UNIVERSITÉ DE GENÈVE

Archive ouverte UNIGE
https://archive-ouverte.unige.ch

Thèse 2009

Open Access

This version of the publication is provided by the author(s) and made available in accordance with the copyright holder(s).

A systematic language engineering approach for prototyping domain specific modelling languages

Venceslau Pedro, Luis Miguel

How to cite

VENCESLAU PEDRO, Luis Miguel. A systematic language engineering approach for prototyping domain specific modelling languages. 2009. doi: 10.13097/archive-ouverte/unige:11946

This publication URL: https://archive-ouverte.unige.ch//unige:11946
Publication DOI: 10.13097/archive-ouverte/unige:11946

© This document is protected by copyright. Please refer to copyright holder(s) for terms of use.

UNIVERSITÉ DE GENÈVE FACULTÉ DES SCIENCES

Centre Universitaire d'Informatique Professeur D. Buchs

A Systematic Language Engineering Approach for Prototyping Domain Specific Modelling Languages

THÈSE

présentée à la Faculté des sciences de l'Université de Genève
pour obtenir le grade de Docteur ès sciences, mention informatique

par

Luis Miguel VENCESLAU PEDRO
du
Portugal

Thèse No 4068

Genève
Atelier d'impression ReproMail
2009

UNIVERSITÉ DE GENÈVE

FACULTÉ DES SCIENCES

Doctorat ès sciences
mention informatique

Thèse de *Monsieur Luis Miguel VENCESLAU PEDRO*

intitulée :

"A Systematic Language Engineering Approach for Prototyping Domain Specific Modelling Languages"

La Faculté des sciences, sur le préavis de Messieurs D. BUCHS, professeur adjoint et directeur de thèse (Département d'informatique), Ph. DUGERDIL, docteur (Haute Ecole de Gestion de Genève – Laboratoire de génie logiciel), J. ARAUJO, professeur (Université de Lisbonne – Faculté des sciences et technologies – Département d'informatique – Caparica, Portugal), N. GUELFI, professeur (Université du Luxembourg – Faculté des sciences technologies et communication – Laboratory of Advanded Software Systems - Luxembourg), et S. SENDALL, docteur (Snowie Group S.A. – Morges, Suisse), autorise l'impression de la présente thèse, sans exprimer d'opinion sur les propositions qui y sont énoncées.

Genève, le 26 janvier 2009

Thèse - 4068 -

Le Doyen, Jean-Marc TRISCONE

N.B.- La thèse doit porter la déclaration précédente et remplir les conditions énumérées dans les "Informations relatives aux thèses de doctorat à l'Université de Genève".

Nombre d'exemplaires à livrer par colis séparé à la Faculté : - 4 -

Acknowledgements

This dissertation would not have been possible without the support of very important people.

I would like to thank to Prof. Didier Buchs for the opportunity to work with him and his team at the University of Geneva. He provided the necessary environment and resources that allowed me to develop the present work. I express my recognition for his insights on the most formal aspects and their importance in the context of this work.

Second, I would like to thank to professors João Araújo, Nicolas Guelfi, Philipe Dugerdil and Shane Sendall that accepted to be members of this thesis jury. It is my pleasure to have such a distinct group of persons and professionals to evaluate my work.

I'd also like to thanks to all members of the Software Modelling and Verification Group of the University of Geneva. For all their comments and hints in several stages of this work. It was a pleasure to work with all of them. A special thank you to Alexis Marechal, Steve Hostetler and Vasco Amaral that where particularly helpful in the last phase of this work.

To Adrien Chantre, Gaelle Ribordy, Helio de Sousa, Judith Dirk, Kerstin Brinkmann, Nadine Oberli, Micaela Santos, Tobias Brosch and Yanick Vuille. All of them, as well as other friends, created an environment that made me feel home.

I am also thankful to Sergio Coelho for all his motivation and help with my integration in Geneva. Above all for his generous friendship.

I want to thank to Matteo Risoldi that have been an 'out of the box' friend and colleague, and that always created the best possible environment. I am very fortunate for all his help.

My great acknowledgment to my long time friends that have always been supportive of my decisions. To Marta Carvalho that has constantly a 'hi' that makes me laugh; to Dinis Klose that has encouraged me without any questions, that helps me anytime he can and is most of the time right; to Nuno Gaspar for all his good mood and constant search for a nice place for having fun; to Nuno Barros that usually helps in the most incredible things

and that is able to do them in a way that amazes me; to Ruben Coelho my deep appreciation for all his serenity, affection and phone calls. All of them helped me to grow up as a person and professional.

I am grateful to my family that helped me to have the opportunities for continuing my studies. A very special thank you to Filipa Burgo, Iolanda Burgo and Egídio Ramos for their interest, encouragement and help whenever needed.

Finally, I am extremely grateful to Joana who was always present. For her support, her patience, her understanding, her encouragement. For always being capable of providing me with the energy to look forward. For all her original comments and sense of humor. For all the small things that only she is capable of.

For Joana and all her support.

Sommaire

Language Driven Engineering (LDE) est un sujet d'application et de recherche relativement récent. Ce qui est encore plus récent est l'intèrêt grossissant porté par la communauté s'intéressant au développement des Domain Specific Modelling Language (DSML) dans le domaine de l'ajout de sémantique aux DSMLs. Pour certains, la sémantique devraient être fournies dans un dialecte spécifique au domaine, alors que pour d'autres, la sémantique devraient être données en utilisant une méthodologie générique et un framework d'enrichissement sémantique. Sans prendre en considération les différences entre ces conjectures concernant le "comment", il subsiste malgré tout un consensus sur le fait que la sémantique est, de nos jours, un sujet de recherche très important dans la communauté des DSMLs. Une preuve de cela est la récente création d'un groupe de travail examinant les aspects de sémantique relatives aux DSMLs dans le dernier workshop OOPSLA DSM'08.

La sémantique des langages et la validation de son comportement sont deux aspects très interconnectés. Il est donc naturel d'espérer qu'une fois que nous serons capable de définir la sémantique des langages nous souhaiterons les valider. Cette opération est évidemment d'une énorme importance puisque la valeur d'un langage est très limitée si ces programmes/modèles ne se comportent pas de la faon souhaitée.

Comme dans la plupart des domaines techniques les processus de validation sont souvent obtenus en utilisant des techniques de prototypage. Bien que considéré comme un moyen de validation de concept ou/et d'implémentation très puissant, les techniques de prototypage sont généralement négligées: leur complexité et leurs difficultés à mettre en oeuvre dans la pratique et dans des environnements de production empêchent leur usage systématique.

Dans ce document nous proposons un framework conceptuel et une possible implémentation d'un environnement permettant le prototypage de DSMLs. Nous nous connecterons sur les aspects de diversité et d'évolution des langages. Il est à la fois prévu qu'un DSML évolue dans le temps et que les ingénieurs de langage soient capable de réutiliser certains concepts déjà implémentés. Le travail ayant donné origine à ce document se concentre

sur la manière de résoudre les problèmes d'évolution et de réutilisation des langages.

Afin de mieux comprendre les problématiques impliquées, nous allons commencer par présenter, de la faon la plus pédagogique possible, une étude exhaustive sur les méthodologie et les environnements de développement des DSMLs. Nous allons comparer les divers outils et techniques disponibles et nous mettrons en évidences les contributions se concentrant sur le processus d'enrichissement sémantique des DSMLs ainsi que sur son application. Cette thèse se poursuivra sur la proposition d'un framework supportant la composition des aspects syntaxiques et sémantiques des DSMLs. Une vue formelle des aspects méthodologiques sera présentée ainsi qu'une implémentation d'un DSML. L'objectif principal de ce document est de présenter les développements couvrant tout les aspects de l'apport sémantique aux DSMLs afin de soutenir le prototypage rapide et un niveau élevé de réutilisation.

En résumé, cette thèse se base sur une approche formelle en ce qui concerne la description de la méthodologie et fournit trois cas d'études de spécification de DSML distincts. En utilisant plusieurs "standards" bien connus de la communauté, ce document propose une méthode cohérente et intégrée de prototypage et de développement des DSMLs.

Abstract

Language Driven Engineering (LDE) is a fairly recent subject of application and research. Even more recent is the growing interest of providing operational semantics to DSMLs. For some, semantics should be provided in a domain-specific dialect, whereas for others, semantics should be provided using a generic methodology and framework for semantics' enrichment. Regardless of the difference between conjectures concerning the 'how', it is consensual that semantics is becoming a very important research topic in the DSML community. Recent evidence is the working group created for discussing semantic-related aspects of DSMLs in the latest OOPSLA DSM'08 workshop.

Language semantics and validation of its behaviour are two aspects that are very interconnected. It is to be expected that once we are able to define a language semantics, we will want to validate it. This operation is very important since language value is very limited if its programs/models do not behave as expected.

As in most of the engineering areas, the validation process is often achieved by using prototyping techniques. Although considered to be a very powerful way of validating a concept and/or an implementation, prototyping techniques are generally neglected due to their complexity and difficulty to use in practical and production environments, thus preventing systematic usage.

In this document, we propose a conceptual framework and a possible implementation for an environment that supports DSML prototyping. We focus on the diversity and evolution aspects of the language. It is expected that a DSML evolves over time but it is also expected that language engineers could re-use several of the concepts implemented. This thesis focused on tackling problems of language evolution and re-usability.

To better understand the problematics involved, we first started providing an exhaustive survey over the methodologies and the DSMLs development environments trying to be as pedagogical as possible. We compared several techniques and tools available. We also enhanced the contributions that focus on the process for DSML semantics enrichment and its application.

The dissertation continues with a proposal for a framework that supports syntactical and semantical composition of domain specific concepts. A formal overview of the methodological aspects is presented. A description of a DSML for implementing these notions is also described. The main goal of this thesis is to present the developments that cover all facets when providing semantics to DSML in order to support rapid prototyping and a high level of re-usability.

To summarise, this thesis is based on a formal approach for the methodology description and provides a set of three distinct case studies for DSML specification. Using several well known standards in the Software Language Engineering (SLE) community, this thesis proposes a coherent and integrated method for DSML prototyping development.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

A language that is used will be changed.
- Meir M. Lehman

This chapter presents a brief introduction to the methodology and concepts used in the context of the work developed for this thesis.

The latest trends in the development of languages suggest that Domain Specific Modelling Languages (DSMLs) are the core in the new research subjects of Software Language Engineering (SLE) and Language Driven Engineering (LDE). The arguments most used in favour of the SLE approach are that:

1. DSMLs solve problems within clearly definable domains of application such that software engineers are able to create DSMLs that solve real problems more rapidly and with higher quality;

2. Developers apply DSMLs to improve productivity and quality in areas ranging from finance to control systems, macro scripting to graphical user interfaces. In order to do this, they use abstractions provided by the DSML that make available the artefacts expressed in the problem domain vocabulary.

This work is a SLE approach to the development of DSML prototyping.

1.1 Context

The Model Driven Architecture (MDA) initiative added a new perspective to the Software Development Process. It puts as fist class the models and the relations between them. The exposure of models at all levels of the development cycle is both a strength of the methodology and a weakness: a strength

because it forces to use different levels of abstraction, which allows one to clearly separate concerns and to split complexity; a weakness because using models in a massive way at different levels of abstraction and complexity induces complex problematics. An extensive use of models raises questions on how to preserve coherence and conformity in what concerns models representing different languages at different levels of reasoning and that evolve at different rhythms.

Using abstraction mechanisms does not make the system's complexity disappear. It's rather a technique of looking at the problem from the concrete perspective representing an actor's point of view. Applying abstraction mechanisms at several levels of the system development process benefits from hiding information and highlighting only the aspects that are more relevant for the perspective under analysis.

The MDA framework is based on defining Platform Independent Model (PIM), Platform Specific Model (PSM) and the transformations between them (Bezivin, 2005). The PIM is a model with a high level of abstraction independent from the technology and PSM a more concrete model. From PIM, several transformations can be defined to different PSM. This methodology defines a standard framework on how models defined in one language can be transformed into models of other languages and transformations are one of the key elements.

The relation between the different levels of abstraction is one of the concerns in the MDA framework and, due to definition of transformations, it is possible to create a coherent relation between them. Albeit, some of the biggest challenges of software engineering these days are still to master: complexity, diversity and evolution (Clark, Sammut, & Willans, 2008). The challenge of managing complexity is the main topic of many approaches and frameworks (Sommerville, 1995; Booch, 2004; Jacobson, 2004). In this work we will thus focus on the challenge of diversity and complexity.

The challenge of diversity reflects how developers and engineers have to manage in a non-homogeneous environment. Life would be much easier if there was only one programming language and one deployment platform. This is, however, not the case and for very good reasons. Diversity is not really a single challenge, but a category of challenges, namely:

- diverse domains;
- diverse customer requirements;
- diverse implementation technologies;
- diverse tools and artefact types.

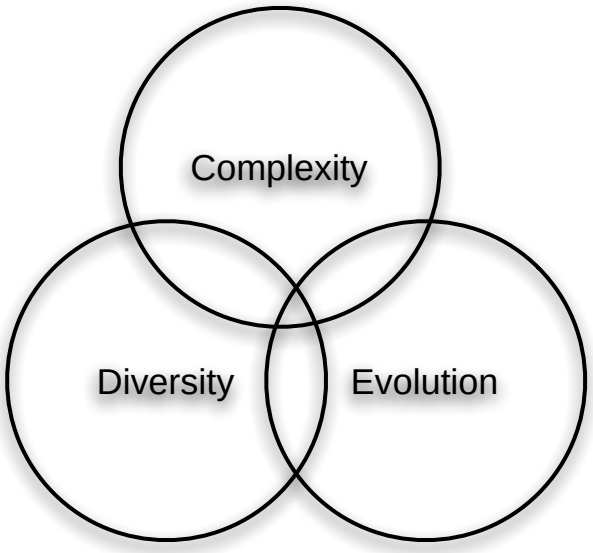


Figure 1.1. Domain Specific Modelling Language (DSML) Challenges

In order to manage complexity, one approache is to focus on domain problems and to try to provide a solution for each one of them. The DSMLs comes as a suitable approach and the Language Engineering Development is a way of providing such a solution.

All these challenges involved in the development of DSMLs or Domain Specific Modelling Language (DSML) development environments intersect at some point. The Fig. 1.1 provides an illustration of this overlapping: when a particular environment needs several solutions in that same domain, complexity transforms into a diversity problem; diversity implies at some point a high level of evolution; and evolution makes developer's life easier but implies to rise complexity in terms of DSML development.

1.2 Domain Specific Modelling Languages

The term DSML is used in software development to indicate a modelling (and sometimes programming) language dedicated to a particular problem domain, a particular problem representation technique and/or a particular solution technique. The concept is not new - special-purpose programming language and all kinds of modeling/specification languages have always existed, but the term DSML has become more popular due to the rise of domain-specific modelling. Domain-specific languages are considered $4GL$ programming languages. Along with the concept of DSMLs, the concept of

families of DSMLs is also often used. This term is mainly introduced because it is very common that there are only variations of the same system, evolutions of the same system, or even because different DSMLs simply share a big number of concerns.

More specific abstractions can be used in fewer products (i.e., by members of a smaller product family), but contribute more to their development. More general abstractions can be used in more products (i.e., by members of a larger product family), but contribute less to their development. Higher levels of specificity allow more systematic re-use (Greenfield, Short, Cook, & Kent, 2004, p. 24) .

Developers increasingly turn to DSMLs to manage diversity and complexity which encapsulates a context (usually refereed to domain) providing (Jackson & Sztipanovits, 2006):

- A set of components or language terms and constructs with which embedded hardware / software can be modelled;
- A set of constraints that enforce proper use of components;
- A set of semantic mappings that generate simulation traces, embedded code, and verifications results from models.

Systems are often specified by capturing complex interrelationships between concepts. A good approach for developing DSMLs is being able to separate domain concepts, concerns or different language modules. At the end of its development cycle, a DSML will capture a set of concepts. By using an incremental development approach the complex product being developed is still manageable. Since language development tends not to be a one-cycle process, providing a modular development environment allows to cope with complexity by incrementally adding new features. Simultaneously, most of the DSML environment are not linked with a single technology and a good DSML development environment should be able to deal with diversity.

1.3 Motivation

DSMLs have been very successful in embedded systems. They have been adopted and used regularly in other domains but with little formal support.

From its very starting days, the purpose of software engineering was to make software development an engineering task. Through the establishment of a methodology, this work aims to be systematically applied to solve some problems in DSMLs development. Providing the process with precise formal

grounds, this work aims to offer a framework that allows DSMLs prototyping for validation of concepts. This makes the DSML development a more engineering process by reducing development time, increasing quality and re-using concepts.

The Language-Driven development is one of the topics under development that approaches diversity as well as complexity. Language has always been a fundamental way of communicating. Developing DSMLs is a step that allows to think at the problem domain correct abstraction and to communicate using an appropriate mechanism. The latest developments in the area suggests that concepts should be close to the problem domain and simultaneously re-usable.

In the subject on which this work is developed, there is usually a lack of reusability of concepts and a propensity to neglect the semantics of the language. It is common to stop at a very syntactic level while defining methodologies and approaches for DSMLs development.

While the idea of using DSMLs is not new, its support in terms of tools is rather limited. Development environments that include features helping the systematisation of tasks for the development of DSMLs only started to be developed within the last decade. Moreover, ideas of reusability and semantics anchoring are not yet very mature and most of the time scarce.

Most DSMLs are designed from scratch using engineers'[1] experience in designing languages. Our motivation while producing this work was to be able to provide a working framework that allows the ease of development of DSML prototypes without the need to create, validate and deploy a whole new development environment. In the approach proposed, we present a methodology that allows to:

- avoid effort duplication;
- cope with the urgency of high-quality reusable metamodel fragments (with a precise transformational semantics);
- significant reduction in the time for creating a new DSML;

The framework we have developed is based on metamodels (domain concepts) fragments that are composable and have associated structural semantics and semantic mappings. A *domain concept* in the context of this work is more than a metamodel: it is an abstract syntax defined in a standard fashion with a transformation to one or several precise target languages. Such languages provide well defined semantics in the context of the domain

[1] In the context of DSMLs development it is common to use the word metamodeler in order to identify the engineers that are part of the DSML development.

concept meaning. A *domain concept* can be seen as a building block that represents a basic idea that can be present in one or several DSMLs. It is an artefact used to express a concept and that can be composed with other domain concept(s) (or even with an already pre-defined DSML) in order to extend it.

This definition of domain concept is very expressive in the sense that rather than only defining an abstract syntax and associating a structural semantics, we work at the level of semantics mappings to PSMs. In the work presented in this thesis the semantic mappings are achieved by model transformation rules defined for each of the domain concepts.

As it will be presented in Chap. 3 most of the available tools for DSML development provide very little or no support for semantic specification. This also applies to re-usability and compositional aspects of a language development cycle. Both of these aspects, together with the notion that DSML prototyping should be a task with more formal and pragmatic support, motivated this work.

1.4 Contribution

The contribution resulting from the development of this work is mainly a proposal for a methodology and an implementation of a framework prototype that allows semantic enrichment of domain concepts. The main results of composing domain concepts as it will be explained in this document are to:

- be able to manage language complexity in terms of language design and maintainability;
- allow re-use of concepts for faster language development;
- cope with domain diversity by being able to define complete DSMLs;
- use a framework that allows fast development of DSMLs for validation of its behaviour without having to define a new development environment from scratch for each one of the DSMLs;
- be able to compose domain concepts both at the level of its metamodel and its semantics. The first by providing model and metamodel composition at the syntax level; the second by allowing the composition model transformation rules relative to its metamodel.

In the next chapters we will detail how to define domain concepts and, mainly, how to place them together. By composition and parameterization, a set of domain concepts are used to form a DSML or a family of DSMLs.

1.5 Document Organisation

This document is organised as follows:

- Chapter 2 provides a detailed overview of the work developed in the area from the methodological point of view. At the end of this chapter we provide a comparison between some of the existing techniques and the our work;

- Chapter 3 presents a set of DSML development environments and highlights each ones' strengths and weaknesses according to pre-defined criteria;

- Chapter 4 gives a formal definition of models and metamodels. While using a standard notation in the context of MDA, this chapter contributes with a set of definitions that allow models and metamodels to be expressed in a formal way;

- Chapter 5 gives an overview of the framework and the main concepts behind it from a more pragmatic point of view. The chapter gives a survey of the tools and frameworks used for the implementation of the methodology. In particular, it presents the Concurrent Object Oriented Petri Nets (CO-OPN) specification language and the ATLAS Transformation Language (ATL).

- Chapter 6 details the proposed methodology. We present how metamodels and transformations are composed. This chapter suggests a set of definitions that allow to parameterize metamodels, its associated transformations, and to provide derived models that are the result of applying parameterization at model level;

- Chapter 7 specifies the CoPsy framework. This chapter details the CoPsy DSML implementation, features, and provides an analysis of its usage;

- Chapter 8 reflects the application of the methodology defined in the document. It presents three case studies:

    1. Development of a DSML for representing the Moving Entities System. This DSML is particularised in two other DSMLs in the course of the chapter: one representing a DSML for a Railway System and the second one a DSML for controlling Robots in an hypothetical world;

2. An implementation of the COntrol systems SPEcification Language (Cospel). Several modules of Cospel were isolated and development of this DSML prototype is achieved using the CoPsy framework;
3. Development of a multi-flavour Petri-Nets environment. This example uses the CoPsy framework and methodology to incrementally provide the implementation of three Petri-Nets formalisms: Standard Petri Nets; Coloured Petri-Nets; and Algebraic Petri-Nets.

- Chapter 9 presents a discussion based on the results obtained and open issues.

Chapter 2

State of the Art: Methodologies

The concepts presented in this chapter will not be described in detail: most of them cover a vast domain of knowledge that represent a wide range of concepts that cannot be described exhaustively. We will concentrate, rather, on the aspects that we think are more useful for understanding our proposal and to highlight the relations between each other.

For clarity and organisational purposes, the state of the art is presented in two separate chapters. The first one covers the more methodological aspects and the second one presents an overview of some of the tools available for DSMLs development.

2.1 Unified Modeling Language

The Unified Modeling Language (UML) (Object Management Group, 2005; Booch, Rumbaugh, & Jacobson, 1999) is a modeling language for specifying, visualising, constructing and documenting the artefacts of software-intensive systems. The Object Management Group (OMG) has approved UML as a standard in 1997.

After several years of experience using UML, the OMG issued requests for proposals to upgrade UML extending it with additional capabilities that were desired in several application domains. Examples of new features included are: Object Constraint Language (OCL) language integration; extended metamodel to eliminate redundant constructs and to allow the reuse of well-defined subsets by other specifications; the availability of profiles to define domain and technology-specific extensions. The UML has evolved notably in the last six years. The more interesting modifications are at the level of the uniformisation its parts and at a more precise formalisation of its constructs. Most of these changes affect the internal representation of UML

constructs (the metamodel) and its relationship to other specifications:

- Unification of the core of UML with the conceptual modeling parts of Meta-Object Facility (MOF) (Object Management Group, 2002a) allowing UML models to be handled by generic MOF tools and repositories. In recent years the Essential MOF (EMOF) standard has been supported by a large number of tools and projects including the ones from the Eclipse foundation.

- Restructuring of the UML metamodel to eliminate redundant constructs and to permit reuse of well-defined subsets by other specifications.

- Availability of profiles to define domain and technology-specific extensions of UML.

UML provides a semantic base in the form of a metamodel that defines well-formed models, and possible relationships between models and model elements. The UML metamodel is finalised in its latest version and is used to provide an important consistency level so to ensure coherence among models. UML is only a language: it is process-independent and therefore does not prescribe how its notations should be used. Consequently, using UML to model any kind of specific systems still requires a method - a choice of models and a process of their elaboration.

2.2 Meta-Modeling

A metamodel is an artefact that can be seen as the definition of a language. It is also known as the abstract syntax of a language being the description of all the constructs that can be used in that language.

There are several general definitions of architectures for meta-data handling and, in particular, for metamodel manipulation. There are two main (Gerber & Raymond, 2003) metamodelling framework definitions: the OMG MOF (Object Management Group, 2002a, 2007a) and the open source Eclipse Meta-Modeling Framework (EMF) (Budinsky, Steinberg, Merks, Ellersick, & Grose, 2004). In general, these frameworks are organised in four different abstraction levels that are presented in 2.1:

i) *Data layer - M0* - usually containing source code derived from the *M1* level for different program languages, e.g. Java and C++;

ii) *Model layer - M1* describing the information in *M0* layer. It contains the models from which the source code in *M0* layer is generated, e.g. UML;

iii) *M2 - Meta model layer* that is a description of the models in layer *M1*. It is this meta model that defines the abstract syntax of a language and precises which artefacts can be present in the model, e.g. Metamodel of UML;

iv) *Meta metamodel layer - M3* is MOF/EMF themselves.

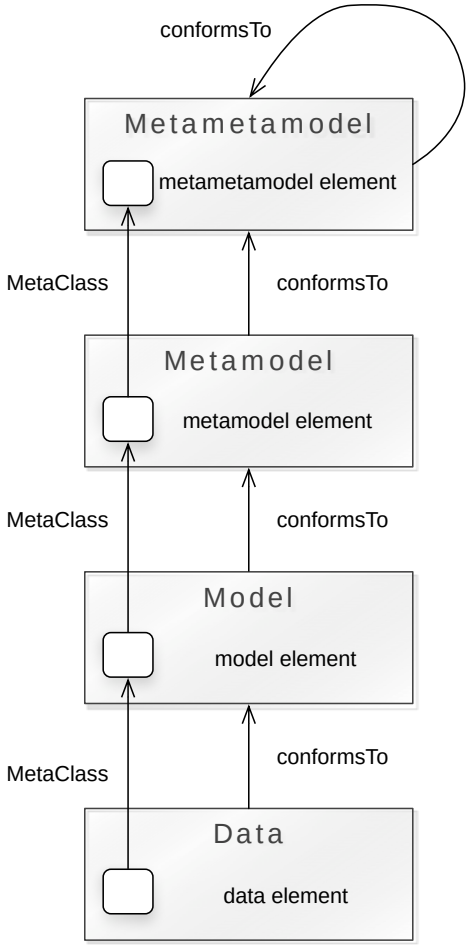


Figure 2.1. Metamodelling Framework 4-layers Architecture

The MOF framework (specified in (Object Management Group, 2002a, 2007a)) describes a standard modeling formalism that supports metamodel

abstractions. It does not define any standard graphic representation of its models but there exists a clear mapping between MOF and UML. Currently OMG defines two different versions of MOF: EMOF that stands for Essential MOF; and CMOF meaning Complete MOF.

The EMF (Budinsky et al., 2004; Eclipse Project, 2008) open source framework is the next generation of metamodelling and object repository technology. It creates Java code for graphically editing, manipulating and serialising data based on a model specified in XML Schema, UML, or annotated Java. EMF tends to take a bottom-up approach whereas MOF tends to take a top-down approach and it is aligned with OMG's EMOF definition.

The definition of a metamodel is frequently a very syntactic operation that lacks rigour (Combemale et al., 2006). There have been recently several attempts to develop methodologies and support tools that tackle this issue. The integration of the OCL (Object Management Group, 2006; Akehurst & Patrascoiu, 2004; Warmer & Kleppe, 2003) is the first step to provide better quality environments of metamodel definition. The Kermeta (Triskell team, 2008; Drey, Faucher, Fleurey, Mahé, & Vojtisek, 2008) workbench is a metaprogramming environment based on an object-oriented DSL optimised for metamodel engineering. Kermeta features a comprehensive environment for metamodel engineering such as an Eclipse plug-in that integrates with several other Eclipse functionalities. The environment includes: specification of abstract syntax; static semantics (OCL) and dynamic semantics with connection to the concrete syntax model; metamodel prototyping and simulation; model transformation; and aspect weaving.

2.3 Model Driven Architecture (MDA)

In line with the idea of raising the level of abstraction in the development of software by the use of transformations, OMG proposed MDA (Object Management Group, 2003). MDA is a framework for software development, where models are not used only to document and communicate the requirements and software design decisions. They are elevated to key artefacts that directly represent executable elements of the software. MDA defines three steps for going from concept to software realisation (Kleppe, Warmer, & Bast, 2003):

- A model of the software system is constructed that is independent of any implementation technology, e.g. independent of JEE (Sun, 2008), .NET (Microsoft, n.d.), etc. This type of model is referred to as a Platform Independent Model (PIM).

- The PIM is transformed into one or more Platform Specific Models

(PSMs), depending on the chosen mapping strategy. A PSM is tailored to specifying a system in terms of the implementation constructs that are available in one specific implementation technology, e.g., a database schema for a DB2 database, EJB Websphere platform, etc.

- The PSMs are, at last, transformed into code.

Using model-driven development to define metamodels for each language provides the necessary artefacts to perform transformations both at the PIM to PIM level and at the PIM to PSM level.

Since the PSMs are typically close to the implementation platform, the transformation step from PIM to PSMs is usually more complex than the step from PSMs to code. The idea behind MDA is that it should be possible to choose different mapping strategies with a different set of PSMs generated from the PIM, and that the transformations between the PIM, PSM, and code should be automated. The effect of automating the transformation from PIM to PSM means that the development team largely deals with models, where modeling, in turn, would become programming on a higher level. In this way, development will become more and more a modeling activity with the goal of producing a good, high-level, technology-independent model of the software system.

Clearly, to realise the MDA vision, one needs to be able to describe the transformation between PIM and different PSMs, and then have tools transform the PIM based on the description provided. Even though the details of MDA are still being refined, there are already a number of tools that offer support for the MDA framework, e.g., (Borland, n.d.; Software, 2007; IBM, 2007; Corporation, 2007).

2.4 Model Transformations

A Meta-Modeling framework can be used to formally define a language or a set of languages, e.g. UML. UML has been described as a family of languages (e.g., (Cook, 2000)) due to its rich set of notations and many possible interpretations of these notations (i.e., semantic variation points in the language). Furthermore, the many different models possible with UML allow developers to make abstractions, projections and refinements of the software system under development. However, the implied use of multiple models to describe the system, each of which describes a view of the system at a certain level of abstraction, increases the likelihood of inconsistencies between views and hence an additional overhead on the maintenance of models. This fact, along with the need to change models in pre-definable ways, has increased

the importance of R&D work on supporting automation of model-to-model transformations using a metamodel as a starting point.

As described in (Sendall, 2003), model transformation can be seen as taking a set of input parameters including source and target models, and producing a set of target model elements as output. Consequently, there is a mapping relation set up between the source model elements and the target model elements (one-to-many or many-to-one). One can imagine a number of different usage contexts that could be beneficial to a software development project; some of these include (Sendall & Kozaczynski, 2003; Selonen, Koskimies, & Sakkinen, 2001):

- Refining Models;
- Abstracting Models;
- Generating Additional Views;
- Synchronisation of models;
- Applying Software Patterns;
- Re-factoring Models.

Model transformations can be used and applied to several tasks/phases of the development process. Some of them include:

Refining Models: The development of an application can be logically viewed as being made up of several steps taking one from vision to realisation; this is essentially the process of stepwise refinement. It is a common problem analysis method of breaking down a problem level-by-level that supports incremental delivery, and parallel work during the development process;

Abstracting Models: Reverse engineering of models, i.e., going from more concrete models to more abstract ones, can be useful for modellers that wish to work/communicate at a higher level of abstraction. The task of abstracting from certain details can be seen as a transformation from a more concrete model to a more abstract model. Most of the time these types of tasks are usefull in the process of understanding concepts in the sense that are abstractions are provided in the domain of application dialect;

Generating Additional Views: The generation of different views from existing models can be useful for concentrating on a particular concern of the system where non-pertinent information is filtered out. Such approaches are useful for combating complexity and for offering a focused view on those parts of the system that relate to the perspective of a particular stakeholder. The generation of views can also be seen as a transformation from existing models;

Synchronisation of models: Ensuring consistency between models requires one to keep models up-to-date with changes made to one another (both in the vertical/refinement and horizontal/aspect sense), where there is an overlap in information shared by models. For example, the addition of a class attribute in one model could require the attribute to be added to other models that also shows the attributes of that class. This activity of model synchronisation can be seen as a transformation. Automating this task facilitates more consistent and correct models;

Applying Software Patterns: The need to apply architectural and design patterns and UML idioms can often arrive over the course of a software development project. The reasons for applying a pattern can be diverse: changing the design strategy (e.g., all uses of MVC-style collaborations are transformed into PAC-style collaborations), improving the design (e.g., remove circular dependencies between two components by introducing an intermediary component), optimising performance (e.g., grouping multiple remote procedure calls into a single call), or meeting other extra-functional requirements and quality attributes. The application of a software pattern can be seen as a transformation that evolves existing models;

Re-factoring: Evolution is a key point in any software project. As requirements change over time, a model may become difficult to maintain, i.e., it is in a form that is not well-suited to perceive future modifications. A way to improve the form of a model is to refactor it according to certain criteria. Re-factoring a model involves changes to the model with the constraint that they must preserve behaviour, i.e., the behaviour of the modelled system is the same before and after the re-factoring: only the form is different. Re-factoring a model implies a transformation in the same domain of application that evolves existing models.

There are two main approaches regarding how models can be transformed:

- Graph transformation systems make use of graph rewriting techniques to manipulate graphs (Rozenberg, 1997). A Graph Transformation

(GT) is defined in terms of a set of production rules. A production rule consists of a Left-Hand Side (LHS) graph and a Right-Hand Side (RHS) graph. Such rules are the graph equivalent of term rewriting rules, i.e., intuitively, if the LHS graph is matched in the source graph, it is replaced by the RHS graph.

- Program transformation techniques (also known as Meta-Programing or Generative Programing (GP)) are used in many areas of software engineering ranging from program synthesis to program optimisation and program re-factoring, to reverse engineering and documentation generation. Program and model transformation are an approach that allows to automatically generate software from a generative domain model (Czarnecki & Eisenecker, 2000). A generative domain model is a model of a system family that consists of a problem space, a solution space, and the configuration knowledge. The problem space defines the appropriate domain-specific concepts and features. The solution space defines the target model elements that can be generated and all possible variations (Moss & Muller, 2005). The configuration knowledge specifies illegal feature combinations, default settings, default dependencies, construction rules, and optimisation rules. GP introduces generators as the mechanisms for producing the target.

As a summary, model transformation is the task of taking a input in a certain domain and producing an output in another (or the same) domain: Taking one or more models as input and producing one or more models as output requires a clear understanding of the abstract syntax and the semantics of both the source and target (Sendall & Kozaczynski, 2003).

2.5 Query / Views / Transformations

Query / Views / Transformations (QVT) OMG proposal is a standard that provides standard means (languages and models) for expressing transformations, which are amenable to use by humans and execution by computers. In QVT, a transformation between candidate models is specified as a set of relations that must hold for the transformation to be successful (Ehrig et al., 2005). A transformation invoked for enforcement is executed in a particular direction by selecting one of the candidate models as the target. Relations in a QVT transformation declare constraints that must be satisfied by the elements of the candidate models. A relation, defined by two or more domains and a pair of **when** and **where** predicates, specifies a relationship that must

hold between the elements of the candidate models. A domain is a distinguished typed variable that can be matched in a model of a given model type. A domain has a pattern, which can be viewed as a graph of object nodes, their properties and association links originating from an instance of the domain's type (Object Management Group, 2007b).

We have been observing in the last year a significant increase of QVT compliant languages implementation (Jouault & Kurtev, 2006; ATLAS Group, 2007; DUPE et al., n.d.; ikv++ technologies, 2008).

The Kermeta workbench (Drey et al., 2008) provides with a metaprogramming environment based on an object-oriented Domain Specific Language (DSL) optimised for metamodel engineering. Kermeta is a metamodelling language which allows describing both the structure and the behaviour of the models. It is intended to be used as the core language of a model-oriented platform. It has been designed to be a common basis to implement Metadata languages, action languages, constraint languages or transformation language.

ATL (ATLAS Group, 2007, 2006; Jouault & Kurtev, 2006) is a model transformation language and toolkit developed in the context of Model-Driven Engineering (MDE). ATL provides ways to produce a set of target models from a set of source models by specifying a transformation logic which is itself a model. ATL is specified as both a metamodel and a textual concrete syntax and it is an implementation of the QVT proposal (Object Management Group, 2007b).

2.6 Meta-Model Composition

The reusability of metamodels is a fundamental point in MDA. In particluar, reusability of domain models from application to application is very important considering that transformations are also a key component of the methodology. Metamodels should be available in order to be extended and composed so that DSL concepts, present in other languages, can be used (Nordstrom, Sztipanovits, Karsai, & Ledeczi, 1999).

In (Ledeczi, Karsai, Volgyesi, & Maroti, 2001), the semantics of several metamodel compositional operators are defined. The following compositional operators are defined:

Equivalence Operator : The equivalence operator is used to show a full union between two UML class objects. The two classes cease to be two separate classes, but instead form a single class. Thus, the union includes all attributes and associations, including generalization, specialization, and containment, of each individual class;

Inheritance Operators : There have been defined two types of metamodel composition via inheritance. They are supported by GME(Institute for Software Integrated Systems, Vanderbilt University, n.d.),(Ledeczi, Maroti, & Volgyesi, 2001):

- Implementation Inheritance Operator: In this type of metamodel composition, the children inherits all of the parents attributes, but only the containment associations where the parent acts as the container. No other associations are inherited;
- Interface Inheritance Operator: This type of inheritance is the second operator of this type supported by GME. This operator means that the inheritance allows no attribute inheritance, but does allow full association inheritance, with one exception: containment relations where the parent functions as the container are not inherited.

Note that the union of *Implementation Inheritance* and *Interface Inheritance* operators gives the common inheritance, and their intersection is null.

These operators support a certain level of metamodel composition. They are fully implemented and integrated in the Generic Modeling Environment (GME).

Other metamodel composition operators have been defined by Emerson and Sztipanovits. The authors present a series of techniques for metamodel composition, such as:

Metamodel Merge: consists of stitching two DSMLs together into a unified whole. When the DSMLs include modeling constructs that capture a shared set of entities they are fused together. This is applied to the meta-objects with the same name and metatype;

Metamodel Interface: allows to compose metamodels of two DSMLs capturing two distinct but related domains. Using this composition requires the definition of an interface between the two modelling languages consisting of new modelling entities and relations. These relations do not strictly belong to either of the composed modelling languages;

Class Refinement: this operator is used when one DSML captures in detail a modelling concept that exists only as a *black box* in a second

DSML. The relationship between the two composed modelling languages is given by the hierarchical containment of the constructs of one metamodel within a single construct of another metamodel;

These types of metamodel composition defined in both (Emerson & Sztipanovits, 2006) and (Ledeczi, Karsai, et al., 2001) can be used in order to compose metamodels but only at the syntax level. It is an interesting technique if the language engineer is only interested in defining DSMLs with very little behaviour.

2.7 Package Merge

This is an approach for model extension defined in the UML2 specification (Object Management Group, 2007d). The model extension technique was defined Package Merge. This operation was defined to assist in modularizing the UML2 metamodel. Package merge is intended to allow modeling concepts defined at one level to be extended with new features.

It also defines compliance levels regarding merged packages. To support the exchange of models and interoperability between UML tools, UML2 partitions the set of all its modeling features into four horizontal layers called compliance levels (Zito, Diskin, & Dingel, 2006). Level L_0 only contains the features necessary for modelling the kind of class-based structures typically encountered in object-oriented languages. Level L_3, on the other hand, encompasses all of UML. It extends level L_2 with features that allow the modeling of information flows, templates, and model packaging.

The package defining level L_{i+1} is obtained from L_i by merging new features into the package describing L_i. For instance, the level L_1 package is created by merging 11 packages (e.g., Classes::Kernel, Actions::BasicActions, Interactions::BasicInteractions, and UseCases) into the level L_0 package. In the UML 2.1 specification, package merge is described by a set of transformations and constraints grouped by metamodel types. The constraints define pre-conditions for the merge, such as when two package elements 'match', while the post-conditions are given by the transformations. The merge of two packages proceeds by merging their contents as follows: Two matching elements are merged recursively. Elements that do not have a matching counterpart in the other package are deep-copied.

To illustrate package merge, consider a simple class model of students as in package `Students` on the left of Fig. 2.2.

Suppose we would like to extend this model with the information concerting the student's faculties as in the `Student Faculties` package. To

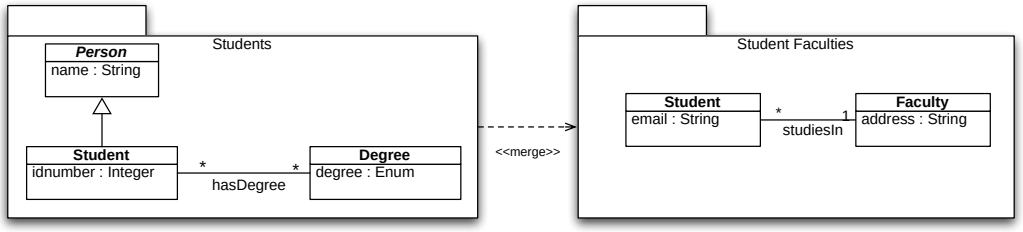


Figure 2.2. Specification of UML2 Package Merge

achieve this, package `Student Faculties` is merged into package `Students`, as indicated by the arrow in between the two packages. The `Students` package is denominated as the receiving package. Its elements (classes `Student`, `Person`, and `Degree` and the association `hasDegree`) are called receiving elements. `Student Faculties` is the merged package. Its elements (classes `Student` and `Faculty` and association `studiesIn`) are called the merged elements. The resulting package is obtained by merging the elements into the receiving package. Since the class `Student` in `Student Faculties` matches the class of the same name in `Students` package, the two are merged recursively by adding the property `email` to the receiving class. The class `Faculty` and the association `studiesIn`, however, do not match any elements in the receiving package. Using the package merge definition these elements are simply copied. The $\ll merge \gg$ arrow merely implies these transformations. Fig. 2.3 shows the result of applying the package merge technique.

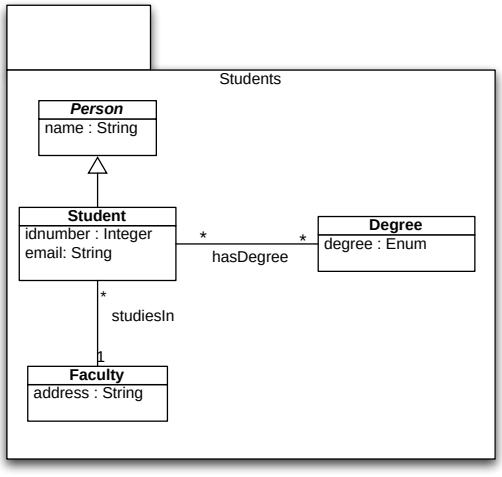


Figure 2.3. UML2 Package Merge Result

2.8 Model Extension

The *Model Extension* method is presented in (Barbero, Jouault, Gray, & Bezivin, 2007). The article presents a conceptual and practical approach to model extensibility, in which new models are created as derivations from base models. There are several situations where such an extensibility mechanism is useful and essential (e.g.,in the case of hierarchies of metamodels). In order to achieve the goal of model extension, the article is based on the existence of an additional relation between models called $extensionOf$.

When a metamodel is used as an extension of the initial metamodel, the $extensionOf$ relation is described as the 'composition' of those two metamodels into a single one. This 'composed' metamodel is derived from the two metamodels: when a concept exists in both metamodels, the result of the extension is the merging of elements from the initial metamodel and from its extension.

Due to the relevance, simplicity and similarities with some examples we present in this document, we will re-use the example provided in (Barbero et al., 2007). The example is based on the Petri formalism. It starts by first describing a classical Petri net: a set of places and transitions linked by directed arcs. Arcs run from a place to a transition or from a transition to a place. The metamodel depicted on Fig. 2.4 specifies all the concepts of a simple Petri net.

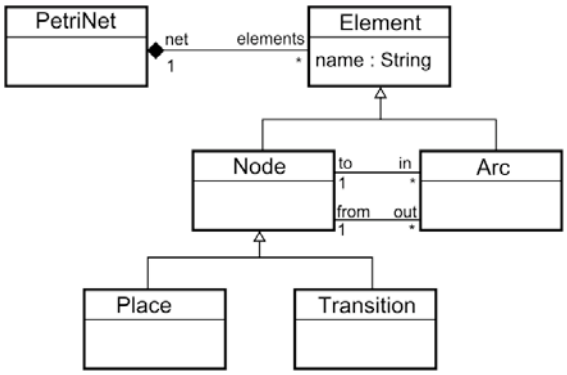


Figure 2.4. Metamodel Extension: Petri Nets Metamodel prior to Extension

The metamodel in Fig. 2.4 does not allow the design of Petri nets with a specific execution state. To overcome this limitation the solution is to add the concept of a state description to the Petri net metamodel. In the example the authors propose to extend the previous metamodel with the concepts describing the state of a Petri net at a given time. Such a Petri net is said to be marked. A marked Petri net can be represented by attaching a set of

Tokens to some Places. This addition does not affect the previously defined structure of the Petri net.

The metamodel of the Petri net with marking is depicted on Fig. 2.5. It adds the concept of Marking as a set of Tokens. Each Token is associated with a Place.

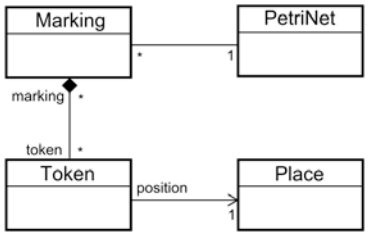


Figure 2.5. Metamodel Extension: Marking Extension

This new metamodel is an extension of the original Petri net metamodel. The final metamodel obtain by applying metamodel extension can be depicted in Fig. 2.6. This combination merges the common concept of Place to build the complete metamodel.

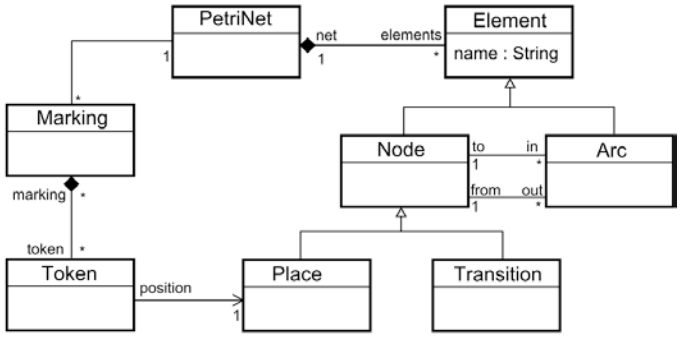


Figure 2.6. Metamodel Extension: Petri Nets Metamodel with Marking Extension

The Model Extension approach presents a conceptual framework with an implementation defined in KM3 (Jouault & Bézivin, 2006), a DSL for Metamodel Specification.

2.9 Transformation Composition

Decomposition and separation are common approaches to solve complex tasks. In the world of transformations, this approach is also important while dealing with complex models and in order to allow reusability. Decomposition may be implemented using several approaches (Marvie, 2004).

- at the level of the transformation definition language. A transformation is defined using basic transformations implying the compilation of the transformation definitions. This approach relies on the definition of a new transformation based upon an existing one;

- defining executable transformations as a composition of executable elementary transformations. This approach is similar to component based engineering.

The key idea of compositional transformation is to decompose the system specification into properties that describe the behaviour of its subsystems. In (Caporuscio, Ruscio, Inverardi, Pelliccione, & Pierantonio, 2005), a technique and a tool are presented that allows decomposition of software applications checking the existence of mutual dependencies among components.

Since transformations are also models described in certain transformation languages, there are also benefits of defining transformation patterns. Some of these patterns include (Bezivin, Jouault, & Palies, 2005):

1. **Transformation parameters** - in many model transformations some attributes of the target metamodel are hard-coded. They have become parameters for the transformations as they have side effects on the result;

2. **Multiple matching patterns** when there has to be one rule that matches a collection of source elements.

2.10 Semantic Anchoring

In (K. Chen, Sztipanovits, Abdelwalhed, & Jackson, 2005), a technique is presented that allows to 'anchor' semantics to a metamodel. This technique is mainly based on the use of Abstract State Machines (ASM) and the Graph Rewriting and Transformation (GReAT) as instruments in the GME (Software Integrated Systems, 2005). Semantic units are defined by attaching a transformation to each metamodel in the GME.

The work presented in (K. Chen et al., 2005) proposes the use of a metamodeling process within GME for rapid and inexpensive development of DSMLs. However, in order to simplify the DSML semantics, this work discusses semantic anchoring which is based on the transformational specification of semantics. It uses the mathematical model, ASM, as a common semantic framework. It shows a formal operational semantics for a set of basic models of computations called semantic units. Semantic anchoring of

DSMLs means the specification of model transformations between DSMLs and selected semantic units.

Although there are several conceptual similarities between the work presented in this thesis and the work on *Semantic Anchoring* (both approaches use a model transformation language to provide semantics), we mostly concentrate on the compositional and extension aspects of metamodels. In particular, we focus on transformations modularization and re-use, while the work defined in the *Semantic Anchoring* approach concentrates mostly in attaching pieces of semantics to a metamodel.

2.11 Summary

This chapter confirms the diversity of methodologies and techniques available in Model-Driven Engineering (MDE) and, in particular, for Domain Specific Modelling Language (DSML) support. In such an heterogeneous environment, it is often not easy to interface and to be able to use the ones one might need to provide a DSML specification. Albeit this fact, there is a clear path going in the direction for the standardisation[1] of the processes for DSML development and a perspicuous concern in the language semantics.

2.11.1 Merging New and Old Ideas

The contents of this section act as a small comparison of the techniques against the work presented in this thesis. Therefore this section could also be titled *Introduction to the Scientific Contribution of This Work*.

The work described so far compares with the one on this thesis as:

Meta-Model Composition All the work available that presents 'techniques for metamodel composition' only defines constructs for the syntax domain of the language. All the techniques present and define a set of operators that allows to compose metamodels or, in other words, to compose the abstract syntax of the language. These techniques are very powerful and easy to use if we don't need to manipulate semantic constructions. Compared with the work presented in this thesis, we can say that metamodel composition is a subset of the operations provided. We do not provide a set of composition operators explicitly but the parameterization mechanism available is powerful enough to be able to express any of these metamodel composition operators;

[1] For example, the evolution of UML from it first version with minimum definition, to UML2 with a comprehensive metamodel and a much more precise definition.

Package Merge: This method is an UML-specific operation that is difficult to express for other kinds of metamodels. The package merge operation has not been provided in a clearly defined conceptual framework and it only works at the metamodel level. It is also a very syntactical operation that lacks precision since it is defined in a very informal fashion. This type of composition is not supported in the work presented in this thesis. We argue that merging domain concepts is a very ambiguous operation. A merge operation implies that none or very small definition on how elements of the domain concepts to be merged is provided. Moreover, the semantics of package merge is perceived as complicated. For instance, the UML manual describes it as *complex and tricky*;

Model Extension: Model extension is a technique for achieving some level of reusability. In the of context model engineering, model extension plays the same type of role as class inheritance in the object technology environments. It is a technique that, again, works at a syntactic level. In the syntactic domain this approach allows to extend models with new concepts. The methodology we propose in this thesis also supports this kind of composition. If we want to compare both approaches we could say that Model Extension is a technique for simple model extension whereas the method in this thesis represents a generic approach for model extension. While Model Extension approach only allows to extend one element of the metamodel at a time, the approach we propose allows to extend and parameterize a subset of the metamodel. However, in the approach presented in this document, the semantical aspects of the language are also considered;

Semantic Anchoring: It is fair to say that semantic anchoring is the closest technique to the work presented in this thesis. The work on semantic anchoring describes how to add semantics to a DSML (or a domain concept). What is not defined in the Semantic Anchoring technique is how to compose the semantic blocks. Semantic Anchoring allows to specify the semantics of a DSML by transforming it to another (more precise domain) but it is not possible to compose the transformations. Consequently, it is impossible to provide reusability of the defined semantic blocks. Compared with Semantic Anchoring this work also allows to 'anchor' semantics to a domain concept. The methodology we present goes a step further and does not allow to have a metamodel without an associated transformation. In addition to Semantic Anchoring, we provide the necessary constructs for composing semantic blocks in a very abstract way.

Furthermore, the work presented describes a method that is not bound to a particular technology. Although we will also express an implementation in an integrated and full featured environment, the technique presented is general enough to allow other implementations. The work that we will start describing can be seen as a generic conceptual framework for:

a) generic **metamodel composition**;

b) automation of the correspondent **model composition**;

c) semantics definition based on **transformation composition**.

Each one of these points allow a high level of reusability at all levels.

Chapter 3

State of the Art: Tools

While the previous chapter concentrated in the more methodological aspects, this one focuses on the pragmatic angle of the development of Domain Specific Modelling Languages (DSMLs).

3.1 Domain Specific Modelling Languages

Software development as an engineering area focuses on the usage of a set of established methodologies that can be systematically applied to solve new problems (Thibault, 1998) namely: reducing time, increasing quality, ensuring user safety, etc.

Domain Specific Language (DSL) (Deursen, Klint, & Visser, 2000) are programming languages or executable specification languages dedicated to a particular domain or problem. They address the issues stated before, especially in the areas in which evolution at a rapid rate is crucial. DSLs provide, through appropriate built-in abstractions and notations, expressive power focused on a particular problem domain.

DSLs have been used because different domains of knowledge demand differentiated treatment concerning their support in terms of software languages. These languages are much more expressive in their domain, allowing domain experts to understand, validate, modify, and often develop DSL programs themselves (Ladd & Ramming, 1994).

There are several approaches used for defining DSL (Luoma, Kelly, & Tolvanen, 2004) most of them based on the metamodel definition. Some of these approaches are: Domain expert's or developer's concepts; Generation output; Look and feel of the system built; Variability space.

