

The final phases of the language engineering process consists in the **Validation and Verification phases**. In the **validation** phase, the language engineer tries to assert if the built DSML is the right one. To do so, he/she performs an experimental evaluation of the built DSML's expressiveness and usability, which in turn involves to evaluate other aspects such as the productivity and effectiveness resulting from using the DSML under evaluation. The results' accuracy from this evaluation strongly depends on the active participation of real users or domain experts while actually using the DSML's implementation [BAGB11a, BAGB11b].

2.2.4 Verification and Validation of a Language

In software engineering in general, the *verification and validation* phase can however occur during the software development process, and even at its early phases. In particular, during the *Design* phase, the design models can be checked by means of specialized model checking tools [JGP99]. With these tools it is possible to check if the design models comply with certain quality properties, such as safety properties or reachability properties (i.e., the ability of the system to reach some state), hence validating the system under development even before its implementation.

Once the design model of a system is validated with respect to a set of quality properties, the software engineer must guarantee that the actual system implementation still has these properties—this is called *Software Verification*—which means that the software engineer asserts if he/she built the system correctly with respect to its design model. Notice that saying that a software implementation is **verified** does not mean that the product (as a whole, including its concepts, and its end-use) is **validated**, which is something usually done according to the end users. Hence, this is no different with DSMLs' implementations as they are

also software products. Nevertheless, software **verification** is one of the most important activities in what respects to assuring the quality of software.

There are software development methodologies such as VDM [MB97] or the B method [Abr96] that promotes the development of systems by successive refinements from design level models to implementation level models. The idea is to perform a special kind of safe refinements on high-level design models into lower-level models, so that the set of already proved quality properties on the initial design models are preserved. The proof of this preservation can be achieved by construction with the help of theorem provers such as PVS [MB97]. These methods can however be painful to be used in the practice of software engineering (and in particular in a SLE process), since they require special (formal) expertise from software engineers (which are usually programmers), and also due to the fact that typically the 'already proven' quality properties themselves have also to be somehow refined.

In the practice of SLE, the *validation* of software languages is usually performed by means of testing. However, if the SLE provides formal representation of the language's requirements, then testing can be used in order to perform its *verification*. For instance, based on the syntactic model of a language as the language's requirements, we can test the language's editor. Intuitively, **testing** involves stimulating a software implementation and observing its results while comparing them with the language's requirements. An **oracle** is an automatic decider/procedure which is able to interpret the results observed in a given test, and use the language's requirements in order to automatically decide if that test should either succeed or fail in a correct implementation. For instance, in order to consider an implementation to be correct, tests that are supposed to fail (e.g., failing tests that check safety properties) should not be observed by the oracle. However depending on both the size and the complexity of a system under test (SUT), the required number of tests to be applied on it in order to be able to give a reasonable decision about the SUT's correctness, can easily be unfeasible to be generated or applied in practice of a typical system's testing. Model based testing (MBT) techniques are being developed and applied in order to solve this problem with relative success in testing software systems in general. The main idea is that, with these techniques, not all but pertinent tests (and/or respective oracles) can be automatically generated (and/or automatically selected) based on the specified models of the implementations (and/or models of their requirements) [UL07]. These techniques are also being researched and applied to test DSMLs' implementations, in particular in order to test DSML's editors

with respect to their metamodel definitions [MPP08, MP10]. MBT techniques are compatible to be used during the DSML's verification since typically, the developed DSML is **verified for functional problems** by extensively testing both of the language's editor and interpreter engine (or compiler), based on their respective syntactic and semantic models. In particular, in what respects to the language's editor, the testing activity verifies if the implemented editor is correct and complies with respect to the defined syntactic models for that language. For testing a DSML's editor, the language engineer (or tester) uses the editor to express all of the language's syntactic constructs and some of its combinations that he/she finds more relevant in order to find problems. Similarly, the implemented interpreter engine (or compiler) is also verified in order to assess that it was implemented according to the respective DSML's semantic model. This verification is also done by means of extensive testing. Again the language tester expresses DSML's sentences that contains all of the language's syntactic constructs, and then check if their behaviour on the system is what it was expected in the defined DSML's semantic model. Depending on how abstract and underspecified are these semantic models, the more difficult is to assert its correctness.

Notice that any reasonable DSML produces an infinite amount of sentences, therefore to perform exhaustive testing of both of the DSML's editor and its interpreter engine (or compiler), virtually means to generate an infinite amount of tests—one for each sentence (which in practice it is simply not possible). Moreover, in order to test a single sentence, we must also observe its effect in a computer machine (or system), so that we can decide about its correctness. However, depending on the semantics of the DSML, there might be sentences which the size of their effect in a computer can not be observed and compared, even in a correct implementation. Nevertheless, this problem can be somehow diminished in some kinds of software languages. For instance, given the large years of use of general purpose programming languages such as Java and C, and the help of large communities of their intensive users (i.e., programmers in the software industry), the applied test coverage on testing the implementations of this kind of software languages can reach an acceptable rate.

2.2.5 Analysability of a Software Language

Intuitively, the ability to analyse models with respect to a given property strongly depends on the expressiveness of the underlying modeling language where we express them. It is well understood that different languages have different expressive powers, and the quality properties that languages can offer in their general use (e.g., confluence and termination) strongly depend on their expressive power. For instance, studies in a particular kind of languages called the process calculi algebras [Par08] indicate that while using recursion, process calculi algebras present different properties depending on how and what we restrict (or allow) them to express. However, the very use of powerful syntactic constructs such as recursion can stop us to even be able to compare between languages. For example, in general, formalizing translations between process calculi algebras is far from trivial, and can in some particular cases even become impossible [Gor10]. Another example is the model checker SPIN which is more suitable for modeling and verifying distributed systems, while the model checker PRISM is specifically designed for probabilistic systems [ASMZS11]. Moreover, while some programming languages provide type checking mechanisms that are able to automatically validate if the specified concepts in the programs are consistent with each other [Pie02], there exist modeling languages which modeling concepts can be also typed, and enable the expression of design models which can then be statically checked by means of constraint solvers, according to a set of pre-established design rules (expressed for instance in OCL [5], or Alloy [Jac06]).

2.2.6 Model-driven development of Software Languages

We defined the language engineering process, as being a process that inevitably uses models—both syntactic models and semantic models—of the software language under development. However, traditional methods used in software language engineering as described in [Fow05], do not necessarily use the presented notions of models of languages. For example, it is possible to build software languages by: *(i)* extending from other languages, while implicitly reusing their syntax or semantics (these are called internal DSLs); *(ii)* using macro definitions in programming languages such as C; *(iii)* defining new type definitions and their syntax in programming languages such as Ruby; *(iv)* using stereotyped UML as language-type definitions.

[5] http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL

Nevertheless, traditional methods used in SLE that do use models, typically also use grammar specifications of the software languages under development as syntactic models [Fow05]. Then, based on the defined grammar, and using parser generator tools (such as yacc [6] or javacc [7]), the software language engineer is able to automatically generate the code that implements a parser which automatically process the input data (i.e., the input sentences) according to the defined language's grammar. As mentioned before in this section, the parser is then able to build an internal abstract syntax tree (AST) in a program's memory based on the data read/loaded from an input stream. However, from here, the SLE typically has to write his/her own AST processor from scratch.

The need to have tools that are able to automatically process this AST tree, based on the semantic models of languages, was recognized by the research community that focus on Model-driven engineering (MDE). MDE is a software development methodology that uses explicit interchangeable descriptions (models) of the software artifacts as first class entities during the whole software engineering process, hence promoting reusability and analysability of all the used software artifacts and even their models. Moreover, MDE focuses on lowering the gap between domain specific models and computing (or algorithmic) concepts, by for instance, providing explicit models of languages (i.e., both syntactic and semantic models) instead of focusing directly on their implementations (i.e., their editors and interpreter engines or compilers). The main advantage of using models to describe software artifacts during software engineering is to be able to reason and analyse their correctness, at an appropriate level of abstraction. The reasoning ability also enables important software management decisions, such as identifying possible reuse strategies or refactoring, which can be an effective way of lowering the complexity of the whole development process.

However, there is an associated cost in using models of software instead of implementing it directly. For instance in the presented software engineering process of languages, this cost comes directly from their specification in the *Design* phase, to the challenge of implementing them in the *Implementation* phase, and further verify that their implementation is correct with respect to them in the *Verification* phase. This happens due to the fact that there is a natural clash between the formal definition of software languages, its concepts, and theorems—which

[6] http://www.techworld.com.au/article/252319/a-z_programming_languages_yacc/

[7] http://javacc.java.net/

is usually done by mathematicians, computer scientists, or gurus—and their implementation and verification—which is done by regular software engineers (developers and testers).

A similar software development methodology named Model Driven Development (MDD) tries to completely get rid of this gap by first focusing on the software design models [FR07], such as the ones presented for Software Languages in the *Design* phase, and relying on the machine alone to completely generate the implementation code from those design artifacts. In MDD, most decisions taken in the code generation are specified by software designers (instead of software developers) either directly in the source models, or by means of **model transformations** [Sel03].

The so called CASE (Computer-Aided Software Engineering) tools [8], constituted one of the first attempts of the software industry to lower this gap. These tools enable the specification of the intended software systems by using general purpose modeling languages (such as UML), and the automatic generation of code based on those specifications [9]. However, the effectiveness of these tools is limited due to several reasons. On the one hand, most of these tools only generate code skeletons (i.e., the class definitions and component interfaces, instead of the full code). On the other hand, the class of software systems that can be targeted by these tools is rather small.

In the particular case of Software Language Engineering process, there exists a wide range of MDD tools for supporting Language editor's implementation (also known as language workbenches [Fow05]) that become specialized in the rapid prototyping of textual and graphical/diagrammatic editors for DSMLs. For instance, on the one hand, Language Workbenches such as Microsoft DSLTools, the Eclipse's Graphical Modeling Framework (GMF) [10], and Meta-Edit are specialized in automatically prototyping/generating graphical editors for DSMLs [VT11]. On the other hand, Language Workbenches such as the Meta Programming System (MPS) [PP08], or the Eclipse's EMFText [HJK+09], to name a few, are specialized in automatically prototyping/generating textual editors for DSMLs. All of the aforementioned language workbenches (among others) are able to automatically generate/prototype the DSML's editor implementations from high-level

[8] http://www.unl.csi.cuny.edu/faqs/software-enginering/tools.html

[9] http://ithandbook.ffiec.gov/it-booklets/development-and-acquisition/development-procedures/software-development-techniques/computer-aided-software-engineering.aspx

[10] http://www.eclipse.org/modeling/gmp/

syntactic descriptions of the languages such as *BNF grammars* or *metamodels*, without any additional human intervention. Moreover, these language workbenches are also called metamodeling tools. This means that they realize the four-layered metamodeling architecture proposed by the Objects Management Group (OMG), namely the Meta-Object Facility (MOF) [11]. This modeling architecture is composed of four layers: meta-metamodels, metamodels, models and data, where the artifacts on the lower layers are instantiations of the immediate upper layers.

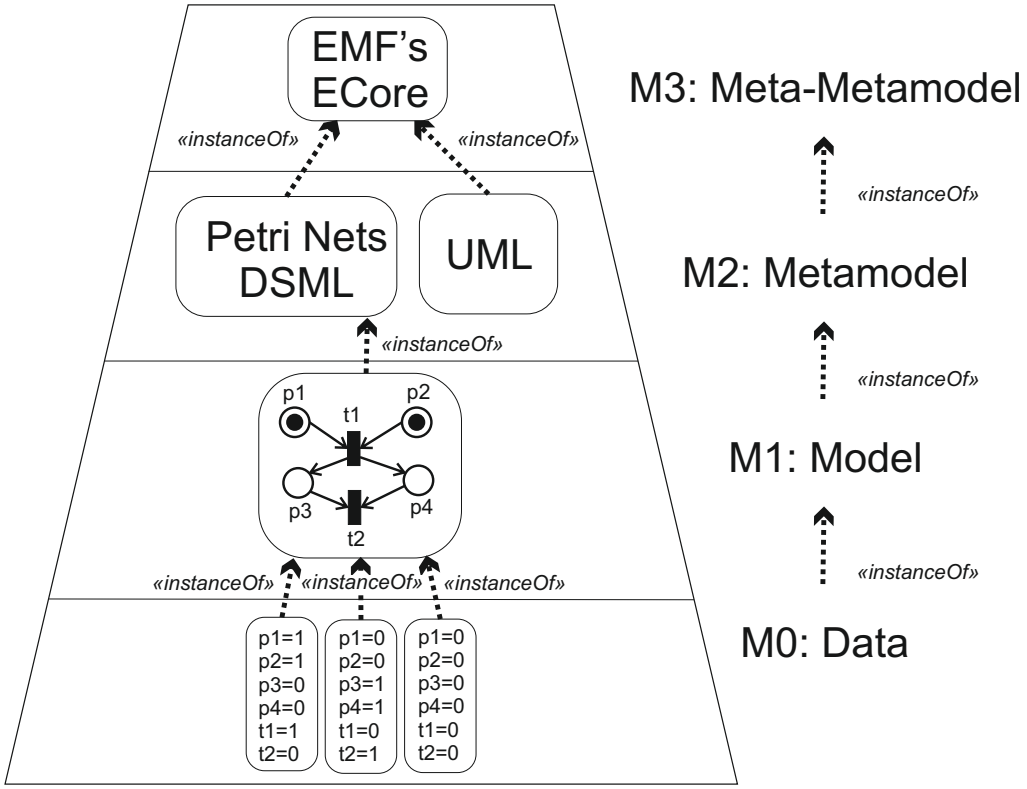

Figure 2.2: The Eclipse's GMF instantiation of the MOF's architecture.

As an example, Figure 2.2 depicts how the Eclipse's GMF instantiates the MOF's architecture. The GMF is in turn, based on the Eclipse Modeling Framework (EMF) where the meta-metamodel ECore is defined (see layer M3 in the Figure 2.2). The ECore is a class diagram that rules the kind of well-formed metamodels that one can use to define new DSMLs in the EMF. Thus, based on this ECore specification, the EMF provides specific tools that allow the language engineers to define metamodels for their DSMLs. Based on the specified ECore-compliant metamodel for the DSML, the GMF is able to automatically generate a graphical editor for that DSML. In the example of Figure 2.2, the language engineer first specified the metamodel for the Petri Nets DSML [Mur89] (i.e., layer

[11] http://www.omg.org/spec/MOF/2.0/PDF/

M2), and then automatically generated a graphical editor that enabled the specification of a Petri Net model (see the diagram at layer M1), composed of four places (labeled p1, p2, etc.), and two transitions (labeled t1 and t2). Finally, the models edited by the graphical editor are regular XML/XMI files compliant with the Petri Nets metamodel, and therefore they can be easily processed (i.e., loaded and managed) by using the EMF's Java API. In the example of Figure 2.2, the language engineer built an interpreter to evaluate all the possible configurations of all possible models expressed in the Petri Nets DSML. The result of this evaluation for the shown Petri Net model is shown in the boxes presented at layer M0, where each configuration shows the number of tokens at each place, and the enabled transitions are marked with '1', whereas the disabled ones are marked with '0'. Notice also that according to the OMG's terminology, Figure 2.2 depicts the *instance of* relations (by means of dashed arrows) between the different layers. Throughout this thesis, we will refine this *instance of* notion, and use instead the *conforms to* relation between model and metamodel (i.e., layers M1 and M2 respectively), and provide a formal definition of an algorithm that is able to check this relation.

In order to evaluate the DSML's sentences, the language engineer can, instead of building interpreters or compilers from scratch, benefit from help of specialized languages and supporting tools. Typically, code generation is best specified using a general programming language by means of code generation APIs such as the $System.Reflection.Emit$ from the .NET's $C\#$ [12]. However, a pure MDD process would even use models to describe the code generation of the DSML editor's implementation in a declarative way [EEHT05]. For instance, code generation tools, such as XPand or JET [13] provide a template based language so that the language engineer is able to specify transformations in order to produce textual code from high level specifications.

An important evidence that the integration of these tools in a sound MDD methodology can actually deal with the increasing software's complexity was realized with the BATICS project [RAB+09]. This project explored the MDD advantages in what concerns to usability and model verification (by means of model checking), and extensibility (by means of reuse of platform independent model transformations) of DSMLs.

[12] http://msdn.microsoft.com/en-us/library/system.reflection.emit.aspx
[13] http://www.eclipse.org/modeling/m2t/

2.2.7 Model Transformation Languages

Model Transformations are models that are able to manipulate other models in a safe and structured way. They are able to describe code generation by means of source-to-source translation specifications. Model transformations inherits their expressiveness from the graph grammars theory [Roz97]. A model transformation describes which models are acceptable as input of the transformation, and if appropriate what models it may produce as output of the transformation, by means of a set of rules—called the transformation rules. In the Figure 2.3, we present an example of such rule expressed in an MTL called EMF Tiger [BET08]. Acceptable input models are usually expressed as a pattern (written using the concepts from the source language metamodel) called the *match pattern*, or also the *left-hand-side* (LHS) of the transformation rule; and the produced output models are expressed as another pattern (written using the concepts from the target language metamodel) called the *apply pattern*, or also the *right-hand-side* (RHS) of the transformation rule. Additionally, one can also express *negative application conditions* (NACs) that extend the expressiveness of *match patterns* by restricting the match conditions of the input models. The rule shown in Figure 2.3, is actually creating a new Petri Net *transition* for every *next* relation found on any element of type *ActivityDiagram*, where both the NAC condition and the *Edge* element are used in order to avoid infinite recursive application of the rule.

The authors of EMF Tiger provided a complete formalization of their language (enabling further implementations in other platforms), including all concepts involved in every transformation expressed in their MTL. Moreover, based on this formalization, the authors were able to specify the particular kind of sentences in which we can decide if they are terminating, or if their results are confluent.

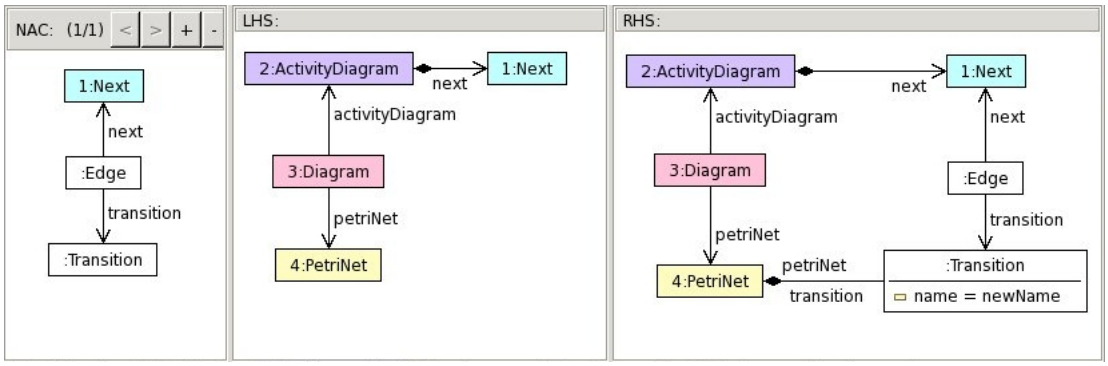

Figure 2.3: A transformation rule expressed in EMF Tiger.

A model transformation may be written in a general purpose programming language such as Java or C, however specialised model transformation languages are also available. There exist a wide range of Model Transformation languages (MTLs) and their supporting tools, such as Operational QVT, VIATRA2, and ATL [VABKP11], among others, that are able to express such model transformations specifications and execute them automatically. Therefore, we can also use these languages in order to automatically generate/prototype the DSML's interpreter engine or compiler. However, there is still a conceptual gap on using the available MTLs in order to describe DSML's semantic specifications such as *structured operational semantics* or *source-to-source translations*. This is due the fact that the available MTLs were not specifically developed to specify the DSML's semantics and automatically generate its implementations. Instead they were designed to enable the specification of many kinds of model transformations that serve many different purposes other than code synthesis, such as model refinement, consistency or evolution.

In practice, according to their different purposes, MTL's expressiveness change dramatically as explored in [CH03] and also in [ADL+12]. For instance, if an MTL supports code synthesis, then it is required for it to be at least **unidirectional**, which means that it should be able to transform models from a source language into models of a target language. However, if we want to express consistency between models, or support model evolution by means of model transformations, then it is required for it to be **bidirectional** (or even **multi-directional**—from many to many), so that models in both source and target languages can accommodate arbitrary changes and trigger transformations in both directions—i.e., from source to target languages, and/or vice-versa. If in a model transformation, the target language happens to be the same as the source language, then the model transformation is said to be an **endogenous** transformation, and **exogenous** if otherwise. Also if the execution of a model transformation is supposed to load an input model and just change it (once, several times, or indefinitely), then it is said to be an **inplace** transformation. If otherwise, the execution of a model transformation is supposed to load an input model and create a new one as its output model, then it is said to be an **outplace** transformation. Finally, if both the inputs and outputs of a model transformation are on the same abstraction level, then it is said to be an **horizontal** model transformation, and **vertical** if otherwise. Therefore, in an MTL that supports most of these features, some special proficiency is required for a software language engineer in order to specify his/her DSML's semantics, so that he/she is able to automatically implement

its interpreter engine (or compiler). In particular, on the one hand, *unidirectional*, *endogenous*, *horizontal* and *inplace* model transformations can be useful to describe model refinement, or also the *structured operational semantics* of DSMLs, so that it can automatically generate their respective interpreter engines. On the other hand, *unidirectional*, *exogenous*, *vertical* and *outplace* model transformations can be useful to describe *source-to-source translations* of DSMLs, so that it can automatically generate their compilers.

MTLs are the most popular modeling languages to express the semantic models of DSMLs. These are usually capable to both express the translational semantics of DSMLs, and their operational semantics. However, since these MTLs are made for many purposes, they introduce a cognitive gap on their users—i.e., their users usually experience a learning curve while writing a new kind of semantics, that is not reused among different semantics. Besides that these MTLs have also a lack of model checking support for the verification and analysis of the expressed model transformations. If we expect MTLs to be used in the context of large and complex industrial software engineering projects such as the development, certification and maintenance of a DSML's compiler, then we will also expect that the specified model transformations to also be large and complex to understand and analyse—even with the help of specialized verification algorithms. Although MTLs were made to help reducing the distance between the translation model and implementation of a DSML's compiler, there is still work to be done in what matters to MTLs' usability. In particular, while using these tools in medium-sized projects, it is common that the language engineer loses the big-picture of the model transformation rules that are being specified, which usually lead to error prone rules. The human interpretation of the transformation's syntax can be as complex as in general purpose programming languages due the recurrent use of low-level concepts used on this kind of tools. Even so there is some research trying to improve the usability of MTLs [SG12].

2.2.8 Analysis of Model Transformation Languages

Kuster [Kus04] presented important guidelines and properties that need to be checked during the validation of some model transformation are presented: Syntactic correctness of both input and output models; Termination and confluence (i.e., determinism and unique results of the transformation); Semantic equivalence or semantics preservation; Safety or liveness (to ensure preservation of structural or security properties). A valid model transformation is supposed to satisfy all or some of these properties. However, the proof that a model transformation is valid for any possible model expressed in some source language of a transformation expressed in the observed MTLs is in general not automatic and not even easy to master for a quality engineer.

There are many examples of work on trying to analyse language transformations at the meta level by reaching proofs from the transformation rules [vBV09], [BGL05]. For instance, in [vBV09] an encoding of lambda calculus to pi calculus (by three simple recursive rules) is presented, as well as the proof that some semantic properties of lambda calculus are preserved after the encoding into pi calculus. Although these languages are relatively small — even minimalist in our context — the proof that these semantic properties hold between them, is something still not trivial to perform by a language engineer. Also, the idea of using a common (canonical) semantic representation for language sentences was first introduced by Pnueli [PSS98] while trying to validate compiler translations of general purpose programming languages. However the problem of validating compilers for general purpose programming languages appears to be in some sense a more general problem, and a more particular problem in some other sense. In the one hand, as they do not use any structured/constrained way to specify their translations, the resulting theories that we (as language engineers) can write in existing theorem provers (such as Coq) in order to validate them can be of any kind. This of course means that the validation of a given translation will be limited to the expertise and knowledge that we have in the semantics of the languages involved in the translation under analysis (besides the handling of the theorem prover itself, which can be hard or impossible depending on the nature of the semantics involved). For instance, in [Chl10], a compiler of an impure functional language to an abstract assembly language is verified—the compiler was itself recoded in Coq theorem prover, hence introducing a gap with the actual implementation. In the other hand, after all the effort done in both the compiler's proof and implementation, the target languages of these compilations are (mostly) restricted to very low-level (machine code) compilations—therefore the

idea of validating translations by generating certificates as presented in [BG11] is feasible, but lack generality for validating translations that are not directly targeting execution platforms (for instance analysis platforms).

In order to aid the construction of the proof of semantic preservation along a set of transformation rules [ALL10] introduced a language to annotate those rules with assertions. The idea is to then pass these annotations to a reasoning framework that will derive, at the meta level, conclusions about the overall transformation. The work presented in [ABK07] aims at validating a model transformation by using the Alloy tool. In this case, Alloy simulates the transformation by generating a model example of the source language and then analyzing the results of the transformation.

In [VP03], it is presented an example of automated verification of the semantic preservation of a transformation between UML statecharts and Petri Nets. They generate instances of the source language and use them to model check for some dynamic property of UML statecharts, and then they again model check for the same dynamic property on the transformed Petri Net. They found practical limitations of this technique due to the state space explosion of the model checking.

The authors of [FHLN08] present a constructive fashion to automatically generate a valid transformation (the authors refers to transformations as ontology alignment) which in principle would preserve the semantic properties of the input and output models. This generation is done by using the Similarity Flooding algorithm which is based on the quantification of a notion of distance between source and target languages.

2.3 Summary

We have seen in this Chapter that the correct application of modeling activities and a correct choice of modeling languages during these activities, in the software engineering practice, can greatly improve its effectiveness by focusing on its *essential* complexity and lowering its *accidental* complexity. We have also seen that by itself, the task of developing a completely new DSML tailored to a specific application domain is far from trivial, therefore we need to provide well founded (formalized) languages and tools in order to support the Software Language Engineer in this task. Methodologies such as MDD can again be applied in the context of SLE by providing adequate language workbenches that are able to automatically prototype both the DSMLs editors and execution engines based on the DSMLs syntactic and semantic models respectively.

More importantly, these language workbenches must be able to guarantee the correctness of the generated execution engines. We have seen that, MTLs (and supporting tools) are the most adequate MDD solutions to automatically generate DSMLs execution engines from DSMLs' semantic models. The ability of analysis (i.e., analyzability) of transformations expressed in these MTLs is still under research, and is mostly dependent on the expressiveness of each MTL and their properties.

3

Overview of the Approach

In order to illustrate our approach, we present as a running example a translation between a toy language called State Machines and Petri Nets [Mur89]. On the one hand, State Machines is a widely-known language typically used to specify system's behaviour. It is a usual language engineer's choice to use such formalism as part of his/her DSML, since this it is able to represent system states in a declarative and diagrammatic fashion. On the other hand, the Petri Nets is a more expressive language, since it can explicitly express most often complex computation concepts such as non-determinism and concurrency. For the sake of simplicity, we avoid the use of inhibitor arcs in our version of the Petri Nets language. Therefore this example illustrates the usual case where we start from a less expressive DSL and translate it into a more expressive language such as programming language in order to make the DSL sentences executable.

Lets consider a scenario where a software language engineer develops his/her DSML by starting from the State Machines as a sub-language. The software language engineer first specified the State Machine language's abstract and concrete syntax, and then assigned its formal semantics by means of SOS. At some point the language engineer finds some convenience to be able to automatically translate the State Machine sentences into sentences expressed in the Petri Nets language. In this particular case, this translation can be particularly useful since it enables the reuse of the simulation facilities given by Petri Nets modeling tools—for instance, these simulation tools are able to automatically explore the specified non-determinism on the petri-net models in order to find inconsistencies or safety

problems.

Once the translation is specified, the final question then is how to assert that the specified translation is indeed correct. If we are able to effectively answer this question, then we will also know that a compiler (automatically derived from this translation) will also be the intended one.

3.1 Syntax of Languages: State Machine and Petri Nets

In this case, both the State Machine Language and the Petri Nets Language's abstract syntax were defined by means of Ecore-based metamodels [1], as shown in Figure 3.1 and Figure 3.2.

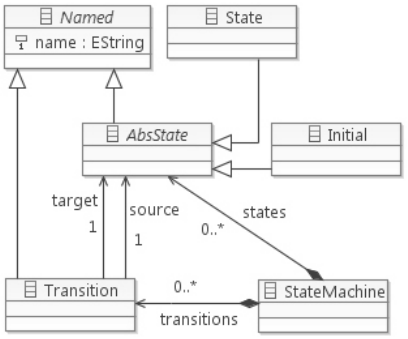

Figure 3.1: The State Machine Language Metamodel

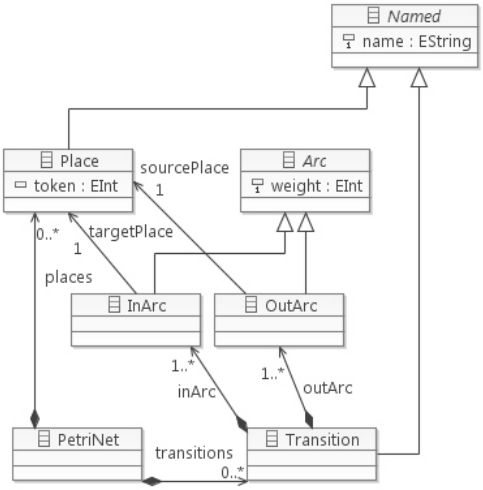

Figure 3.2: The PetriNet Language Metamodel

[1] http://www.eclipse.org/modeling/emf/?project=emf

Once the abstract syntax of the language is defined, the language engineer defines its concrete syntax and its semantics definition. The concrete syntax definition of a language usually extends the existing abstract syntax with symbols and usable metaphors that enable the domain experts to quickly understand the sentences in that language. In our examples, for readability, we prefer to use the concrete syntax versions of both State Machine and Petri Nets models. However notice that since most of the existing model transformation languages use their abstract syntax versions, we here present both versions. As a reference, we exemplify how the same sentences written in the State Machines Language look like by using its concrete syntax definitions (shown in Figure 3.3), and its abstract syntax version (shown in Figure 3.4). In the language engineers' intuition this model represents some computational behaviour of an hypothetical system which starts on some start state. From there the system can change its state by means of a transition named 'fire' into another state named 'Running' which represents that the system is already in execution. Then the system has two possible choices to evolve: either it changes its state into the state 'Stopped' by means of the transition 'end'; or it changes its state into state 'Fault' by means of the transition 'error'. From the state 'Fault' it is possible to change the state of the system back to the initial state named 'Start' by means of the transition 'reset'.

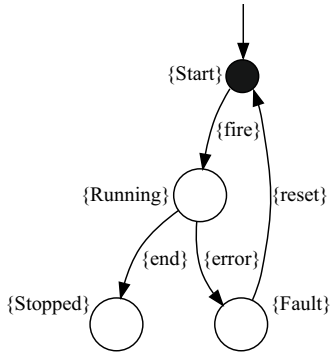

Figure 3.3: The standard visual representation of a State Machine using the State Machine language's concrete syntax.

This model can be expressed using the terminology of the Petri Nets Language. We here show how it looks like by using its concrete syntax definitions (shown in Figure 3.5), and its abstract syntax version (shown in Figure 3.6).

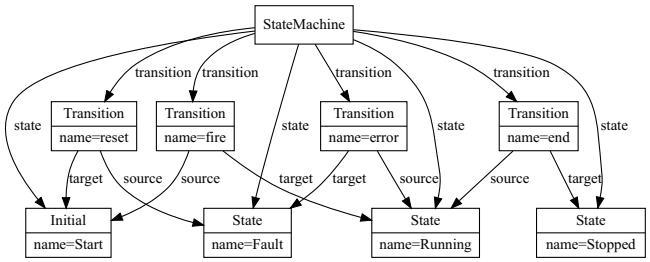

Figure 3.4: An internal hierarchical EMF representation of an instance model of the State Machine Language's metamodel as presented in Figure 3.1.

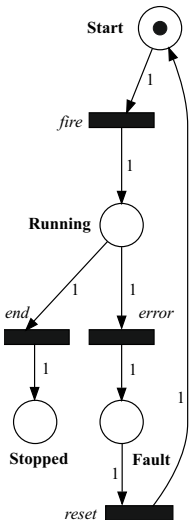

Figure 3.5: The standard visual representation of a Petri Net using the language's concrete syntax.

3.2 Software Language Translations

In our running example, the software language engineer implemented a compiler for his/her State Machine language. The compiler implements the translation shown in Table 3.1. The patterns in both columns refer to elements and their relations of the State Machine and Petri Net metamodels, respectively.

A correct implementation of this translation would in principle translate any kind of instance models of the State Machine Language, such as the one shown in Figure 3.3, into its correspondent instance model expressed in the Petri Net Language, such as the one shown in Figure 3.5. In fact, the model shown in Figure 3.5 could be created from the model shown in Figure 3.3 using the information presented in Table 3.1.

When we translate a model expressed in a source language into another model expressed in a target language, the first thing to consider while analysing its validity, is that it preserves the semantics of all of the expressible models in the source language of that translation.

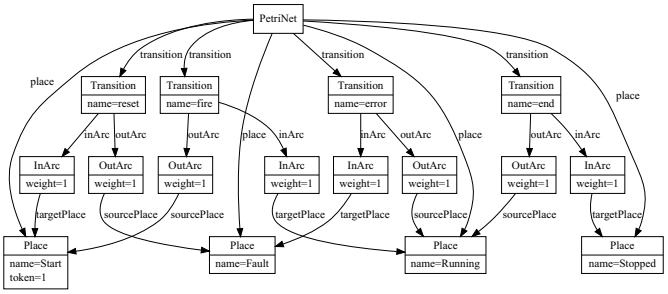

Figure 3.6: An internal hierarchical EMF representation of an instance model of the Petri Nets metamodel as presented in Figure 3.2.

State Machine Element	Petri Net Element
StateMachine	*PetriNet*
State	*Place with Token=0*
Initial	*Place with Token=1*
Transition	*Transition*
$StateMachine \xrightarrow{states} AbsState$	$PetriNet \xrightarrow{places} Place$
$StateMachine \xrightarrow{transitions} Transition$	$PetriNet \xrightarrow{transitions} Transition$
$Transition \xrightarrow{source} AbsState$	$Transition \xrightarrow{outArc} OutArc \xrightarrow{sourcePlace} Place^{(*)}$
$Transition \xrightarrow{target} AbsState$	$Transition \xrightarrow{inArc} InArc \xrightarrow{targetPlace} Place^{(**)}$

$^{(*)}$ Where OutArc.weight = 1
$^{(**)}$ Where InArc.weight = 1

Table 3.1: Translation table between State Machine Language and the Petri Net Language.

3.3 Operational Semantics of the Languages: State Machine and Petri Nets

For instance, in the Petri Nets version, the State Machine's states are encoded into resources (or 'Places' in the Petri Nets terminology), and the transitions of the State Machine are transitions of the Petri Nets with outgoing and incoming arcs having $weight = 1$. Despite the fact that these models are expressed in different languages, in the language engineer's own intuition they actually have the same meaning, because implicitly both specifications allow exactly the same 'moves'. But, how can we be sure of this?

Clearly, we need a way to explicitly define the meaning of both of these models — which for now remains implicit in the language engineer's intuition — so that we can actually compare them and conclude that they indeed have the same meaning.

The implicit meaning of each and every model expressed in a software language can be made explicit by means of formal mathematical descriptions such as the one proposed by Plotkin [Plo04]). Plotkin proposed that all the computation steps of a valid computer

program while running in a hypothetical computer system can be generically described by means of a finite set of pre/pos condition rules. These rules form what we call the **Structural Operational Semantics** (SOS) of a software language.

If we take a program expressed in a given software language, we can use these SOS rules to collect all of its possible computation steps, and then build up a graph which we call the program's **transition system**. In this graph, each edge is a transition generated by the conclusion of an application of a SOS rule while symbolically executing that program. These edges relates source and target vertices, where each vertex represents the computation state (i.e., values in a hypothetical machine's memory) before and after that transition occurred in the symbolic execution. A path in a given transition system, is called a **symbolic execution trace**. Note also that with these SOS rules, the implicit meaning of a finite sentence cannot simply be made explicit because there may be a possible infinite amount of possible SOS rule applications (i.e., its transition system can be infinite).

In this example, the language engineer defined two small (platform independent) semantics using the SOS terminology: one for each language. The **operational semantics of the State Machine language** is defined (for an arbitrary State Machine instance model s), by the minimum set of transitions in a transition system TS that satisfies the following rules:

$$\frac{\begin{array}{c}(Transition_s \xrightarrow{source} Initial_s) \in E^s, \\ (Transition_s \xrightarrow{target} AbsState_s) \in E^s\end{array}}{[cs(Initial_s) \xrightarrow{Transition_s.name} cs(AbsState_s)] \in TS_s}$$

$$\frac{\begin{array}{c}[cs(AbsState_s) \xrightarrow{Transition_s.name} cs(AbsState'_s)] \in TS_s, \\ (Transition_s \xrightarrow{source} AbsState'_s) \in E^s, \\ (Transition_s \xrightarrow{target} AbsState''_s) \in E^s\end{array}}{[cs(AbsState'_s) \xrightarrow{Transition_s.name} cs(AbsState''_s)] \in TS_s}$$

These inference rules define a transition system for any particular sentence s expressed in the State Machine Language—the s symbol represents a symbolic instance model which is conforming with the State Machine's metamodel. A transition system is a set of all the possible transitions $\xrightarrow{Transition_s.name}$ between current states of the specified state machine s denoted as $cs(\_)$. For instance, as depicted in Figure 3.7, if we consider a sentence s as being the state machine sentence depicted in Figures 3.3 and 3.4, then the transition system TS inferred from the rules above will only have four transitions. With the rules, it is easy to conclude that the transition system for any finite sentence s expressed in the State Machine Language is also finite—intuitively, given a particular state machine, the value of the current state value will range on all the defined states, and the number of $\xrightarrow{Transition_s.name}$ between the current states will be bounded by the number of

transitions defined in that particular state machine.

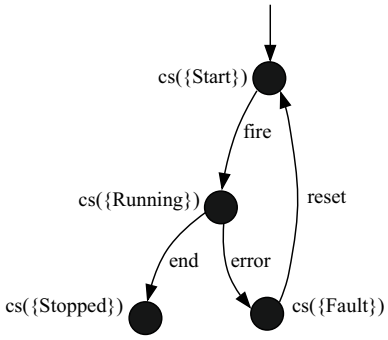

Figure 3.7: The transition system inferred from the State Machine rules when considering the State Machine presented in both Figures 3.3 and 3.4.

The first inference rule says that if we have a $Transition$ defined in a given State Machine sentence s, which is connected to both a state $Initial_s$ (by means of a $source$ relation), and to a state $AbsState_s$ (by means of a $target$ relation), then the transition system TS_s also contains a transition labeled $Transition_s$ connecting a state cs—which stores an instance element of type $Initial$—to another state—cs which stores an instance element of type $AbsState$. The second SOS rule says that, if *(i)* the current state of execution happens to be $AbsState'_s$ (written $cs(AbsState'_s)$), and *(ii)* there exists a State Machine's $Transition$ pointing to another state named $AbsState''_s$ (which by the way can be itself), then there also exists a $Transition_s$ element in TS_s which connects the current state $cs(AbsState'_s)$ to the next state $cs(AbsState''_s)$.

Notice that the interpretation of these rules strongly depend on what is actually defined in the State Machine's sentence s. The $Transition_s$ element is a syntactic construct of the State Machine's language defined in its metamodel as a meta-class with the same name, as shown in Figure 3.1. E^s represents the set of edges of the symbolic instance model s—in other words, if we look to instance model s as a graph, then E^s stores all of the associations of the instance model s as edges. The arrows $\xrightarrow{source}$ and $\xrightarrow{target}$ are also syntactic constructs of the State Machine's language defined in its metamodel as (non-containment) associations. Notice that the $name$ attribute selector $\xrightarrow{Transition_s.name}$ in the conclusion part of the rule refers to the value of the $name$ attribute (defined in the State Machine's $Transition$'s meta-class) from the $Transition_s$ element which also satisfies the rule's preconditions. Moreover, $AbsState_s$ represent elements of type $AbsState$ abstract states (which can be either $State$ elements or $Initial$ elements as defined in the State Machine's metamodel).