Each (independent) metamodel can be considered analogous to an abstract syntax for a DSL. Taking this into account, model-based DSLs play

an important role in MDA. Its contribution enabled the support for definition and use of new standards (Gerber & Lawley, 2004) (e.g. UML Profile for enterprise distributed Object Computing (Object Management Group, 2004) standard, with domain-specific metamodels for component collaboration and business process modelling). Tools like the the ones described in (Kosayba, Marvie, & Geib, 2004) and (Vangheluwe & Lara, 2004) allow a domain-specific visual environment through a visual reconfiguration of the metamodel. Such tools profit from the MDA approach (Bichler, 2003) to go from a PIM to a specific modelling tool also considering different graphical representations for it.

3.2 Domain Specific Languages Development Environments

A number of tools exist which allow one to define a domain-specific visual language by the specification of a metamodel or by using *domain model* diagrammatical conventions. Some of these Integrated Development Environments are compared in this section by using the following criteria:

Language Evolution the possibility to add, remove and modify language concepts and how these modifications propagate;

Verification the availability of metamodel and model verification; the ability of producing test case and oracle generation;

Graphical Mapping if it allows graphical mapping for elements and connections, and the generation of menus and tool-bars;

Transformation the capability for defining transformation; which transformation language is supported; possibility of re-factoring models and metamodels; the availability of model analysis functionalities;

Composition the availability of features to manage re-use of concepts and to compose them.

The Integrated Development Environments (IDEs) that we will consider in this comparison are:

- The Generic Modeling Environment (GME) (Software Integrated Systems, 2005);
- Atom3 (Vangheluwe & Lara, 2004);

- Eclipse Modeling Framework (EMF)/Graphical Modeling Framework (GMF) (Budinsky et al., 2004);
- MetaEdit+ (MetaCase Consulting, 2008)
- Microsoft DSL Tools (Cook, Jones, Kent, & Wils, 2007)

The next sections of this chapter are dedicated to describing each one of the tools according to the criteria previously defined.

3.2.1 Generic Modeling Environment (GME)

GME is an modelling environment developed at Vanderbilt University. In GME, a DSML is described as a paradigm which is essentially a metamodel. GME comes with a DSML plugin that, according to our defined criteria, is characterised as:

Language Evolution `High`: Evolving paradigms can be preserved if elements, links and references are not renamed or removed;

Verification `Medium`: It provides a model and a metamodel checker for their structure and constraints.

No test case or Oracle generator is provided.

Custom validation categories can be added and the code generators can be modified to allow test and oracle generation;

Graphical Mapping `Medium`: This IDE supports decorators for both elements and connections. The visual drawing of an object is achieved with the definition of Component Object Model (COM) also called decorators. In addition simple bitmaps can be used as icons;

Composition `Medium`: GME provides some syntactical constructs for composition:

- Metamodel Merge operator that identifies metamodel join points as metamodel elements with identical names;
- Interface Inheritance;
- Implementation Inheritance.

It also allows re-use of registered paradigms by importing them into paradigms under development.

Transformation `High`: Transformations in GME are possible by using one of two possibilities:

1. Integrating with the GReAT system;
2. Using the generated C++ and Java interfaces generated from the metamodel structure.

Composition of transformations is not available.

3.2.2 Atom3

The Atom3 IDE is a Tool for Multi-formalism and Meta-Modelling. The main idea behind this tool is 'Everything is a model'. Models and metamodels are always manipulated as graphs. The metamodel based formalism is based on a Entity-Relationship model with additional constraints. Defined metamodels (formalisms) can by used as the base formalism for defining new metamodels.

Language Evolution `Low`: Language Evolution is not supported by Atom3;

Verification `Medium`: This tool provides neither test case nor oracle generator. Model validation is executed on-the-fly during specification: model integrity and constraint solver are applied to the graph;

Graphical Mapping `High`: All elements defined in an Atom3 metamodel can be provided with a graphical representation.

In this development environment, the user can easily create or import new graphical elements. Bitmaps can be used to represent any kind of objects and connections and an internal graphical editor is made available for designing graphical elements.

Tool bars can be defined and generated;

Composition `None`: No explicit composition mechanisms are available within the Atom3 framework;

Transformation `Medium`: Only graph grammars can be used to perform model to model manipulation or to generate textual representations. In order to express a model transformation, Python scripting language must be used.

No explicit functions of transformation composition is available.

3.2.3 EMF/GMF

The Eclipse Modeling Framework (EMF) project is a modelling framework and code generation facility for building tools based on a structured data model. The metamodel formalism which is used for defining metamodels is the Ecore formalism.

The Graphical Modeling Framework (GMF) provides a generative component and runtime infrastructure for developing graphical editors based on EMF.

Language Evolution `High`: Evolving paradigms can be preserved if elements, links and references are not renamed or removed. User can perform manual modifications and, by using a special comment in the Java source code, the modifications will be kept even if modifications are performed in the metamodel;

Verification `High`: Within the EMF/GMF, framework it is possible to easily use JUnit test case generator and Oracle generator for the JUnit tests generated.

Simple serialisation tests are generated automatically from the metamodel. A functionality for model and metamodel validation is available by user action.

Taking into account that EMF/GMF are part of the Eclipse framework, it is easy to integrate with EclEmma a Java Code Coverage tool;

Graphical Mapping `Very-High`: In this metamodelling development framework all elements present in a metamodel can be graphically represented. Groups of objects are possible to define as well as compartments and containers. An associated behaviour can be defined to each one of these;

Tool bars which may include menu items can be generated.

The user can easily use new graphical elements in most of the image formats including Scalable Vector Graphics (SVG). GMF allows to import an extensive set of Icons and Connection elements present in the standard Eclipse repository;

Composition `None`: The EMF/GMF environment does not support any kind of composition mechanisms;

Transformation `High`: Most of the transformation languages available are Ecore-based and can be installed as an Eclipse plug-in. This is an

advantage of using EMF/GMF. Graph grammars and QVT-based languages are available (e.g. ATL and Kermeta).

In addition, it is possible to use the generated Java interfaces and to extend the automatic generated Java programs for model manipulation.

Functionalities for transformation composition are not available.

3.2.4 MetaEdit+

MetaEdit+ is a multi-user and multi-platform environment that supports simultaneously both system development and method development. It is a commercial tool developed by MetaCase. It is know for its successful case studies at Nokia, AT&Tm Fusjitsu Siemens, etc.

This environment uses Graph-Object-Property-Port-Role-Relationship (GOPPRR) as the formalism for defining metamodels. Each one of these are denominated metatypes.

Language Evolution `High`: Generated and hand written code can be separated. Protected blocks can be defined into the generated result;

Verification `Medium/High`: Provides metrics and model checking.

Does not generate automatically test cases but the code generator is configurable.

This tool also allows model animation for simple validation operations;

Graphical Mapping `High`: This tool provides a Symbol Editor and an extensive Library of available symbols. Symbols can be changed according to model data. Functionalities of Import/Export symbols are also available.

The user can create and configure menus, toolbars and a printing autolayout is present;

All elements in the metamodel can be mapped into a graphical representation using multiple behaviours;

Composition `Low`: Provides a library of 70+ metamodels that can be reused;

Transformation `Medium`: Transformation under MetaEdit+ is possible by using the Generator Editor, i.e. customisation of code generators. Another option is to use the automatically generated interfaces.

There is no support for transformation composition.

3.2.5 Microsoft DSL Tools

DSL tools from Microsoft are able to generate visual designers that are customised for a problem domain. It is based on the .NET framework and some of the Visual Studio modules were developed using DSL tools. For example the Distributed Systems Designers or Class Designer are modules entirely developed using Microsoft DSL Tools.

It uses a Domain Model as metamodeling formalism. Domain Model is built using pre-defined diagrammatic conventions. Many of these diagrammatics are derived from UML and include conventions representing structure and representing behaviour.

Language Evolution `High`: DSL Tools provide a 'Custom Code' repository in which users can add all their manual instructions. This repository is always preserved during the DSML development cycle;

Verification `High`: Allows model and metamodel, toolbox, and menu validation; Provides constraint solver against models in order to check its integrity against the defined metamodel.

No automatic unit test generators are available but Microsoft Visual Studio provides functionalities that help define them and to automate its execution. A code coverage tool is also available.

Custom validation categories can be added and code generators can be modified to allow test and oracle generation;

Graphical Mapping `Very-High`: A graphical definition can be assigned to all elements and all connectors. Elements can be of one of five shapes: geometry, compartment, image, ports and swim lanes.

This environment supports Inheritance between shapes which have a high degree of configuration.

Menus and toolbars are automatically generated and customisable;

Composition `None`: No composition mechanisms are available in Microsoft DSL Tools;

Transformation `Medium`: Transformations in this environment can only be specified by using the derived C# interfaces.

Composition of transformation is not supported.

3.3 Summary

Table 3.1 provides a summary of the criterias used for comparison of each one of the tools presented.

As we can see from the previous description, there are already quite some tools available in the market for DSMLs development. Some are open source, others commercial and others are academic tools. All of them base the their DSML development philosophy by starting to create a metamodel of the language.

In terms of abstract and concrete syntax, they all meet very well the requirements for a DSML development environment. Some contain fancier features than others but, overall, all of them can cope with all syntax aspects of a language. We should just emphathise that, although it might not be the easiest tool to use or the more intuitive, the EMF/GMF framework is probably the most powerful due to one key aspect: it is integrated in the Eclipse platform and a lot of plugins exist and can be integrated in order to have a more powerful development environment. It is probably the only platform that allows, for example, to develop textual DSLs using the same framework and integrate it with a graphical part.

Concerning semantical aspects, all the tools presented are less powerful than what is desired for a DSML development environment. When providing language semantics, they all rely very much on either the built-in code generator templates or in the specific APIs generated for each one of the DSMLs. These tools, in general, offer the user one of three different architectural approaches for defining transformations: the direct model manipulation approach, the intermediate representation approach, or the transformation language support approach. It is our belief that this is not enough and that, in the future, a better manner for defining (or attaching) semantic information should be available.

As we can see from table 3.1 at least one of the tools presented has `High` or `Very High` in all of the comparison criteria except for the `Composition` one. We consider this to be a lack on how tools have thus been so far developed. This fact, together with the idea that a more semantical point of view must be added to DSML development environments, justifies the approach presented in the thesis.

Because the Eclipse platform is open source, easier to interface and provides a lot of extra features, the implementation of the methodology we are going to present, will be implemented as a plugin for this platform.

	DSML Tools Development Environments				
Criteria	GME	Atom3	EMF/GMF	MetaEdit+	DSL Tools
Language Evolution	High	None	High	High	High
Verification	Medium	Medium	High	Medium/High	High
Graphical Mapping	Medium	High	Very-High	High	Very-High
Transformation	High	Medium	High	Medium	Medium
Composition	Medium	None	None	Low	None

Table 3.1. DSML Tools Comparison Summary

Chapter 4

Model Structure Formalisation

In this chapter we provide a formalisation for metamodels and models. This will allow us to describe the models and metamodels from a structural perspective.

The first part of the chapter is dedicated to the definition of metamodels while the second section focuses on the definition and formalisation of models construction.

4.1 ECore Formalism

In this section we formalise a metamodel. For this definition our work is based on the meta-metamodel definition of Ecore (Eclipse Project, 2008) provided in (Budinsky et al., 2004). Ecore is a formalism for describing metamodels based on a set of concepts and relations between them. Ecore is compliant with the standard definition (Object Management Group, 2007a) of metamodelling languages for industrial modelling tools provided by the OMG. The Fig. 4.1 shows a graphical representation of a simplified version of the Ecore metamodel using UML-like class diagrams syntax. The Ecore metamodel is a meta-metamodel. It corresponds to the top of the Fig. 2.1 (Sec. 2.2) that shows the 4-layers Architecture of a metamodelling framework

An Ecore metamodel defines a set of concepts used for defining metamodels. Metamodels are characterised by:

- a set of metaclasses of type `EClass`. Each `EClass` has a `name`, zero or more attributes, and zero or more references;
- attributes that are described by the Ecore class `EAttribute`. Attributes are defined by a `name` and by a type `EDataType`;

- references that are used to represent one end of an association between classes. A reference is represented by the metaclass `EReference`. A reference is defined by a `name`, a boolean flag to indicate if it represents containment, a multiplicity defined by a lower and an upper bound;
- data types represented by the class `EDataType`. A data type can be a primitive type (e.g. `EInteger`, `EString`, etc.) or user defined `EENum`;
- inheritance relations allowing each metaclass to have a super-class. This relation is represented by the `eSuperTypes` association relation in `EClass` class.

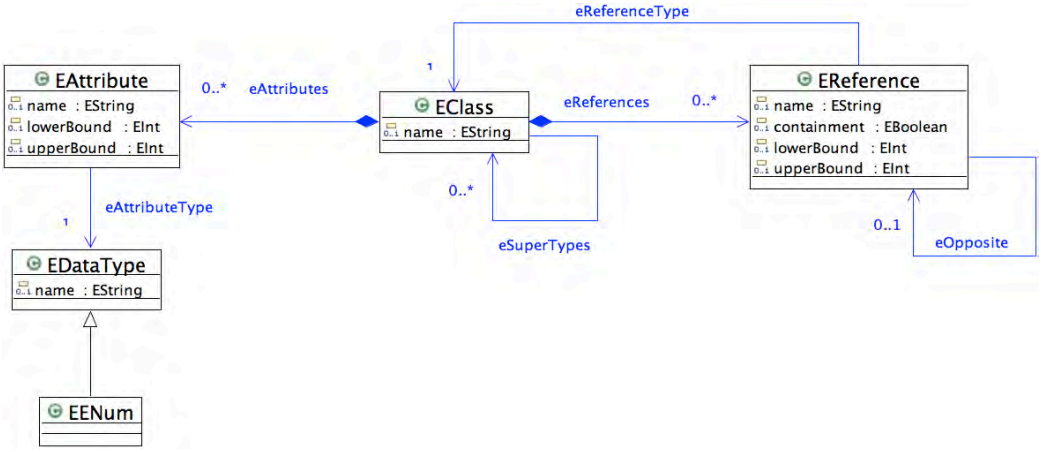


Figure 4.1. Ecore Metamodel

4.2 Metamodels Formalisation

According to this definition and the Ecore meta-model in Fig. 4.1 we propose a metamodel formalisation. This formalisation is based on a set of metaclasses and the relations between them.

Definition 4.1. Universe of Metamodels.
MM corresponds to the universe of Metamodels.

Definition 4.2. Metamodel.
A metamodel is a 13-tuple

$$mm = \langle MC, MAT, MR, PT, MetaClass, MAttrType, MAttrLower,$$
$$MAttrUpper, RefType, MContainment, MLower, MUpper, MSuper \rangle$$

where:

- MC is a finite set of meta-classes;
- MAT is the finite multiset of meta-attributes. MAT is the union of the set of meta-attributes in each meta-class: $\bigcup_{mc \in MC} MAT_{mc}$
- MR is the finite set of meta-references;
- $PT = \{PT_1, PT_2, ..., PT_n\}$ is the set of primitive types where each primitive type is defined by the set of values D_{PT_n};
- $MetaClass : MAT \cup MR \longrightarrow MC$ The function that returns the meta-class that each meta-attribute or meta-reference belongs to;
- $MAttrType : MAT_{mc} \longrightarrow PT$ The function that links to each meta-attribute a Primitive Type;
- $MAttrLower : MAT \longrightarrow \mathbb{Z}^+$ The function that returns the lower bound of the meta-attribute.
- $MAttrUpper : MAT \longrightarrow \mathbb{Z}^+$ The function that returns the upper bound of the meta-attribute;
- $RefType : MR \longrightarrow MC$ The function that links to each meta-reference a meta-class type;
- $MContainment : MR \longrightarrow \{true, false\}$ The function that specifies if a meta-reference is a containment relation;
- $MLower : MR \longrightarrow \mathbb{Z}^+ \cup \{*\}$ The function that returns the lower bound of the meta-reference;
- $MUpper : MR \longrightarrow \mathbb{Z}^+ \cup \{*\}$ The function that returns the upper bound of the meta-reference.
- $MSuper : MC \times MC$ The inheritance relation at the meta-class level. This function function defines a specialisation relation between two meta-classes.

Definition 4.3. Metamodel elements singularity.
The set of elements in a metamodel are unique:

$$(MC \cap MAT) = \emptyset$$

$$(MR \cap PR) = \emptyset$$

Moreover, the completeness of a metamodel formalisation follows the specification of the following definitions.

This formalisation defines a metamodel as a set of meta-classes MC, meta-attributes MAT and meta-references MR. The function $MetaClass$ allows to link each meta-reference and each meta-attribute to its unique meta-class; function $MAttrType$ returns the primitive type of a meta-attribute; and function $MRefType$ that returns the type of a meta-reference. In addition, the metamodel is defined by an inheritance function named $MSuper$.

4.2.1 Metamodel example

As an example let us use an excerpt of the Petri Nets metamodel presented in in Fig. 4.2. This example uses Ecore standard graphical representation. It is a simplified version of a Petri Nets metamodel used in this section for simplicity reasons. In further sections more detailed Petri Nets metamodels will be introduced.

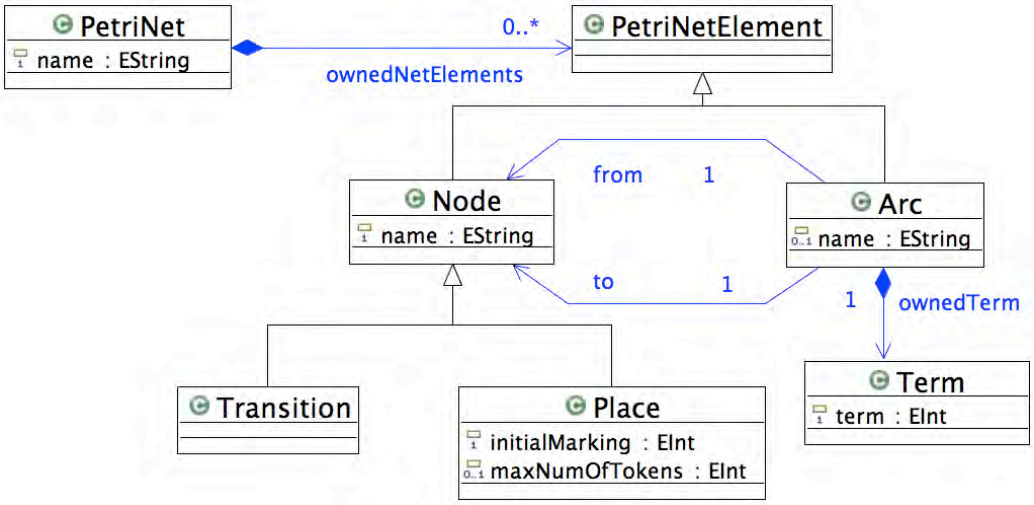


Figure 4.2. Simplified Petri Nets Metamodel

In a formal fashion, this particular Petri Nets metamodel is defined as follows:

- $MC = \{PetriNet, PetriNetElement, Node, Arc, Transition, Place, Term\}$ Representing the set of meta-classes in the metamodel;
- $MAT = MAT_{PetriNet} \cup MAT_{PetriNetsElement} \cup MAT_{Node} \cup MAT_{Arc} \cup MAT_{Transition} \cup MAT_{Place} \cup MAT_{Term}$

  The set of meta-attributes indexed by meta-class.

- $MAT_{PetriNet} = \{name\}$
  - $MAttrLower(name) = 1$, $MAttrUpper(name) = 1$
    Meaning that the meta-attribute `name` for the meta-class `PetriNet` has a $1-1$ cardinality. This implies a mandatory `name` while defining Petri Net models.
- $MAT_{PetriNetsElement} = \{\}$
- $MAT_{Node} = \{name\}$
  - $MAttrLower(name) = 1$, $MAttrUpper(name) = 1$
- $MAT_{Arc} = \{name\}$
  - $MAttrLower(name) = 0$, $MAttrUpper(name) = 1$
    The fact that $MAttrLower(name)$ is set to 0 and $MAttrUpper(name)$ is set to 1 means that Petri Nets' Arcs are not forced to have a name.
- $MAT_{Transition} = \{\}$
- $MAT_{Term} = \{term\}$
  - $MAttrLower(term) = 1$, $MAttrUpper(name) = 1$
- $MAT_{Place} = \{initialMarking, maxNumOfTokens\}$
  - $MAttrLower(initialMarking) = 1$, $MAttrUpper(initialMarking) = 1$
  - $MAttrLower(maxNumOfTokens) = 1$, $MAttrUpper(maxNumOfTokens) = 1$
- $MR = \{ownedNetElements, from, to, ownedTerm\}$
- $PT = \{EString, EInt\}$
- $MetaClass(ownedNetElements) = PetriNet$,
  $MRefType(ownedNetElements) = PetriNetElement$
  $Containment(ownedNetElements) = true$
  $Lower(ownedNetElements) = 0$
  $Upper(ownedNetElements) = *$

- $MetaClass(ownedTerm) = Arc$
  $MRefType(ownedTerm) = Term$
  $MContainment(ownedTerm) = true$
  $MLower(ownedTerm) = 1$
  $MUpper(ownedTerm) = 1$

- $MetaClass(from) = Arc$
  $MRefType(from) = Node$
  $MContainment(from) = false$
  $MLower(from) = 1$
  $MUpper(from) = 1$

- $MetaClass(to) = Arc$
  $MRefType(to) = Node$
  $MContainment(to) = false$
  $MLower(to) = 1$
  $MUpper(to) = 1$

- $MAttrType(name) = EString$

- $MSuper =$
  $\{(Transition, Node), (Place, Node)\}$

4.3 Models formalisation

Models are artefacts that are written in the language of a particular metamodel. Therefore, a model is a set of constructions that are in conformity with a metamodel as described in the previous section.

This section provides a set of definitions that allows to formally define a model in conformity with a particular metamodel.

Definition 4.4. Universe of Models.
M is the universe of Models.

Definition 4.5. Model.
Giving a metamodel mm

$$mm = \langle MC, MAT, MR, PT, MetaClass, MAttrType, MAttrLower,$$

$$MAttrUpper, RefType, MContainment, MLower, MUpper, MSuper\rangle$$

a model is defined as set of model elements:

$$m_{mm} = \langle ME, MClass, ElemAttr, ElemRef, Mult \rangle$$

where

- ME is the set of the model elements;
- $MClass : ME \longrightarrow MC$ is the function that gives the unique meta-class for a model element; For each element of a model this functions associates a unique meta-class.
- $ElemAttr : ME \times AT \longrightarrow D_{pt}$ the function $ElemAttr$ that takes as input a model element and an attribute. It returns a set of values of this attribute on its primitive type, having:
  - $me \in ME$, an element me of the model ME
  - AT is the set of attributes of the meta-class: $AT = MAT_{MClass(me)}$, the set of attributes of the meta-class $MClass(me)$;
  - $\forall at \in AT, pt = MAttrType(at)$, the domain of the values of the attribute at;
  - $ElemAtr(me, at) \in D_{MAttrType(at)}$.
- $$ElemRef : ME \times MR \longrightarrow (ME)^j$$
  the function $ElemRef$ that takes as input a model element and a reference. It returns one or several elements of the model of the meta-reference type, with:
  - $me \in ME$, a model element of ME;
  - $ref \in MR$ a meta-reference from the set of meta-refences of the meta-class of me;
  - $MetaClass(ref) = MClass(me)$.

  the function that associates to (me, ref) a k-uplet so that $ElemRef(me, r_k) = (me_1, ..., me_j)_k \in (ME)^k$ in the domain of $MRefType(ref_k)$, where $j \in \mathbb{Z}^+ : MLower(r_k) \leq j \leq MUpper(r_k)$
- $Mult : ME \times MR \longrightarrow \mathbb{Z}^+$ the function $Mult$ that returns the number of model elements of type $MetaClass(me)$:
  - $me \in ME$, a model element of ME

- $ref \in MR$ a meta-reference from the set of meta-references of the meta-class of me
- $MetaClass(ref) = MClass(me)$

Definition 4.6. Model Conformity.

Giving mm a metamodel and m a model, the $conformsTo$ is a relation that returns the metamodel to which a given model is in conformity with:

$$mm \models_{conformsTo} m$$

A model is a set of 'model elements' that satisfy a given reference metamodel.

4.3.1 Model Example

In the Fig. 4.3 we show an example of a Petri Nets model. This model is representative of a machine with two states: `stopped` and `running`, as well as two transitions `start` and `stop` that allow to go from one state to the other. The arcs that connect the transitions and states are also depicted.

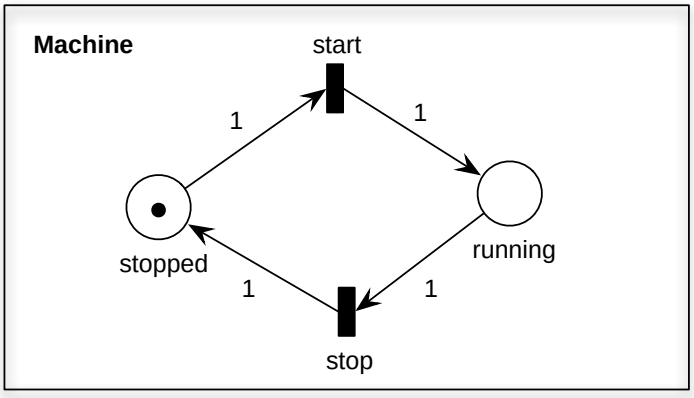


Figure 4.3. Petri Nets Two State Machine Model

Formalising this model using the previous definitions results in the following description:

Model elements The elements that defined the Petri Net in Fig. 4.3 are: $\{me_{pn}, me_{p1}, me_{p2}, me_{tr1}, me_{tr2}, me_{a1}, me_{a2}, me_{a3}, me_{a4}, t_1, t_2, t_3, t_4\} \in ME$ with:

- me_{pn} the Petri Net machine: $MClass(me_{pn}) = PetriNet$;

- me_{p1} and me_{p2} Petri Net `stopped` and `running` places: $MClass(me_{p1}) = Class(me_{p2}) = Place$;
- me_{tr1} and me_{tr2} the `start` and `stop` transitions respectively: $MClass(me_{tr1}) = Class(me_{tr2}) = Transition$
- me_{a1}, me_{a2}, me_{a3} and met_4, the Arcs that connect transitions to places and vice-versa: $MClass(me_{a1}) = Class(me_{a2}) = Class(me_{a3}) = Class(me_{a4}) = Arc$
- t_1, t_2, t_3 and t_4, the Terms associated to the Arcs.: $MClass(t_1) = Class(t_2) = Class(t_3) = Class(t_4) = Term$

Model Element Attributes The functions that characterise the attributes at the model level are defined as:

- $ElemAttr(me_{pn}, name) = \{Machine\}$, $Mult(me_{pn}, name) = 1$
- $ElemAttr(me_{p1}, name) = \{stopped\}$, $ElemAttr(me_{p1}, initialMarking) = 1$
- $ElemAttr(me_{p2}, name) = \{running\}$, $ElemAttr(me_{p2}, initialMarking) = 0$
- $ElemAttr(me_{tr1}, name) = \{start\}$
- $ElemAttr(me_{tr2}, name) = \{stop\}$
- $ElemAttr(t_1, term) = \{1\}$
- $ElemAttr(t_2, term) = \{1\}$
- $ElemAttr(t_3, term) = \{1\}$
- $ElemAttr(t_4, term) = \{1\}$

Model Element References the definition of relation between elements are defined as follows:

- $ElemRef(me_{pn}, ownedNetElements) = \{me_{p1}, me_{p2}, me_{tr1}, me_{tr2}, me_{a1}, me_{a2}, me_{a3}, me_{a4}, t_1, t_2, t_3, t_4\}$, $Mult(me_{pn}, ownedNetElements) = 12$
- $ElemRef(me_{a1}, ownedTerm) = \{t_1\}$, $Mult(me_{a1}, ownedTerm) = 1$
- $ElemRef(me_{a2}, ownedTerm) = \{t_2\}$, $Mult(me_{a2}, ownedTerm) = 1$
- $ElemRef(me_{a3}, ownedTerm) = \{t_3\}$, $Mult(me_{a3}, ownedTerm) = 1$

- $ElemRef(me_{a3}, ownedTerm) = \{t_4\}$,
  $Mult(me_{a4}, ownedTerm) = 1$
- $ElemRef(me_{a1}, from) = \{me_{p1}\}$, $Mult(me_{a1}, from) = 1$,
  $ElemRef(me_{a1}, to) = \{me_{t1}\}$, $Mult(me_{a1}, to) = 1$;
- $ElemRef(me_{a2}, from) = \{me_{t1}\}$, $Mult(me_{a2}, from) = 1$,
  $ElemRef(me_{a2}, to) = \{me_{p2}\}$, $Mult(me_{a2}, to) = 1$;
- $ElemRef(me_{a3}, from) = \{me_{p2}\}$, $Mult(me_{a3}, from) = 1$,
  $ElemRef(me_{a3}, to) = \{me_{t2}\}$, $Mult(me_{a3}, to) = 1$;
- $ElemRef(me_{a4}, from) = \{me_{t2}\}$, $Mult(me_{a4}, from) = 1$,
  $ElemRef(me_{a4}, to) = \{me_{p1}\}$, $Mult(me_{a4}, to) = 1$;

The relation between the metamodel and model levels of abstraction is illustrated in Fig. 4.4. The model and metamodel formalised previously are related with each other as depicted.

4.4 Summary

This chapter provided a formal definition of a model and metamodel. It presented a series of definitions that allows to be very precise in what concerns metamodel and model elements. At the end of the chapter we introduced the relation of conformity between a model and a metamodel. The $conformsTo$ relation will be used several times in the chapters that follows allowing to describe a model in relation to its metamodel. This relation definition creates a 'typing' relationship between models and the metamodel that is in its origin.

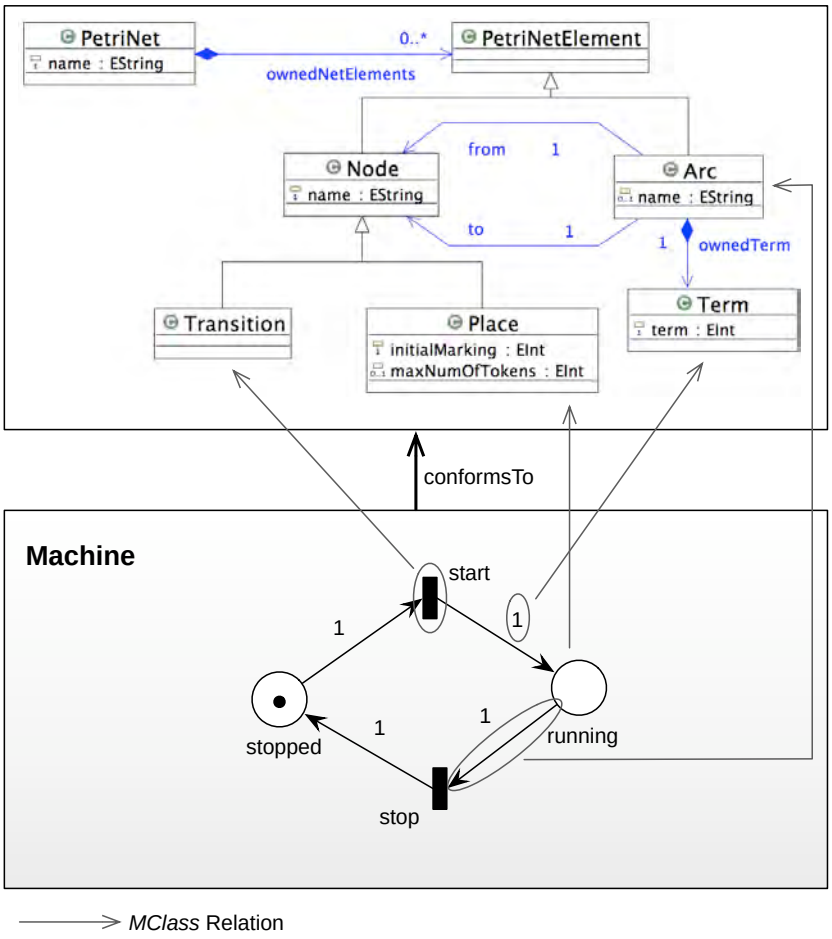


Figure 4.4. Conformity Relation between Petri Nets Model and Metamodel

Chapter 5

Tools for Domain Syntactic and Semantic Composition

In this chapter we will go through the general process of providing semantics to domain concepts from the point of view of the tools used. Although the methodology described in this document is general enough to be used with several target languages and several transformation languages, in this chapter we focus in a specific target and transformation language. We describe Concurrent Object Oriented Petri Nets (CO-OPN) language as target in order to give semantics to the domain concepts and in ATLAS Transformation Language (ATL) as transformation language as a mean to achieve transformations.

The goal of this chapter is to provide background on CO-OPN and ATL so that examples in the following chapters can be easily assimilated.

By using CO-OPN as the target formalism and ATL as the transformation language and engine we can provide semantics as the composition of small blocks that will define a prototype for simulation and validation of DSMLs.

5.1 Transformation of Domain Concepts

As previously mentioned, the semantics of a domain concept is given by transformation. The approach proposed can use several target languages in order to provide the desired semantics to the domain concept (details of the methodology are presented in Sec. 6.1. Albeit this fact we will mostly concentrate on transformations that have as domain the CO-OPN language.

CO-OPN is an object-oriented specification formalism based on Petri Nets and algebraic specifications. It is suitable to fully support specifications of complex concurrent and distributed systems. Modelling with CO-OPN pro-

vides the advantage of dealing with specifications that can be executed from an unambiguous representation of the system. The CO-OPN language and COOPBuilder IDE have been provided with MDA concepts and functionalities that allow to integrate with solutions for the prototyping of Domain Specific Languages.

For transforming from a domain concept to CO-OPN (or to any other target formalism) we used the ATLAS Transformation Language (ATL). Using ATL was just a question of choice and an other transformation languages could have been used. ATL has been chosen because of its features in the field of Model-Driven Engineering (MDE). It provides ways to produce a set of target models from a set of source models.

In order to transform models we have to have a metamodel defined for each one of the domain concept. The metamodels we use are defined using the ECore formalism. The ECore formalism is supported by ATL and we defined a CO-OPN metamodel completely based on ECore so that we could use it as target formalism in the transformations.

For being able to use all these concepts we have integrated our work in the Eclipse Modeling Framework (EMF). As the name suggests the EMF framework is integrated in the Eclipse platform. It allows to use several features of the code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. Model Query, Model Transaction, and Validation Framework, are also available.

5.2 CO-OPN as Target Formalism

Concurrent Object Oriented Petri Nets (CO-OPN) (Buchs & Guelfi, 1991, 2000; Biberstein, Buchs, & Guelfi, 1997; Biberstein, 1997) is a formal specification language allowing symbolic execution and state space exploration. It consists in combining Petri nets and object oriented data structures to provide a framework for the design and specification of concurrent and distributed systems.

Under some restrictions, the CO-OPN existing utilities (A. Chen, Buchs, Lucio, Pedro, & Risoldi, 2006) allow to:

a) generate a prototype for a given specification;

b) enrich it with user specific code;

c) execute it or perform verification of implementations through testing;

d) high level of modularisation of both CO-OPN specifications and generated code used for prototyping.

In this sense CO-OPN is both the intermediary semantic format for a given domain concept allowing code generation for validation and verification purposes.

There are various reasons why CO-OPN is suitable to be chosen as the target format for domain concepts transformation. Some of the more relevant are:

- It is modular specification language allowing to specify different DSL components and their relationships;
- The specifications are described in a completely abstract axiomatized fashion;
- The system states can be completely defined and explored;

CO-OPN specifications are made by three types of modules: *Algebraic Abstract Data Types (ADTs)*, *Classes*, and *Contexts*:

- *Algebraic Abstract Data Type (ADT)* modules represent data and their associated operations;
- *Class* modules are an encapsulation of algebraic Petri nets that allows to describe both structure and component behaviour;
- *Context* modules are a higher level of encapsulation defining the contextual coordination between components.

All three types of modules have the same basic structure. They are composed of three parts: a header, an interface and a body. Fig. 5.1 shows an excerpt of the CO-OPN metamodel focused on the basic CO-OPN modules.

- The header section contains information about inheritance and genericity.
- The interface section contains components of the modules that are accessible by other modules.
- The Body section contains information about the behavior and the state of the module.

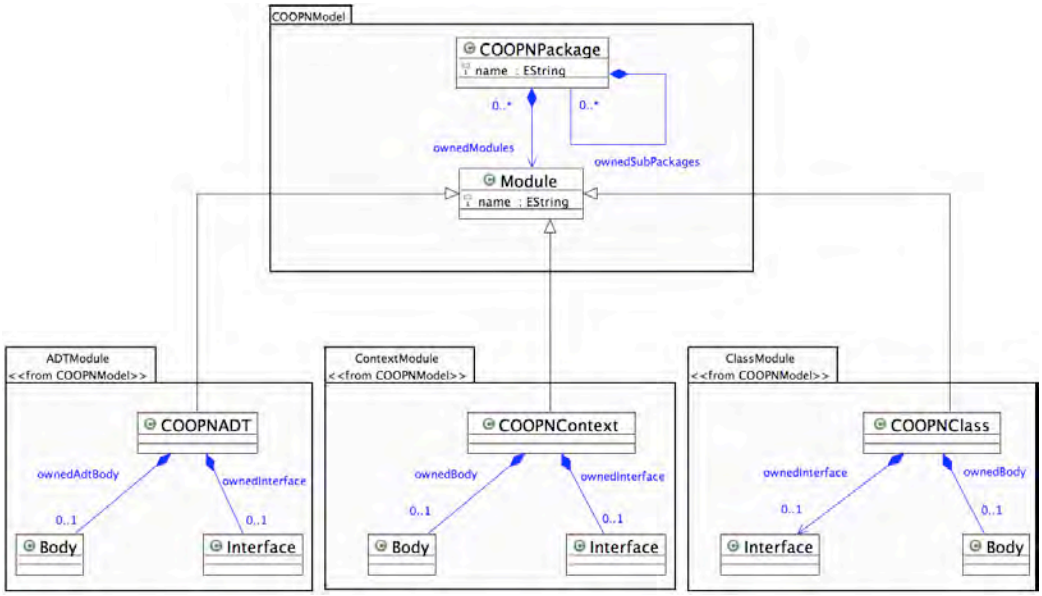


Figure 5.1. CO-OPN metamodel: focus on CO-OPN modules

Figure 5.1 presents part of the CO-OPN that describes *Package*. A *Package* can be defined as a set of related modelling elements, grouped together to share the same namespace. The structure of the CO-OPN package is similar to the ones known in other object oriented modelling languages, except that a `COOPNPackage` is the outermost element in CO-OPN. In other words, this means that any modelling component in CO-OPN must be in a *Package*.

CO-OPN element is the root element of the metamodel, and represents *CO-OPN Metamodel*. A metamodel root element is required by the *Ecore* metamodelling language, on which our transformation framework is based.

COOPNPackage represents the CO-OPN element called *package*. A `COOPNPackage` contains none or several `subPackages` (see `ownedSubPackages` composition). A *Package* can be defined as a set of related modelling elements, grouped together to share the same namespace. The structure of the CO-OPN package is similar to the ones known in other object oriented modelling languages. `COOPNPackage` is the outermost modelling element of the CO-OPN metamodel and contains none or several modelling elements called `Module` (see `ownedModules` composition).

Module is an abstract element of the CO-OPN and represents the three different type of modules, Algebraic Abstract Data Type (ADT), Classes

and Contexts.

The following subsection describes each one of the CO-OPN modules. Fragments of each one of the modules' metamodel will be used in order to provide essential information regarding CO-OPN. This information will be later needed in order to understand the transformations defined from the domain concepts to CO-OPN .

5.2.1 Algebraic Abstract Data Type Module

The Algebraic Abstract Data Type (ADT) module describes data types that can be used by other CO-OPN ADT or Class modules. The header section of an ADT defines inheritance features. The `Interface` section includes, among other constructions, `Sorts`, `Generators` and `Operations`. The `Body` section regroups the definition of `axioms`: axioms define the ADT semantics by defining the behaviour of operations and generators.

Firstly we will show the acADT `Interface` in more detail. Fig. 5.2 shows part of the CO-OPN metamodel that define the ADT focusing on the interface section of it.

Interface The interface section specifies module components that are going to be accessible by other modules. An ADT interface aggregates none or several `Use`, `Operation`, `Sort` and `Generator`;

Use represents a reference to another CO-OPN module. It is represented by the `Use EClass` element. Not shown in the ADT metamodel excerpt for readability reasons there the `usedModuleForAdt` association that allows to identify which module is referenced. This association is the `opposite` relation for the `ownedInterfaceUses` composition;

Sort specifies a set of sorts that identify one of the ADT signature's part. A `Sort` is identified by `sortName` attribute represents the type name;

Operation define the operations that can be performed on the data type under specification. An `Operation` is characterised by a `CoDomain` and none or several `Domain`. The `name` attribute represents the unique name of the operation in the ADT;

Codomain is the definition of the operation result type. It is characterised represented by an attribute `name`. Since `Codomain` is an ADT type it also has a mandatory reference (`codomainSort` in the metamodel) to `Sort EClass` elemtent;

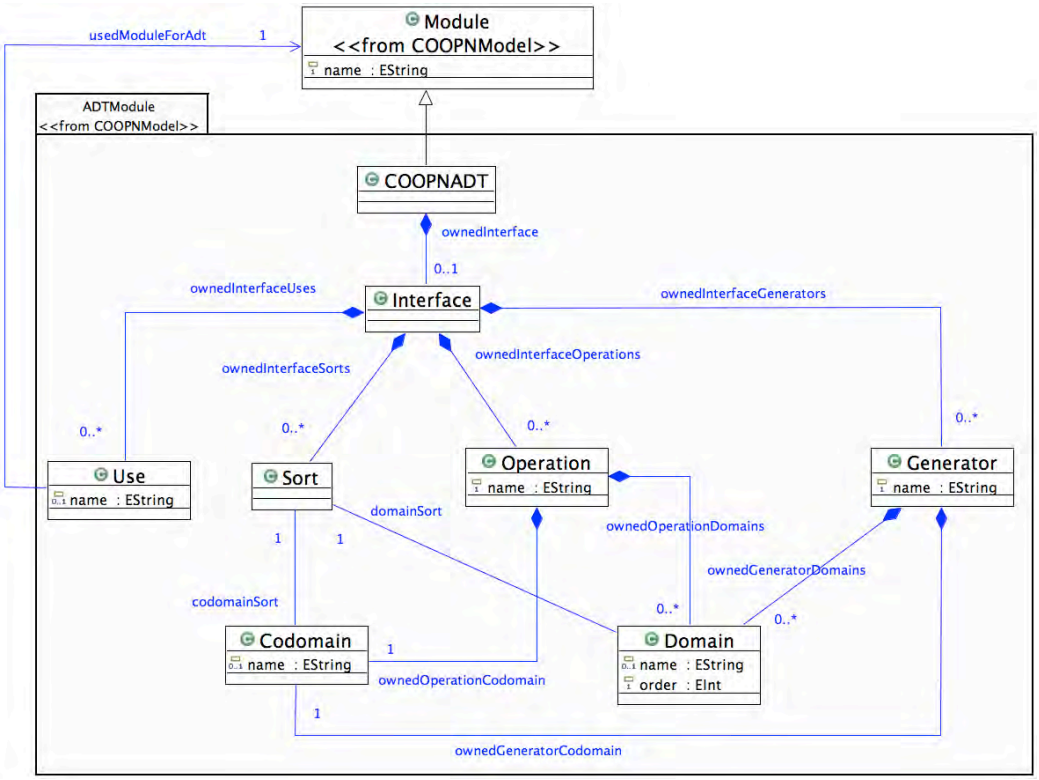


Figure 5.2. CO-OPN metamodel: focus on ADT Interface

Domain defines the type of each operation parameter. It is described by the `name` and `order` attributes that represent, respectively, the name of the operation parameter type, and the order in which will figure in the operation signature. Similarly as the `Codomain` each domain provides a mandatory reference to the ADT type (`domainSort` association in the metamodel);

Generator is the function that builds all the possible values of the ADT's sort. It can be seen as a constructor for the data type being specified. The `name` attribute contains the syntactic expression of the generator function. A `Generator` is also defined by one `Codomain` and none or several `Domains`.

The interface section of an ADT basically specifies the ADT's signature. Its semantics is defined in the `Body` section. Fig. 5.3 depicts a subset of the ADT `Body` metamodel section focused on the definition of the ADT axioms.

Axiom is the root element for defining the ADT semantics. The `Axiom` meta-class is a specialisation of the `AxiomTheorem` meta-class meaning

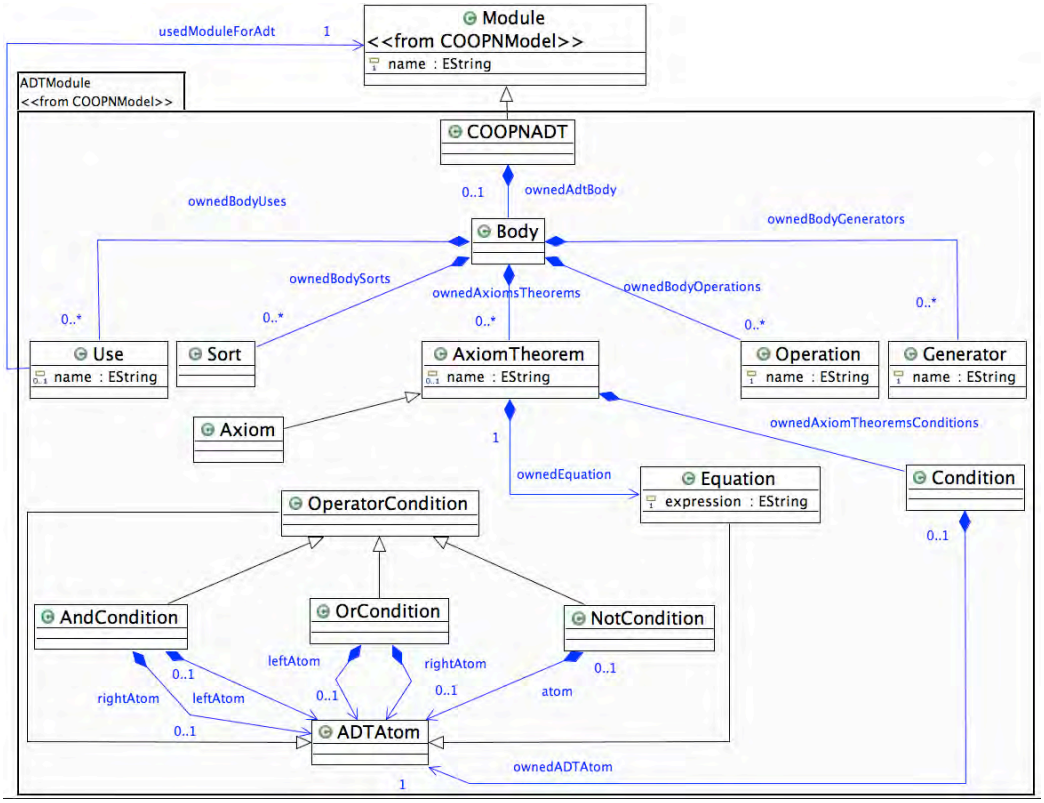


Figure 5.3. CO-OPN metamodel: focus on ADT Body and Axioms Definition

that, axioms and theorems have the same abstract syntax in CO-OPN ADTs. An axiom is identified by a name represented in the meta-model as the `name` attribute `AxiomTheorem` meta-class. Both `Axiom` and `Theorem` have the same syntactic structure. In the CO-OPN meta-model they are a specialisation of the `AxiomTheorem` meta-class;

Sort specifies set of sorts can be defined for ADT's 'internal' use. A `Sort` is identified by `sortName` attribute represents the type name;

Use Similar to the meaning of the `Use` meta-class in the `Interface` section, represents a reference to another CO-OPN module;

Operation Like in the `Interface` section of the ADT definition defines the operations that can be performed on the data type. Operations defined in the `Body` section are to be used only within the ADT under specification;

Generator Again, it is the function that builds all the possible values of the ADT's sort. They are of private use to the `ADT`;

Equation represents the set of equations allowing to specify the semantics of an ADT axiom. It is defined by an `expression` allowing to express the equation's statement;

Condition allows to express a boolean condition that is evaluated. The specification of a condition in the ADT axiom implies that axiom can only be executed if the condition is evaluated to *true*. The `AndCondition`, `OrCondition` and `NotCondition` meta-classes are a specialisation of the `OperatorCondition` meta-class which, on its turn, is specialises the `ADTAtom` meta-class. All this setup of meta-classes that specialise and are specialised define a recursive structure that allow to specify conditions of the type $(var1 = var2) = true \& (var3 > 0) = true$.

ADTAtom Defined as the class that specialises `OperatorCondition` and `Equation`, it allows to specify the leaf of the axiom's definition.

As an example of the definition of an ADT using CO-OPN language please see Appendix B.1. In this appendix we use both CO-OPN textual concrete syntax and the syntax based on ECore in order to show both ways of specifying an ADT.

5.2.2 Class Module

In CO-OPN Classes act as object templates. Objects are net instances of these classes. An object is an independent entity, composed of an internal state (defined by the collection of all places and transitions in the class). The internal state of the object is encapsulated, and the only way to interact with an object from the outside is to ask one of its services (also called `Methods`). Interaction between objects is realized by using synchronization expressions, through methods and gates or by an internal transition.

The header section of a Class module contains information about inheritance. The interface section contains module's components that are accessible by other modules. The Fig. 5.4 highlights some of the components that define a CO-OPN `Class` module. This figure focuses on the `Interface` part of it.

Module is abstract and represents the three different type of modules, ADTs, Classes and Contexts. `name` attribute represents the name of the module.

COOPNClass represents the CO-OPN `Class`. A Class is composed by an `Interface` and a `Body`. `Interface` is defined here by the `ownedInterface` composition;

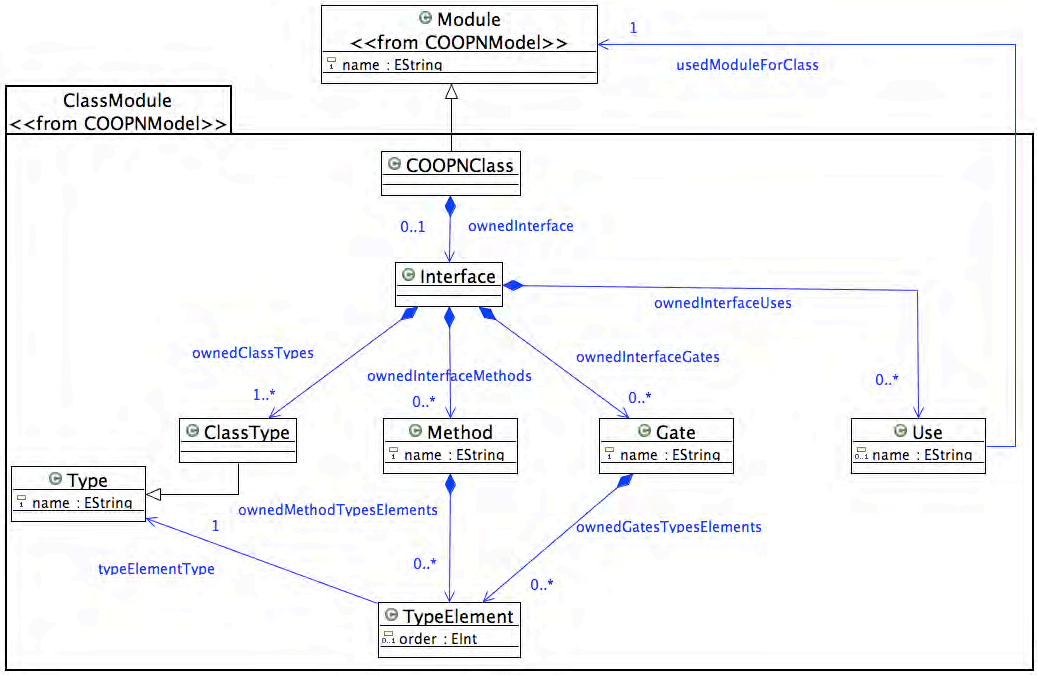


Figure 5.4. CO-OPN metamodel: focus on Class Interface

Interface A Class interface contains none or several `Use`, `Gate` and `Method` represent respectively by the `ownedInterfaceUses`, `ownedInterfaceGates` and `ownedInterfaceMethods` respectively. It is also composed by one ore several `ClassType`;

ClassType represents the type of the Class. `name` attribute in inherited from the `Type` meta-class and identifies the class type;

Use it allows to create a reference to another module. This reference is represented by the `Use` modelling element by the `usedModuleForClass` association. This association points to the `Module` meta-class that can be either a CO-OPN `Class` or `ADT`;

Method are the services that the class provides to the outside. The `name` serves as an unique identification of it. A CO-OPN `Class` method is composed of none or several typed elements. This defines the signature of a method in the class. The `TypeElement` meta-class is related with the `Method` meta-class by the `ownedMethodTypeElements` composition. Creating one instance of the `TypeElement` EClass is the equivalent of creating a method parameter in Java. In `CO-OPN` a method parameter is designated by an _ that can be present in any place of the method's

signature. As an example, the method *sum*(_ , _) can also be written as _ *sum* _ . The number of elements of type `TypeElement` must be equal to the number of _ in the method's `name`. The `order` attribute is an unique identifier that allows to specify the expected type of each method's attribute;

Gate is a specific transition which represent an outgoing event, the required service. Gates and methods are complementary and have the same syntactical structure;

One other section of a CO-OPN Class is its body. The most important characteristic of the CO-OPN Class body section is that groups the underlying petri-nets and axioms' definition. In other words the semantics of objects are defined in this part of the CO-OPN specification, while providing the necessary information to their templates. In Fig. 5.5 we present a simplified metamodel of the body section of a CO-OPN Class.