The transition system TS_s for a given sentence s is therefore a set of transitions, where each transition means that the specified state machine in s is able to move from a current state of $AbsState_s$ towards another state $AbsState'_s$. The labels of the transitions give

the possibility to identify the reason of why the state machine changed its state. Therefore, these two rules combined together gives the behavioural meaning to every possible expressible sentence s in the State Machine Language. In particular all of the *symbolic execution traces* for any expressible State Machine model s will start with a current state cs that refers to an existing $Initial_s$ element in model s.

The **semantics of the Petri Net language** is defined by the minimum set of transitions in the transition system TS_p that satisfies the following inference rules, which are defined for an arbitrary instance model p of the Petri Net Language:

$$\frac{(initial_p \ - \ pre(Transition_p)) \ \geq \ 0}{[initial_p \xrightarrow{Transition_p.name} pos(Transition_p) + (initial_p \ - \ pre(Transition_p))] \in TS_p}$$

$$\frac{\begin{array}{c}[prev \xrightarrow{name} curr] \in TS_p, \\ (curr \ - \ pre(Transition_p)) \ \geq \ 0\end{array}}{[curr \xrightarrow{Transition_p.name} pos(Transition_p) + (curr \ - \ pre(Transition_p))] \in TS_p}$$

In the above rules the semantic domain is defined by the notion of marking. A marking represents the number of tokens on each specified $Place$ at some point in time during the execution of Petri Net p. It is represented as being a pair $Place \times Token$, where $Place$ is a $Place$ element in p and $Token \subseteq \mathcal{N}$ represents the number of tokens in that $Place$ obtained from the respective attribute $Place.token$.

The transition system TS_p for an arbitrary petri net p is therefore the set of all the possible $\xrightarrow{Transition_p.name}$ transitions between marking states. For instance, as depicted in Figure 3.8, if we consider a sentence s as being the state machine sentence depicted in Figures 3.5 and 3.6, then the transition system TS inferred from the rules above will only have four transitions. Notice that in this case there may be instance models q of the Petri Net Language such that its transition system TS_q may not be finite. For instance, consider a Petri Net instance model which has one $Place$, one $Transition$ and an $InArc$ that connects the $Transition$ to the $Place$, then for each transition firing, we will have an increasing number of tokens: in this situation there will be an infinite number of transitions between markings, where each marking is a state in TS_q.

Similarly to the State Machine semantics, the Petri Net semantics is also defined by means of two rules: one for inferring the initial computational state of p, and another for inferring the remaining computational states. The function $initial_p$ computes the initial marking of the instance model p based on the token information of each specified $Place$. The function $pre(Transition_p)$ computes a marking where each $Place$ that is connected with a $Transition_p$ by means of an $OutArc$, is mapped to a number which represents

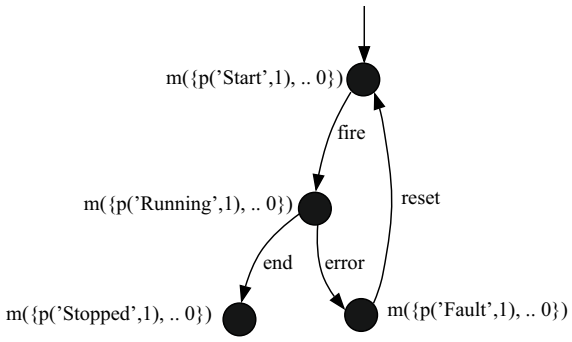

Figure 3.8: The transition system inferred from the Petri Net inference rules when considering the Petri Net presented in both Figures 3.5 and 3.6. In this example we used the term $m(\{p('Fault', 1), ..0\})$ as an abbreviation of $marking = \{Place('Start', 0), .., Place('Fault', 1)\}$—i.e., except the 'Fault' place, all other places have no tokens.

the weight of that $OutArc$. Similarly, the function $pos(Transition_p)$ computes a marking where each $Place$ that is connected with a $Transition_p$ by means of an $InArc$, is mapped to a number which represents the weight of that $InArc$.

The arithmetic operations '+', '-' and '$\geq\ 0$' were defined for these markings. Adding two markings A and B means the union of the pairs whose $Place$s do not intersect, and adding the token/weight values on the pairs whose $Places$ do intersect. The $\geq\ 0$ comparison returns true if for a given marking, all the pairs have a positive number of token values. Finally, $prev$ and $curr$ are free variables also of type marking.

Despite the fact that the language sentences' transition systems can be infinite, it is still possible to use these SOS descriptions for comparing their meanings or proving that two different sentences in a language have the same meaning or value, by establishing a semantic equivalence relation between their transition systems. Examples of these relations are the strong **bisimulation-equivalence**, or some other weaker forms such as the **simulation equivalence** [Par81]. Intuitively, two transition systems are bisimilar-equivalent if all of their possible moves (*symbolic execution traces*) match each other. In our case, we will use a weaker notion of **bisimulation-equivalence** which discards the labels on both states and transitions, and only considers both the shape of the transition systems under comparison, and a starting marker which represents the initial state of both transition systems.

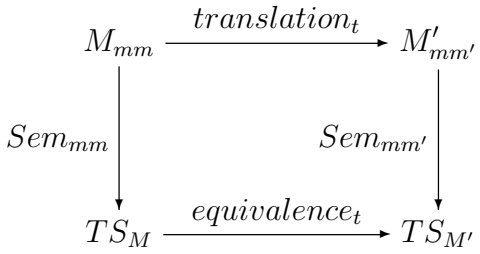

Figure 3.9: A commutative diagram illustrating the logical principles of our approach.

3.4 Analyzing Software Language Translations

We can therefore use these relations in order to validate software translations and their implementations (the compilers). Which means that the transition systems of every model expressed in the source language of a given valid translation, and of the transition system of its respective translated version (expressed in the target language) must be bisimilar. In other words, to prove that a given translation is valid, we would need to verify this equivalence between the transition systems of every possible sentence expressed in the source language, and of their counter-parts in the target language.

Since a language may produce an infinite amount of possible sentences, this proof would never terminate. Clearly, we have to take a closer look on how the translation is being specified, and extract from it a finite amount of relevant sentence pairs (from both source and target language) in order to check the semantic equivalence of their transition systems. These relations are illustrated in the commutative diagram in the Figure 3.9. Here, M and M' are representative sentences from source and target languages of the translation. By representative we mean that these sentences can be somehow extracted from a translation specification. Also, mm and mm' are metamodels identifying each language, Sem_{mm} and $Sem_{mm'}$ are each one a set of SOS rules defined for each language, and finally TS_M and $TS_{M'}$ are both the resulting transition systems from each sentence M and M' respectively. In practice, there may be several ways of transversing this diagram. However, in our running example we only show one operational method to transversing it in a tractable way.

This approach is therefore based on the principles of model based testing, where in this particular case, from a model of the translation under test, we are able to generate a finite set of relevant test cases including their respective oracles. On the one hand, each test case is formed by a pair stimulus (the source pattern) and its respective observation (i.e., the respective target pattern). On the other hand, the oracle is a procedure that is able to automatically compare (by means of a notion of bissimulation-equivalence) the semantic values of both the source and target patterns.

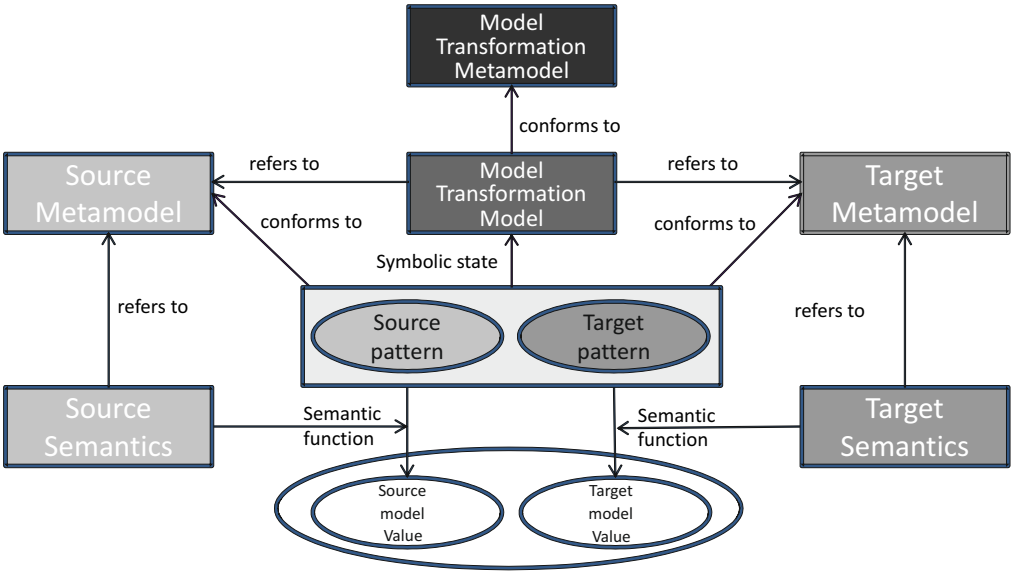

Figure 3.10: A framework for validating software language translations.

Our hypothesis, is depicted in Figure 3.10. If our translation under analysis is expressed as a transformation model in a graph-based model transformation language, then these 'relevant' sentence pairs can be obtained from the model transformation itself by analysing its rules and combining the source (match) and target (apply) patterns: the left hand side and right hand side graphs respectively, on a graph-based transformation language.

In other words, we should be able to execute any given model transformation without having any particular input model, but by computing a symbolic input model from the specified transformation rules. Notice that the relation of these 'relevant' sentence pairs with the translation under analysis is depicted in Figure 3.10 as being its 'Symbolic States'. We call these pairs of relevant sentences as symbolic states of the translation, since they represent intermediate or final relations between source and target sentences during the translation's execution. Following this line of reason, if we are able to compute a finite amount of relevant combinations of rule applications, and multiply by all the possible combinations of composing or merging these patterns together, then we might get a finite amount of relevant sentence pairs, hence giving the possibility to further check their semantic equivalence.

However, this represents a difficult challenge, since depending on the expressivity of the used model transformation language to express a given translation, it might be even impossible to extract a finite number of relevant translation's symbolic states from it. Moreover, validating such a translation depends directly on validating this set of symbolic states (given that it is a finite set). In order to do so, we have to find a way to compare both the source and target patterns on each symbolic state. This means that we have to find a way to compare sentences together in different languages. One approach

can be to provide a semantic function (defined by means of a Semantics model) that is able to interpret each pattern into a unique canonical representation for its meaning, so that we can compare them in a common ground (see bottom of the Figure 3.10).

3.5 Conclusions and Outlook

The hypothesis presented in this Chapter indicates that our research question can be answered in a generic way by building a methodology (including its associated tools and specialized modeling languages) that facilitates the instantiation of the framework shown in Figure 3.10. By 'generic', we mean that it should hold for any kind of software translation, involving any kind of DSMLs. In the next two Chapters, we use the presented illustrating example in order to instantiate the framework shown in Figure 3.10 on this concrete application, as depicted in Figure 3.11. In particular, we start by showing how to express the translation shown in the Table 3.1 using a proper model transformation language, which is able to automatically translate any model expressed in the State Machine Language into its respective representation in the Petri Nets Language. Then, we show how to express the presented operational semantics, again using a proper language, which is able to compute the meaning of any model expressed in both the State Machine Language or the Petri Nets Language, into a canonical algebraic representation that enable their comparison. Finally, we validate this translation according to their respective operational semantics definitions, by extensively comparing the canonical representations of both of the languages present in the symbolic execution space generated from this translation.

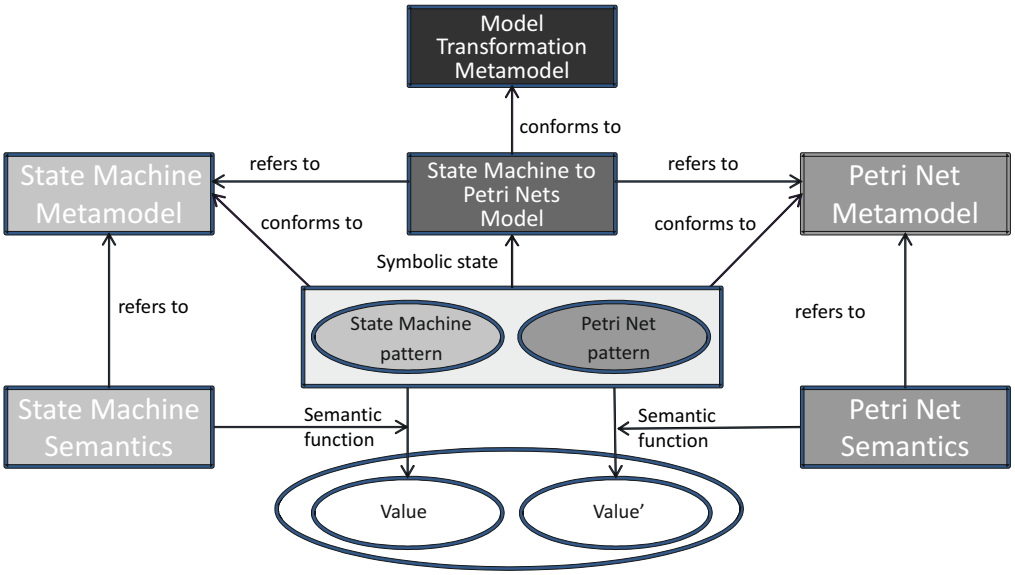

Figure 3.11: A framework for validating the State Machines to Petri Nets translation Model.

4

Models of Modeling Languages

In this chapter, it is introduced the formal foundations of the notion of Language, in particular the models used to describe Modeling Languages. On the one hand, in order to describe the syntax of DSMLs, we make use of the notion of models and metamodels. On the other hand, in order to describe the semantics of DSMLs, we use two of the most important types of semantic definitions for the purposes of our approach, namely translation semantics and operational semantics.

4.1 Syntactic Models

In our approach, models are first class entities. Models are descriptions of real life artifacts [MFBC10], and these descriptions are expressed in terms of some language (or languages). Moreover, we use formal syntactic models of software languages called **linguistic metamodels**, grammars (or just metamodels throughout this thesis), in order to be able to decide if a given model is a syntactically valid expression of a given language. This decision is realized by relating both the terms and their composition on a given model, with the terms and their composition on a given metamodel.

For instance, if we look at the shapes of these models as being graphs (i.e., made out of vertices and edges relating vertices), then we can relate the expressed sentences with the metamodel of its language by means of an instance relation (also referred in the literature as the **conformance relation**).

4.1.1 Typed Graphs

In order to clarify this formally, let us first define what is a typed graph.

Definition 4.1. *Typed Graph*

Let $\Sigma = \Sigma_v \cup \Sigma_e$ be a finite set of symbols that uniquely identify a given type, where Σ_v is the set of symbols for vertex types, and Σ_e is the set of symbols for the edge types.

A typed graph defined w.r.t. Σ is a 4-tuple $\langle V, E, \tau_v, \tau_e \rangle$ where:

1. *V is a finite set of vertices (also called terms),*
2. *$E \subseteq V \times \Sigma_e \times V$ is a finite set of directed edges connecting the vertices (the tuple members of this set are represented with an arrow $\xrightarrow{label}$, where $label \in \Sigma_e$),*
3. *$\tau_v : V \to \Sigma_v$ is a total typing non-injective function for labeling the vertices,*
4. *$\tau_e : E \to \Sigma_e$ is an auxiliary function such that $\forall y = (v1, label, v2) \in E \; . \; \tau_e(y) = label$, where $label \in \Sigma_e$*

The set of all typed graphs is called TG. Also, we denote V^g, E^g, τ_v^g and τ_e^g to be the sets of vertices, edges and the typing functions of the graph $g = \langle V, E, \tau_v, \tau_e \rangle \in TG$, respectively.

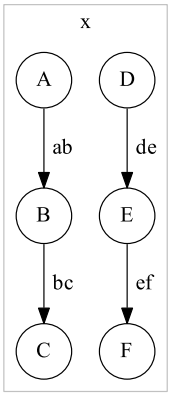

Figure 4.1: An example of a typed graph named x.

In Figure 4.1, we show an example of a typed graph which we named x. The white circles represents the vertices of graph x, and their labels represent the respective result of the τ_v typing function when applied on them. Similarly, the arrows represent the edges of graph x and their labels the respective result of the τ_e typing function when applied on them.

Next we defined some of the common operations that we can perform with the set TG.

Definition 4.2. *Typed Graph Union*

Let $\langle V, E, \tau_v, \tau_e \rangle, \langle V', E', \tau'_v, \tau'_e \rangle \in TG$ *be typed graphs.*

The typed graph union is the function $\sqcup : TG \times TG \to TG$ *defined as:* $\langle V, E, \tau_v, \tau_e \rangle \sqcup \langle V', E', \tau'_v, \tau'_e \rangle = \langle V \cup V', E \cup E', \tau_v \cup \tau'_v, \tau_e \cup \tau'_e \rangle$.
Also $\forall x \in V, x' \in V'$*, if* $x = x'$ *then* $\tau_v(x) = \tau'_v(x')$.
Similarly, $\forall y \in E, y' \in E'$*, if* $y = y'$ *then* $\tau_e(y) = \tau'_e(y')$.

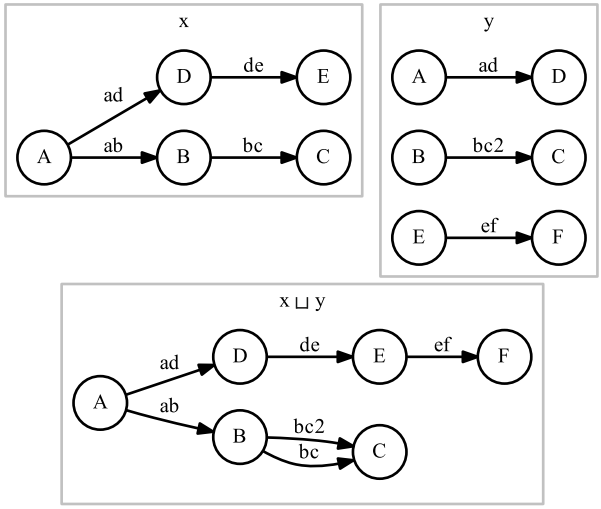

Figure 4.2: An example of two typed graphs (x and y), and their union.

In the example shown in Figure 4.2, the graph named $x \sqcup y$ (presented in the bottom of the Figure) represents the resulting graph of applying the typed graph union function $\sqcup$ to the graphs x and y (presented on the top of the Figure). Here, for simplicity, we assume that vertices with the same type label are the same (i.e., have the same internal identifier), which is not the general case. Therefore, since vertices labeled A, B, C, D, appear in both graphs x and y with the same label, the typed graph union do not produce duplicate vertices for each one of them. The same happens with the edges: the edge named ad which appears in both x and y, appears only once in $x \sqcup y$. Notice however that this do not happen with edges named bc and $bc2$ respectively: since they have different names, they are considered to be different edges and therefore they will both appear in the union graph $x \sqcup y$.

Definition 4.3. *Typed Subgraph*

Let $g, h \in TG$ be two typed graphs.

The graph h can be called a typed subgraph of graph g, written $h \blacktriangleleft g$ if and only if, $V^h \subseteq V^g$, $E^h \subseteq E^g$, $\tau_v^h = \tau_v^g|_{V^h}$, and $\tau_e^h = \tau_e^g|_{E^h}$.

Notice that $\tau_v^g|_{V^h}$ is a simplification of $\{(x \to \sigma) \in \tau_v^g \mid x \in V^h \ \wedge \ \sigma \in \Sigma_v^g\}$, where Σ_v^g is the set of symbols for vertex types from graph g. Similarly, $\tau_e^g|_{E^h}$ is a simplification of $\{(y \to \sigma) \in \tau_e^g \mid y \in E^h \ \wedge \ \sigma \in \Sigma_e^g\}$, where Σ_e^g is the set of symbols for edge types from graph g.

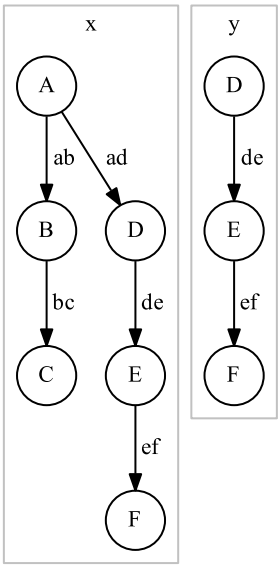

Figure 4.3: Typed graph y is a typed subgraph of typed graph x.

In Figure 4.3, the typed graph named y is a subgraph of typed graph x, written $y \blacktriangleleft x$. Here, for simplicity, we assume that vertices with the same type label are the same, which is not the general case. It is trivial to see that all of y vertices and edges are subsets of the vertices and edges of x respectively.

Definition 4.4. *Typed Graph Isomorphism*

Let g and $h \in TG$ be two typed graphs.

A typed graph isomorphism is a bijective function $\Theta : V^{TG} \to V^{TG}$ such that the following conditions are satisfied:

1. *for all $x \in V^g$, it is true that $\tau_v^g(x) = \tau_v^h(\Theta(x))$;*
2. *for all $x \in V^h$, it is true that $\tau_v^h(x) = \tau_v^g(\Theta^{-1}(x))$;*
3. *for all $e^g = (x \xrightarrow{lbl} x') \in E^g$, there exists $e^h = (\Theta(x) \xrightarrow{lbl'} \Theta(x')) \in E^h$;*
4. *for all $e^h = (x \xrightarrow{lbl} x') \in E^h$, there exists $e^g = (\Theta^{-1}(x) \xrightarrow{lbl'} \Theta^{-1}(x')) \in E^g$;*

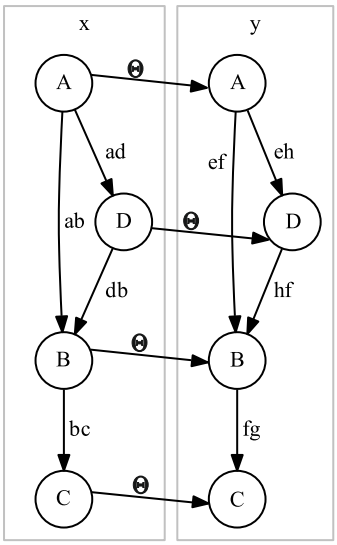

Figure 4.4: Θ is a typed graph isomorphism between typed graphs x and y.

The presented notion of typed graph isomorphism only requires that the involved typed graphs have the same shape, while using the bijective function Θ to map the vertices types on both graphs. In the example shown in Figure 4.4, Θ is represented as the following set of pairs:

$$\{(x_A, y_A), (x_B, y_B), (x_C, y_C), (x_D, y_D)\},$$

where $x_A..x_D$ are vertices of typed graph x, $y_A..y_D$ are vertices of typed graph y, the typing functions for typed graphs x and y are respectively $\tau_v^x = \{(x_A, A), .., (x_D, D)\}$, and $\tau_v^y = \{(y_A, A), .., (y_D, D)\}$.

Definition 4.4 extends the general notion of graph isomorphism. Trivially, this is still an equivalence relation on typed graphs.

Definition 4.5. *Typed Graph Equivalence*

If there exists a typed graph isomorphism Θ defined for two typed graphs $g, h \in TG$, and the following two conditions are satisfied:

1. *for all $e^g = (x \xrightarrow{lbl} x') \in E^g$ there exists $e^h = (\Theta(x) \xrightarrow{lbl} \Theta(x')) \in E^h$ such that $\tau_e^g(e^g) = \tau_e^h(e^h)$;*
2. *for all $e^h = (x \xrightarrow{lbl} x') \in E^h$ there exists $e^g = (\Theta^{-1}(x) \xrightarrow{lbl} \Theta^{-1}(x')) \in E^g$ such that $\tau_e^h(e^h) = \tau_e^g(e^g)$;*

then we say that g and h are equivalent, written: $g \cong h$.

Definition 4.5 uses the notion of typed graph isomorphism, by also checking the types of the edges of the vertices from both source and target graphs referred by the bijective function typed graph isomorphism Θ.

4.1.2 Models and Metamodels

We will now define both models and metamodels as being a special kind of typed graphs.

Definition 4.6. *Metamodel*

A metamodel is a typed graph such that:

1. $\Sigma_v \subseteq SymName \times \{abstract,\ concrete\}$ *is the set of vertex types, where* $SymName$ *is a finite set of possible names for symbols;*

2. $\Sigma_e \subseteq RelName \times RelKind \times RelCard$ *is the set of relation types, where* $RelName$ *is a finite set of possible names for relations,* $RelCard = \mathbb{N} \times \mathbb{N}$*, and* *RelKind* = $\{\ inheritance,\ attribute,\ reference,\ containment\ \}$.

The pair $RelCard$ refers respectively to the meta-edge definition of the minimum and maximum cardinality of occurences allowed for that particular edge type in a given model. In some metamodeling frameworks the maximum cardinality in $RelCard$ can take the value of $\star$, representing an unbound number of values. In our formalization, this symbol can be represented by the maximum number supported by a reference implementation. The set of all metamodels is called MM.

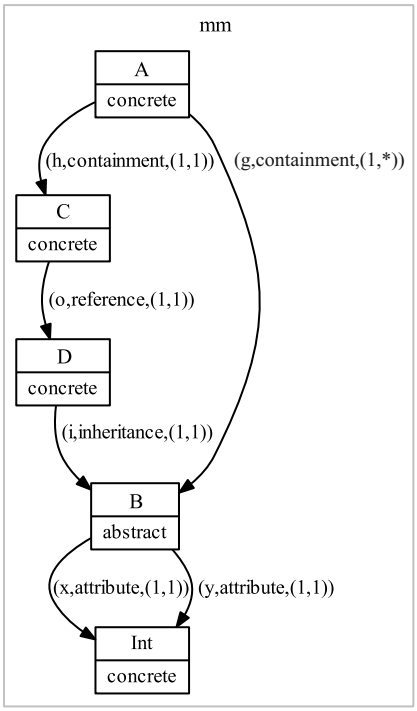

Figure 4.5: An example of a metamodel typed graph.

The typed graph presented in Figure 4.5 is also called a metamodel. In this example, the vertex types are pairs which are represented as records inside rectangles. For the $inheritance$ typed edges, both the name of the edge (i) and its cardinality $(1, 1)$, are

irrelevant and could be removed or hidden, although we represent them in the Figure in order to better illustrate the presented formalization. The vertex typed Int represents the basic type of Integers, which are usually the types of the attributes. However, the design decision of including basic types in a metamodeling framework strongly depends on its implementation, and therefore we do not compromise with further details here.

Now we define some convenient functions over metamodels which will allow us to easily extract/select the required information from the metamodel graph structures.

Definition 4.7. *Metamodel Functions*

Let $mm \in MM$ be a metamodel.

(i): The function $Name^{mm} : V^{mm} \to SymName$ is defined such that $Name^{mm}(x) = n$ if and only if $\tau_v^{mm}(x) = (n, s)$, where $x \in V^{mm}$ and $(n, s) \in \Sigma_v^{mm}$— i.e., it returns the symbol name for a given vertex $x \in V^{mm}$;

(ii): The function $Name^{mm} : E^{mm} \to RelName$ is defined such that $Name^{mm}(x) = n$ if and only if $\tau_e^{mm}(x) = (n, s, m)$, where $x \in E^{mm}$ and $(n, s, m) \in \Sigma_e^{mm}$ — i.e., it returns the symbol name for a given edge $x \in E^{mm}$;

(iii): The function $Kind^{mm} : V^{mm} \to \{abstract, concrete\}$ is defined such that $Kind^{mm}(x) = s$ if and only if $\tau_v^{mm}(x) = (n, s)$, where $x \in V^{mm}$ and $(n, s) \in \Sigma_v^{mm}$ — i.e., it returns the symbol kind for a given vertex $x \in V^{mm}$;

(iv): The function $Kind^{mm} : E^{mm} \to RelKind$ is defined such that $Kind^{mm}(x) = s$ if and only if $\tau_e^{mm}(x) = (n, s, m)$, where $x \in E^{mm}$ and $(n, s, m) \in \Sigma_e^{mm}$ — i.e., it returns the symbol kind for a given edge $x \in E^{mm}$;

(v): The function $MinCard^{mm} : E^{mm} \to \mathbb{N}$ is defined such that $MinCard^{mm}(x) = min$ if and only if $\tau_e^{mm}(x) = (n, s, (min, max))$, where $x \in E^{mm}$ and $(n, s, (min, max)) \in \Sigma_e^{mm}$ — i.e., it returns the minimum cardinality for a given edge $x \in E^{mm}$;

(vi): The function $MaxCard^{mm} : E^{mm} \to \mathbb{N}$ is defined such that $MaxCard^{mm}(x) = max$ if and only if $\tau_e^{mm}(x) = (n, s, (min, max))$, where $x \in E^{mm}$ and $(n, s, (min, max)) \in \Sigma_e^{mm}$ — i.e., it returns the maximum cardinality for a given edge $x \in E^{mm}$;

Definition 4.8. *Inheritance Partial Order Relation*

Let $mm \in MM$ be a metamodel, and $(E^{mm})^*$ be the transitive closure of the edge set of mm.

*(i): We say that the pair $(\tau_v(y_s), \tau_v(y_t))$ belongs to the **Inheritance relation** $Inherits_{mm} \subseteq \Sigma_{mm} \times \Sigma_{mm}$, if and only if there exists $y = (y_s \xrightarrow{label} y_t) \in (E^{mm})^*$ such that $Kind^{mm}(y) = inheritance$, where $\tau_v(y_s), \tau_v(y_t) \in \Sigma_v^{mm}$;*

*(ii): The **partial order relation** $\leq_{mm}$ uses the above defined relation such that:*
$\leq_{mm} = (Inherits_{mm})^* \cup \{ (\tau_v(y), \tau_v(y)) \mid y \in V^{mm}\}$,
having that if $(a, b), (b, a) \in \leq_{mm}$ then $a = b$ (i.e., it must not have inheritance cycles);

*(iii): We say that mm is a **valid metamodel**, if and only if the subgraph formed by $\langle V^{mm}, \{y \in E^{mm} | Kind^{mm}(y) = inheritance\}\rangle$ is acyclic. From now on we refer to metamodels as being valid ones.*

In the above definitions, we used the symbol * to denote the transitive closure of the defined binary relations. Notice also that with the defined conditions, it is trivial to observe that $\leq_{mm}$ is indeed a partial order relation: it is reflexive, transitive and antisymmetric.

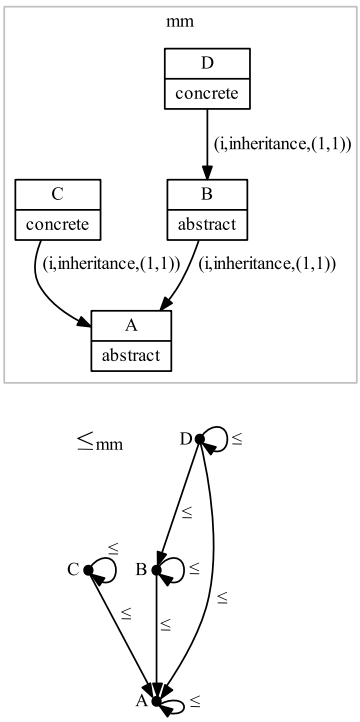

Figure 4.6: An example of a metamodel typed graph mm (on top), and the inheritance partial order relation $\leq_{mm}$ (on bottom) induced from metamodel mm.

In Figure 4.6, we show an example of the inheritance partial order relation induced from the presented metamodel named mm (on top of the Figure). If x_s is connected to x_t by means of an edge y, such that $Kind^{mm}(y) = inheritance$, then we read that x_s *inherits from* x_t, or x_s *is more concrete than* x_t, or x_s *is less or equal abstract than* x_t.

In the presented example, we can say that type C is less or equal abstract than A, or itself. Notice however that in this example there is no direct relation between types C and B (nor D).

Definition 4.9. *Vertex-Wise Type Satisfaction*

Let $m \in TG$ be a typed graph, and $mm \in MM$ be a metamodel, $x \in V^m$ be a vertex in the vertices of m, and $y_t \in V^{mm}$ be a vertex in metamodel mm.

The vertex-wise type satisfaction is a relation $\vdash_{mm}$: $V \times MM \times V$, such that if $x \vdash_{mm} y_t$ then it means that either:

(i): $\tau_v^m(x) = Name^{mm}(y_t)$, and $Kind^{mm}(y_t) = concrete$;

or (ii): there exists $y_s \in V^{mm}$ such that $\tau_v^m(x) = Name^{mm}(y_s)$, $\tau_v^{mm}(y_s) \leq_{mm} \tau_v^{mm}(y_t)$ and $Kind^{mm}(y_s)$ = concrete.

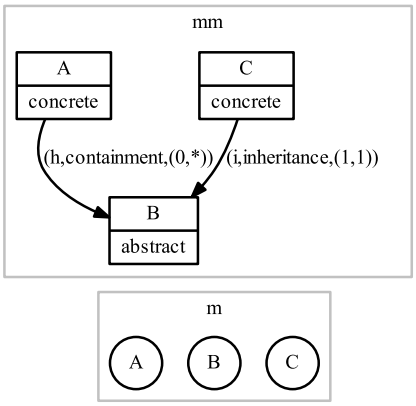

Figure 4.7: An example of a metamodel typed graph mm (on top), and a typed graph m, where the vertex typed B does not satisfy the metamodel mm.

In order to illustrate the relation vertex-wise type satisfaction defined on an arbitrary metamodel mm ($\vdash_{mm}$), we present the example shown in Figure 4.7. Here we can easily see that the $\vdash_{mm}$ relation can be represented as the following set of pairs:

$$\vdash_{mm} = \{(m_A, mm_A), (m_C, mm_B), (m_C, mm_C)\},$$

where m_A, m_B, and m_C are vertices of typed graph m; mm_A, mm_B, and mm_C are vertices of metamodel mm; the typing function for typed graph m is $\tau_v^m = \{(m_A, A), (m_B, B), (m_C, C)\}$; and the typing function for metamodel mm is $\tau_v^{mm} = \{(mm_A, A), (mm_B, B), (mm_C, C)\}$. Notice also that $\{(m_B, mm_B), (m_B, mm_C), (m_B, mm_A)\} \cap \vdash_{mm} = \emptyset$, because none of these pairs respect the two conditions defined in Definition 4.9, in particular due to the fact that m_B is an abstract vertex.

Definition 4.10. *Kind of a Typed Graph Edge*

Let $m \in TG$ be a typed graph, and $mm \in MM$ be a metamodel.

The kind of an edge $x = (x_s \xrightarrow{label} x_t) \in E^m$ w.r.t. the metamodel mm can be given by the function $Kind_m^{mm} : E \to RelKind$, such that $Kind_m^{mm}(x) = kind$ if and only if there exists $y = (y_s \xrightarrow{label'} y_t) \in E^{mm}$, where $\tau_e^m(x) = Name^{mm}(y)$, $Kind^{mm}(y) = kind$, $x_s \vdash_{mm} y_s$ and $x_t \vdash_{mm} y_t$.

In order to illustrate the function $Kind$, we present the example shown in Figure 4.8. Here we considered that m_A, and m_C are vertices of typed graph m; mm_A, mm_B, and mm_C are vertices of metamodel mm; the typing function for typed graph m is $\tau_v^m = \{(m_A, A), (m_C, C)\}$; and the typing function for metamodel mm is $\tau_v^{mm} = \{(mm_A, A), (mm_B, B), (mm_C, C)\}$. It is trivial to notice that $m_A \vdash_{mm} mm_A$, and the naming function $Name^{mm}(mm_A \xrightarrow{(h,containment,(0,*))} mm_B) = \tau_e^m(m_A \xrightarrow{h} m_C) = h$. Therefore, if we consider the inheritance relation between the elements typed B and C on metamodel mm, then we can see that when we apply function Kind to the edge $m_A \xrightarrow{h} m_C$, it is true that $Kind_m^{mm}(m_A \xrightarrow{h} m_C) = Kind^{mm}(mm_A \xrightarrow{(h,containment,(0,*))} mm_B) = containment$. This

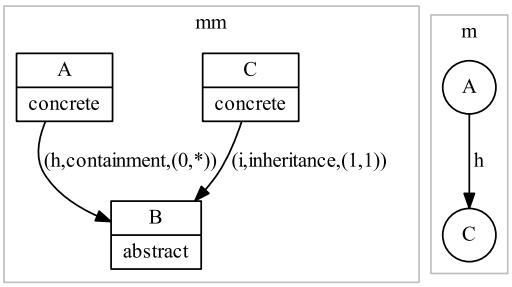

Figure 4.8: In typed graph m (on the right), the Kind of the edge typed h w.r.t. metamodel mm (on the left) is $containment$.

happens because $m_C \vdash_{mm} mm_B$. In other words, since the C element m_C is a specialization of the B element, the defined h association on model m must be a containment w.r.t. metamodel mm.

We will now define a model also as being a typed graph. However in our formalization, a typed graph can only be considered a model in the context of a given metamodel.

Definition 4.11. *Model*

Let $m \in TG$ be a typed graph, and $mm \in MM$ be a metamodel. m can be considered to be a model if and only if the following conditions are satisfied:

(i) for all $x \in V^m$, there exists $y \in V^{mm}$ such that $\tau_v(x) = Name^{mm}(y)$ and $Kind^{mm}(y) = concrete$;

(ii) for all $x = (x_s \xrightarrow{lbl} x_t)$, $x' = (x'_s \xrightarrow{lbl'} x_t) \in E^m$,
$Kind_m^{mm}(x) = Kind_m^{mm}(x') = containment \implies x_s = x'_s$. The idea here is that elements x_t can only be contained at most in one element.

A model being an instance of (or conforming to) a metamodel means that *(i)* every concept in the sentence is also present in the metamodel of a given language, and *(ii)* every relation between two concepts present in the model are also present in the metamodel of that language relating those concepts.

Additional constraints can be introduced in order to distinguish the types of these relations (e.g containment or reference relations), and their cardinalities (e.g one to one associations, one to many, etc.).

The cardinality constraint can be formally defined as a satisfaction relation, as follows.

Definition 4.12. *Cardinality Satisfaction*

Let $m \in TG$ be a typed graph, $mm \in MM$ be a metamodel, $\{x_s, x_t\} \subseteq V^m$ be vertices of m, $(x_s \xrightarrow{lbl} x_t) \in E^m$ be an edge in model m, and $y = (y_s \xrightarrow{lbl'} y_t) \in E^{mm}$ be an edge in metamodel mm.

The cardinality satisfaction is a binary relation $\#_{m,mm} : V \times E$, such that $x_s \ \#_{m,mm} \ y$ is true if and only if:

$$MinCard^{mm}(y_s \xrightarrow{lbl'} y_t) \leq$$
$$\|\{(x_s \xrightarrow{lbl} x_t) \in E^m \mid \tau_e^m(x_s \xrightarrow{lbl} x_t) = Name^{mm}(y_s \xrightarrow{lbl'} y_t) \ \wedge \ x_s \vdash_{mm} y_s \ \wedge \ x_t \vdash_{mm} y_t\}\|$$
$$\leq MaxCard^{mm}(y_s \xrightarrow{lbl'} y_t)$$

In the above definition, the operator $\|\|$ counts the size of a set (e.g., $\|\{\}\| = 0$). Intuitively, the above definition says that given a vertex x_s from a model m, and an edge y, from a metamodel mm, if the sum of all the outgoing edges of vertex x_s that have the same name as the name of the reference edge y lies within the range defined by $MinCard$ and $MaxCard$ functions (when applied to that edge y), then we say that x_s satisfies the cardinality relation written: $x_s \ \#_{m,mm} \ y$. Furthermore, we will use a relaxed version of the Cardinality Satisfaction (written $x_s \ \#_{m,mm} \ \lceil y \rceil$), where we only check that the value of the sum ranges instead between 0 and the maximum defined cardinality.

This relation between a vertex of a typed graph, and an edge from a metamodel is useful to check in a given typed graph if the sum of all edges of a given type that connect a given source vertex to any other vertices, is respecting the cardinality restrictions of that edge type in the metamodel. In particular, this satisfaction relation is used on the following satisfaction relation on edges.

Definition 4.13. *Edge-Wise Type Satisfaction*

Let $m \in TG$ be a typed graph, $mm \in MM$ be a metamodel, $x = (x_s \xrightarrow{lbl} x_t) \in E^m$ be an edge in the edges of m, and $y = (y_s \xrightarrow{lbl'} y_t) \in E^{mm}$ be an edge in metamodel mm.

The edge-wise type satisfaction is a relation $\vdash_{mm}$: $E \times E$, such that $x \vdash_{mm} y$ if and only if the following conditions are satisfied:

(i): $\tau_e^m(x) = Name^{mm}(y)$,

(ii): $x_s \vdash_{mm} y_s$,

(iii): $x_t \vdash_{mm} y_t$,

and (iv): $x_s \ \#_{m,mm} \ y$.

Similarly, the relaxed version of edge-wise type satisfaction for an edge x of model m with an edge y of metamodel mm, written $x \vdash_{mm} \lceil y \rceil$, uses the relaxed version of cardinality satisfaction $x_s \ \#_{m,mm} \ \lceil y \rceil$.

Let us now formally define the conformity relation.

Definition 4.14. *Conformity Relation*

Let $mm \in MM$ be a metamodel and $m \in TG$ be a typed graph. We say that m conforms with the metamodel $mm \in MM$ (written $m \vdash mm$) iff all of the following conditions are satisfied:

(i): m is a model with respect to mm,

(ii): for all $x \in V^m$, there must exist $y \in V^{mm}$ such that $x \vdash_{mm} y$,

(iii): for all $z \in E^m$, there must exist $w \in E^{mm}$ such that $z \vdash_{mm} w$,

and (iv): for all $w = (w1 \xrightarrow{lbl} w2) \in E^{mm}$, if $MinCard^{mm}(w) > 0$, then for all $z1, z2 \in E^m \cdot (z1 \vdash_{mm} w1) \wedge (z2 \vdash_{mm} w2)$ there must exist $z = (z1 \xrightarrow{lbl'} z2) \in E^m \cdot z \vdash_{mm} w$;

The relaxed version of this conformity relation (written $m \vdash_0 mm$) is similar to the presented conformity relation with the exception that condition (iii) is replaced by the following:
(iii) for all $z \in E^m$, there exists $w \in E^{mm}$ such that $z \vdash_{mm} \lceil w \rceil$.

The set of all models $m \in TG$ that conform with a given metamodel $mm \in MM$ is called M_{mm}.

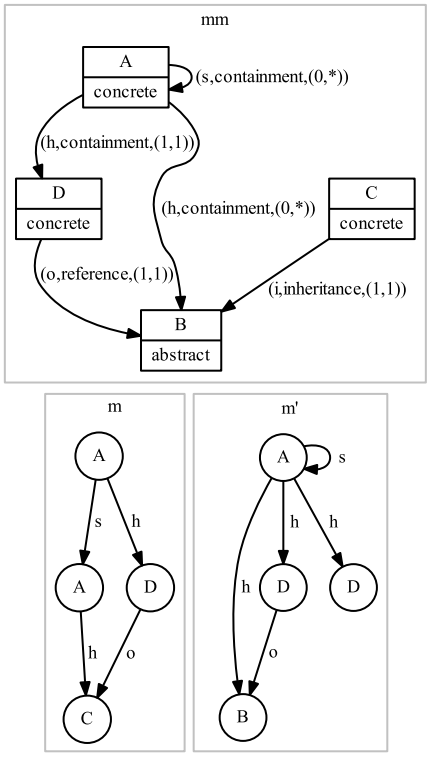

Figure 4.9: An example of a metamodel typed graph mm (on top), a typed graph m which conforms to mm (on the bottom left), and a typed graph m' which do not conforms to mm (on the bottom right).

We illustrate the above definitions with the example presented in Figure 4.9. There are several reasons from which typed graph m' do not conform to mm (or in other words $m' \not\vDash mm$):

1. The B labeled vertex on typed graph m' is marked as $abstract$ in metamodel mm. This violates the vertex-wise satisfaction shown in Definition 4.9.

2. The A labeled vertex on typed graph points to itself by means of an edge labeled s, which is marked as being a $containment$ edge in metamodel mm. This violates the model definition shown in Definition 4.11, that says that the containment graph of a model w.r.t. a metamodel must be acyclic.

3. There are two D labeled vertices on typed graph m' connected to the same A labeled vertex (m'_A) by means of two h labeled edges. This violates the cardinality satisfaction defined in Definition 4.12: $(m'_A, (mm_A \xrightarrow{label} mm_D)) \notin \#_{m',mm}$ because $MinCard^{mm}(mm_A \xrightarrow{label} mm_D) = MaxCard^{mm}(mm_A \xrightarrow{label} mm_D) = 1$, where in this case *label = (h,containment,(1,1))*. Also the sum of all edge labeled h starting in m'_A is 2. Consequently this also violates Definition 4.13.

4. The cardinality satisfaction $\#_{m',mm}$ is also violated when the D labeled vertex is not connect to the B labeled vertex, and in the metamodel mm, the minimum cardinality for the o labeled edges should be at least 1.

These metamodels can in principle be used to generate sentences on a given language. However, the complete set of models that conform to a metamodel of a language is typically infinite. Nevertheless, this conformance relation is merely syntactic, which means that the referred metamodels corresponds to the so called **abstract syntax of a language**, since they filter the structure and shape of valid expressions of that language by only looking to their explicit structure regardless of their implicit value/meaning.

4.2 Translational Semantics with the DSLTrans Language

In this section, we first introduce the language that we developed in order to define the semantics of a language by means of translations. Then, we formally describe both of its syntax and semantics. We conclude with implementation remarks on the tool support developed for this language, and how this approach can be used in order to automatically derive DSML compilers.

4.2.1 DSLTrans Overview

The translation presented in our motivating example, shown in Table 3.1, can be expressed using the DSLTrans language, as shown in Listing 4.1. DSLTrans, is a graph-based model transformation language that enables the specification of translations. These translations are expressed by means of groups of **rules** organized in a list of sequential **layers** — this means that the group of *rules* belonging to the first *layer* of a DSLTrans transformation is executed before every other groups, and so on. As in a regular graph-based model transformation language, these *rules* are formed by a left-hand-side graph (which we call the **match** model of the transformation *rule*), and by a right-hand-side graph (which we call the **apply** model of the transformation *rule*).

A transformation expressed in DSLTrans is formed by a set of input model sources called **file-ports** ($'model/input.xmi'$ in the Listing 4.1) and a list of *layers* ($'Entities'$ and $'Associations'$ *layers* in the Listing 4.1). Both *layers* and *file-ports* are typed according to **metamodels**. DSLTrans executes sequentially the list of *layers* of a transformation specification. A *layer* is a set of transformation *rules*, which executes in a non-deterministic fashion. Each transformation *rule* is a pair *(match,apply)* where *match* is a pattern holding elements from the source *metamodel*, and *apply* is a pattern holding elements of the target *metamodel*. In this textual syntax of DSLTrans, attributes are defined inside class elements as equations in the form $attributename = attributevalue$. We use the attribute name $'\_'$ to denote anonymous attributes. The anonymous attributes are intended to mark the apply class elements to remember which match class elements are responsible for their creation, in order to be further matched (in subsequent layers) by referenced apply class elements in the 'restrictions' section[1]. Notice that every element that is referenced in the 'restrictions' section is being matched instead of being created.

[1]In the DSLTrans graphical notation, the links written in the 'restrictions' section are represented as dashed lines (also called Backward Links or Restrictions) , and the links written in the 'subject to' section are represented as simple lines (Apply Links).

Listing 4.1: Translation StateMachine2PetriNet expressed in DSLTrans

```
File
  uri = 'model/input.xmi'
  metamodel(
        mmname = statemachine.Statemachine
        uri = 'model/StateMachine.ecore'
      )
def 'Entities': layer 'Entities'
 previous = '' output = ''
 metamodel(
     mmname = petrinet.Petrinet
     uri = 'model/PetriNet.ecore'
 )
rule'Transition'
  match with
  cl0:
      any  statemachine::Transition(  at0 :  name )
    apply
  cl1:
       petrinet::Transition(
            name= sameAs(at0)
            _= 'Transition'
          )
  end rule

  rule'State'
    match with
  cl2:
      any  statemachine::State(  at1 :  name )
    apply
  cl3:
       petrinet::Place(
            name= sameAs(at1)
            token= '0'
            _= 'State'
          )
  end rule

  rule'StateMachine'
    match with
  cl4:
      any  statemachine::StateMachine
    apply
  cl5:
       petrinet::PetriNet(
            _= 'PetriNet'
          )
  end rule

  rule'Initial'
    match with
  cl6:
      any  statemachine::Initial(  at2 :  name )
    apply
  cl7:
       petrinet::Place(
            name= sameAs(at2)
            token= '1'
            _= 'Initial'
          )
  end rule

end def

def 'Associations' : layer  'Associations'
  previous = 'Entities'
  output = 'model/output.xmi'
  metamodel(
      mmname = petrinet.Petrinet
      uri = 'model/PetriNet.ecore'
    )
  rule'source'
    match with
  cl8:
      any  statemachine::Transition
  cl9:
      any  statemachine::AbstractState
      subject to
        cl8 --(_source)-> cl9
    apply
  cl10:
       petrinet::Transition
  cl11:
       petrinet::Place
  cl12:
       petrinet::OutArc(
            weight= '1'
          )
      subject to
        cl10 --(outArc)-> cl12
        cl12 --(sourcePlace)-> cl11
    restrictions
      cl10 derived from cl8
      cl11 derived from cl9
  end rule

  rule'target'
    match with
  cl13:
      any  statemachine::Transition
  cl14:
      any  statemachine::AbstractState
      subject to
        cl13 --(_target)-> cl14
    apply
  cl15:
       petrinet::Transition
  cl16:
       petrinet::Place
  cl17:
       petrinet::InArc(
            weight= '1'
          )
      subject to
        cl17 --(targetPlace)-> cl16
        cl15 --(inArc)-> cl17
    restrictions
      cl15 derived from cl13
      cl16 derived from cl14
  end rule

  rule'states'
    match with
  cl18:
      any  statemachine::StateMachine
  cl19:
      any  statemachine::AbstractState
      subject to
        cl18 --(states)-> cl19
    apply
  cl20:
       petrinet::PetriNet
  cl21:
       petrinet::Place
      subject to
        cl20 --(places)-> cl21
    restrictions
      cl20 derived from cl18
      cl21 derived from cl19
  end rule

  rule'transitions'
    match with
  cl22:
      any  statemachine::StateMachine
  cl23:
      any  statemachine::Transition
      subject to
        cl22 --(transitions)-> cl23
    apply
      cl24:
       petrinet::PetriNet
      cl25:
       petrinet::Transition
      subject to
        cl24 --(transitions)-> cl25
    restrictions
      cl24 derived from cl22
      cl25 derived from cl23
  end rule

end def
```

In the example presented above, the language engineer wrote the first *layer* $Entities$ which specifies a direct 1 to 1 mapping between all the concepts of the State Machines language with some significant ones from the Petri Nets' language (e.g., in the rule $'State'$, from lines 25 to 36 on the left column, specifies that the $State$ concept is to be translated into $Place$). Then, the language engineer wrote the second *layer* $Associations$ specifying the translation between all the expressible (relevant) State Machine Language term compositions with Petri Net Language term compositions. These compositions may involve the introduction of new concepts from the target language. For instance, in order to translate the relation named $source$ (see from line 71 on the left column to the line 16 on the right column) between $State$ and $Transition$, the language engineer had to introduce the Petri Net concept of $OutArc$ (i.e., outgoing arc) while setting its **attribute** $weight$ value to 1.

In the transformation *rule* $'State'$ in the $'Entities'$ *layer* (see from lines 25 to 36 on the left column in the Listing 4.1) the *match* pattern holds one $'State'$ **class** from the $'statemachine'$ *metamodel* — the source *metamodel*; the *apply* pattern holds one $'Place'$ **class** from the $'petrinet'$ *metamodel* — the target *metamodel*. This means that in every execution of this transformation, all elements in the input source which are of type $'State'$ of the source *metamodel* will be transformed into elements of type $'Place'$ of the target *metamodel*. Let us first define the constructs available for expressing transformation *rules*' *match* patterns. We will illustrate the constructs by referring to the transformation in the Listing 4.1.

1. *Match Elements*: are variables typed by elements of the source *metamodel* which can assume as values elements of that type (or subtype) in the input model. In our example, a *match* element is the $'State'$ element in the $'State'$ transformation *rule* of *layer* $'Entities'$ layer;

2. *Attribute Conditions*: conditions over the attributes of a *match* element. For instance, in the Listing 4.1, one could specify in line 16 that the attribute 'at0 : name' should instead start by a string 'Sup'.

3. *Direct Match Links*: are variables typed by labelled relations of the source *metamodel*. These variables can assume as values relations having the same label in the input model. A *direct match link* is always expressed between two *match* elements. In the textual syntax presented in the Listing 4.1, direct match links are presented in the 'subject to' section inside the 'match with' section—for instance, in line 77 and 78.

4. *Indirect Match Links*: or simply indirect links, are labelled relations similar to *direct match links*, but there may exist a path of containment associations between the matched instances. In our DSLTrans' implementation, the notion of *indirect links* captures only EMF containment associations;

5. *Backward Links*: backward links connect elements of the *match* and the *apply* models. They exist in our example in all transformation *rules* in the *'Associations' layer*, shown in the *'restrictions'* section. *Backward links* are used to refer to elements created in a *previous layer* in order to use them in the current one. Notice that in the textual syntax presented in the Listing 4.1, each *layer* has a reference to a *previous layer*, or empty (see *previous* =" in line 8 of the left column) if we are otherwise defining the first *layer* of the transformation. An important characteristic of DSLTrans is that throughout all the *layers*, the source model remains intact as an input source. Therefore, the only possibility to reuse elements created from a *previous layer* is to reference them using *backward links*;

6. *Negative Conditions*: it is possible to express negative conditions over *match* elements, *backward*, *direct* and *indirect* match links. As these conditions are very similar to the positive ones, for simplicity reasons, we will not detail these on this thesis. For further details please consult the User Manual of the DSLTrans' reference implementation [2].

The constructs for building transformation *rules' apply* patterns are:

1. *Apply Elements and Apply Links*: *apply* elements, as *match* elements, are variables typed by elements of the target metamodel. *Apply* elements in a given transformation *rule* that are not connected to *backward links* will create elements of the same type in the transformation output. A similar mechanism is used for *apply* links. These output elements and links will be created as many times as the *match* model of the transformation *rule* is instantiated in the input model. In our example, the 'transitions' transformation *rule* of *layer* $'Associations'$ takes the instances $PetriNet$ and $Transition$ (belonging to the Petri Net language's *metamodel*) which have to be created before in a *previous layer* from existing instances of $StateMachine$ and $Transition$ (from the StateMachine language's *metamodel*), and connects them using a $'transitions'$ relation;

2. *Apply Attributes*: DSLTrans includes a small attribute language allowing the composition of attributes of *apply* model elements from references to one or more *match* model element *attributes*.

4.2.2 DSLTrans' Syntactic Structures

The abstract syntax of the DSLTrans language is defined by the BNF production rules shown in Listing 4.2. These production rules are able to produce/parse the sentences

[2]The DSLTrans manual is publicly available at: https://github.com/githubbrunob/DSLTransGIT/blob/master/DSLTransManual/document.pdf?raw=true

shown before. Moreover, the non-terminal symbols 'ExistsMatchClass' and 'Negative-MatchAssociation' were defined in the tool for the convenience of the DSLTrans' users, but are left from the formalization, and therefore are out of the scope of this thesis. It is important to notice however that the claims and conclusions taken in this thesis are not affected by these constraints, since they are considered to be particular cases of the general ones that will be further defined.

Listing 4.2: The DSLTrans' syntax expressed using the BNF notation

```
TransformationModel ::=  AbsSource* ;
AbsSource ::= FilePort | Layer ;
FilePort ::= "File"
             ("id" "=" Id)?
             ("uri" "=" Id)?
            MetaModelIdentifier ;

MetaModelIdentifier ::= "metamodel""("
                        ("mmname" "=" Id)?
                        ("uri" "=" Id)?
                    ")" ;

Layer ::=
    "def" (Id ":" ) ? ("layer" Id)?
    ("previous"= Id)?
    ("output" "=" Id)?
     MetaModelIdentifier
     (Rule)*
     "end" "def";

Rule ::= "rule" (String)?
            "match" MatchModel
            "apply" ApplyModel
            ("restrictions" BackwardLinks+)?
        "end" "rule";

MatchModel ::= "with" MatchClass* ("subject" "to" MatchAssociation*)?;
ApplyModel ::=  ApplyClass* ("subject" "to" ApplyAssociation*)? ;

MatchClass ::= AnyMatchClass | ExistsMatchClass;
AnyMatchClass ::= String? (Id":")?
                  "any" Id "::" Id ("(" (MatchAttribute)+ ")")? ;
ExistsMatchClass ::= String? (Id":")?
                  "existing" Id "::" Id ("(" (MatchAttribute)+ ")")?;
NegativeMatchClass ::= String? (Id":")?
                  "not" Id "::" Id ("(" (MatchAttribute)+ ")")?;

ApplyClass ::= String? (Id":")? Id "::" Id ("(" (ApplyAttribute)+ ")")? ;

MatchAssociation ::= PositiveMatchAssociation | NegativeMatchAssociation |
                  PositiveIndirectAssociation | NegativeIndirectAssociation;
PositiveMatchAssociation ::= Id "--" "(" Id ")" "->" Id;
NegativeMatchAssociation ::= Id "!-" "(" Id ")" "->" Id ;
PositiveIndirectAssociation ::= Id "~~" "~>" Id;
NegativeIndirectAssociation ::= Id "!~" "~>" Id;

ApplyAssociation ::= Id "--" "(" Id ")" "->" Id;

MatchAttribute ::=  (Id":")?  Id ("=" Atom)?  ;
ApplyAttribute ::=   (Id":")? Id ("=" Term)?  ;

BackwardLinks ::= PositiveBackwardLink | NegativeBackwardLink;
PositiveBackwardLink  ::= Id "derived" "from" Id;
NegativeBackwardLink ::= Id "not" "derived" "from" Id;

Term ::= AttributeRef | Atom | Concat;
Atom ::=  String;
AttributeRef ::= "sameAs" "(" Id ")" ;
Concat ::= "concat" "(" Term "with" Term ")";
```

For the sake of clarity and simplicity in the proofs, we will next introduce the following syntactic structures based on graph abstractions, which constitute a simplification of the ones presented above.

Definition 4.15. *Match-Apply Model*

A Match-Apply Model is a 7-tuple $\langle V,\ E,\ \tau_v,\ \tau_e,\ MatchPart,\ ApplyPart,\ Bl\rangle$*, where* $MatchPart = \langle Match, s\rangle$ *and* $ApplyPart = \langle Apply, t\rangle$*. Both* s *and* t *are metamodels, where* s *is called the* source *metamodel and* t *the target metamodel. Also we require that* $Match = \langle V', E', \tau_v', \tau_e'\rangle$ *is a model w.r.t.* s *and* $Match \vdash_0 s$*. Similarly,* $Apply = \langle V'', E'', \tau_v'', \tau_e''\rangle$ *is also a model w.r.t.* t *and* $Apply \vdash_0 t$*. And finally, we require that* $\langle V, E \setminus Bl, \tau_v, \tau_e\rangle = Match \sqcup Apply$*.*

It is always true that $\langle V, E \setminus Bl, \tau_v, \tau_e\rangle = Match \sqcup Apply$*. Edges* $Bl \subseteq V' \times V'' \subseteq E$ *are called* backward links*, where for all* $b \in Bl$*,* $\tau_e(b) = backwardlink$*.*

The set of all Match-Apply models for a source metamodel s *and a target metamodel* t *is called* MAM_t^s*.*

Vertices in the $Apply$ *model which are not connected to* backward links *are called* free vertices*.*

The $back : MAM_t^s \to MAM_t^s$ *function connects all vertices in the* $Match$ *model to all* free vertices*, by creating new* backward link *edges on the resulting Match-Apply model.*

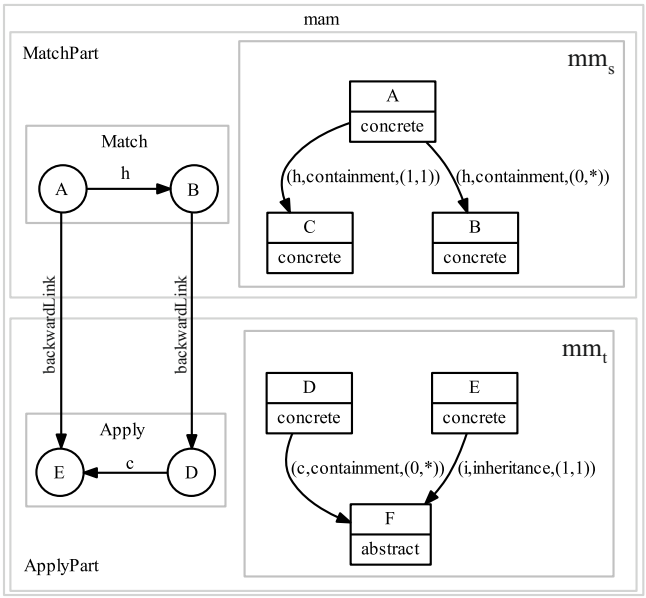

Figure 4.10: An example of a match apply model mam.

In Figure 4.10, we show an example of a match apply model, which represents an input graph of an A labeled vertex connected by an h labeled edge to a B labeled vertex. It also represents an E labeled vertex and a D labeled vertex, connected by a c labeled edge from the target output model. These vertices were previously created from the vertices connected to them by means of backwardLinks. Also notice that $Match \vdash_0 mm_s$ and $Apply \vdash_0 mm_t$, which means that here we do not require that these graphs comply with the minimum cardinality requirements from both the source and target metamodels of the transformation.

Definition 4.16. *Transformation Rule*

A Transformation Rule is a 8-tuple $\langle V, E, \tau_v, \tau_e, MatchPart, ApplyPart, Bl, Il \rangle$*, where* $MatchPart = \langle Match, s \rangle$ *and* $ApplyPart = \langle Apply, t \rangle$*.*

Let $MatchNoIl = \langle V', E' \setminus Il, \tau'_v, \tau'_e \rangle$ *be a model w.r.t.* s*,*
and $MatchPartNoIl = \langle MatchNoIl, s \rangle$*, then it is true that* $\langle V, E \setminus Il, \tau_v, \tau_e, MatchPartNoIl, ApplyPart, Bl \rangle \in MAM_t^s$ *is a match-apply model.*

It is also true that $Match = \langle V'', E'', \tau''_v, \tau''_e \rangle$*, and the edges* $Il \subseteq E'' \subseteq E$ *are called* indirect links*, which means that for all* $i \in Il$ *it is true that* $\tau_e(i) = \tau''_e(i) = indirectlink$*.*

The set of all transformation rules is called TR_t^s*.*

A Transformation Rule Specification is a relaxed version of a Transformation Rule, where we no longer require that $MatchPart = \langle Match, s \rangle$ *and* $ApplyPart = \langle Apply, t \rangle$*. The set of all transformation rule specifications is called* $SpecTR_t^s$*.*

We have just defined a *transformation rule* as a kind of match-apply model which allows *indirect links* only in the $match$ pattern.

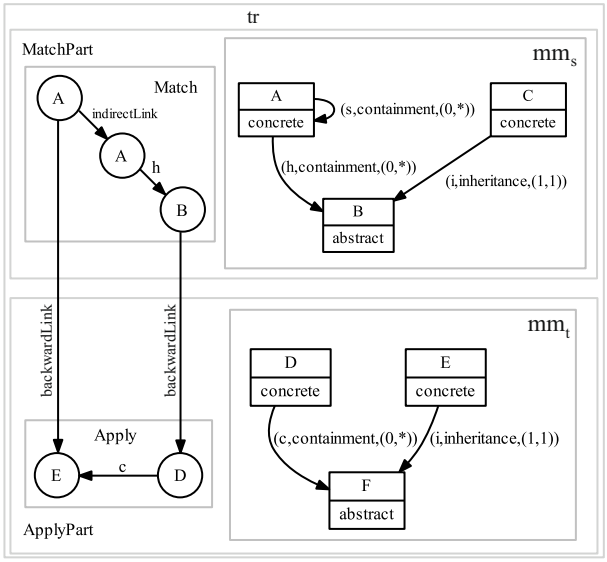

Figure 4.11: An example of a transformation rule tr.

As shown in Figure 4.11, the tr typed graph could be considered as a match apply model except for the *indirectLink* labeled edge. We will further see that the edge labeled *indirectLink* causes the match procedure to not only match (from a given input model) A elements that are connected together by means of a containment edge, but also A elements that are connected to other elements (possibly of other types) in a chain of containment edges, ending (in this case particular) in another element of type A. Notice also that despite the fact that element B is abstract, it can however be represented in the MatchPart of th rule tr: the idea is that it will match all the elements from a given input model that inherits from element B, namely elements of type C.

Definition 4.17. *Layer*

A layer is a finite set of transformation rule specifications $tr \subseteq SpecTR_t^s$. The set of all layers for a source metamodel s and a target metamodel t is called $Layer_t^s$.

Definition 4.18. *DSLTrans Transformation*

Let $dsltrans \in MM$ be the DSLTrans' metamodel. A DSLTrans Transformation $m \in TG$ is a model $m \vdash dsltrans$ containing a finite list of layers denoted $m = [l_1 :: l_2 :: \ldots :: l_n]$ where $l_k \in Layer_t^s$ and $1 \leq k \leq n$. The set of all transformations for a source metamodel s and a target metamodel t is called $Transformation_t^s$.

In definition 4.18, we just defined DSLTrans' metamodel as a mathematical concept that aggregates all the DSLTrans' syntactic structures defined also as mathematical concepts. The mathematical semantics of these syntactic structures will be defined in the next subsection, however, as further reference, the reader can take a look to Figure 4.16 where it is presented the metamodel used in DSLTrans' actual implementation. This implementation metamodel is coherent with both this mathematical conceptualisation and the BNF form presented in Listing 4.2.

4.2.3 DSLTrans' Semantics

We will now define the DSLTrans' Semantics by using all the defined DSLTrans' syntactic structures presented above.

Definition 4.19. *Strip Function*

The $strip : TR_t^s \rightarrow TR_t^s$ function removes from a transformation rule all free vertices and associated edges. Formally, the strip function is such that $strip(\langle V, E, \tau_v, \tau_e, \langle \langle V_m, E_m, \tau_{v_m}, \tau_{e_m} \rangle, s \rangle, \langle \langle V_a, E_a, \tau_{v_a}, \tau_{e_a} \rangle, t \rangle, Bl, Il \rangle) = \langle V', E', \tau_v, \tau_e, \langle \langle V_m, E_m, \tau_{v_m}, \tau_{e_m} \rangle, s \rangle, \langle \langle V'_a, E'_a, \tau_{v_a}, \tau_{e_a} \rangle, t \rangle, Bl, Il \rangle$, where the following conditions are satisfied:

1. *$V'_a \subseteq V_a$ such that for all $x_t \in V'_a$ there exists at least one $x_s \xrightarrow{backwardLink} x_t \in Bl$. This means that all nodes in the apply model are connected to at least one node in the match model by means of a* backward link*;*
2. *$E'_a = E_a|_{V'_a}$.*

The strip function is illustrated in Figure 4.12, which when applied to transformation rule tr (on the left) returns the transformation rule $strip(tr)$ (on the right) by removing all of the vertices from the $ApplyPart$ that were not connected to $MatchPart$ by means of $backwardLink$ labeled edges.

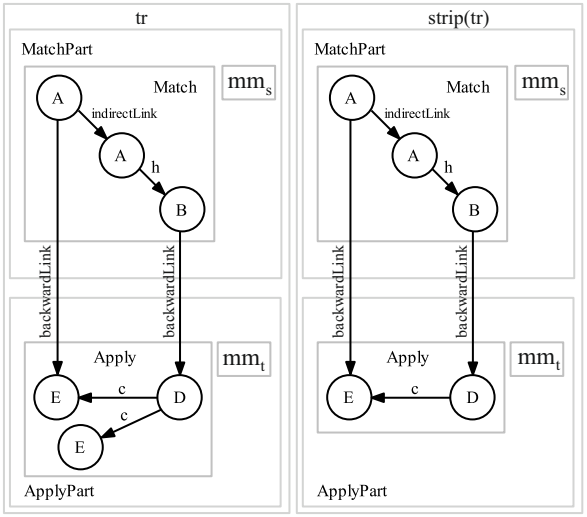

Figure 4.12: An example of a transformation rule tr and the result of applying the $strip$ function on it.

Definition 4.20. *Subgraph of a Match-Apply Model*

Let $tr = \langle V, E, \tau_v, \tau_e, MatchPart, ApplyPart, Bl, Il \rangle \in TR_t^s$ *be a transformation rule, and* $mam = \langle V', E', \tau_v', \tau_e', MatchPart', ApplyPart', Bl' \rangle \in MAM_t^s$ *be a match-apply model, where* $MatchPart' = \langle Match, s \rangle$*, and* $Match$ *is a model w.r.t. metamodel* s*.*

We define that tr *is a subgraph of* mam *(written* $tr \lhd mam$*) if and only if the following conditions are satisfied:*

1. $\langle V, E \setminus Il, \tau_v, \tau_e \rangle \blacktriangleleft \langle V', E', \tau_v', \tau_e' \rangle$
2. *for all* $x_s \xrightarrow{indirectLink} x_t \in Il$*, there exists exactly one* $x_s' \xrightarrow{lbl} x_t' \in E_c^*$ *where* $\tau_v(x_s) = \tau_v(x_s')$*,* $\tau_v(x_t) = \tau_v(x_t')$ *and* E_c^* *is obtained by the transitive closure of* $E_c = \{y = (y_s \xrightarrow{lbl'} y_t) \in E' | Kind^s_{Match}(y) = containment\}$*.*

As an example, in Figure 4.13, both tr and tr' (on the bottom of the figure) are considered to be subgraphs of the presented match-apply graph mam (presented on the top of the Figure). On the one hand, if we neglect the $indirectLink$ edges, it is easy to see that both tr and tr' are typed subgraphs of typed graph mam, which satisfies the first condition of Definition 4.20. On the other hand, in both tr and tr', the $indirectLink$ labeled edge can be replaced by the transitive closure of s labeled edges in typed graph mam, which satisfies the second condition of Definition 4.20.

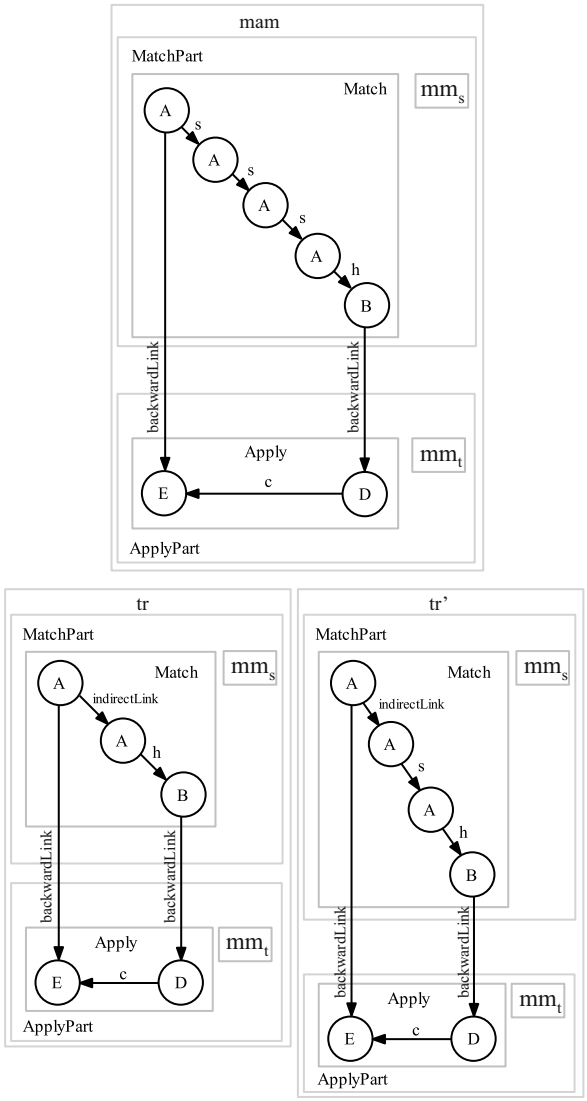

Figure 4.13: Both tr and tr' transformation rules are subgraphs of match-apply graph mam.

Definition 4.21. *Match Function*

Let $m \in MAM_t^s$ be a model and $tr \in TR_t^s$ be a transformation rule. The $match : MAM_t^s \times TR_t^s \to \mathcal{P}(MAM_t^s)$ is defined as follows:

$$match_{tr}(m) = remove\big(\{g \mid g \lhd m \wedge g \cong strip(tr)\}\big)$$

Due to the fact that the $\cong$ relation is based on the notion of graph isomorphism, permutations of the same match result may exist in the $\{g \mid g \lhd m \wedge g \cong strip(tr)\}$ set. Notice also that despite g being a transformation rule (i.e., $g \in TR_t^s$), the function $match$ only returns a set of MAM_t^s, which means that the indirect link information was implicitely removed. The — undefined — $remove : \mathcal{P}(TR_t^s) \to \mathcal{P}(TR_t^s)$ function is such that it removes such undesired permutations.

Definition 4.22. *Apply Function*

Let $m \in MAM_t^s$ be a match-apply model and $tr \in TR_t^s$ a transformation. The $apply$: $MAM_t^s \times TR_t^s \to MAM_t^s$ is defined as follows:

$$apply_{tr}(m) = \bigsqcup_{g \in match_{tr}(m)} back(g \sqcup g_\Delta)$$

$$\textit{where } g_\Delta \textit{ is such that } g \sqcup g_\Delta \cong tr$$

The freshly created vertices of g_Δ in the flattened $apply_{tr}(m)$ set are disjoint.

Definitions 4.21 and 4.22 are complementary: the former gathers all subgraphs of a match-apply graph which match a transformation rule; the latter builds the new instances which are created by applying that transformation rule as many times as the number of subgraphs found by the $match$ function. The $strip$ function is used to enable matching over backward links but not elements to be created by the transformation rule. The $back$ function (defined in Definition 4.15) connects all newly created vertices to the elements of the source model that originated them. Therefore, the apply function calls the match function defined for transformation rule tr, and for each of the match results (g) it computes what must be created (g_Δ) by looking again to the remaining information in the transformation rule tr.

At the transformation level, the transformation rules have to be unfolded into several ones according to the elements present in the match pattern and their inheritance relations on their respective match metamodel.

Definition 4.23. *Inheritance Unfold*

Let $tr = \langle V, E, \tau_v, \tau_e \rangle \in SpecTR_t^s$ be a transformation rule specification. The function $\uparrow$: $SpecTR_t^s \to \mathcal{P}(TR_t^s)$ is such that $\uparrow_{tr}$=$\{\langle V, E, \tau'_v, \tau_e \rangle \in TR_t^s\}$, where the new typing function on the vertices $\tau'_v \in \mathcal{P}(V \to \Sigma_v)$ is such that each

$$\begin{aligned} \tau'_v = \big\{ (v_i \to Name^s(x_i)) \mid \tau_v = \bigcup_{i=1}^{n} (v_i \to t_i) \wedge \\ \big(\exists\, x_i, y_i \in V^s \,.\, (Name^s(y_i) = t_i) \wedge \\ (x_i \leq_s y_i) \wedge \\ (Kind^s(x_i) = concrete) \big) \big\}, \end{aligned}$$

where n is the number of tuples in the function τ_v.

Let $l \in Layer_t^s$ be a layer. We define the function $\uparrow$: $Layer_t^s \to \mathcal{P}(TR_t^s)$ such that:

$$\uparrow_l = \bigcup_{tr \in l} \uparrow_{tr}$$

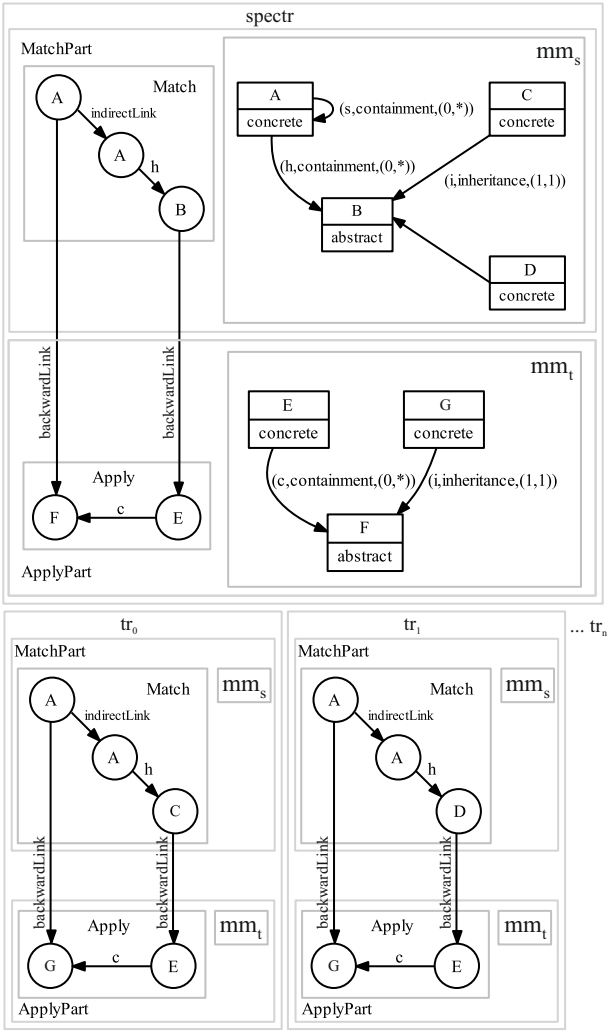

Figure 4.14: The unfold function ↑ when applied to $spectr$ returns a set of n transformation rules, where no abstract classes are found in both Match and Apply patterns.

Note that the new typing function do not possess anymore the ability to reference abstract classes, which means that our unfolded rules will only have concrete classes in both match and apply parts. Moreover, the result is a set of transformation rules, which means that each one of the MatchPart and ApplyPart have now to be conforming with the respective source and target metamodels of the transformation—in particular to the relaxed version of the conformity relation; $MatchPart = Match \vdash_0 s$ and $ApplyPart = Apply \vdash_0 t$.

The application of the ↑ function is illustrated in Figure 4.14. Here we applied the ↑ function to an abstract transformation rule called $spectr$. The result is a set of n transformation rules, where all of the vertices from both of the $Match$ and $Apply$ typed graphs have concrete labels w.r.t. their respective metamodels mm_s and mm_t. Notice that in general there might be several combinations of this unfolding. The ↑ function explores

all of the possible combinations of turning abstract labels into concrete labels.

Definition 4.24. *Layer Step Semantics*

Let $l \in \mathcal{P}(TR_t^s)$ be a finite set of transformation rules. The layer step relation $\overset{layerstep}{\rightarrow} \subseteq MAM_t^s \times MAM_t^s \times \mathcal{P}(TR_t^s) \times MAM_t^s$ *is defined by the minimum set that satisfies the following rules:*

$$\frac{}{\langle m, m', \emptyset \rangle \xrightarrow{layerstep} m \sqcup m'}$$

$$\frac{tr \in l,\ apply_{tr}(m) = m''', \langle m, m'' \sqcup m''', l \backslash \{tr\} \rangle \xrightarrow{layerstep} m'}{\langle m, m'', l \rangle \xrightarrow{layerstep} m'}$$

where $\{m, m', m''\} \subseteq MAM_t^s$ are match-apply models.

The freshly created vertices in m''' are disjoint from those in m''.