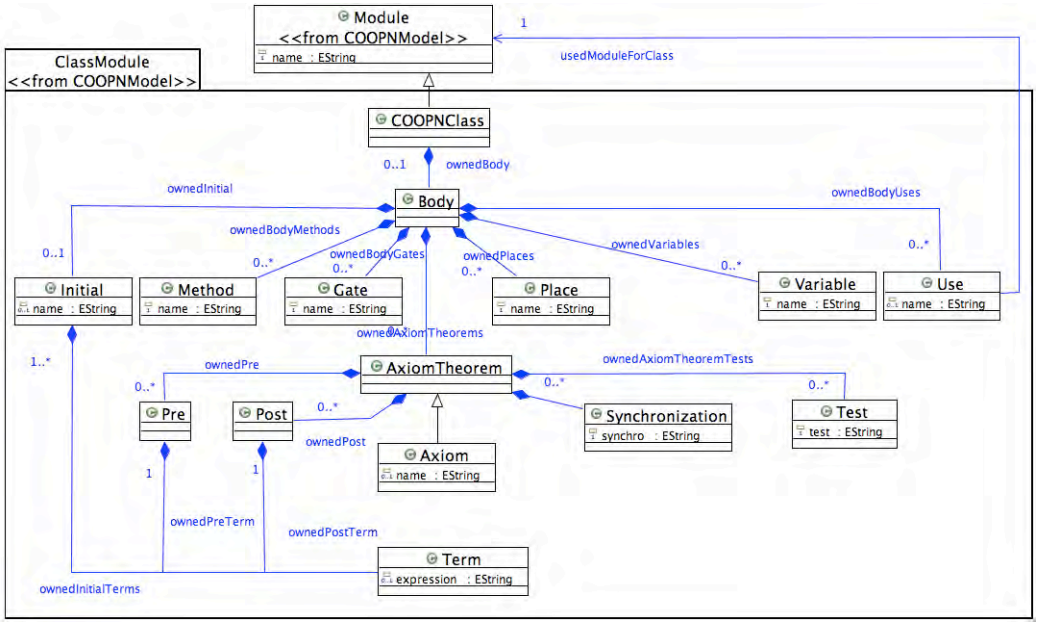


Figure 5.5. CO-OPN metamodel: focus on Class Body ***without*** *Axiom definitions*

Method A method in the CO-OPN body section represents a private service for the objects instances of the `ClassType` defined in the interface section. It has the same overall structure as a method in the interface;

Gate is a private required service. Syntactically it is defined similarly to the gate in the interface section;

Place represents a state (as in the Petri Net formalism). The collection of all places represents the state of a class instance. A place is identified by an unique name. The `name` attribute of the `Place` meta-class in the metamodel. A variable is typed by an association `placeType` to the meta-class `Type` (not shown on the figure). The type is either of type `ClassType` or `Sort`;

Use Similar to the `Use` definition in the interface section;

Variable A variable is an internal 'attribute'. It is used to parameterize axioms, methods and gates. As for a place, a variable is typed by an association `variableType` to the meta-class `Type`;

Initial The `Initial` element of a CO-OPN class is used to initialise the defined variables and places. If `Initial` is defined for a place or variable in a CO-OPN class, the objects of that class will have their places and variables initialised to the values defined in the `Initial` field;

Axiom As for ADTs, a class body is also composed by axioms and theorems. Again, they define the behaviour of the class. An axiom is composed of none or several pre conditions, post conditions, synchronisations, and tests. The `Pre`, `Post`, `Synchronization` and `Test` meta-classes defines these entities in the metamodel. An axiom is also composed by zero or one condition and one event. A detailed description of these will be provided while presenting some details of the class metamodel in Fig. 5.6;

Pre The pre-condition that must be satisfied by a place when an axiom is evaluated;

Post The post-condition that must be satisfied by a place when an axiom is evaluated;

Test Test is a part of the axiom that must be evaluated to true in order for the axiom to be executed. Tests act on variables or on services provided by objects.

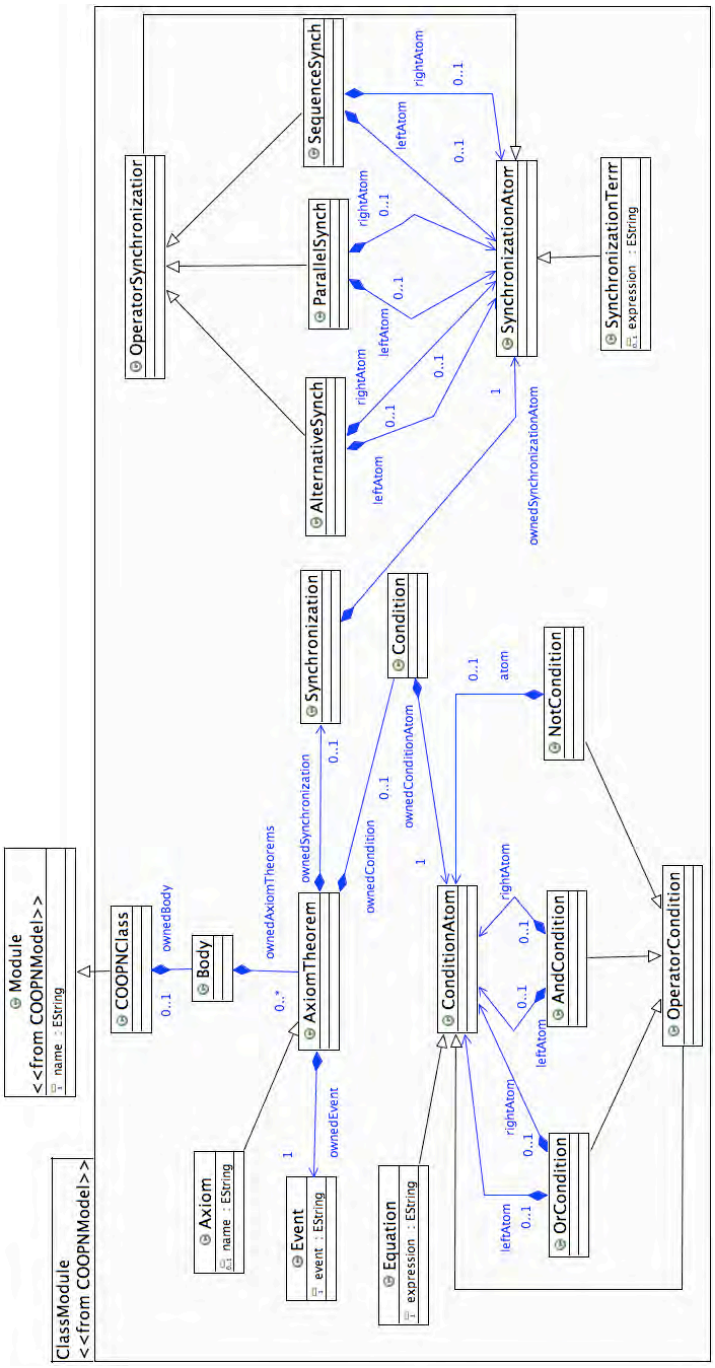


Figure 5.6. CO-OPN metamodel: focus on Class Body Axiom definition

Having described the generic aspects of CO-OPN body structure we proceed by providing a more detailed characterisation of some aspects of it. The Fig. 5.6 focus on the conditions and events components of a class axiom.

Condition An axiom can have zero or one condition represented in the figure by the `Condition` meta-class. The condition is the way of expressing in which circumstances (state of the system) the axiom can be triggered. A condition is a composition of one condition atom (`ConditionAtom` meta-class in the metamodel). This meta-class is specialised by the `Equation` and `OperatorCondition` meta-classes. The `OperatorCondition` is specialised by the `OrCondition`, `AndCondition` or `NotCondition` meta-classes that, themselves are composed by other condition atoms. This allows to have conditions as a composition of statements like $(a > b) = true\ and\ (a > 0) = true$.

Event is the name of the service that will trigger the axiom evaluation and, in case of evaluation to *true*, its execution. An event can be either a gate or a method defined in the class. The attribute `name` in the `Event` class defines which method or gate it corresponds to;

Synchronization In CO-OPN the syncrhonisation part of an axiom defines what services are going to be executed if the axiom pre-conditions, post-conditions and condition are evaluated to *true*. These services can be methods or gates defined both in the class or defined in other classes. A synchronisation is a composition of a synchronisation atom (`SynchronizationAtom` meta-class in the metamodel) that is specialised by the `SynchronisationTerm` and `OperatorSynchronization` meta-classes. This last meta-class is specialised by the `AlternativeSynch`, `ParallelSynch` and `SequenceSynch` meta-classes which. These meta-classes are a composition of `SynchronizationAtom` elements and they represent the types synchronisation available in CO-OPN:

- Alternative is represented as + in CO-OPN textual concrete syntax. It allows to express service calls in a non-deterministic fashion. E.g. *o.methodA + o.methodB* means that either *o.methodA* or *o.methodB* is going to be executed;
- Parallel is represented as // in CO-OPN textual concrete syntax. It allows to call services in parallel. The services expressed with this operator will be executed simoultaneously. E.g. *o.methodA // o.methodB* means that *methodA* and *methodB* of object *o* are executed in parallel;

- Sequence is represented by .. CO-OPN textual concrete syntax. This operator allows to express services call in sequence, i.e. one after the other. The *o.methodA .. o.methodB* expression means that *methodA* is called and after which *methodB* is called.

CO-OPN contexts are units of computation where included entities are coordinated following the same synchronization rules used for class methods and gates. CO-OPN components are organized hierarchically: contexts can contain sub-contexts and objects; objects can have places and transitions; places can contains tokens. Furthermore, if we ignore the transactional aspect of CO-OPN transitions, CO-OPN models with different types of tokens correspond to different kinds of Petri nets.

Contexts are coordination entities. They are the responsible to coordinate activities among classes. Coordination is done by mean s of behavioral axioms that connect methods and gates of the embedded entities.

5.2.3 Context Module

As the previous two CO-OPN modules, contexts can be separated in interface and body sections. The metamodel's interface section is presented in Fig. 5.7. This part of a CO-OPN context is composed of:

Context Use is a pointer to another CO-OPN context. It allows the 'inclusion' of contexts into other contexts so that services defined by them can be used;

Method A method is a public service that allows coordination with other context services and services available through objects;

Gate is a public required service that will coordinate with other services either defined in other contexts or made available through objects;

Use it allows to create a reference to another CO-OPN module. Similar to the `Use` field in a CO-OPN class or ADT.

In what concerns the body section, a CO-OPN context is organised as follows:

Use is the reference to other CO-OPN modules used in the body section;

Method is the private service that allows coordination with other context services and services available through objects;

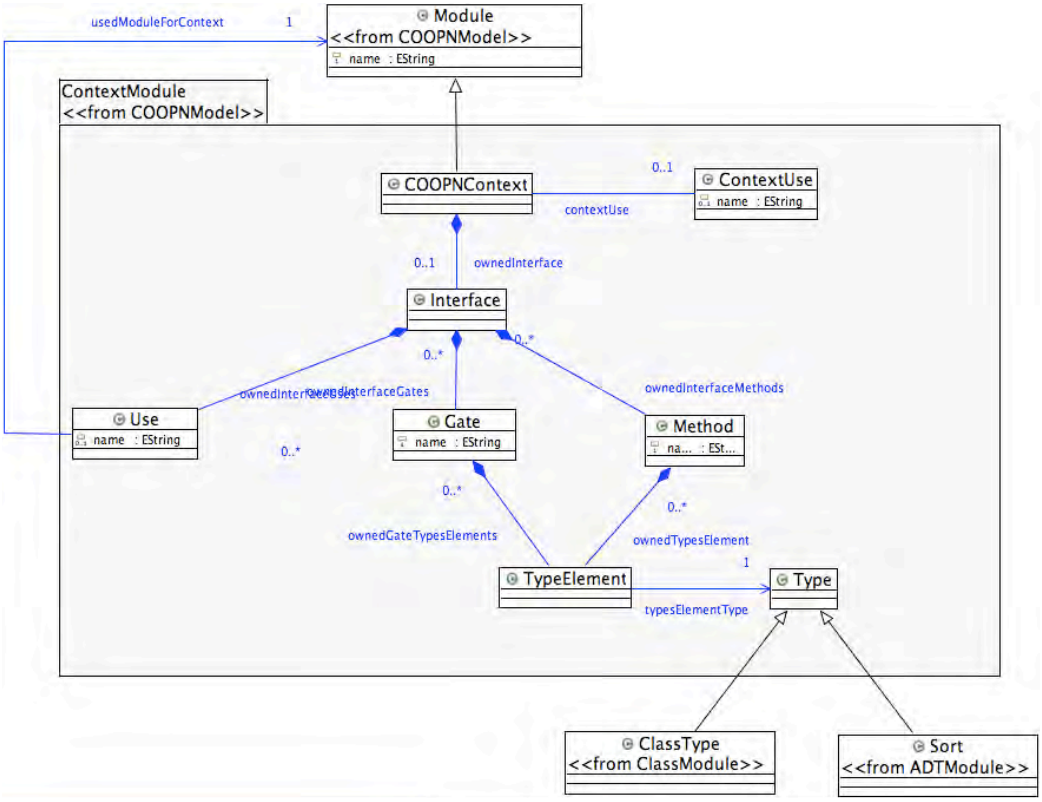


Figure 5.7. CO-OPN metamodel: focus on Context Interface

Gate is a private required service that will coordinate with other services either defined in other contexts or made available through objects;

AxiomTheorem the way of providing the semantics within a context. An axiom or theorem are composed by a condition, a required and a provided event. In the metamodel, these constructions are presented as `Condition`, `RequiredEvent` and `ProvidedEvent` meta-classes. The required event is what triggers the axiom's execution. It is either a CO-OPN method or gate. The condition defines in which circumstances the axiom is going to be executed. A condition is a composition of equations possible separated by an operator. Operators can by the logical *or*, *and* or *not*.

The provided event section represents what operations have to be executed in case the axiom's condition is evaluated to *true*. This part of the CO-OPN language is a composition of events by using the synchronisation operators available. Its usage and semantics is similar to the one presented for CO-OPN classes in Sec. 5.2.2 on page 61. Again,

the provided event is built with the sequential, parallel and alternative synchronisation operators.

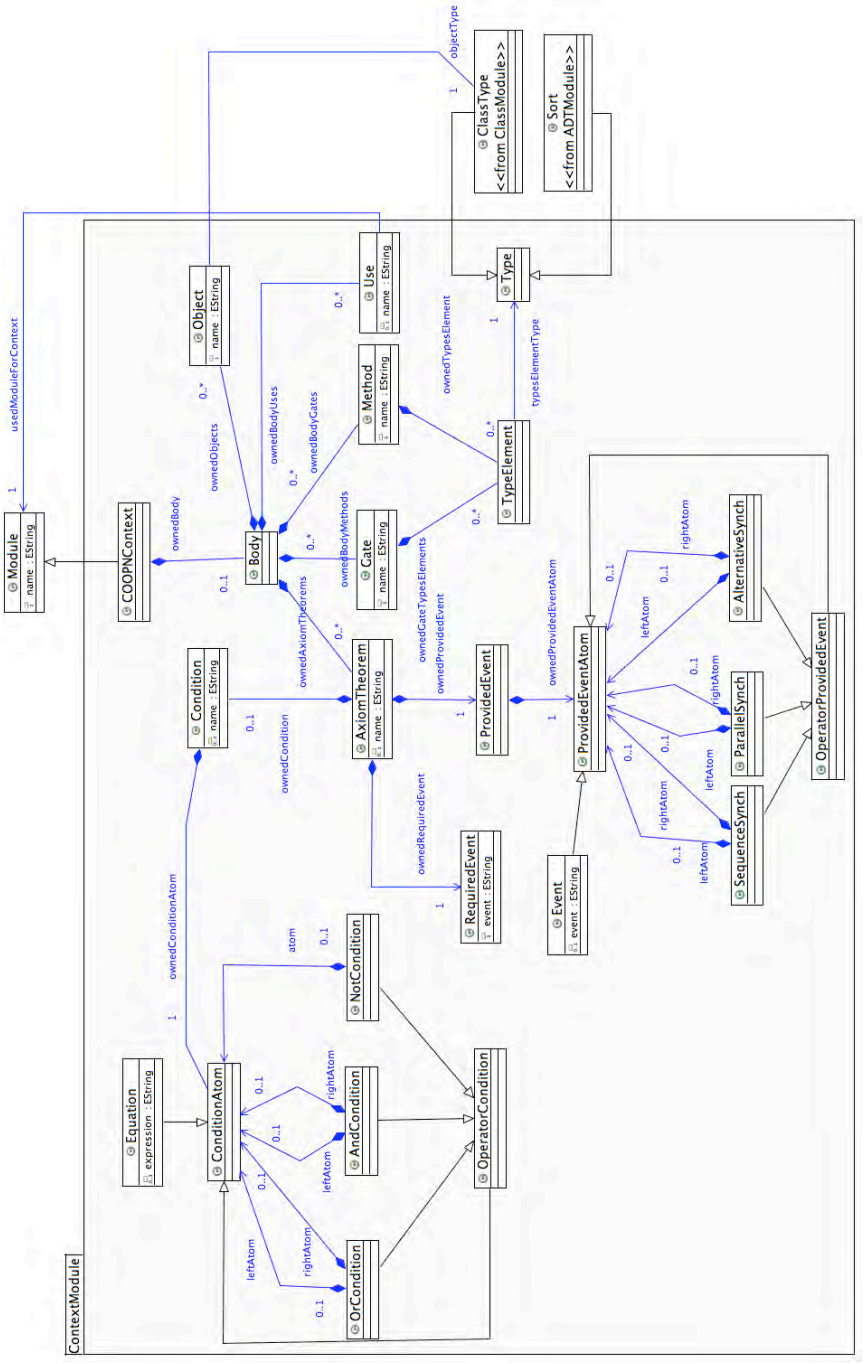


Figure 5.8. CO-OPN metamodel: focus on Context Body

Sec. B.3 shows a CO-OPN context using the different concrete systaxes available.

5.2.4 Sub-typing and Inheritance in CO-OPN

Sub-typing and inheritance are two different notions in CO-OPN. An inheritance relationship is a syntactic mechanism which allows to reuse parts of existing specifications to create a new specification. Sub-typing relationship is based on the strong substitutability principle, which implies that, in any context, any class instance of a type may be substituted for a class instance of its super-type while the behavior of the whole system remains unchanged (Buchs & Guelfi, 2000). Two classes related by inheritance are not necessarily related by sub-typing. CO-OPN offers a much less restrictive mechanism than those provided by other object oriented formalisms, such like UML, MOF or Ecore. A complete illustration of the distinction between inheritance and sub-typing in the CO-OPN language is illustrated in (Buchs & Guelfi, 2000).

Fig. 5.9 presents part of the CO-OPN metamodel that defines the `Module` inheritance and sub-typing mechanism.

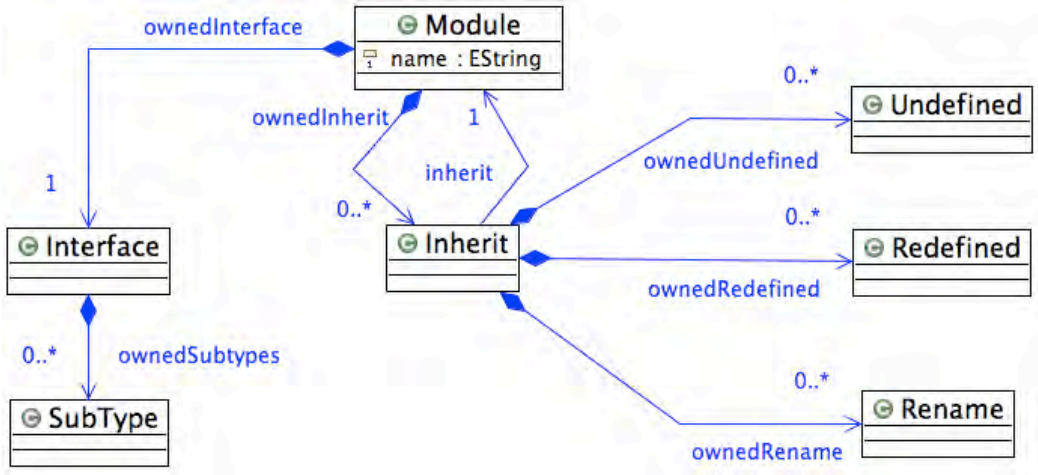


Figure 5.9. CO-OPN metamodel: focus on Module Inheritance

Module is abstract and, as described in the three previous sections, represents the three different types of modules, ADTs, Classes and Contexts. A `Module` contains none or several modelling elements `Inherit` that allow to express inheritance relations;

Inherit represents an inheritance. Each inheritance point to a CO-OPN module by means of the `inherit` EReference. The `Inherit` meta-class is composed none or several `Rename`, `Redefined` and `Undefined`.

Rename element represents the action of renaming a inherited module property into another. Properties that can be renamed depends on the type of module;

Redefined element represents the action of redefining a inherited module property. Inherited properties are ignored and replaced by the new properties definition. Properties that can be redefined depends on the type of module;

Undefined element represents the action of eliminating a inherited module property. Properties that can be eliminated depends on the type of module;

SubType represents the action of sub-typing a module. The structure of a `SubType` depends on the type of module.

Listing 5.1. Module Inheritance abstract syntax

```
ModuleType moduleName
Inherit inheritedModuleName
Rename
sourcePropertyName -> targetPropertyName ;
Redefine
sourcePropertyName -> targetPropertyName ;
Undefine
sourcePropertyName -> targetPropertyName ;
Interface
...
Body
...
End moduleName
```

The structure of `Module` components involved in the sub-typing and inheritance mechanism depends on the type of `Module`. For instance `Rename`, `Redefined`, `Undefined` and `SubType` meta-classes. In particular for the case of a CO-OPN class the `Inherit` meta-class is composed of none or several `Rename` associated to the abstract meta-class `RenamableProperty`. It represents *Class* module components which attribute `name` can be modified in the renaming process. Properties that can be renamed are the `ClassType`, `Method` and `Gate`, which all extends the abstract `RenamableProperty`. A new name can be assigned to each one of the properties that are being renamed.

The structure of the redefined and undefined processes are similar to this one.

Class sub-typing mechanism involves essentially the `ClassType` meta-class. A CO-OPN class interface contains none or several `SubType` that represent the sub-typing. A `SubType` is composed by a `SuperTypeElement` (which represents the super-type), and a `SubTypeElement` (which representes the sub-type). Both elements have a `Type` that is a `ClassType`. Note that the structure is exactly the same for the ADT sub-typing mechanism except that `ClassType` gives its place to an ADT `Sort`.

5.3 ATL as Transformation Language and Engine

In this section we will provide an overview of the ATLAS Transformation Language (ATL). For the context of this work it is important to highlight the availability and usability of ATL. The implementation of the methodology presented relies on ATL in what concerns specification and execution of transformations.

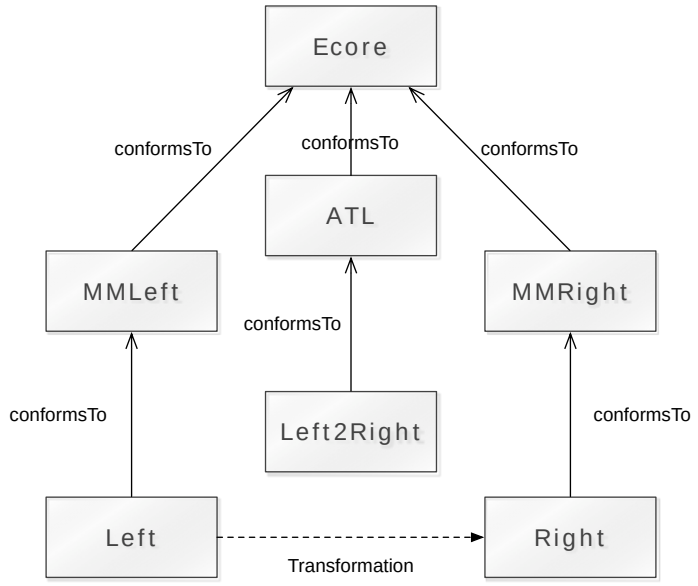


Figure 5.10. An overview of ATL Model Transformation

ATL(ATLAS Group, 2006) is a model transformation language and toolkit developed by the ATLAS Group and integrated in the Eclipse Model-to-Model (M2M) project. ATL is an answer to the OMG MOF (Object Management Group, 2007c) and QVT RFP (Object Management Group, 2002b). It

is a model transformation language specified as both metamodel and textual concrete syntax with an hybrid of declarative and imperative programming approach. ATL mainly focuses on model to model transformations which are specified by means of ATL modules. ATL also allows to create model to primitive data type programs. These units are denominated queries and aim to compute a primitive value, such as a string or an integer.

Fig. 5.10 summarizes the full model transformation process using ATL. A model `Left`, conforming to a metamodel `MMLeft`, is here transformed into a model `Right` that conforms to a metamodel `MMRight`. The transformation is defined by the model transformation model `Left2Right` which itself conforms to a model transformation `ATL` metamodel. The set of `ATL`, `MMLeft` and `MMRight` conform to the metametamodel that in this case is Ecore. Sec. 5.3.3 presents an example that shows how an ATL transformation is defined according to the models and metamodels involved. Fig. 5.12 shows an 'instance' of Fig. 5.10 taking into account the example of Petri Nets transformation.

5.3.1 Structure of ATL Module

The structure of an ATL module is briefly defined as (ATLAS Group, 2006):

Header Section

A header section that defines some attributes that are relative to the transformation module. It defines the name of the transformation module and the name of the variables corresponding to the source and target models

Listing 5.2. ATL Header Definition

```
module module_name;
create output_models [from|refines] input_models;
```

Listing 5.2 shows the generic definition of the ATL module where:

- `module` is the name of the module;
- `create` specifies the target models declaration;
- the source models are defined either by the keyword `from` (in normal mode) or `refines` (in case of refining transformation). It is possible to declare several input or output models by separating the declared models by a coma. The name of the declared models is used to identity them.

Import Section

An optional import section allows to import some existing ATL libraries. The declaration of an ATL library is defined as shown in Listing 5.3.

Listing 5.3. ATL Import Definition

```
uses extensionless_library_file_name;
```

Helpers

A set of helpers that can be viewed as an ATL equivalent to Java methods; An ATL helper is defined by:

- a `name` corresponding to the method's name;
- a `context type` that defines the context in which it is defined. This is analogous to the way a method is defined in the context of given class in Object-Oriented (OO) programming.

  It defines the kind of elements the helper applies to, the type of the elements from which it will be possible to invoke it. The context may be omitted in a helper definition. In such a case, the helper is associated with the global context of the ATL module, which means that, in the scope of such a helper, the variable self refers to the run module/query itself.

  Besides helpers, the ATL language makes it possible to define attributes. When compared to a helper, an attribute can be viewed as a constant that is specified within a specific context. The major difference between a helper and an attribute definition is that the attribute accepts no parameter;
- a `return` value type. In ATL, each helper must have a return value;
- an ATL expression representing the code of the helper;
- an optional set of `parameters`. A parameter is identified by a couple (parameter name, parameter type).

The scheme of an ATL helper is provided in Listing 5.4.

Listing 5.4. ATL Helper Definition

```
helper [context context_type]? def:helper_name(parameters):return_type=exp;
```

Rules

A set of rules that defines the way target models are generated from source ones. In ATL, there exist two different kinds of rules that correspond to the two different programming modes provided: the declarative and imperative programming. The *matched rules* are the one allowing to program in a declarative fashion; and the *called rules* allowing to specify imperative programming.

The matched rules constitute the core of an ATL declarative transformation since they make it possible to specify:

- for which kinds of source elements target elements must be generated;
- the way the generated target elements have to be initialized.

A matched rule is identified by its `name`. It matches a given type of source model element, and generates one or more kinds of target model elements. The rule specifies the logic how generated target model elements must be initialized from each matched source model element.

A matched rule (introduced by the keyword `rule`) is composed by the mandatory source and target patterns. Two optional patterns can be used and represent the local variables and the imperative sections. When defined, the local variable section is introduced by the keyword `using`. It enables to locally declare and initialize a number of local variables;

The source pattern of a matched rule is defined by using the keyword `from`. It allows to specify a model element variable that corresponds to the type of source elements the rule has to match. This type corresponds to an entity of a source metamodel of the transformation.

The target pattern of a matched rule is defined after the keyword `to`. It allows to specify the elements to be generated when the source pattern of the rule is matched, and how these generated elements are initialized. A target pattern element corresponds to a model element variable declaration associated with its corresponding set of initialization bindings. This model element variable declaration has to correspond to an entity of the target metamodels of the transformation.

The optional imperative section, introduced by the keyword `do`, makes it possible to specify some imperative code that will be executed after the initialization of the target elements generated by the rule.

Listing 5.5 provides the schema of an ATL rule.

Listing 5.5. ATL Rule Definition

```
rule rule_name {
2 from
```

```
in_var : in_type [(
condition
)]?
[using {
var1 : var_type1 = init_exp1;
...
varn : var_typen = init_expn;
: = ; }]?
to
out_var : out_type (
bindings1
),
out_var : distinct out_type foreach(e in collection)(
bindings2
),
...
out_varn : out_typen (
bindingsn
)
[do {
statements
}]?
}
```

The called rules provide the imperative programming functionalities. They can be seen as a particular type of helpers: they have to be explicitly called to be executed and they can accept parameters. However, as opposed to helpers, called rules can generate target model elements as matched rules do. A called rule has to be called from an imperative code section, either from a match rule or another called rule.

Since the called rule does not match any source model element, the initialization of the target model elements that are generated by the target pattern has to be based on a combination of local variables, parameters and module attributes. The target pattern of a called rule is defined in the same way the target pattern of a matched rule is.

5.3.2 ATL Data Types

The ATL data type scheme is very similar to the one defined by OCL. Fig. 5.11 provides an overview of the data types structure present in ATL.

The class `OclType` corresponds to the definition of the type instances specified by OCL. It is associated with a specific OCL operation: `allInstances()`. This operation, which accepts no parameter, returns a set containing all the currently existing instances of the type `self`. In ATL there is another additional operation that enables to get all the instances of a given type that belong to a given metamodel. For example, the `allInstancesFrom(metamodel : String)` operation returns a set containing the instances of type self that are defined within the model namely identified by metamodel.

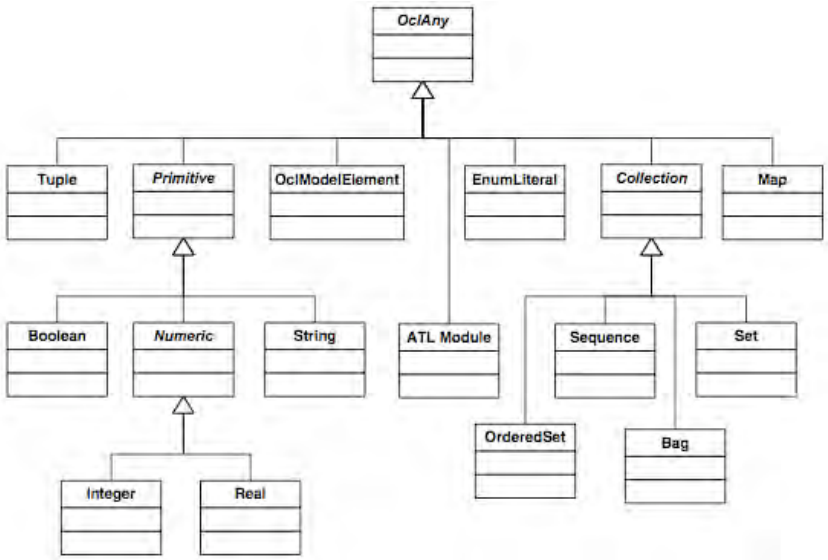


Figure 5.11. ATL Data Types metamodel

The operation defined over the class `OclAny` are the set of operations common to all existing data types and include:

- comparison operators: `=, <>`;
- `oclIsUndefined()` returns a boolean value stating whether self is undefined;
- `oclIsKindOf(t:oclType)` returns a boolean value stating whether self is an either an instance of t or of one of its subtypes;
- `oclIsTypeOf(t:oclType)` returns a boolean value stating whether self is an instance of t.

ATL also implements a number of additional operations:

- `toString()` returning a string representation of self
- `oclType()` returns the oclType of self;
- `asSequence(), asSet(), asBag()` respectively return a sequence, a set or a bag containing self;
- `output(s:String)` writes the string s to the Eclipse console.
- `debug(s:String)` returns the self value and writes the s : self_value string to the Eclipse console;

- `refSetValue(name:String, val:oclAny)` is a reflective operation that enables to set the self feature identified by name to value val. It returns self;
- `refGetValue(name:String)` is a reflective operation that returns the value of the self feature identified by name;
- `refImmediateComposite()` is a reflective operation that returns the immediate composite;
- `refInvokeOperation(opName:String, args:Sequence)` is a reflective operation that enables to invoke the self operation named `opName` with the sequence of parameter contained by `args`.

OCL also defines a set of primitive data types that are implemented in ATL. The primitive datatypes are the types corresponding to the `Boolean`, `Integer`, `Real` and `String` data types. A set of operations are also defined over these types.

In addition to primitive types, ATL also supports collection data types. Namely:

- `Set` is a collection without duplicates. Set has no order;
- `OrderedSet` is a collection without duplicates. OrderedSet is ordered;
- `Bag` is a collection in which duplicates are allowed. Bag has no order;
- `Sequence` is a collection in which duplicates are allowed. Sequence is ordered.

A collection can be seen as a template data type. This means that the declaration of a collection data type has to include the type of the elements that will be contained by the type instances.

ATL provides a large number of operations in the context of the different supported collection types. The description of the supported ATL operations' one collections is out of the scope of this document. A detailed description is provided

5.3.3 ATL Example

Taking the Petri Nets presented in Fig. 4.2 the general path for transforming it to CO-OPN is obtained by defining an ATL transformation that takes as source the Petri Nets metamodel and model, the CO-OPN metamodel and

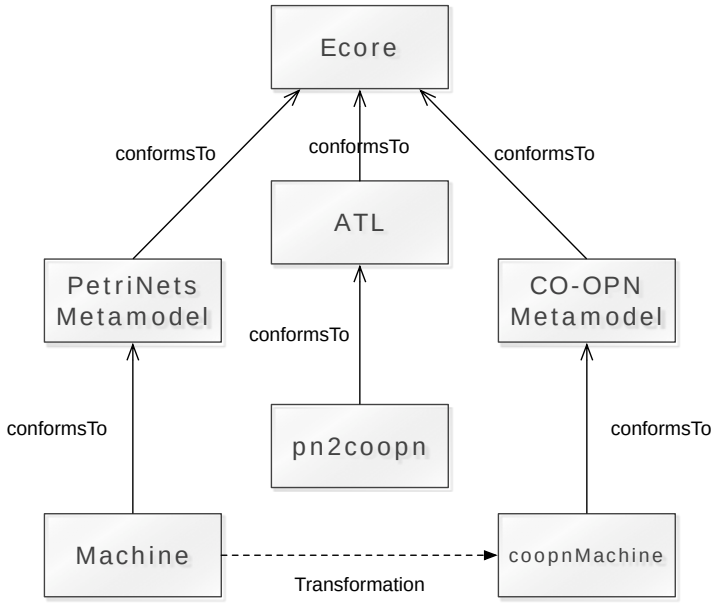


Figure 5.12. An overview of ATL Model Transformation

producing a target CO-OPN model. The Fig. 5.12 is a specialisation of the general schema of an ATL transformation in Fig. 5.10.

In Fig. 5.12 `MMLeft` and `MMRight` are replaced by `PetriNets MetaModel` and `CO-OPN Metamodel` respectively. The `Left` model is replaced by the Petri Nets model `Machine` (in Fig. 4.4) and the `Right` model by the `coopnMachine` CO-OPN model that will be produced by the transformation. `pn2coopn` stands for the ATL transformation specification that we detail in the rest of this subsection.

The equivalent of the `Machine` Petri Nets model in CO-OPN language will be as presented in Listing 5.6.

Listing 5.6. CO-OPN Equivalent of Petri Nets Machine Model

```
Class CLASSMachine;
Interface
Type machineType;
Methods start; stop;

Body
Use BlackTokens;
Places
stopped _ : blackToken;
running _ : blackToken;
Initial
stopped @;
Axioms
start::stopped @ ->running @;
stop::running @ ->stopped @;
End machine;
```

5.4 Summary

In this chapter we went through an overview of the tools used in order to apply the methodology that we are going to describe on Chap. 6. In particular we described CO-OPN from the perspective of a language engineering. In the context of this work we implemented a metamodel for CO-OPN that we have described some parts in this chapter. In Sec. 5.3 we presented the ATL language that we are going to use while applying transformations for the examples presented in Chap. 8 of this document.

Chapter 6

Providing Semantics for Domain Models

The aim of this section is to provide an overall definition of a methodology that allows to easily deal with semantics of DSMLs. Adding semantics to a domain is a two-step operation:

1. Define a metamodel of the domain concept;
2. Specify a transformation that maps the semantics of the domain concepts in a target formalism;

The metamodel is defined by means of a metamodelling formalism whereas transformations are defined in relation to a source and a target metamodel. Moreover transformations take a model in conformity with a source metamodel and produce another model that is in conformity with the metamodel of the target formalism.

Instead of focusing on defining the complete semantics of a DSML by defining a complete transformation to a target plat form, the purpose of this work is to allow modularity of semantic constructions. DSMLs are built using iteractive and incremental processes and so should be their semantics. Taking this into account we propose a generic method that allows modularisation of metamodels, models and associated transformations that represent the semantic mapping.

In this chapter we go into the most formal aspects of the work. It focuses on the fundamentals of the methodology by describing all the concepts involved. A set of formal definitions are provided. We detail how composition of metamodels, models and transformations are specified and instantiated. The result of these compositions provide the basis for the Compositional Platform for Domain Specific Modelling Languages Prototyping (CoPsy) implementation presented later in Chap. 7.

6.1 General Approach

For the context of this work both a methodology and a tool was developed. The methodology is not tied to a particular technology or language. Its formal definition supports this statement and allows to explain the applied philosophy at a high level of abstraction.

Fig. 6.1 provides a high level overview of the methodology developed during this thesis' work.

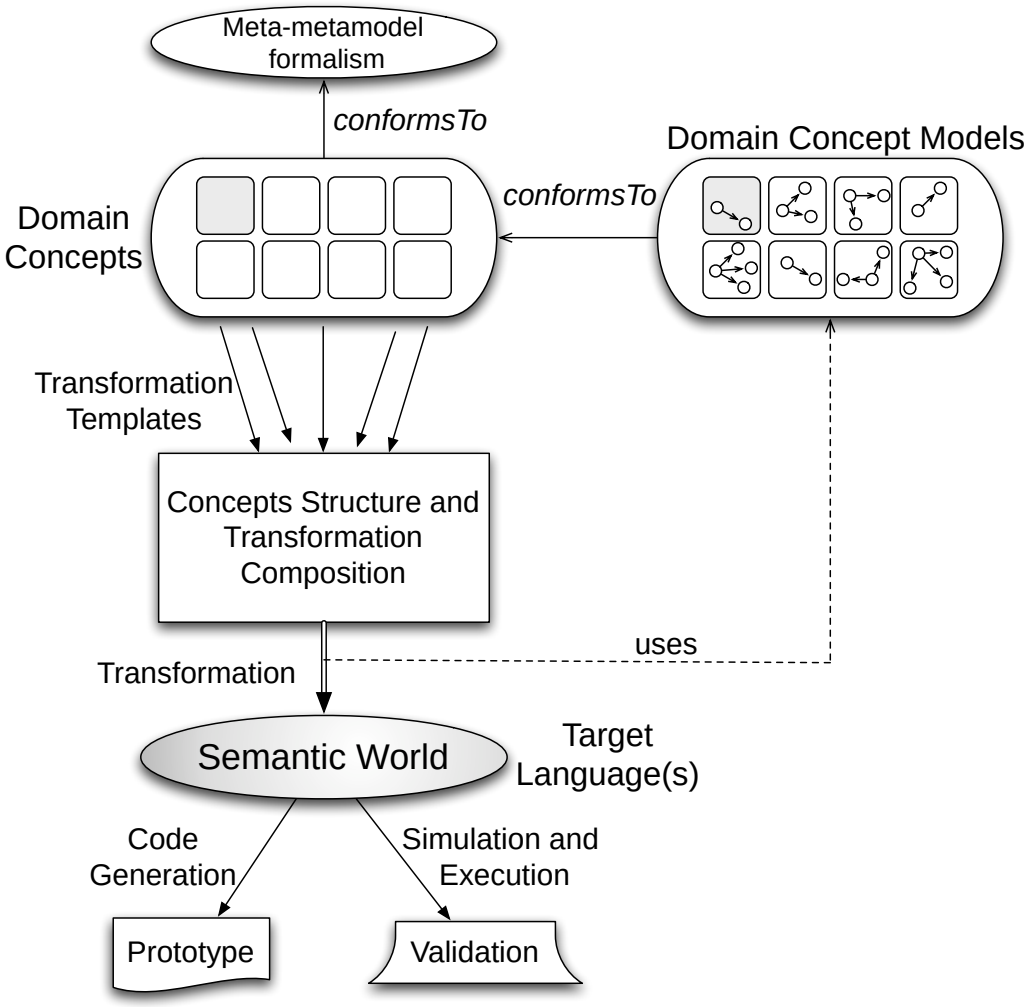


Figure 6.1. Composition Framework Overview

On the top of the figure is represented the meta-metamodelling formalism. It is the root for defining metamodels. It is a formalism that allows to specify the abstract syntax of a domain concept by providing constructs to define

metamodels.

The *Domain Concepts* box includes the metamodels of each DSML block. These metamodels are instances of the meta-metamodelling formalism and have associated transformation templates represented by the vertical arrows coming out from the box. Transformations are the operations that provide semantics to the domain concepts by mapping them to target models. These target models have an associated semantics. The transformation has as co-domain a formalism that provides to the models the desired semantics imposed by the transformation definition. The transformation of each block is a self-contained and atomic operation - transformation defined for each fragment of the domain should be executable independently from other domain blocks.

The $conformsTo$ is the relation defined in Definition 4.6. The conformity relation between *Domain Concepts* and the metamodel formalism, and between *Domain Concept Models* and *Domain Concepts* define the type of models of each one of the instances.

For being able to define a DSML using these blocks, the metamodels of the blocks must be composed. This is done by means of language block parameterization. The *Concepts Structure and Transformation Composition* rectangle represents the action of composing the concepts according to the defined parameterization. As it will be exposed in Sec. 6.2 and Sec. 6.3 this is a two step operation involving first the composition of metamodels and second the composition of the transformations.

The definition of a transformation for each domain block, together with the instance models that are in conformity to the block metamodels, allows to build specifications in the target languages. Each domain concept, which has an associated transformation model, can be parameterized by another domain concept, which also has a transformation model. This parameterization means we can replace a concept in a block with a concept from another block which is richer, more refined, or has a different transformation template.

A transformation is represented by one of the arrows labelled *Transformation Templates*. Each transformation is defined as a set of other transformations $Tr_{mm \to mmt} = \{Tr^1_{mm \to mmt}, Tr^2_{mm \to mmt}, \ldots, Tr^n_{mm \to mmt}\}$ each one of them corresponding to the rule(s) of transforming certain elements of the source model into elements of the target model.

A transformation is a function $Tr_{mm \to mmt} : im, ctx \to imt, ctx'$, where im is a model in the source DSML (i.e. an instance of the source DSML metamodel mm), imt a model in the target formalism (i.e. an instance of the target formalism metamodel mmt), ctx and ctx' the system's execution state before and after transformation.

Each domain block can be parameterized by defining the elements that

serve as parameter and the ones that replace them. The parameterization is processed in two steps: the syntactical one implying the metamodel composition, and the semantic one that takes care of transformations' composition.

6.2 Metamodel Parameterization

A parameterized metamodel is a metamodel where a parameter is described by its formal parameter. This parameter can be replaced by an effective parameter. For each parameterization, several possible replacements can be accomplished.

A parameterized metamodel is defined as follows.

Definition 6.1. Parameterized Metamodel.

Having $mm \in \mathbf{MM}$, and taking into account that **MM** is the universe of metamodels a parameterized metamodel is:

$$mmpar = \langle mm, fp, F_{fp} \rangle$$

where:

- $fp \in \mathbf{MM}$ is a formal parameter that acts as a template for the possible replacements;
- and F_{fp} is a set of formulas representing constraints over fp that must be respected.

The formal parameter $fp \subseteq mm$ since it defines a sub-set of the elements in mm.

The parameterization can be instantiated by the effective parameter $ep \in MM$, iff $ep \models \varphi(F_{fp})$. This means that conditions expressed in the fp world must be satisfied by ep. The φ is a total function that creates a map between elements of fp and ep.

The fp defines a template of what elements can be replaced in the metamodel mm. These elements are a set of the `ECore` model composed by `EClass` and their relations, the `EReference` elements. The fp metamodel is a subset of the metamodel mm. It can not define elements outside the set of metamodel elements defined from mm.

The effective parameter ep is, in fact, a metamodel that defines at least the elements defined by fp. This is the main difference between fp and ep: fp defines the minimum template and ep provides the real metamodel to serve as concrete parameter used for instantiation. Instantiation is allways performed by using ep.

In order to proceed with a metamodel composition, the formal parameter fp is then replaced by an effective parameter ep.

Definition 6.2. Metamodel parameterization.

Given mm, mm', fp, ep metamodels and F_{fp} a set of constrait formulas, a parameterization is defined as:

$$mm' = mm[fp \overset{\varphi}{\leftarrow} ep, F_{fp}]$$

where,

- $ep \supset \varphi(fp)$ re-defines, at least, the elements defined by fp;
- φ is a total function that creates a map between elements of fp and ep

  $$\varphi : fp \rightarrow ep$$

  in order to establish the replacement of nodes (classes) and references (associations, aggregations and generalisations).
- F_fp the set of formulas over the formal parameter fp that must be satisfied after parameterization.

This definition presents how metamodel parameterization is specified. Three metamodels must be provided in order to have a complete specification of a parameterized metamodel. The metamodel composition execution depends on the possible instantiation of the specified parameterization. The conditions expressed in the fp domain must be satisfied by ep. Otherwise metamodel composition, as specified, can not be executed.

A simplified diagram of the metamodel parameterization is presented in Fig. 6.2.

This figure shows that a DSML metamodel is extended by defining its formal parameter and by substituting it with an effective parameter.

On the left-hand side the metamodel of the DSML is represented. In the white background ellipse a grey background rectangle shows the part of the DSML metamodel that is defined by fp. Several formal parameters fp can be defined for each DSML each one of them representing a distinct parameterization[1]. On the right hand-side of the figure the ep metamodel is presented. The latest is going to substitute fp and a new DSML metamodel is generated. The metamodel resulting from parameterization is depicted in the bottom part of the figure.

[1]For simplification of the theory presented we only express one formal parameter fp.

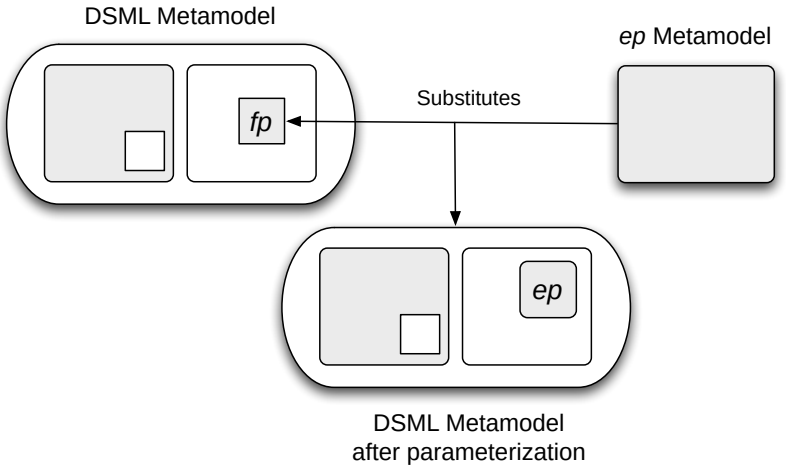


Figure 6.2. Metamodel Extension by Parameterization

6.2.1 Metamodel Parameterization Example

Let us use simple system's DSML specification in order to illustrate the methodology. Suppose we want to define a modelling language for the purpose of specifying the simulation of moving entities. Although we might understand the general principle that suits several domains demanding for DSMLs to implement their concepts (like train systems control, street traffic control, etc.), we do not have yet the full details to completely describe a DSML.

The general concepts involved, that might influence the syntax description of a DSML, could be described as having both `World Structure` information, and the mentioned moving `Entities` (i.e. trains, cars, etc.).

The `World Structure` could be composed by `Junction Points` and `Way Segments` (i.e. cross-roads, etc.) that are responsible to connect `Way Segments` (that depending on the domain could be particularised to rails, streets, channels, etc.) . Each `Way Segment` is composed by two end points, each of them could be connected to one `Junction Point`. However how many segments are plugged to junction points, we define as rule that we can not plug both end points of a given to the same `Junction Point`.

The corresponding metamodel of the general concepts described previously are depicted in Fig. 6.3. This figure defines the metamodel of a generic DSML here named $mmDSML_{gen}$.

This example allows to build metamodels for several (two in the example we are presenting) different DSMLs starting from the same generic metamodel: one DSML for describing a a Train system; and the other a metamodel

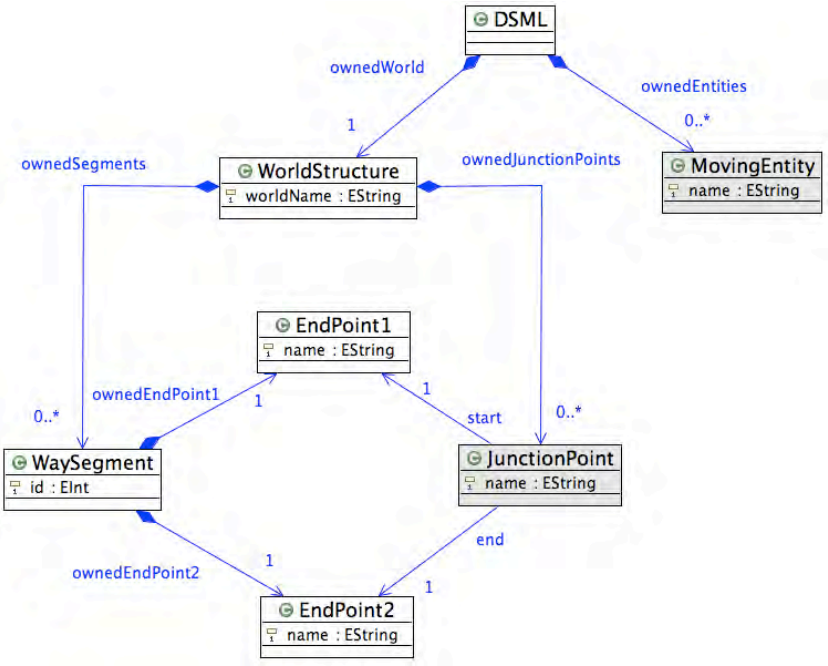


Figure 6.3. Generic Moving Entities DSML Metamodel $mmDSML_{gen}$

for representing a Robot System. Both metamodels are going to be defined by using the parameterization definition 6.2. In the metamodel from Fig. 6.3 we represent the parameterized elements `JunctionPoint` and `MovingEntity` with a grey background in order to better distinguish them.

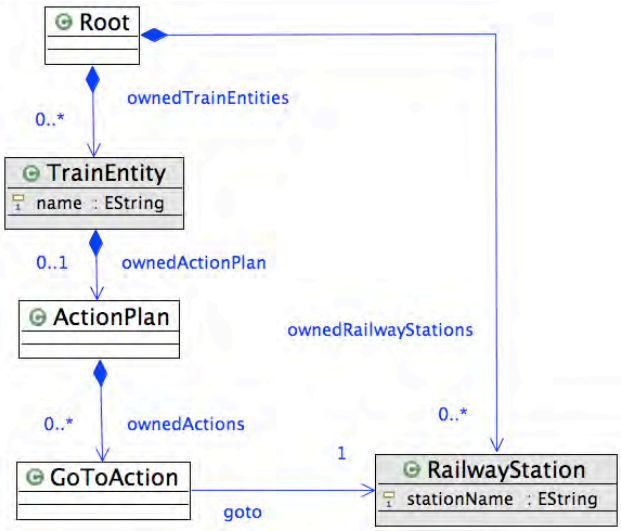


Figure 6.4. The Train Entity Metamodel $mmTrain$

The Railway System Metamodel

Now we want to define a modelling language for the purpose of specifying the simulation of a simplified railway system. The particularisation of our previously defined Entity in the general metamodel to the concept of train is depicted in Fig. 6.4.

Basically, we define the concept `Train` as having a Structure with an attribute `name`, and the behaviour as an `Action Plan`.

A possible particularisation of the concept `Junction Point` could be to the concept of `Railway Station`.

The referred `Action Plan` is a sequence of possible `GoTo Actions`. Informally the behaviour of a `GoTo` action is to send the a particular train to a given Railway Station.

Having $mmTrain$ the metamodel corresponding to the Train System, we define the parameterization as follows:

- The formal parameter fp the metamodel corresponding to the `MovingEntity` and `JunctionPoint` elements of $mmDSML_{gen}$;
- $mmTrain$ in 6.4 the effective parameter ep;
- $\varphi = \{\langle MovingEntity, TrainEntity\rangle,$
  $\langle JunctionPoint, RailwayStation\rangle$
  $\langle ownedEntities, ownedTrainEntities\rangle$
  $\langle ownedJunctionPoints, ownedRailwayStations\rangle\}$

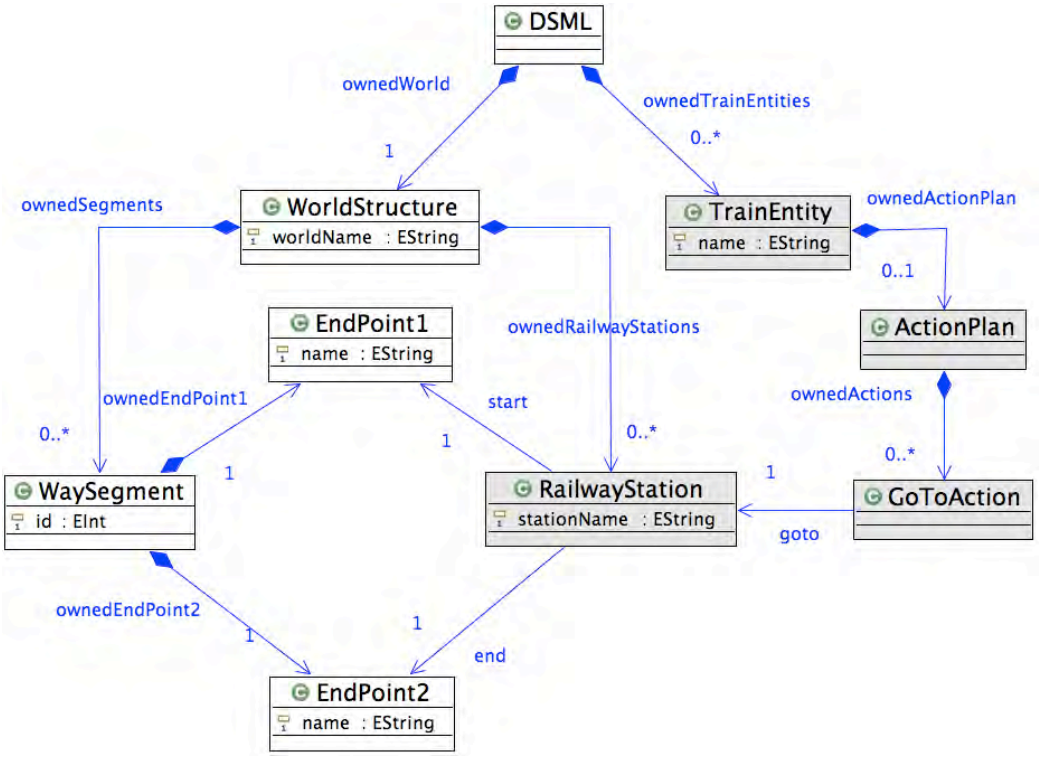


Figure 6.5. Railway System Metamodel $mmDSML_{RailwaySystem}$

The result of the metamodel parameterization is presented in Figure 6.5 with the new and affected elements prior to transformation with a grey background. As it was previously introduced, the new metamodel is obtained by applying the metamodel parameterization as:

$$mmDSML_{RailwaySystem} =$$

$$mmDSML_{gen}[\{MovingEntity\} \cup \{JunctionPoint\} \overset{\varphi}{\leftarrow} mmTrain, true]$$

The *true* F_{fp} means that, for this example, they are allways satisfied. This means the metamodel can be composed without having to take into account particular constraints regarding the formal parameter structure.

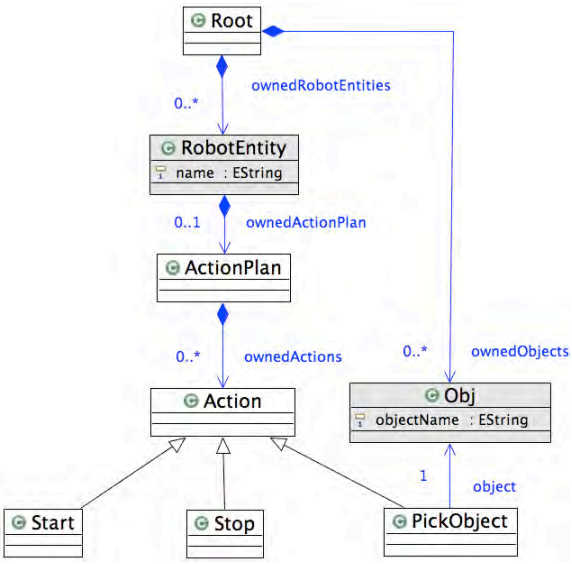


Figure 6.6. The Robot Entity Metamodel mmRobot

The Robot System Metamodel

Suppose now that we want a DSML to describe the domain of robot systems. Let us define a simplified robot system. The Robot has no particular way segment to follow, nevertheless the Junction Points can be seen as intermediate Pickable Object.

The sequence of possible actions for our Robot as defined in Figure 6.6 is: Start, Stop and Pick Object. The informal semantics associated to these three actions can be described in natural language in the following way:

- Start - to start moving the robot forward in order to reach the Pickable Object and make it disappear once reached. The robot stops immediately after waiting for the next target. If no goal is set the robot does not move;
- Stop - to stop moving the robot;
- Pick Object - this action sets the target (Pickable Object) where the robot should move. In other words the robot gets the reference to the object to pick, rotating a certain angle in order to be facing the object and be able to mode forward in a straight line to find the object when the Start action is called.

Having $mmDSML_{RobotSystem}$ the DSML metamodel corresponding to the Robot System, we define it as follows:

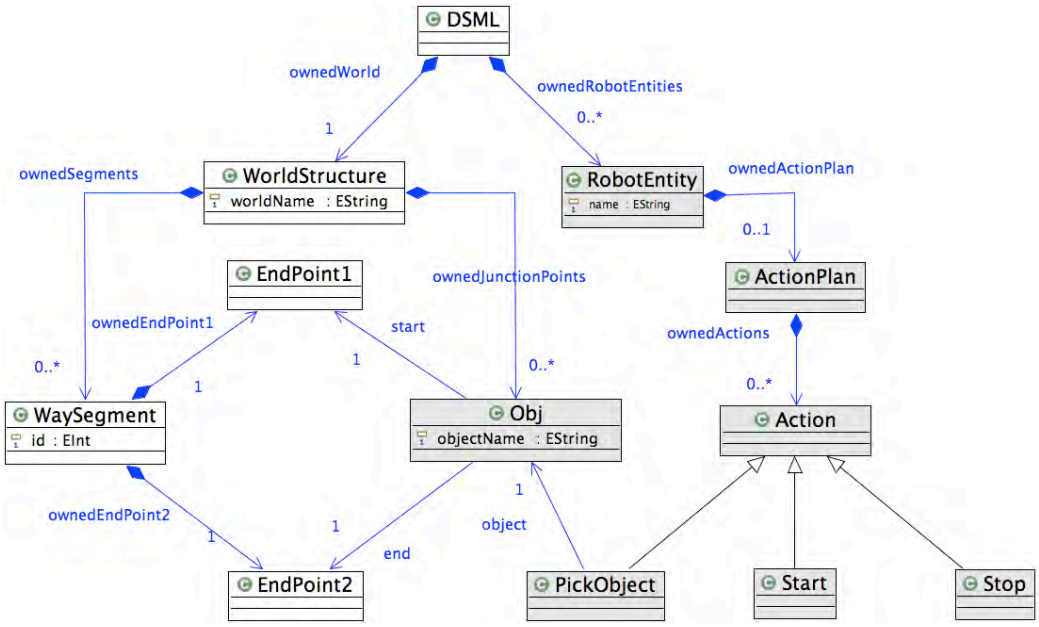


Figure 6.7. Robot System Metamodel

- the formal parameter fp the metamodel corresponding to the `MovingEntity` and `JunctionPoint` elements of $mmDSML_{gen}$;
- the effective parameter ep the metamodel $mmRobot$ in Fig. 6.6;
- $\varphi = \{\langle MovingEntity, RobotEntity\rangle, \langle JunctionPoint, Object\rangle\}$

The result of the metamodel parameterization is presented in Figure 6.7. The Robot System metamodel is obtained by applying:

$$mmDSML_{RobotSystem} =$$

$$mmDSML_{gen}[\{MovingEntity\} \cup \{JunctionPoint\} \overset{\varphi}{\leftarrow} mmRobot, true]$$

As we exposed in this section we can observe that parameterization of metamodels is a suitable technique for building DSMLs in an incremental procedure. At the same time it was exposed that the same approach also allows to re-use concepts in the definition of several DSMLs. This technique permits to have a more constructive and engineering approach while defining metamodels for DSMLs when compared to the traditional away of defining every DSML metamodel from scratch and as a single block without having any common boundary between other metamodels.

6.2.2 Models Composition

As a consequence of the metamodel parameterization and creation of new metamodels in conformity with the parameterization's definition, models that are $conformTo$ the metamodels involved in the paramameterization must also be composed. This composition follows the same rules as the metamodel composition resulting from the parameterization. Models of the two metamodel involved in the parameterization stay intact (in order to allow traceability and compatibility to its initial metamodels). Rather than modifying the existing models, new models are generated that will respect the $conformTo$ relationship with the metamodel resulting from parameterization.

In what concerns this point, some practical aspects need to be analysed and in which will focus on Sec. 7.4.2. This analysis and the solutions proposed will allow the language engineer to be sure that the desired composition is being executed.

6.3 Transformation Definition

A transformation is a set of mapping rules that define the semantics of a DSML in a target formalism.

Definition 6.3. Transformation.
A transformation is the set transformation blocks

$$Tr_{mm \to mmt} = \{Tr^1_{mm \to mmt}, Tr^2_{mm \to mmt}, ..., Tr^n_{mm \to mmt}\}$$

where,

- $mm \in \mathbf{MM}$ represents the metamodel for which the transformation relates to;
- $mmt \in \mathbf{MM}$ is the metamodel of the target formalism;
- n is the number of transformation blocks (or transformation rules) that compose a transformation.

Each $Tr^n_{mm \to mmt}$ block defines a semantic mapping between a part of the source and target formalisms.

A transformation is the application of the set of rules in a model of the formalism.

Definition 6.4. Transformation Application.

$$Tr_{mm \to mmt}(im) = im'$$

where,

- im is the model that is $conformTo$ mm metamodel;
- im' the model in the target DSML. im' is in conformity with mmt.

Applying a transformation $Tr_{mm \to mmt}(im)$ implies that the model im that is $conformTo$ the metamodel mm is transformed into another model im' in the space of conformity of the target metamodel mmt.

6.4 Parameterized Transformations

A DSML or domain concept have associated transformations providing the necessary semantics in the target formalism. Since a DSML is parameterized this means that the transformations must also be parameterized. Such parameterization is defined in this section.

In the semantic domain, when defining a parameterization we also have to define it onto transformations associated to the metamodels. A transformation parameterization is defined as:

Definition 6.5. Transformation Parameterization.

$$Tr_{mm' \to mmt} = Tr_{mm \to mmt}[Tr_{fp \to mmt} \xleftarrow{\varphi,\psi} Tr_{ep \to mmtp}]$$

where:

- mm is the metamodel being parameterized;
- mm' is the metamodel resulting from the mm metamodel parameterization: $mm' = mm[fp \xleftarrow{\varphi} ep, F_{fp}]$
- mmt the metamodel of the target language;
- $Tr_{fp \to mmt}$ the template transformation defined for fp;
- $Tr_{ep \to mmtp}$ the template transformation defined for ep;
- φ the source mapping function: $Dom(Tr_{fp \to mmtp}) \to Dom(Tr_{ep \to mmtp})$ where Dom stands for $Domain$ of a transformation;

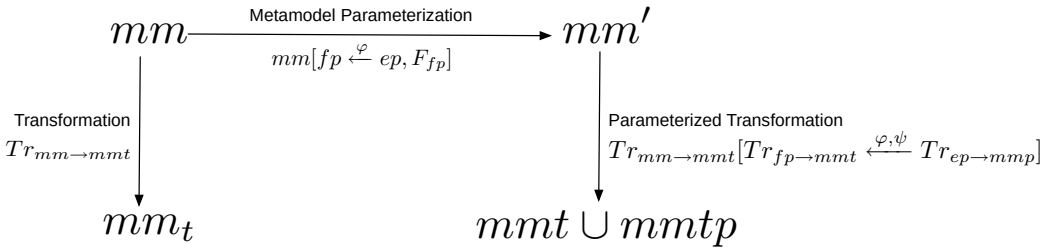


Figure 6.8. Relationship between Metamodels and Transformations

- ψ the target mapping function : $Cod(Tr_{fp \to mmtp}) \to Cod(Tr_{ep \to mmtp})$ where Cod is the $CoDomain$ of a transformation;

A parameterized transformation acts from models $conformTo$ the mm' metamodel that is already a the result of a parameterization. The resulting models of a parameterization are in the domain of the mmt metamodel. The transformation's parameterization allows to compose transformations defined for models $conformTo$ the formal parameter fp and having as domain of application models $conformTo$ the metamodel of the target platform mmt, with the transformation of models $conformTo$ to the effective parameter ep into models of on the domain of the target platform metamodel $mmtp$. The target platform of the transformation defined for fp and ep can be different. Fig. 6.8 shows the relation between metamodels and the transformations involved in the parameterization process.

Similarly as for the metamodel parameterization's instantiation, transformations also have an associated behaviour when it comes to its instantiation.

Definition 6.6. Transformation Instantiation.

$$im = Tr_{mm' \to mmt}(im') = Tr_{mm \to mmt}[Tr_{fp \to mmt} \xleftarrow{\varphi,\psi} Tr_{ep \to mmtp}](im')$$

where,

- mm is the metamodel being parameterized;
- mm' is the metamodel resulting from the mm metamodel parameterization $mm[fp \xleftarrow{\varphi} ep, F_{fp}]$;
- mmt the metamodel of the target formalism;
- im' a model $conformTo$ mm';
- im a model $conformTo$ the target metamodel mmt.

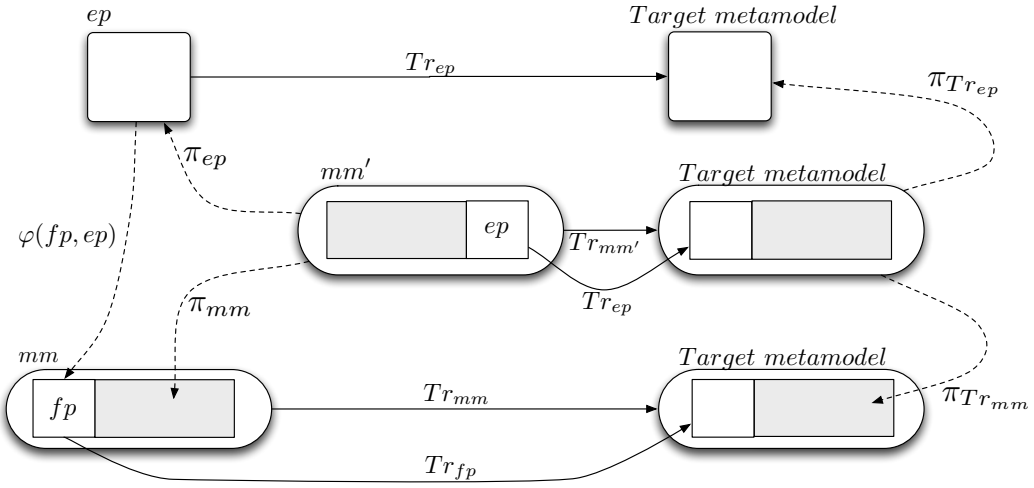


Figure 6.9. Parameterization of Transformations (without instances)

In other terms, if the formal parameter fp relates to ep by φ in the source DSML domain, and if we apply the transformations $Tr_{fp \to mmt}$ and $Tr_{ep \to mmtp}$ to instances ifp and iep of their respective metamodels, then ψ expresses the relation between $Tr_{fp \to mmt}(ifp)$ and $Tr_{ep \to mmtp}(iep)$ in the target language domain. From the operational point of view, ψ defines which transformations in fp are replaced by what transformation in ep.

Fig. 6.9 resumes how the mm, fp, ep and mm' metamodels and their transformations relate to each other[2].

The arrows marked with π represent projection of metamodels: the mm' metamodel, for example, if projected by mm (i.e. πmm), gives the grey part of mm, i.e. the part that does not include fp. This figure does not presents the ψ function, as this figure does not represents any information at the level of the instantiation. Each one of the transformations present in the DSML specification has an associated projection that represents which elements of the metamodel it relates with.

Via these parameterizations and by applying the resulting transformations on the instances of the resulting metamodels, it is possible to obtain models in the target language with a high level of re-usability of previously existing transformations.

Fig. 6.10[3] shows a more concrete overview over the transformations. It

[2]In order to improve the readability of the figure and we simplified the notation for the representation of the transformation. We represent Tr_{ep} instead of $Tr_{ep \to mmtp}$, Tr_{mm} replacing $Tr_{mm \to mmt}$ and Tr_{fp} instead of $Tr_{fp \to mmt}$

[3]For readability purposes, this figure does not expicits the target metamodels in the transformation functions.

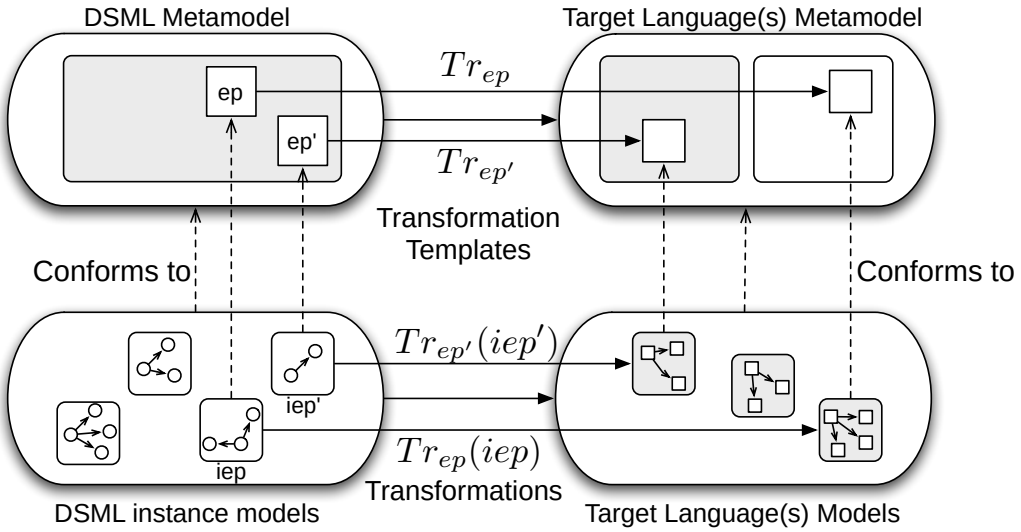


Figure 6.10. Instantiation of the Transformation Parameterization

represents generic schematic view from the point of view of the transformation instantiation and execution.

On the top of the figure it is represented the relation between several possible *ep*, corresponding transformation, and the part of the target's formalism metamodel. In this figure, Tr_{ep} stands for $Tr_{ep \to mmtp}$ and $Tr_{ep'}$ stands for $Tr_{ep' \to mmtp'}$. The $Tr_{ep \to mmtp}$ and $Tr_{ep' \to mmtp'}$ are not to be seen as transformation instantiations. They are rather a way of showing to which parts of the metamodel they are related to, e.g. $Tr_{ep \to mmtp}$ relates the *ep* metamodel with the target formalism metamodel represented by the small white background rectangle. The bottom of the figure show the instances of the metamodels' and how transformations produce models in the target formalism domain. Instantiating the transformation $Tr_{ep \to mmtp}(iep)$ produces a model corresponding to the graph in the grey background square at the bottom right side of the figure.

Definition 6.7. Parameterized Transformation Construction.

Giving $Tr_{mm \to mmt}$ a parameterized transformation construction is defined as:

$$Tr_{mm \to mmt \cup mmtp} = Tr_{mm \to mmt} \backslash Tr_{fp \to mmt} + Tr_{ep \to mmtp}$$

- mm is the metamodel of the DSML;
- fp is a metamodel sub-set of mm;

- *ep* is a parameterization metamodel;
- *mmt* and *mmtp* are metamodels of the target formalisms.

The final set of transformation rules are obtained by subtracting[4] from the initial set of rules the ones corresponding to the formal parameter and adding to the rules that define the transformation of the effective parameter.

6.5 DSML Definition

Transformations provide a way of going from a source to a target formalism. In the particular case of this work the goal is to start from a DSML and to finish in the dialect of a target formalism. The target formalism being the one that provides the necessary semantics. As a consequence from the fact that both metamodels and transformations are parameterized, a DSML defined using this technique, will also be a parameterized DSML.

Having formalised metamodels, composition of metamodels and associated transformations, we continue by providing a definition of a DSML and what a parameterized DSML stands for.

Definition 6.8. Domain Specific Modeling Language.
A DSML is as a 4-tuple

$$DSML = \langle mm, mmt, F, Tr_{mm \rightarrow mmt} \rangle$$

where,

- mm is the DSML metamodel;
- mmt the metamodel of the target formalism;
- F the set of constraints over the DSML metamodel[5];
- $Tr_{mm \rightarrow mmt}$ is the set of transformation rules providing the DSML semantics.

The mm metamodel and the transformation $Tr_{mm \rightarrow mmt}$ have both been obtained by parameterization.

[4]The + and *backslash* are the set operators for union and minus

[5]Constraints can be expressed in any kind of constraint language supported by the metamodelling formalism used in the implementation of the methodology, e.g. OCL.

6.6 Transformation Composition in Action

Previous sections defined both metamodel and transformation compositions. Although in what concerns metamodels we think it is clear how the approach applies, regarding transformation we believe that some more detail will help to clarify the approach presented. Taking this into account, this section is dedicated to show how transformations are composed.

For the purpose of better understanding of transformations' composition we consider two scenarios:

1. The case in which the transformation domain element in fp defined by ψ is in the *edge* of the metamodel. This means that the element has no linked elements from it. It is an element with no connected nodes from it;

2. The case in which the element is not an *edge* element of the metamodel.

For the first case, the transformation composition is performed as shown in Figg. 6.11, 6.12 and 6.13.

The three figures present three different views which illustrate, step by step, what happens while composing the transformation rules.

For what concerns these descriptions we consider that:

- $Tr_{fp \to mmt}$ is the set of n transformation rules defined for fp metamodel: $Tr_{fp \to mmt} = \{Tr^1_{fp \to mmt}, Tr^2_{fp \to mmt}, ..., Tr^n_{fp \to mmt}\}$;

- analogously, Tr_{ep} is the set of transformation rules for ep metamodel: $Tr_{ep \to mmt} = \{Tr^1_{ep \to mmt}, Tr^2_{ep \to mmt}, ..., Tr^n_{ep \to mmt}\}$;

- $TrDSML_{mm' \to mmt}$ stands for the complete definition of the transformation after composition. In the diagrams representing each one of the views, only blocks corresponding to the rules of $Tr_{ep \to mmt}$ and $Tr_{fp \to mmt}$ are represented. This was done explicitly for simplicity reasons although the full transformation$TrDSML_{mm' \to mmt}$ might contain other rules;

For the case of a composed DSML in which the fp element in the φ function is an edge element like in Fig. 6.11 the ψ composition function is defined as:

$$\psi = \{\langle Tr^2_{fp \to mmt}, Tr_{ep \to mmt} \rangle\}$$

Meaning the transformation defined for the element in the fp metamodel to which corresponds the transformation $Tr^2_{fp \to mmt}$ is to be replaced by the transformation defined for ep

View 1 from Fig. 6.11 shows the state of the transformation rules before anything happens;

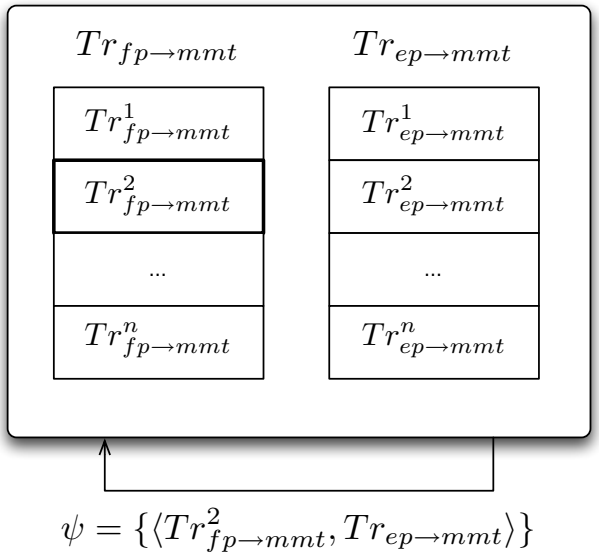


Figure 6.11. Transformation Composition for Edge Elements - View 1

View 2 in Fig. 6.12 presents the step in which the left side of the ψ function is erased from the full transformation view. This a necessary intermediary step to allow replacement of *ep* corresponding transformation in the next step;

View 3 as depicted from Fig. 6.13 shows how transformation rules are packed together by the end of the execution of the composition. This is the last step in the composition chain that places the transformation rules defined for *ep* in the empty spot left from the previous step.

Since the transformations are, as defined previously, a set of transformation rules, the order in which the rules are applied is not important. Moreover, rules that are applied by using a graph based matching technique will always be applied by following the matched patterns and not the order which they were specified.

Transformation composition defined over elements from non-edge elements are more complex. The main reason for the increase of complexity is because, typically, we don't want to replace the totality of the element's transformation for a new one. By complete replacing its transformation, we'll lose transformation information we don't want to: we might have transformation rules for the linked elements that depend on the execution of that element transformation.

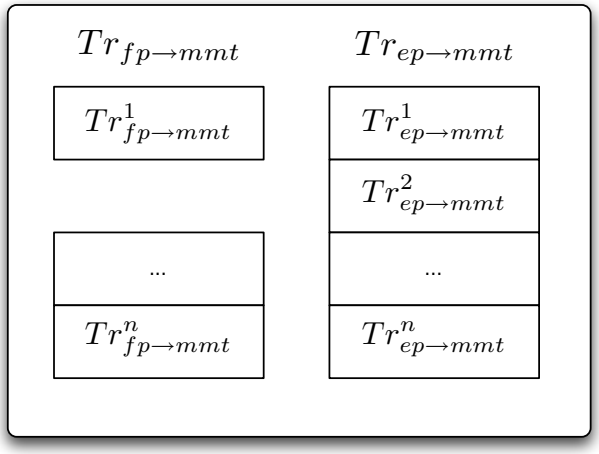


Figure 6.12. Transformation Composition for Edge Elements - View 2

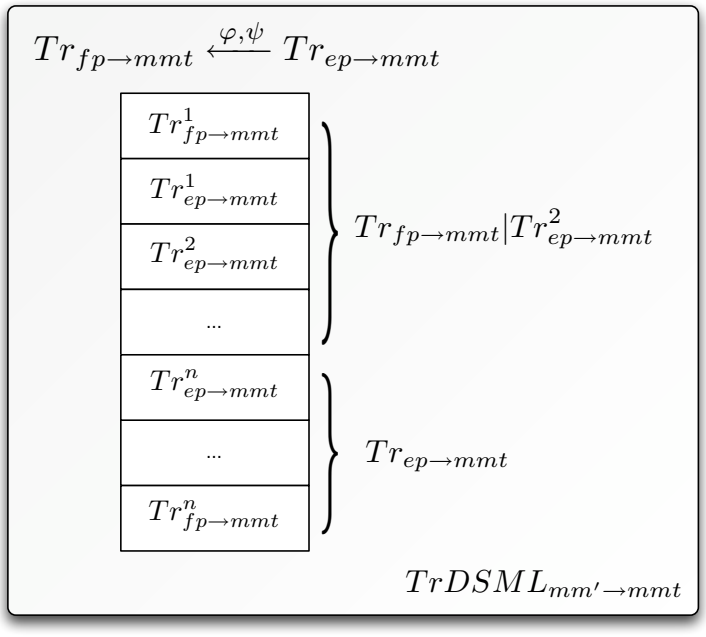


Figure 6.13. Transformation Composition for Edge Elements - View 3

The main goal behind this strategy is to be able to compose transformation blocks without the need to replace transformation rules in the fp side, but rather to do it in the ep side. Figg. 6.14, 6.15 and 6.16 provide a schematic perspective on the three views that build the composition process:

View 1 presented in Fig. 6.14 shows the state of the transformation rules after composition execution. In the left-hand we have the set of rules corresponding to $Tr_{fp \to mmt}$. The rule $Tr^1_{fp \to mmt}$ is what we define as TF the rule. This is the rule we are parameterizing but we don't want to replace its transformation definition. This rule is $Tr_{fp \to mmt}|TF$ a restriction of $Tr_{fp \to mmt}$ to TF. On the right side we define $Tr_{ep \to mmt} - TE$ as the piece of $Tr_{ep \to mmt}$ that is going to replace $Tr^1_{fp \to mmt}$;

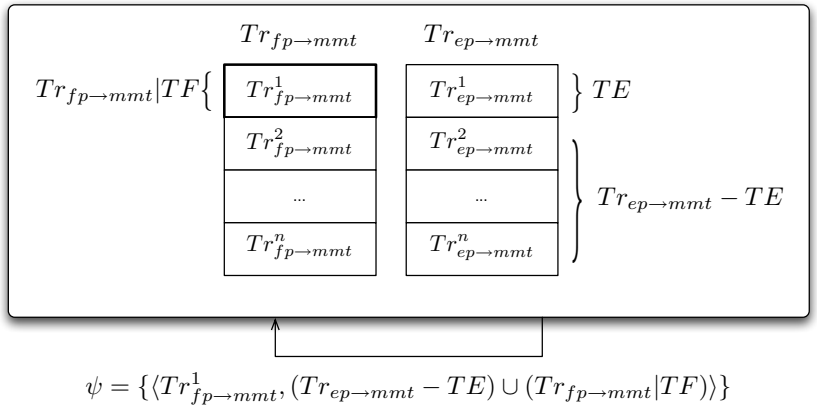


Figure 6.14. Transformation Composition for non-Edge Elements - View 1

View 2 depicted from Fig. 6.15 presents the step in which the right side of the ψ function is removed from the full transformation view. $Tr_{ep \to mmt}$ is now only the part $Tr_{ep \to mmt} - TE$ as TE is unwanted in the final version of the transformation;

View 3 in Fig. 6.16 shows how transformation rules are packed together by the end of the execution of the composition.

The composition definition for these cases is

$$\psi = \{\langle Tr^n_{fp \to mmt}, (Tr_{ep \to mmt} - TE) \cup (Tr_{fp \to mmt}|TF)\rangle\}$$

where $Tr^n_{fp \to mmt}$ is the rule we want to parameterize but not to replace in the source DSML transformation. TE the rule in the effective parameter transformation set that is to be removed so that $Tr^n_{fp \to mmt}$ can be maintained.

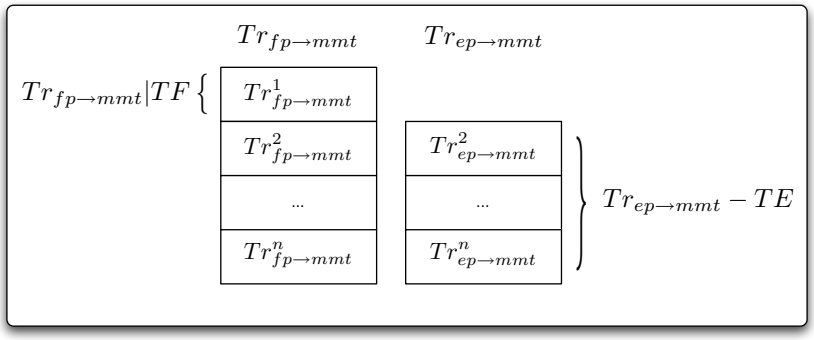


Figure 6.15. Transformation Composition for non-Edge Elements - View 2

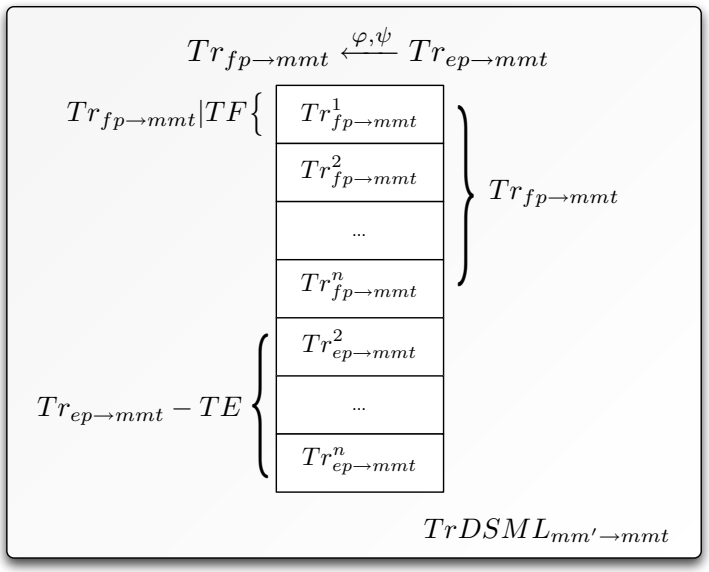


Figure 6.16. Transformation Composition for non-Edge Elements - View 3

6.6.1 Railway System DSML Transformation Composition

For illustrating the composition of transformations lets consider the definition of the Rail Way System DSML base on the Moving Entities generic DSML defined previously. First let's start by showing what is an Edge and a non-Edge element in metamodel. In Fig. 6.17 the `EClass MovingEntity` is what we consider an Edge element of the metamodel. The `EClass JunctionPoint` has linked elements such as `EndPoint1` and `EndPoint2` thus it is what we define as a non-Edge element.

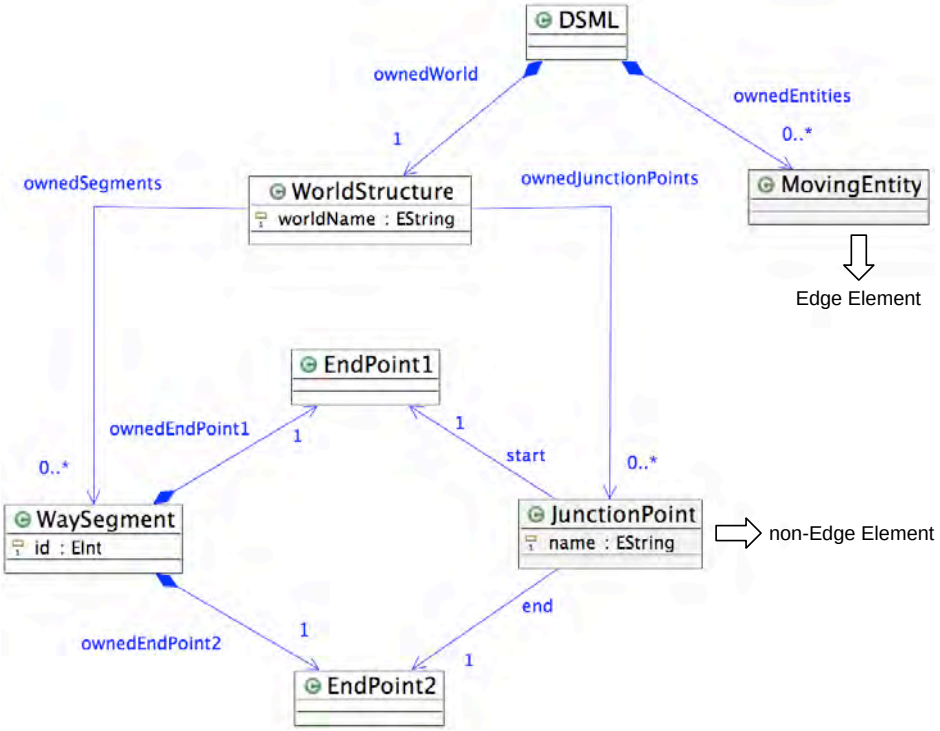


Figure 6.17. Example of Edge and non-Edge Metamodel Elements

The metamodel parameterization definition of this DSML was specified using the φ function

$$\varphi = \{\langle MovingEntity, TrainEntity\rangle, \langle JunctionPoint, RailwayStation\rangle, \langle ownedEntities, ownedTrainEntities\rangle, \langle ownedJunctionPoints, ownedRailwayStations\rangle\}$$

In this specification the elements that correspond to rules in the transformation definition are the $\langle MovingEntity, TrainEntity\rangle$ and

$\langle JunctionPoint, RailwayStation \rangle$. The first one is an edge element. It means that transformation composition is performed by replacing the transformation defined in the fp side by the one ep side. Schematically in Fig. 6.18 this means that the rule that corresponds to the `ruleMovingEntities` will be replaced by the rules defined in $Tr_{TrainEntity}$.

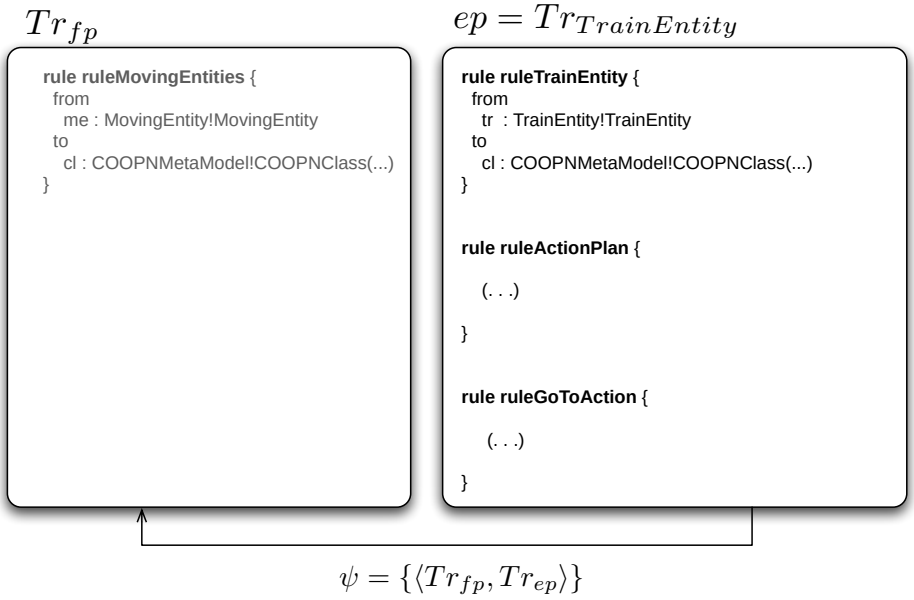


Figure 6.18. Example of Transformation Composition for an Edge Element

In terms of transformation composition for non-edge elements like the `JunctionPoint` being parameterized by `RailwayStation`, the composition works slightly diferent. We cannot just eliminate the transformation rule from the fp side because it (an usually does) might contain references to other transformation rules. By replacing it, we would lose all the transformation defined from it. Taking this into account, the composition in non-edge elements keeps the transformation rule defined for the element in fp, ignores the rule from the ep side that corresponds to the element in ep defined by φ, and adds to the fp transformation all other rules in the ep side. This procedure is illustrated in Fig. 6.19

The generic approach for composing transformations is as presented in this section. A complete description and definition of transformation compositions is presented in Chap. 8 where the case studies are described.

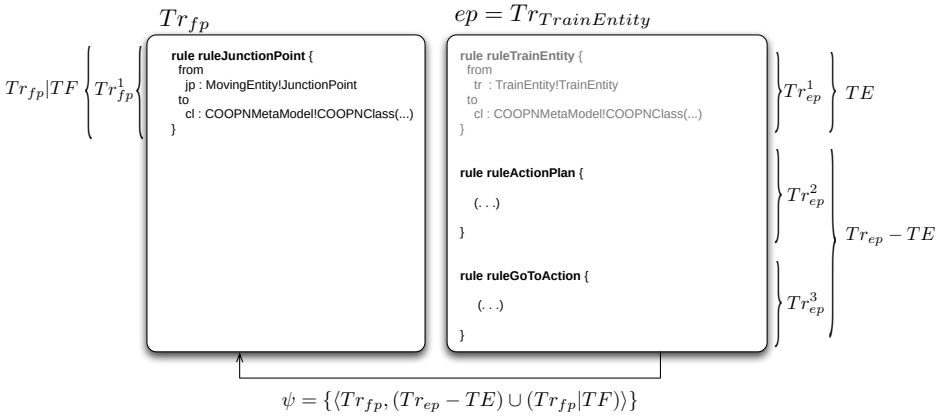


Figure 6.19. Example of Transformation Composition for an Non-Edge Element

6.7 Target Languages

The approach defined in this document supports multi-formalism approaches in the sense that several target languages can easily used together. The precondition for this to be allowed it to have a metamodel that represents the abstract syntax of the language. In the examples presented in Chap. 8 we mostly use CO-OPN as target formalism but we also demonstrate that other languages can be used. Specifically we show how Structured Query Language (SQL) can be used.

The choice of target formalism depends heavily on the kind of semantics we need to give to our DSML. We decided to use CO-OPN as a target language for most of the implementation of this methodology. The main justification relies on the fact that using CO-OPN allows using several language features, such as concurrency and transactionality, that are included in CO-OPN and that would be complex to express with a classic general-purpose language. Models expressed in CO-OPN are transformed into an equivalent semantics in Java by using COOPNBuilders' prototype generator(Chachkov, 2004).

Although we can argue that we can provide any type of semantics in CO-OPN language it is our believe that a methodology as presented in this document should be more general than to be constraint to a single target language and platform. In addition, while developing DSMLs we feel the need to allow users to specify and understand their models in an environment the most close to the problem domain as possible. Having saying that, it is easy to imagine that, for example when dealing with $3D$ representation of objects in a spacial environment, there exist a quantity of languages that will cope better than CO-OPN.

Chapter 7

CoPsy Environment

While the approach is general with respect to the transformation framework used, its implementation has some framework-specific adjustments. We chose to focus on the concrete framework provided from ATL (ATLAS Group, 2007). This framework provides a rule-based transformation language which offers both declarative and imperative constructs, lending itself well to complex transformations; and a framework making it easy to apply a set of transformation rules to a model, given a source and a target metamodel. We also use the Eclipse Modelling Framework infrastructure in order to develop CoPsy.

Compositional Platform for Domain Specific Modelling Languages Prototyping (CoPsy) is itself a DSML. It is available as an Eclipse pluggin developed using EMF allowing to specify DSML composition.

7.1 Problem Analysis

Being a DSML CoPsy captures the concepts of its domain of application. Its nature and containment allows to narrow its functionalities to a small and unbiguous list.

The requirements covered by CoPsy DSML are the following:

- Specification of metamodel composition;
- Execution of metamodel composition;
- Composition of instances of the metamodels;
- Specification of transformation composition;
- Execution of transformation composition.

These five items represent domain concepts that must be captured by CoPsy DSML. The CoPsy metamodel for this domain of application is presented in Fig. 7.3.

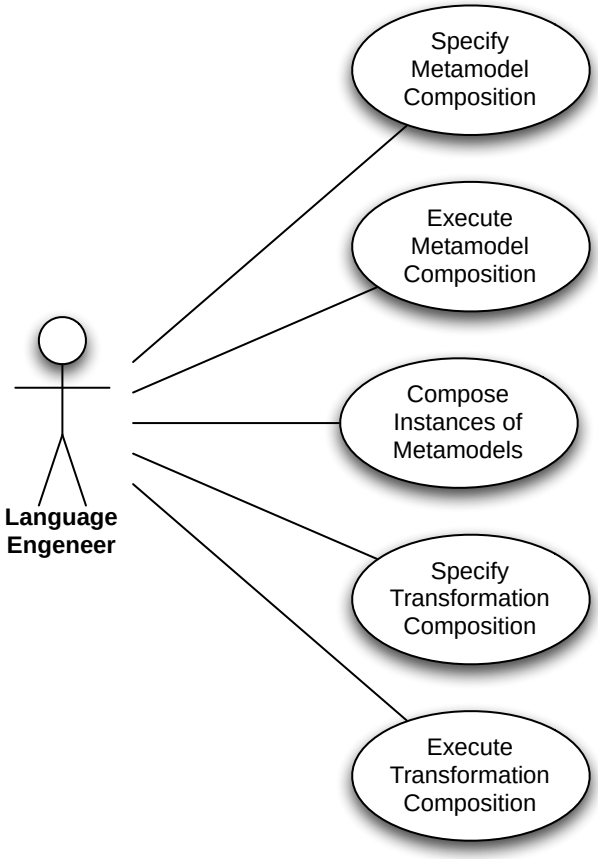


Figure 7.1. CoPsy Use Case Diagram

The Fig. 7.1 captures CoPsy requirements in form of UML use case diagram.

Remembering that a metamodel parameterization is defined by

$$mm' = mm[fp \overset{\varphi}{\leftarrow} ep, F_{fp}]$$

and that a transformation parameterization is defined by :

$$Tr_{mm' \to mmt} = Tr_{mm \to mmt}[Tr_{fp \to mmt} \overset{\varphi,\psi}{\longleftarrow} Tr_{ep \to mmtp}]$$

A more detailed version of CoPsy's functionalities is the follwoing:

- Express metamodel composition by specifying:

  - the base metamodel (mm). This represents the metamodel of the DSML that is being parameterized;
  - the parameter metamodel. It is the ep effective parameter metamodel that is going to replace the elements defined as fp formal parameter metamodel;
  - φ elements representing the mapping between ep and fp.

- Produce the metamodel resulting from parameterization;
- Produce an ATL transformation for composing instances of the base metamodel mm and the efective metamodel ep;
- Express transformation composition by specifying:
  - the transformation for the base DSML;
  - the transformation for the DSML of the ep domain;
  - ψ elements representing the mapping function between transformation rules for ep and fp.

The process of specifying and executing a CoPsy is present in form of UML activity diagram in Fig. 7.2. The activities involved are:

- `Configure` represents the process of configuration meaning that all necessary parameters are initialized;
- `Specify` φ is the activity in which the map between ep and fp elements are specified;
- `Execute Metamodel Composition` the action of composing metamodel mm and ep into a new metamodel mm';
- `Execute Model Composition` is the activity that allow to take models in conformity with mm and ep and make them in conformity with the new metamodel mm;
- `Specify` ψ the action of specifying the mapping function between transformation rules of fp and dp;
- `Execute Transformation Composition` the activity of composing the transformations from mm and ep into a single one taking into account the ψ mapping function;
- `Execute Transformation` the activity that takes the models in conformity with mm' and transforms them into the target formalism by applying the composed transformation.

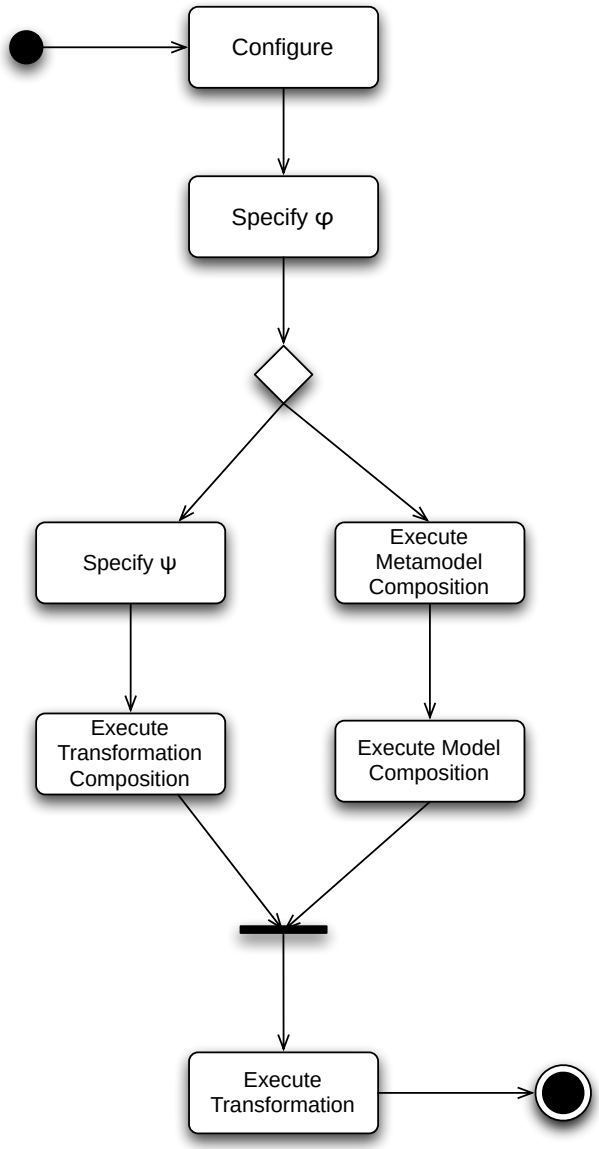


Figure 7.2. CoPsy Execution Activity Diagram

7.2 CoPsy Implementation

After having defined the CoPsy domain of application its implementation (as for defining any DSML using a Software Language Engineering approach) starts by providing an abstract syntax in form of metamdel. The CoPsy metamodel represented in Fig. 7.3. When instantiated it allows to create specifications of DSML composition.

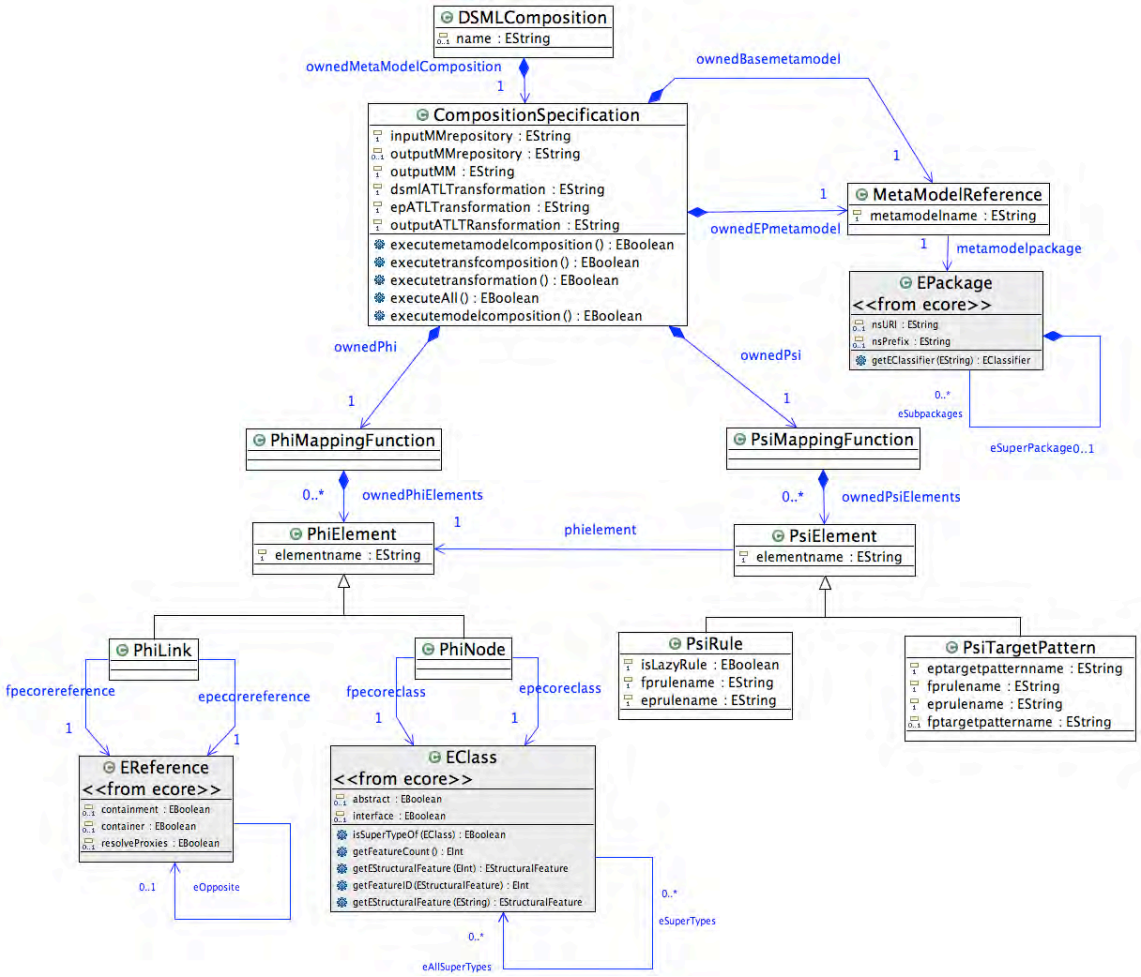


Figure 7.3. CoPsy Metamodel

7.2.1 CoPsy Metamodel

The CoPsy metamodel captures the specification of DSMLs composition. It can be divided in three main parts:

1. Default Parameters: they represent the attributes used in the `Configure` activity and allow the specification to be configured in what concerns all the necessary default parameters;

2. φ Mapping Function: the part of the metamodel that allows to specify metamodel composition relations;

3. ψ Mapping Function: where transformation rules relations are possible to define.

Each one of the CoPsy's metamodel sections is defined by the following parameters:

Default Parameters The `CompositionSpecification` meta-class defines the parameters that must be specified in order to define the configuration parameter for DSML composition:

- `inputMMRepository` is the name of the repository in which the metamodels and transformations are localised;
- `outputMMRepository` is the name of the repository in which the results of metamodel and transformation composition are going to be stored. This parameter is optional. If not provided it will be assumed as equal to `inputMMRepository`;
- `outputMM` is the name for the metamodel resulting from composition;
- `dsmlATLTransformation` the name of the ATL file containing the transformation rules of the DSML under parameterization;
- `epATLTransformation` the ATL file name with the transformation definition of the DSML of the effective parameter;
- `outputATLTRansformation` the ATL file in which the composed transformation is going to be stored;
- `ownedBaseMetamodel` aggregation to `MetaModelReference` metaclass allow to specify the metamodel of the DSML being parameterized;
- `ownedEPmetamodel` aggregation to `MetaModelReference` enables specification of the efective parameter metamodel. Both `ownedBaseMetamodel` and `ownedEPmetamodel` are defined by a `metamodelname` and a reference to an `EPackage`. This reference points to the Ecore metamodel. The `EPackage` metaclass in the CoPsy metamodel is not owned by the CoPsy DSML metamode. It is rather a reference to

the default Ecore metamodel. The fact of providing a reference to the Ecore `EPackage` explicitly allows CoPsy to be automatically *aware* of the structure and contents of the metamodels part of the composition process. This facilitates consistency checks and unambiguous specifications of φ mapping functions.