For each layer we go through all the transformation rules and build for each one of them the set of new instances created by their application. These instances are built using the $apply$ function in the second rule of definition 4.24. The new instance results of the $apply$ function for each transformation rule are accumulated until all transformation rules are treated. Then, the first rule of definition 4.24 will merge all the new instances with the starting match-apply model. The merge is performed by uniting (using the non-disjoint $\sqcup$ union) match-apply graphs including the new instances with the starting match-apply model.

Definition 4.25. *Transformation Step Semantics*

Let $[layer :: R] \in Transformation_t^s$ be a Transformation, where $layer \in Layer_t^s$ is a Layer and R a list. The transformation step relation $\overset{trstep}{\rightarrow} \subseteq MAM_t^s \times Transformation_t^s \times MAM_t^s$ *is defined as follows:*

$$\frac{}{\langle m, [] \rangle \xrightarrow{trstep} m}$$

$$\frac{\left\langle m, \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle, \uparrow_{layer} \right\rangle \xrightarrow{layerstep} m'',\ \langle m'', R \rangle \xrightarrow{trstep} m'}{\langle m, [layer :: R] \rangle \xrightarrow{trstep} m'}$$

where $\{m, m', m''\} \subseteq MAM_t^s$ are match-apply models.

A model transformation is a sequential application of transformation layers to a match-apply model containing the source model and an empty apply model. The transformation output is the apply part of the resulting match-apply model.

Definition 4.26. *Model Transformation*

Let $\{s,t\} \subset MM$ *be metamodels,* $m_s \in M_s$ *and* $m_t \in M_t$ *be models and also let* $tr \in Transformation_t^s$ *be a transformation.*

A model transformation $\overset{transf}{\rightarrow} \subseteq M_s \times Transformation_t^s \times M_t$ *is defined as follows:*

$$\langle m_s, tr\rangle \xrightarrow{transf} m_t \Leftrightarrow$$

$$\langle\langle V, E, \tau_v, \tau_e, \langle m_s, s\rangle, \langle\emptyset, t\rangle, \emptyset\rangle, tr\rangle \xrightarrow{trstep} \langle V', E', \tau_v', \tau_e', \langle m_s, s\rangle, \langle m_t, t\rangle, Bl\rangle$$

Notice that in Definition 4.26, the match apply graph $\langle V, E, \tau_v, \tau_e..\rangle$ (recall Definition 4.15) induced from m_s is changed after the execution of the transformation tr into another match apply graph $\langle V', E', \tau_v', \tau_e'..\rangle$ which now includes the output model m_t, despite the fact that the source input model m_s remains intact. Also by Definition 4.15 we know that we always start executing the transformation tr knowing that $m_s \vdash_0 s$ (i.e., it is conformant with), and in the end of the transformation execution we also know that $m_t \vdash_0 t$.

4.2.4 DSLTrans' Language Properties

We now present two important properties about DSLTrans' transformations, and their respective proofs.

Proposition 4.27. *Confluence*

Every model transformation is confluent regarding typed graph equivalence.

Proof. (Sketch) We want to prove that for every transformation $tr \in Transformation_t^s$ having as input a model $m_s \in M_s$, if $\langle m_s, tr\rangle \xrightarrow{transf} m_t$ and $\langle m_s, tr\rangle \xrightarrow{transf} m_t'$ then $m_t \cong m_t'$. Note that we only have to prove typed graph equivalence between m_t and m_t' because the identifiers of the objects produced by a model transformation are irrelevant. If we assume $\neg(m_t \cong m_t')$ then this should happen because of non-determinism points in the rules defining the semantics of a transformation:

1. in definition 4.22 g_Δ is non-deterministic up to typed graph equivalence, which does not contradict the proposition;
2. in definition 4.24 transformation rule tr is chosen non-deterministically from layer l.

Thus, the order in which the transformation rules are treated is non-deterministic. However, the increments to the transformation by each rule of a layer are united using $\sqcup$, which is commutative and thus renders the transformation result of each layer deterministic. Since there are no other possibilities of non-determinism points in the semantics of a transformation, $\neg(m_t \cong m_t')$ provokes a contradiction and thus the proposition is proved. □

Proposition 4.28. *Termination*

Every model transformation terminates.

Proof. (Sketch) Let us assume that there is a transformation which does not terminate. In order for this to happen there must exist a section of the semantics of that transformation which induces an algorithm with an infinite amount of steps. We identify three points of a transformation's semantics where this can happen:

1. if definition 4.25 induces an infinite amount of steps. The only possibility for this to happen is if the transformation has an infinite amount of layers, which is a contradiction with definitions 4.17 and 4.18;

2. if definition 4.24 induces an infinite amount of steps. The only possibility for this to happen is if a layer has an infinite amount of transformation rules, which is a contradiction with definition 4.16;

3. if the result of the $match_{tr}(m)$ function in definition 4.21 is an infinite set of match-apply graphs. The match-apply graph m is by definition finite, thus the number of isomorphic subgraphs of m is infinite only if the transitive closure of containment edges of m is infinite. The only possibility for this to happen is if the graph induced by the containment edges of m has cycles, which contradicts definition 4.1.

Since there are no more points in the semantics of a transformation that can induce an infinite amount of steps, the proposition is proved.

□

It is important to notice that these two properties (Confluence and Termination) could also be achieved using other model transformation languages such as EMF Tiger or ATL. In such languages one could for instance devise an analysis algorithm that checks these conditions. Our approach in DSLTrans is that these properties are necessary conditions for the analysability of model transformation specifications and therefore they should be enforced by construction in the model transformation language itself. Moreover in the perspective of the model transformation engineer, it is much more convenient to specify analysable model transformations by construction using DSLTrans than to reengineer an existing ATL model transformation in order to make it analysable. Clearly, the enforcement of these properties by construction in DSLTrans brings along drastic consequences on its expressiveness. While comparing the expressiveness of DSLTrans with other model transformation languages such as ATL, we conclude that on the one hand with ATL we can design the same translation in much more different ways than we could by using DSLTrans—in this case however we can say that DSLTrans specifications are more straightforward and easier to understand/read; and on the other hand there are some translations that despite the fact of being expressible in ATL, are simply not expressible using a single DSLTrans specification. In some cases we needed to devise a

chain of several DSLTrans specifications, or even use another language to perform the translation—e.g., when we need to perform complex calculations while interpreting the meaning of values of the syntactic structures of the source model, or when we need to generate unique identifiers on the target model.

4.2.5 DSLTrans' Tool Support

The implementation of the DSLTrans language, involved the implementation of several supporting tools, ranging from edition tools, to execution engines and analysis tools. Moreover, all of the described tools were implemented and deployed as Eclipse plugins based on the Eclipse Modeling Framework (EMF). At this point, we will only describe the DSLTrans' editors and the DSLTrans' execution engine [3], and leave the developed analysis tools to be described in the following Chapter. Figure 4.15 shows how one of the DSLTran's visual editors interacts with the DSLTrans'execution engine in order to translate input models into output models, where the depicted numbers inside circles denote a logical order of events in time. Here we consider two different actors: *(i)* the software language engineer (SLE) produces a DSLTrans model of his/her DSL using the DSLTrans Editor; and *(ii)* the DSL user that writes his/her models expressed in the built DSL.

In what matters to the edition tools, two different editors for the DSLTrans language were implemented: a visual editor and a textual editor. Both of the editors were automatically generated based on a common description of the language syntax (i.e., its abstract syntax) expressed by means of an EMF metamodel as shown in Figure 4.16.

In particular, the DSLTrans visual editor plugin, was automatically generated using an Eugenia/GMF project [4] by annotating the metamodel presented in Figure 4.16 with visual concrete syntax directives. The definition of these directives allowed for instance to define which entities will be nodes, and which entities will be arrows, in the generated diagrammatic edition panel of the DSLTrans visual editor.

The DSLTrans visual editor allows the edition of syntax-to-syntax translations expressed in DSLTrans in a graphical way. Figure 4.17 shows an example of the graphical view of the transformation from StateMachines Language to the Petri Nets Language presented in Listing 4.1.

Moreover, the DSLTrans textual editor plugin was automatically generated using an EMFText [5] Project, also by annotating the metamodel presented in Figure 4.16 with textual concrete syntax directives. In this case, the annotations were expressed in the Concrete Syntax Specification Language (CS), which is a language very similar to BNF

[3] All of the DSLTrans tools are publicly available at: https://github.com/githubbrunob/DSLTransGIT/blob/master/DSLTrans-Release/DSLTrans-Suite-06062k12.zip?raw=true

[4] http://www.eclipse.org/epsilon/doc/eugenia/

[5] http://www.emftext.org/index.php/EMFText

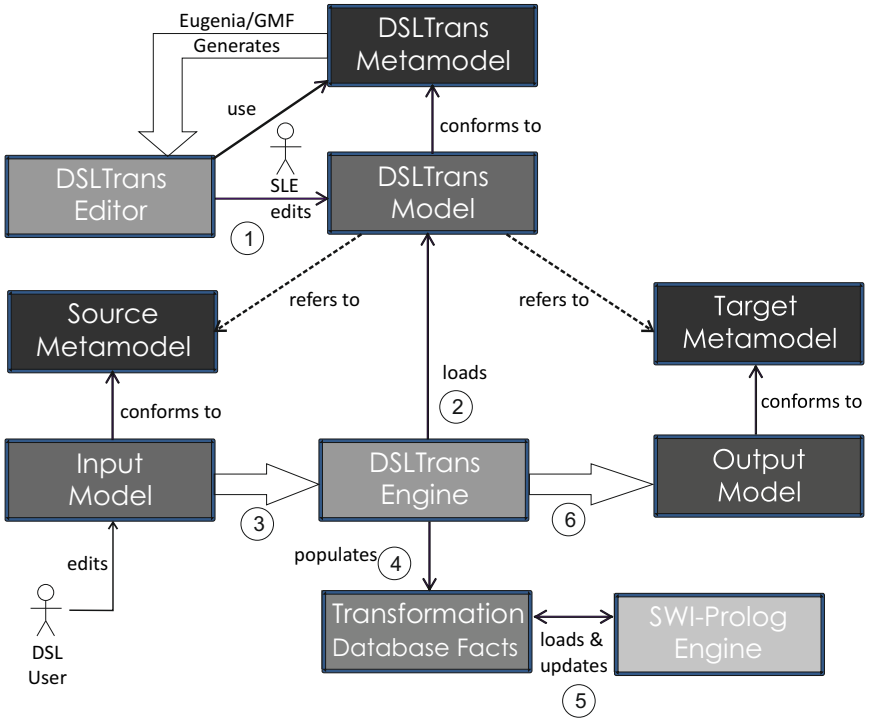

Figure 4.15: The reference implementation of DSLTrans as a set of Eclipse plug-ins.

strongly typed with the types of the referenced metamodel (in our case the DSLTrans metamodel). The resulting textual editor has already (by default) the syntax-highlighting capabilities deduced from the .cs specification, and enabled the edition of the example presented in Listing 4.1.

Both of these edition plugins have an additional (by default) capability of producing an XML/XMI version of the edited translations expressed in DSLTrans, regardless if they are textual or visual.

In what matters to the DSLTrans' execution, the DSLTrans' transformation engine plugin was fully coded in Java while using both of the EMF, and the SWI-Prolog API. On the one hand, the EMF API was used in order to read the XML/XMI files corresponding to the transformation specification expressed in DSLTrans; and in order to read/write the XML/XMI files corresponding to the inputs and outputs of the model transformation. The transformation specification expressed in DSLTrans (in an XMI/XML format) is parsed (using the EMF API), and instantiates in memory an abstract syntax tree (AST) which class definitions were also previously automatically generated from the DSLTrans metamodel. While in execution, the input models referenced in the AST, are then loaded (currently only EMF based XMI/XML formats are supported) into memory, and converted into an internal relational representation. On the other hand, the SWI-Prolog API was used in order to instantiate the SWI-Prolog engine in an embedded prolog program. The internal relational representation of the input models, is then converted to prolog

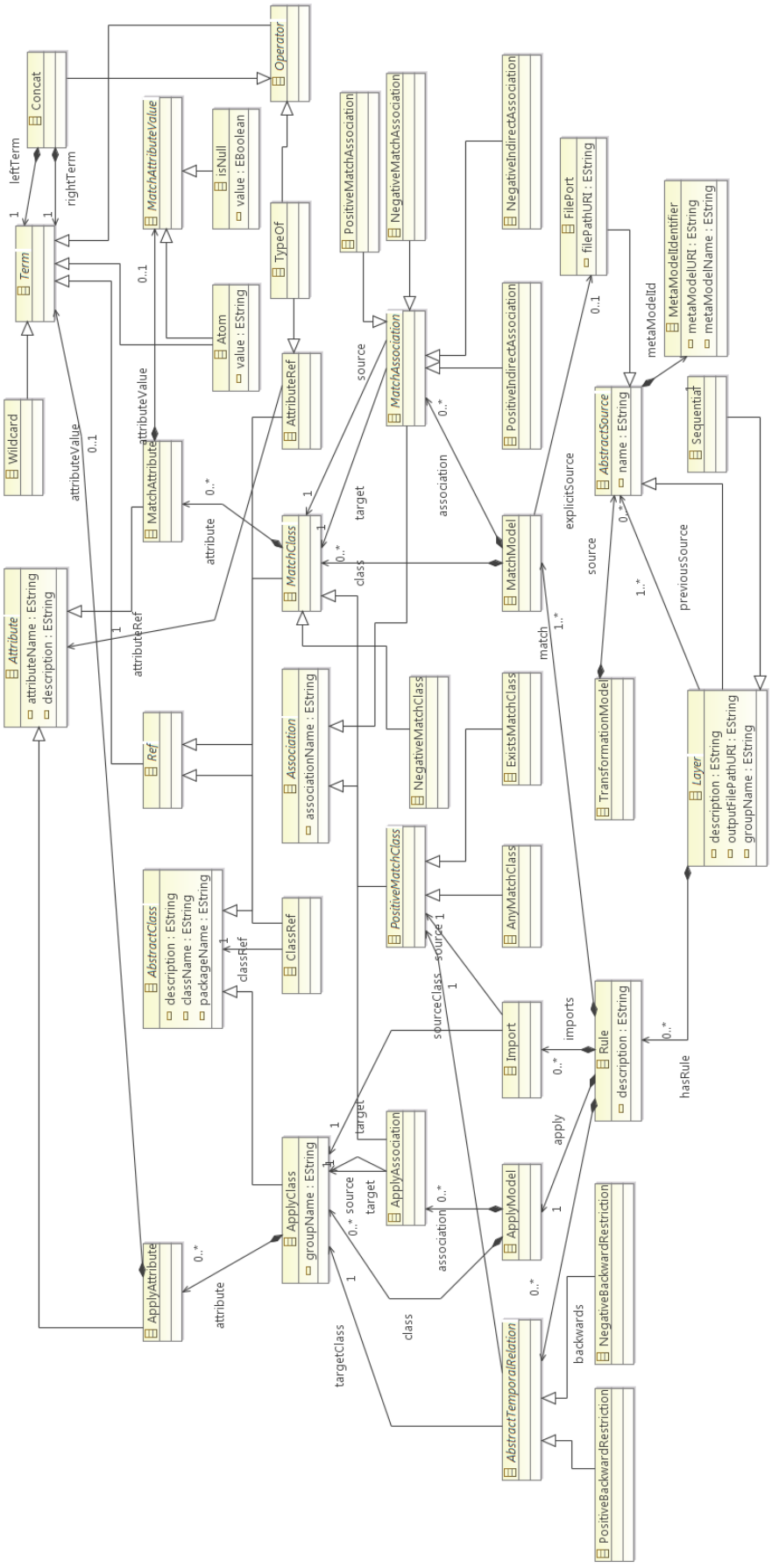

Figure 4.16: The DSLTrans Metamodel.

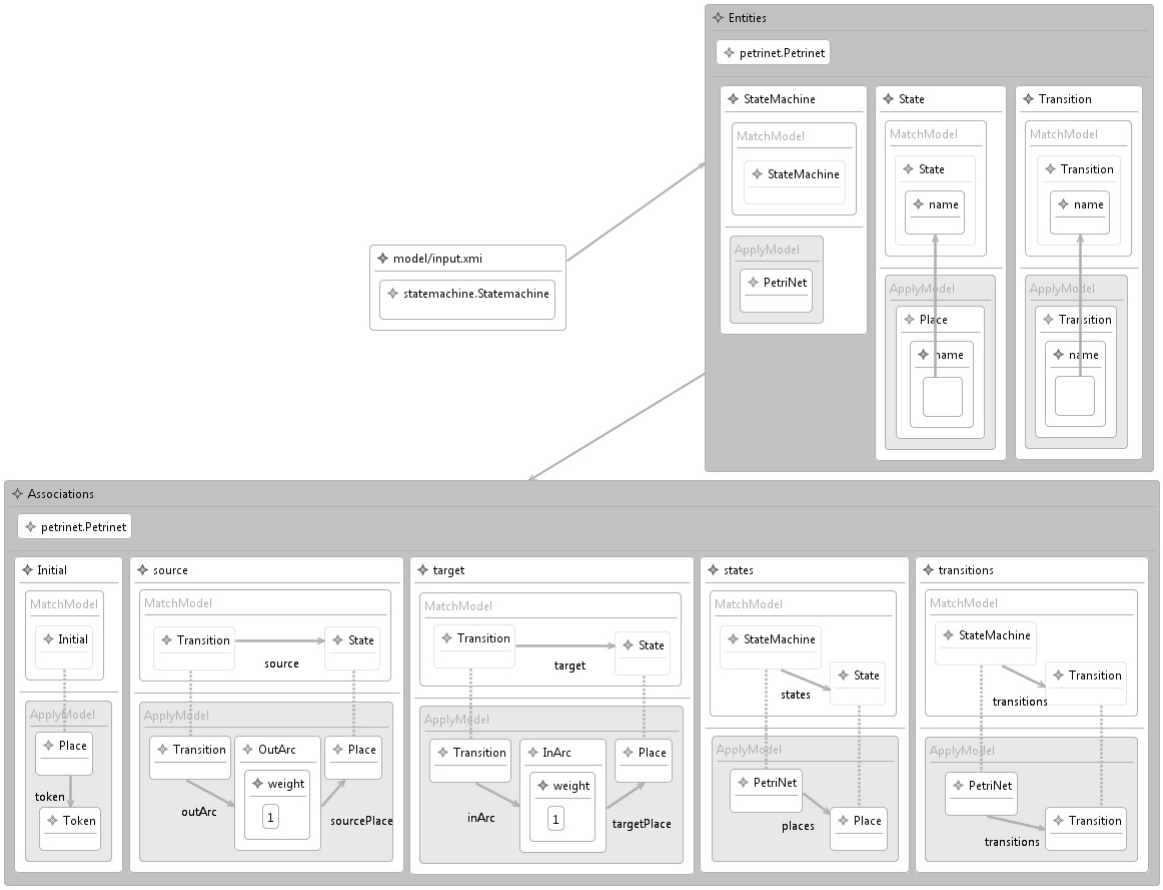

Figure 4.17: A visual representation of the StateMachines to Petri Nets translation, presented in Listing 4.1.

facts that are loaded into the instantiated prolog program's facts base. The match patterns specified in the AST leafs (corresponding to the match rules of the specified translation) are also translated into prolog clauses representing queries to be further applied on the instantiated prolog program's facts base.

The problem of executing the matches in a DSLTrans translation specification can therefore be mapped into a constraint satisfaction problem, where the prolog engine is left to find the solutions. For each solution found, the respective apply pattern in the associated translation rule is executed by instantiating the specified output classes from the target metamodel in memory. Notice that the definitions of the target classes were automatically generated by using the information in the target metamodel. Currently, only EMF-XML/XMI versions of both of the inputs and outputs are supported in DSLTrans implementation—the main limitation here is that both of the source language and target language editors, should be able to produce and deliver files in this particular format, in order to be processed by this transformation engine.

Concluding, the DSLTrans execution engine is able to execute any translation specification expressed in DSLTrans, in an equivalent way such as a DSML compiler would do

if it would be implemented based on the very same syntax-to-syntax translation specification. This results from the fact that this execution engine was implemented having the reference semantic model of DSLTrans. Moreover, since we know that this reference semantic model have the properties of confluence and termination (i.e., for any given input model, every execution run will eventually terminate, returning a finite and unique output model), we also know that the implemented DSLTrans execution engine also shares these properties.

4.3 Operational Semantics with the SOS Language

In this section, we first introduce the language that we developed in order to define the operational semantics of a language. The main goal to achieve in this part of the research work is to produce a language where software language engineers are able to design the meaning of the sentences of his/her own DSL in an appropriate and comprehensive way. In other words we should strictly follow the principles of domain specific modeling in the design and development of this language that we will call SOS, which stands for Structured Operational Semantics.

However there are already many ways of designing the operational semantics of languages, we still have the need to have a domain specific SOS language that is able to to express the operational semantics of DSLs. On the one hand, amongst the MDD community, MTLs are the the most popular languages to express and devise the operational semantics of languages, typically by means of rewriting in-place transformation rules that manipulate inputs as graphs (i.e., transformation rules based on graph grammars)—depending on the used MTLs, these specifications can be used in order to for instance animate and simulate the computational behaviour of a given sentence expressed in the specified DSL. Here we can argue that MTLs are used for so many different things that a software language engineer can have serious problems understanding the intent and meaning of a given operational semantics specification using that MTL. Moreover, it is not easy to extract properties about the language under design just by analysing a specification written in an MTL which was designed to specify arbitrary model transformations. On the other hand, programming language gurus and theoreticians are already used to use mathematical algebraic/set theory and inference rules in order to specify the operational semantics of their languages. Despite the fact that these specifications can be used to manually derive proofs about the properties of the DSL in question, they can serve only as a reference model to possible implementations by means of compilers or interpreters—i.e., there is a gap between the semantic models of the DSLs and their actual implementations. Furthermore, in the next chapter, we will use as oracles the specifications expressed in the SOS language, in order to be able to automatically decide about the semantic correctness of a given language translation expressed in DSLTrans.

In the remaining of this Chapter we formally describe both syntax and semantics of the SOS language for operational semantics, and conclude with implementation remarks about the tool support developed for this language.

4.3.1 The SOS Language Overview

The SOS language is able to specify the operational semantics definitions presented in Chapter 3. In particular, the SOS rules presented for the State Machine Language are shown in Listing 4.3. A typical SOS specification is composed by a set of rules in the form *Assuming*, *Then*, and *Where*. The rule preconditions are placed in the *Assuming* section and typically refer to the source metamodel syntactic structures or to the current execution state of the abstract machine that is interpreting a given input model. The rule post-condition (also called conclusion) is placed in the *Then* section, and it defines what is the next current state of the interpreting abstract machine. Therefore, on each execution, these rules rewrite the current state of the machine into a new one, while creating a new transition in the Transition System.

Listing 4.3: SOS definition for the State Machine Language

```
Assuming
  in(@t,Model)=true,
 in(@i,Model)=true,
 in(@s0,Model)=true,
 in(@t -> source -> @i, Model)=true,
 in(@t -> target -> @s0, Model)=true
Then
 initial(@i) ->> buildString(@t.name) ->>
  state(@s0) in Transition_System
Where
 i : class("statemachine","Initial");
 s0 : class("statemachine","AbstractState");
 t : class("statemachine","Transition");

Assuming
 state(@s0) ->> @nameX ->>
  state(@s1) in Transition_System,
 in(@t,Model)=true,
 in(@s1,Model)=true,
 in(@s2,Model)=true,
 in(@t -> source -> @s1, Model)=true,
 in(@t -> target -> @s2, Model)=true
Then
 state(@s1) ->> buildString(@t.name) ->>
  state(@s2) in Transition_System
Where
 s0 : class("statemachine","AbstractState");
 s1 : class("statemachine","AbstractState");
 s2 : class("statemachine","AbstractState");
 t : class("statemachine","Transition");
 nameX : string;
```

Listing 4.4: SOS ADT definition for the State Machine Language's semantic domain

```
ADT CurrState
 Sorts cs
 Generators
  state : class("statemachine","State") -> cs;
  initial: class("statemachine","Initial") -> cs;
 Operations
  equals: cs cs -> bool;
 Axioms
  equals(@x1 @x1) = true;
  (@x1 != @y1) => equals(@x1 @y1) = false;
 Where
  x1:cs;
  y1:cs;
```

The state values of the interpreting abstract machine are defined as abstract algebraic data types defined on the *ADT* section in the form *Generators*, *Operations*, *Axioms*, and *Where*. While the generators define the syntactic structure of the defining type, the defined operations are functions that can be used to manipulate the defined structures. The axioms are rewrite rules that are used to define the meaning of operations—i.e., how can a given operation be rewritten in order to properly perform its intended function.

Listing 4.5: SOS definition for the Petri Net Language

```
Assuming
 in(@t,Model)=true,
 positive(subtract(
    build(pre(@t))
    build(initial)
 ))=true
Then
 build(initial) ->>
      buildString(@t.name) ->>
             add(
              build(pos(@t))
              subtract(
               build(pre(@t))
               build(initial)
              )
             ) in Transition_System
Where
  t : class("petrinet","Transition");

Assuming
 @oldmarking ->> @nameX ->>
  @newmarking in Transition_System,
 in(@t,Model)=true,
 positive(subtract(build(pre(@t)) @newmarking))=true
Then
 @newmarking ->>
     buildString(@t.name) ->>
            add(
             build(pos(@t))
             subtract(
              build(pre(@t))
              @newmarking
             )
            ) in Transition_System
Where
 newmarking : markingsort;
 oldmarking : markingsort;
 t : class("petrinet","Transition");
 nameX : string;

ADT Tokens
 Sorts token
 Generators
  p : class("petrinet", "Place") rel -> token;
 Operations
  pre: class("petrinet","Transition") -> Set(token);
  pos: class("petrinet","Transition") -> Set(token);
  initial: -> Set(token);
 Axioms
  pre(@t) = { p(@p1 suc^@outarc.weight(zero)) |
   in(@p1, Model)=true,
   in(@outarc, Model)=true,
   in(@t -> outArc -> @outarc, Model)=true,
   in(@outarc -> sourcePlace -> @p1, Model)=true
  };
  pos(@t) = { p(@p1 suc^@inarc.weight(zero)) |
   in(@p1, Model)=true,
   in(@inarc, Model)=true,
   in(@t -> inArc -> @inarc, Model)=true,
   in(@inarc -> targetPlace -> @p1, Model)=true
  };
  initial = { p(@p1 suc^@p1.token(zero)) |
                        in(@p1,Model)=true };
 Where
  t : class("petrinet","Transition");
  outarc : class("petrinet", "OutArc");
  inarc : class("petrinet", "InArc");
  p1 : class("petrinet", "Place");
```

```
ADT Marking
 Sorts markingsort
 Generators
  e:->markingsort;
  marking: token markingsort -> markingsort;
 Operations
  build: Set(token) -> markingsort;
  member: token markingsort -> bool;
  add: markingsort markingsort
     -> markingsort;
  subtract: markingsort markingsort
     -> markingsort;
  positive: markingsort -> bool;
  remove: token markingsort -> markingsort;
  equals: markingsort markingsort -> bool;
 Axioms
// build axiom
  (existsIn(@m, @s)=true) =>
    build(@s) =
     marking(@m build(Excluding(@s, @m)));
  build({}) = e;
// member axiom
  member(@m e) = false;
  member(@m marking(@m @mark1)) = true;
  (@m != @m1) =>
    member(@m marking(@m1 @mark1)) =
     member(@m @mark1);
// add
  (member(p(@p1 @n2) @mark2) = true) =>
  add(marking(p(@p1 @n1) @mark1) @mark2)
  = marking(
        p(@p1 plus(@n1 @n2) )
        add(@mark1
         remove(p(@p1 @n2) @mark2)));
  (member(p(@p1 @n2) @mark2) = false) =>
  add(marking(p(@p1 @n1) @mark1) @mark2)
  = marking( p(@p1 @n1) add(@mark1 @mark2));
  add(e @mark1) = @mark1;
// subtract
  (member(p(@p1 @n2) @mark2) = true) =>
  subtract(marking(p(@p1 @n1) @mark1)
  @mark2) = marking(p(@p1 minus(@n2 @n1))
   subtract(@mark1
    remove(p(@p1 @n2) @mark2)));
  (member(p(@p1 @n2) @mark2) = false) =>
  subtract(marking(p(@p1 @n1) @mark1)
  @mark2) = marking( p(@p1 @n1)
    subtract(@mark1 @mark2));
  subtract(e @mark1) = @mark1;
  positive(e) = true;
  (leq(@n1 pred(zero)) = false) =>
    positive(marking(p(@p1 @n1) @mark1)) =
     positive(@mark1);
  (leq(@n1 pred(zero)) = true) =>
    positive(marking(p(@p1 @n1) @mark1)) =
     false;
// remove
  remove(@m e) = e;
  remove(@m marking(@m @mark1))=
   remove(@m @mark1);
  (@m != @m1) =>
    remove(@m marking(@m1 @mark1))=
     marking(@m1 remove(@m @mark1));
// equals
  equals(e e) = true;
  (member(@m @mark2) = true ) =>
   equals(marking(@m @mark1) @mark2) =
   equals(@mark1 remove(@m @mark2));
  (member(@m @mark2) = false ) =>
   equals(marking(@m @mark1) @mark2) =
    false;
 Where
  m : token;
  m1 : token;
  mark1: markingsort;
  mark2: markingsort;
  s : Set(token);
  n1: rel;
  n2: rel;
  p1 : class("petrinet", "Place");
```

The first rule shown in Listing 4.3 (from lines 1 to 12) refers to the first SOS rule

for the State Machine Language shown in Chapter 3, which creates semantic transitions $\xrightarrow{Transition_s.name}$ for the initial states. Similarly, the second rule shown in Listing 4.3 (from lines 14 to 29), refers to the second SOS rule for the State Machine Language shown in Chapter 3, which creates semantic transitions $\xrightarrow{Transition_s.name}$ for $AbstractStates$ given that there exists already some matching semantic transition in the transition system TS_s (this dependency is expressed in line 15). Here we define the semantic domain called cs which is composed by the terms $state()$ and $initial()$ is defined by means of an algebraic data type called $CurrState$, defined also using the SOS language (see Listing 4.4). The $equals$ operation is the minimum required operation on an SOS data type: it is used so that the SOS engine knows when we have reached a fixed-point in our symbolic execution.

The SOS rules for the Petri Nets language presented in Chapter 3, can also be expressed using the SOS language. The presented SOS rules are shown in the Listing 4.5. The first rule shown in Listing 4.5 (from lines 1 to 18 on the left column) refers to the first SOS rule for the Petri Net Language shown in Chapter 3, which creates semantic transitions $\xrightarrow{Transition_s.name}$ for the initial marking $initial_p$. Similarly, the second rule shown in Listing 4.5 (from lines 20 to 40 on the left column), refers to the second SOS rule for the Petri Net Language shown in Chapter 3, which creates semantic transitions $\xrightarrow{Transition_s.name}$ for the subsequent $Transitions$ given that there exists already some matching semantic transition in the transition system TS_s (in particular, this dependency is expressed in lines 21-23).

Notice that the semantic domain $markingsort$ is defined by means of an algebraic data type called $Marking$, defined also using the SOS language (see Listing 4.5 on the right column). Here it is important to be sure that these rewrite rules are converging in terminal values, or otherwise they will enter an infinite loop, which would eventually break the SOS engine.

As referred before in Chapter 3, a marking is a set of pairs of type $Place$ and number of tokens. In our SOS definition, both the token ADT and the marking ADT are defined in Listing 4.5 (the remaining lines on the left column, and the whole right column respectively). We defined $token$ in ADT $Tokens$, as a pair p of elements of type $Place$ and a relative number rel which is an extension of an algebraic natural numbers that also considers negative numbers. For the sake of simplicity we will not include the rel type defined in the ADT $Relatives$, however note that it has the following generators: $zero$ for the base case, suc to represent the successor of a relative number, and $pred$ to represent the predecessor of a relative number. We then use this definition of $token$, in order to define the $markingsort$ in ADT $Marking$ as being a list of tokens:
$marking$: $token\ markingsort$ -> $markingsort$.

In our SOS language implementation, the resulting SOS's transition systems (if finite) can be visualized by means of a dot file supported by the Graphviz language [6].

[6] http://www.graphviz.org/

For instance, in the Figure 4.18, we present a State Machine sentence (on the top of the Figure) and its respective transition system (on bottom of the Figure). This transition system was produced by the SOS engine execution when applied to the semantics presented in Listing 4.3 and the given State Machine model which conforms to the presented StateMachine metamodel. Similarly, we also present a Petri Net sentence (on top of the Figure 4.19), and its respective transition system (on the bottom of the Figure 4.19), this one by using the semantics presented in Listing 4.5.

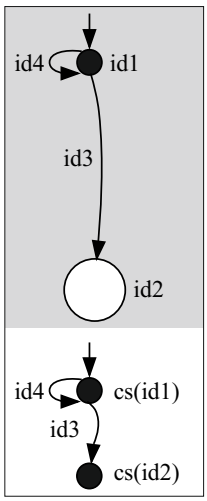

Figure 4.18: A sentence expressed in the State Machine Language, and the resulting transition system.

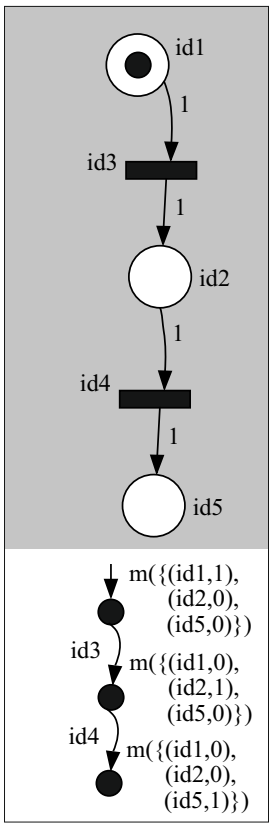

Figure 4.19: A sentence expressed in the Petri Net Language, and the resulting transition system.

By these examples at the instance level, we can clearly see that despite the fact that the two presented languages do share similar concepts, their semantics are very different in nature. However, when the language engineer considers to specify a translation between them, there is some assumption of semantic compatibility between them. In other words, the language engineer assumes that there exists a translation where every state machine, and its translated version in petri nets, have exactly the same behaviour. Therefore everything meaningful that we can express in the State Machines language should also have the same meaning in the Petri Nets Language—i.e., in the sense of equivalence taken in this thesis, the later should accept the same execution traces of the former, regardless of their values.

4.3.2 The SOS Language's Syntax

The abstract syntax of the SOS language is defined by the BNF production rules shown in Listing 4.6. These rules are able to produce the sentences shown before. For instance, it is easy to observe that the production rules for expressing the SOS algebraic data types (i.e., the ADT non-terminals from lines 10 to 25) are able to produce the sentences shown in Listings 4.4 and 4.5 (on the right column). Similarly, the rules for expressing the whole SOS specification (i.e., the Semantics non-terminals from line 1 to 24) are able to produce the sentences shown in Listings 4.3 and 4.5 (on the left column).

Listing 4.6: The SOS syntax expressed using the BNF notation

```
    SOS ::= "Semantics" (Rule | ADT)*;
    Rule ::= ("Assuming" PremisseList "Then" Conclusion |
              "Fact" Conclusion )
             ("Where" Variable+)?;
5   PremisseList ::= Condition ("," PremisseList)?; .
    AlgebraicConditionList ::= "(" AbsEquation ")"
                               ("," AlgebraicConditionList)?;
    Conclusion ::= Term "->>" Term "->>" Term
                   "in" "Transition_System";
10  Condition ::= Conclusion | "(" AbsEquation ")";

    ADT ::= "ADT" Id
                ("Sorts" SortDeclaration+)?
                ("Generators" Generator+)?
15              ("Operations" Operation+)?
                ("Axioms" Axiom+)?
                ("Where" Variable+)?;
    Generator ::= AbsOperation;
    Operation ::= AbsOperation;
20  AbsOperation ::= Id ":" Sort* "->" Sort ";";
    Variable ::= Id (":" Sort)? ";";
    Axiom ::= CondEquation;
    CondEquation ::= ("(" AbsEquation+ ")" "=>")? Equation ";";
    VariableRef ::= "@"Id;
25  CTerm ::= Id("^" (Integer | Term))? ("(" Term+ ")")?;
    AbsEquation ::= Equation | Inequation;
    Equation ::= Term "=" Term;
    Inequation ::= Term "!=" Term;
    SortDeclaration ::= Id;
30  AtomicSort ::= Id;

    ModelSet ::= "Model";
    SetConstructor ::= "{" ( Term ("|"
                       AlgebraicConditionList )? )? "}";
35  ForAllIn ::= "in" "(" Term "," Term ")";
    ExistsIn ::= "existsIn" "(" Term "," Term ")";
    Union ::= "Union" "(" Term "," Term ")";
    Excluding ::= "Excluding" "(" Term "," Term")";
    Intersection ::= "Intersect" "(" Term "," Term ")";
40  ModelRelation ::= VariableRef "->" Id "->" VariableRef;
    ModelClassAttribute ::= VariableRef "." Id;
    ModelSort ::= "class" "(" String "," String ")" ;
    Set ::= "Set" "(" Sort ")";

45  Sort ::= ModelSort | Set | AtomicSort;

    Term ::= VariableRef | CTerm |
             ModelRelation | ModelClassAttribute |
             ModelSet | ForAllIn | ExistsIn | Union |
50           Excluding | Intersection | SetConstructor;
```

We will use these syntactic constructs in order to define the operational semantics of the SOS language.

4.3.3 The SOS Language's Semantics

In addition to the defined syntactic constructs we define a construct to store the intermediate interpretation results while computing a given semantics. The following definitions are part of the SOS Language operational semantics. Notice that we could also define the operational semantics of the SOS Language using the SOS Language, however for readability reasons we prefer to refer to a mathematical formalism, hence avoiding confusions between the SOS's syntactic and semantic structures. Here, similar to the SOS language, the preconditions of the defined inference rules should be read from top to bottom, from left to right—i.e., the bound variables resulting from the evaluation of the first preconditions are naturally propagated to the evaluation of the next ones. Also for readability reasons, we will denote SOS syntactic textual expressions inside double brackets '$[\![]\!]$'.

Definition 4.29. *Environment*

The $Environment \subseteq (Variable \times Term)$*, is a set of pairs of variables and their corresponding values. Also these values must be closed terms (i.e., without variables).*

The environment is used to store the intermediate values of a set of variables, during a given abstract computation. Therefore, for each variable identifier, we will store its respective value which must be a Term (containing no variables), and which sort must be defined in the ADT section.

Definition 4.30. *SOS Match Term Relation*

Let $\{T_0, .., T_N, T'_0, .., T'_N\} \subset Term$ *be SOS Terms, and* $\{env_0, .., env_N\} \subset Environment$ *be environments. Also let* $sos \in SOS^{mm}$ *be a semantics for a language whose metamodel is* $mm \in MM$*, and a model* $m \in M^{mm}$ *defined according to that metamodel.*

The $\overset{match}{\rightarrow} \subseteq Environment \times Term \times Term \times Sort \times Semantics \times TG \times Environment$ *relation is defined by the minimum set that satisfies the following rules:*

a) Match of two CTerms

$$\frac{\begin{array}{l}\exists\ AbsOperation^{sos}\ \cdot\ AbsOperation^{sos} = [\![Id : S_0..S_N \rightarrow S]\!], \\ \langle env, [\![T_0]\!], [\![T'_0]\!], [\![S_0]\!], sos, m\rangle \xrightarrow{match} env_0, \\ \ldots \\ \langle env_{N-1}, [\![T_N]\!], [\![T'_N]\!], [\![S_N]\!], sos, m\rangle \xrightarrow{match} env_N\end{array}}{\langle env, [\![Id(T_0..T_N)]\!], [\![Id(T'_0..T'_N)]\!], [\![S]\!], sos, m\rangle \xrightarrow{match} env_N}$$

b) Match of a CTerm with a Variable

$$\frac{\begin{array}{l}\exists\ AbsOperation^{sos}\ \cdot\ AbsOperation^{sos} = [\![Id : S_0..S_N \rightarrow S]\!], \\ \exists\ Variable^{sos}\ \cdot\ Variable^{sos} = [\![Id' : S]\!], \\ env' = env \cup \{(Id', Id(T_0..T_N))\}\end{array}}{\langle env, [\![Id(T_0..T_N)]\!], [\![@Id']\!], [\![S]\!], sos, m\rangle \xrightarrow{match} env'}$$

c) Match of a Variable with a CTerm

$$\frac{\begin{array}{l}\exists\, Variable^{sos} \cdot Variable^{sos} = [\![Id'' : S]\!],\\ \exists\, AbsOperation^{sos} \cdot AbsOperation^{sos} = [\![Id' : S_0..S_N \rightarrow S]\!],\\ (Id'', Id(T_0..T_N)) \in env,\\ \langle env, [\![Id(T_0..T_N)]\!], [\![Id'(T'_0..T'_N)]\!], [\![S]\!], sos, m\rangle \xrightarrow{match} env'\end{array}}{\langle env, [\![@Id'']\!], [\![Id'(T'_0..T'_N)]\!], [\![S]\!], sos, m\rangle \xrightarrow{match} env'}$$

d) Match of two Variables

$$\frac{\begin{array}{l}\exists\, Variable^{sos} \cdot Variable^{sos} = [\![Id'' : S]\!],\\ \exists\, Variable^{sos} \cdot Variable^{sos} = [\![Id' : S]\!],\\ (Id'', Id(T_0..T_N)) \in env,\\ env' = env \cup \{(Id', Id(T_0..T_N)))\}\end{array}}{\langle env, [\![@Id'']\!], [\![@Id']\!], [\![S]\!], sos, m\rangle \xrightarrow{match} env'}$$

The evaluation of a given algebraic data type Term strictly depends on the axioms (also called rewriting rules) defined for that ADT. Therefore, while evaluating a given Term, we have to perform a match on the defined rewriting rules for that Term.

Intuitively, the definition of the match relation for two sets of SOS Terms says that it inserts in the environment the values of the variables with its respective closed terms, considering that they have compatible sorts. In particular, rule *a)* says that two CTerms have compatible sorts (namely *S*) if they have the same name (namely *Id*), the same number of composed arguments (namely *N*), and each of them have also compatible sorts, respectively. Each evaluation of the *match* relation produces a new (possibly changed) environment. This can be explained in rule *b)*, where the matching of a closed term with a variable copies both the variable and the term as its value into the new environment. Rule *d)* is similar to *b)*, where the term value (i.e., *Id*($T_0..T_N$)) from one variable is copied and associated with another variable, in a new environment. Finally, rule *c)*, just checks if a term value (namely, *Id*($T_0..T_N$)) associated with a variable in a given environment, is compatible with another given term value (i.e., *Id'*($T_0..T_N$)) .

We further use this match relation in order to evaluate Terms that can be either resolved to ADT generators, or (in the case of operators) rewritten into other Terms according to the defined rewrite axioms.

Definition 4.31. *SOS Term and AbsEquation Evaluation*

Let $\{T_0, .., T_N, T'_0, .., T'_N\} \subset Term$ *be SOS Terms,* $\{env_0, .., env_N\} \subset Environment$ *be environments,* $sos \in SOS^{mm}$ *be a semantics for a language whose metamodel is* $mm \in MM$*, and a model* $m \in M^{mm}$ *defined according to that metamodel.*

The evaluation relation on SOS terms and AbsEquations $\overset{eval}{\rightarrow} \subseteq Environment \times \{Term \cup AbsEquation\} \times SOS \times M_{mm} \times Environment \times \{Term \cup AbsEquation\}$ *is defined by the minimum set that satisfies the following rules:*

a) CTerm with Generator

$$\frac{\begin{array}{l}\exists\, Generator^{sos} \cdot Generator^{sos} = [\![Id : S_0..S_N \rightarrow S]\!], \\ \langle env, [\![Id(T_0..T_N)]\!], [\![S]\!], sos, m\rangle \xrightarrow{match} env', \\ \langle env', [\![T_0]\!], sos, m\rangle \xrightarrow{eval} \langle env_0, [\![T'_0]\!]\rangle, \\ \ldots \\ \langle env_{N-1}, [\![T_N]\!], sos, m\rangle \xrightarrow{eval} \langle env_N, [\![T'_N]\!]\rangle\end{array}}{\langle env, [\![Id(T_0..T_N)]\!], sos, m\rangle \xrightarrow{eval} \langle env_N, [\![Id(T'_0..T'_N)]\!]\rangle}$$

b) CTerm with Axioms

$$\frac{\begin{array}{l}\exists\, Operator^{sos} \cdot Operator^{sos} = [\![Id : S_0..S_N \rightarrow S]\!], \\ \exists\, Axioms^{sos} \cdot Axioms^{sos} = [\![AbsEquation_0 \,..\, AbsEquation_K => Id(T'_0..T'_N) = RTerm]\!], \\ \langle env, [\![Id(T_0..T_N)]\!], [\![Id(T'_0..T'_N)]\!], [\![S]\!], sos, m\rangle \xrightarrow{match} env', \\ \langle env', [\![AbsEquation_0]\!], sos, m\rangle \xrightarrow{eval} \langle env_0, [\![]\!]\rangle, \\ \ldots \\ \langle env_{K-1}, [\![AbsEquation_K]\!], sos, m\rangle \xrightarrow{eval} \langle env_K, [\![]\!]\rangle, \\ \langle env_K, [\![RTerm]\!], sos, m\rangle \xrightarrow{eval} \langle env'', [\![RTerm']\!]\rangle\end{array}}{\langle env, [\![Id(T_0..T_N)]\!], sos, m\rangle \xrightarrow{eval} \langle env'', [\![RTerm']\!]\rangle}$$

c) Equation

$$\frac{\begin{array}{c}\langle env, [\![LTerm]\!], sos, m\rangle \xrightarrow{eval} \langle env', R\rangle, \\ \langle env', [\![RTerm]\!], sos, m\rangle \xrightarrow{eval} \langle env'', R\rangle\end{array}}{\langle env, [\![LTerm = RTerm]\!], sos, m\rangle \xrightarrow{eval} \langle env'', [\![]\!]\rangle}$$

d) Inequation

$$\frac{\begin{array}{c}\langle env, [\![LTerm]\!], sos, m\rangle \xrightarrow{eval} \langle env', R\rangle, \\ \langle env', [\![RTerm]\!], sos, m\rangle \xrightarrow{eval} \langle env'', R'\rangle, R \neq R'\end{array}}{\langle env, [\![LTerm\ !=\ RTerm]\!], sos, m\rangle \xrightarrow{eval} \langle env'', [\![]\!]\rangle}$$

e) ForAllIn with a ModelSort Variable

$$\frac{\begin{array}{l}\exists\, Variable^{sos} \;\cdot\; Variable^{sos} = [\![Id : class(PackageName, ClassName)\,]\!],\\ \forall\, x \in\; V^m \;\cdot\; \tau_v^m(x) = [\![PackageName.ClassName]\!],\\ env' \;=\; env \cup \{(Id, x)\}\end{array}}{\langle env, [\![in(@Id, Model)]\!], sos, m\rangle \xrightarrow{eval} \langle env', [\![x]\!]\rangle}$$

f) ExistsIn with a ModelSort Variable

$$\frac{\begin{array}{l}\exists\, Variable^{sos} \;\cdot\; Variable^{sos} = [\![Id : class(PackageName, ClassName)\,]\!],\\ \exists\, x \in\; V^m \;\cdot\; \tau_v^m(x) = [\![PackageName.ClassName]\!],\\ env' \;=\; env \cup \{(Id, x)\}\end{array}}{\langle env, [\![existsIn(@Id, Model)]\!], sos, m\rangle \xrightarrow{eval} \langle env', [\![x]\!]\rangle}$$

g) ForAllIn with a ModelRelation

$$\frac{\begin{array}{l}\exists\, x, x' \in V^m \;\cdot\; (Id, x), (Id', x') \in env,\\ \forall\, (x \xrightarrow{Label} x') \in\; E^m \;\cdot\; \tau_e^m((x \xrightarrow{Label} x')) = Label\end{array}}{\langle env, [\![in(@Id \twoheadrightarrow Label \twoheadrightarrow @Id', Model)]\!], sos, m\rangle \xrightarrow{eval} \langle env, [\![(x \xrightarrow{Label} x')]\!]\rangle}$$

h) ExistsIn with a ModelRelation

$$\frac{\begin{array}{l}\exists\, x, x' \in V^m \;\cdot\; (Id, x), (Id', x') \in env,\\ \exists\, (x \xrightarrow{Label} x') \in\; E^m \;\cdot\; \tau_e^m((x \xrightarrow{Label} x')) = Label\end{array}}{\langle env, [\![existsIn(@Id \twoheadrightarrow Label \twoheadrightarrow @Id', Model)]\!], sos, m\rangle \xrightarrow{eval} \langle env, [\![(x \xrightarrow{Label} x')]\!]\rangle}$$

where Label is of type String.

i) ModelClassAttribute

$$\frac{\begin{array}{l}\exists\, x, x' \in V^m \;\cdot\; (Id, x) \in env,\\ \exists\, (x \xrightarrow{Id'} x') \in E^m \;\cdot\; \tau_e^m((x \xrightarrow{Id'} x')) = Id'\end{array}}{\langle env, [\![@Id.Id']\!], sos, m\rangle \xrightarrow{eval} \langle env, [\![x']\!]\rangle}$$

j) SetConstructor

$$\frac{\begin{array}{l} T = \{R \mid \langle env, Term, sos, m\rangle \xrightarrow{eval} \langle env', R\rangle, \\ \quad \langle env', AbsEquation_0, sos, m\rangle \xrightarrow{eval} \langle env_0, [\![]\!]\rangle \\ \quad \ldots \\ \quad \langle env_{N-1}, AbsEquation_N, sos, m\rangle \xrightarrow{eval} \langle env_N, [\![]\!]\rangle \\ \} \end{array}}{\langle env, [\![\{Term \mid AbsEquation_0 \,..\, AbsEquation_N\}]\!], sos, m\rangle \xrightarrow{eval} \langle env_N, [\![T]\!]\rangle}$$

k) Free Algebraic Variable

$$\frac{\begin{array}{l} (Id, \_) \notin env, \\ \exists\, Variable^{sos} \;\cdot\; Variable^{sos} = [\![Id : S]\!], \\ \forall\, Generator^{sos} \;\cdot\; Generator^{sos} = [\![Id' : \star \to S]\!], \\ env' = env \cup \{(Id, Id')\} \end{array}}{\langle env, [\![@Id]\!], sos, m\rangle \xrightarrow{eval} \langle env', [\![Id']\!]\rangle}$$

l) Bounded Algebraic Variable

$$\frac{(Id, T) \in env}{\langle env, [\![@Id]\!], sos, m\rangle \xrightarrow{eval} \langle env, [\![T]\!]\rangle}$$

m) Union

$$\frac{\begin{array}{c} \langle env, [\![T]\!], sos, m\rangle \xrightarrow{eval} \langle env', [\![R]\!]\rangle, \\ \langle env', [\![T]\!], sos, m\rangle \xrightarrow{eval} \langle env'', [\![R']\!]\rangle \end{array}}{\langle env, [\![Union(T, T')]\!], sos, m\rangle \xrightarrow{eval} \langle env'', [\![R \;\cup\; R']\!]\rangle}$$

n) Intersection

$$\frac{\begin{array}{c} \langle env, [\![T]\!], sos, m\rangle \xrightarrow{eval} \langle env', [\![R]\!]\rangle, \\ \langle env', [\![T]\!], sos, m\rangle \xrightarrow{eval} \langle env'', [\![R']\!]\rangle \end{array}}{\langle env, [\![Intersection(T, T')]\!], sos, m\rangle \xrightarrow{eval} \langle env'', [\![R \;\cap\; R']\!]\rangle}$$

o) Excluding

$$\frac{\begin{array}{c} \langle env, [\![T]\!], sos, m\rangle \xrightarrow{eval} \langle env', [\![R]\!]\rangle, \\ \langle env', [\![T]\!], sos, m\rangle \xrightarrow{eval} \langle env'', [\![R']\!]\rangle \end{array}}{\langle env, [\![Excluding(T, T')]\!], sos, m\rangle \xrightarrow{eval} \langle env'', [\![R \;\setminus\; R']\!]\rangle}$$

Note that the operational semantics described above is similar to the semantics of traditional algebraic data types. In particular, rule *a)* evaluates a given term named *Id* by matching a compatible generator using the *match* relation, and evaluating its arguments using the *eval*. We stress that by 'compatible', we mean having the name and the same number of arguments (i.e., *N*) with compatible sorts respectively. Similarly, rule *b)* evaluates a given term named *Id* by matching a compatible left hand side of an axiom the *match* relation, and using the *eval*, it checks all of the conditions (i.e., from $AbsEquation_0$ to $AbsEquation_K$), and evaluates its respective right hand side, which is the returned value of the evaluated term. Rules *c)* and *d)* are intuitive and complementary: the evaluation of a given equation succeeds if and only if the evaluation of both of its left and right hand sides have the same result (namely *R*); conversely, the evaluation of a given inequation succeeds if and only if the evaluation of both of its left and right hand sides have the different results. The value of successfully evaluated equations (or inequations) is always an empty term.

However, we extended it with syntactic constructions to enable both the definition of arbitrary sets on defined algebras, and on a given model; and also to use these definitions by means of powerful universal and existential quantifiers. In particular, rules *e)* and *f)* define the semantics of the above described quantifiers when applied to variables which sorts are defined on the metamodel of the language under specification as a pair of names *PackageName* and *ClassName*. Moreover, the values related with this evaluation result directly from the formal meaning of (respectively) universal and existential quantification of these kind of terms identified by this pair, inside a given input model (i.e., *m*), instance of the language under specification. Similarly, rules *g)* and *h)* define the semantics of both universal and existential quantifiers when applied to relations which names are defined on the metamodel of the language under specification. Again, the values related with this evaluation result directly from the (quantified) query in a given input model (i.e., *m* which is instance of the language under specification) for associations (e.g., containment or simply references) identified by its name (i.e., *Label*), considering the values found for both the source and target model elements, *Id* and *Id'* respectively. In what matters to attributes (see rule *i)*), since in every input model (i.e., *m* which is instance of any language under specification), they are uniquely identified by its name (in this case *Id'*), we only defined the case of existential semantics for querying these attributes within a given model entity (namely *Id*). Rule *j)* formally describes the meaning of the *SetConstructor* when applied to a term *Term*, and a set of *AbsEquation*: it can be rewritten as a set *T* of *R*'s such that *R* is the evaluation value of *Term*, and all of the defined *AbsEquation* evaluations are successful. Notice, that each evaluation produces a new environment propagating the variable's values from environment env to env_N. Rules *k)*, and *l)* are complementary: they describe the meaning of evaluating a variable which can be either free or bounded respectively. In the former rule, the variable *Id* under evaluation do not exists in the environment, so the evaluation uses all of the generators which have the same sort *S* of the

variable *Id* in order to instantiate its possible value, and introduces it in the environment. In the latter, the variable *Id* under evaluation do not exists in the environment, so the environment is maintained and the value of the evaluation is the value associated with that variable in the environment. Finally, the last three rules *m)*, *n)* and *o)* describe the meaning of the *Union*, *Intersection* and *Excluding* operators, directly from the mathematical formal definitions of set theory.

For the sake of readability, in the above definition, we do not formally describe how the exponentiation of terms is evaluated. Intuitively, we write an $Id^K(T)$ as an abbreviation of $Id(..Id(Id^K(T))..)$, where the first sequence '$Id(..Id$' and the final '$)..)$' are both of size K. For instance, having K = 3, and N = 2, we have that the expression $Id^3(T)$ can be rewritten into the following: $Id(Id(Id(T)))$.

In order to complete the description of the semantics of the SOS language, we need to provide its semantic domain, which is a transition system. That is, given a SOS semantic definition of a language, and a given sentence expressed in that language, the semantics of that sentence results in a transition system which represents a symbolic execution of the sentence in a virtual/abstract machine.

Definition 4.32. *SOS Transition System*

A semantic transition SemTransition is a 3-tuple $\langle PreState, \langle Label, V\rangle, PosState\rangle$ *where* $PreState, Label, PosState$ *are Terms with no variable references, and* V *is a set of vertices. A set of semantic transitions is called a* $TransitionSystem$*, and the set of all* $TransitionSystem$ *is called* TS*.*

This transition system is generated by the evaluation of the defined SOS Rules when applied to a concrete model m. As shown before, a SOS Rule consists in a set of assumptions (or conditions) and a conclusion. The assumptions can either be algebraic equations (or inequations), or conclusions from previous rule applications.

Definition 4.33. *SOS Condition Evaluation Semantics*

Let $mm \in MM$ *be a metamodel,* $m \in M_{mm}$ *be a model both defined w.r.t. metamodel* mm*, and* $\{env, env', env''\} \subseteq Environment$ *are environments.*

The evaluation relation on a conditions $\overset{evalCondition}{\rightarrow} \subseteq Environment \times TS \times Condition \times SOS \times M_{mm} \times Environment$ *is defined by the minimum set that satisfies the following rules:*

a) A previous Conclusion

$$\frac{\begin{array}{l}\langle env, Pre, sos, m\rangle \xrightarrow{eval} \langle env', Pre'\rangle,\\ \langle env', Label, sos, m\rangle \xrightarrow{eval} \langle env'', Label'\rangle,\\ \langle env'', Pos, sos, m\rangle \xrightarrow{eval} \langle env''', Pos'\rangle,\\ transition = \langle Pre', \langle Label, V\rangle, Pos'\rangle,\\ transition \in ts\end{array}}{\langle env, ts, [\![Pre \twoheadrightarrow Label \twoheadrightarrow Pos]\!], sos, m\rangle \xrightarrow{evalCondition} env'''}$$

b) An Abstract Equation

$$\frac{\langle env, [\![AbsEquation]\!], sos, m\rangle \xrightarrow{eval} env',}{\langle env, ts, [\![(AbsEquation)]\!], sos, m\rangle \xrightarrow{evalCondition} env'}$$

For each successful rule application (meaning that all of its assumptions are satisfied), we will evaluate its respective condition and generate a transition in the resulting transition system, as we show in the following definitions.

Definition 4.34. *SOS Conclusion Evaluation Semantics*

Let $mm \in MM$ be a metamodel, $m \in M_{mm}$ be a model both defined w.r.t. metamodel mm, $transition \in TransitionSystem$ a semantic transition, and $\{env, env', env'', env'''\} \subseteq Environment$ are environments.

The evaluation relation on a conclusion $\overset{evalPos}{\rightarrow} \subseteq Environment \times Conclusion \times SOS \times M_{mm} \times TransitionSystem$ is defined by the minimum set that satisfies the following rule:

$$\frac{\begin{array}{c}\langle env, Pre, sos, m\rangle \xrightarrow{eval} \langle env', Pre'\rangle,\\ \langle env', Label, sos, m\rangle \xrightarrow{eval} \langle env'', Label'\rangle,\\ \langle env'', Pos, sos, m\rangle \xrightarrow{eval} \langle env''', Pos'\rangle,\\ transition = \langle Pre', \langle Label, V\rangle, Pos'\rangle\end{array}}{\langle env, [\![Pre \twoheadrightarrow Label \twoheadrightarrow Pos]\!], sos, m\rangle \xrightarrow{evalPos} transition}$$

where $\{Pre, Label, Pos, Pre', Lbl', Pos'\} \subseteq Term$ are terms, and
$V = \{v \in V^m \mid \langle var, \langle v, SD\rangle\rangle \in (env'' \setminus env')\}$, is the set of vertices from model m that were read while evaluating the label terms $Pre, Label, Pos$ into $Pre', Label', Pos'$ respectively, and also $SD \in Sorts$ is the sort of the ModelTerm.

Definition 4.35. *SOS Evaluation Semantics*

Let $mm \in MM$ be a metamodel, $m \in M_{mm}$ be a model both defined w.r.t. metamodel mm, $transition \in TransitionSystem$ be a semantic transition, and $\{env, env'\} \subseteq Environment$ are environments.

The evaluation relation on an arbitrary SOS rule specification $\overset{eval}{\rightarrow} \subseteq TS \times Rule \times SOS \times M_{mm} \times TransitionSystem$ is defined by the minimum set that satisfies the following rules:

a) Fact Evaluation

$$\frac{\langle \{\}, Conclusion, sos, m\rangle \xrightarrow{evalPos} transition}{\langle ts, [\![Fact\ Conclusion]\!], sos, m\rangle \xrightarrow{eval} transition}$$

b) Sequent Evaluation

$$\frac{\begin{array}{l}\langle\{\}, ts, Condition_0, sos, m\rangle \xrightarrow{evalCondition} env_0,\\ \ldots\\ \langle env_{N-1}, ts, Condition_N, sos, m\rangle \xrightarrow{evalCondition} env_N,\\ \langle env_N, Conclusion, sos, m\rangle \xrightarrow{evalPos} transition\end{array}}{\langle ts, [\![Assuming\ Condition_0..Condition_N\ Then\ Conclusion]\!], sos, m\rangle \xrightarrow{eval} transition}$$

Definition 4.36. *SOS Fixpoint Semantics*

Let $mm \in MM$ be a metamodel, $sos \in SOS_{mm}$ be a SOS specification, $m \in M_{mm}$ be a model both defined w.r.t. metamodel mm, $Rule \in RuleSpec$ be a rule specification, $ts \in TS$ be a transition system, and $transition \in TransitionSystem$ be a semantic transition.

The fixpoint relation $\overset{fixpoint}{\rightarrow} \subseteq TS \times SOS_{mm} \times M_{mm} \times TransitionSystem$ is defined by the minimum set that satisfies the following rules:

$$\frac{\begin{array}{l}Rule \in Rule^{sos},\\ \langle Rule, sos, m\rangle \xrightarrow{eval} transition,\\ transition \notin ts\end{array}}{\langle ts, sos, m\rangle \xrightarrow{fixpoint} transition}$$

$$\frac{\begin{array}{l}Rule \in Rule^{sos},\\ \langle Rule, sos, m\rangle \xrightarrow{eval} transition,\\ transition \notin ts,\\ \langle ts \cup \{transition\}, sos, m\rangle \xrightarrow{fixpoint} transition',\\ transition' \notin ts\end{array}}{\langle ts, sos, m\rangle \xrightarrow{fixpoint} transition'}$$

We also define the rule to compute the initial relation $\overset{initial}{\rightarrow} \subseteq TS \times SOS_{mm} \times M_{mm} \times TransitionSystem$ is defined by the minimum set that satisfies the following rule:

$$\frac{\begin{array}{l}Rule \in Rule^{sos},\\ \langle Rule, sos, m\rangle \xrightarrow{eval} transition\end{array}}{\langle\{\}, sos, m\rangle \xrightarrow{initial} transition}$$

Definition 4.37. *Fix Point Functions*

The function $computeFixPoint : SOS_{mm} \times M_{mm} \rightarrow TS$ *uses the relation* $fixpoint$ *in order to return a* TransitionSystem *based on a SOS specification and a model w.r.t.* mm*, such that:*

$$computeFixPoint(sos, m) = \{st \in TransitionSystem \mid \langle\{\}, sos, m\rangle \xrightarrow{fixpoint} st\}$$

We also define the function $computeInitial : SOS_{mm} \times M_{mm} \rightarrow TS$ *uses the relation* $initial$ *in order to return a* TransitionSystem *based on a SOS specification and a model w.r.t.* mm*, such that:*

$$computeInitial(sos, m) = \{st \in TransitionSystem \mid \langle\{\}, sos, m\rangle \xrightarrow{initial} st\}$$

Intuitively, the above functions can be used to compute the semantics of any sentence in a given language, assuming that we have an SOS semantics definition for that same language. We will further use both of these functions in order to validate translations between two different arbitrary languages.

4.3.4 The SOS Tool

The implementation of the SOS language, involved the implementation of both of its editor and its execution engine. Moreover, these were also deployed as Eclipse plugins based on the Eclipse Modeling Framework (EMF) [7].

Figure 4.20 shows how the SOS textual editor interacts with the SOS' (execution) engine in order to produce graphical representations of the semantic values of every model conforming to an arbitrary language. Also, the depicted numbers inside circles denote a logical order of events in time. Notice that this interaction involves the intensive use of model transformations expressed in DSLTrans (here denoted as translations in light-grey boxes), namely in steps 3, 5, 7 and 9. Notice also that Ecore to DSLTrans translation is an high-order transformation (from now on denoted as high-order translation).

The SOS textual editor plugin was automatically generated by creating an EMFText [8] Project, where we annotated the metamodel presented in Figure 4.21 with textual concrete syntax directives. Notice that, the complete metamodel of the SOS language consists of several packages which are metamodels themselves—in this Figure, we present the whole version, with all of its internal packages expanded. The concrete syntax annotations were expressed in a CS (Concrete Syntax Specification Language) file, using

[7] All of the tools associated with the SOS language are included in the DSLTrans toolset, available publicly at: https://github.com/githubbrunob/DSLTransGIT/blob/master/DSLTrans-Release/DSLTrans-Suite-06062k12.zip?raw=true

[8] http://www.emftext.org/index.php/EMFText

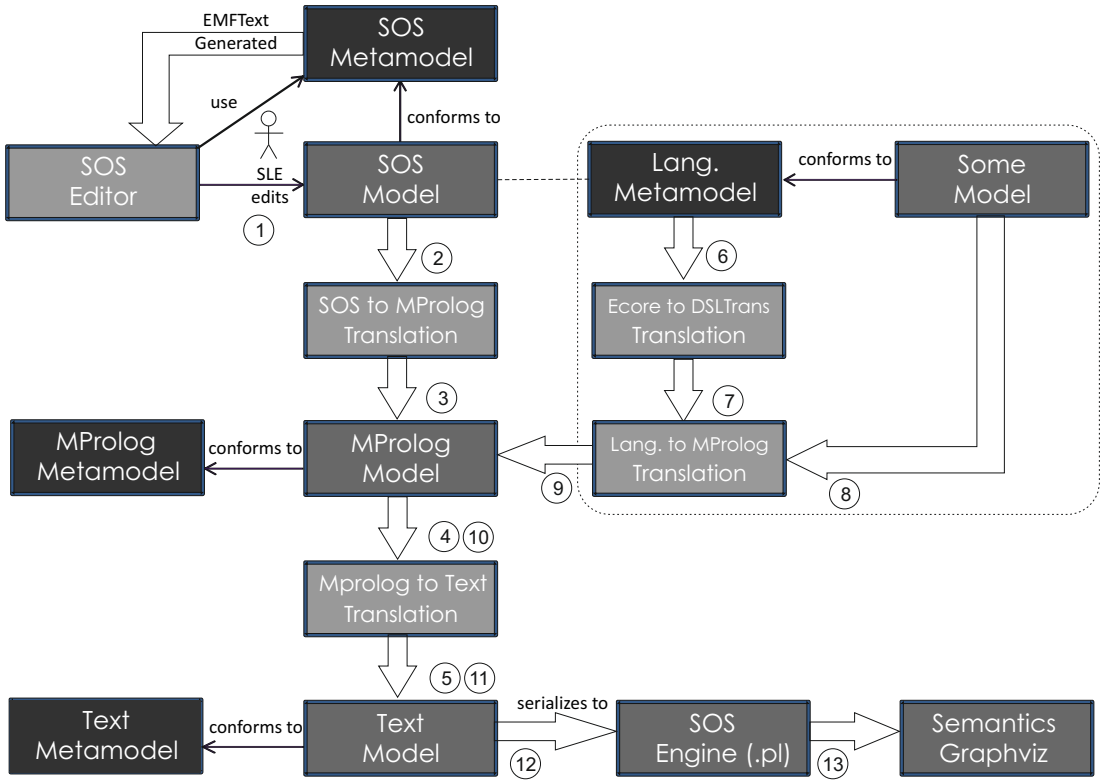

Figure 4.20: The reference implementation of SOS as a set of Eclipse plug-ins.

the types defined in the presented SOS metamodel. The resulting textual editor has already (by default) the syntax-highlighting capabilities deduced from the .cs specification, and enabled the edition of the example presented in Listings 4.3 and 4.5. Furthermore, this edition plugin have an additional (by default) capability of producing an XML/XMI version of the edited operational semantic specifications expressed in SOS.

In what matters to the execution of SOS semantics specifications, the SOS execution engine plugin was first fully coded in prolog, and then interfaced with the Eclipse plugin development API and EMF API. The development of this interface with prolog involved the design of a metamodeled version of the prolog language called MProlog, which metamodel is presented in Figure 4.22.

The MProlog language captures all of the concepts present in the prolog language. Furthermore, a new language called Text was also developed, which metamodel is presented in Figure 4.23. The Text language captures all of the concepts required to output structured text into files and directories—structured text means that text can be organized into blocks composed of lines.

The semantics of the Text language is implemented by means of a small compiler written in java, that reads the Text specification and outputs a set of files in the specified directories. The semantics of the MProlog language was defined by means of a DSLTrans translation which was developed in order to automatically translate sentences expressed in this MProlog language, into sentences expressed in the Text Language, which is then

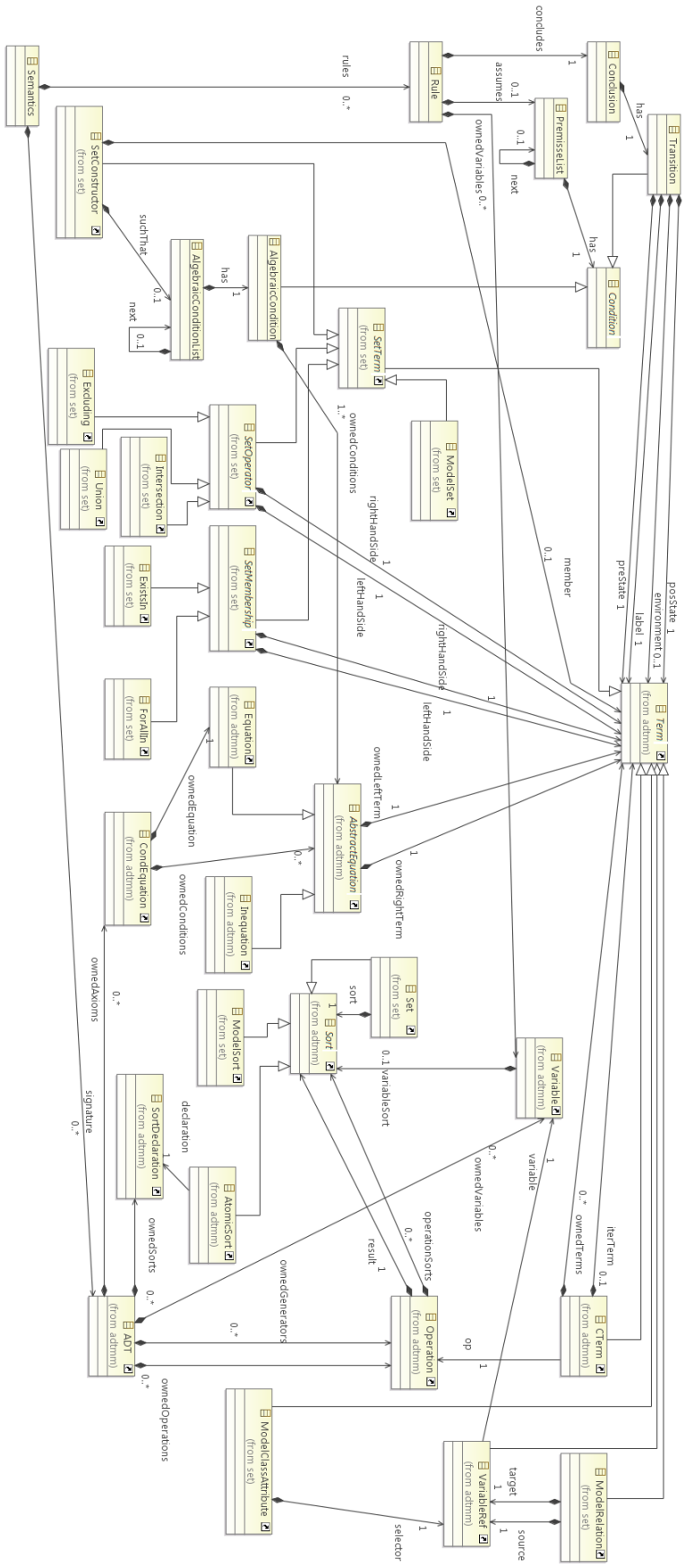

Figure 4.21: The SOS Metamodel (all of the packages expanded).

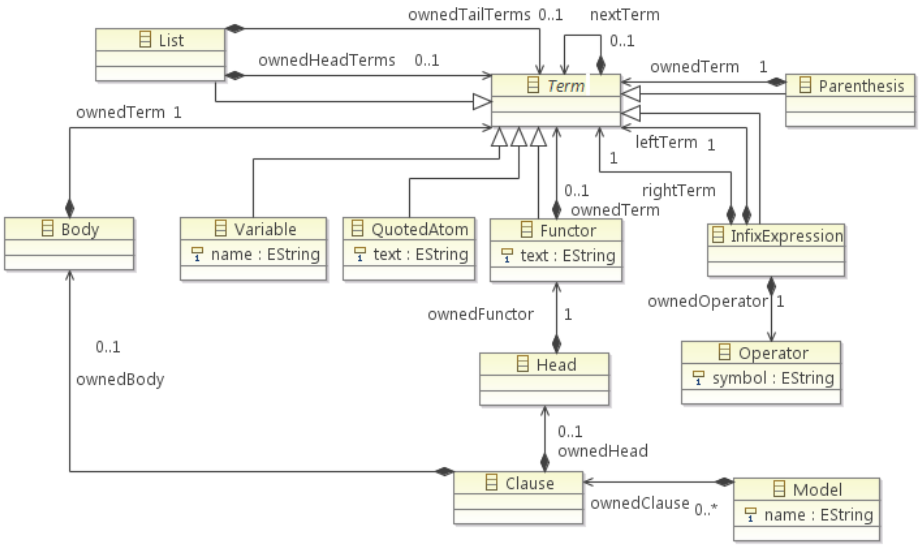

Figure 4.22: The MProlog Metamodel.

able to automatically produce the equivalent prolog in textual form (i.e., in .pl files).

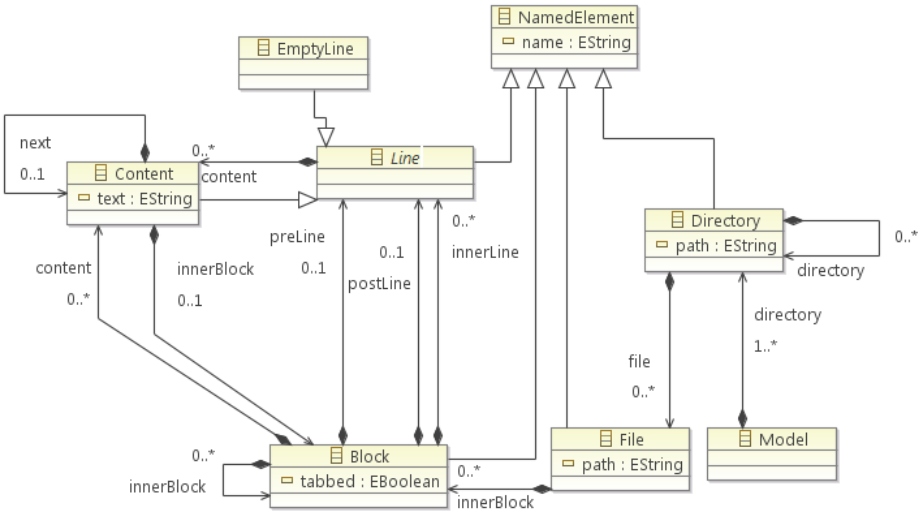

Figure 4.23: The Text Metamodel.

Also, a DSLTrans high-order translation was developed in order to take any kind of metamodel specification (i.e., from any language) expressed in Ecore (i.e., a language that is used to express metamodels), in order to produce another DSLTrans translation that is able to take any sentence expressed in that language and produce a relational version of it expressed in the MProlog Language. Notice that a translation that produces another translation is called an high-order translation.

Moreover, we also defined a translational semantics for the SOS language by means of a DSLTrans translation, that is able to translate any SOS files in the format XML/XMI

(i.e., conforming to the SOS metamodel presented in Figure 4.21), and produce its representation in the MProlog Language. In this translation, SOS constructions are translated into MProlog constructions whose evaluation procedures mimics the same evaluation procedures as defined in the SOS language semantics definition. For instance, the specified axioms of an ADT are translated into special clauses, that the SOS execution engine (written in prolog) is able to process. Another example is the SOS rules that are also translated into prolog clauses named *rule*, with a particular signature, so that the execution of the SOS execution engine is able to process it in such a way that it mimics what was defined in the SOS language semantics definition.

In conclusion, the SOS execution engine will take *(i)* a relational version of a sentence (conforming to any kind of metamodeled language) expressed in prolog; and *(ii)* a SOS specification also expressed in prolog (after the translations to MProlog, and Text); and produces a graph (representing the effect of the input sentence in an abstract computation system) in a file expressed in the dot language [9].

4.4 Conclusions

Based on the mathematical notions of graphs and sets, we defined theories for both models, and syntactic models of DSMLs. Then we reused the same notions in order to define two languages for enabling the specification of the semantics models of DSMLs.

In particular, we first defined and implemented a model transformation language called DSLTrans that can be used to define the translational semantics of DSMLs, and automatically generate DSML compilers based on them. There are many model transformation tools available, such as GReAT[AKS03], EMF Tiger[BET08], Moflon[AKRS06], Kermeta[DFF+09], IBM's MTF [IBM07] or ATL[JK05]. Some of them are already starting to be used in the industry. These tools are presented in Table 4.1.

Tools	Editor		Expressiveness		Guarantees		EMF
	Textual	Visual	Layers	Allows Recursion	Confluence	Termination	Compatible
EMF Tiger		✓	✓	✓	✓*	✓*	✓
Great		✓		✓			
ATL	✓			✓			✓
Moflon		✓		✓			
IBM Rational		✓	✓	✓			
Kermeta	✓		✓	✓			✓
DSLTrans	✓	✓	✓		✓	✓	✓

Table 4.1: State of the art model transformation tools and languages. For EMF Tiger, the proven guarantees are valid only for a particular shape of transformation rules.

While comparing these tools with DSLTrans, only EMF Tiger and GReAT present a

[9] The Graphviz application available online at `http://www.graphviz.org/` is able to consume dot specifications and automatically render the contained graph in a pdf file

syntactic structure based on layers to specify its rules.

Then, we also defined and implemented an original language called SOS that can be used in order to define the operational semantics of languages, and automatically convert any SOS sentence into a graphical representation in the .dot language. While the former language (DSLTrans) is totally platform dependent, and is most useful for the DSML's implementation phase (i.e., the generation of the DSML's compiler), the SOS language is purely platform independent, and it seems to be useful for debugging DSMLs' design at the DSML's validation phase.

In the next Chapter, we combine the defined languages in the definition and implementation of several analysis methods, so that we are able to provide some guarantees of correctness for the software language engineer while using them. Intuitively, we can use a DSML's specification expressed in one of the languages (SOS), in order to validate another specification of the same DSML but expressed in the other language (i.e., DSLTrans).

5

Analysis of Translations

In this Chapter, we describe how can we analyse a given translation expressed in DSLTrans in order to assert about its validity. Namely what are the conclusions that we can take about translation's correctness. We start by describing how to symbolically execute a translation and how we can search the symbolic execution space for properties that we want to analyse. Then, we describe how to use the SOS definitions of both the languages involved in a given translation as oracles, in order to automatically validate each symbolic state in the symbolic execution space of a translation expressed in DSLTrans. Finally, we present the related work on translation validation, and detail the contributions of this research work on this particular subject.

5.1 Structural Analysis

Let us now define some useful functions for the construction of a transformation's symbolic space.