Metamodel Composition (φ function) The function that allows to map metamodel elements of the DSML metamodel *mm* and the effective parameter metamodel:

- From the `CompositionSpecification` metaclass the `ownedPhi` aggregation to `PhiMappingFunction` metaclass with 1..1 cardinality imposes the definition of a φ mapping function;
- The φ is a set of elements represented in the CoPsy metamodel by the metaclass `PhiElement` identified by an `elementname`. This metaclass is abstract and specialised the `PhiLink` and `PhiNode` metaclasses;
- The `PhiLink` metaclass allows to specify mappings between elements of the metamodel of type `EReference`. The definition of `fpecorereference` and `epecorereference` association with 1..1 cardinality forces the definition of correct map between an EReference in the DSML metamodel and an EReference in the metamodel of the *ep* DSML. The `EReference` metaclass is not part of the CoPsy metamodel - it is an element from the standard ECore metamodel.
- The `PhiNode` metaclass allows to specify mappings between elements of the metamodel of type `EClass`. Two references with 1..1 cardinality (`fpecoreclass` and `epecoreclass`) allows to enforce the coherence of the φ map in what concerns the nodes. As for the `EReference`, the `EClass` metaclass is an element of the ECore standard metamodel.

Transformations Composition ($\psi function$) The part of the metamodel that allows to map transformation rules defined for the DSML under parameterization and the rules in the effective parameter DSML:

- From the `CompositionSpecification` metaclass the `ownedPsi` aggregation to `PsiMappingFunction` metaclass with 1..1 cardinality allows to specify the ψ mapping function;
- The *psi* is a set of elements represented in the CoPsy metamodel by the metaclass `PsiElement` identified by an `elementname`. This

metaclass is abstract and specialised the `PsiRule` and `PsiTargetPattern` metaclasses;

- The `PsiRule` metaclass allows to specify ψ element that are related to ATL transformation rules. This element has three attributes: `isLazyRule`, `fprulename` and `eprulename`. All of them with 1..1 cardinality meaning that they must be provided while defining a CoPsy specification. The `isLazyRule` states if the mapping function under specification is an ATL lazy rule or a stardard rule. The `fprulename` and `eprulename` stand for the name of the rule in the DSML being parameterized and the rule in the effective parameter domain;
- Instead of defining a map between two rules, language engineer might just want to specify a map between a part of a rule. Since in ATL a rule is defined as a set of target patterns, in CoPsy we allow to create a map between fp and ep target patterns inside of an ATL rule. This is done by specifying the `PsiTargetPattern` element. This element is described by the `fprulename`, `eprulename` that points to which rule the target pattern is part of. The `fptargetpatternname` and `eptargetpatternname` define which target pattern part of the rules are to be mapped.

Defining Models This is were models in conformity to both DSML under parameterization and the DSML defined by the effective parameter are specified. These models are specified so that, after performing metamodel composition, the language is able to compose them in conformity to the composed metamodel. In the CoPsy metamodel this is represented by `DSMLModel` metaclass. This metaclass has `dsmlModel`, `epModel` and `composedModel`. The first two define a model of the DSML under parameterization and a model of the DSML of the effective parameter respectivly. The last one is optional and represents the model to be generated by the model composition activity.

This simple, yet powerfull metamodel allows to provide all information needed for specifying a DSML composition. Next section presents an example based on the Moving entities example in case study presented in Sec. 8.1.

7.3 Defining a CoPsy Specification

Lets assume we want to specify the composition of the Generic Moving Entities as in the example we presented while describing the methodology in Sec. 6.2.1, with the Railway System DSML in Sec. 6.2.1. The metamodels used for this specification are the ones defined in Figg. 6.3 and 6.4.

We start to define a CoPsy DSML composition by setting the configuration parameter. This means to create an instance of the `CompositionSpecification` metaclass with:

- `inputMMRepository` set to $/CoPsySpecs/MovingEntities$;
- `outputMMrepository`. The same value as the `inputMMRepository` will be assumed;
- `outputMM` the name of the output metamodel set to: $DSMLRailwaySystem.ecore$;
- `dsmlATLTransformation` set to $GenericMovingSystem.atl$;
- `epATLTransformation` set to $TrainEntity.atl$;
- `outputATLTRansformation` as $RailWaySystem.atl$
- One instance of `MetaModelReference` for the `ownedBaseMetamodel` aggregation that provides a reference to the $MovingEntitiesDSML$ EPackage. This represents the mm' metamodel in the formal definition;
- Another instance of `MetaModelReference` for the `ownedEPMetamodel` that provides a reference to the $TrainEntityDSML$ EPackage. This instance of `MetaModelReference` is the equivalent to the ep metamodel in the formal definition.

In Fig. 7.4 it is shown the CoPsy Editor with focus on the parameters' configuration as stated previously.

7.3.1 Specification of Metamodel Composition

The next step on defining a CoPsy specification is to define the metamodel composition parameters. Again we'll use the example defined in Sec. 6.2.1 in which the following metamodel parameterization was defined:

- The formal parameter fp the metamodel corresponding to the `MovingEntity` and `JunctionPoint` elements of $mmDSML_{gen}$;
- $mmTrain$ in Fig. 6.4 the effective parameter ep;

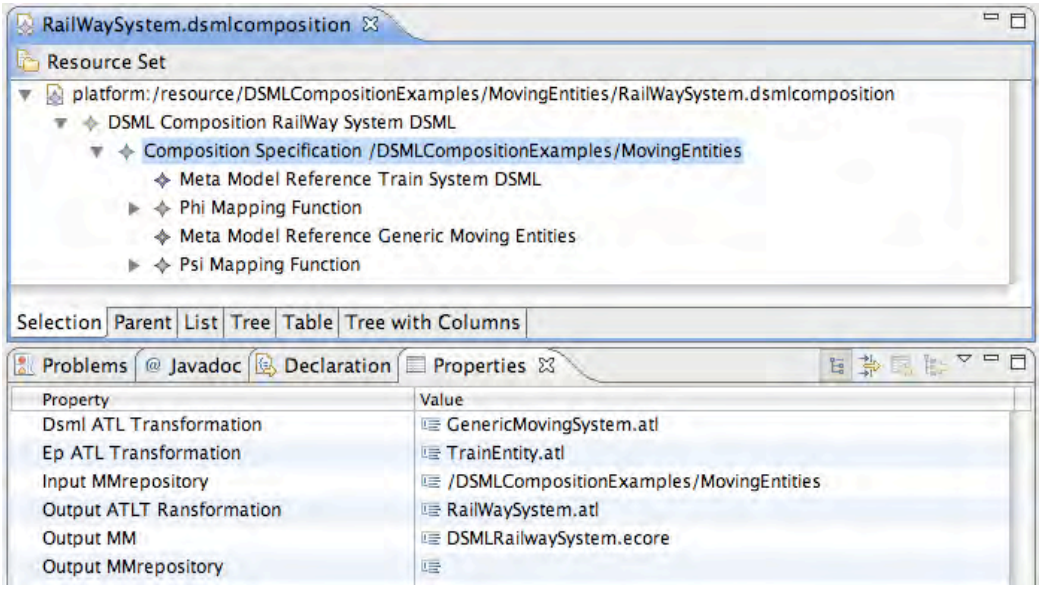


Figure 7.4. CoPsy Specification of Configuration Parameters

- $\varphi = \{\langle MovingEntity, TrainEntity\rangle,$
  $\langle JunctionPoint, RailwayStation\rangle,$
  $\langle ownedEntities, ownedTrainEntities\rangle,$
  $\langle ownedJunctionPoints, ownedRailwayStations\rangle\}$

In the CoPsy editor we represent this by specifying a set of instances of `PsiNode` and `PsiLink`. Fig. 7.5 shows how the φ mapping function is defined using the CoPsy editor. The CoPsy editor suggests the available EClasses or EReferences for each one of the metamodels specified in the configuration step. This means that, defining $MovingEntitiesDSML$ as the EPackage for the `ownedBaseMetamodel` in the previous step, implies that CoPsy editor will only suggests as possible nodes in the `fpecoreclass` field the EClasses from $MovingEntitiesDSML$. In the bottom part of Fig. 7.5 this result is explicit.

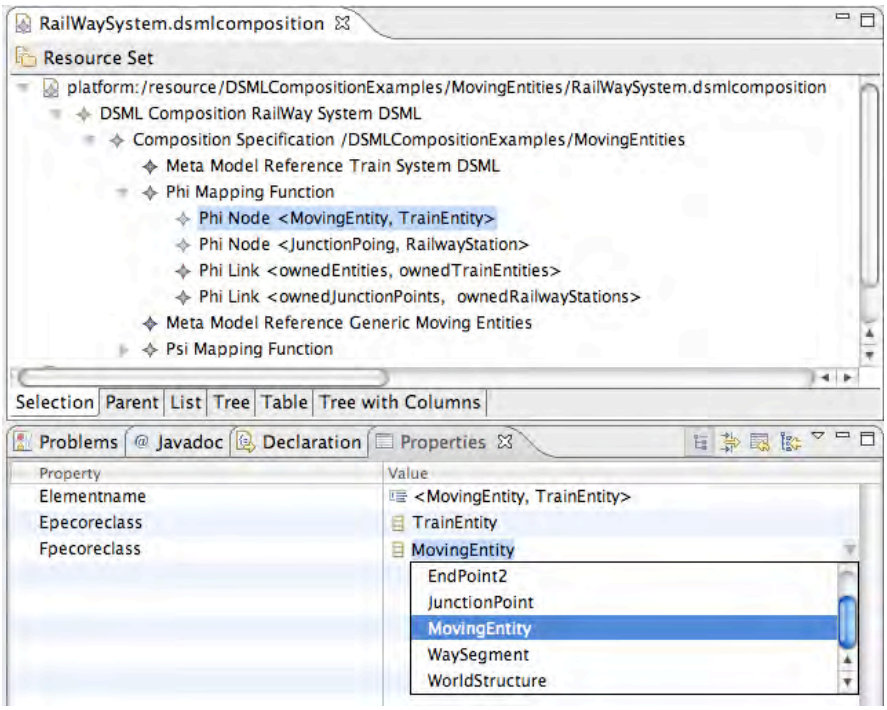


Figure 7.5. CoPsy Specification of φ Mapping Function

By executing this `executemetamodelcomposition` operation defined in the CoPsy metamodel we are going to obtain the metamodel similar to the one in Fig. 6.5. The new and affected elements prior to transformation with a grey background.

7.3.2 Specification of Transformation Composition

The specification of the ψ mapping function follows the specification of the φ mapping function. This time, instead of metamodel elements we are interested in ATL rule elements. We are not going to explain all the details of the

transformations and its composition in this chapter. They are extensively presented in the case study chapter. For the particular case of the Moving Entities example the full description of transformation composition is presented in Sec. 8.1.2. So, without going into details, there are two ψ mapping functions for being able to obtain the Railway System DSML. One relates with the φ element $langleMovingEntity, TrainEntity\rangle$ and the other with the φ element $\langle JunctionPoint, RailwayStation\rangle$.

In the CoPsy environment they are specified as follows:

- ψ_1 as an instance of the `PsiTargetPattern` metaclass with:
  - `elementname` set to $\langle MovingEntities, TrainEntities\rangle$;
  - `eprulename` set to $ruleInit$;
  - `eptargetpatternname` set to $coopnclasstr$;
  - `fprulename` set to $init$;
  - `eptargetpatternname` set to $coopnclassme$;
  - `psielement` reference set to the φ element $langleMovingEntity, TrainEntity\rangle$.
- ψ_2 as an instance of the `PsiTargetPattern` metaclass with:
  - `elementname` set to $\langle JunctionPoint, RailwayStation\rangle$;
  - `eprulename` set to $ruleInit$;
  - `eptargetpatternname` set to $coopnclassstation$;
  - `fprulename` set to $init$;
  - `eptargetpatternname` set to $coopnclassjp$;
  - `psielement` reference set to the φ element $\langle JunctionPoint, RailwayStation\rangle$.

Fig. 7.6 shows the CoPsy specification with emphasis on the specification of ψ elements. The name of the transformation rules are not important at this stage. The are explained in detail in Sec. 8.1.2 when we introduce the ATL transformations for each one of the DSMLs and its composition.

7.3.3 Specification of Models

There are three models of interest when providing a CoPsy specification:

1. A model for the original DSML (before parameterization). This model is represented by the `dsmlModel` attribute in the `DSMLModel` metaclass of the CoPsy metamodel. In

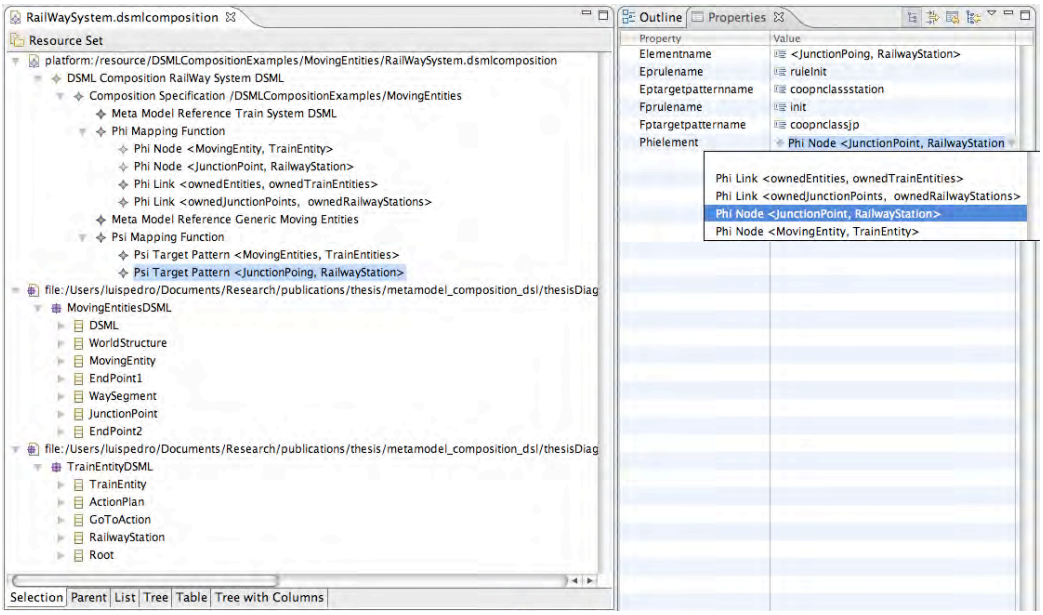


Figure 7.6. CoPsy Specification emphasis in ψ Mapping Function

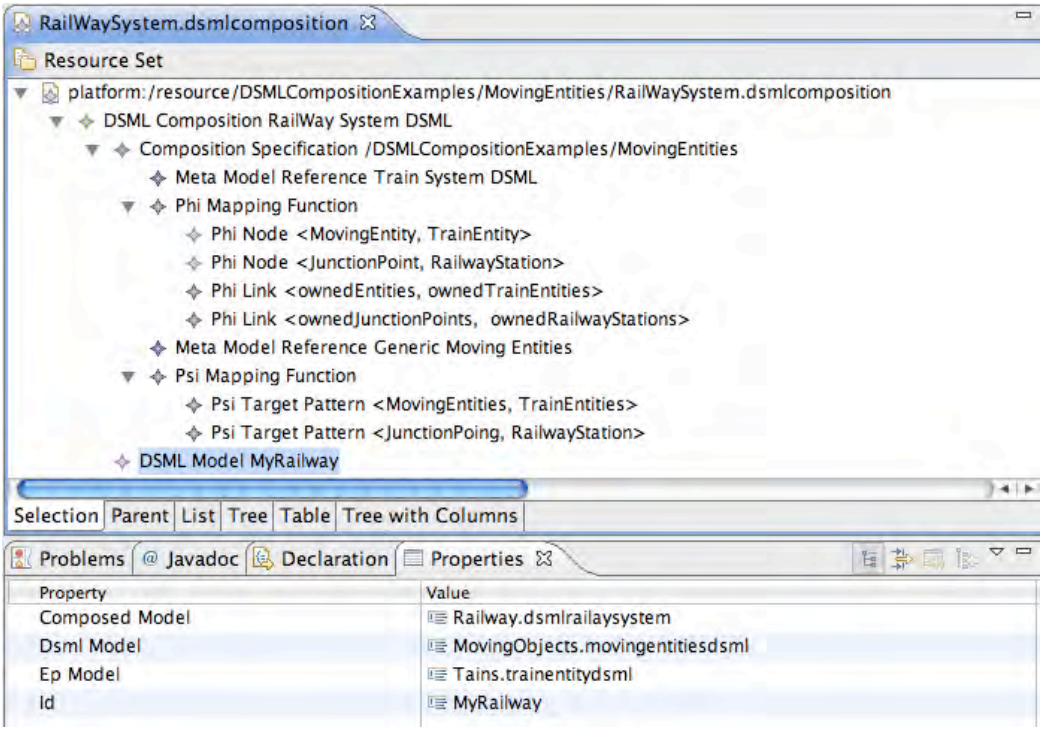


Figure 7.7. Models Definition in a CoPsy Specification

The CoPsy DSML allows to specify several 3-tuples of models. Each one of them creates a environment of application for the execution of model composition and transformation execution. In Fig. 7.7 is possible to see the part of CoPsy editor that allows to define the concerned models.

7.4 CoPsy Behaviour

After defining a CoPsy specification the natural next step its to execute it. As defined in the CoPsy activity diagram in Fig. 7.2 there are four options:

1. to execute metamodel composition;
2. to execute model composition;
3. to execute transformation composition;
4. to execute transformation;

As it is explicit from the activity diagram there is a dependency from one activity to its predecessor. Dispite this fact, it is possible to execute metamodel composition without previously specifying the ψ mapping function.

The next subsection illustrates the behaviour of executing a CoPsy specification using the order privouslly defined.

7.4.1 Composition of Metamodels

The first step to achieve DSML composition is to execute metamodel composition based on the CoPsy specification. The metamodel composition operation takes the DSML base metamodel (the DSML under parameterization) and parameterizes it with the DSML metamodel from the effective parameter *ep*.

The metamodel compostion execution is described by performing the following sequence os activities:

1. Initialize the objects for the DSML base metamodel and the *ep* metamodel;
2. Initialize a new Ecore package according to the `outputMM` parameter in the instance of `CompositionSpecification` of the CoPsy specification;
3. In the CoPsy specification search for both instances of `PhiNode` and `PhiLink` metaclasses;
4. Take all elements of the DSML base metamodel that are not referenced in any of the `PhiElement` instance and clone them to the composed metamodel;
5. Take all elements of the *ep* DSML metamodel defined in the `PhiElement` and clone them to the composed metamodel;

6. For each of those elements get the element to which it is related in the DSML base metamodel. Check for relations that the element in the base DSML metamodel might have and propagate them to the *ep* clone element in the composed metamodel.

Taking into account that a metamodel parameterization is

$$mm' = mm[fp \overset{\varphi}{\leftarrow} ep, F_{fp}]$$

and using the previously defined CoPsy specification for the Railway System we get the resulting metamodel as:

$$DSMLRailwaySystem.ecore =$$

$$MovingEntitiesDSML.ecore[fp \overset{\varphi}{\leftarrow} TrainEntity.ecore, true]$$

with fp and φ defined as in Sec. 7.3.1.

7.4.2 Models Composition

For being able to execute any transformation a metamodel is not enough. So far the composed DSML metamodel is what we have. Starting from here there are two possibilities. Either to create new models that are in conformity with the composed metamodel or to re-use models that were already defined for both DSMLs. The first approach might be the one to be used generally since we expect developers to start developing in the new version of the language. The second is a very useful approach, specially in what concerns re-usability and fast validation/prototyping.

The CoPsy environment supports an automatic composition of models. In the case of the previous example, models in conformity with $TrainEntity$ are composed into models of $MovingEntitiesDSML$. The result are models in conformity with the new $DSMLRailwaySystem$ metamodel.

The `executemodelcomposition` operation defined in the CoPsy metamodel is reponsible for implementing this action.

Although most of the metamodel composition execution can be automatically managed there are some limitation that forces user intervention. In particular the following cases must be resolved by the language engineering :

- When an association/aggregation is parameterized with a different cardinality. If the upper bound of the original cardinality is set to $*$ and the upper bound of the new element is different from $*$, the language engineer is prompted to choose which model elements should be kept and which one should be ignored;

- When an attribute of a metaclass is parameterized with a different cardinality, the same previous behaviour applies.

This procedure allows to provide a coherent conformity relation between the composed models and the metamodel generated from composition.

7.4.3 Transformation Composition Algorithm

Lets first recapitulate the definition of a DSML:

$$DSML = \langle mm', mmt, F, Tr_{mm' \rightarrow mmt} \rangle$$

in which the final transformation $Tr_{mm' \rightarrow mmt}$ is given by

$$Tr_{mm' \rightarrow mmt} = Tr_{mm \rightarrow mmt}[Tr_{fp \rightarrow mmt} \xleftarrow{\varphi,\psi} Tr_{ep \rightarrow mmtp}]$$

The composition algorithm when the `executetransfcomposition` is called from the language engineer. The behaviour of this operation is generically as presented in Sec. 6.6. In this section we present a particular approach for ATL language. Replacing ATL by another transformation language should be a straight forward operation if the transformaiton language is based on a set of transformation rules that are applied using a graph pattern matching approach.

As a summary in what concerns the elements that play a role in the transformation composition:

- The initial DSML is provided with a semantics by applying $Tr_{mm \rightarrow mmt}$;
- The formal parameter is part of the metamodel of the initial DSML. As a consequence also its transformation. The formal parameter transformation is $Tr_{fp \rightarrow mmt}$;
- The effective parameter represents a DSML that will replace the effective parameter. Its transformation is defined as $Tr_{ep \rightarrow mmtp}$;
- The domain of application of the parameterized DSML is the metamodel $mm' = mm[fp \xleftarrow{\varphi} ep, F_{fp}]$.

Thus, the transformation composition algorithm works as follows:

- Update the transformation defined for the initial DSML in order to adjust it to the new source metamodel. All occurrences of mm metamodel are replaced by mm';

- In the transformation defined for the effective parameter, update all occurrences of ep metamodel to mm';
- For all elements in the CoPsy specification of type `PhiElement`, update their references in the $Tr_{mm \rightarrow mmt}$ transformation;
- For parameterized elements that are edges :
  - On the $Tr_{mm \rightarrow mmt}$ transformation search for the rule or target pattern defined by the `PsiElement` in the CoPsy specification. Replace the part corresponding to fp by the part defined by ep;
- For parameterized elements that are non-edge elements:
  - On the $Tr_{mm \rightarrow mmt}$ transformation search for the rule or target pattern defined by the `PsiElement` in the CoPsy specification. Merge the rule defined for the fp part of the `PsiElement` with the rule defined in the ep part of the `PsiElement`. Merging rules implies to merge body of the rules that have the same domain of application;
  - If in the merged rule more than one target pattern is found, keep the one from the effective parameter;
  - Search for transformation patterns that are repeated and eliminate them.

Taking the same exemple as in previous sections we get:

- The initial DSML is provided with a semantics by applying $Tr_{MovingEntitiesDSML \rightarrow COOPN}$;
- The formal parameter is part of the metamodel of the initial DSML. As a consequence also its transformation. The formal parameter transformation is $Tr_{fp \rightarrow COOPN}$. Where fp is a subset of the $MovingEntitiesDSML$ metamodel;
- The effective parameter represents a DSML that will replace the effective parameter. Its transformation is defined as $Tr_{TrainEntity \rightarrow COOPN}$;
- The composed transformation as $Tr_{DSMLRailwaySystem \rightarrow COOPN}$.

Applying this algorithm in the context of the ATL framework we get a transformation $Tr_{DSMLRailwaySystem \rightarrow COOPN}$ that provides the desired semantics in the CO-OPN specification language.

The next and last step is to execute this transformation.

7.4.4 Transformation Execution

The execute transformation operation is the simpliest one in the process of all activities that are available through CoPsy environment. This operation executes the ATL transformation obtained from transformation rules composition.

For the example being used for demonstrating CoPsy functinality, this operation represents the execution of the ATL transformation `RailWaySystem.atl`. This has been defined in for the `outputATLTransformation` attribute in the instance of `CompositionSpecification` as specified in Sec. 7.3.3. From executing this transformation we get the `RailWaySystem.coopnmodel` CO-OPN model.

Chapter 8

Case Studies

This chapter concentrates on the case studies developed during the execution of this work. Three studies are going to be presented:

1. The moving entities example. This is a case study where we incrementally build two DSMLs starting from a generic metamodel. By composition we create two other DSMLs that represent a Railway and a Robot system;

2. A DSML developed for Control Systems (CSs) specifications is presented using the techniques described in this thesis. The CS DSML is presented using an incremental and modular development fashion;

3. Petri Nets Multi Flavours: we start by presenting a generic definition of a Petri Net DSML. Step by step we enhance it with specific concepts that will allow to generate a Coloured Petri Nets (CPN) and Algebraic Petri Nets (APN) DSML prototypes.

8.1 Moving Entities

This case study is base on the description already provided in Sec. 6.2.1. As a summary, this case study is based in the following characteristics:

We want to define a modelling language for the purpose of specifying the simulation of moving entities. This language will be able to be extended in order to implement concepts that suits several domains.

The general concepts involved in the requirements of a DSML of this kind is as having both World Structure and Moving Entities (i.e. trains, cars, etc.).

The World Structure is composed by Junction Points and Way Segments. The Way Segments are the entities responsible to connect Junction Points.

Depending on the domain they are particularised to rails, streets, channels, or any other entity that helps moving entities to go to their destination. Each Way Segment is composed by two end points, each of them are connected to one Junction Point. A constraint is defined that says that it is not possible to connect both end points of a given Way Segment to the same Junction Point.

Junction Points are special locations in the World Structure. Depending on the domain they are, for example, cross-roads, train stations, or any other entity that allows connection of two Way Segments.

Starting with this generic description of a DSML that allows the specification of Way Segments, Junction Points and Moving Entities we can particularise this DSML in two other DSMLs. To each one of them we'll add new information that allows to cope with the semantics of the domain. The two examples chosen were a DSML for a Railway System and another one for a Robot System. The metamodels for the generic DSML, the Train Entity and the Robot Entity are summarised in 8.1. Both Train Entity and Robot Entity metamodels are used to parameterize the generic Moving Entities metamodel in order to obtain the two desired DSMLs.

8.1.1 Generic Moving Entities Transformation

The generic moving entities has a transformation defined using ATL to the CO-OPN language. This transformation is based on the $mmDSML_{gen}$ metamodel as source (see top of Fig. 8.1) and the CO-OPN metamodel as target (presented in Sec. 5.2).

The transformation of this generic DSML is defined as follows:

- The transformation rule for the world structure creates a CO-OPN context. The CO-OPN context represents the interface and the modeling element in which the segments, moving entities and junction points are managed;

- A Way Segment in CO-OPN language is a CO-OPN Class with CO-OPN places representing the end points. For each way segment present in the model, an object instance of this class is also generated in the World Structure context;

- The transformation for a Junction Point is defined as CO-OPN class that allows to map EndPoint1 to EndPoint2. Analogously to the way segment, for each Junction Point element in the specification, an object of this class type in the World Structure context;

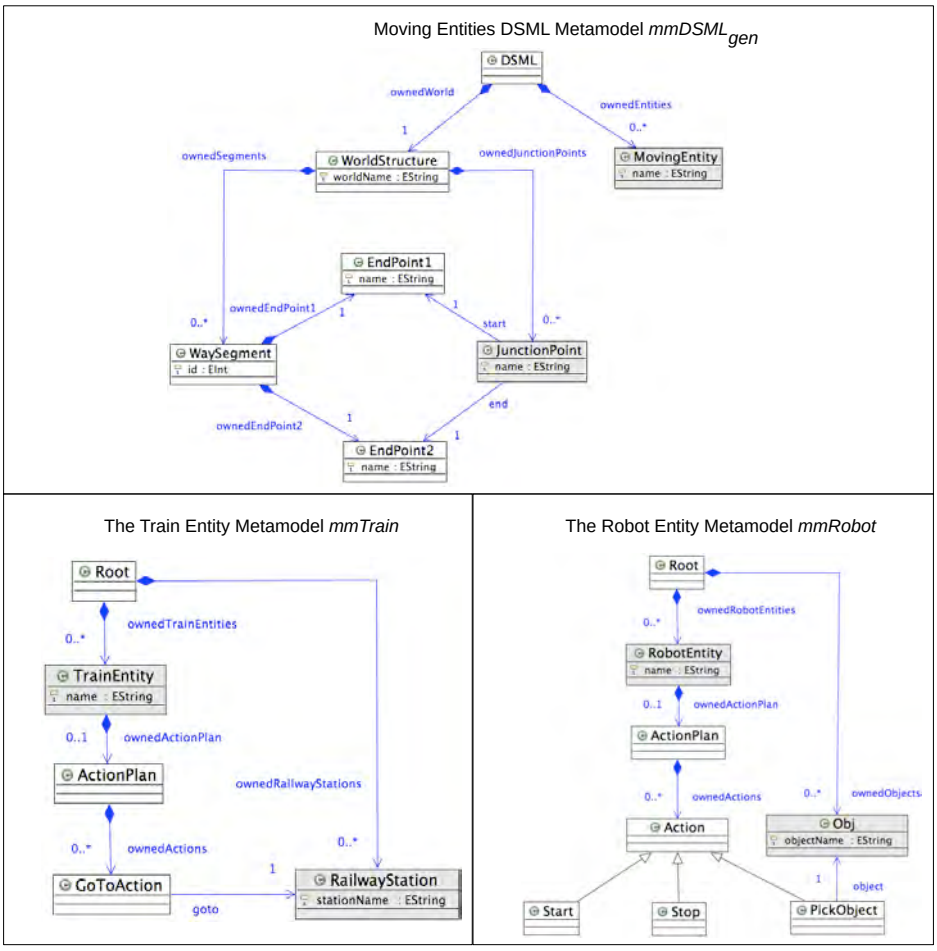


Figure 8.1. Metamodels for the Moving Entities Case Study

- Moving Entity transformation implies the creation of a another CO-OPN class for each one of the moving entities in the model.

Listing 8.1. Fragment of Moving System Transformation to CO-OPN

```
module GenericMovingSystem; -- Module Template
create entitiesCOOPN : COOPNModel from entities : MovingEntitiesDSML;
helper def : createInitAxiom(movingEntities : Set(MovingEntitiesDSML!
    MovingEntities)) : String =
    '( obj' + movingEntities.first().name + '.' + (' setName ' + (
        movingEntities.first().name.toString())) + ')' +
    if (movingEntities.size() = 1) then
      ''
    else
      ' // ' +
      (thisModule.createInitAxiom(movingEntities.excluding(movingEntities.
          first())) )
    endif;
```

```
rule init {
  from
    dsml : MovingEntitiesDSML!DSML
  to
    -- Way Segment --
    coopnclassws : COOPNModel!"COOPNModel::ClassModule::COOPNClass"(
      name <- 'CLASSWaySegment',
      ownedBody <- bodyws,
      ownedInterface <- interfacews
    ),
    bodyws : COOPNModel!"COOPNModel::ClassModule::Body"(
      ownedPlaces <- endpoint1,
      ownedPlaces <- endpoint2,
      ownedPlaces <- namews
    ),
    (...)
    interfacews : COOPNModel!"COOPNModel::ClassModule::Interface"(
      ownedInterfaceClassTypes <- classtypews,
      ownedInterfaceMethods <- setNamews,
      ownedInterfaceMethods <- setEndPoint1,
      ownedInterfaceMethods <- setEndPoint2
    ),
    (...)
    -- Junction Point --
    coopnclassjp : COOPNModel!"COOPNModel::ClassModule::COOPNClass"(
      name <- 'CLASSJunctionPoint',
      ownedBody <- bodyjp,
      ownedInterface <- interfacejp
    ),
    bodyjp : COOPNModel!"COOPNModel::ClassModule::Body"(
      ownedPlaces <- startjp,
      ownedPlaces <- endjp,
      ownedPlaces <- namejp
    ),
    (...)
    interfacejp : COOPNModel!"COOPNModel::ClassModule::Interface"(
      ownedInterfaceClassTypes <- classtypejp,
      ownedInterfaceMethods <- setStartjp,
      ownedInterfaceMethods <- setEndjp
    ),
    (...)
    -- Moving Entity --
    coopnclassme : COOPNModel!"COOPNModel::ClassModule::COOPNClass"(
      name <- 'CLASSMovingEntity',
      ownedBody <- bodyme,
      ownedInterface <- interfaceme
    ),
    bodyme : COOPNModel!"COOPNModel::ClassModule::Body"(
      ownedPlaces <- nameme
    ),
    (...)
    interfaceme : COOPNModel!"COOPNModel::ClassModule::Interface"(
      ownedInterfaceClassTypes <- classtypeme,
      ownedInterfaceMethods <- setNameme
    ),
    (...)
    -- World Structure Context --
    ctx : COOPNModel!"COOPNModel::ContextModule::COOPNContext"(
      name <- 'CTX' + dsml.ownedWorld.worldName,
      ownedBody <- body,
      ownedInterface <- interface
    ),
```

```
    body: COOPNModel!"COOPNModel::ContextModule::Body"(
      ownedObjects <- dsml.ownedWorld.ownedSegments->collect(e | thisModule.
          ruleWaySegmentObj(e)),
      ownedObjects <- dsml.ownedWorld.ownedJunctionPoints->collect(e |
          thisModule.ruleJunctionPointObj(e)),
      ownedObjects <- dsml.ownedEntities->collect(e | thisModule.
          ruleEntitiesObj(e)),
      ownedAxiomTheorems <- axiom

    ),
    (...)
    interface : COOPNModel!"COOPNModel::ContextModule::Interface"(
      ownedInterfaceMethods <- ctxInterfaceMethods
    ),
    (...)
}

lazy rule ruleWaySegmentObj {
  from
    ws : MovingEntitiesDSML!WaySegment
  to
    obj : COOPNModel!"COOPNModel::ContextModule::Object"(
      name <- 'objWaySegment' + ws.id.toString()
    )
}

lazy rule ruleJunctionPointObj {
  from
    jp : MovingEntitiesDSML!JunctionPoint
  to
    obj : COOPNModel!"COOPNModel::ContextModule::Object"(
      name <- 'objJunctionPoint' + jp.name
    )
}

lazy rule ruleEntitiesObj {
  from
    me : MovingEntitiesDSML!MovingEntitiy
  to
    obj : COOPNModel!"COOPNModel::ContextModule::Object"(
      name <- 'obj' + me.name
    )
}
```

The ATL code in listing 8.1 is a fragment of the generic moving entities DSML. For the complete ATL transformation of this DSML please refer to Sec. C.1. This transformation provides the semantics described in the beginning of this section to the generic moving entities DSML. As we can see, the semantics provided is very limited. This is due to the fact that the domain to which it applies is still very general. Albeit this fact, it is already a complete CO-OPN specification that can be simulated and executed.

8.1.2 The Railway System DSML

A railway system is one of the possibilities to particularise the moving entities DSML. In order to achieve the specification of a DSML representing a railway

system, we particularise the previously defined `MovingEntity` in the generic metamodel to the concept of train as depicted at the bottom left side of Fig. 8.1. The Train Entity metamodel is designated by $mmTrain$.

The Train Entity metamodel defines the concept Train as having a structure with an attribute `name` and the behaviour as an Action Plan.

As for the Junction Point, in the context of a Railway system, it is particularised as the concept of Railway Station.

The Action Plan is a sequence of possible GoTo Actions. Informally the behaviour of a GoTo action is to send a train to a given Railway Station.

Railway System Metamodel

As defined previously in Sec. 6.2.1, the metamodel resulting from composition is defined by:

- The formal parameter fp the metamodel corresponding to the `MovingEntity` and `JunctionPoint` elements of $mmDSML_{gen}$;
- $mmTrain$ in 6.4 the effective parameter ep;
- $\varphi = \{\langle MovingEntity, TrainEntity\rangle, \langle JunctionPoint, RailwayStation\rangle\}$

The result of the metamodel parameterization is presented in Fig. 6.5 with the new and affected elements prior to transformation with a grey background. The new metamodel is obtained by applying the metamodel parameterization as:

$$mmDSML_{RailwaySystem} =$$

$$mmDSML_{gen}[MovingEntity \cup JunctionPoint \overset{\varphi}{\leftarrow} mmTrain, true]$$

The *true* F_{fp} means that, for this example, they are allways satisfied. This means the metamodel can be composed without having to take into account particular constraints regarding the formal parameter structure.

The metamodel resulting from the composition can be seen in Fig. 8.2 (diagram on the top of the figure).

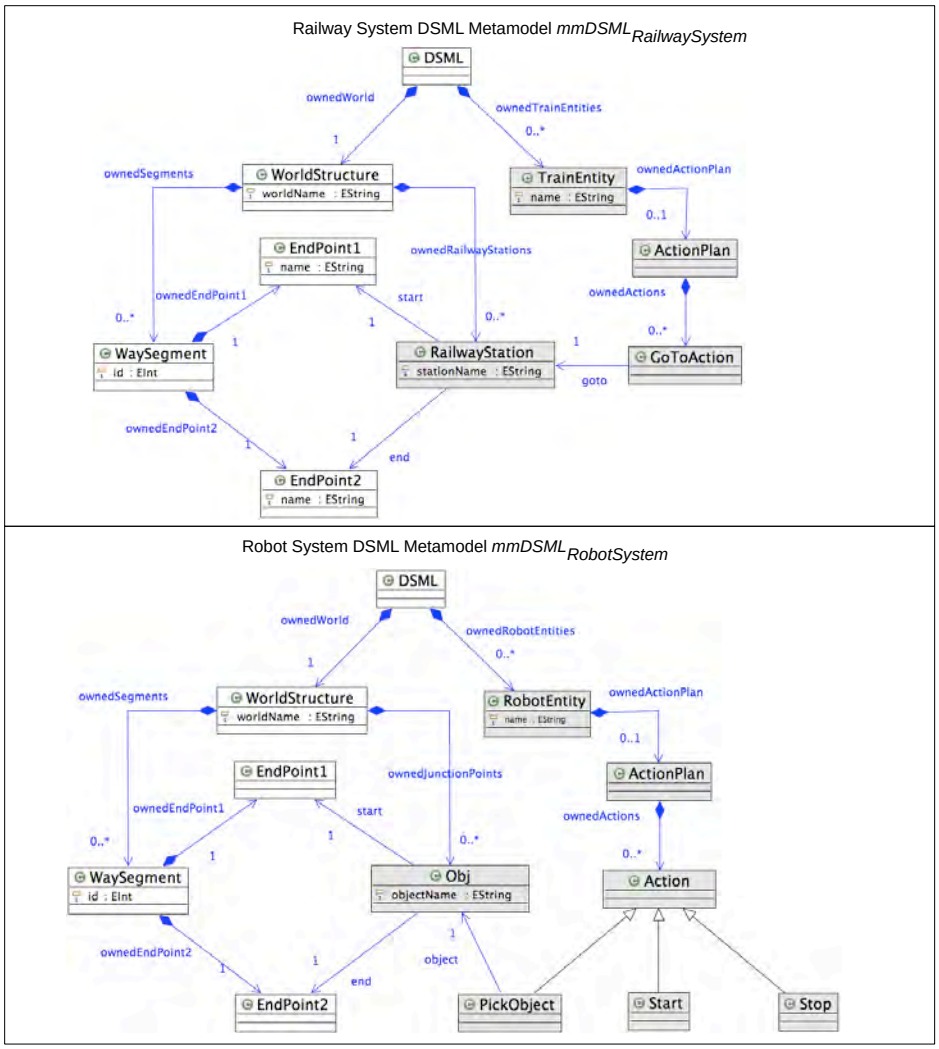


Figure 8.2. Composed Metamodels for the Moving Entities Case Study

Railway System Transformation

First of all we present the transformation that corresponds to the Train Entity DSML. This is the transformation that will be then used to parameterize the Moving Entities DSML transformation in order to get a full transformation for the Railway System.

The Train Entity transformation provides more precise semantics. This semantics is a match to the much more concrete definition of the domain. The composition of Train Entities with the Generic Moving Entities DSMLs allows to narrow the domain of application of the DSML and simultaneously

to provide a richer semantics on that domain. Listing **??** presents a fragment of this transformation highlighting the rules that add more precise semantics to the domain in question. The complete transformation of the Train Entities DSML can be found in Sec. C.2.

Listing 8.2. Fragment of Train Entities Transformation to CO-OPN

```
module TrainEntity; -- Module Template
create  trainsCOOPN : COOPNModel from trains : TrainEntityDSML;
helper  def : createActionPlanAxiom(gotoActions : Set(TrainEntityDSML!
    GoToAction), objname : String) : String =
    (...);
rule ruleInit{
  from
    root : TrainEntityDSML!Root
  to
    -- Railway Station --
    coopnclassstation : COOPNModel!"COOPNModel::ClassModule::COOPNClass"(
      name <- 'CLASSTrainStation',
      ownedBody <- bodystation,
      ownedInterface <- interfacestation
    ),
    (...)
    interfacestation : COOPNModel!"COOPNModel::ClassModule::Interface"(
      ownedInterfaceClassTypes <- classtypestation,
      ownedInterfaceMethods <- methodgoto
    ),
    (...)
    methodgoto : COOPNModel!"COOPNModel::ClassModule::Method"(
      name <- 'goTo _'
    ),
    -- Train Entity  --
    coopnclasstr : COOPNModel!"COOPNModel::ClassModule::COOPNClass"(
      name <- 'CLASSTrain',
      ownedBody <- bodytr,
      ownedInterface <- interfacetr
    ),
    bodytr : COOPNModel!"COOPNModel::ClassModule::Body"(
      ownedPlaces <- locationtr,
      ownedPlaces <- nametr,
      ownedVariables <- vartr,
      ownedAxiomTheorems <-axiomstr
    ),
    (...)
    axiomstr : COOPNModel!"COOPNModel::ClassModule::Axiom"(
      ownedEvent <-eventtr,
      ownedPost <- posttr,
      ownedCondition <- conditiontr
    ),
    (...)
    interfacetr : COOPNModel!"COOPNModel::ClassModule::Interface"(
      ownedInterfaceClassTypes <- classtypetr,
      ownedInterfaceMethods <- setNametr,
      ownedInterfaceMethods <- setLocationtr
    ),
    (...)
    setLocationtr : COOPNModel!"COOPNModel::ClassModule::Method"(
      name <- 'setLocation _'
    ),
    setNametr : COOPNModel!"COOPNModel::ClassModule::Method"(
      name <- 'setName _'
```

```
    ),
    (...)
}

lazy rule executeActionPlanAxiom {
  from
    train : TrainEntityDSML!TrainEntity
  to
    axiom : COOPNModel!"COOPNModel::ContextModule::Axiom"(
      ownedRequiredEvent <- reqevent,
      ownedProvidedEvent <- providedevent,
      ownedCondition <- condition
    ),
    reqevent : COOPNModel!"COOPNModel::ContextModule::RequiredEvent"(
      event <- 'executeActionPlan ' + train.name
    ),
    (...)
    providedeventatom : COOPNModel!"COOPNModel::ContextModule::Event"(
      event <- (.. .)
      thisModule.createActionPlanAxiom(train.ownedActionPlan.ownedActions, '
          obj' + train.name)
    ),
    (...)
    conditionatom : COOPNModel!"COOPNModel::ContextModule::Equation"(
      expression <- '(vartrain = ' + train.name + ') = true'
    )
}

lazy rule ruleTrainEntitiesObj {
  (...)
}

lazy rule ruleRailWayStationObj {
  (...)
}
```

8.1.3 The Robot System DSML

Suppose now that we want a DSML to describe the domain of robot systems. Let us define a simplified robot system. The Robot has no particular way segment to follow, nevertheless the Junction Points can be seen as intermediate Pickable Object.

The sequence of possible actions for our Robot as defined in conformity with the Robot Entity metamodel in Fig. 6.6 is: Start, Pick Object and Stop. The informal semantics associated to these three actions can be described in natural language in the following way:

- Start - to start moving the robot forward in order to reach the Pickable Object and make it disappear once reached. The robot stops immediately after waiting for the next target. If no goal is set the robot does not move;
- Stop - to stop moving the robot;

- Pick Object - this action sets the target (Pickable Object) where the robot should move. In other words the robot gets the reference to the object to pick, rotating a certain angle in order to be facing the object and be able to mode forward in a straight line to find the object when the Start action is called.

The Robot System Metamodel

As already defined in Sec. 6.2.1 the metamodel corresponding to the Robot System $mmDSML_{RobotSystem}$ is the following:

- the formal parameter fp the metamodel corresponding to the `MovingEntity` and `JunctionPoint` elements of $mmDSML_{gen}$;
- the effective parameter ep the metamodel $mmRobot$ in Fig. 6.6;
- $\varphi = \{\langle MovingEntity, RobotEntity\rangle, \langle JunctionPoint, Object\rangle\}$

The Robot System metamodel is obtained by applying:

$$mmDSML_{RobotSystem} =$$

$$mmDSML_{gen}[MovingEntity \cup JunctionPoint \overset{\varphi}{\leftarrow} mmRobot, true]$$

The metamodel resulting from the composition can be seen in Fig. 8.2 (diagram on the bottom of the figure).

The Robot System Transformation

The transformation of the Robot Entity DSML gives the previously stated semantics of this DSML in the CO-OPN formalism. The complete transformation of the Robot Entity that will parameterize the Generic Moving Entities DSML is presented in Sec. C.4. Taking into account that there are a lot of similarities between the transformation of the Robot Entity DSML and the Train Entity DSML we'll not go into all the details of it. The procedure for the transformation composition is also very similar to the one used in the Rail Way System DSML. Instead of providing an exhaustive list of the ATL transformation rules of the Robot Entity as for the Train Entity we focus only in some parts that allows to provide the semantics described previously.

The Listing 8.3 presents the transformation that gives semantics to the `Start` and `Stop` actions. The axioms for the execution of these events have different pre and post conditions. The transformation shows that the `Start`

action can only be executed when the Robot is on `stopped` state. On the other hand the `Stop` acction can only be sucessfull if the robot is on state `moving`.

Listing 8.3. Robot Entities `Start` and `Stop` Semantics

```
rule ruleInit{
  from
    root : RobotEntityDSML!Root
  to
    (...)
    bodyrobot : COOPNModel!"COOPNModel::ClassModule::Body"(
      (...)
      ownedAxiomTheorems <-axiomsrobotstart,
      ownedAxiomTheorems <-axiomsrobotstop
    ),
    (...)
    axiomsrobotstart : COOPNModel!"COOPNModel::ClassModule::Axiom"(
      name <- 'start',
      ownedEvent <-eventrobotstart,
      ownedPost <- postrobotstart,
      ownedPre <- prerobotstart,
      ownedCondition <- conditionrobotstart
    ),
    axiomsrobotstop : COOPNModel!"COOPNModel::ClassModule::Axiom"(
      name <- 'stop',
      ownedEvent <- eventrobotstop,
      ownedPost <- postrobotstop,
      ownedPre <- prerobotstop,
      ownedCondition <- conditionrobotstop
    ),
    (...)
    eventrobotstart : COOPNModel!"COOPNModel::ClassModule::Event"(
      event <- 'start'
    ),
    postrobotstart : COOPNModel!"COOPNModel::ClassModule::Post"(
      ownedPostTerm <- postTermrobotstart
    ),
    postTermrobotstart : COOPNModel!"COOPNModel::ClassModule::Term"(
      expression <-  'state moving'
    ),
    prerobotstart : COOPNModel!"COOPNModel::ClassModule::Pre"(
      ownedPreTerm <- preTermrobotstart
    ),
    preTermrobotstart : COOPNModel!"COOPNModel::ClassModule::Term"(
      expression <-  'state stopped'
    ),
    conditionrobotstart : COOPNModel!"COOPNModel::ClassModule::Condition"(
      ownedConditionAtom <-   conditionatomrobotstart
    ),
    conditionatomrobotstart:COOPNModel!"COOPNModel::ClassModule::Equation"(
      expression <- '(this = Self) = true'
    ),
    eventrobotstop : COOPNModel!"COOPNModel::ClassModule::Event"(
      event <- 'stop'
    ),
    postrobotstop : COOPNModel!"COOPNModel::ClassModule::Post"(
      ownedPostTerm <- postTermrobotstop
    ),
    postTermrobotstop : COOPNModel!"COOPNModel::ClassModule::Term"(
      expression <-  'state stopped'
```

```
    ),
    prerobotstop : COOPNModel!"COOPNModel::ClassModule::Pre"(
      ownedPreTerm <- preTermrobotstop
    ),
    preTermrobotstop : COOPNModel!"COOPNModel::ClassModule::Term"(
      expression <- 'state moving'
    ),
    (...)
```

The transformation for providing semantics for the action plan execution is also different from the Train Entity DSML. In the Train Entity example the action plan execution was simply a sequence of `goTo` actions. In the Robot Entity DSML there are three types of actions. Listing 8.4.

Listing 8.4. Robot Entities Action Plan Transformation

```
helper def : createActionPlanAxiom(gotoActions : Set(RobotEntityDSML!
    GoToAction), objname : String) : String =
    if (gotoActions.first().oclIsTypeOf(RobotEntityDSML!PickObject)) then
      '(' + objname + '.' + (' pickObject ' + (gotoActions.first().object.
          objectName.toString())) + ')' +
      if (gotoActions.size() = 1) then
      ''
      else
        ' .. ' +
        (thisModule.createActionPlanAxiom(gotoActions.excluding(gotoActions.
            first()), objname ))
      endif
    else if (gotoActions.first().oclIsTypeOf(RobotEntityDSML!Start)) then
      '(' + objname + '. start' + ')' +
      if (gotoActions.size() = 1) then
      ''
      else
        ' .. ' +
        (thisModule.createActionPlanAxiom(gotoActions.excluding(gotoActions.
            first()), objname ))
      endif
    else if (gotoActions.first().oclIsTypeOf(RobotEntityDSML!Stop)) then
      '(' + objname + '. stop' + ')' +
      if (gotoActions.size() = 1) then
      ''
      else
        ' .. ' +
        (thisModule.createActionPlanAxiom(gotoActions.excluding(gotoActions.
            first()), objname ))
      endif
    else
      ''
    endif
    endif
    endif;
```

As we can see from the ATL transformation on Listing 8.4 the `executeActionPlan` axiom is a sequence of `Start`, `Stop` and `PickObject` actions.

The composition of the Robot Entity DSML with the Generic Moving Entities DSML is presented in Sec. C.5 following the transfomation composition definition in Sec. 6.6.

The Fig. 8.3 provides an example of a specification of a Robot Entity model and its equivalent in CO-OPN after executing the transformation. The Robot Entity model is defined by two objects (`Bottle` and `Apple`), two robots (`Robot1` and `Robot2` and its respective action plans. The right side of Fig. 8.3 shows the CO-OPN specification of the Moving Entity model highlighting the CO-OPN instances of `CLASSRobot` for each one of the Robots instances of the `CLASSObject` for each one of the objects defined in the Moving Entity model. It is also possible to see the sequence of events for the definition of the `executActionPlan` method.

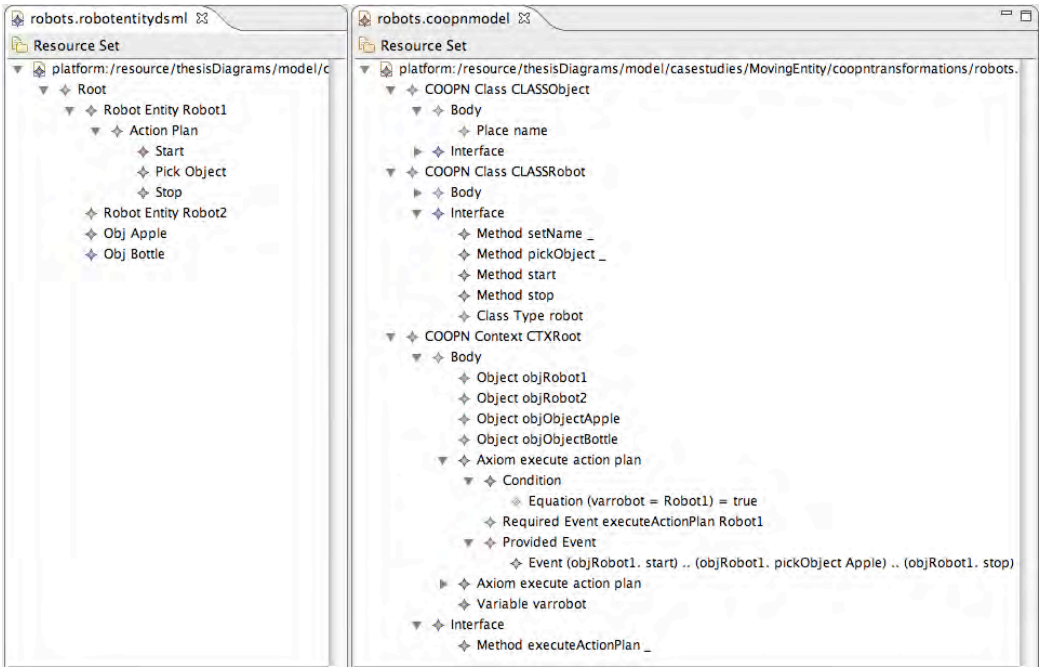


Figure 8.3. Robot Entity model and Transformation to CO-OPN

8.2 Control Systems DSML

Cospel stands for COntrol systems SPEcification Language (Cospel). This language is based on the work in DSML development for control systems, the BATIC^{3}S project(SMV Group, 2008). A language named *Cospel*(Risoldi & Amaral, 2007; Risoldi & Buchs, 2007; Risoldi et al., 2009, to appear) has been designed to specify user interfaces for complex control systems using domain abstractions. The language includes structure, behaviour and communication aspects of a system, as well as interface-related features like user and task

models. An associated framework transforms the language into an executable system simulator and a user interface prototype. Due to the several aspects related to control systems that the Cospel language models, the structure of its metamodel is rather modular.

In this section we will analyse some of its features, and for each one of them to:

- treat them as individual metamodels;
- identifying transformation for each one of them;
- show how these metamodels (and respective transformation models) are composed into a more complex language.

This can be seen as a demonstration of how this methodology can be used by language designers to add features to a core language in a relatively agile way, hence maximizing reusability of partial models.

8.2.1 Generic Cospel DSML

Fig. 8.4 shows the metamodel of a Generic Cospel DSML.

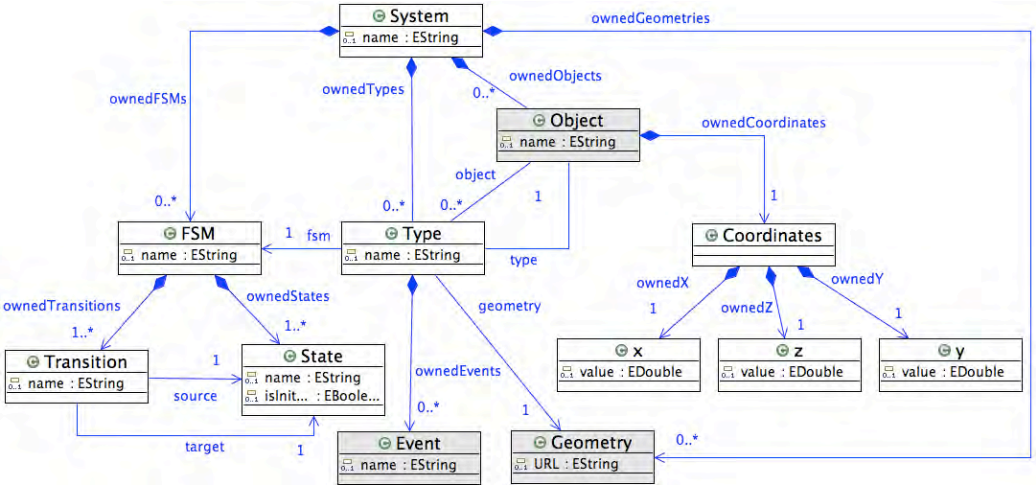


Figure 8.4. Generic Cospel Metamodel: $mmCospel_{gen}$

This is the metamodel that acts as the starting point for a modular creation of the complete control systems DSML. It is the core structure of the language and provides all the abstractions to define a simple control system. It includes the concepts of:

- `Object`

- `Type`
- `Finite State Machine`
- `Event`
- `Geometry`
- `Coordinates`

Despite the fact that all the concepts needed for defining the Cospel DSML are represented in this first metamodel, some of them (e.g. `Geometry`, `Coordinates`, `Event`) do not provide enough granularity in order to associate them with the desired semantics. The main goal of this section is then to show how to build a Cospel DSML prototype by incrementally adding details in a sequence of compositions of metamodels and transformations.

Transforming Generic Cospel models to CO-OPN models is done as follows, without going into too much detail of the CO-OPN code.

`Types` are transformed into CO-OPN `Classes`. `States` and `Transitions` of the associated FSMs become respectively `places` and `methods` moving tokens among these places. `Events` are also transformed into methods in the CO-OPN Classes. Associated `geometry` is transformed into a place containing the URL of the geometry data. The skeleton of the transformation rule for Types is shown in Listing 8.5. Rules for the other classes follow a similar schema.

Note that rules include a source model which is also a CO-OPN model; this model contains instances of CO-OPN algebraic data types which are mapped to data types of created methods. This model is only read, never modified; its presence however forces us to make lazy rules and call them explicitly, rather than relying on matching only.

Listing 8.5. Excerpt of GenericCospel Type element to CO-OPN

```
1  lazy rule ruleType {
2    from
3      c : COOPNMetaModel!COOPNPackage,
4      t : GenericCospel!Type
5    to
6      coopnclass: COOPNMetaModel!"COOPNMetamodel::ClassModule::COOPNClass"(
7    name <- t.name,
8      ownedBody <- body,
9      ownedInterface <- interface
10   ),
11   (...)
```

`Objects` are transformed into CO-OPN `Contexts` which instantiate the classes of their associated type. `Coordinates` of Objects are transformed into places in the classes. The skeleton rule for Objects is presented in Listing 8.6

Listing 8.6. Skeleton Rule for GenericCospel Objects' Transformation

```
1  lazy rule ruleObject {
2    from
3      obj : GenericCospel!Object,
4      c : COOPNMetaModel!COOPNPackage
5    to
6      ctx : COOPNMetaModel!"COOPNMetamodel::ContextModule::COOPNContext"(
7      name <- 'CTX' + obj.name,
8      ownedBody <- body,
9      ownedInterface <- interface
10   ),
11   (...)
```

8.2.2 Event Model extension of Cospel

The first modification we present is the refinement of the `Event` concept. In the $mmCospel_{gen}$ metamodel, the Events were associated to Types. However, they only had a name, and were not characterized by any reactive behavior. The goal for an event model would be to have events triggering transitions or other events. In this perspective, we built an Event metamodel *mmEvent* (Fig. 8.5) in which an `Event` can have several `Conditions`, representing constraints of pre- and post-conditions. Each condition is associated to a `Transition`, and/or to another `Event`. The idea of this association is to have a trigger: an Event, when satisfying certain pre- and post-conditions, can trigger a Transition, and/or it can trigger another Event. The `Root` element is only used to allow creation of objects of the Transition and Event class when modeling and is irrelevant for the composition.

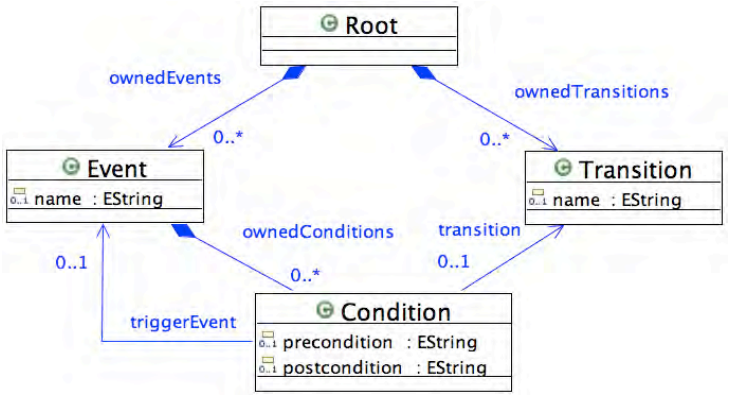


Figure 8.5. mmEvent Cospel Metamodel

With the *mmEvent* metamodel by itself, we can build models where we declare events and their behavior. Transformation of these models to CO-OPN is relatively simple: an `event` becomes a CO-OPN `method` declaration;

for each of its `conditions`, an `axiom` is created, synchronizing the method with the corresponding transition and/or event.

Composing this metamodel with the $mmCospel_{gen}$ one is done by substituting the `Event` class in $mmCospel_{gen}$ with the `Event` class in $mmEvent$. The `Conditions` class and all related associations are also part of the ep. Using the definitions presented in sec. 6.1 we can express this substitution as follows.

Having $mmEvent$ the metamodel in Fig. 8.5 and $mmCospel_{gen}$ the Generic Cospel metamodel of Fig. 8.4, the metamodel of the new DSML with the Event extension is given by the following parameterization:

- fp the metamodel corresponding to the `Event` of $mmCospel_{gen}$;
- the effective parameter ep a set of elements $\{Event, Condition, ownedConditions, transition, triggerEvent\}$ from $mmEvent$;
- $\varphi = \{\langle Event, Event\rangle\}$

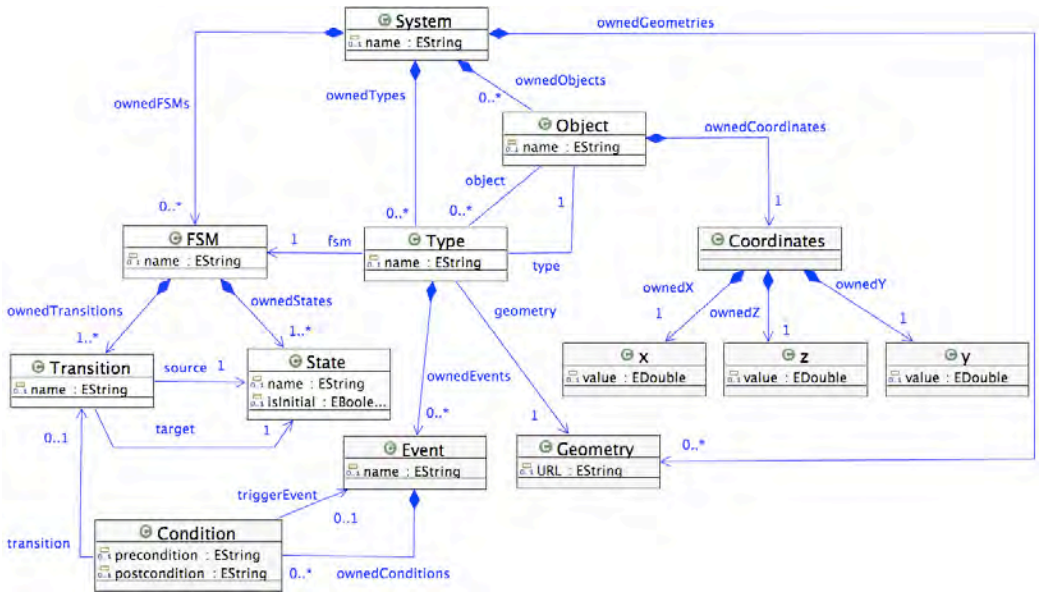


Figure 8.6. Event Extension of Cospel: $mmCospel_{event}$ metamodel

The result is a new metamodel $mmCospel_{event}$ resulting from the application of:

$$mmCospel_{event} = mmCospel_{gen}[fp \overset{\varphi}{\leftarrow} ep, F_{fp}]$$

and is presented in Fig. 8.6 with the new elements, and those affected by the transformation, in a grey background. F_{fp} constraints are empty in this example.

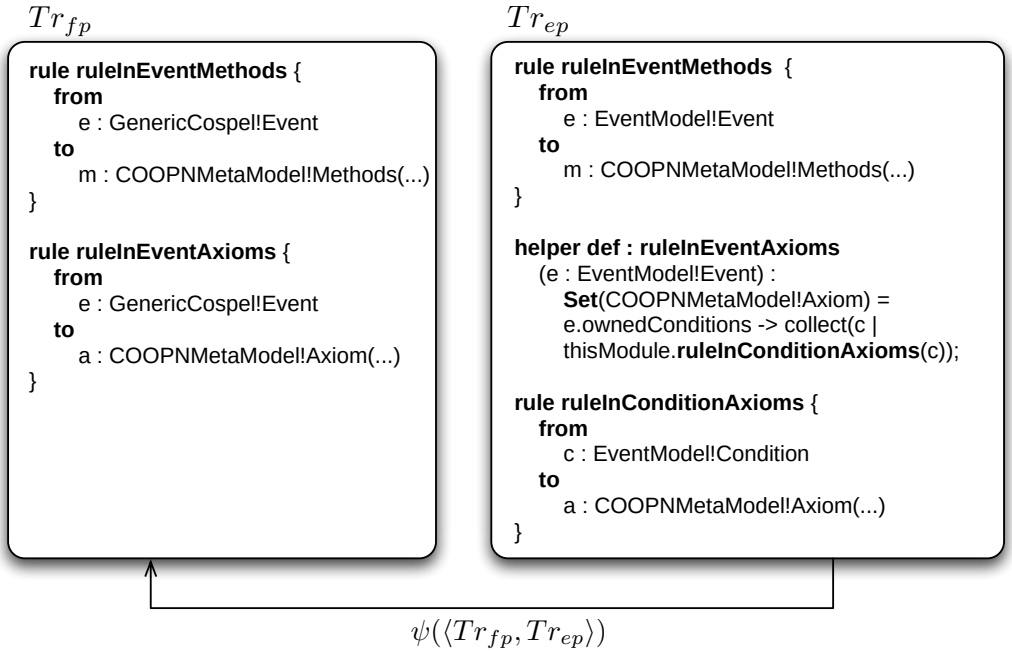


Figure 8.7. Transformation substitution for the Event model extension

For the transformation composition, rules for the formal parameter, Tr_{fp}, are replaced by those for the effective parameter, Tr_{ep}. No restrictions on Tr_{fp} and Tr_{ep} are specified, i.e., the whole Tr_{fp} is substituted by the whole Tr_{ep}. The transformation composition is shown in Fig. 8.7 and is the result of applying the definition presented in Sec. 6.1.

Having $ie \in \mathbf{IM}|_{\mathbf{mmCospel_{event}}}$ the models that are in conformity with the $mmCospel_{event}$ metamodel, it the models in the target language(s), the application of transformation composition is defined as:

$$it = Tr_{mmCospel_{event}}[Tr_{fp} \xleftarrow{\varphi,\psi} Tr_{mmEvent}](ie)$$

In this particular implementation, the system execution state ctx as defined in Sec. 6.1 is not explicitly mentioned, as ATL manages it automatically.

With reference to Figg. 6.2 and 6.10, we can see in Fig. 8.8 how the $mmEvent$ and $mmCospel_{gen}$ metamodels are composed into the $mmCospel_{Event}$ metamodel, and how the composed transformation $Tr_{mmCospel_{event}}$ includes the transformation rules from $Tr_{mmEvent}$.

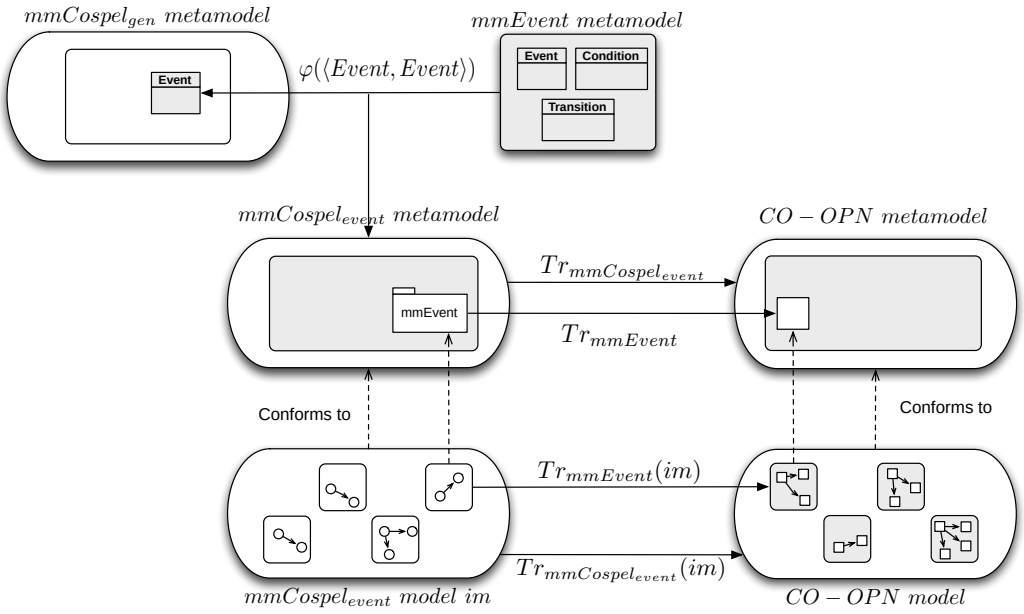


Figure 8.8. Parameterization of Transformations for the Event model extension

8.2.3 Hierarchical Model Extension of Cospel

Objects in a system have thus far only been represented as a collection with no particular structure. In real control systems however, a desired feature is to have hierarchies of objects, as this generally corresponds to the way systems are built. To achieve this, we introduce a Hierarchy metamodel $mmHierarchy$, in which the `Object` class has a `children` association to itself (0-* cardinality), as well as an opposite `parent` association (0-1). Transforming this to CO-OPN gives, in the CO-OPN Context of a "parent object", the references to the CO-OPN Contexts of the "children objects". The metamodel expressing this description is given by Fig. 8.9.

Composing this metamodel with the result from the previous composition is done by substituting the `Object` class in the previous model with the one in this hierarchical model. Having $mmHierarchy$ the metamodel introducing hierarchy, and $mmCospel_{event}$ the result of the previous composition, the metamodel of the enriched DSML with the Hierarchy extension is defined by:

- fp the metamodel of the `Object` class of $mmCospel_{event}$;
- the effective parameter ep a set of elements $\{Object, children, parent\}$ from $mmHierarchy$;
- $\varphi = \{\langle Object, Object\rangle\}$

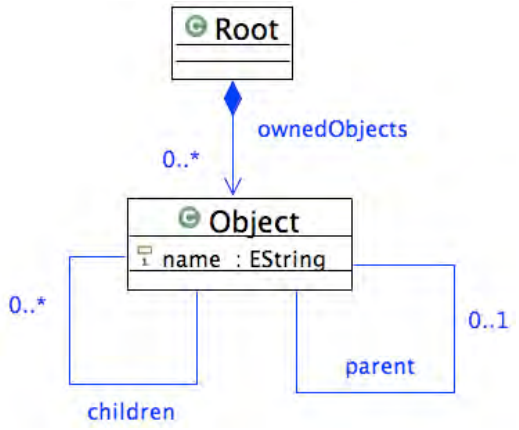


Figure 8.9. mmHierarchy Cospel Metamodel

For what concerns transformation composition, the previously defined transformation rule for objects, `ruleObject` (see 8.2.1) must not change in this case, or we would lose information concerning the associations of `Object` which are present in $mmCospel_{event}$ but not in $mmHierarchy$; rather, a new rule
`ruleObjectWithChildren` is added which extends `ruleObject` by specifying a more specialized transformation for objects with children. In defining the ψ function for this composition, instead of using the whole Tr_{ep} as a parameter, we use

$$(Tr_{ep} - TE) \cup (Tr_{fp}|TF)$$

where TE is a subset of Tr_{ep} formed by all Tr^i_{ep} such that $\exists Tr^j_{fp}$: $Dom(Tr^j_{fp}) = \varphi(Dom(Tr^i_{ep}))$ for any i, j (in other terms, all rules in Tr_{ep} who have a corresponding rule in Tr_{fp} with the corresponding domain after parameterization); TF is a subset of Tr_{fp} formed by all the Tr^j_{fp} as just defined; $(Tr_{ep} - TE)$ is the rules in Tr_{ep} minus those in TE; and $(Tr_{fp}|TF)$ is the subset of Tr_{fp} including only the rules in TF.

The transformation composition is shown in Fig. 8.10. Rules in black text are those which will appear in the result.

8.2.4 Geometry Model extension of Cospel

In this section of the case study it is illustrated the fact that the approach used is not tied to a single formalism. In addition to the transformations to the CO-OPN, SQL will be also used as a target language. SQL is used to

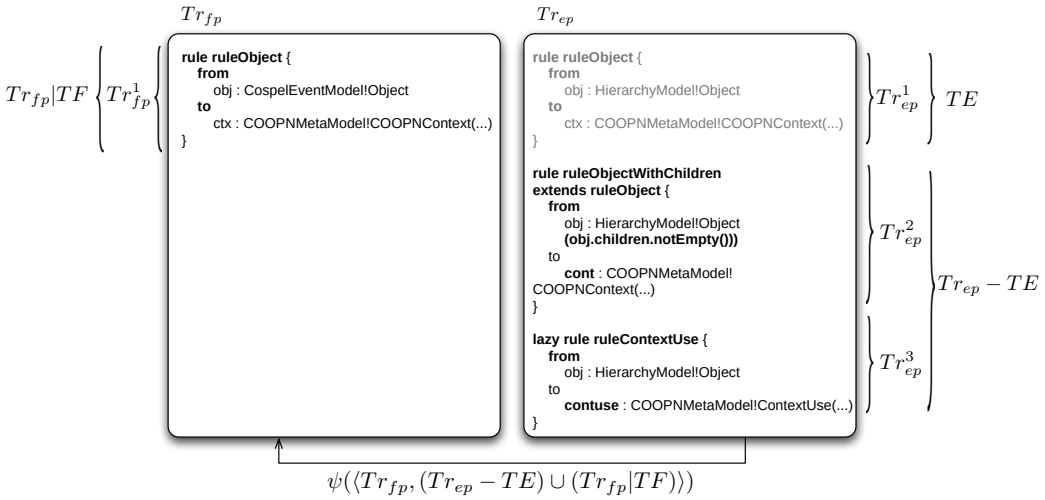


Figure 8.10. Transformation substitution for the Hierarchical model extension

transform the module of the Cospel language that needs to be stored in a relational database.

In the Generic Cospel metamodel (and subsequent compositions), the `Geometry` class represents only a link to a file location, containing geometrical data (vertexes and faces) of a Type. A useful feature for modeling physical control systems is to have available a certain number of geometrical primitive shapes, which can be parametrized quickly, instead of having to deal with vertexes and faces. For example, a cylinder can be defined by its height, an outer radius, and an inner radius (for empty cylinders, or tubes).

We design a simple metamodel which achieves this by making the `Geometry` class abstract and having several classes (`Box`, `Sphere`, `Cylinder` and `GeomFile`) implement it. The metamodel for this Cospel's module is shown in 8.11.

The transformation of this metamodel is interesting in that in the framework of the BATIC^{3}S project this geometrical information is not stored in the CO-OPN model, but rather in a database (used then by a user interface prototype to load a 3D scene).

This metamodel is thus transformed into a set of SQL queries (using a simplified SQL metamodel, called *sql4Cospel*, with Strings representing queries). For each geometry, an INSERT query is made in the appropriate table (according to the kind of primitive), and an association with objects of that type is established in another table.

Composing this metamodel with the previous ones is a matter of substituting the previous `Geometry` class with the new abstract `Geometry` class (the `Box`, `Sphere`, `Cylinder` and `GeomFile` classes follow). Having $mmGeometry$ the metamodel refining geometry, and $mmCospel_{eventHierarchy}$ the result of

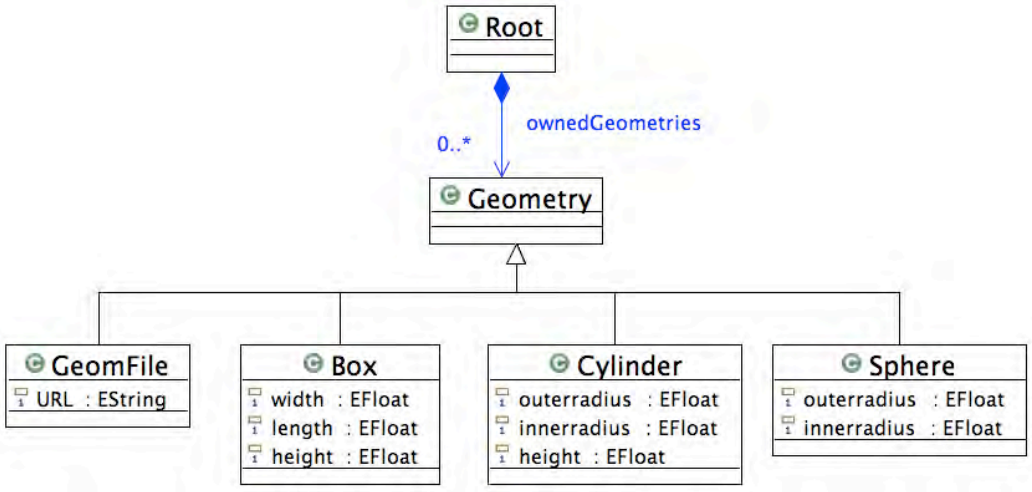


Figure 8.11. mmGeometry Cospel Metamodel

the previous compositions, the metamodel of the new DSML with the Geometry extension is defined by:

- fp the metamodel of the `Geometry` class of $mmCospel_{eventHierarchy}$;
- the effective parameter ep a set of classes $\{Geometry, GeomFile, Box, Cylinder, Sphere\}$ from $mmGeometry$;
- $\varphi = \{\langle Geometry, Geometry\rangle\}$

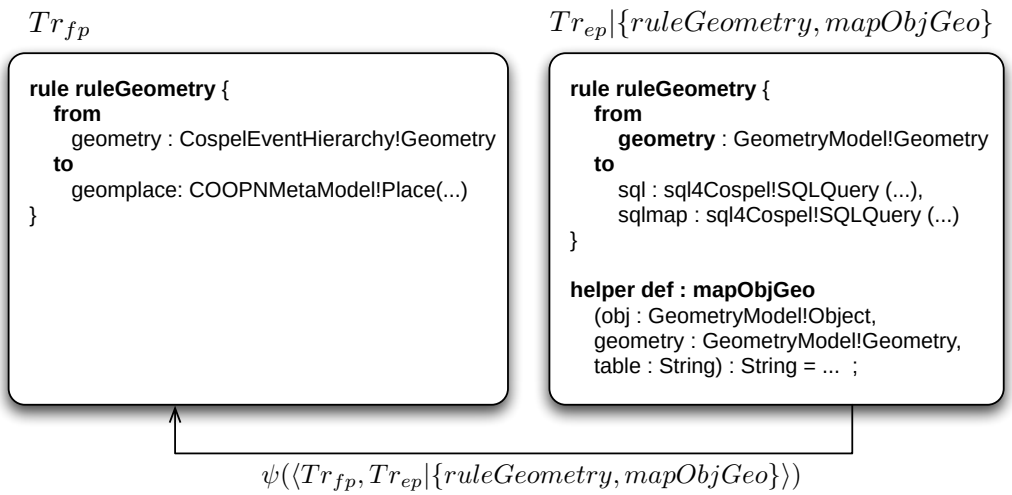


Figure 8.12. Transformation composition for the Geometry model extension

For the transformation composition in this case, the codomain of Tr_{fp} and Tr_{ep} is different: Tr_{fp} creates models conforming to the CO-OPN Meta-Model, while Tr_{ep} creates models conforming to *sql4Cospel*. The composition is shown in Fig. 8.12.

The final metamodel resulting from the discussed compositions is shown in Fig. 8.13, where classes highlighted in grey are the ones who underwent substitution or have been created anew.

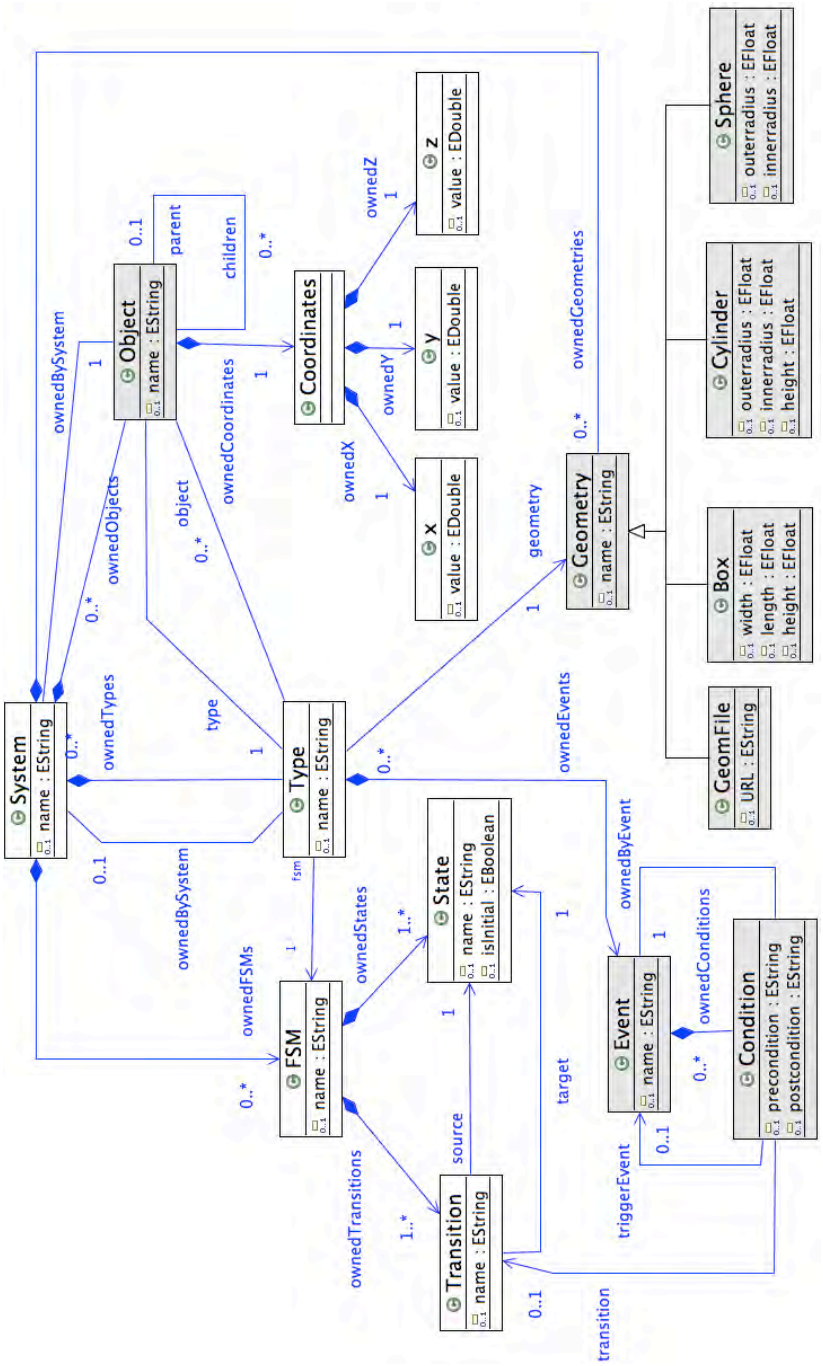


Figure 8.13. Full Cospel Metamodel $mmCospel_{full}$

8.3 Multi-Flavours Petri Nets

This example fucuses on defining Petri Nets DSMLs for several versions of this formalism. This is probably the example with less interest in what concerns the methodology of composition and parameterization. This is mostly due to the fact that CO-OPN is a Petri Nets based formalism and we are proposing to develop DSMLs that are themselves variations of the Petri Nets formalism. Rather than providing the most interesting example on what concerns the composition framework, this example allows to show that the method presented in this thesis allows to provide semantics to a set of Petri Nets formalisms starting from a generic representation of it. Instead of starting from the Standard Petri Nets formalism and to extend it we define a Generi Petri Nets formalism and then parameterize it to obtain the desired concrete Petri Nets formalism variation.

8.3.1 Generic Petri Nets

The Fig. 8.14 shows the metamodel of the Generic Petri Nets metamodel *mmGenPN*. This is the starting point for us to incrementally specify three Petri Nets formalisms.

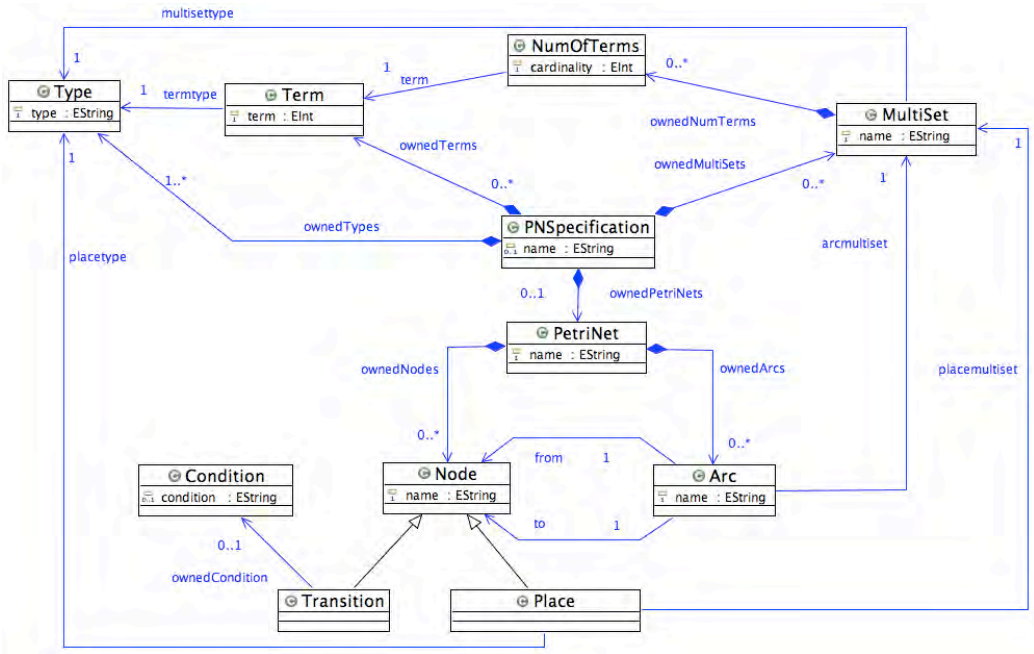


Figure 8.14. Generic Petri Nets Metamodel: mmGenPN

The metamodel of the Generic Petri Nets contains the notions of:

PNSpecification The top node element of the metamodel. Is used to aggregate the other constructions of a Generic Petri Nets Specification;

PetriNet Represents the root element for the elements of type `Node` and `Arc`;

Place, Transition and Arc : Represent the standard elements of a Petri Net. Places are allways typed and a Transition is composed by a Condition;

MultiSet : The construction that allows to create expressions associated to Arcs and Places;

Term : The element that represent an value entity (variable or constant) used for some particular 'thing'. Terms have a type;

Type : By instantiating this metamodel element it is possible to provide a type to Terms, MultiSets and Places;

Condition : This metamodel element allows to specify a condition that must be evaluated to *true* at the moment of transition fire.

The transformation of Generic Petri Nets to CO-OPN formalism is quite strait forward.

- The `PetriNet` element is transformed to a `CO-OPN Class`;
- `Place` is transformaed to `CO-OPN Places` in the `CO-OPN Class`;
- `Transition, Arc` and `Condition` are used to form `CO-OPN Axioms`;
- `Multiset` and `Term` are transformed into marking of the `CO-OPN Places` when related with places. And transformed into values of the pre and post conditions of the `Axioms` when related with the `Arc` element of the metamodel;
- Instances of the `Type` element are transformed into `CO-OPN ADTs` with a corresponding `Sort` name.

8.3.2 Standard Petri Nets

The Standard Petri Nets DSML is obtained by parameterizing the Generic Petri Nets DSML using the DSML which metamodel is presented in Fig. 8.15.

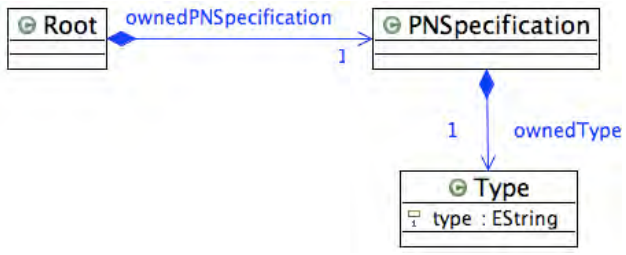


Figure 8.15. Standard Petri Nets Extension Parameter: mmPNStandard

This DSML provides more specific information in order to obtain the semantics of a Petri Nets without extension. The *mmPNStandard* metamodel in Fig. 8.15 is used to parameterize the *mmGenPN* Generic Petri Nets metamodel by defining:

- The formal parameter fp the metamodel corresponding to the `PNSpecification`, `Type`, `ownedTypes` and `Condition` elements of $mmGenPN$;
- $mmPNStandard$ in 8.15 the effective parameter ep;
- $\varphi = \{\langle PNSpecification, PNSpecification\rangle,$
  $\langle ownedTypes, ownedType\rangle,$
  $\langle Condition, \emptyset\rangle\}$

The $\langle Condition, \emptyset\rangle$ part of the φ specification means that the `Condition` element is removed from the metamodel. Transformations corresponding to the `Condition` element will also be removed from the final transformation.

The result of the metamodel parameterization is presented in Fig. 8.16 with the new and affected elements with a grey background. The new metamodel is obtained by applying the metamodel parameterization as:

$$mmStandardPN =$$
$$mmGenPN[PNSpecification \cup Type \cup ownedTypes$$
$$\cup\ Condition \overset{\varphi}{\leftarrow} mmPNStandard, true]$$

The metamodel resulting from the composition can be seen in Fig. 8.16.

In what concerns transformation composition the most relevant differences between the transformation of the Generic Petri Nets DSML and the Standard Petri Nets DSML are:

- The rules regarding the transformation of the `Condition` element are ignored. They are not part of the final transformation for the Standard Petri Nets DSML;

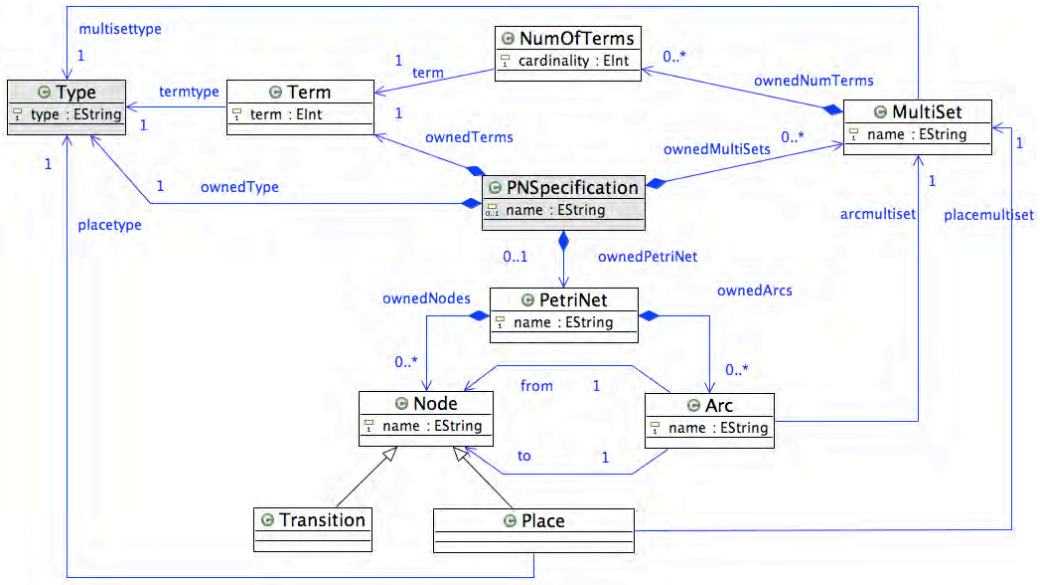


Figure 8.16. Standard Petri Nets Metamodel: mmStandardPN

- The transformation of the `Type` element is replaced by the transformation defined in the effective parameter. In the Standard Petri Nets formalism there is only one data type. It is generally known as `Black Token`. The CO-OPN environment provides a set of pre-defined ADTs ready to use. One of them provides the semantics of a Black Token. By using CO-OPN inheritance features the transformation of the `Type` element in the effective parameter defines an ADT that inherits from the CO-OPN BlackToken ADT.

The Fig. 8.17 shows an excerpt of the transformation composition in order to obtain the Standard Petri Nets DSML.

8.3.3 Colored Petri Nets

The Colored Petri Nets formalism adds typing functionalities to the Standard Petri Nets. The data types in the Colored Petri Nets formalism are usually known as colors. Each colour is a different data type. However, the Colored Petri Nets formalism does not defines how to provide semantics to each colour. It is common approach in the Petri Nets community to define a set of *know colors* that have a precise semantics.

In order to obtain a DSML that represents the Colored Petri Nets formalism we use a Petri Net colour DSML. The metamodel *mmPNColor* for this DSML is presented in Fig. 8.18 and will act as the effective parameter

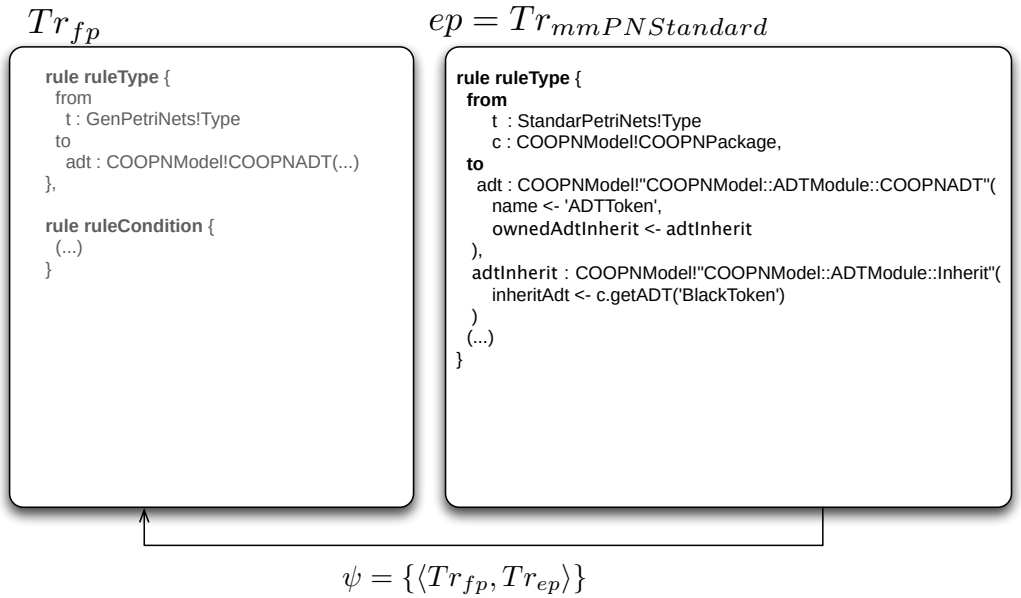


Figure 8.17. Standard Petri Nets Transformation Substitution Excerpt

for parameterizing the Generic Petri Nets DSML.

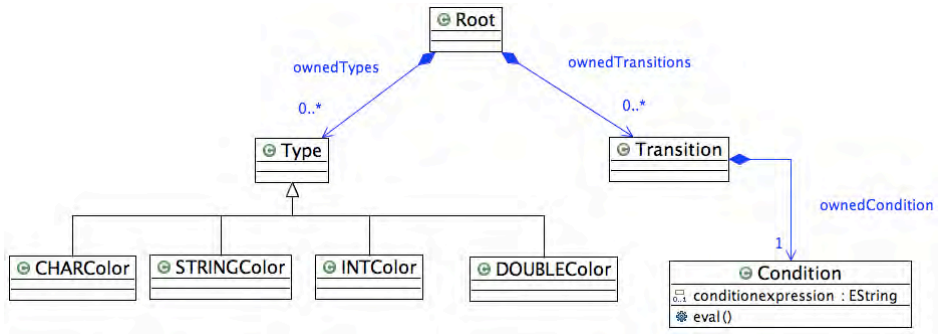


Figure 8.18. colour Petri Nets Extension Parameter: mmPNColor

In the metamodel depicted in Fig. 8.18 the `Type` metaclass is abstract and is specialised by four colours we defined to be the ones we provide support in our colour Petri Nets DSML. The `CHARColor`, `STRINGColor`, `INTColor` and `DOUBLEColor` allows to specify places, terms and multisets with a type which semantics is equivalent to the one provided by most of the programming languages.

The *mmPNColor* metamodel in Fig. 8.18 is used to parameterize the *mmGenPN* Generic Petri Nets metamodel in order to obtain the metamodel of the Colored Petri Nets DSML. The paremeterization is defined as:

- The formal parameter fp the metamodel corresponding to the `Type` and `Transisition` elements of $mmGenPN$;
- $mmPNColor$ in 8.18 the effective parameter ep;
- $\varphi = \{\langle Type, Type\rangle, \langle Transition, Transition\rangle\}$

The result of the metamodel parameterization is presented in Fig. 8.19 with the new and affected elements with a grey background. The new metamodel is obtained by applying the metamodel parameterization as:

$$mmColoredPN =$$
$$mmGenPN[Type \cup Transition \overset{\varphi}{\leftarrow} mmPNColor, true]$$

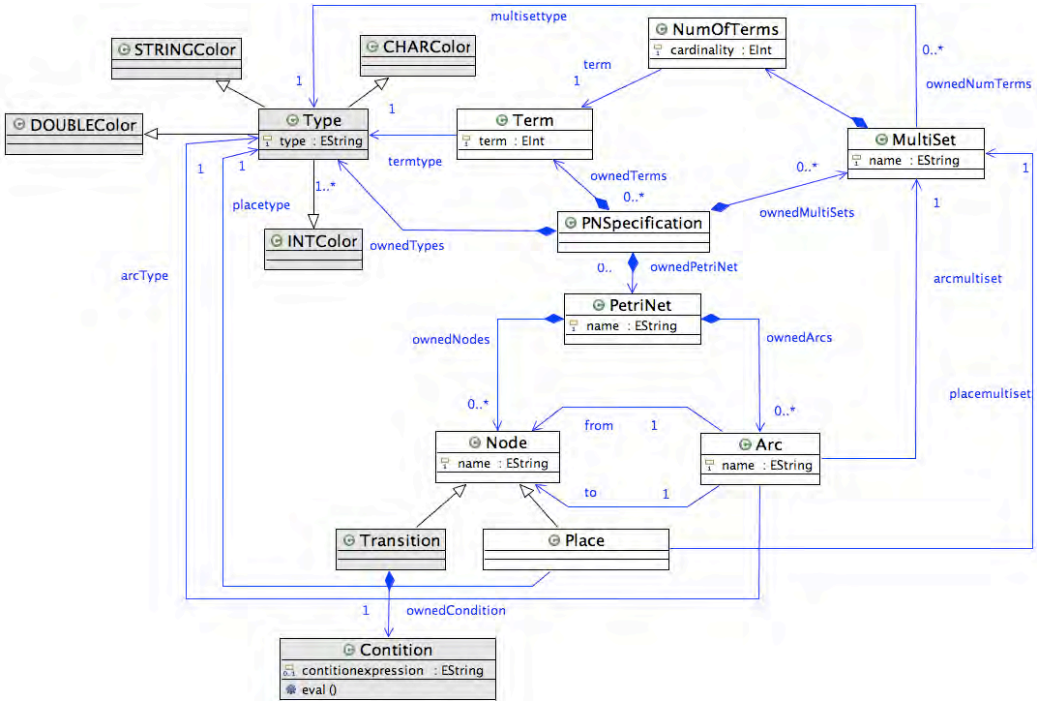


Figure 8.19. Colored Petri Nets Metamodel: mmColoredPN

The transformation differences between the Generic Petri Nets DSML and the Colored Petri Nets DSML are equivalent to the ones observed in the composition performed for the Standard Petri Nets DSML. The main particularity is that the `ruleType` is more complex in the case of Colored Petri Nets DSML. By composing with the transformation rules of the effective parameter of the $mmColoredPN$ we will add a semantics that allows to map

each one of the colour types into the equivalent CO-OPN ADTs. The Fig. 8.20 shows a small part of the transformation substitution. As it is possible to see, each one of the instances of the metaclasses defining the colour types, are transformed into CO-OPN into the ADTs that provides them with the semantics. As an example, instances of `INTColor` metaclass are transformed into a CO-OPN ADT that inherits from the pre-defined ADT Naturals.

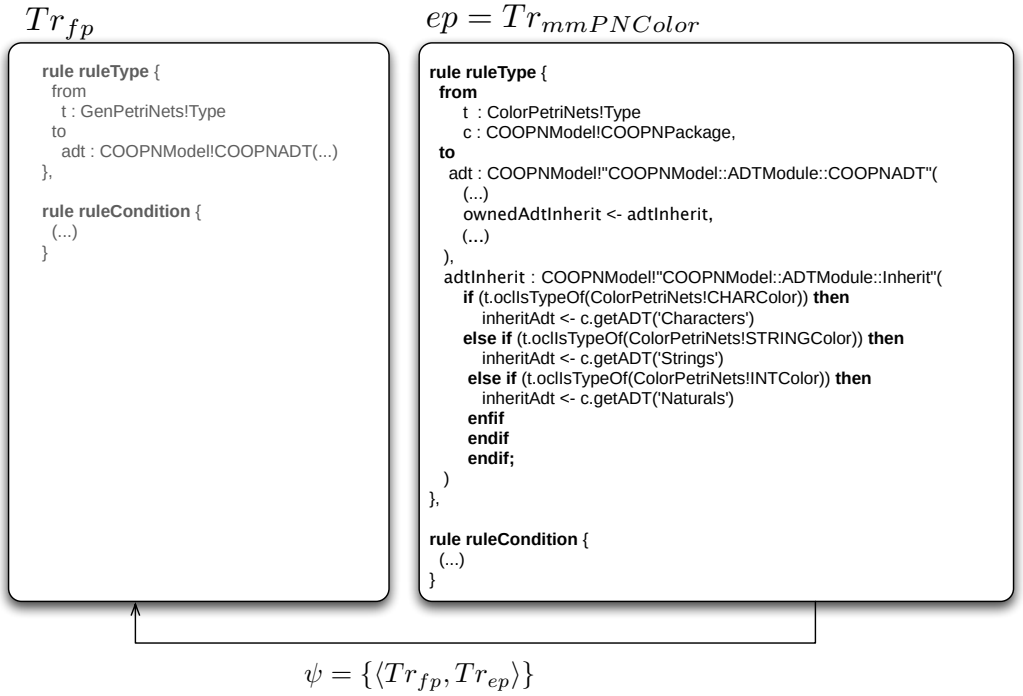


Figure 8.20. colour Petri Nets Transformation Substitution Excerpt

In addition to the substitution of the rules that correspond to the Types' transformation, the ATL rule that is responsible for the transformation Condition transformation is also substituted. In the case of the Colored Petri Nets DSML the new `ruleCondition` creates a CO-OPN axiom that allows to call an `eval` method. This method is responsible to evaluate in its domain the expression provided.

8.3.4 Algebraic Petri Nets

The Algebraic Petri Nets formalism adds the functionality to be able to specify specific data type. These data types are also known as Algebraic Data Types. When creating a Algebraic Petri Net we do not only create a net model with a particular behaviour but also specify the full semantics

of the data types involved. The parameterization of the Generic Petri Nets DSML in order to develop an Algebraic Petri Nets DSML is done using the metamodel in Fig. 8.21.

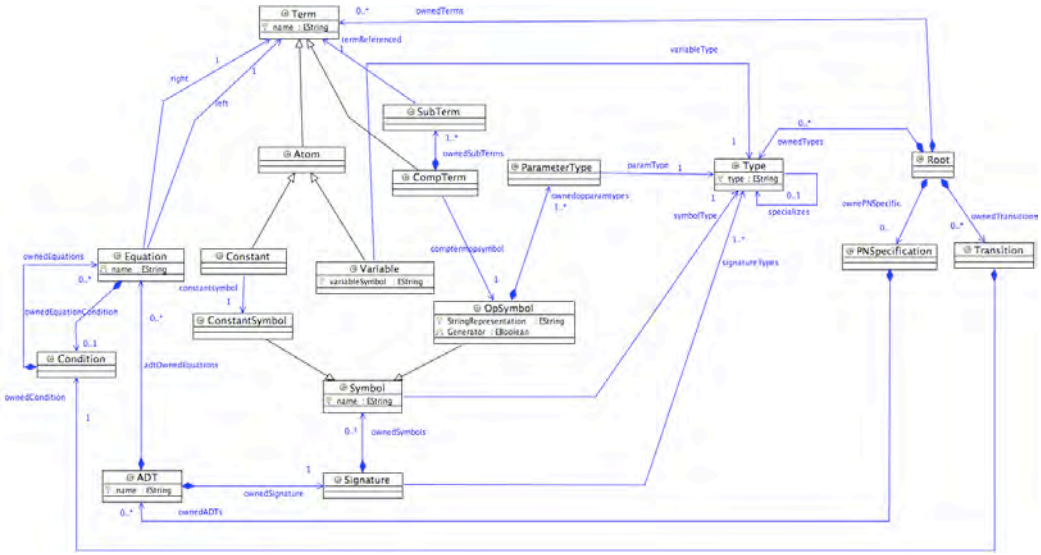


Figure 8.21. Algebraic Petri Nets Extension Parameter: mmPNAlg

The metamodel (and associated transformation rules) in Fig. 8.21 provides the feature of being able to completely specify the semantics of a data type. An Algebraic Data Type is define here by:

- A signature. It is composed by a set of Symbols that are typed. These symbols are either of type `OpSymbol` or `ConstantSymbol`. Each one of them are used with operations or constants respectively. The signature defines the more syntactical part of the Algebraic Data Type;

- A set of equations. Equations provides the data type with a precise semantics. Each equation is composed by a a left and a right `Term` and one or none `Conditions`. A condition is possibly a composition of other equations. A Term is specialised in `Atom` or `CompTerm`. An atom is then specialised by `Constant` and `Variable` that is also typed. The `CompTerm` metaclass stands for composite term and it is used to create term that is a composition of several sub terms.