Definition 5.1. *Vertex Combinations*

Let $\{m, m'\} \subseteq TG$ *be two typed graphs,* $pairset \in \mathcal{P}(V \times V)$ *be a set of vertex pairs. Also, let* $V \subseteq V^m$ *and* $V' \subseteq V^{m'}$ *be sets of vertices contained in the vertices of* m *and* m' *respectively.*

The relation on Vertex Combinations *is defined for typed graphs* m *and* m' *as a relation between two sets of vertices and a sets of pairs of those vertices* $\xrightarrow{vc} \subseteq TG \times TG \times V \times V \times \mathcal{P}(V \times V)$*:*

(i)

$$\frac{}{\langle V, V' \rangle \xrightarrow{vc_{\langle m,m' \rangle}} \{\ \}}$$

(ii)

$$\frac{x \in V,\ y \in V',\ \tau_v^m(x) = \tau_v^{m'}(y)}{\langle V, V' \rangle \xrightarrow{vc_{\langle m,m' \rangle}} \{\ (x,\ y)\ \}}$$

(iii)

$$\frac{\begin{array}{c} x \in V,\ y \in V',\ \tau_v^m(x) = \tau_v^{m'}(y), \\ \langle V \setminus \{x\}, V' \setminus \{y\} \rangle \xrightarrow{vc_{\langle m,m' \rangle}} pairset \end{array}}{\langle V, V' \rangle \xrightarrow{vc_{\langle m,m' \rangle}} \{\ (x,\ y)\ \} \cup pairset}$$

The relation *Vertex Combinations* computes for two typed graphs, all the possible combinations of pairing together zero, two or more than two nodes. In order for the pairing to occur, the selected vertices have to have the same type.

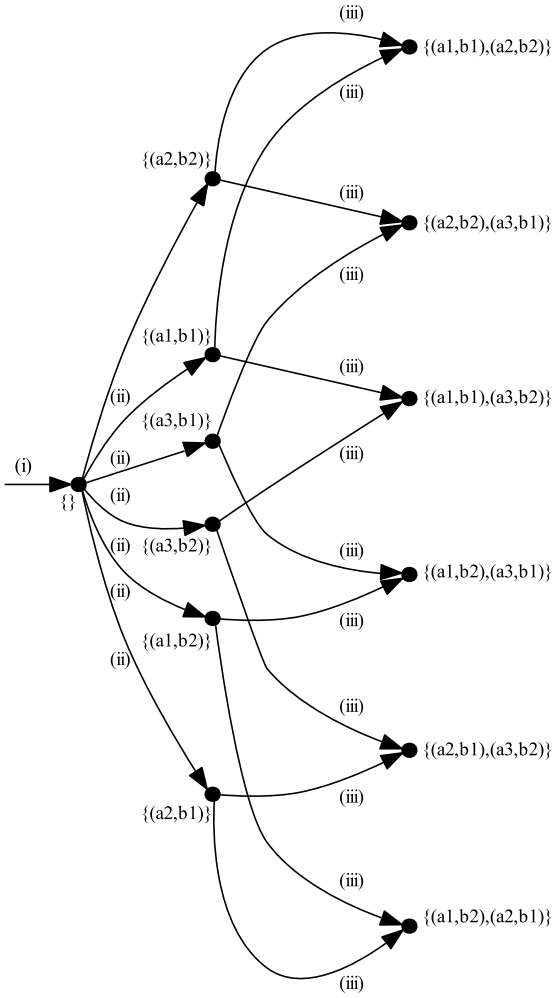

Figure 5.1: The *Vertex Combinations* relation between the vertices $a1$, $a2$ and $a3$ from typed graph a, and the vertices $b1$ and $b2$ from a typed graph b, given that they all have the same type.

For instance, consider two typed graphs named a and b. Typed graph a has three vertices: $a1$, $a2$, and $a3$. Typed graph b has two vertices: $b1$ and $b2$. Consider also that all

of these vertices have the same type. In other words: $\tau_v^a(a1) = \tau_v^a(a2) = \tau_v^a(a3) = \tau_v^b(b1) = \tau_v^b(b2)$. In this conditions, the *Vertex Combinations* relation between all of these vertices is shown in Figure 5.1. The presented graph shows the effect of the application of each of the rules defined above. The first rule *(i)* encodes the possibility of no pairing for any two sets of vertices V and V'. The second rule *(ii)* encodes the possibility of pairing one pair of vertices arbitrarily chosen from both vertex set V and V'. In our example, we can see that with rule *(ii)* there are six different choices of pairing up two vertices from each of the typed graphs a and b. Finally, the third rule *(iii)* encodes the possibility of pairing together two vertices (one from each of the vertex sets V and V') and recursively pair an arbitrary number of additional vertices from the same sets without having the possibility to select the same ones. In our example, we can see that starting with a previous choice, we can apply rule *(iii)* and get in total more six different choices of pairing up two vertices from each of the typed graphs a and b. Therefore, in the presented example, the relation *Vertex Combinations* $vc_{\langle a,b \rangle}$ contains a total of thirteen different combinations of vertices of typed graphs a and b.

Proposition 5.2. *Maximum Number of Possible Pairs*

Let $\{t, t'\} \subseteq TG$ be two typed graphs, and $V \subseteq V^t$ and $V' \subseteq V^{t'}$ be two sets of vertices belonging to each of graphs' vertex sets V^t and $V^{t'}$. Also let $n = max(|V|, |V'|)$ be the number of elements of the largest vertex set: either V or V'; and $m = min(|V|, |V'|)$ be the number of elements of the smallest vertex set: either V or V'.

The maximum number of possible pairs on two typed graphs (written $\xi_{\langle m,m' \rangle}$) is calculated on the assumption that all the vertices from both graphs are of the same type, and therefore they can be paired up together. The following sum reflects this calculation:

$$\xi_{\langle t,t' \rangle} = 1 \ + \ \Sigma_{p=1}^{m} \frac{n! \times m!}{p! \times (n \ - \ p)! \times (m \ - \ p)!}$$

Proof. Given a vertex set V of size n elements, a vertex set V' of size m elements, lets try to select p elements such that $p \leq m \leq n$. In this case, we can divide both sets V and V' by p, which means the number of possible p selections from the multiplication of the size of the elements from both sets — in other words, the number of possible combinations from the second rule:

$$\frac{n \times m}{p}$$

After this, we have to remove the one element from each of the sets (as shown in the third rule), and select $p \ - \ 1$ elements from each of the sets:

$$\frac{(n \ - \ 1) \times (m \ - \ 1)}{p \ - \ 1}$$

Multiplying the results of all of the combinations from both of these rules will give the result of selecting p elements from each of the sets:

$$\frac{n \times m}{p} \times \frac{(n-1) \times (m-1)}{p-1} \cdots \frac{(n-(p-1)) \times (m-(p-1))}{p-(p-1)}$$

Clearly, we can express the denominator in terms of factorial $p!$, and also express the numerator in terms of $n! \times m!$ and cancel out the the remainder of the multiplications $n - (p - 1)$ to 1 and $m - (p - 1)$ to 1 by dividing them using the factorials $(n - p)!$ and $(m - p)!$ respectively. Therefore, the number of combinations of selecting p elements from both sets of size n and m respectively is given by the following expression:

$$\frac{n! \times m!}{p! \times (n - p)! \times (m - p)!}$$

Then, we have to sum all of the possible combinations for an arbitrary number of p bounded to m, that since we cannot remove from a set more elements that its size, this will be the minimum of the sets: m. Therefore, each sum of the m-bounded series Σ reflects the number of combinations of selecting p elements from each of the sets, and pairing them together. Finally, we add the possibility of not pairing any element from both of the sets. □

Definition 5.3. *Transformation Rule Vertex Pairs*

Let $\{tr0, tr1\} \subseteq TR_t^s$*, be two transformation rules, and* $\{pairset, mpairset, applypairset\} \subseteq \mathcal{P}(V \times V)$ *be sets of vertex pairs.*
The function $RulePairs : TR_t^s \times TR_t^s \rightarrow \mathcal{P}(\mathcal{P}(V \times V))$ *is such that:*

$$\begin{aligned} RulePairs(tr0, tr1) \; = \; \big\{ pairset \in \mathcal{P}(V \times V) \mid \\ \langle V^{Match^{tr0}}, V^{Match^{tr1}} \rangle \xrightarrow{vc_{\langle Match^{tr0}, Match^{tr1} \rangle}} mpairset \wedge \\ mpairset = \bigcup_{i=1}^{n} (x_i, y_i) \wedge \\ pairset = \bigcup_{i=1}^{n} \big((x_i, y_i) \cup applypairset_{(x_i, y_i)} \big) \big\}, \end{aligned}$$

where $applypairset_{(x_i, y_i)}$ *is a set of pairs of vertices that is related with the following sets of vertices:*

$$V0_i = \{v0_i \mid (x_i \xrightarrow{backwardLink} v0_i) \in Bl^{tr0}\}$$

and

$$V1_i = \{v1_i \mid (y_i \xrightarrow{backwardLink} v1_i) \in Bl^{tr1}\},$$

by means of the Vertex Combinations relation on the apply parts of both transformations. In particular:

$$\langle V0_i, V1_i \rangle \xrightarrow{vc_{\langle Apply^{tr0}, Apply^{tr1} \rangle}} applypairset_{(x_i,y_i)}$$

$$\wedge$$

$$|applypairset_{(x_i,y_i)}| = min(|V0_i|, |V1_i|).$$

with the additional restriction that we only select the set of pairs $applypairset$ with a number of pairs equal to the minimum number of vertices from both of the sets $V0_i$ and $V1_i$.

The intuition of this function is, for a given pair of transformation rules, to collect the set of all combinations of pairing together the match vertices from both transformation rules that have the same type. For instance, with the transformation rules x and y presented in Figure 5.2, we can pair up the vertex A_1^x (vertex A_1 from graph x) with the vertex A_1^y (vertex A_1 from graph y), or pair up the vertex B_1^x with the vertex B_{1y}, or both pairings. In Definition 5.3, all of these pairings are collected in the set named $mpairset$. For each of these pairs, also we compute the set named $applypairset$ from the respective apply vertices which are connected from the vertices in these pairs by means of backward links. For instance, in the Figure 5.2, the vertex A_1^x has no apply vertices, and the vertex A_1^y has one apply vertex called C_3^y. Since the minimum of the apply vertices from each of the A_1 match vertices is zero, the set $applypairset$ on this case will be empty (see the first line in the bottom of the Figure 5.2). In the case where we pair together B_1 match vertices from both graphs, the minimum of the apply vertices from those B_1 match vertices is two—namely $|\{C_1^x, C_2^x\}| = 2$—hence all sets $applypairset$ will have exactly two pairs. Moreover, in total there will be six different combinations of pairing together exactly two of their respective apply vertices (see the remaining lines in the bottom of the Figure 5.2). The idea to fix the size of these sets $applypairset$ is to avoid useless and inappropriate pair combinations where the apply vertices of the transformation rule with the smallest ApplyPart is not completely paired up together with the ApplyPart of the other transformation rule.

Proposition 5.4. *Maximum Number of Possible Transformation Rule Pairs*

Given two transformation rules $\{tr0, tr1\} \subseteq TR_t^s$, we observe that the maximum number of possible vertex pairs on those transformation rules (written $\Xi_{\langle tr0,tr1 \rangle}$) results directly from the multiplication of the number of combinations from their match parts, and the number of combinations from their apply parts:

$$\Xi_{\langle tr0,tr1 \rangle} = \xi_{\langle Match^{tr0}, Match^{tr1} \rangle} \times \frac{n! \times p!}{p! \times (n - p)!}$$

where $p = min(|V^{Apply^{tr0}}|, |V^{Apply^{tr1}}|)$ is the size of the smallest vertex set from the apply parts of both transformation rules, and $n = max(|V^{Apply^{tr0}}|, |V^{Apply^{tr1}}|)$ is the size of the largest vertex set from the apply parts of both transformation rules.

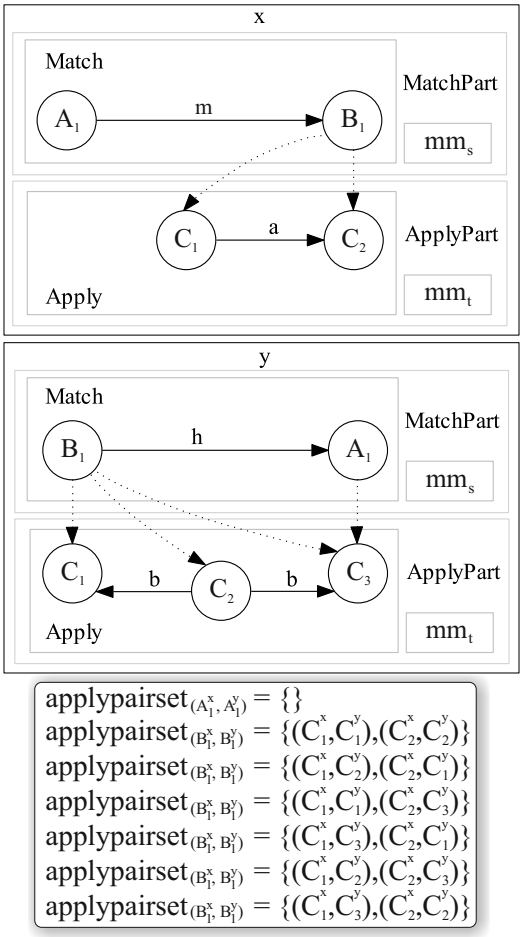

Figure 5.2: On top, two transformation rules named x and y. The backward links are represented as dashed arrows. Vertices of the same type are indexed with additional numbers. Both the transformation rules share the same source and target metamodels. On the bottom, the elements from the Vertex Combination relation w.r.t. pairing together B_1 elements from graphs x and y, are pairs of vertices.

Proof. The first part of the multiplication can be deduced from the maximum number of vertex combinations from the vc relation on the match vertices of both transformation rules. The second part corresponds to maximum number of vertex combinations again from the vc relation on the apply vertices of both transformations rules, but now we only select combinations of vertex pairs which size equals to the size of the smallest vertex set. This means that on the vertex combination calculation we only select p where $p = m = min(|V^{Apply^{tr0}}|, |V^{Apply^{tr1}}|)$ in the sum

$$\Sigma_{p=m}^{m} \frac{n! \times m!}{p! \times (n - p)! \times (m - p)!}.$$

□

Definition 5.5. *Collapse Rule Pairs*

Let $\{tr0, tr1\} \subseteq TR_t^s$ *be transformation rules, and* $pairset \in \mathcal{P}(V \times V)$ *be a set of vertex pairs. The function*
$CollapseRulePairs : TR_t^s \times TR_t^s \to \mathcal{P}(TR_t^s)$ *is defined for a pair of transformation rules such that*

$$CollapseRulePairs(tr0, tr1) = \big\{\langle V, E, \tau_v, \tau_e\rangle\big\},$$

where for any $pairset \in RulePairs(tr0, tr1)$ *we have*

1. $V = (V^{tr0} \cup V^{tr1}) \setminus \{y \mid (x, y) \in pairset\}$,
2. $E = (E^{tr0}|_V \cup E^{tr1}|_V) \cup \big\{(w, x) \mid (w, y) \in E^{tr1} \wedge (x, y) \in pairset\big\} \cup \big\{(x, w) \mid (y, w) \in E^{tr1} \wedge (x, y) \in pairset\big\} \cup \big\{(x, x) \mid (y, y) \in E^{tr1} \wedge (x, y) \in pairset\big\}$,
3. $\tau_v = \tau_v^{tr0}|_V \cup \tau_v^{tr1}|_V$,
4. $\tau_e = \tau_e^{tr0}|_E \cup \tau_e^{tr1}|_E$.

The intuition of the function $CollapseRulePairs$, is to generate a new transformation rule from each set of pairs computed by the $RulePairs$ function. For instance, the transformation rules named x and y presented before in Figure 5.2 are now used to generate a finite set of transformation rules according to the finite set of pairs computed by the $RulePairs$ function. In Figure 5.3, we present three of those combinations. On the top is the transformation rule generated from the case where all of the vertices are paired up except for C_1 which refers to the C_1^y vertex from transformation rule y. On the middle is the transformation rule generated from the case where all of the vertices are paired up except for C_2 (namely C_2^y). On the bottom of the Figure, is the transformation rule generated from the case where all of the vertices are paired up except for $A_{1\_}$ (namely A_1^x), A_1 (namely A_1^y) and C_3 (namely C_3^y).

Proposition 5.6. *Maximum Number of Possible Collapsed Transformation Rules from a Pair of Transformation Rules*

Given two transformation rules $\{tr0, tr1\} \subseteq TR_t^s$*, it is trivial to observe that the maximum number of possible collapsed transformation rules computed with the function* $CollapseRulePairs$ *in definition 5.5 results directly from the maximum number of possible* $pairset \in RulePairs(tr0, tr1)$ *— i.e.,*

$$|RulePairs(tr0, tr1)| = \Xi_{\langle tr0, tr1\rangle}.$$

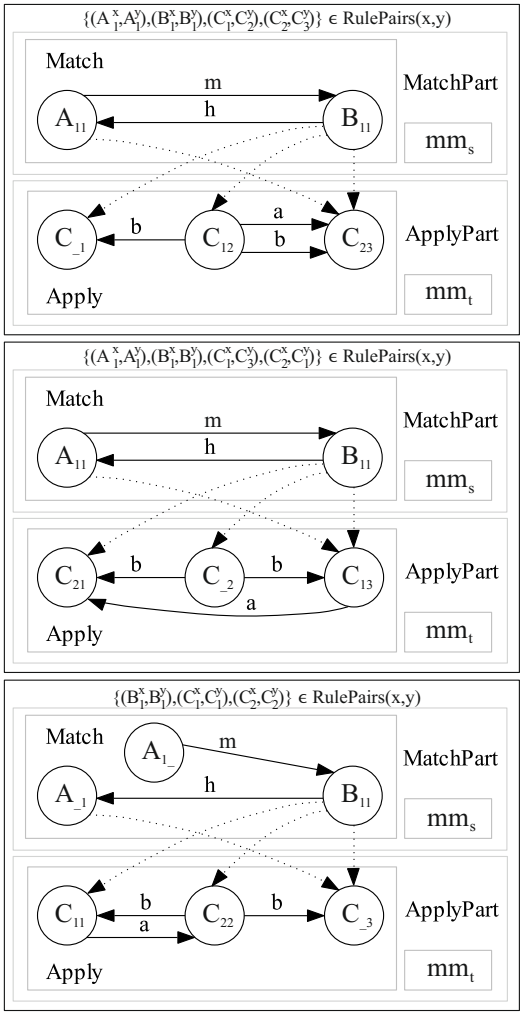

Figure 5.3: Three transformation rules, each one generated from a result of the $RulePairs$ function when applied to the transformation rules x and y presented in Figure 5.2.

Definition 5.7. *Collapse of a Set of Transformation Rules*

Let $\{past, tlayer, clayer, nlayer\} \subseteq \mathcal{P}(TR_t^s)$, be sets of transformation rules, and $\{tr_1, \ldots tr_i\} \subseteq TR_t^s$ be transformation rules, where $1 \leq i \leq n$.

The function $\Lambda : \mathcal{P}(TR_t^s) \rightarrow \mathcal{P}(TR_t^s)$ is recursively defined for a set of rules. In the case of a set with only one transformation rule:

$$(i)\ \Lambda(\{tr0\}) = \{tr0\},$$

and in the case of a set with more than one transformation rule:

$$(ii)\ \Lambda(\bigcup_{i=1}^{n} tr_i) = \bigcup_{tr_j \in \Lambda(\bigcup_{i=2}^{n} tr_i)} CollapseRulePairs(tr_1, tr_j),$$

where $tr_1 \in TR_t^s$ is the first transformation rule of the transformation rule set (given that we decomposed it into a ordered set union of n elements), and $tr_j \in \Lambda(\bigcup_{i=2}^{n} tr_i))$ is one of the results from the recursive computation of the function Λ on the remainder of the transformation set. It is trivial to observe that the transformation set monotonically decreases in the recursions, leading inevitably to the recursion base case on line (i).

The intuition of this function, is that we can collapse any set of transformation rules by collapsing together any pair of transformation rules, and further collapsing the results of those collapses with another rule of the initial transformation rule set, until there is no more rules to collapse.

Proposition 5.8. *The result of function Λ is finite*

Proof. We only have to prove that when we apply the function Λ to a set of transformation rules bigger than one—i.e., case (ii). From proposition 5.6 we know that the function $CollapseRulePairs$ will return a maximum number of possible collapse rules, and it is trivial to observe that this number is always finite. Therefore if the set of transformation rules has two elements, then the result is also finite. Since the function Λ is recursively defined as a union of finite results from the function $CollapseRulePairs$, where on each step of the recursion the set $\bigcup_{i=1}^{n} tr_i$, then we know that when we apply the function Λ to a set of transformation rules bigger than two, that the recursion will eventually stop, and the global number of collapsed rules returned by that function will also be finite. □

Definition 5.9. *Symbolic Execute*

The function $SymExecute : \mathcal{P}(TR_t^s) \times TR_t^s \to \mathcal{P}(TR_t^s)$, is defined for a given set of rules $past \in \mathcal{P}(TR_t^s)$ and one transformation rule $trc \in TR_t^s$ which is supposed to be the result of the Λ function. Here, we have to consider two cases. In the case where the set of backward links from the transformation rule trc is empty — i.e., $Bl^{trc} = \emptyset$ — then $SymExecute(past, trc) = \{trc\}$. Otherwise, in the case where the set of backward links from the transformation rule trc is not empty — i.e., $Bl^{trc} \neq \emptyset$ — then:

$$SymExecute(past, trc) = \{g_\Delta \sqcup g \sqcup h_\Delta \mid past_i \in past \wedge g \lhd past_{i|Bl^{past_i}} \wedge g \cong trc_{|Bl^{trc}}\},$$

where g_Δ is such that $(g_\Delta \sqcup g) \cong trc$, and h_Δ is such that $(h_\Delta \sqcup g) \cong past_i$.

5.1.1 State space

Definition 5.10. *Collapse Function*

Let $\{past, tlayer, clayer, nlayer\} \subseteq \mathcal{P}(TR_t^s)$*, be sets of transformation rules. The function* $Collapse : \mathcal{P}(TR_t^s) \times \mathcal{P}(TR_t^s) \rightarrow \mathcal{P}(TR_t^s)$ *computes the complete set of collapsed transformation rules from an existing set of transformation rules such that:*

$$\begin{aligned} Collapse(past, tlayer) = \{future \in TR_t^s \mid \\ clayer \in \mathcal{P}(tlayer), \\ nlayer \in \Lambda(clayer), \\ future \in SymExecute(past, nlayer)\} \end{aligned}$$

Proposition 5.11. *Finiteness of the result of the Collapse function*

Let $\{past, tlayer\} \subseteq \mathcal{P}(TR_t^s)$*, be sets of transformation rules. The result of the function* $Collapse(past, tlayer)$ *is always a finite set of graphs, where each graph in that set have a finite set of nodes.*

Proof. We first need to prove that $clayer \in \mathcal{P}(tlayer)$ is finite: this is trivial since the powerset of a finite set is also a finite set. Then we need to prove that $nlayer \in \Lambda(clayer)$ is also finite, which can be directly concluded by the result of proposition 5.8. Finally, we need to prove that $future \in SymExecute(past, nlayer)$ is finite, which can also be directly observed since by definition 5.9, this function is based on both the subgraph on transformation rules and typed graph equivalence relations, which are by Definition 4.5 and Definition 4.20 relations between finite graphs. □

We now build the symbolic space for a transformation by gathering all the combinations of transformations for each layer, the result of collapsing them, and building the state space.

Definition 5.12. *Symbolic Space*

Let $\{past, tlayer, nlayer, trset\} \subseteq \mathcal{P}(TR_t^s)$*, be sets of transformation rules,* $tr \in TR_t^s$ *be a transformation rule, and* $trans \in Transformation_t^s$ *be a transformation. The transformation symbolic space*

$SymSpace \subseteq \mathcal{P}(TR_t^s) \times Transformation_t^s \times \mathcal{P}(\ TR_t^s\ )$ *is the least set that satisfies the following rules:*

$$\frac{}{\langle past, [] \rangle \xrightarrow{SymSpace} \{\}}$$

$$\frac{\begin{array}{l} nlayer = back(Collapse(past, tlayer)), \\ \langle nlayer, R \rangle \xrightarrow{SymSpace} trset \end{array}}{\langle past, [tlayer :: R] \rangle \xrightarrow{SymSpace} nlayer \cup trset}$$

The function $computeSymSpace : Transformation_t^s \rightarrow \mathcal{P}(TR_t^s)$ *uses the above relation in order to compute the symbolic space of transformation* $trans$*:*

$$computeSymSpace(trans) = \{tr \in TR_t^s \mid \langle\{\}, trans\rangle \xrightarrow{SymSpace} trset \ \wedge \ tr \in trset\}$$

In this case the $back : \mathcal{P}(TR_t^s) \rightarrow \mathcal{P}(TR_t^s)$ function is applied recursively for each transformation rule in $clayer$ by creating new backward links that connect together all match nodes of each transformation rule with its free apply nodes, in such a way that simulates the application of the whole layer on an arbitrary model. Notice also that since $clayer$ is a set of transformation rules where each one results from the union of several other transformation sub-rules, the $back$ function is applied on each sub-rule while preserving the sub-rule individuality.

When we apply the above defined $computeSymSpace$ function to the transformation specification presented in Listing 4.1, we get a set of around 11k collapsed transformation rules. Examples of those collapsed transformation rules generated from this transformation specification are shown in Figure 5.4 and Figure 5.5. It is easy to observe which transformation rules of the transformation specification presented in Listing 4.1 were used in order to compose the presented examples, by means of the dashed horizontal lines that connects the elements from the Statemachine sentences with PetriNet sentences.

The $computeSymSpace$ function explores all of the relevant combinations of the patterns defined in the transformation specification. We will further analyse all of these combinations using both of the semantics definitions of the StateMachine and PetriNet Languages, in order to validate the presented transformation.

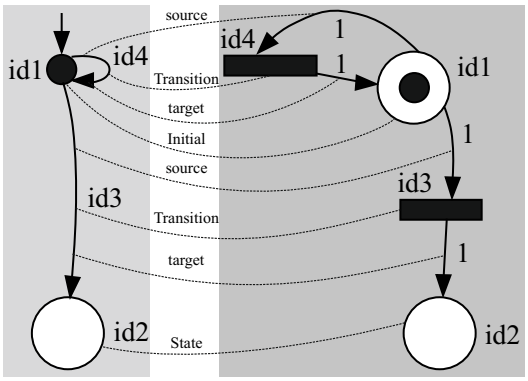

Figure 5.4: An example of a collapsed transformation rule from the transformation specification presented in Listing 4.1. The left side of the Figure represents the match part of the transformation rule, and the right side of the Figure represents the apply part of the transformation rule. This collapsed transformation rule is a result of the collapse of the nodes from the all of the following transformation rules: $Initial$+$(2 \times Transition)$+$State$+$(2 \times source)$+$(2 \times target)$.

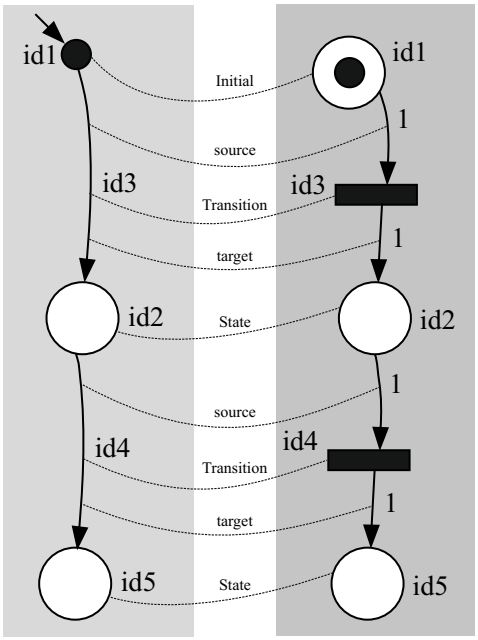

Figure 5.5: An example of a collapsed transformation rule from the transformation specification presented in Listing 4.1. The collapsed transformation rule is a result of the collapse of the nodes from the all of the following transformation rules: $Initial$+$(2 \times Transition)$+$(2 \times State)$+$(2 \times source)$+$(2 \times target)$.

Proposition 5.13. *Finiteness of the transformation symbolic space*

Let $[l_1 \dots l_n] \in Transformation_t^s$ *be a transformation. The result of the computation of the transformation symbolic space* $computeSymSpace([l_1 \dots l_n])$ *is finite.*

Proof. Let us start by proving by induction on the inference rules of definition 5.12 that the amount of states produced for each layer $l_1 \dots l_n$ is finite. The state space is recursively computed using these inference rules, where in each step of the recursion, we take the head of the list (i.e., $tlayer$) and compute the set of collapsed transformation rules $nlayer$, which size is (as shown in proposition 5.11) a finite number. Therefore, the recursion will eventually stop in a finite amount of steps. Since a finite union of finite sets is also a finite set, the transformation symbolic space $computeSymSpace([l_1 \dots l_n])$ is also finite. □

The result in proposition 5.13 is crucial since by definition model checking can only be performed on finite state spaces.

5.1.2 Structural Checking

One of the uses for the symbolic space generated by the rules of definition 5.12 is to check structural properties.

Definition 5.14. *Property*

A Property is a 8-tuple $\langle V, E, \tau_v, \tau_e, MatchPart, ApplyPart, Bl, Il\rangle$*, where* $MatchPart = \langle Match, s\rangle$ *and* $ApplyPart = \langle Apply, t\rangle$*.*

A property is in fact a match-apply model with a special kind of edges labeled as $indirectLink$ *in both the match and apply parts. On the one hand, if we remove these edges, we have that* $MatchNoIl = \langle V^1, E^1 \setminus Il, \tau_v^1, \tau_e^1\rangle$ *is a model w.r.t.* s*,* $MatchPartNoIl = \langle MatchNoIl, s\rangle$*,* $ApplyNoIl = \langle V^2, E^2 \setminus Il, \tau_v^2, \tau_e^2\rangle$ *is a model w.r.t.* t *and* $ApplyPartNoIl = \langle ApplyNoIl, t\rangle$*. Therefore* $\langle V, E \setminus Il, \tau_v, \tau_e, MatchPartNoIl, ApplyPartNoIl, Bl\rangle \in MAM_t^s$ *is a match-apply model. On the other hand,* $Match = \langle V^3, E^3, \tau_v^3, \tau_e^3\rangle$*,* $Apply = \langle V^4, E^4, \tau_v^4, \tau_e^4\rangle$*, and the edges* $Il \subseteq E^3$ *or* $Il \subseteq E^4$ *are called* indirect links*, which means that for all* $i \in Il$ *it is true that either* $\tau_e(i) = \tau_e(i)^3 = indirectlink$ *or* $\tau_e(i) = \tau_e(i)^4 = indirectlink$*.*

The set of all properties having source metamodel s *and target metamodel* t *is called* $Property_t^s$*.*

The language to describe properties is in fact very similar to the language to express transformations, with the additional possibility of expressing indirect links in the $apply$ pattern—thus allowing more abstract patterns than the ones expressed in transformations. This is natural given that the properties of a transformation can be more abstract than the rules implementing them. A property can be *satisfiable*, *unsatisfiable* or *non provable*. We start with the definition of a state in a state space (formally defined as a transformation) being model of a property. As a reminder, each state of the state space is a symbolic representation of a set of models given as input to the transformation being validated and their corresponding transformations. In fact, a state holds a set of patterns that should be instantiated in the input model — the $match$ part of the state — as well as in the output model — the $apply$ part of the state. By validating a property at the level of the symbolic states, we validate it for the whole set of input and output models of a given transformation. Despite the fact that structural checking is an important feature to be explored and delivered to the software language engineer while verifying his/her translation, in this thesis we will not focus our attention on this feature. Further references and examples of structural checking on DSLTrans translations can be found in [LBA10].

Definition 5.15. *Model of a Property*

A transformation rule $tr = \langle V^{tr}, E^{tr}, \tau_v^{tr}, \tau_e^{tr}, Match^{tr}, Apply^{tr}, Bl^{tr}, Il^{tr} \rangle \in TR_t^s$ *is a model of a property* $p = \langle V^p, E^p, \tau_v^p, \tau_e^p, Match^p, Apply^p, Bl^p, Il^p \rangle = P \in Property_t^s$*, written* $tr \vDash^s p$ *if:*

1. $\langle V^p, E^p \setminus Il^p, \tau_v^p, \tau_e^p \rangle \blacktriangleleft\cong \langle V^{tr}, E^{tr}, \tau_v^{tr}, \tau_e^{tr} \rangle$
2. *if* $x^p \xrightarrow{indirectLink} y^p \in Il^p$ *then there exists* $x^{tr} \xrightarrow{label} y^{tr} \in (E^{tr})^*$ *where* $\tau_v^p(x^p) = \tau_v^{tr}(x^{tr})$, $\tau_v^p(y^p) = \tau_v^{tr}(y^{tr})$ *and* $(E^{tr})^*$ *is obtained by the transitive closure of* E^{tr}*.*

Definition 5.16. *Satisfiable Property*

Let $trans = [l_1 :: \ldots :: l_n] \in Transformation_t^s$ *be a transformation, and* $tr \in TR_t^s$ *be a transformation rule computed from the transformation analysis.*

The transformation $trans$ satisfies *property* $p \in Property_t^s$*, written* $trans \vDash p$*, where:*
$trans \vDash p \Leftrightarrow \exists\, tr \in computeSymSpace(trans) \cdot tr \vDash^s p$

Definition 5.17. *Unsatisfiable Property*

Let $trans = [l_1 :: \ldots :: l_n] \in Transformation_t^s$ *be a transformation, and* $tr \in TR_t^s$ *be a transformation rule computed from the transformation analysis.*

The transformation $trans$ do not satisfies *property* $p \in Property_t^s$*, written* $trans \nvDash p$*, where:*

$$trans \nvDash p \Leftrightarrow \exists\, tr \in computeSymSpace(trans) \;\cdot\; tr \vDash^s \; match(p) \;\wedge\; tr \nvDash^s \; p$$

Note that the projection function $match$ returns the match pattern of a property. Informally, the property's $match$ pattern is found in a given symbolic state, but the $apply$ pattern of the property is not satisfied.

Definition 5.18. *Non Provable Property*

Let $trans = [l_1 :: \ldots :: l_n] \in Transformation_t^s$ *be a transformation, and* $tr \in TR_t^s$ *be a transformation rule computed from the transformation analysis.*

Property $p \in Property_t^s$ *is non provable w.r.t. transformation* $trans$*, written* $trans \nVdash p$*, where:*
$trans \nVdash p \Leftrightarrow \nexists\, tr \in computeSymSpace(trans) \;\cdot\; tr \vDash^s \; match(p)$

Again informally, the $match$ pattern can never be found in any state of the symbolic space of $trans$.

5.1.3 DSLTrans' Structural Analysis Tool

This approach for validating translations expressed in DSLTrans is depicted in Figure 5.6. Since the number of symbolic states is a finite number, we can query the symbolic space resulting from the symbolic execution of the translation. The structural properties expressed (at the bottom of the Figure) are queried on the resulting (finite) symbolic state of a given DSLTrans Transformation. The satisfaction procedure then follows the formalization just presented: if the source pattern of a structural property is found included in a given symbolic state, then the corresponding apply pattern should also be included in the respective apply pattern of that symbolic state.

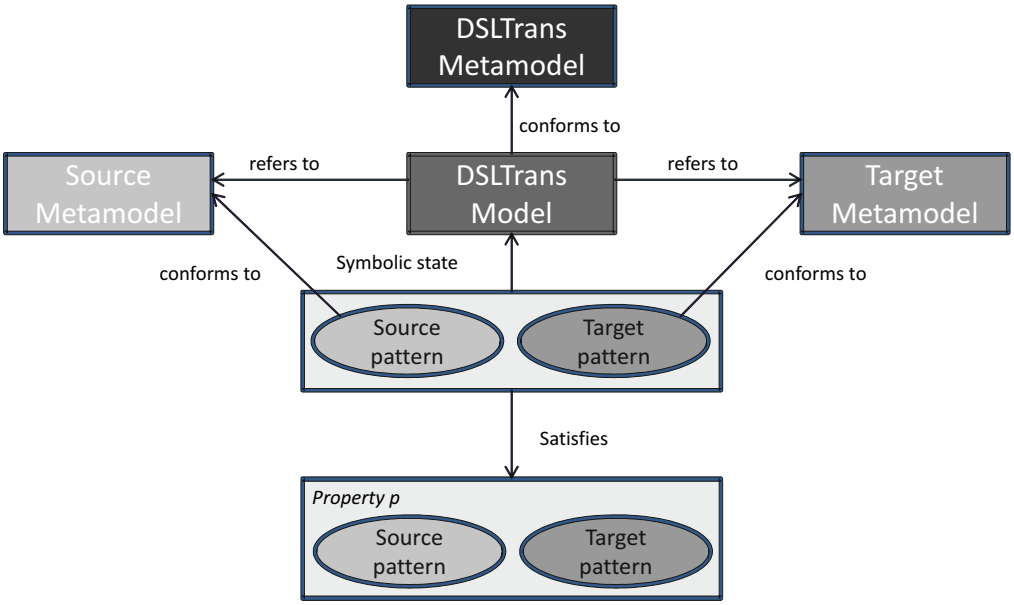

Figure 5.6: A framework for validating translations expressed in DSLTrans based on the satisfaction of properties.

Given the fact that DSLTrans is a metamodeled language (which means that its syntactic model is expressed by means of a metamodel), the implementation of this structural analysis tool also explored this fact. This is shown in Figure 5.7, where it is shown the interaction between both DSLTrans and Properties Editors with the structural analysis tool in order to assert the validity of the defined translations by generating their symbolic execution space, and using the defined properties as oracles. The depicted numbers inside circles denote a logical order of events in time. Notice that the Properties Language extends the DSLTrans Language by introducing the capability of expressing indirect links in the *apply model*—the syntax of DSLTrans was slightly extended in order to allow indirect links on the apply patterns—this allowed the expression of properties, and their checking.

As we can see, the structural analysis tool was modeled and implemented solely by means of a translation that converts every construct in the extended version of DSLTrans

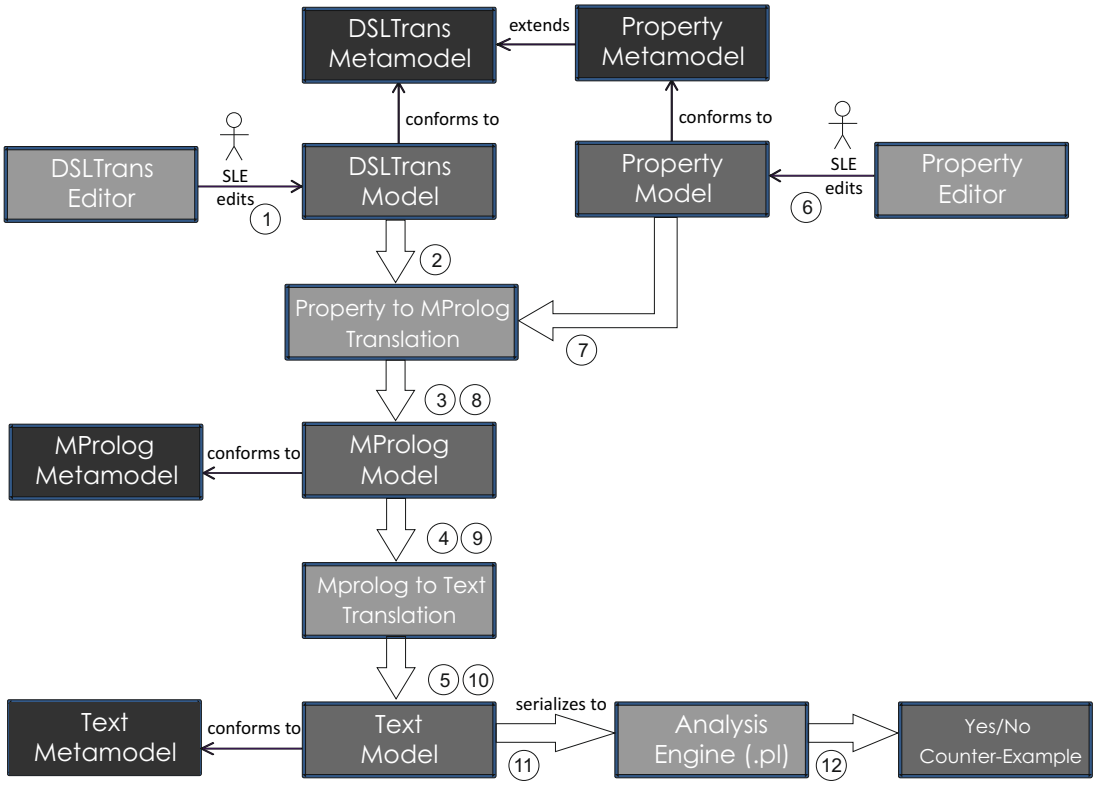

Figure 5.7: The reference implementation of the Structural Analysis Tool as a set of Eclipse plug-ins.