In order to obtain the Algebraic Petri Nets DSML metamodel we define the following parameterization:

- The *mmPNAlg* metamodel in Fig. 8.21 is used to parameterize the *mmGenPN* Generic Petri Nets;

- The formal parameter fp the metamodel corresponding to the `Type`, `Transisition`, `PNSpecification` and `Term` elements of $mmGenPN$;
- $mmPNAlg$ in 8.21 the effective parameter ep;
- $\varphi = \{\langle Type, Type\rangle, \langle Transition, Transition\rangle, \langle PNSpecification, PNSpecification\rangle, \langle Term, Term\rangle\}$

The result of the metamodel parameterization is presented in Fig. 8.22 with the new and affected elements with a grey background. The new metamodel is obtained by applying the metamodel parameterization as:

$$mmAlgPN =$$
$$mmGenPN[Type \cup Transition \cup$$
$$PNSpecification \cup Term \overset{\varphi}{\leftarrow} mmPNAlg, true]$$

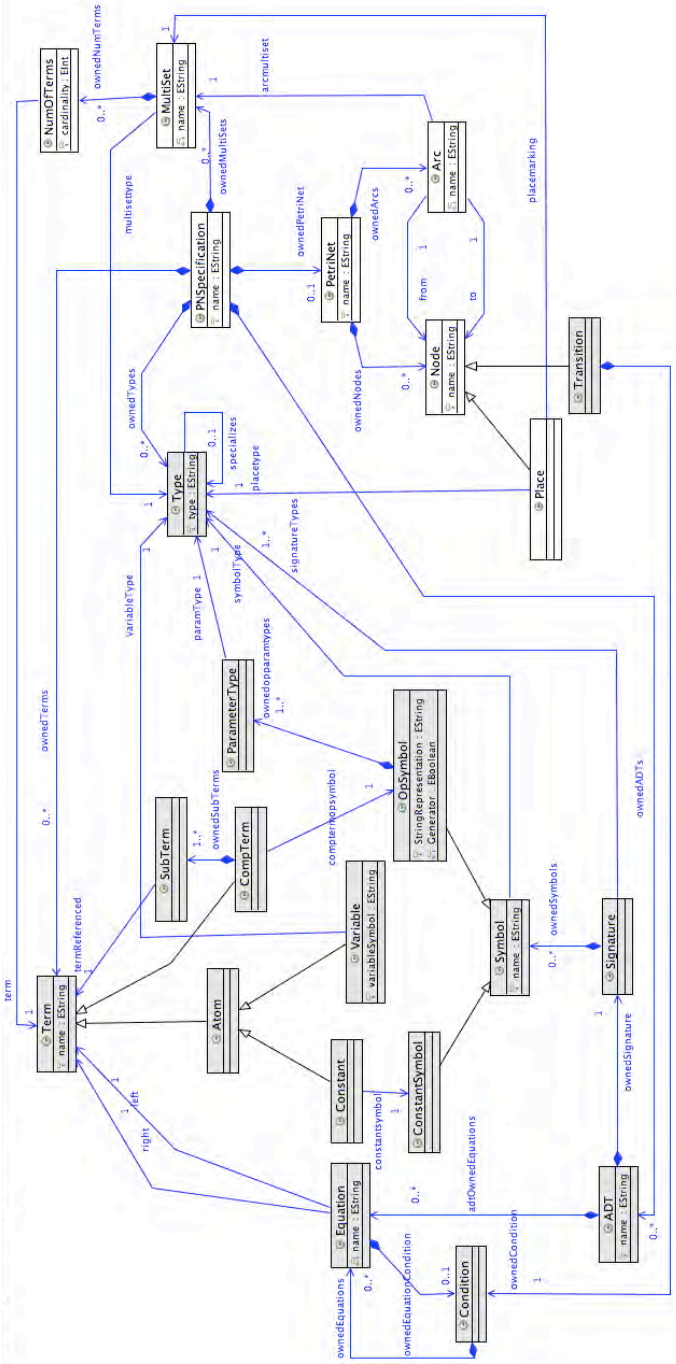


Figure 8.22. Algebraic Petri Nets Metamodel: mmAlgPN

In Fig. 8.23 we present a small excerpt of the transformation composition. This figure presents a part of the transformation of an Algebraic Data Type in the Algebraic Petri Nets DSML domain to an Abstract Algebraic Data Type in the CO-OPN domain. The transformation is mostly a one-to-one transformation. The concept of Algebraic Data Types in the source and target formalism are very similar.

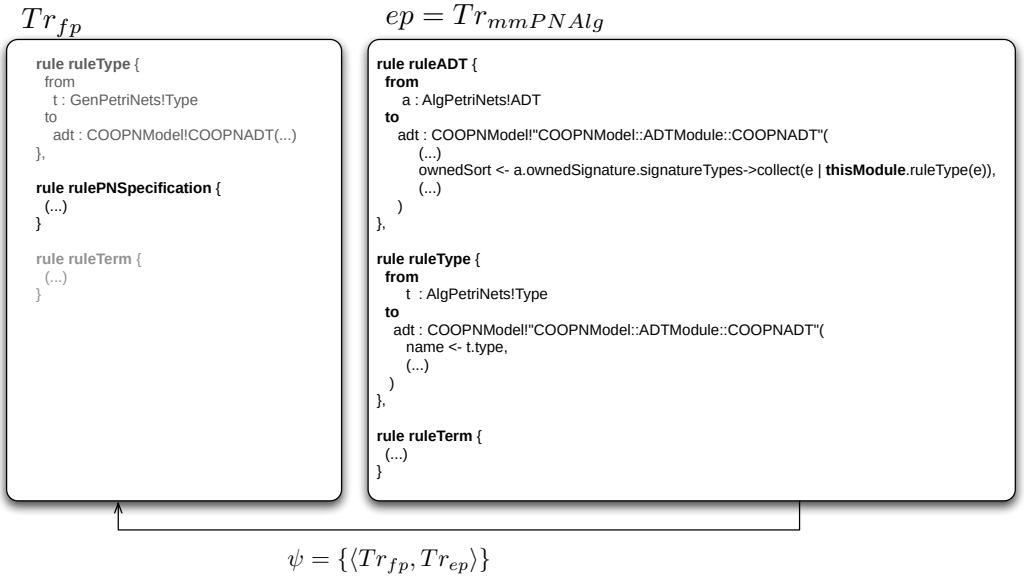


Figure 8.23. Algebraic Petri Nets Transformation Substitution Excerpt

The ATL transformation in Fig. 8.23 shows that the rule `ruleADT` is going to be added to the composed transformation. Rules `ruleType` and `ruleTerm` are replaced by the equivalent in the right side. On the other hand `rulePNSpecification` is not completly replaced.

Having the metamodel in Fig. 8.22 allows to produce Algebraic Petri Nets models as the example presented in Fig. 8.24. This example shows the specification of the well known dining philosophers problem for three philosophers seated around the table.

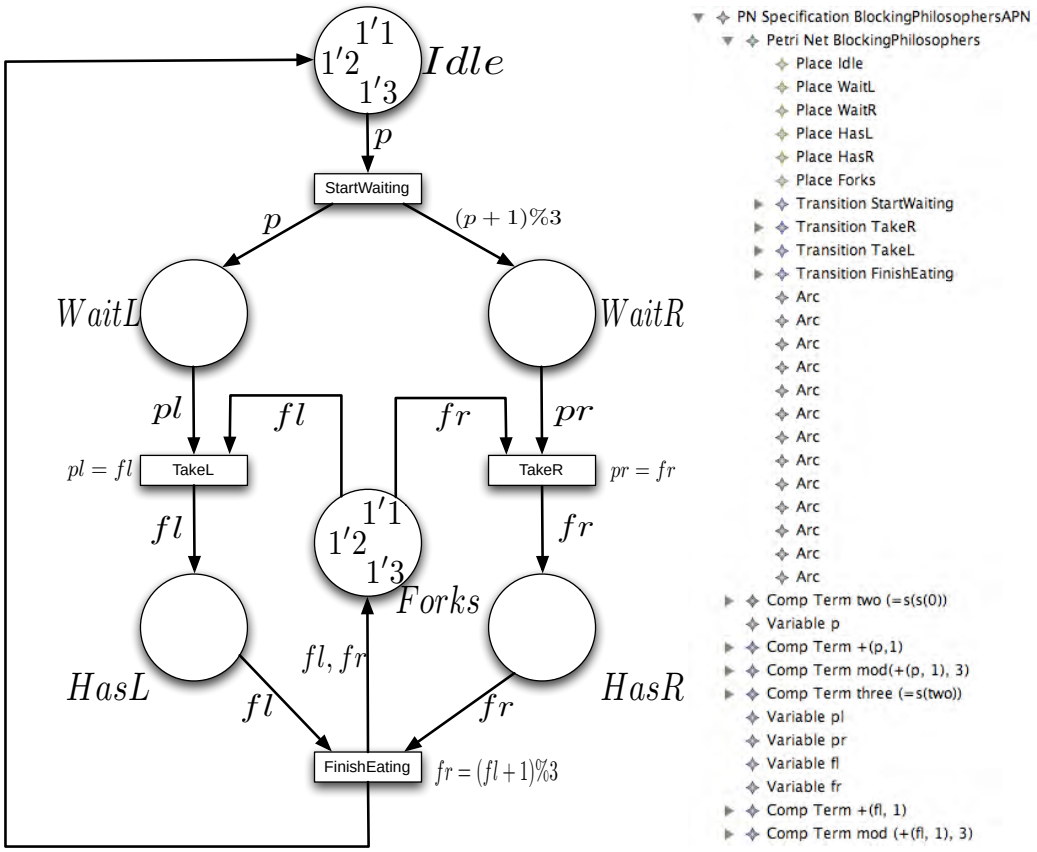


Figure 8.24. Algebraic Petri Nets Model Example

8.4 Comparison Between Case Studies

In order to compare the three presented case studies we defined a set of criteria that allow to better observe the similarities and differences between them. It also allows to highlight the features of the methodology in comparison to different language engineering approaches. The criteria defined cover the topics that we think are important to be managed by a DSML development environment when we thing about diversification, complexity and evolution.

The criteria we use to compare the are:

Concepts Reuse : if the example provides reusability of concepts;

Language Refinement : the act of building a language step-by-step refining parts of it when new requirements arrive;

Semantic Mapping to Several Languages : if the example provides semantic mapping for several semantic domains.

Incremental Language Development : if languages are built by composing blocks representing domain concepts in an incremental approach;

Multi Language Development : if the example allows to show that starting from generic concepts of a domain, it is possible to derive different DSML, i.e., from a common definition we are able to specialize several DSMLs.

The table 8.1 resumes which criteria are covered for each one of the case studies. This table highlights the diversity of the case studies presented. From this table it is also possible to highlight the different approaches of language development used in each one of the case studies.

8.5 Summary

Although the three case studies correspond to rather different domains, by using a Language Engineering approach like the one presented in this document, we can emphasise that two very important criteria are matched in each one o them. Both reusability of concepts and incremental language development is achieved in all of the three. Another very important aspect is that for each one of the case studies, the approach allows to provide a well defined operational semantics.

For one of the case studies we showed that it is possible to use the approach in order to provide semantics in more than one domain.

	Case Studies		
Criteria	Moving Entities	COSPEL	Petri Nets
Concepts Reuse	Yes	Yes	Yes
Language Refinement	No	Yes	No
Semantic Mapping to Several Languages	No	Yes	No
Incremental Language Development	Yes	Yes	Yes
Multi Language Development	Yes	No	Yes

Table 8.1. Case Studies Comparison

The Moving Entities case study is a typical example of specialisation of the domain. By starting from a domain specific but still broader concepts we were able to define two specialisations of one language. These specialisations define two DSMLs in two different domains.

The COSPEL case study is a fairly illustrative example on how to cope with requirements evolution. The domain of application does not change in time but its requirements might vary over time. Language evolution is a key point in all LDE approaches and we showed that this approach is suitable for tackling this problem.

As a summary, although not all of the case studies meet all the defined criteria (which was not the goal of providing a comparison either) we provide an highlight on how different techniques and different types of languages can be developed using the methodology we propose. The table 8.1 shows that in the context of the three case studies together we are able to provide support for all the criteria defined.

Chapter 9

Conclusions

In the context of this thesis' development a lot of topics have been covered. The work presented often interfaced with more than one domain. The complexity of the work was not allways trivial to manage. Interfacing domains that are not completely mature can be of difficult manipulation and represented one of the challenges of this thesis.

As in any research project, some of the work developed was more successful than other. Some of it was ignored after a few weeks as concepts proof to not be suitable - lessons learned we continued trying to avoid the same errors. Other parts of the work are not described in detail in the document of this thesis. Its development was an accessory for supporting the other major parts. Overall this proves that there is a lot to do and a lot to be investigated in the context Software Language Engineering. A lot of other developments can be produced.

9.1 Discussion

In order to be efficient a specialized modelling language must be: intuitive, expressive enough to represent all domain concepts and to have a clear and precise semantics. In the work we presented we showed a framework that facilitates the tasks for providing semantics to DSMLs. Simultaneously we proposed to do it in an environment and using standard and well-known techniques. We think that the proposed solution is a suitable step-forward compared with the existing methods for DSMLs semantics' specification. By using the technique we developed it is possible to develop a DSML using a very modular approach. We also think that this technique allows to enhance quality of the produced DSMLs. The technique we develop provides a method that allows improvement of DSML quality because:

a) DSML development is based on several iterations by adding small concepts to already existing DSML;

b) Validation of iterations is possible after execution transformation compositions;

c) Several iterations can be performed in order to verify which ones are more suitable;

d) Coherence between versions of the language are maintained : models of a previous version are not updated but merged into new models that are in conformity with the new version.

Limitations of the methodology include that not everything can be composed. New and old transformation blocks must have compatible domains, and during the case study we found some coding patterns in transformation which can cause problems (especially with imperative transformation programming). For some of these limitations, an editor which detects problematic compositions could help. In this perspective, a thorough classification of composition problems should be done. Another limitation is that this work obviously applies only to the cases where it makes sense to compose transformations. If DSML development requires a radical rewriting of all associated transformations, there is no particular advantage in using this methodology.

9.2 Possibilities for Future Work

A list of possible followups regarding the research work we developed in the Software Language Engineering are include:

- Properties check. A functionality for providing specification of properties to be guaranteed by a DSML is helpful in order to assure correctness of the language. This is a very interesting topic in the DSML and Model Checking communities;

- Semantics preservation. One of the problems we went across during the development of this work was to answer the following question: *How can we be sure that the semantics of the domain is correctly mapped in the target formalism(s)?* This question raises several others. In particular on the part concerning the specification of certain properties to be checked after and prior to transformation. To research and add functionalities for 'properties transformation' is another aspect from which the current work would surely have something to gain;

- Testing. Test case generation from domain specific aspects is a valuable feature that is still not covered in the DSML development environments. A lot open issues makes this subject a very interesting one to pursuit.;
- Language Versioning and traceability. As language evolves find ourselves with a set of old versions of the language and of its models. New and old models should be able to coexist in the latest DSML environment. Accordingly, old models should be able to access new language features and preserve its behaviour in the old environment;
- Integration with other DSML development environments;
- Graphical extensions. Several DSMLs have a very well defined and strong graphical syntax. To provide constructions for graphical composition and re-utilisation is a crucial aspect for the success of LDE techniques.

9.3 Outcome

Among other indirect results that have been produced during the time of the development of this thesis, the following achievements can be highlighted.

Articles Published articles in Journals and refereed international Conference Proceedings

1. (Pedro, Risoldi, Buchs, Barroca, & Amaral, 2009) *Incremental Prototyping of Domain Specific Languages GUIs*, in the Rapid User Interface Prototyping, Human-Computer Interaction (To be published in 2009);
2. (Pedro, Amaral, & Buchs, 2008) *Foundations for a Domain Specific Modeling Language Prototyping Environment: A compositional approach* in Proceedings of the 8th OOPSLA ACM-SIGPLAN Workshop on Domain-Specific Modeling (DSM), October 2008;
3. (Pedro, Lucio, & Buchs, 2007) *System Prototype and Verification Using Metamodel-Based Transformations*, in IEEE Distributed Systems Online, 2007;
4. (Pedro, Buchs, & Lucio, 2007) *Model and Metamodel Semantics Enrichment Using Transformations and Domain Composition*, in Rapid Integration of Software Engineering techniques 2007 (to be published);

5. (Pedro, Lucio, & Buchs, 2006a) *Principles for System Prototype and Verification Using Metamodel Based Transformations*, in Proceedings of IEEE International Workshop on Rapid System Prototyping, 2006;
6. (A. Chen, Buchs, Lucio, Pedro, & Risoldi, 2006) *Modeling Distributed Systems using Concurrent Object Oriented Petri Nets*, in Proceedings of the Fourth International Workshop on Modelling of Objects, Components and Agents, 2006
7. (Pedro, Lucio, & Buchs, 2006b) *Prototyping Domain Specific Languages With CO-OPN*, in Proceedings of Springer-Verlag Rapid Integration of Software Engineering techniques, 2005;

Technical Reports

1. (Pedro, 2008) *Metamodeling with Eclipse*, Centre Universitaire D'Informatique, Université de Genève, 2008;
2. (Pedro, 2006) *UML2 to CO-OPN transformation: State Machines and Class Diagrams*, Centre Universitaire D'Informatique, Université de Genève, 2006;

CO-OPN Metamodel The goal of this task was to provide a metamodel for CO-OPN in a standard description . We have chosen to use the EMF framework for defining it. Selecting EMF allowed us to, out of the box, integrate with other tools without having to spend a lot of effort in the technical details. Due to CO-OPN structure's complexity its metamodel definition implied several iterations over the time. The task was completed successfully and its accomplishment was very important in the context of providing a practical application of the work defined in this thesis. Moreover, having a well defined CO-OPN metamodel in a standard fashion allows other to be able to easily manipulate CO-OPN specifications. A manipulation that can be wither manual by using the generated CO-OPN pluggin for Eclipse, or manual semi-automatic by defining matamodel-based transformations. The availability of a CO-OPN metamodel allowed other elements of the Software Modelling and Verification group to use CO-OPN in an integrated framework that supports other MDA functionalities.

CO-OPN Serializer CO-OPN is much more than a language metamodel. Its well defined semantics is what makes it interesting. In order to allow re-use of CO-OPN semantics we had to provide a small tool that interfaces the specifications described in conformity with the metamodel with the CO-OPN kernel. This CO-OPN serializer is a transformation that takes specifications in the new format and transforms them into

the format that CO-OPN understands. With this transformation we can re-use all the functionalities of prototyping generation, validation and simulation.

CoPsy To provide a practical application of the methodology we described in this thesis we developed the Compositional Platform for Domain Specific Modelling Languages Prototyping (CoPsy) DSML. This language allows to specify model and transformation composition of other DSMLs. Its behaviour allows to execute composition specification. Based on this specification, a new DSML will be generated.

Appendix A

Acronyms

ATL ATLAS Transformation Language

ADT Algebraic Abstract Data Type

APN Algebraic Petri Nets

ASM Abstract State Machines

CO-OPN Concurrent Object Oriented Petri Nets

CPN Coloured Petri Nets

CS Control System

Cospel COntrol systems SPEcification Language

COM Component Object Model

CoPsy Compositional Platform for Domain Specific Modelling Languages Prototyping

DSML Domain Specific Modelling Language

DSL Domain Specific Language

EMF Eclipse Modeling Framework

EMOF Essential MOF

GME Generic Modeling Environment

GMF Graphical Modeling Framework

GOPPRR Graph-Object-Property-Port-Role-Relationship

GP Generative Programing

GReAT Graph Rewriting and Transformation

GT Graph Transformation

IDE Integrated Development Environment

LDE Language Driven Engineering

LHS Left-Hand Side

MOF Meta-Object Facility

MDA Model Driven Architecture

M2M Model-to-Model

MDE Model-Driven Engineering

OMG Object Management Group

OCL Object Constraint Language

OO Object-Oriented

PIM Platform Independent Model

PSM Platform Specific Model

QVT Query / Views / Transformations

RHS Right-Hand Side

SLE Software Language Engineering

SQL Structured Query Language

SVG Scalable Vector Graphics

UML Unified Modeling Language

Appendix B

CO-OPN Example Specifications

B.1 CO-OPN Booleans ADT

The Listing B.1 shows the definition of a Boolean type by means of a CO-OPN ADT. It uses CO-OPN ADT's concrete textual syntax. The Fig. B.1 presents the same Boolean ADT. This time in the EMF platform using the CO-OPN metamodel definition.

Listing B.1. CO-OPN Boolean ADT Specification

```
ADT Booleans;
Interface
 Sort
   boolean;
 Generators
   true : -> boolean;
   false : -> boolean;
 Operations
   not _ : boolean -> boolean;
   _ and _ : boolean, boolean -> boolean;
   _ or _ : boolean, boolean -> boolean;
   _ xor _ : boolean, boolean -> boolean;
   _ = _ : boolean boolean -> boolean;
Body
 Axioms
   not true = false;
   not false = true;
   true and booleanVar1 = booleanVar1;
   false and booleanVar1 = false;
   true or booleanVar1 = true;
   false or booleanVar1 = booleanVar1;
   false xor booleanVar1 = booleanVar1;
   true xor booleanVar1 = not booleanVar1;
   (true = true) = true;
   (true = false) = false;
   (false = true) = false;
   (false = false) = true;
 Theorems
;; reflexivity
   (booleanVar1 = booleanVar1) = true;
```

```
;; symetry
   (booleanVar1 = booleanVar2) = true => (booleanVar2 = booleanVar1) = true;
;; transitivity
   (booleanVar1 = booleanVar2) = true & (booleanVar2 = booleanVar3) = true
       => (booleanVar1 = booleanVar3) = true;
 Where
   booleanVar1 : boolean;
   booleanVar2 : boolean;
   booleanVar3 : boolean;
End Booleans;
```

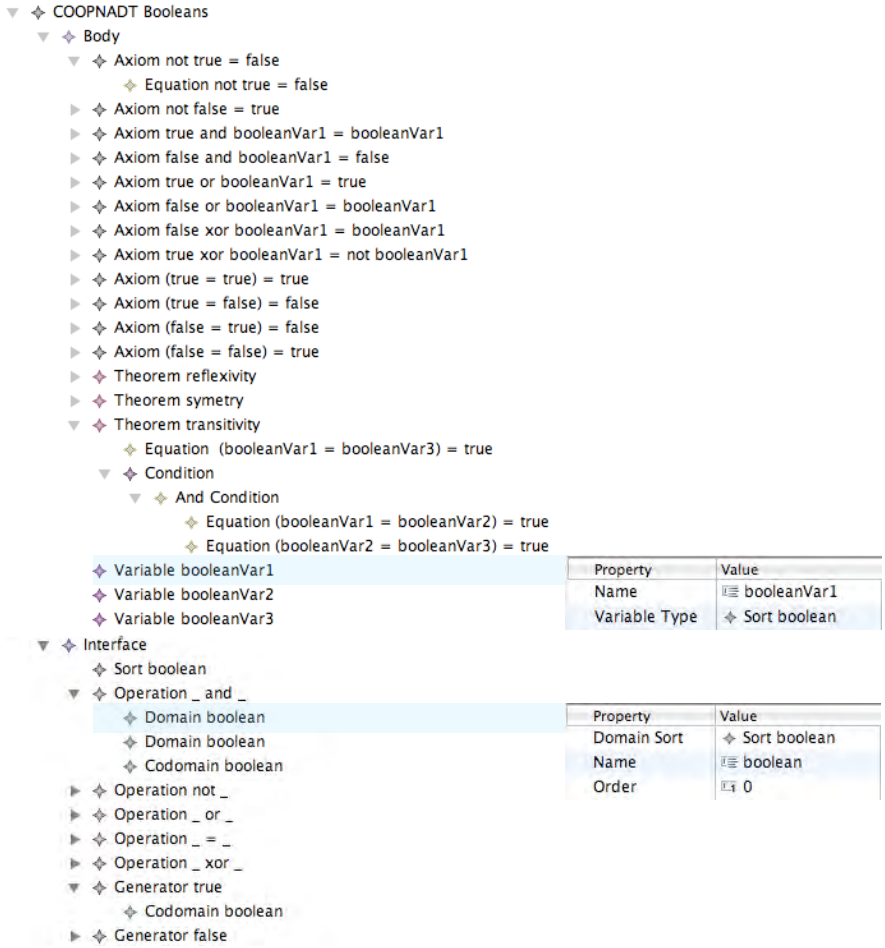


Figure B.1. CO-OPN ADT Booleans Specification

B.2 CO-OPN Machine Class

In this section we present an example of a machine specification using a CO-OPN Class. The example specifies a state machine that has two states: *running* and *stopped*. The machie can switch stated from one to the other using the *start* and *stop* methods. In addition the machine contains a *counter* place that allows to count the number of cycles that it went to the *start* state. Moreover a *reset* method is specified and it allows, while the machine is at state *stopped* to reset the counter to an integer value. The *counter* attribute is initialised to 0. The *increase* method is a private one (defined in the body section) and is called as a synchronisation when the *start* method is called. Simoultaneously the $numOfStartCycles$ gate is called in parallel which allows to output the actual number of start cycles.

Listing B.2 shows the machine class specification in the CO-OPN concrete textual syntax and the Fig. **??** shows the exact same specification using CO-OPN graphical syntax: Methods are represented as black rectangles and gates as white backgound rectangles; places are the big circles with a label; the arrows represent pre and post conditions; and the small circle represents the parallel synchronisation. The Fig. B.3 shows the machine class specification using the CO-OPN EMF editor.

Listing B.2. CO-OPN Machine Class Specification

```
Class CLASSMachine;
Interface
 Use
   Naturals;
 Type
   machineType;
 Gate
   numOfStartCycles _ : natural;
 Methods
   start;
   stop;
   reset _ : natural;
Body
 Use
   BlackTokens;
 Methods
   increase;
 Places
   stopped _ : blackToken;
   running _ : blackToken;
   counter _ : natural;
 Initial
   stopped @;
   counter 0;
 Axioms
   this = Self => start With this . increase // this . numOfStartCycles n::
       stopped @, counter n -> running @, counter n;
   stop::running @ -> stopped @;
   increase::counter n -> counter n + 1;
```

```
    reset n::stopped @ -> stopped @, counter n;
30  Where
      n : natural;
32    this : machineType;
End CLASSMachine;
```

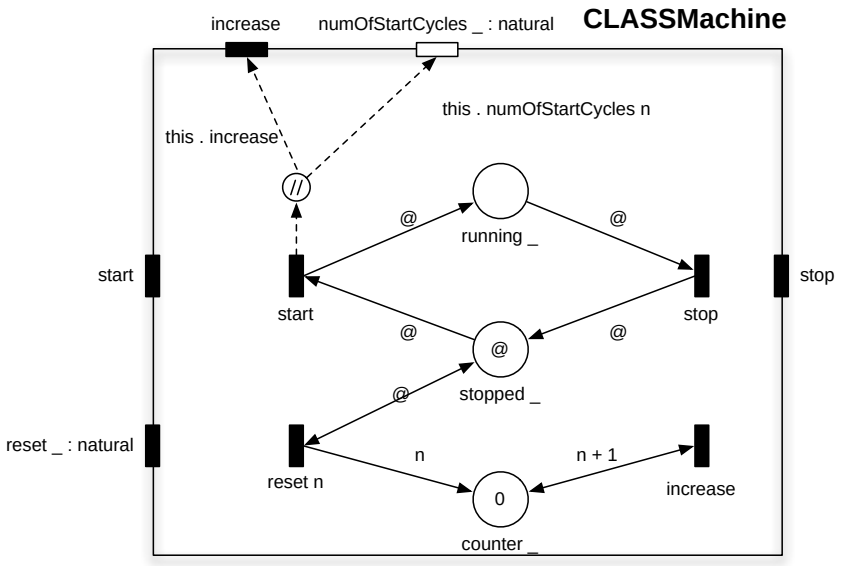


Figure B.2. CO-OPN Machine Class Specification

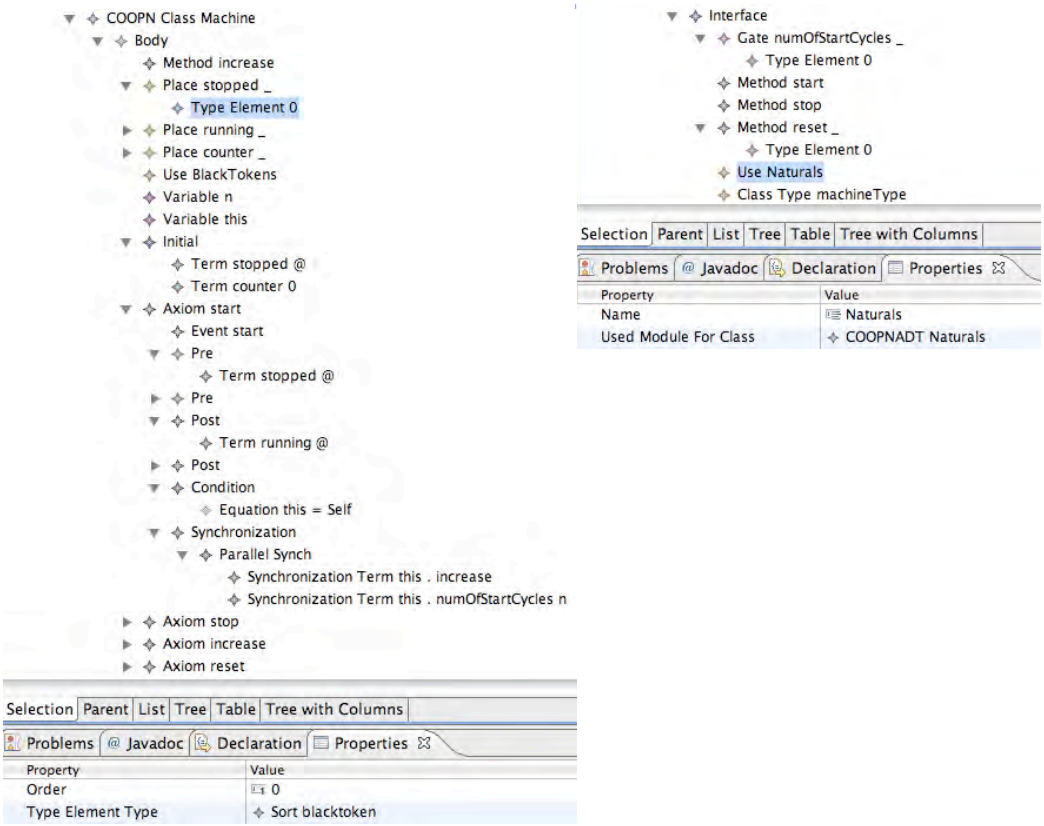


Figure B.3. CO-OPN Machine Class Specification using EMF Editor

B.3 CO-OPN Machine Context Specification

This example illustrates how to specify a CO-OPN context. The example presented uses the previous definition of the Machine CO-OPN class in appendix B.2.

Listing B.3. CO-OPN Machine Context Specification

```
Context CTXMachine;
Interface
 Use
   Naturals;
 Gate
   numOfStartCycles _ : natural;
 Methods
   start;
   stop;
   reset _ : natural;
Body
 Use
   CLASSMachine;
 Object
   obj : machineType;
 Axioms
   start With obj . start;
   stop With obj . stop;
   reset n With obj . reset n;
   obj . numOfStartCycles n With numOfStartCycles n;
 Where
   n : natural;
End CTXMachine;
```

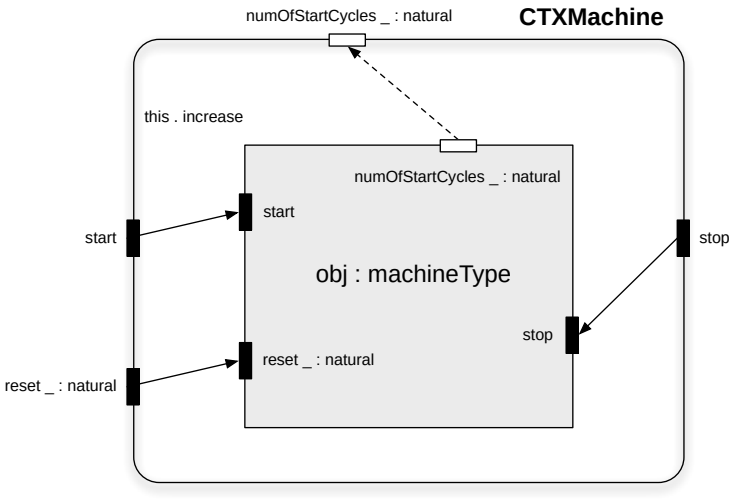


Figure B.4. CO-OPN Machine Context Specification using CO-OPN Graphical Resp resentation

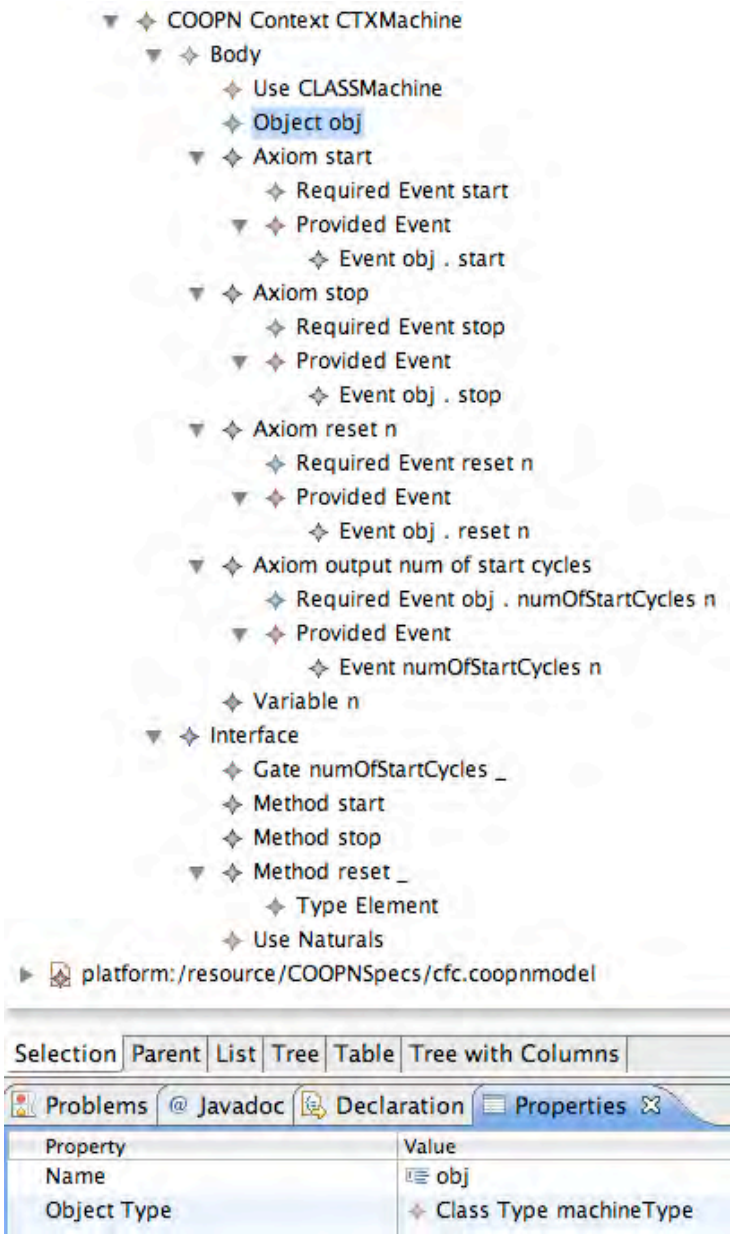


Figure B.5. CO-OPN Machine Context Specification using EMF Editor

Appendix C

Moving Entities Transformations

C.1 Generic Moving Entities DSML Transformation

Listing C.1. Complete Transformation of Moving Entities to CO-OPN

```
-- @atlcompiler atl2006
module GenericMovingSystem; -- Module Template
create entitiesCOOPN : COOPNModel  from entities : MovingEntitiesDSML;

helper  def : createInitAxiom(movingEntities : Set(MovingEntitiesDSML!
    MovingEntities)) : String =
    '( obj' + movingEntities.first().name + '.' + (' setName ' + (
        movingEntities.first().name.toString())) + ')' +
    if (movingEntities.size() = 1) then
      ''
    else
      ' // ' +
      (thisModule.createInitAxiom(movingEntities.excluding(movingEntities.
          first())))
    endif;

rule init {
  from
    dsml : MovingEntitiesDSML!DSML
  to
    ----------------
    -- Way Segment --
    ----------------
    coopnclassws : COOPNModel!"COOPNModel::ClassModule::COOPNClass"(
      name <- 'CLASSWaySegment',
      ownedBody <- bodyws,
      ownedInterface <- interfacews
    ),
    bodyws : COOPNModel!"COOPNModel::ClassModule::Body"(
      ownedPlaces <- endpoint1,
      ownedPlaces <- endpoint2,
      ownedPlaces <- namews
    ),
    endpoint1 : COOPNModel!"COOPNModel::ClassModule::Place"(
```

```
    name <- 'EndPoint1'
  ),
  endpoint2 : COOPNModel!"COOPNModel::ClassModule::Place"(
    name <- 'EndPoint2'
  ),
  namews : COOPNModel!"COOPNModel::ClassModule::Place"(
    name <- 'ID'
  ),

  interfacews : COOPNModel!"COOPNModel::ClassModule::Interface"(
    ownedInterfaceClassTypes <- classtypews,
    ownedInterfaceMethods <- setNamews,
    ownedInterfaceMethods <- setEndPoint1,
    ownedInterfaceMethods <- setEndPoint2
  ),
  classtypews : COOPNModel!"COOPNModel::ClassModule::ClassType"(
    name <- 'waysegment'
  ),
  setNamews : COOPNModel!"COOPNModel::ClassModule::Method"(
    name <- 'setName _'
  ),
  setEndPoint1 : COOPNModel!"COOPNModel::ClassModule::Method"(
    name <- 'setEndPoint1 _'
  ),
  setEndPoint2 : COOPNModel!"COOPNModel::ClassModule::Method"(
    name <- 'setEndPoint2 _'
  ),
  ----------------------
  -- Junction Point --
  ----------------------
  coopnclassjp : COOPNModel!"COOPNModel::ClassModule::COOPNClass"(
    name <- 'CLASSJunctionPoint',
    ownedBody <- bodyjp,
    ownedInterface <- interfacejp
  ),
  bodyjp : COOPNModel!"COOPNModel::ClassModule::Body"(
    ownedPlaces <- startjp,
    ownedPlaces <- endjp,
    ownedPlaces <- namejp
  ),
  startjp : COOPNModel!"COOPNModel::ClassModule::Place"(
    name <- 'Start'
  ),
  endjp : COOPNModel!"COOPNModel::ClassModule::Place"(
    name <- 'End'
  ),
  namejp : COOPNModel!"COOPNModel::ClassModule::Place"(
    name <- 'name'
  ),
  interfacejp : COOPNModel!"COOPNModel::ClassModule::Interface"(
    ownedInterfaceClassTypes <- classtypejp,
    ownedInterfaceMethods <- setStartjp,
    ownedInterfaceMethods <- setEndjp
  ),
  classtypejp : COOPNModel!"COOPNModel::ClassModule::ClassType"(
    name <- 'junctionpoint'
  ),
  setStartjp : COOPNModel!"COOPNModel::ClassModule::Method"(
    name <- 'setStart _'
  ),
  setEndjp : COOPNModel!"COOPNModel::ClassModule::Method"(
    name <- 'setEnd _'
```

```
    ),

    ----------------------
    -- Moving Entity --
    ----------------------
    coopnclassme : COOPNModel!"COOPNModel::ClassModule::COOPNClass"(
      name <- 'CLASSMovingEntity',
      ownedBody <- bodyme,
      ownedInterface <- interfaceme
    ),
    bodyme : COOPNModel!"COOPNModel::ClassModule::Body"(
      ownedPlaces <- nameme
    ),

    nameme : COOPNModel!"COOPNModel::ClassModule::Place"(
      name <- 'name_'
    ),
    interfaceme : COOPNModel!"COOPNModel::ClassModule::Interface"(
      ownedInterfaceClassTypes <- classstypeme,
      ownedInterfaceMethods <- setNameme
    ),
    classstypeme : COOPNModel!"COOPNModel::ClassModule::ClassType"(
      name <- 'movingentity'
    ),

    setNameme : COOPNModel!"COOPNModel::ClassModule::Method"(
      name <- 'setName_'
    ),

    ---------------------------------
    -- World Structure Context --
    ---------------------------------
    ctx : COOPNModel!"COOPNModel::ContextModule::COOPNContext"(
      name <- 'CTX' + dsml.ownedWorld.worldName,
      ownedBody <- body,
      ownedInterface <- interface
    ),

    body: COOPNModel!"COOPNModel::ContextModule::Body"(
      ownedObjects <- dsml.ownedWorld.ownedSegments->collect(e | thisModule.
          ruleWaySegmentObj(e)),
      ownedObjects <- dsml.ownedWorld.ownedJunctionPoints->collect(e |
          thisModule.ruleJunctionPointObj(e)),
      ownedObjects <- dsml.ownedEntities->collect(e | thisModule.
          ruleEntitiesObj(e)),
      ownedAxiomTheorems <- axiom

    ),
    axiom : COOPNModel!"COOPNModel::ContextModule::Axiom"(
      ownedRequiredEvent <- reqevent,
      ownedProvidedEvent <- providedevent
    ),
    reqevent : COOPNModel!"COOPNModel::ContextModule::RequiredEvent"(
      event <- 'initDSML '
    ),
    providedevent : COOPNModel!"COOPNModel::ContextModule::ProvidedEvent"(
      ownedProvidedEventAtom <-    providedeventatom
    ),
    providedeventatom : COOPNModel!"COOPNModel::ContextModule::Event"(
      event <- if (dsml.ownedEntities.oclIsUndefined()) then
        ''
      else
```

```
        thisModule.createInitAxiom(dsml.ownedEntities)
      endif
    ),

    interface : COOPNModel!"COOPNModel::ContextModule::Interface"(
      ownedInterfaceMethods <- ctxInterfaceMethods
    ),
    ctxInterfaceMethods : COOPNModel!"COOPNModel::ContextModule::Method"(
      name <- 'initDSML'
    )
}

lazy rule ruleWaySegmentObj {
  from
    ws : MovingEntitiesDSML!WaySegment
  to
    obj : COOPNModel!"COOPNModel::ContextModule::Object"(
      name <- 'objWaySegment' + ws.id.toString()
    )
}

lazy rule ruleJunctionPointObj {
  from
    jp : MovingEntitiesDSML!JunctionPoint
  to
    obj : COOPNModel!"COOPNModel::ContextModule::Object"(
      name <- 'objJunctionPoint' + jp.name
    )
}

lazy rule ruleEntitiesObj {
  from
    me : MovingEntitiesDSML!MovingEntitiy
  to
    obj : COOPNModel!"COOPNModel::ContextModule::Object"(
      name <- 'obj' + me.name
    )
}
```

C.2 Train Entity Transformation

Listing C.2. Complete Transformation of Train Entities to CO-OPN

```
-- @atlcompiler atl2006
module TrainEntity; -- Module Template
create trainsCOOPN : COOPNModel from trains : TrainEntityDSML;

helper def : createActionPlanAxiom(gotoActions : Set(TrainEntityDSML!
    GoToAction), objname : String) : String =
    '(' + objname + '.' + (' goTo ' + (gotoActions.first().goto.stationName.
        toString())) + ')' +
    if (gotoActions.size() = 1) then
      ''
    else
      ' .. ' +
      (thisModule.createActionPlanAxiom(gotoActions.excluding(gotoActions.
          first()), objname ))
    endif;
```

```
rule ruleInit{
  from
    root : TrainEntityDSML!Root
  to
    ----------------------
    -- Railway Station --
    ----------------------
    coopnclassstation : COOPNModel!"COOPNModel::ClassModule::COOPNClass"(
      name <- 'CLASSTrainStation',
      ownedBody <- bodystation,
      ownedInterface <- interfacestation
    ),
    bodystation : COOPNModel!"COOPNModel::ClassModule::Body"(
      ownedPlaces <- namestation
    ),
    namestation : COOPNModel!"COOPNModel::ClassModule::Place"(
      name <- 'name'
    ),
    interfacestation : COOPNModel!"COOPNModel::ClassModule::Interface"(
      ownedInterfaceClassTypes <- classtypestation,
      ownedInterfaceMethods <- methodgoto
    ),
    classtypestation : COOPNModel!"COOPNModel::ClassModule::ClassType"(
      name <- 'trainstation'
    ),
    methodgoto : COOPNModel!"COOPNModel::ClassModule::Method"(
      name <- 'goTo _'
    ),

    ---------------------
    -- Train Entity  --
    ---------------------
    coopnclasstr : COOPNModel!"COOPNModel::ClassModule::COOPNClass"(
      name <- 'CLASSTrain',
      ownedBody <- bodytr,
      ownedInterface <- interfacetr
    ),
    bodytr : COOPNModel!"COOPNModel::ClassModule::Body"(
      ownedPlaces <- locationtr,
      ownedPlaces <- nametr,
      ownedVariables <- vartr,
      ownedAxiomTheorems <-axiomstr
    ),
    locationtr : COOPNModel!"COOPNModel::ClassModule::Place"(
      name <- 'Location _'
    ),

    nametr : COOPNModel!"COOPNModel::ClassModule::Place"(
      name <- 'name _'
    ),
    vartr : COOPNModel!"COOPNModel::ClassModule::Variable"(
      name <- 'vartrainname'
    ),
    axiomstr : COOPNModel!"COOPNModel::ClassModule::Axiom"(
      ownedEvent <-eventtr,
      ownedPost <- posttr,
      ownedCondition <- conditiontr
    ),
    eventtr : COOPNModel!"COOPNModel::ClassModule::Event"(
      event <- 'setName vartrainname'
    ),
```

```
    posttr : COOPNModel!"COOPNModel::ClassModule::Post"(
      ownedPostTerm <- postTermtr
    ),
    postTermtr : COOPNModel!"COOPNModel::ClassModule::Term"(
      expression <-  'name vartrainname'
    ),
    conditiontr : COOPNModel!"COOPNModel::ClassModule::Condition"(
      ownedConditionAtom <-   conditionatomtr
    ),
    conditionatomtr : COOPNModel!"COOPNModel::ClassModule::Equation"(
      expression <- '(this = Self) = true'
    ),
    interfacetr : COOPNModel!"COOPNModel::ClassModule::Interface"(
      ownedInterfaceClassTypes <- classtypetr,
      ownedInterfaceMethods <- setNametr,
      ownedInterfaceMethods <- setLocationtr
    ),
    classtypetr : COOPNModel!"COOPNModel::ClassModule::ClassType"(
      name <- 'train'
    ),
    setLocationtr : COOPNModel!"COOPNModel::ClassModule::Method"(
      name <- 'setLocation _'
    ),
    setNametr : COOPNModel!"COOPNModel::ClassModule::Method"(
      name <- 'setName _'
    ),

    ------------------------------------
    -- Dummy Context to create Objects--
    ------------------------------------
    ctx : COOPNModel!"COOPNModel::ContextModule::COOPNContext"(
      name <- 'CTXRoot',
      ownedBody <- body,
      ownedInterface <- interface
    ),

    body: COOPNModel!"COOPNModel::ContextModule::Body"(
      ownedObjects <- root.ownedTrainEntities->collect(e | thisModule.
          ruleTrainEntitiesObj(e)),
      ownedObjects <- root.ownedRailwayStations->collect(e | thisModule.
          ruleRailWayStationObj(e)),
      ownedAxiomTheorems <- root.ownedTrainEntities->collect(e | thisModule.
          executeActionPlanAxiom(e)),
      onwedBodyVariables <- ctxvariable
    ),
    interface : COOPNModel!"COOPNModel::ContextModule::Interface"(
      ownedInterfaceMethods <- ctxInterfaceMethods
    ),
    ctxInterfaceMethods : COOPNModel!"COOPNModel::ContextModule::Method"(
      name <- 'executeActionPlan _'
    ),
    ctxvariable : COOPNModel!"COOPNModel::ContextModule::Variable"(
      name <- 'vartrain'
    )
}

lazy rule executeActionPlanAxiom {
  from
    train : TrainEntityDSML!TrainEntity
  to
    axiom : COOPNModel!"COOPNModel::ContextModule::Axiom"(
      ownedRequiredEvent <- reqevent,
```

```
      ownedProvidedEvent <- providedevent,
      ownedCondition <- condition
    ),
    reqevent : COOPNModel!"COOPNModel::ContextModule::RequiredEvent"(
      event <- 'executeActionPlan ' + train.name
    ),
    providedevent : COOPNModel!"COOPNModel::ContextModule::ProvidedEvent"(
      ownedProvidedEventAtom <-   providedeventatom
    ),
    providedeventatom : COOPNModel!"COOPNModel::ContextModule::Event"(
      event <- if (train.ownedActionPlan.oclIsUndefined()) then
        ''
      else if (train.ownedActionPlan.ownedActions.oclIsUndefined()) then
        ''
      else
        thisModule.createActionPlanAxiom(train.ownedActionPlan.ownedActions,
            'obj' + train.name)
      endif
      endif
    ),
    condition : COOPNModel!"COOPNModel::ContextModule::Condition"(
      ownedConditionAtom <-   conditionatom
    ),
    conditionatom : COOPNModel!"COOPNModel::ContextModule::Equation"(
      expression <- '(vartrain = ' + train.name + ') = true'
    )
}

lazy rule ruleTrainEntitiesObj {
  from
    train : TrainEntityDSML!TrainEntity
  to
    obj : COOPNModel!"COOPNModel::ContextModule::Object"(
      name <- 'obj' + train.name
    )


}

lazy rule ruleRailWayStationObj {
  from
    rs : TrainEntityDSML!RailWayStation
  to
    obj : COOPNModel!"COOPNModel::ContextModule::Object"(
      name <- 'objTrainStation' + rs.stationName
    )
}
```