(denoted Properties) into prolog clauses so that they can then be manipulated freely in a relational fashion (i.e., DSLTrans' constructs are encoded into relational entities, and their associations are encoded in to relational relations). Furthermore, this translation is in fact called an high-order transformation (as previously defined in [TCJ10]), since it was also specified in DSLTrans and transforms DSLTrans transformation models into a meta-modeled version of prolog called MProlog and further translated into the Text language, which is already very close to textual code (in this case prolog code). In our reference implementation, the relational version of DSLTrans is then manipulated in a program written in prolog (denoted Analysis Engine), that closely follows the rules described in the presented formalization.

5.2 Semantic Analysis

In this section, we start by presenting the formal definitions of the analysis algorithm of DSLTran's translations using both of the source and target language semantics expressed in SOS specifications, and then we explain how the analysis algorithm was realized in a tool, and how this tool is integrated in the overall MDD methodology for DSML compiler's design, implementation, verification and validation.

5.2.1 The Analysis Algorithm

We now define what is a bisimulation relation between semantic domains of two different languages. We will then use this relation in order to validate a given translation.

Definition 5.19. *Mapper Function and its inverse*

Let $tr \in TR_t^s$ be a transformation, $X \subseteq V$ and $Y \subseteq V$ are finite sets of vertices.

The $mapper : TR_t^s \to (\mathcal{P}(V) \to \mathcal{P}(V))$ function, is such that

$mapper(tr) =$

$$\{(X \to Y) \mid X \subseteq V^{Match^{tr}} \wedge \forall x \in X \cdot (x \xrightarrow{backwardLink} y) \in Bl^{tr} \implies y \in Y\}$$

Its inverse is the function $mapper^{-1} : TR_t^s \to (\mathcal{P}(V) \to \mathcal{P}(V))$ such that

$mapper^{-1}(tr) =$

$$\{(Y \to X) \mid Y \subseteq V^{Apply^{tr}} \wedge \forall y \in Y \cdot (x \xrightarrow{backwardLink} y) \in Bl^{tr} \implies x \in X\}$$

Informally the mapper function converts any transformation into a mapping function such that for a given set of match vertices it returns their correspondent apply vertices, according to the defined backward links in that transformation. Conversely, the inverse of the mapper function converts any transformation into a mapping function such that for a given set of apply vertices it returns their correspondent match vertices, according to the defined backward links in that transformation.

Definition 5.20. *Notion of Bisimulation Relation*

Let $g \in TG$ be a typed graph, $ts^g \in TS$ be a transition system, where $ts^g = \mathcal{P}(\langle PreState, \langle Label, V\rangle, PosState\rangle)$, $\{p, q, PreState, Label, Label', PosState, PosState'\} \subseteq Term$ are arbitrary terms with no variable references, and $V, V' \subseteq V^g$ are finite sets of vertices of typed graph g.

The Bisimulation is a relation written $\sim_{ts^g} \subseteq TS \times Term \times Term$ between terms which were defined on a particular transition system ts^g. If we pick arbitrary p and q, we can say that $p \sim_{ts^g} q$ if and only if all of the following conditions are satisfied:

1. *there exists either $\langle p, \langle Label, V\rangle, PosState\rangle \in ts^g$ or $\langle PreState, \langle Label', V'\rangle, p\rangle \in ts^g$, or both.*

2. *there exists either* $\langle q, \langle Label, V\rangle, PosState\rangle \in ts^g$ *or* $\langle PreState, \langle Label', V'\rangle, q\rangle \in ts^g$, *or both.*

3. *for all* $\langle p, \langle Label, V\rangle, PosState\rangle \in ts^g$, *there exists* $\langle q, \langle Label', V'\rangle, PosState'\rangle \in ts^g$, *and* $PosState \sim_{ts^g} PosState'$.

4. *for all* $\langle q, \langle Label, V\rangle, PosState\rangle \in ts^g$, *there exists* $\langle p, \langle Label', V'\rangle, PosState'\rangle \in ts^g$, *and* $PosState \sim_{ts^g} PosState'$.

The first two conditions basically say that p and q are terms belonging to the states of the transition system ts^g. The third condition says that given that p and q are bisimilar related, then for every move (i.e., every outgoing transition in the transition system ts^g) starting from p there must also be a matching move starting from p. The matching move is written in the above definition as: the next term state $PosState$ starting from p must be also bisimilar related with the next term state $PosState'$ starting from q. Finally, the forth condition is similar to the third one but now it says that all the possible moves starting from q should also be matched with moves starting from p.

Definition 5.21. *General Bisimulation Relation*

Let $tr \in TR_t^s$ *be a transformation defined for metamodels* $s \in MM$ *and* $t \in MM$, *and let* $\{g, g'\} \subseteq TG$ *be two models, such that* $g \vdash s$ *and* $g' \vdash t$. *Also let* $\{ts^g, ts^{g'}\} \in TS$ *be their respective transition systems produced according to some operational semantics,* $\{p, q, PreState, P_0, P_1, Label, Label', PosState, PosState'\} \subseteq Term$ *are arbitrary terms with no variable references, and* $X, V \subseteq V^g$ *is a finite set of vertices of typed graph* g, *and* $V', Y \subseteq V^{g'}$ *is a finite set of vertices of typed graph* g.

The General Bisimulation is a relation written $\sim_{(ts^g, ts^{g'}, tr)} \subseteq TS \times TS \times TR_t^s \times Term \times Term$ *between terms which were defined on the two transition systems* ts^g *and* $ts^{g'}$.
$p \sim_{(ts^g, ts^{g'}, tr)} q$ *means that all of the following conditions are satisfied:*

1. *there exists either* $\langle p, \langle Label, X\rangle, PosState\rangle \in ts^g$ *or* $\langle PreState, \langle Label', X'\rangle, p\rangle \in ts^g$, *or both.*

2. *there exists either* $\langle q, \langle Label, Y\rangle, PosState\rangle \in ts^{g'}$ *or* $\langle PreState, \langle Label', Y'\rangle, q\rangle \in ts^{g'}$, *or both.*

3. *for all* $x_0 \in \{ P_0 \mid \langle P_0, \langle Label, V\rangle, PosState\rangle \in computeInitial(ts^g) \}$, *there exists an* $y_0 \in \{ P_1 \mid \langle P_1, \langle Label', V'\rangle, PosState'\rangle \in computeInitial(ts^{g'}) \}$, *such that* $x_0 \sim_{(ts^g, ts^{g'}, tr)} y_0$.

4. *for all* $x_0 \in \{ P_0 \mid \langle P_0, \langle Label, V\rangle, PosState\rangle \in computeInitial(ts^{g'}) \}$, *there exists an* $y_0 \in \{ P_1 \mid \langle P_1, \langle Label', V'\rangle, PosState'\rangle \in computeInitial(ts^g) \}$, *such that* $x_0 \sim_{(ts^g, ts^{g'}, tr)} y_0$.

5. *for all* $\langle p, \langle Label, X\rangle, PosState\rangle \in ts^g$*, there exists a* $\langle q, \langle Label', mapper(tr)(X)\rangle, PosState'\rangle \in ts^{g'}$*, such that* $PosState \sim_{(ts^g, ts^{g'}, tr)} PosState'$.

6. *for all* $\langle q, \langle Label, Y\rangle, PosState\rangle \in ts^{g'}$*, there exists a* $\langle p, \langle Label', mapper^{-1}(tr)(Y)\rangle, PosState'\rangle \in ts^g$*, such that* $PosState \sim_{(ts^g, ts^{g'}, tr)} PosState'$.

The General Bisimulation Relation, extends the notion of Bisimulation Relation by using a transformation tr in order to relate together two different transition systems as they were two parts of the same transition system. Notice that the Bisimulation relation presented in Definition 5.20 is a relation between two (possibly disjoint) parts of the same transition system ts^g. Here the two different transition systems (i.e., one from each language) are brought together by means of the $mapper$ and $mapper^{-1}$ functions produced from the transformation tr. Notice that these functions take a transformation as a parameter and return mapping functions from sets of vertices to sets of vertices.

Lemma 5.22. *Reflexivity, Symmetry, and Transitivity* $\sim$

The General Bisimulation relation $\sim$ *between different languages is an equivalence relation.*

Proof. To be an equivalence relation it has have the following properties: reflexivity, symmetry and transitivity. We will follow show each one these properties:

1. **Reflexivity:** Lets assume that we have a typed graph $g \in TG$, a transition system defined on that graph $ts^g \in TS$ an identity transformation $id \in TR_t^s$, such that $\forall V \subseteq V^g \cdot mapper(id)(V) = mapper^{-1}(id)(V)$. Within these conditions it is trivial to observe that the identity relation $\mathcal{R} = \{(s, s) \mid (\langle s, \langle Label, X\rangle, PosState\rangle) \in ts^g \vee (\langle PreState, \langle Label, X\rangle, s\rangle) \in ts^g\}$ is a bisimulation relation for (ts^g, ts^g, id). In other words since $\mathcal{R}$ satisfies all of the six conditions on definition 5.21, then $\mathcal{R} \subseteq \sim_{(ts^g, ts^g, id)}$.

2. **Symmetry:** Lets assume that we have two typed graphs $\{g, g'\} \subseteq TG$, two transition systems defined on those graphs respectively $\{ts^g, ts^{g'}\} \subseteq TS$, and $tr \in TR_t^s$. Also, lets assume that $\mathcal{R} \subseteq \sim_{(ts^g, ts^{g'}, tr)}$. If we consider the relation $\mathcal{R}^{-1} = \{(s', s) \mid (s, s') \in \mathcal{R}\}$, which is obtained by swapping the states of any pair in $\mathcal{R}$, and $tr^{-1} \in TR_s^t$ which is also obtained by swapping the match vertices with the apply ones on the given transformation rule tr — i.e., $mapper(tr^{-1})(V) = mapper^{-1}(tr)(V)$ and conversely $mapper^{-1}(tr^{-1})(V) = mapper(tr)(V)$ — then $\mathcal{R}^{-1} \subseteq \sim_{(ts^{g'}, ts^g, tr^{-1})}$ is also a bisimulation relation because *(i)* the first four conditions on definition 5.21 are obviously true, and *(ii)* the last two conditions are true by their symmetric nature.

3. **Transitivity:** Let assume that both $R_{1,2} \subseteq \sim_{(ts^g, ts^{g'}, tr_{1,2})}$ and $R_{2,3} \subseteq \sim_{(ts^{g'}, ts^{g''}, tr_{2,3})}$ are bisimulations. Then, in this case, we need to prove that there also exists a relation $R = \{(s_1, s_3) \mid \exists\ (\langle s_2, \langle Label, X\rangle, PosState\rangle \in ts^{g'} \vee \langle PreState, \langle Label, X\rangle, s_2\rangle \in ts^{g'})\ \cdot\ (s_1, s_2) \in R_{1,2}\ \wedge\ (s_2, s_3) \in R_{2,3}\}$ which is also a bisimulation. In other words, $R \subseteq \sim_{(ts^g, ts^{g''}, tr_{1,3})}$, where $tr_{1,3}$ is such that for any $V \subseteq V^g$ it is true that $mapper(tr_{1,3})(V) = mapper(tr_{2,3})(\ mapper(tr_{1,2})(V))$, and for any $V'' \subseteq V^{g''}$ it is true that $mapper^{-1}(tr_{1,3})(V'') = mapper^{-1}(tr_{1,2})(mapper^{-1}(tr_{2,3})(V''))$.

   This can be demonstrated by checking all of the conditions for a bisimulation. The first two conditions of definition 5.21 are true by definition. For the next two conditions, lets first consider for any initial state $s_1 \in \{\ PreState \mid (\langle PreState, \langle Label, V\rangle, PosState\rangle) \in computeInitial(ts^g)\ \}$. Then on the one hand, since $R_{1,2}$ is a bisimulation, we know that there exists a $s_2 \in \{\ PreState \mid (\langle PreState, \langle Label, V\rangle, PosState\rangle) \in computeInitial(ts^{g'})\ \}$ such that $(s_1, s_2) \in R_{1,2}$. And, on the other hand, since $R_{2,3}$ is also a bisimulation, we know that there exists a $s_3 \in \{\ PreState \mid (\langle PreState, \langle Label, V\rangle, PosState\rangle) \in computeInitial(ts^{g''})\ \}$ such that $(s_2, s_3) \in R_{2,3}$. Therefore, in w.r.t. this condition we can say that $(s_1, s_3) \in R_{1,3}$. In the same line of thought, we can also say that for any $s_3 \in \{\ PreState \mid (\langle PreState, \langle Label, V\rangle, PosState\rangle) \in computeInitial(ts^{g''})\ \}$, there is an initial state $s_1 \in \{\ PreState \mid (\langle PreState, \langle Label, V\rangle, PosState\rangle) \in computeInitial(ts^g)\ \}$ such that $(s_1, s_3) \in R_{1,3}$. Finally, for the last two conditions, if we assume that $(s_1, s_3) \in R_{1,3}$, then on the one hand, since $(s_1, s_2) \in R_{1,2}$, it follows that $(\langle s_1, \langle Label, X\rangle, s_1'\rangle) \in ts^g$, and there exists $(\langle s_2, \langle Label', mapper(tr_{1,2})(X)\rangle, s_2'\rangle) \in ts^{g'}$, where $s_1' \sim_{(ts^g, ts^{g'}, tr_{1,2})} s_2'$; and on the other hand, since $(s_2, s_3) \in R_{2,3}$ and we consider that $Y = mapper(tr_{1,2})(X)$, then it follows that $(\langle s_2, \langle Label', Y\rangle, s_2'\rangle) \in ts^{g'}$, and there exists $(\langle s_3, \langle Label'', mapper(tr_{2,3})(Y)\rangle, s_3'\rangle) \in ts^{g''}$, where $s_2' \sim_{(ts^{g'}, ts^{g''}, tr_{2,3})} s_3'$. Therefore, it is true that $s_1' \sim_{(ts^g, ts^{g''}, tr_{1,3})} s_3'$ or in other words: $(s_1', s_3') \in R_{1,3}$. The proof for the last condition is similar to this one, using the function $mapper^{-1}$.

□

Definition 5.23. *Semantic Validity of a Translation*

Let $m \in Transformation_t^s$ be a transformation defined for metamodels $\{s, t\} \subseteq MM$. Also consider $sos_s \in SOS_s$ to be an operational semantics definition for language s, $sos_t \in SOS_t$ to be an operational semantics definition for language t, $\{P_0, P_1, Label, Label', PosState\} \subseteq Term$ are arbitrary terms with no variable references, and V, V' are finite sets of vertices. We say that the transformation m is a valid translation if and only if for all $tr \in computeSymSpace(m)$, there exists a relation $x_0 \sim_{(ts^{Match^{tr}}, ts^{Apply^{tr}}, tr)} y_0$, where

1. $ts^{Match^{tr}} = computeFixPoint(sos_s, Match^{tr})$;
2. $ts^{Apply^{tr}} = computeFixPoint(sos_t, Apply^{tr})$;

3. $x_0 \in \{\ P_0 \mid \langle P_0, \langle Label, V\rangle, PosState\rangle \in computeInitial(ts^{Match^{tr}})\ \}$;

4. $y_0 \in \{\ P_1 \mid \langle P_1, \langle Label', V'\rangle, PosState\rangle \in computeInitial(ts^{Apply^{tr}})\ \}$;

Informally, the checking algorithm to validate a particular translation m will first compute the symbolic space of the translation for each of the $match$ and $apply$ patterns specified in m. The algorithm will compute their respective transition systems, and finally check the general bisimulation relation on their initial states. There can be several options on the implementation of this algorithm, for instance, instead of computing the whole transition system for each one of the languages, one could compute individual states on both languages and check the bisimilarity relation starting from the initial states on both languages' transition systems. Notice that the computation of the complete transition system may not terminate — e.g., in the Petri Nets example shown in Section 4.3 we might have infinitely many relations between markings.

Back to our running example, the checking algorithm will in this case compute the transformation symbolic space using the $computeSymSpace$ function as shown in Figures 5.4 and 5.5. Then for each collapsed transformation rule, it will unfold the transition system of both its match and apply patterns using the function $computeFixPoint$ resulting in the transition systems shown in the bottom of both Figures 5.8 and 5.9. For readability purposes, we used on the one hand the term $m(\{(id1, 1), (id2, 0)\})$ as an abbreviation of $marking(p(id1, suc(zero)), marking(p(id2, zero), e))$ in Figure 5.8, and on the other hand the term $m(\{(id1, 1), (id2, 0), (id5, 0)\})$ as an abbreviation for the algebraic value $marking(p(id1, suc(zero)), marking(p(id2, zero), marking(p(id5, zero), e)))$.

The final step is then to check if these transition systems are bisimilar. In the example from Figure 5.8 we can note that the $mapper$ function is formed by the set:

$$\begin{aligned}\{&(\{id1\}, \{id1\}),\\ &(\{id2\}, \{id2\}),\\ &(\{id1, id2, id3\}, \{id3\}),\\ &(\{id1, id1, id4\}, \{id4\})\}\end{aligned}$$

Also, in the example from Figure 5.9 the $mapper$ function is formed by the following set:

$$\begin{aligned}\{&(\{id1\}, \{id1\}),\\ &(\{id2\}, \{id2\}),\\ &(\{id1, id2, id3\}, \{id3\}),\\ &(\{id2, id5, id4\}, \{id4\}),\\ &(\{id5\}, \{id5\})\}\end{aligned}$$

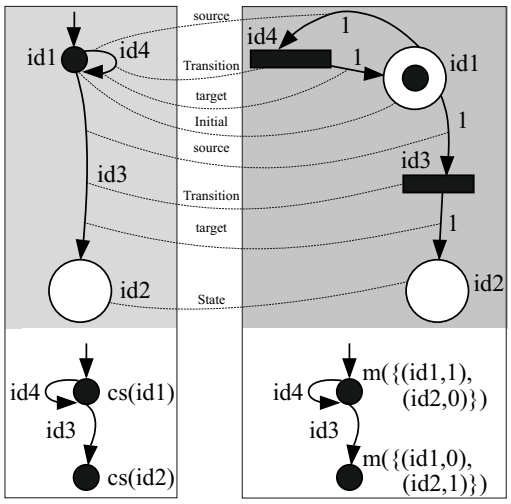

Figure 5.8: An example of a collapsed transformation rule from the transformation specification presented in Listing 4.1 (on top), and their respective transition systems (on bottom).

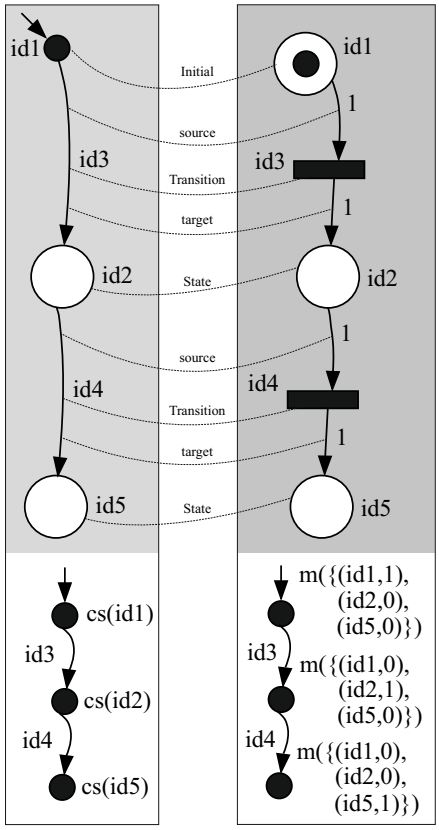

Figure 5.9: An example of a collapsed transformation rule from the transformation specification presented in Listing 4.1 (on top), and their respective transition systems (on bottom).

5.2.2 Methodology and Tool

The general framework proposed for our validation approach presented in Chapter 3, in particular in Figure 3.10, can now be instantiated in Figure 5.10. The framework is instantiated with two languages: DSLTrans for expressing software language translations, and SOS for expressing the abstract semantics of the involved languages in a platform independent way.

This instantiation specifically supports a particular kind of methodology, where the software language engineer defines both the syntax and semantics of all of the involved languages in the most platform independently way, namely, by means of metamodels (syntax) and SOS model (semantics). During the language development, the language engineer might feel the need to **translate** towards a particular platform which already has a language with appropriate level of abstraction, and most importantly, both the syntax and semantics of this language is also completely defined in a similar way, namely, by

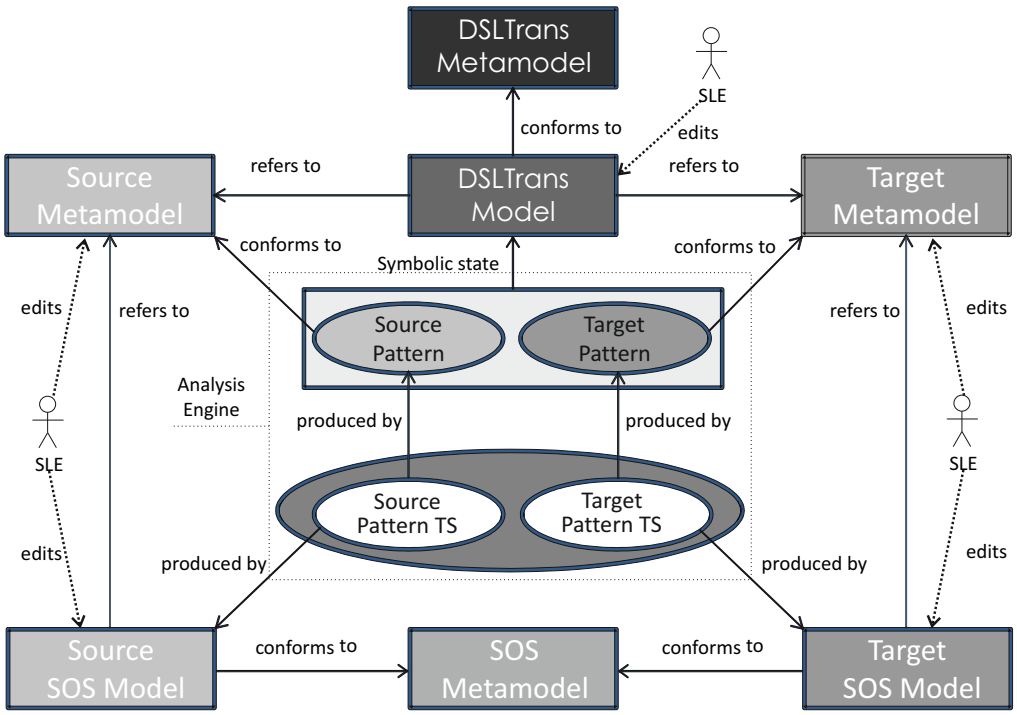

Figure 5.10: The instantiation of the proposed framework for validating software language translations.

means of metamodels (syntax) and SOS model (semantics). If this translation can be also suitably formalized/expressed using a DSLTrans Model, then the semantic analysis tool can use both of the involved language's semantic definitions in order to validate the translation, before it is automatically realized in a DSML compiler.

As shown in Figure 5.11, this methodology is also aligned with the goals of Model Driven Development (MDD) of tackling complexity (e.g., platform dependency) by having several intermediate levels of abstraction and small (and most importantly analysable) translations between them. Here it also important that each intermediate level of abstraction is completely formalized by means of languages in what respects to their syntax and semantics, preferably in the most platform independent way—so that these specifications can be effectively reused among platforms. Notice that the depicted numbers inside circles denote a logical order of events in time. Also notice that in the Figure 5.11, the arrows labeled as step 3 (*serializes to*), are in fact hiding all the intermediate translation events described before in Figures 4.15 and 4.20. Moreover, these events must occur simultaneously in order to produce the input parameters of the implemented analysis engine in prolog.

The semantic analysis tool [1] internally uses the semantic definitions of both the source and target languages (expressed in the SOS language) involved in a DSLTrans' translation

[1]The DSLTrans' analysis tool is an open-source project available publicly at: https://github.com/githubbrunob/DSLTransGIT/tree/master/dsltransAnalysis

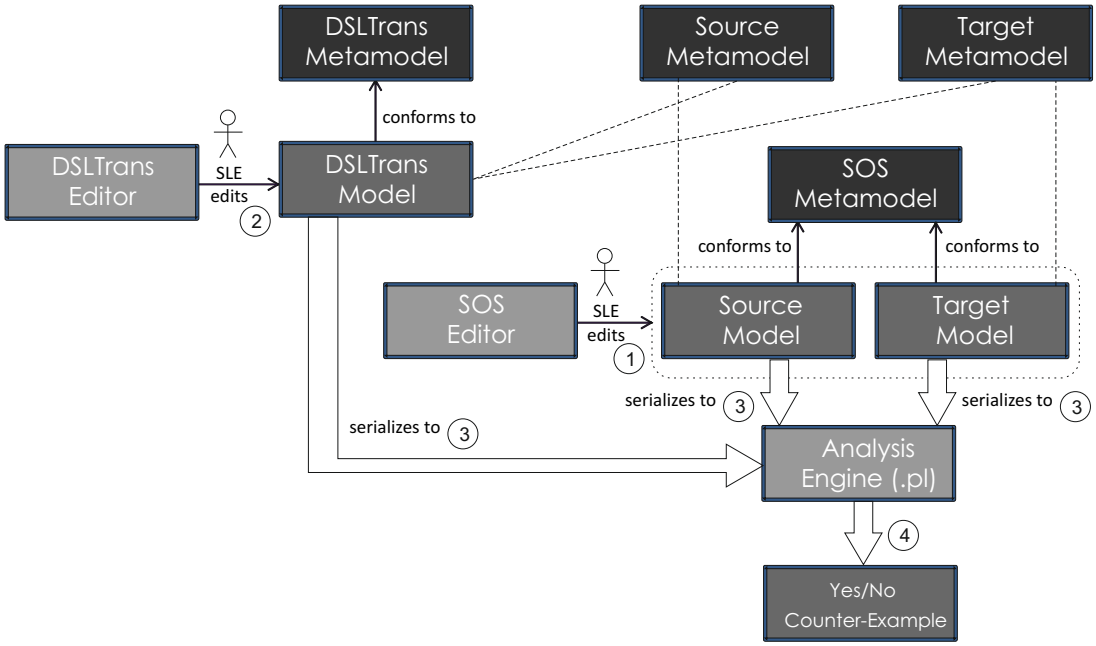

Figure 5.11: The reference implementation of the Semantic Analysis Tool as a set of Eclipse plug-ins.

under analysis in order to produce, for each source and target pattern on each symbolic state of that translation, their respective canonical representations as transition systems (TS), and compare them in a common ground. According to the languages involved this comparison procedure may not reach a valid conclusion as we will further discuss in Chapter 7.

Furthermore, the presented semantic analysis tool itself was also developed following the principles of MDD. Firstly, the translation under analysis is itself translated into a relational representation expressed as prolog facts. Secondly, the SOS specifications of both of the source and target languages referred in the translation, are also translated into a similar relational representation expressed as prolog facts. The checking program is completely written in prolog, and it manipulates the translation under analysis and the SOS specifications as graphs, in a very similar ways as presented in the formalization described in this Chapter. Actually, the main concern was to provide a proof of concept by means of an implementation that is very close to the formalization, hence tackling soundness and eventual computation problems that typical mathematical formalizations usually neglect. However, the aspect of computational performance was not addressed—i.e., the analysis process might need a huge amount of both time and memory, in order to return a satisfactory result.

Notice that all of the described translations were expressed in DSLTrans. Given that we used a metamodeled version of prolog (named MProlog) as target of all of our described translations, we can ultimately analyse this high-order transformation by the same means—it is just a question of providing a suitable well-understood semantics of

prolog. These translation is however quite trivial, and therefore no further analysis procedure was taken.

5.3 Conclusions and Related Work

We have introduced a language (DSLTrans) for expressing translations which do not allow the expression of any kind of recursion on its syntax—here we presented a similar formalization to what was published in [BLA+10]. This restriction in the language and the identification of its properties (i.e., confluence and termination) allowed the design of a verification mechanism for this kind of translations, and its associated model checking tool [LBA10]—here we presented a slightly different formalization based on the implementation of the verification mechanism in prolog.

Similarly to our approach, the authors of [NK08] enable the declaration of a syntactic structural correspondence between terms in source and target languages. However, they use this structural correspondence to automatically verify the results at the end of each transformation. With this approach, the quality engineer will only realize that the transformation is invalid when some pair of models input/output violates the declared structural correspondence.

Finally, we also provide a framework with its respective languages and tools to automatically validate translations expressed in DSLTrans w.r.t. both the source and target operational semantics [BA11]. The authors in [AvdBE12] also present a similar framework. However they do not present any concrete implementation. Instead, the presented framework can be used as a reference to further implementations of specialized theorem provers that are able to symbolically validate a translation. Therefore they did not felt the computation problems associated with model transformations' validation, nor the need to restrict their MTL in order to avoid them.

In our framework, we are able to use any DSML's semantic specification expressed in SOS, as an oracle, in order to validate another specification of the same DSML expressed in DSLTrans. The reason to do so, and not the other way around (i.e., using DSLTrans, to validate SOS specifications) is that DSLTrans have properties (i.e., termination and confluence) that makes it analysable. In this particular case, the analysability property comes from the fact that the resulting symbolic execution space of any DSLTran's translation is finite. The analysability of the SOS language is however a topic under research: one could find a way to, for instance, restrict the expressiveness of SOS in order to make it also analysable.

Nevertheless, the validation of model transformations (and in particular language translations) is a very difficult task to be performed by a software language engineer. Therefore, the software language engineer has a recognized need for well founded (formalized and language-based) tool support for SLE specify translations, automatically

validate them, and generate their respective compilers.

However, the success of the presented methodology mostly depends on an optimal implementation of the presented analysis tool. The next Chapter tries to analyze and evaluate the expected success of presented methodology in light of a concrete case study.

6

Case Study: A Language for Role Playing Games

In order to demonstrate how the presented methodology can be used in practice, we will introduce a concrete case study with a realistic application, and then discuss the limitations and borders of the validation method in the application context of the language engineering of Domain Specific Modeling Languages (DSMLs) in general.

6.1 Language Overview

We selected as a case study the DSML for Role Playing Games (RPG). The RPG DSML introduced in [MBB+12] was specifically designed to enable game designers to specify their RPGs, analyse their correctness by means of powerful analysis algorithms and data structures, and finally automatically deploy them in a given computational platform. Figure 6.1, details the RPG framework that provides both execution and analysis support to the RPG game designers: the Corona Framework [1], and the Algebraic Petri Nets (APN) language used in the AlPiNA framework [HML+12], respectively. Notice that this involved several intermediate translations between intermediate levels of abstraction. The model to model transformations (denoted in the Figure as 'M2M') were originally expressed in the ATL transformation language [2], and the model to code transformations (denoted in the Figure as 'M2C') were expressed using the XPand [3] language. Here the

[1] http://www.anscamobile.com/corona/
[2] http://www.eclipse.org/atl/
[3] http://wiki.eclipse.org/Xpand

authors distinguish 'code' from 'model' whenever we generate artifacts that are no longer in the XMI format nor conforming with an EMF metamodel. Notice also that having orthogonal transformations towards different platforms naturally brings the need to assure the overall consistency of the framework—i.e., how do we prove that the APN analysis model is semantically equivalent with the generated code in the Corona Framework? In [ABC12], it is presented a vision of the methodology that tries to solve this challenging question by specifically using both DSLTrans and the associated verification methods presented in this thesis.

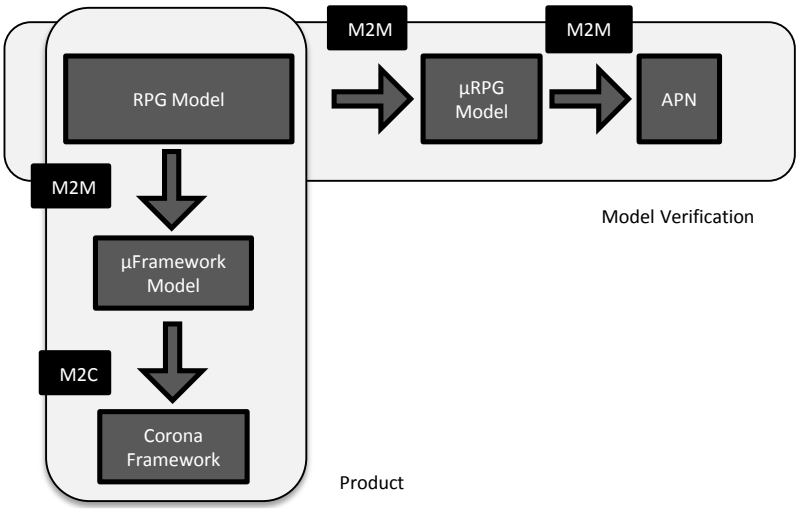

Figure 6.1: The RPG Framework as introduced in [MBB+12]

The language itself is small and simple, having only the necessary concepts to specify an RPG: the game map (which includes scenes that contain cells inside), the hero, agents (which can be friendly or enemies), items (which can be keys, doors, etc), dialogues (with multiple choice conditions on the answers), and challenges or goals. For the sake of clarity, we restricted our case study by focusing only on the translations related to the Model Verification (i.e., the horizontal transformation path depicted in Figure 6.1). Also instead of using the APN language, we used the Petri Nets language defined before in Chapter 3. Notice that the Petri Nets language was developed as the simplification of APNs specifically to be presented in this thesis, while maintaining all of the essential features of APNs. Finally, instead of starting directly from the RPG language, we instead start from the language depicted in Figure 6.1, which is a smaller version of the RPGs restricted to the relevant set of concepts that are to be analysed in the APN analysis platform. For instance, in order to analyse the possible paths of the players during a given game, we focus on concepts such as the hero, the doors, the keys to open doors, and their cell positions. Therefore, the RPG language metamodel that we present in Figure 6.2, only presents Cells, Keys, Doors and Heros as the main concepts of the language—for instance, Challenges are not represented.

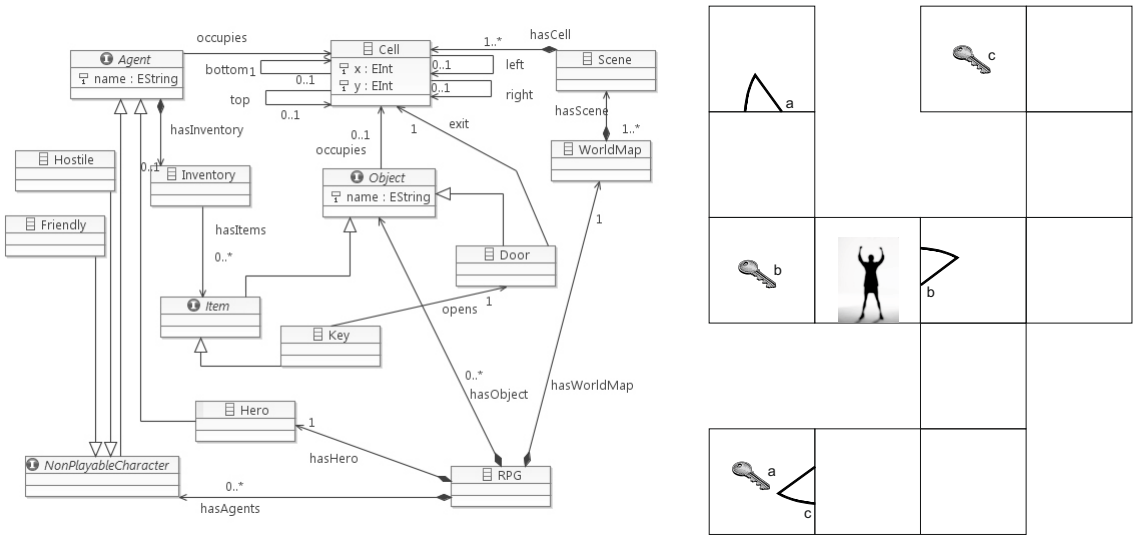

Figure 6.2: The metamodel of the RPG Language (on the left). An RPG sentence expressed in a simple visual editor (on the right), representing a maze with cells, doors, keys, and the hero's initial position.

6.2 Experimental Report

In this Case Study, we want to evaluate the presented methodology by applying it on the above presented RPG DSML and its translation to Petri Nets. In particular, we want to observe: *i)* if the DSLTrans language is able to specify the translation from RPGs to Petri Nets; *ii)* if the SOS language is able to specify both the RPG and Petri Nets languages; and finally *iii)* if the presented verification tool is able to effectively determine the validity of the DSLTrans Translation. Therefore, after specifying the RPG's semantics (both operational and by translation to the Petri Nets language), we show the results of the verification tool when applied to the specified translation.

6.2.1 RPG's Semantics Specification

The operational semantics of the RPG language (when restricted to these concepts) can be described using our SOS language, as shown in Listing 6.1. The semantic domain (i.e., the algebraic structure that we used to represent the states in the resulting transition system) is a pair containing the Cell which represents where the Hero currently is, and the set of Keys that the Hero currently have.

The first rule (from lines 10 to 27 on the left column) says that the hero can always move left (denoted as the transition labeled as al) if its current Cell is adjacent with another one on the left, and there is no Door occupying it. Also note that the new state maintains the same set of keys for the hero. Similarly, the second rule (from lines 28 to 44 on the left column) says the same for the top movement, where the transition was in this case labeled as at.

Listing 6.1: SOS semantic definition of the RPG Language

```
Semantics

ADT RGPState
    Sorts rpgstate
    Generators
        hero: class("rpg","Cell")
              Set(class("rpg","Key"))
              -> rpgstate;

Assuming
    @state ->> @movement ->> hero(@c1 @keyset)
        in Transition_System,
    in(@c1,Model)=true,
    in(@c1 -> left -> @c2,Model)=true,
    in(@d,Model)=true,
    in(@d -> occupies -> @c2,Model)=false
Then
    hero(@c1 @keyset) ->> al ->> hero(@c2 @keyset)
        in Transition_System
Where
    state: rpgstate;
    c1: class("rpg","Cell");
    c2: class("rpg","Cell");
    d: class("rpg","Door");
    movement: char;
    keyset: Set(class("rpg","Key"));

Assuming
    @state ->> @movement ->> hero(@c1 @keyset)
        in Transition_System,
    in(@c1,Model)=true,
    in(@c1 -> top -> @c2,Model)=true,
    in(@d,Model)=true,
    in(@d -> occupies -> @c2,Model)=false
Then
    hero(@c1 @keyset) ->> at ->> hero(@c2 @keyset)
        in Transition_System
Where
    state: rpgstate;
    c1: class("rpg","Cell");
    c2: class("rpg","Cell");
    movement: char;
    d: class("rpg","Door");
    keyset: Set(class("rpg","Key"));

// skipping: the same for bottom and right movements

Assuming
    @state ->> @movement ->> hero(@c1 @keyset)
        in Transition_System,
    in(@k,@keyset)=true,
    in(@k -> opens -> @d,Model)=true,
    in(@c1 -> left -> @c2,Model)=true,
    in(@d -> occupies -> @c2,Model)=true,
    in(@d -> exits -> @c3,Model)=true
Then
    hero(@c1 @keyset) ->> al ->> hero(@c3 @keyset)
        in Transition_System
Where
    state: rpgstate;
    c1: class("rpg","Cell");
    c2: class("rpg","Cell");
    c3: class("rpg","Cell");
    movement: char;
    d: class("rpg","Door");
    k: class("rpg","Key");
    keyset: Set(class("rpg","Key"));

Assuming
    @state ->> @movement ->> hero(@c1 @keyset)
        in Transition_System,
    in(@k,@keyset)=true,
    in(@k -> opens -> @d,Model)=true,
    in(@c1 -> top -> @c2,Model)=true,
    in(@d -> occupies -> @c2,Model)=true,
    in(@d -> exits -> @c3,Model)=true
Then
    hero(@c1 @keyset) ->> at ->> hero(@c3 @keyset)
        in Transition_System
Where
    state: rpgstate;
    c1: class("rpg","Cell");
    c2: class("rpg","Cell");
    c3: class("rpg","Cell");
    movement: char;
    d: class("rpg","Door");
    k: class("rpg","Key");
    keyset: Set(class("rpg","Key"));

// skipping: the same for bottom and right movements


Assuming
    @state ->> @movement ->> hero(@c1 @keyset)
        in Transition_System,
    in(@k,Model)=true,
    in(@k -> occupies -> @c1,Model)=true
Then
    hero(@c1 @keyset) ->> ak ->>
        hero(@c1 Union({@k}, @keyset) )
        in Transition_System
Where
    state: rpgstate;
    c1: class("rpg","Cell");
    movement: char;
    k: class("rpg","Key");
    keyset: Set(class("rpg","Key"));

Assuming
    in(@hero,Model)=true,
    in(@c1,Model)=true,
    in(@hero -> occupies -> @c1,Model)=true,
    in(@c1 -> left -> @c2,Model)=true,
    in(@d,Model)=true,
    in(@d -> occupies -> @c2,Model)=false
Then
    hero(@c1 @keyset) ->> al ->> hero(@c2 @keyset)
        in Transition_System
Where
    state: rpgstate;
    hero: class("rpg","hero");
    c1: class("rpg","Cell");
    c2: class("rpg","Cell");
    d: class("rpg","Door");
    movement: char;
    keyset: Set(class("rpg","Key"));

Assuming
    in(@hero,Model)=true,
    in(@c1,Model)=true,
    in(@hero -> occupies -> @c1,Model)=true,
    in(@c1 -> top -> @c2,Model)=true,
    in(@d,Model)=true,
    in(@d -> occupies -> @c2,Model)=false
Then
    hero(@c1 @keyset) ->> at ->> hero(@c2 @keyset)
        in Transition_System
Where
    state: rpgstate;
    hero: class("rpg","hero");
    c1: class("rpg","Cell");
    c2: class("rpg","Cell");
    movement: char;
    d: class("rpg","Door");
    keyset: Set(class("rpg","Key"));

// skipping: the same for bottom and right movements
```

Notice that for the sake of brevity we omitted the bottom and right movements as

RPG Element	Petri Net Element
RPG	PetriNet
Cell with no Hero	Place with Token=0
Cell with Hero	Place with Token=1
Key in some Cell	$\text{Transition}_{getKey} \xrightarrow{outArc} \text{OutArc} \xrightarrow{sourcePlace} \text{Place}_{Cell}$, $\text{Transition}_{getKey} \xrightarrow{inArc} \text{InArc} \xrightarrow{targetPlace} \text{Place}_{Key}$ (*)
Cell (a) is adjacent to another Cell (b) without a Door	$\text{Transition}_{move} \xrightarrow{outArc} \text{OutArc} \xrightarrow{sourcePlace} \text{Place}_{Cell_{(a)}}$, $\text{Transition}_{move} \xrightarrow{inArc} \text{InArc} \xrightarrow{targetPlace} \text{Place}_{Cell_{(b)}}$
Cell (a) is adjacent to another Cell (b) with a Door that exits to Cell (c)	$\text{Transition}_{move} \xrightarrow{outArc} \text{OutArc} \xrightarrow{sourcePlace} \text{Place}_{Cell_{(a)}}$, $\text{Transition}_{move} \xrightarrow{outArc} \text{OutArc} \xrightarrow{sourcePlace} \text{Place}_{Key}$, $\text{Transition}_{move} \xrightarrow{inArc} \text{InArc} \xrightarrow{targetPlace} \text{Place}_{Cell_{(c)}}$ (**)

(*) The place $Place_{Key}$ stores the key as a token.

(**) $Transition_{move}$ is created between Places from Cells (a) and (c) and their respective InArc and OutArc, and also an InArc and OutArc connecting this transition with the Place associated with the Key that opens the referred Door.

Table 6.1: Translation table between the RPG Language and the Petri Net Language.

they are similar to the top and left movements. The third and fourth rules (respectively the remaining lines in the left column, and from lines 1 to 22 on the right column) say that there can be a movement al (or at in the case of the fouth rule) if the current Cell of the Hero is adjacent with another one on the left (or top), and there is a Door occupying, then as long as the Hero has the Key that opens it, the Hero will move to the Cell where the Door exits. The fifth rule (from lines 26 to 40) say that if the Hero is currently on a Cell that has a Key, then it can pick it up and put it in its set of keys—the transition is labeled as ak. The remaining rules refer to the initial movements of the Hero (i.e., when we do not require to have already a transition before in the Transition System).

The presented semantic description constitutes a formal reference for the operational semantics of the RPG language, which then can be used as a model for a compiler, or in the case we will now present, for a translation to a PetriNet language.

The informal description of the transformation between the RPG Language and the PetriNet Language is presented in Table 6.1. Note also that in this informal description, it is implicit that all InArcs and OutArcs have the same $weight = 1$, and also that all the generated Places and Transitions will be contained inside the PetriNet model. The actual transformation expressed in DSLTrans is then presented in Listing 6.2. This transformation specification is formed by two layers (the 'Entities' and 'Associations') which basically implements what was informally presented in Table 6.1.

Listing 6.2: The first version of the DSLTrans transformation from RPG Language to PetriNets Language (Entities Layer).

```
File
    id = _
    uri = 'models\\RPG.xmi'
    metamodel(
                mmname = rpg.Rpg
                uri = 'models\\RPG.ecore'
            )
def 'Entities' : layer  'Entities'
    previous = ''
    output = ''
    metamodel(
            mmname = petrinet.Petrinet
            uri = 'models\\PetriNet.ecore'
        )
    rule'Cell Free'
        match with
    cl0:
            any  rpg::Cell(at0: name at1: x at2: y )
    cl1:
            any  rpg::Hero
            subject to
                cl1 !-(occupies)-> cl0
        apply
    cl2:
             petrinet::Place(
                        self = 'Cell'
                        name= concat(sameAs(at0) with
                              concat(sameAs(at1) with
                              sameAs(at2)))
                        token= '0'
                    )
    end rule

    rule'Key'
        match with
    cl3:
            any  rpg::Key(  at3 :  name )
        apply
    cl4:
             petrinet::Place(
                        self = 'Key'
                        name= sameAs(at3)
                    )
    cl5:
             petrinet::Transition(
                        self = 'Key'
                        name= sameAs(at3)
                    )
    cl6:
             petrinet::InArc(
                        weight= '1'
                    )
            subject to
                cl6 --(targetPlace)-> cl4
                cl5 --(inArc)-> cl6
    end rule

    rule'RPG'
        match with
    cl7:
            any  rpg::RPG
        apply
    cl8:
             petrinet::PetriNet(
                        self = 'RPG'
                    )
    end rule

    rule'Cell Occupied'
        match with
    cl9:
            any  rpg::Cell(at4: name at5: x at6: y)
    cl10:
            any  rpg::Hero
            subject to
                cl10 --(occupies)-> cl9
        apply
    cl11:
             petrinet::Place(
                        self = 'Cell'
                        name= concat(sameAs(at4)
                              with concat(sameAs(at5)
                              with sameAs(at6)))
                        token= '1'
                    )
    end rule

end def
```

Listing 6.3: The first version of the DSLTrans transformation from RPG Language to PetriNets Language (Associations Layer).

```
def 'Associations' : layer  'Associations'
    previous = 'Entities'
    output = 'models\\pn.xmi'
    metamodel(
            mmname = petrinet.Petrinet
            uri = 'models\\PetriNet.ecore'
        )
    rule'hasCell'
        match with
    cl12:
            any  rpg::RPG
    cl13:
            any  rpg::WorldMap
    cl14:
            any  rpg::Scene
    cl15:
            any  rpg::Cell
            subject to
                cl12 --(hasWorldMap)-> cl13
                cl13 --(hasScene)-> cl14
                cl14 --(hasCell)-> cl15
        apply
    cl16:
             petrinet::PetriNet
    cl17:
             petrinet::Place
            subject to
                cl16 --(places)-> cl17
        restrictions
            cl16 derived from cl12
            cl17 derived from cl15
    end rule

    rule'hasObject'
        match with
    cl18:
            any  rpg::RPG
    cl19:
            any  rpg::Key
            subject to
                cl18 --(hasObject)-> cl19
        apply
    cl20:
             petrinet::PetriNet
    cl21:
             petrinet::Transition
    cl22:
             petrinet::Place
            subject to
                cl20 --(transitions)-> cl21
                cl20 --(places)-> cl22
        restrictions
            cl20 derived from cl18
            cl21 derived from cl19
            cl22 derived from cl19
    end rule
```

```
    rule'left'
        match with
    cl23:
            any  rpg::Cell
    cl24:
            any  rpg::Cell
    cl25:
            not rpg::Door
    cl26:
            any  rpg::RPG
            subject to
                cl23 --(left)-> cl24
                cl25 !-(occupies)-> cl24
        apply
    cl27:
             petrinet::Place
    cl28:
             petrinet::Place
    cl29:
             petrinet::Transition(
                        name= 'moveLeft'
                    )
    cl30:
             petrinet::InArc(
                        weight= '1'
                    )
    cl31:
             petrinet::OutArc(
                        weight= '1'
                    )
    cl32:
             petrinet::PetriNet
            subject to
                cl31 --(sourcePlace)-> cl27
                cl30 --(targetPlace)-> cl28
                cl29 --(inArc)-> cl30
                cl29 --(outArc)-> cl31
                cl32 --(transitions)-> cl29
        restrictions
            cl27 derived from cl23
            cl28 derived from cl24
            cl32 derived from cl26
    end rule

\\ skipping right, top and bottom:
\\    same as left
```

```
    rule'leftDoor'
3       match with
    cl63:
            any  rpg::Cell
    cl64:
            any  rpg::Cell
8   cl65:
            any  rpg::Door
    cl66:
            any  rpg::Cell
    cl67:
13          any  rpg::Key
    cl68:
            any  rpg::RPG
            subject to
                cl63 --(left)-> cl64
18              cl65 --(occupies)-> cl64
                cl65 --(exit)-> cl66
                cl67 --(opens)-> cl65
        apply
    cl69:
23          petrinet::Place
    cl70:
            petrinet::Place
    cl71:
            petrinet::Transition(
28                  name= 'moveLeft'
                )
    cl72:
            petrinet::InArc(
                    weight= '1'
33              )
    cl73:
            petrinet::OutArc(
                    weight= '1'
                )
38  cl74:
            petrinet::Place
    cl77:
            petrinet::PetriNet
            subject to
43              cl73 --(sourcePlace)-> cl69
                cl72 --(targetPlace)-> cl70
                cl71 --(inArc)-> cl72
                cl71 --(outArc)-> cl73
                cl77 --(transitions)-> cl71
48      restrictions
            cl69 derived from cl63
            cl70 derived from cl66
            cl74 derived from cl67
            cl77 derived from cl68
53  end rule

\\ skipping rightDoor, topDoor and bottomDoor:
\\     same as leftDoor
```

```
    rule'getKey'
3       match with
    cl123:
            any  rpg::Cell
    cl124:
            any  rpg::Key
8           subject to
                cl124 --(occupies)-> cl123
        apply
    cl125:
            petrinet::Place
13  cl126:
            petrinet::Transition
    cl127:
            petrinet::InArc(
                    weight= '1'
18              )
    cl128:
            petrinet::OutArc(
                    weight= '1'
                )
23          subject to
                cl128 --(sourcePlace)-> cl125
                cl127 --(targetPlace)-> cl125
                cl126 --(inArc)-> cl127
                cl126 --(outArc)-> cl128
28      restrictions
            cl125 derived from cl123
            cl126 derived from cl124
    end rule

33 end def
```

6.2.2 RPG to Petri Nets Translation Analysis

At this point, with the presented translation specification, we can use it in order to configure the DSLTran's execution engine to work as a compiler of the RPGs language to the target language of Petri Nets. However, the truth is that in the end, we are never sure if there is some conceptual mistake in the informal description itself, or in its interpretation into the translation specification expressed in DSLTrans. In fact, after applying the validation method over the specified translation presented in Listings 6.2 and 6.3, it returns the following counter-example shown in Figure 6.3. In this case it resulted from the collapse of the following transformation rules: $'RPG' + (4 \times' hasCell') + (2 \times' hasObject') + 'CellOccupied' + (3 \times' CellFree') + 'right' + 'top' + 'bottom' + 'leftDoor' + 'getKey'$. If we compare their transition systems (in the bottom of the Figure), it is

easy to see that they are not bisimilar, given that the mapper function is the following set: $\{(\{id4\},\{id4\}),\ (\{id2\},\{id2\}),\ (\{id5\},\{id1\}),\ (\{id3\},\{id3\}),\ (\{id2,id4,id6\},\{id6\}),$ $(\{id2,id5,id9\},\{id9\}),(\{id2,id5,id8\},\{id8\}),(\{id3,id4,id2,idd,idk\},\{id7\}),(\{idk\},\{id5,$ $id10\})\ \}$. Note that for readability, we omitted the elements RPG, $WorldMap$, $Scene$, and also did not assign any identifiers to the PetriNet $InArcs$ and $OutArcs$. After careful analysis, we concluded that the main problem with the above transformation specification was that the Hero could pass through the Door even without having its respective Key.

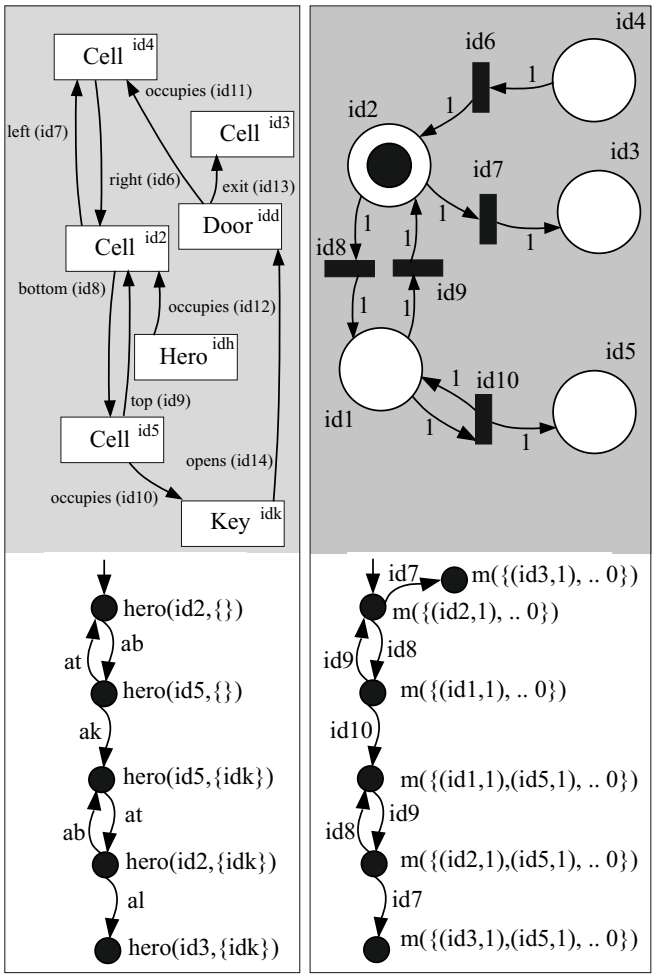

Figure 6.3: An example of a collapsed transformation rule from the transformation specification presented in Listings 6.2 and 6.3 (on top), and their respective transition systems (on bottom). In this example we used the term $m(\{(id1,1),..0\})$ as an abbreviation of $marking(p(id1,suc(zero)),marking(p(id2,zero),..,e)..)$—i.e., all the other places are empty.

The corrected version is then presented in Listing 6.4, which for the sake of brevity we only present the affected rules with the new lines denoted with the plus sign. In other words, the Place associated with the Key is now being connected with the Transition associated with the movement (top, bottom, right or left) by means of an *InArc* and an

OutArc of $weight = 1$. This will protect the Transition associated with the *Door*, to be fired if there is no *Token* in the *Place* associated with the *Door*'s *Key*.

Listing 6.4: The corrected version of the DSLTrans transformation from RPG Language to PetriNets Language. The rules 'leftDoor'

```
    rule'leftDoor'
2       match with
    cl63:
            any  rpg::Cell
    cl64:
            any  rpg::Cell
7   cl65:
            any  rpg::Door
    cl66:
            any  rpg::Cell
    cl67:
12          any  rpg::Key
    cl68:
            any  rpg::RPG
            subject to
                cl63 --(left)-> cl64
17              cl65 --(occupies)-> cl64
                cl65 --(exit)-> cl66
                cl67 --(opens)-> cl65
        apply
    cl69:
22          petrinet::Place
    cl70:
            petrinet::Place
    cl71:
            petrinet::Transition(
27                  name= 'moveLeft'
                )
    cl72:
            petrinet::InArc(
                    weight= '1'
32              )
    cl73:
            petrinet::OutArc(
                    weight= '1'
                )
```

```
    cl74:
            petrinet::Place
4   +   cl75:
    +           petrinet::OutArc(
    +                   weight= '1'
    +               )
    +   cl76:
9   +           petrinet::InArc(
    +                   weight= '1'
    +               )
    cl77:
            petrinet::PetriNet
14          subject to
                cl73 --(sourcePlace)-> cl69
                cl72 --(targetPlace)-> cl70
                cl71 --(inArc)-> cl72
                cl71 --(outArc)-> cl73
19  +           cl76 --(targetPlace)-> cl74
    +           cl75 --(sourcePlace)-> cl74
    +           cl71 --(inArc)-> cl76
    +           cl71 --(outArc)-> cl75
                cl77 --(transitions)-> cl71
24      restrictions
            cl69 derived from cl63
            cl70 derived from cl66
            cl74 derived from cl67
            cl77 derived from cl68
29  end rule

\\ skipping rightDoor, topDoor and bottomDoor:
\\     same as leftDoor

34 end def
```

Finally, the Figure 6.4 shows the same collapsed transformation rule, and its associated transition systems. It is now easy to observe that their transition systems (in the bottom) are in fact bisimilar. Although notice that in both versions of the translation, there seems to be a strange *Place* (id4) which is always empty (i.e., without *Tokens*), but our analysis was unable to detect this problem. Moreover, computationally speaking, this *Place* (id4) adds no additional computational behaviour to the presented Petri Net, and therefore it can be removed without affecting the analysis result.

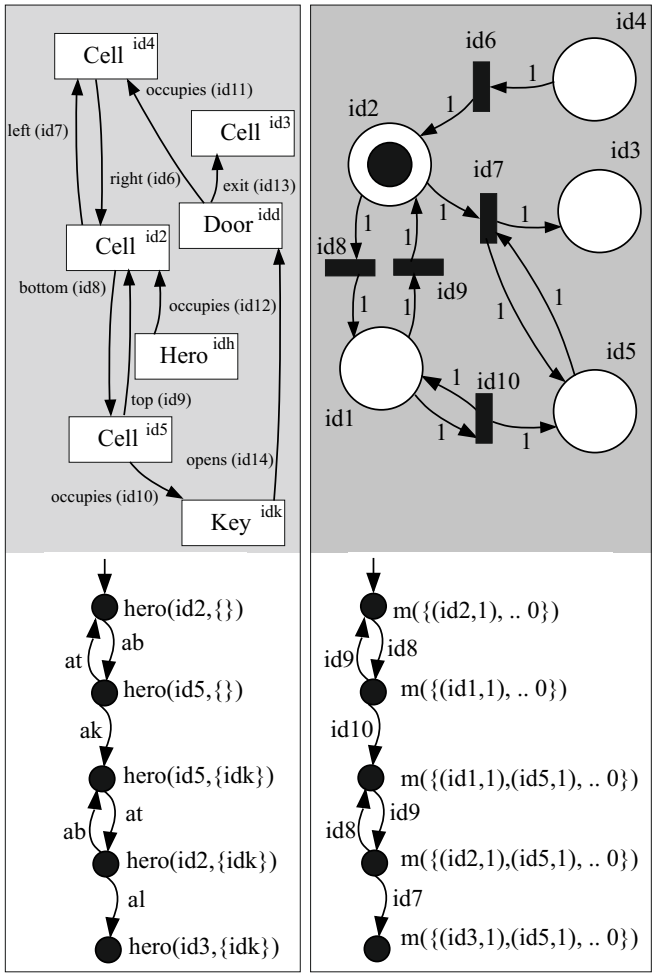

Figure 6.4: An example of a collapsed transformation rule from the transformation specification presented in Listing 6.2 (on top), and their respective transition systems (on bottom). In this example we used the term $m(\{(id1, 1), ..0\})$ as an abbreviation of $marking(p(id1, suc(zero)), marking(p(id2, zero), .., e)..)$—i.e., all the other places are empty.

6.3 Discussion of the Results

We applied the described methodology in the engineering of a more realistic DSML called RPG (Role Playing Games). This DSML was specifically designed to enable game designers to specify their RPGs, analyse their correctness by means of powerful analysis algorithms and data structures (in a Petri Net model checker called AlPina [4]), and finally automatically deploy them in a given computational platform. In this case study, both DSLTrans and SOS were expressive enough to specify both the denotational and operational semantics of the RPG DSML. Also, with the later specification, we were able to effectively validate the defined translation. However, given the size of this case study,

[4] http://alpina.unige.ch/

we observed that the checking procedure is not computationally efficient: the verification technique has to be optimized in order to be applicable in the practice of SLE, namely it has to be faster (i.e., in the scale of) hours instead of days) and also take less amount of memory (i.e., less than 4 Gigabyte).

7

Conclusions

In this PhD research work, we explored the application of the MDD approach as a solution to build DSML compilers with guarantees of correctness. We developed a language engineering methodology and its supporting languages that enables language engineers to specify and analyse language translations w.r.t. *(i)* a given property, or *(ii)* both the source and target language's semantics specifications. Moreover, with these specifications, the language engineer can automatically translate any expressible model in the source language into its respective in the target language.

In other words, in this research work, almost every designed languages and tools were designed and implemented following an MDD approach, which was shown to be an effective way to validate the soundness of the resulting work. Therefore, the MDD approach of lowering the gap between software models and their implementations during the software engineering process was also validated during this research work. In particular, with the tight connection between the formalization of the proposed conceptual framework (for the automatic validation of language translations), and its implementation.

7.1 Limitations and Future Work

One important limitation while using this approach is that it will only work properly, in the practice of software language engineering, if we are able to check semantic equivalence relations between the transition systems of each model on each pair provided by the analysis of the given translation. It is intuitive that, depending on the kind of semantic equivalence that we are trying to prove, if the source language enables sentences can

have infinite sized transition systems, we can no longer use this method to assert the correctness of that translation — note that the source language of RPGs (or StateMachines) — as some other DSMLs — did had a finite transition system. Notice that we do not require that the transition systems of all sentences in the target language of a translation to be finite — this is due the fact that our translations are not (by definition) bi-directional, and our assumptions rely only on that.

Besides that, having the fact that DSMLs' semantics are usually realized by means of code generators without any use of operational semantics, it is questionable the use of this technique in practice. However, this technique could already be applied if the software language engineer builds up an appropriate intermediate DSML based on component models, and then generate code from it — i.e by making the generated code conforming with a component language, and by using its associated SOS semantics.

Further enhancements of this technique will involve the definition of general evaluation rules for the evaluation of SOS rules. With these general rules we could be able to predict the sizes of the resulting transition systems. Therefore, we could be able to automatically decide the adequacy of this technique, and instruct/guide the Software Language Engineer to design its DSML with expressiveness concerns while remaining inside the borders of analysability.

In what respects to the analysis tool itself, future work will rely on finding efficient ways of both generating the symbolic execution space of translations on the fly and check them using the involved SOS semantics. In order to do so, it might involve the use of existing state-of-the-art model checkers and constraint solvers, specialized on performing such kind of computations, namely combinatorial search. For instance, there exists symbolic techniques, such as the ones explored in the model checking tool AlPiNa [HML+12], that could be adapted in order to reason in symbolic state spaces constituted by compact/comprehensive representations of combination sets, instead of having an extensive set of individual combinations as we just presented in this thesis.

7.2 Final Remarks and Expected Impact

The described technique is able to take advantage of any kind of MDD implementations based on DSLTrans' translations, namely in order to provide more guarantees about their correctness. As an example, in the implemented analysis tool, a complete formalisation of the operational semantics of MProlog (using the SOS language) could be provided in order to certify the DSLTrans translation between SOS and MProlog. Therefore, we believe that further improvements of this technique and its application in the current practice of software engineering, will greatly increase the quality of MDD based implementations based on translations or other similar analysable specifications.

The presented work has been capturing the interest of the research community from

the emerging research field of software language engineering, while presenting it in related forums such as the DSM-TP [1] Summer School. We envision, in the near future, the development of a robust and sound software language engineering supported by meta-modeling tools and language workbenches that can give to the Software Language Engineer the capacity to easily prototype new software languages, and have some level of correctness guarantees about not only the developed language (e.g., the language as a product, where we evaluate its usability, its cognitive adequacy to the experts of the domain, etc.), but also all of its products (e.g., the language as a product line, where we evaluate each product's implementation in a target platform, etc.).

From this, we believe that the results from this PhD research work constitutes an important advance in the research field of Software Language Engineering. On the one hand, this work can be helpful to further devise design guidelines of DSMLs where their analysis is an important issue. On the other hand, the explored nature of the best models of computation (e.g., with termination and confluence guarantees, cognitive aspects, usability, etc.) that should be used in the context of software language engineering (and software engineering in general sense), is still a challenge to be explored in the future research work.

[1] www.dsm-tp.org

Bibliography

[ABC12] Vasco Amaral, Bruno Barroca, and Paulo Carreira. Towards a robust solution in Building Automation Systems: supporting rapid prototyping and analysis. In *8th International Conference on the Quality of Information and Communications Technology*. IEEE, 9 2012.

[ABK07] Kyriakos Anastasakis, Behzad Bordbar, and Jochen Küster. Analysis of Model Transformations via Alloy. In B. Baudry, A. Faivre, S. Ghosh, and A. Pretschner, editors, *Proceedings of the workshop on Model-Driven Engineering, Verification and Validation (MoDeVVA 2007), Nashville, TN (USA)*, pages 47–56, Berlin/Heidelberg, October 2007. Springer.

[Abr96] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[ADL+12] Moussa Amrani, Juergen Dingel, Leen Lambers, Levi Lùcio, Gehan Selim, Eugene Syriani, and Manuel Wimmer. Towards a Model Transformation Intent Catalog. In *Proceedings of the 1st Workshop on the Analysis of Model Transformations*. ACM, 2012.

[AKRS06] Carsten Amelunxen, Alexander Königs, Tobias Rötschke, and Andy Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In *ECMDA-FA*, pages 361–375, 2006.

[AKS03] Aditya Agrawal, Gabor Karsai, and Feng Shi. Graph Transformations on Domain-Specific Models. Technical report, Institute for Software Integrated Systems, Vanderbilt University, 2003.

[ALL10] Mark Asztalos, Laszlo Lengyel, and Tihamer Levendovszky. Towards Automated, Formal Verification of Model Transformations. In *ICST 2010: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, pages 15–24. IEEE Computer Society, 2010.

[ASMZS11] Pathiah Abdul Samat, Abdullah Mohd Zin, and Zarina Shukur. Analysis of the model checkers' input languages for modeling traffic light systems. *Journal of Computer Science*, pages 225–233, 2011.

[AvdBE12] Suzana Andova, Mark G. J. van den Brand, and Luc Engelen. Reusable and Correct Endogenous Model Transformations. In *ICMT*, pages 72–88, 2012.

[BA11] Bruno Barroca and Vasco Amaral. Asserting the Correctness of Translations. In *Proceedings of the 6th Workshop on Multi-paradigm Modeling - MODELS 2011*. EASST, 10 2011.

[BAGB11a] Ankica Barisic, Vasco Amaral, M. Goulao, and Bruno Barroca. How to reach a usable DSL? moving toward a Systematic Evaluation. In *Proceedings of the 6th Workshop on Multi-paradigm Modeling - MODELS 2011*. EASST, 10 2011.

[BAGB11b] Ankica Barisic, Vasco Amaral, M. Goulao, and Bruno Barroca. Quality in Use of Domain Specific Languages: a Case Study. In *Proceedings of the PLATEAU 2011 Workshop on Evaluation and Usability of Programming Languages and Tools - SPLASH 2011*. ACM, 10 2011. URL=http://http://dx.doi.org/10.1145/2089155.2089170.

[BAGB11c] Ankica Barisic, Vasco Amaral, M. Goulão, and Bruno Barroca. Quality in Use of Domain Specific Languages: a Case Study. In *Proceedings of the PLATEAU 2011 Workshop on Evaluation and Usability of Programming Languages and Tools - SPLASH 2011*. ACM, 10 2011.

[BAGB12] Ankica Barisic, Vasco Amaral, M. Goulão, and Bruno Barroca. *Evaluating the Usability of Domain-Specific Languages*. IGI Global, 09 2012.

[BDtmM+06] Benoit Baudry, Trung Dinh-trong, Jean marie Mottu, Devon Simmonds, Robert France, Sudipto Ghosh, Franck Fleurey, and Yves Le Traon. Model Transformation Testing Challenges. In *In Proceedings of IMDT workshop in conjunction with ECMDA'06*, 2006.

[BET08] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Precise Semantics of EMF Model Transformations by Graph Transformation. In *MODELS'08: Proceedings of ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems*, pages 53–67, Berlin, Germany, 2008. Springer.

[BG11] Jan Olaf Blech and Benjamin Grégoire. Certifying compilers using higher-order theorem provers as certificate checkers. *Form. Methods Syst. Des.*, 38(1):33–61, February 2011.

[BGL05] Jan Olaf Blech, Sabine Glesner, and Johannes Leitner. Formal Verification of Java Code Generation from UML Models. In *Formal Verification of Java Code Generation from UML Models*. Fujaba Days, september 2005.

[BLA+10] Bruno Barroca, Levi Lucio, Vasco Amaral, Vasco Sousa, and Roberto Felix. DSLTrans: A Turing Incomplete Transformation Language. In *Proc. 3rd International Conference on Software Languages Engineering - SLE 2010*. Springer-Verlag, 2010.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[CHE04] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration using feature models. In *Software Product Lines: Third International Conference, SPLC 2004*, pages 266–283. Springer-Verlag, 2004.

[Chl10] Adam Chlipala. A verified compiler for an impure functional language. In *POPL*, pages 93–106, 2010.

[Cho56] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956. http://www.chomsky.info/articles/195609--.pdf – last visited 14^{th} January 2009.

[CS63] Noam Chomsky and Marcel Paul Schützenberger. The Algebraic Theory of Context-Free Languages. In P. Braffort and D. Hirshberg, editors, *Computer Programming and Formal Systems*, Studies in Logic, pages 118–161. North-Holland Publishing, Amsterdam, 1963.

[Dat04] Chris J. Date. *An Introduction to Database Systems*. Pearson Addison-Wesley, Boston, MA, 8. edition, 2004.

[DFF+09] Zoé Drey, Cyril Faucher, Franck Fleurey, Vincent Mahé, and Didier Vojtisek. *Kermeta language - Reference Manual*. Institut de Recherche en Informatique et Systèmes Aléatoires, France, April 2009.

[dLVA04] Juan de Lara, Hans Vangheluwe, and Manuel Alfonseca. Meta-modelling and graph grammars for multi-paradigm modelling in AToM3. *Software and Systems Modeling*, 3:194–209, 2004. 10.1007/s10270-003-0047-5.

[EEHT05] Karsten Ehrig, Claudia Ermel, Stefan Hänsgen, and Gabriele Taentzer. Towards Graph Transformation Based Generation of Visual Editors Using Eclipse. *Electron. Notes Theor. Comput. Sci.*, 127(4):127–143, April 2005.

[Fav04] Jean-Marie Favre. Foundations of Model (Driven) (Reverse) Engineering : Models - Episode I: Stories of The Fidus Papyrus and of The Solarus. In *Language Engineering for Model-Driven Software Development*, 2004.

[Fer09] Maribel Fernndez. *Models of Computation: An Introduction to Computability Theory*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[FHLN08] Jean-Rémy Falleri, Marianne Huchard, Mathieu Lafourcade, and Clémentine Nebut. Metamodel Matching for Automatic Model Transformation Generation. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Volter, editors, *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*, volume 5301 of *Lecture Notes in Computer Science*, pages 326–340. Springer, 2008.

[Fow05] Martin Fowler. Language workbenches: The Killer-App for Domain Specific Languages?, 2005.

[FR07] Robert France and Bernhard Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society.

[Gor10] Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. *Inf. Comput.*, 208(9):1031–1053, September 2010.

[GS98] Wolfgang Goerigk and Friedemann Simon. Towards Rigorous Compiler Implementation Verification. In *Proc. of the 1997 Workshop on Programming Languages and Fundamentals of Programming*, pages 62–73. Springer, 1998.

[HJK+09] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and Refinement of Textual Syntax for Models. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, ECMDA-FA '09, pages 114–129, Berlin, Heidelberg, 2009. Springer-Verlag.

[HML+12] Steve Hostettler, Alexis Marechal, Alban Linard, Matteo Risoldi, and Didier Buchs. High-Level Petri Net Model Checking with AlPiNA. *Fundamenta Informaticae*, 113(3-4):229–264, 2012.

[HVV08] Zef Hemel, Ruben Verhaaf, and Eelco Visser. WebWorkFlow: An Object-Oriented Workflow Modeling Language for Web Applications. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and

Markus Völter, editors, *11th International Conference on Model Driven Engineering Languages and Systems (MODELS 2008)*, volume LNCS 5301, pages 113–127, Toulouse, France, 2008. Springer.

[IBM07] IBM. IBM Model Transformation Framework, 2007. http://www.alphaworks.ibm.com/tech/mtf.

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[JGP99] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[JIMK03] T. Jokela, N. Iivari, J. Matero, and M. Karukka. The standard of user-centered design and the standard definition of usability: analyzing ISO 13407 against ISO 9241-11. In *Proceedings of the Latin American conference on Human-computer interaction*, pages 53–60. ACM, 2003.

[JK05] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, Montego Bay, Jamaica, 2005.

[Jr.75] Frederick P. Brooks Jr. *The mythical man-month - Essays on Software-Engineering*. Addison Wesley, 1975.

[Jr.87] Frederick P. Brooks Jr. No Silver Bullet - Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, 1987.

[Kle09] A.G. Kleppe. *Software language engineering: creating domain-specific languages using metamodels*. Addison-Wesley, 2009.

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling*. Wiley-IEEE Computer Society Press, March 2008.

[Küh04] Thomas Kühne. What is a Model? In *Language Engineering for Model-Driven Software Development*, 2004.

[Küh06] Thomas Kühne. Matters of (Meta-)Modeling. *Software and System Modeling*, 5(4):369–385, 2006.

[Kus04] Jochen M. Kuster. Systematic Validation of Model Transformations. In *Essentials of the 3rd UML Workshop in Software Model Engineering (WiSME 2004)*, 2004.

[LBA10] Levi Lucio, Bruno Barroca, and Vasco Amaral. A Technique for Automatic Validation of Model Transformations. In *ACM/IEEE MoDELS 2010*. Springer-Verlag, 10 2010. URL=http://models2010.ifi.uio.no/.

[MB97] Savi Maharaj and Juan Bicarregui. On the Verification of VDM Specification and Refinement with PVS. In *Proof in VDM: Case Studies, FACIT (Formal Approaches to Computing and Information Technology) , chapter 6*, pages 157–190. Springer-Verlag, 1997.

[MBB+12] Eduardo Marques, Valter Balegas, Bruno Barroca, Vasco Amaral, and Ankica Barisic. The RPG DSL: a case study of language engineering using MDD for Generating RPG Games for Mobile Phones. In *Proceedings of the 12th Workshop on Domain-Specific Modeling at OOPSLA/SPLASH*. ACM Digital Library, 10 2012.

[MFBC10] Pierre-Alain Muller, Frédéric Fondement, Benoit Baudry, and Benoit Combemale. Modeling Modeling Modeling. *Journal of Software and Systems Modeling (SoSyM)*, 2010.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[Mil93] Robin Milner. Elements of Interaction - Turing Award Lecture. *Commun. ACM*, 36(1):78–89, 1993.

[MP10] Janne Merilinna and Juha Pärssinen. Verification and validation in the context of domain-specific modelling. In *Proceedings of the 10th Workshop on Domain-Specific Modeling*, DSM '10, pages 9:1–9:6, New York, NY, USA, 2010. ACM.

[MPP08] Janne Merilinna, OlliPekka Puolitaival, and Juha Pärssinen. Towards Model-Based Testing of Domain-Specific Modelling Languages. In *Proceedings of the 8th Workshop on Domain-Specific Modeling*, DSM '08, pages 19–20, 2008.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[NK08] Anantha Narayanan and Gabor Karsai. Verifying Model Transformations by Structural Correspondence. *ECEASST*, 10, 2008.

[Par81] David Park. Concurrency and Automata on Infinite Sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pages 167–183, London, UK, 1981. Springer-Verlag.

[Par08] Joachim Parrow. Expressiveness of Process Algebras. *Electr. Notes Theor. Comput. Sci.*, 209:173–186, 2008.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[PJM04] Hans Vangheluwe Pieter J. Mosterman. *Computer Automated Multi-Paradigm Modeling: An Introduction. Simulation: Transactions of the Society for Modeling and Simulation International*. Society for Modeling and Simulation International, 2004.

[Plo04] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[Plu98] D. Plumpf. Termination of graph rewriting is undecidable. *Fundam. Inf.*, 33(2):201–209, 1998.

[PP08] Michael Pfeiffer and Josef Pichler. A Comparison of Tool Support for Textual Domain-Specific Languages. In *Proceedings of the 8th OOPSLA Workshop on Domain-Specific Modeling*, pages 1–7, October 2008.

[PSS98] A. Pnueli, M. Siegel, and F. Singerman. Translation Validation. pages 151–166. Springer, 1998.

[RAB+09] Matteo Risoldi, Vasco Amaral, Bruno Barroca, Kaveh Bazargan, Didier Buchs, Fabian Cretton, Gilles Falquet, Anne Le Calvé, Stéphane Malandain, and Pierrick Zoss. A Language and a Methodology for Prototyping User Interfaces for Control Systems. In *Human Machine Interaction*, pages 221–248. Springer-Verlag, 2009.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[Sel03] B. Selic. The pragmatics of model-driven development. *Software, IEEE*, 20(5):19–25, 2003.

[SG12] Eugene Syriani and Jeff Gray. Challenges for Addressing Quality Factors in Model Transformation. In *ICST*, pages 929–937, 2012.

[TCJ10] Massimo Tisi, Jordi Cabot, and Frédéric Jouault. Improving higher-order transformations support in ATL. In *Proceedings of the Third international conference on Theory and practice of model transformations*, ICMT'10, pages 215–229, Berlin, Heidelberg, 2010. Springer-Verlag.

[UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[VABKP11] Marcel Van Amstel, Steven Bosems, Ivan Kurtev, and Luís Ferreira Pires. Performance in model transformations: experiments with ATL and QVT.

In *Proceedings of the 4th international conference on Theory and practice of model transformations*, ICMT'11, pages 198–212, Berlin, Heidelberg, 2011. Springer-Verlag.

[Van00] Hans Vangheluwe. *Multi-formalism modelling and simulation*. PhD thesis, Ghent University, 2000.

[vBV09] Steffen van Bakel and Maria Grazia Vigliotti. A logical interpretation of the Lambda-Calculus into the Pi-Calculus, Preserving Spine Reduction and Types. In *CONCUR 2009 - Concurrency Theory - Lecture Notes in Computer Science*, volume 5710, pages 84–98. Springer Berlin / Heidelberg, 2009.

[VP03] Dániel Varró and András Pataricza. Automated Formal Verification of Model Transformations. In Jan Jürjens, Bernhard Rumpe, Robert France, and Eduardo B. Fernandez, editors, *CSDUML 2003: Critical Systems Development in UML; Proceedings of the UML'03 Workshop*, number TUM-I0323 in Technical Report, page 63–78. Technische Universität München, Technische Universität München, September 2003.

[VT11] Naveneetha Vasudevan and Laurence Tratt. Comparative Study of DSL Tools. *Electron. Notes Theor. Comput. Sci.*, 264(5):103–121, July 2011.