C.3 Railay System DSML Transformation

Listing C.3. Complete Transformation of Railway System DSML to CO-OPN

```
-- @atlcompiler atl2006
module GenericMovingSystem; -- Module Template
create entitiesCOOPN : COOPNModel from entities : DSMLRailwaySystem;

helper def : createInitAxiom(movingEntities : Set(DSMLRailwaySystem!
    MovingEntities)) : String =
```

```
    '( obj' + movingEntities.first().name + '.' + (' setName ' + (
        movingEntities.first().name.toString())) + ')' +
    if (movingEntities.size() = 1) then
      ''
    else
      ' // ' +
     (thisModule.createInitAxiom(movingEntities.excluding(movingEntities.
         first())))
    endif;


helper  def : createActionPlanAxiom(gotoActions : Set(DSMLRailwaySystem!
    GoToAction), objname : String) : String =
    '(' + objname + '.' + (' goTo ' + (gotoActions.first().goto.stationName.
        toString())) + ')' +
    if (gotoActions.size() = 1) then
      ''
    else
      ' .. ' +
     (thisModule.createActionPlanAxiom(gotoActions.excluding(gotoActions.
         first()), objname ))
    endif;

rule init {
  from
    dsml : DSMLRailwaySystem!DSML
  to
    ------------------
    -- Way Segment --
    ------------------
    coopnclassws : COOPNModel!"COOPNModel::ClassModule::COOPNClass"(
      name <- 'CLASSWaySegment',
      ownedBody <- bodyws,
      ownedInterface <- interfacews
    ),
    bodyws : COOPNModel!"COOPNModel::ClassModule::Body"(
      ownedPlaces <- endpoint1,
      ownedPlaces <- endpoint2,
      ownedPlaces <- namews
    ),
    endpoint1 : COOPNModel!"COOPNModel::ClassModule::Place"(
      name <- 'EndPoint1'
    ),
    endpoint2 : COOPNModel!"COOPNModel::ClassModule::Place"(
      name <- 'EndPoint2'
    ),
    namews : COOPNModel!"COOPNModel::ClassModule::Place"(
      name <- 'ID'
    ),

    interfacews : COOPNModel!"COOPNModel::ClassModule::Interface"(
      ownedInterfaceClassTypes <- classtypews,
      ownedInterfaceMethods <- setNamews,
      ownedInterfaceMethods <- setEndPoint1,
      ownedInterfaceMethods <- setEndPoint2
    ),
    classtypews : COOPNModel!"COOPNModel::ClassModule::ClassType"(
      name <- 'waysegment'
    ),
    setNamews : COOPNModel!"COOPNModel::ClassModule::Method"(
      name <- 'setName _'
    ),
```

```
    setEndPoint1 : COOPNModel!"COOPNModel::ClassModule::Method"(
      name <- 'setEndPoint1 _'
    ),
    setEndPoint2 : COOPNModel!"COOPNModel::ClassModule::Method"(
      name <- 'setEndPoint2 _'
    ),
    ------------------------
    -- Railway Station --
    ------------------------
    coopnclassjp : COOPNModel!"COOPNModel::ClassModule::COOPNClass"(
      name <- 'CLASSTrainStation',
      ownedBody <- bodyjp,
      ownedInterface <- interfacejp
    ),
    bodyjp : COOPNModel!"COOPNModel::ClassModule::Body"(
      ownedPlaces <- startjp,
      ownedPlaces <- endjp,
      ownedPlaces <- namejp

    ),
    startjp : COOPNModel!"COOPNModel::ClassModule::Place"(
      name <- 'Start'
    ),
    endjp : COOPNModel!"COOPNModel::ClassModule::Place"(
      name <- 'End'
    ),
    namejp : COOPNModel!"COOPNModel::ClassModule::Place"(
      name <- 'name'
    ),
    interfacejp : COOPNModel!"COOPNModel::ClassModule::Interface"(
      ownedInterfaceClassTypes <- classtypejp,
      ownedInterfaceMethods <- setStartjp,
      ownedInterfaceMethods <- setEndjp,
      ownedInterfaceMethods <- methodgoto
    ),
    classtypejp : COOPNModel!"COOPNModel::ClassModule::ClassType"(
      name <- 'trainstation'
    ),
    setStartjp : COOPNModel!"COOPNModel::ClassModule::Method"(
      name <- 'setStart _'
    ),
    setEndjp : COOPNModel!"COOPNModel::ClassModule::Method"(
      name <- 'setEnd _'
    ),
    methodgoto : COOPNModel!"COOPNModel::ClassModule::Method"(
      name <- 'goTo _'
    ),

    ------------------------
    -- Train Entity --
    ------------------------
    coopnclasstr : COOPNModel!"COOPNModel::ClassModule::COOPNClass"(
      name <- 'CLASSTrain',
      ownedBody <- bodytr,
      ownedInterface <- interfacetr
    ),
    bodytr : COOPNModel!"COOPNModel::ClassModule::Body"(
      ownedPlaces <- locationtr,
      ownedPlaces <- nametr,
      ownedVariables <- vartr,
      ownedAxiomTheorems <-axiomstr
    ),
```

```
125     locationtr : COOPNModel!"COOPNModel::ClassModule::Place"(
          name <- 'Location _'
        ),

        nametr : COOPNModel!"COOPNModel::ClassModule::Place"(
130       name <- 'name _'
        ),
        vartr : COOPNModel!"COOPNModel::ClassModule::Variable"(
          name <- 'vartrainname'
        ),
135     axiomstr : COOPNModel!"COOPNModel::ClassModule::Axiom"(
          ownedEvent <-eventtr,
          ownedPost <- posttr,
          ownedCondition <- conditiontr
        ),
140     eventtr : COOPNModel!"COOPNModel::ClassModule::Event"(
          event <- 'setName vartrainname'
        ),
        posttr : COOPNModel!"COOPNModel::ClassModule::Post"(
          ownedPostTerm <- postTermtr
145     ),
        postTermtr : COOPNModel!"COOPNModel::ClassModule::Term"(
          expression <-  'name vartrainname'
        ),
        conditiontr : COOPNModel!"COOPNModel::ClassModule::Condition"(
150       ownedConditionAtom <-   conditionatomtr
        ),
        conditionatomtr : COOPNModel!"COOPNModel::ClassModule::Equation"(
          expression <- '(this = Self) = true'
        ),
155
        interfacetr : COOPNModel!"COOPNModel::ClassModule::Interface"(
          ownedInterfaceClassTypes <- classtypetr,
          ownedInterfaceMethods <- setNametr,
          ownedInterfaceMethods <- setLocationtr
160     ),
        classtypetr : COOPNModel!"COOPNModel::ClassModule::ClassType"(
          name <- 'train'
        ),
        setLocationtr : COOPNModel!"COOPNModel::ClassModule::Method"(
165       name <- 'setLocation _'
        ),
        setNametr : COOPNModel!"COOPNModel::ClassModule::Method"(
          name <- 'setName _'
        ),
170


        ------------------------------
        -- World Structure Context --
        ------------------------------
175     ctx : COOPNModel!"COOPNModel::ContextModule::COOPNContext"(
          name <- 'CTX' + dsml.ownedWorld.worldName,
          ownedBody <- body,
          ownedInterface <- interface
        ),
180
        body: COOPNModel!"COOPNModel::ContextModule::Body"(
          ownedObjects <- dsml.ownedWorld.ownedSegments->collect(e | thisModule.
              ruleWaySegmentObj(e)),

          ownedAxiomTheorems <- axiom,
185
```

```
      ownedObjects <- dsml.ownedTrainEntities->collect(e | thisModule.
          ruleTrainEntitiesObj(e)),
      ownedObjects <- dsml.ownedWorld.ownedRailwayStations->collect(e |
          thisModule.ruleRailWayStationObj(e)),
      ownedAxiomTheorems <- dsml.ownedTrainEntities->collect(e | thisModule.
          executeActionPlanAxiom(e)),
      onwedBodyVariables <- ctxvariable

    ),
    axiom : COOPNModel!"COOPNModel::ContextModule::Axiom"(
      ownedRequiredEvent <- reqevent,
      ownedProvidedEvent <- providedevent
    ),
    reqevent : COOPNModel!"COOPNModel::ContextModule::RequiredEvent"(
      event <- 'initDSML '
    ),
    providedevent : COOPNModel!"COOPNModel::ContextModule::ProvidedEvent"(
      ownedProvidedEventAtom <-   providedeventatom
    ),
    providedeventatom : COOPNModel!"COOPNModel::ContextModule::Event"(
      event <- if (dsml.ownedTrainEntities.oclIsUndefined()) then
        ''
      else
        thisModule.createInitAxiom(dsml.ownedTrainEntities)
      endif
    ),

    interface : COOPNModel!"COOPNModel::ContextModule::Interface"(
      ownedInterfaceMethods <- ctxInterfaceMethodsw,
      ownedInterfaceMethods <- ctxInterfaceMethods
    ),
    ctxInterfaceMethodsw : COOPNModel!"COOPNModel::ContextModule::Method"(
      name <- 'initDSML'
    ),
    ctxInterfaceMethods : COOPNModel!"COOPNModel::ContextModule::Method"(
      name <- 'executeActionPlan _'
    ),
    ctxvariable : COOPNModel!"COOPNModel::ContextModule::Variable"(
      name <- 'vartrain'
    )
}

lazy rule executeActionPlanAxiom {
  from
    train : DSMLRailwaySystem!TrainEntity
  to
    axiom : COOPNModel!"COOPNModel::ContextModule::Axiom"(
      ownedRequiredEvent <- reqevent,
      ownedProvidedEvent <- providedevent,
      ownedCondition <- condition
    ),
    reqevent : COOPNModel!"COOPNModel::ContextModule::RequiredEvent"(
      event <- 'executeActionPlan ' + train.name
    ),
    providedevent : COOPNModel!"COOPNModel::ContextModule::ProvidedEvent"(
      ownedProvidedEventAtom <-   providedeventatom
    ),
    providedeventatom : COOPNModel!"COOPNModel::ContextModule::Event"(
      event <- if (train.ownedActionPlan.oclIsUndefined()) then
        ''
      else if (train.ownedActionPlan.ownedActions.oclIsUndefined()) then
        ''
```

```
245         else
              thisModule.createActionPlanAxiom(train.ownedActionPlan.ownedActions,
                   'obj' + train.name)
            endif
            endif
        ),
250     condition : COOPNModel!"COOPNModel::ContextModule::Condition"(
          ownedConditionAtom <-   conditionatom
        ),
        conditionatom : COOPNModel!"COOPNModel::ContextModule::Equation"(
          expression <- '(vartrain = ' + train.name + ') = true'
255     )
}

lazy rule  ruleTrainEntitiesObj {
  from
260     train : DSMLRailwaySystem!TrainEntity
  to
    obj : COOPNModel!"COOPNModel::ContextModule::Object"(
      name <- 'obj' + train.name
    )
265


}

lazy rule  ruleRailWayStationObj {
270   from
      rs : DSMLRailwaySystem!RailWayStation
  to
    obj : COOPNModel!"COOPNModel::ContextModule::Object"(
      name <- 'objTrainStation' + rs.stationName
275   )
}


lazy rule  ruleWaySegmentObj {
280   from
      ws : DSMLRailwaySystem!WaySegment
  to
    obj : COOPNModel!"COOPNModel::ContextModule::Object"(
      name <- 'objWaySegment' + ws.id.toString()
285   )
}
```

C.4 Robot Entity Transformation

Listing C.4. Complete Transformation of Robot Entities to CO-OPN

```
-- @atlcompiler atl2006
module RobotEntity; -- Module Template
create  robotsCOOPN : COOPNModel from robots : RobotEntityDSML;

5 helper  def : createActionPlanAxiom(gotoActions : Set(RobotEntityDSML!
    GoToAction), objname : String) : String =
    if (gotoActions.first().oclIsTypeOf(RobotEntityDSML!PickObject)) then
      '(' + objname + '.' + (' pickObject ' + (gotoActions.first().object.
          objectName.toString())) + ')' +
```

```
      if (gotoActions.size() = 1) then
      ''
      else
        ' .. ' +
        (thisModule.createActionPlanAxiom(gotoActions.excluding(gotoActions.
            first()), objname ))
      endif

    else if (gotoActions.first().oclIsTypeOf(RobotEntityDSML!Start)) then
      '(' + objname + '. start' + ')' +

      if (gotoActions.size() = 1) then
      ''
      else
        ' .. ' +
        (thisModule.createActionPlanAxiom(gotoActions.excluding(gotoActions.
            first()), objname ))
      endif
    else if (gotoActions.first().oclIsTypeOf(RobotEntityDSML!Stop)) then
      '(' + objname + '. stop' + ')' +

      if (gotoActions.size() = 1) then
      ''
      else
        ' .. ' +
        (thisModule.createActionPlanAxiom(gotoActions.excluding(gotoActions.
            first()), objname ))
      endif
    else
      ''
    endif
    endif
    endif;

rule ruleInit{
  from
    root : RobotEntityDSML!Root
  to
    ----------------
    -- Object --
    ----------------
    coopnclassstation : COOPNModel!"COOPNModel::ClassModule::COOPNClass"(
      name <- 'CLASSObject',
      ownedBody <- bodyobj,
      ownedInterface <- interfaceobj
    ),
    bodyobj : COOPNModel!"COOPNModel::ClassModule::Body"(
      ownedPlaces <- namesobj
    ),
    namesobj : COOPNModel!"COOPNModel::ClassModule::Place"(
      name <- 'name'
    ),
    interfaceobj : COOPNModel!"COOPNModel::ClassModule::Interface"(
      ownedInterfaceClassTypes <- classtypesobj
    ),
    classtypesobj : COOPNModel!"COOPNModel::ClassModule::ClassType"(
      name <- 'obj'
    ),

    ----------------------
    -- Robot Entity --
    ----------------------
```

```
    coopnclasstr : COOPNModel!"COOPNModel::ClassModule::COOPNClass"(
      name <- 'CLASSRobot',
      ownedBody <- bodyrobot,
      ownedInterface <- interfacerobot
    ),
    bodyrobot : COOPNModel!"COOPNModel::ClassModule::Body"(
      ownedPlaces <- locationrobot,
      ownedPlaces <- namerobot,
      ownedPlaces <- staterobot,
      ownedVariables <- varrobot,
      ownedAxiomTheorems <-axiomsrobot,
      ownedAxiomTheorems <-axiomsrobotstart,
      ownedAxiomTheorems <-axiomsrobotstop
    ),
    locationrobot : COOPNModel!"COOPNModel::ClassModule::Place"(
      name <- 'Location_'
    ),

    namerobot : COOPNModel!"COOPNModel::ClassModule::Place"(
      name <- 'name_'
    ),
    staterobot : COOPNModel!"COOPNModel::ClassModule::Place"(
      name <- 'state_'
    ),
    varrobot : COOPNModel!"COOPNModel::ClassModule::Variable"(
      name <- 'varrobotname'
    ),
    axiomsrobot : COOPNModel!"COOPNModel::ClassModule::Axiom"(
      name <- 'set name',
      ownedEvent <-eventrobot,
      ownedPost <- postrobot,
      ownedCondition <- conditionrobot

    ),
    axiomsrobotstart : COOPNModel!"COOPNModel::ClassModule::Axiom"(
      name <- 'start',
      ownedEvent <-eventrobotstart,
      ownedPost <- postrobotstart,
      ownedPre <- prerobotstart,
      ownedCondition <- conditionrobotstart

    ),
    axiomsrobotstop : COOPNModel!"COOPNModel::ClassModule::Axiom"(
      name <- 'stop',
      ownedEvent <- eventrobotstop,
      ownedPost <- postrobotstop,
      ownedPre <- prerobotstop,
      ownedCondition <- conditionrobotstop

    ),

    eventrobot : COOPNModel!"COOPNModel::ClassModule::Event"(
      event <- 'setName vartrobotname'
    ),
    postrobot : COOPNModel!"COOPNModel::ClassModule::Post"(
      ownedPostTerm <- postTermrobot
    ),
    postTermrobot : COOPNModel!"COOPNModel::ClassModule::Term"(
      expression <-  'name varrobotname'
    ),
    conditionrobot : COOPNModel!"COOPNModel::ClassModule::Condition"(
      ownedConditionAtom <-   conditionatomrobot
```

```
    ),
    conditionatomrobot : COOPNModel!"COOPNModel::ClassModule::Equation"(
      expression <- '(this = Self) = true'
    ),

    eventrobotstart : COOPNModel!"COOPNModel::ClassModule::Event"(
      event <- 'start'
    ),
    postrobotstart : COOPNModel!"COOPNModel::ClassModule::Post"(
      ownedPostTerm <- postTermrobotstart
    ),
    postTermrobotstart : COOPNModel!"COOPNModel::ClassModule::Term"(
      expression <-  'state moving'
    ),
    prerobotstart : COOPNModel!"COOPNModel::ClassModule::Pre"(
      ownedPreTerm <- preTermrobotstart
    ),
    preTermrobotstart : COOPNModel!"COOPNModel::ClassModule::Term"(
      expression <-  'state stopped'
    ),
    conditionrobotstart : COOPNModel!"COOPNModel::ClassModule::Condition"(
      ownedConditionAtom <-   conditionatomrobotstart
    ),
    conditionatomrobotstart : COOPNModel!"COOPNModel::ClassModule::Equation"
        (
      expression <- '(this = Self) = true'
    ),


    eventrobotstop : COOPNModel!"COOPNModel::ClassModule::Event"(
      event <- 'stop'
    ),
    postrobotstop : COOPNModel!"COOPNModel::ClassModule::Post"(
      ownedPostTerm <- postTermrobotstop
    ),
    postTermrobotstop : COOPNModel!"COOPNModel::ClassModule::Term"(
      expression <-  'state stopped'
    ),
    prerobotstop : COOPNModel!"COOPNModel::ClassModule::Pre"(
      ownedPreTerm <- preTermrobotstop
    ),
    preTermrobotstop : COOPNModel!"COOPNModel::ClassModule::Term"(
      expression <-  'state moving'
    ),
    conditionrobotstop : COOPNModel!"COOPNModel::ClassModule::Condition"(
      ownedConditionAtom <-   conditionatomrobotstop
    ),
    conditionatomrobotstop : COOPNModel!"COOPNModel::ClassModule::Equation"(
      expression <- '(this = Self) = true'
    ),


    interfacerobot : COOPNModel!"COOPNModel::ClassModule::Interface"(
      ownedInterfaceClassTypes <- classtyperobot,
      ownedInterfaceMethods <- setNamerobot,
      ownedInterfaceMethods <- pickobjectrobot,
      ownedInterfaceMethods <- startrobot,
      ownedInterfaceMethods <- stoprobot
    ),
    classtyperobot : COOPNModel!"COOPNModel::ClassModule::ClassType"(
      name <- 'robot'
    ),
```

```
    pickobjectrobot : COOPNModel!"COOPNModel::ClassModule::Method"(
      name <- 'pickObject _'
    ),
    setNamerobot : COOPNModel!"COOPNModel::ClassModule::Method"(
      name <- 'setName _'
    ),
    startrobot : COOPNModel!"COOPNModel::ClassModule::Method"(
      name <- 'start'
    ),
    stoprobot : COOPNModel!"COOPNModel::ClassModule::Method"(
      name <- 'stop'
    ),

    ----------------------------------------
    -- Dummy Context to create Objects--
    ----------------------------------------
    ctx : COOPNModel!"COOPNModel::ContextModule::COOPNContext"(
      name <- 'CTXRoot',
      ownedBody <- body,
      ownedInterface <- interface
    ),

    body: COOPNModel!"COOPNModel::ContextModule::Body"(
      ownedObjects <- root.ownedRobotEntities->collect(e | thisModule.
          ruleRobotEntitiesObj(e)),
      ownedObjects <- root.ownedObjects->collect(e | thisModule.
          ruleObjectsObj(e)),
      ownedAxiomTheorems <- root.ownedRobotEntities->collect(e | thisModule.
          executeActionPlanAxiom(e)),
      onwedBodyVariables <- ctxvariable
    ),
    interface : COOPNModel!"COOPNModel::ContextModule::Interface"(
      ownedInterfaceMethods <- ctxInterfaceMethods
    ),
    ctxInterfaceMethods : COOPNModel!"COOPNModel::ContextModule::Method"(
      name <- 'executeActionPlan _'
    ),
    ctxvariable : COOPNModel!"COOPNModel::ContextModule::Variable"(
      name <- 'varrobot'
    )
}

lazy rule executeActionPlanAxiom {
  from
    robot : RobotEntityDSML!RobotEntity
  to
    axiom : COOPNModel!"COOPNModel::ContextModule::Axiom"(
      name <- 'execute action plan',
      ownedRequiredEvent <- reqevent,
      ownedProvidedEvent <- providedevent,
      ownedCondition <- condition
    ),
    reqevent : COOPNModel!"COOPNModel::ContextModule::RequiredEvent"(
      event <- 'executeActionPlan ' + robot.name
    ),
    providedevent : COOPNModel!"COOPNModel::ContextModule::ProvidedEvent"(
      ownedProvidedEventAtom <-   providedeventatom
    ),
    providedeventatom : COOPNModel!"COOPNModel::ContextModule::Event"(
      event <- if (robot.ownedActionPlan.oclIsUndefined()) then
          ''
      else if (robot.ownedActionPlan.ownedActions.oclIsUndefined()) then
```

```
250           ''
          else
            thisModule.createActionPlanAxiom(robot.ownedActionPlan.ownedActions,
                'obj' + robot.name)
          endif
          endif
255     ),
        condition : COOPNModel!"COOPNModel::ContextModule::Condition"(
          ownedConditionAtom <-   conditionatom
        ),
        conditionatom : COOPNModel!"COOPNModel::ContextModule::Equation"(
260       expression <- '(varrobot = ' + robot.name + ') = true'
        )
    }

    lazy rule  ruleRobotEntitiesObj {
265   from
        robot : RobotEntityDSML!RobotEntity
      to
        obj : COOPNModel!"COOPNModel::ContextModule::Object"(
          name <- 'obj' + robot.name
270     )
    }

    lazy rule  ruleObjectsObj {
      from
275     o : RobotEntityDSML!Obj
      to
        obj : COOPNModel!"COOPNModel::ContextModule::Object"(
          name <- 'objObject' + o.objectName
        )
280 }
```

C.5 Robot System DSML Transformation

Listing C.5. Complete Transformation of Robot System DSML to CO-OPN

```
-- @atlcompiler atl2006
```

Appendix D

COSPEL Transformations

Listing D.1. Generic COSPEL ATL Transformation

```
-- @atlcompiler atl2006
module ControlSystemGenDSMLTransformation; -- Module Template
create  dvmCOOPN : COOPNModel from dvm :  GenericCospel, cfc : COOPNModel;

rule init {
  from
    c : COOPNModel!COOPNPackage,
    system : GenericCospel!System
  to
    coopnpackage: COOPNModel!COOPNPackage(
      name <- system.name,
      ownedModules <- system.ownedTypes->collect(e | thisModule.ruleType(c,e
          )),
      ownedModules <- system.ownedObjects->collect(e | thisModule.ruleObject
          (e,c))
    )
}

lazy rule ruleType {
  from
    c : COOPNModel!COOPNPackage,
    t : GenericCospel!Type

  to
    coopnclass: COOPNModel!"COOPNModel::ClassModule::COOPNClass"(
      name <- t.name,
      ownedBody <- body,
      ownedInterface <- interface
    ),
    ----------------------------
    ------- Interface -------
    interface : COOPNModel!"COOPNModel::ClassModule::Interface"(
      ownedInterfaceUses <-  Set{'Naturals', 'Booleans', 'Strings'}->collect
          (e | thisModule.ruleStandardUsesCLASS(e, c)),
      ownedInterfaceMethods <- t.fsm.ownedTransitions->collect(e |
          thisModule.ruleInMethodsCLASS(e)),
      ownedInterfaceMethods <- t.ownedEvents->collect(e | thisModule.
          ruleInEventMethodsCLASS(e)),
      ownedInterfaceClassTypes <- classtype
```

```
36      ),
        classtype : COOPNModel!"COOPNModel::ClassModule::ClassType" (
38        name <- t.name.toLower()
        ),
40      ----------------------------
        ---------- Body ----------
42      body: COOPNModel!"COOPNModel::ClassModule::Body"(
          ownedPlaces <- t.fsm.ownedStates->collect(e | thisModule.ruleInPlaces(
              e)),
44        ownedPlaces <- thisModule.ruleGeometry(c, t.geometry),
          ownedAxiomTheorems <- t.fsm.ownedTransitions->collect(e | thisModule.
              ruleInAxioms(e)),
46        ownedAxiomTheorems <- t.ownedEvents->collect(e | thisModule.
              ruleInEventAxioms(e)),
          ownedPlaces <- thisModule.ruleCoordinatesX((t.ownedBySystem.
              ownedObjects->any(o | o.type = t)).ownedCoordinates.ownedX),
48        ownedPlaces <- thisModule.ruleCoordinatesY((t.ownedBySystem.
              ownedObjects->any(o | o.type = t)).ownedCoordinates.ownedY),
          ownedPlaces <- thisModule.ruleCoordinatesZ((t.ownedBySystem.
              ownedObjects->any(o | o.type = t)).ownedCoordinates.ownedZ),
50        ownedInitial <- initial
        ),
52      initial : COOPNModel!"COOPNModel::ClassModule::Initial"(
            ownedInitialTerms <- t.fsm.findInitials()->collect(e | thisModule.
                ruleInTerms(e))
54      )
    }
56
    lazy rule ruleInEventMethodsCLASS{
58    from
        e : GenericCospel!Event
60    to
        m : COOPNModel!"COOPNModel::ClassModule::Method"(
62        name <- e.name
        )
64  }

66  lazy rule ruleInEventAxioms{
    from
68      e : GenericCospel!Event
    to
70      a : COOPNModel!"COOPNModel::ClassModule::Axiom"(
            ownedEvent <- ev
72      ),
        ev : COOPNModel!"COOPNModel::ClassModule::Event"(
74          event <- e.name
        )
76  }

78  lazy rule ruleInMethodsCLASS{
    from
80      t : GenericCospel!Transition
    to
82      m : COOPNModel!"COOPNModel::ClassModule::Method"(
          name <- t.name
84      )
    }
86
    lazy rule ruleInMethodsCTX{
88    from
        t : GenericCospel!Transition
90    to
```

```
    m : COOPNModel!"COOPNModel::ContextModule::Method"(
92      name <- t.name
    )
94 }

96 lazy rule ruleInPlaces{
  from
98    s : GenericCospel!State

100 to
    p : COOPNModel!"COOPNModel::ClassModule::Place" (
102     name <- s.name

104   )
}
106
lazy rule ruleInTerms{
108 from
    s : GenericCospel!State
110 to
    t: COOPNModel!"COOPNModel::ClassModule::Term" (
112     expression <- s.name + ' @'
    )
114 }

116 lazy rule ruleInAxioms {
  from t : GenericCospel!Transition
118
  to
120   a : COOPNModel!"COOPNModel::ClassModule::Axiom" (
      name <-t.name,
122   ownedPre <- pre,
      ownedPost <- post,
124   ownedEvent <- e
    ),
126 pre : COOPNModel!"COOPNModel::ClassModule::Pre" (
          ownedPreTerm <-  preterm
128 ),
    preterm : COOPNModel!"COOPNModel::ClassModule::Term" (
130         expression <- t.source.name + ' @'
    ),
132 post : COOPNModel!"COOPNModel::ClassModule::Post" (
          ownedPostTerm <-   postterm
134 ),
    postterm : COOPNModel!"COOPNModel::ClassModule::Term" (
136         expression <- t.target.name + ' @'
    ),
138 e : COOPNModel!"COOPNModel::ClassModule::Event" (
          event <- t.name
140 )
}
142
lazy rule ruleStandardUsesCLASS{
144 from
    s : String,
146 c : COOPNModel!COOPNPackage
  to
148 u : COOPNModel!"COOPNModel::ClassModule::Use" (
      name <- s,
150   usedModuleForClass <- c.getADT(s)
    )
152
```

```
}

lazy rule ruleStandardUsesCTX{
  from
    s : String,
    c : COOPNModel!COOPNPackage
  to
    u : COOPNModel!"COOPNModel::ContextModule::Use" (
        name <- s,
        usedModuleForContext <- c.getADT(s)
    )
}
--------------------
--- geometry -------
--------------------
 lazy rule ruleGeometry {
  from
    c : COOPNModel!COOPNPackage,
    geometry : GenericCospel!Geometry
  to
    geomplace: COOPNModel!"COOPNModel::ClassModule::Place"(
      name <- geometry.URL,
      ownedPlaceTypeElements <- inPlaceType
    ),

    inPlaceType :  COOPNModel!"COOPNModel::ClassModule::TypeElement"(
      typeElementType <- c.getADT('Strings').ownedInterface.
          ownedInterfaceSorts->first()
    )
}
--------------------
---- Object --------
--------------------
lazy rule ruleObject {
  from
    obj : GenericCospel!Object,
    c : COOPNModel!COOPNPackage
  to
    ctx : COOPNModel!"COOPNModel::ContextModule::COOPNContext"(
      name <- 'CTX' + obj.name,
      ownedBody <- body,
      ownedInterface <- interface
    ),
    ---------------------------
    ---------- Body ----------
    body: COOPNModel!"COOPNModel::ContextModule::Body"(
      ownedObjects <-  inObjects
    ),
    inObjects:  COOPNModel!"COOPNModel::ContextModule::Object" (
      name <- obj.name
    ),
    ---------------------------
    ------- Interface -------
    interface : COOPNModel!"COOPNModel::ContextModule::Interface"(
      ownedInterfaceUses <-  Set{'Naturals', 'Booleans', 'Strings'}->collect
          (e | thisModule.ruleStandardUsesCTX(e, c)),
      ownedInterfaceMethods <- obj.ownedBySystem.ownedTypes->any(t | t = obj
          .type).fsm.ownedTransitions->collect(e | thisModule.
          ruleInMethodsCTX(e))
    )
}
-------------------------
```

```
-- cooedinates ------
-------------------------
lazy rule ruleCoordinatesY {
  from
    c : GenericCospel!y
  to
    coordy: COOPNModel!"COOPNModel::ClassModule::Place"(
      name <- 'y'
    )
}

lazy rule ruleCoordinatesX {
  from
    c : GenericCospel!x
  to
    coordx: COOPNModel!"COOPNModel::ClassModule::Place"(
      name <- 'x'
    )
}

lazy rule ruleCoordinatesZ {
  from
    c : GenericCospel!z
  to
    coordz: COOPNModel!"COOPNModel::ClassModule::Place"(
      name <- 'z'
    )
}

helper context GenericCospel!Type def : findGeometry() : GenericCospel!
    Geometry =
  self.geometry;


helper context GenericCospel!FSM def : findInitials() : Set(GenericCospel!
    State) =
  self.ownedStates->select(s | s.isInitial=true);


helper context COOPNModel!COOPNPackage def : getADT(nameADT : String) :
    COOPNModel!COOPNADT =
    self.ownedModules->any(m |
      m.name = nameADT and m.oclIsTypeOf(COOPNModel!COOPNADT)
    );

helper context COOPNModel!COOPNClass def : getTypeFromClass() : COOPNModel!
    ClassType =
  self.ownedInterface.ownedInterfaceClass->first();

helper context COOPNModel!COOPNPackage def : getAllADT() :OrderedSet (
    COOPNModel!"COOPNModel::ADTModule") =
  self.packageContainsModules->iterate( m ; elements : OrderedSet (
      COOPNModel!"COOPNModel::ADTModule") =
  OrderedSet {} |
    if(m.oclIsTypeOf(COOPNModel!"COOPNModel::ADTModule"))
      then
        elements.append (m)
      else
        true
    endif
);
```

Listing D.2. Event Model ATL Transformation

```
--@atlcompiler atl2006
module EventModelTransformation; -- Module Template
create  coopn : COOPNModel from eventModel :  EventModel;

rule init {
  from
    root : EventModel!Root
  to
    interface : COOPNModel!"COOPNModel::ClassModule::Interface"(
      ownedInterfaceMethods <- root.ownedEvents->collect(e|thisModule.
          ruleInEventMethods(e))
    ),
    body : COOPNModel!"COOPNModel::ClassModule::Body"(
      ownedAxiomTheorems <- root.ownedEvents->collect(e|e.createAxioms())
    )
}

-- rule for events
lazy rule ruleInEventMethods {
  from
    e : EventModel!Event
  to
    m : COOPNModel!"COOPNModel::ClassModule::Method"(
      name <- e.name
    )
}

-- Iterate over conditions for creating axioms
helper context EventModel!Event def : createAxioms() : Set(COOPNModel!"
    COOPNModel::ClassModule::Axiom") =
  self.ownedConditions->collect(c|thisModule.ruleInConditionAxioms(c));


-- Rules for creating axioms from conditions

lazy rule ruleInConditionAxioms { --this is for conditions with no triggered
     events and transitions
  from
    c : EventModel!Condition
  to
    a : COOPNModel!"COOPNModel::ClassModule::Axiom"(
        ownedPre <- pre,
        ownedPost <- post,
        ownedEvent <- event
    ),
    event : COOPNModel!"COOPNModel::ClassModule::Event" (
            event <- c.ownedByEvent.name
    ),
    pre : COOPNModel!"COOPNModel::ClassModule::Pre" (
            ownedPreTerm <-  preterm
    ),
    preterm : COOPNModel!"COOPNModel::ClassModule::Term" (
            expression <- c.precondition
    ),
    post : COOPNModel!"COOPNModel::ClassModule::Post" (
            ownedPostTerm <-  postterm
    ),
    postterm : COOPNModel!"COOPNModel::ClassModule::Term" (
            expression <- c.postcondition
    )
```

```
58 }

60 lazy rule axiomsWithTransitionOrEvent extends ruleInConditionAxioms { --this
       is for conditions with transitions only or Event Only
   from
62     c : EventModel!Condition (
         not (c.transition.oclIsUndefined() and c.triggerEvent.oclIsUndefined()
             ) -- they are NOT both undefined
64       and (c.transition.oclIsUndefined() or c.triggerEvent.oclIsUndefined())
               -- and one of them is undefined
     )
66   to
       a : COOPNModel!"COOPNModel::ClassModule::Axiom"(
68       ownedSynchronization <- synchronization
     ),
70     synchronization : COOPNModel!"COOPNModel::ClassModule::Synchronization"(
         synchro <-
72           if (c.triggerEvent.oclIsUndefined())
             then
74             c.transition.name
           else
76             c.triggerEvent.name
           endif
78   )
}
80
lazy rule axiomsWithTransitionAndEvent extends ruleInConditionAxioms { --
    this is for conditions with both transition and Event
82   from
       c : EventModel!Condition (
84       not (c.transition.oclIsUndefined() or c.triggerEvent.oclIsUndefined())
             -- they are both defined
     )
86   to
         a : COOPNModel!"COOPNModel::ClassModule::Axiom"(
88       ownedSynchronization <- synchronization
     ),
90     synchronization : COOPNModel!"COOPNModel::ClassModule::Synchronization"(
         synchro <- c.transition.name+' // '+c.triggerEvent.name
92   )
}
```

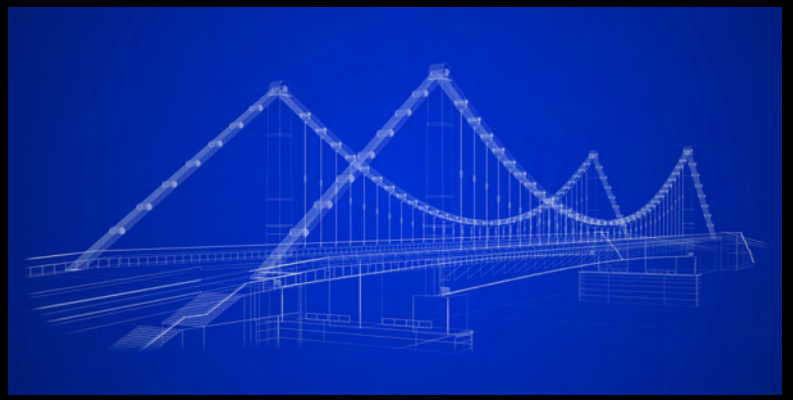
Appendix E

Thesis Presentation

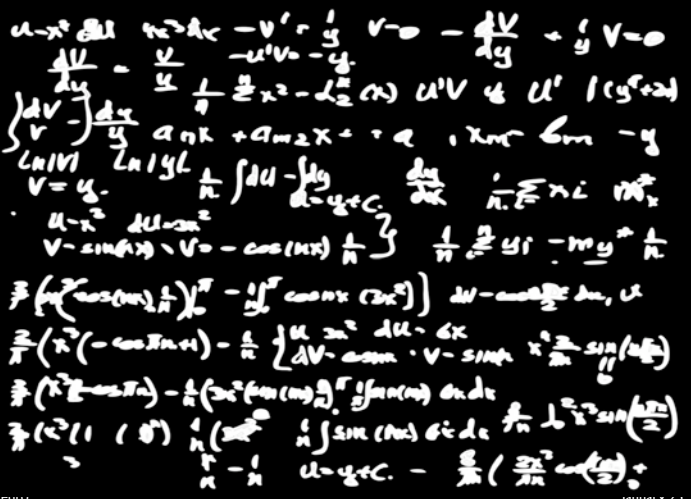
A Systematic Language Engineering Approach for Prototyping Domain Specific Modelling Languages

Luis Pedro
University of Geneva

Project



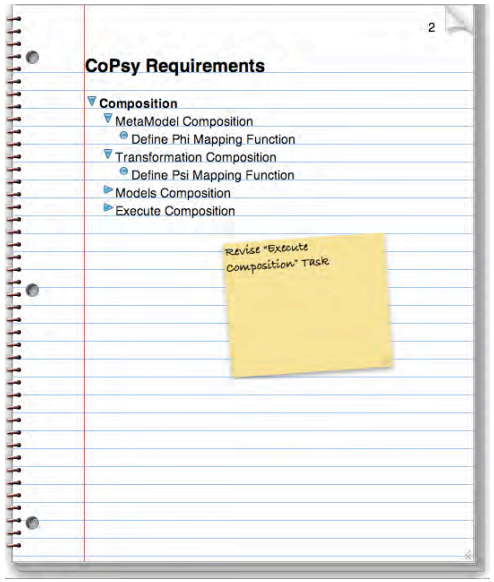
Model



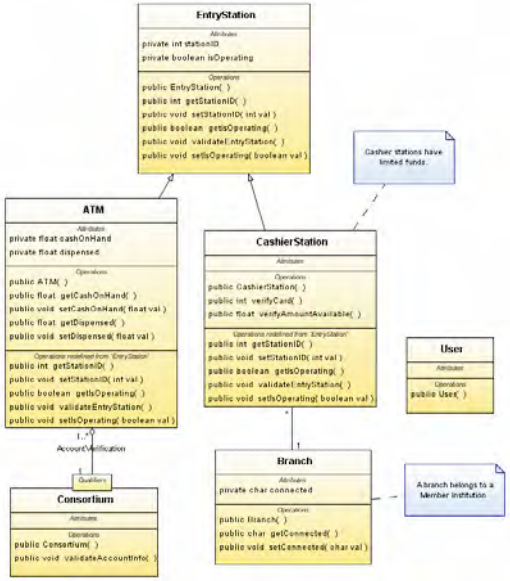
Prototype

Build

Build



Analysis/ Requirements



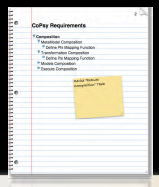
Model

Prototype

```
private EPackage mepMetamodel;

/**
 * the base reference Metamodel.
 */
private EPackage mrefMetamodel;

 * Phi mapping function.
private PhiMappingFunction mphi;

 * Name of the file containing the new metamodel.
private String mOutputMetamodelName;

 * Name of the path for the new metamodel.
private String mOutputFilePath;

 * The output metamodel EPackage.
private EPackage mOutputEPackage = EcoreFactory.eINSTANCE.createEPackage();

/**
 * Copsy Utils instance.
 */
private CopsyUtils mUtils;

 * Public constructor.
public MetaModelCompositionExecution(EPackage ep, EPackage ref,

 * Executes metamodel composition.
public Boolean executeComposition() {

 * Initializes List with elements providing. From phi function
private Boolean init() {

 * Write the composition to a file.
public void writeComposition() throws IOException {
```

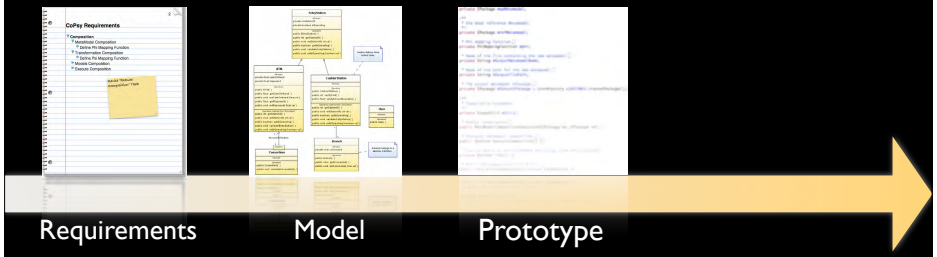

Implementation

```
119+    * Executes metamodel composition.
123-   public Boolean executeComposition() {
124        if (this.init()) {
125
126            /*
127             * for (String e : mfpEClass.keySet()) { EClass eeee = (EClass)
128             * mfpEClass.get(e); System.out.println(eeee.getName()); }
129             */
130
131            if (mepMetamodel == null || mrefMetamodel == null) {
132                return false;
133            } else {
134                mUtils.log("Starting Metamodel Composition Algorithm...");
135                mUtils.log("Add EClasses in the Refence metamodel:");
136                Copier copier = new Copier();
137                for (Object objClassifier : mrefMetamodel.getEClassifiers()) {
138                    EClassifier ec = (EClassifier) objClassifier;
139                    if (ec instanceof EClass) {
140                        EObject eClonedObject = copier.copy(ec);
141                        mOutputEPackage.getEClassifiers().add(
142                                (EClass) eClonedObject);
143                        mUtils.log("\tElement from Reference metamodel "
144                                + ec.getName() + " (" + ec.getEPackage().getName()
145                                + ")");
146
147                    }
148                }
                copier.copyReferences();
```

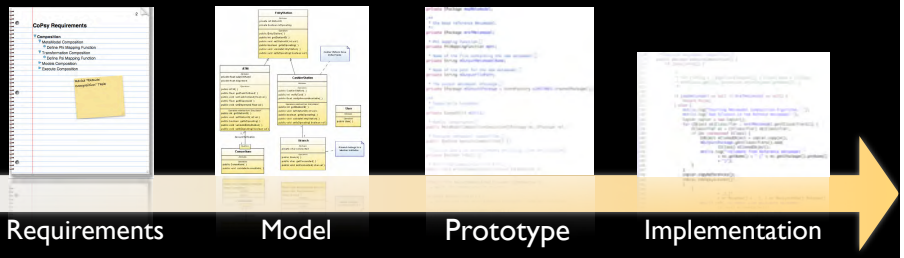



Requirements

Requirements Model



Requirements Model Prototype

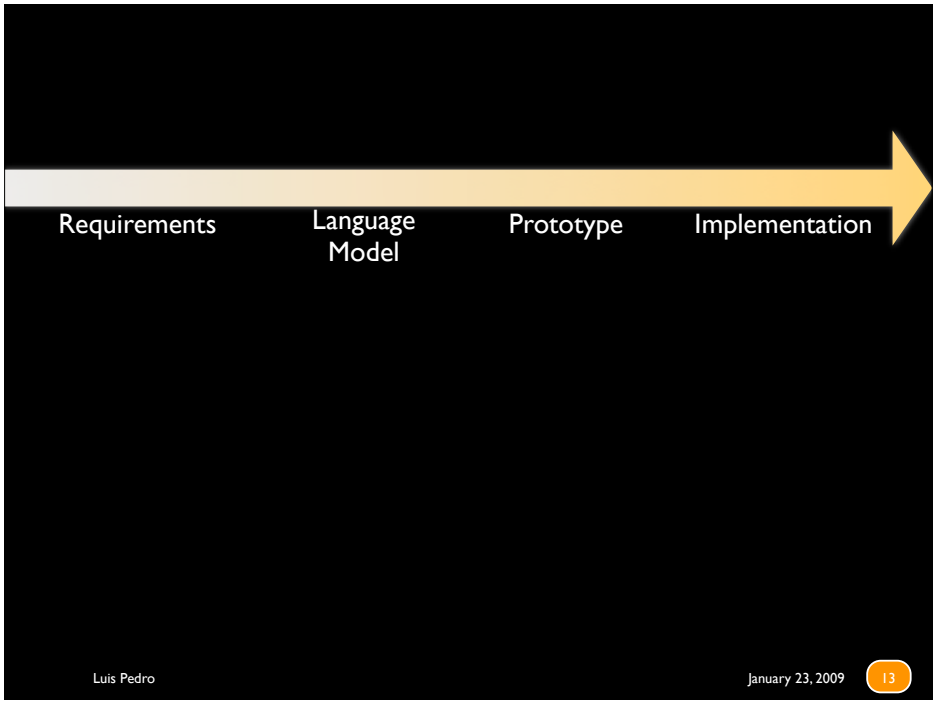


Requirements Model Prototype Implementation

Language Engineering?

A language that is used will be changed.

- Meir M. Lehman



What Language?

Domain Specific Language

Luis Pedro January 23, 2009 15

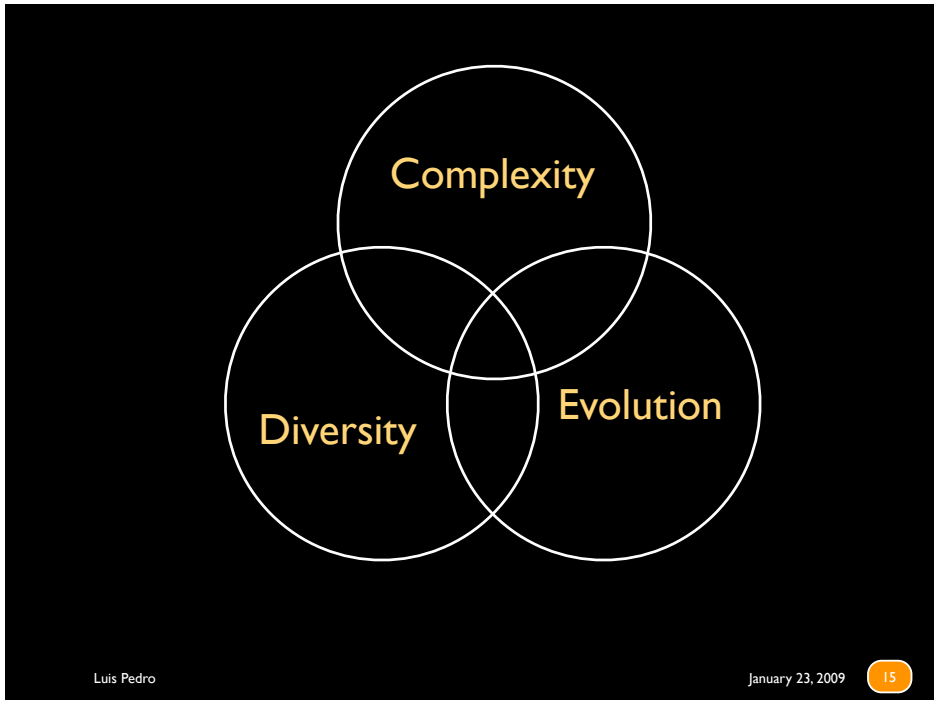
Complexity

Luis Pedro January 23, 2009 15

Complexity

Diversity

Luis Pedro January 23, 2009 15


Complexity
Diversity
Evolution
Luis Pedro
January 23, 2009
15


Abstraction through rich concepts
helps to manage complexity


Integration of multiple languages
helps to manage diversity


Flexible and modular languages
help managing evolution

Goals and Motivation

Goals and Motivation

Goals and Motivation

Re-use for Faster Language Development

Goals and Motivation

Re-use for Faster Language Development

Modularity

Goals and Motivation

Re-use for Faster Language Development

Manage Language Complexity

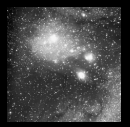
Modularity

Goals and Motivation

Re-use for Faster Language Development

Incremental DSML development

Manage Language Complexity

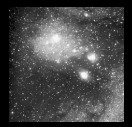


Modularity

Goals and Motivation

Re-use for Faster Language Development

Manage Language Complexity



Modularity

Incremental DSML development

State of The Art

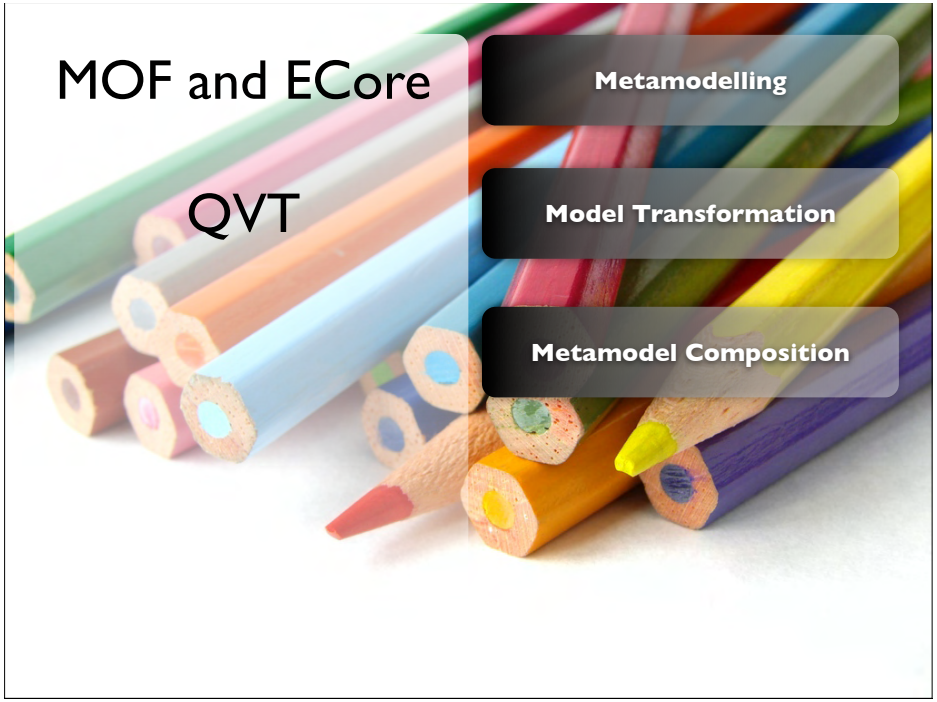

State of The Art: Methodologies

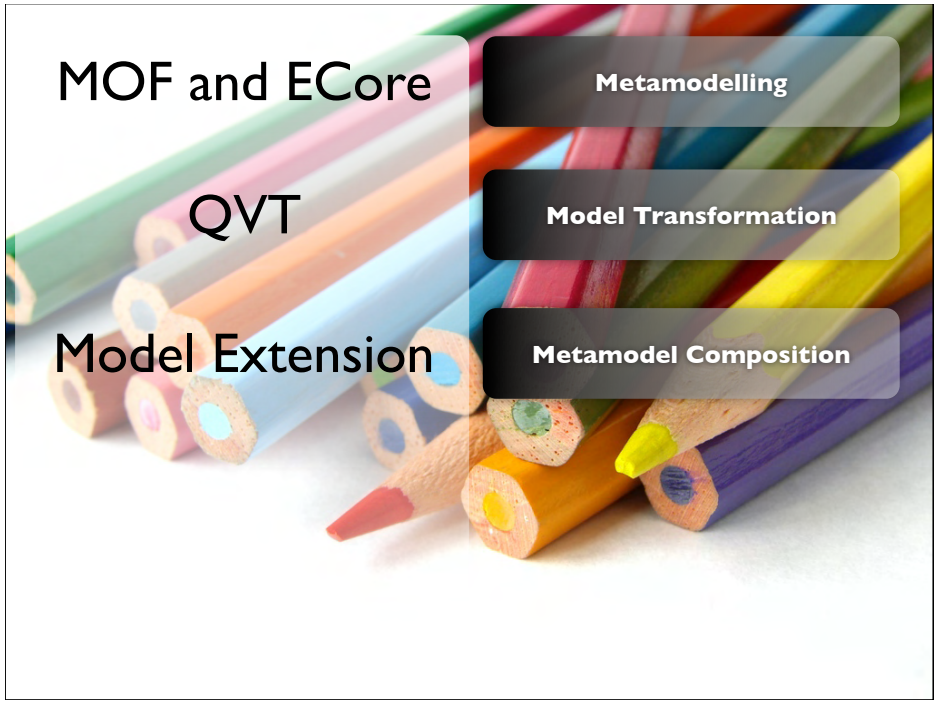

Metamodelling

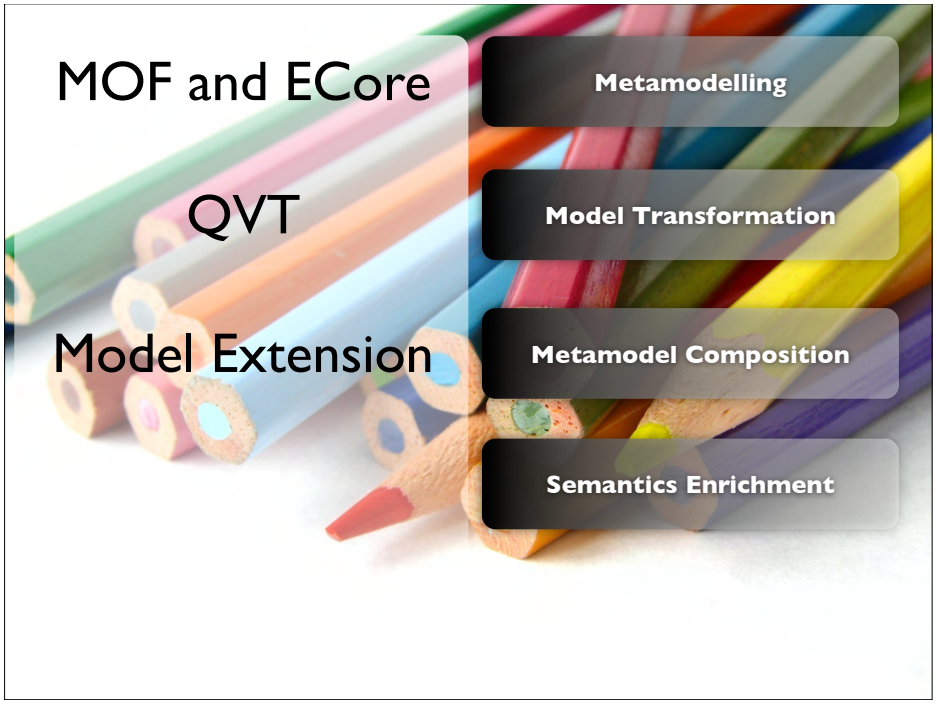

MOF and ECore
Metamodelling

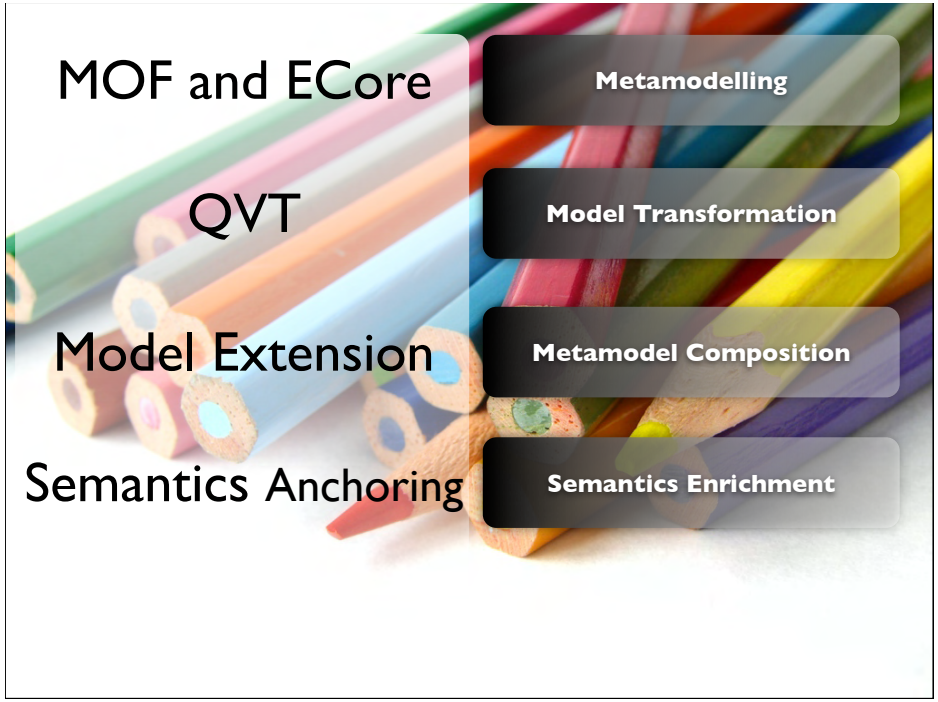

MOF and ECore
Metamodelling
Model Transformation

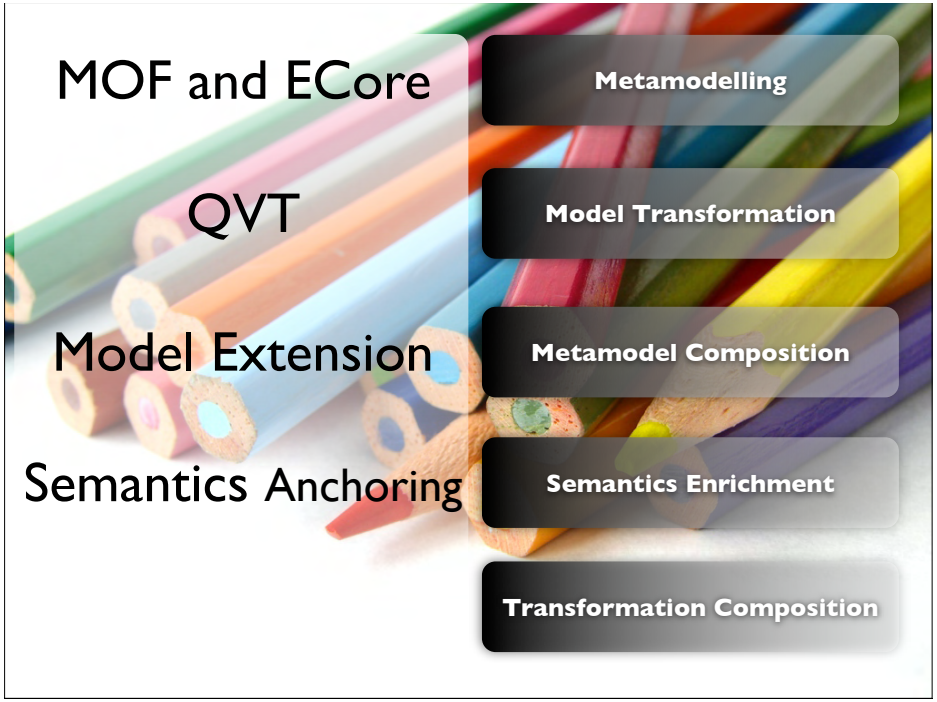

MOF and ECore
QVT
Metamodelling
Model Transformation

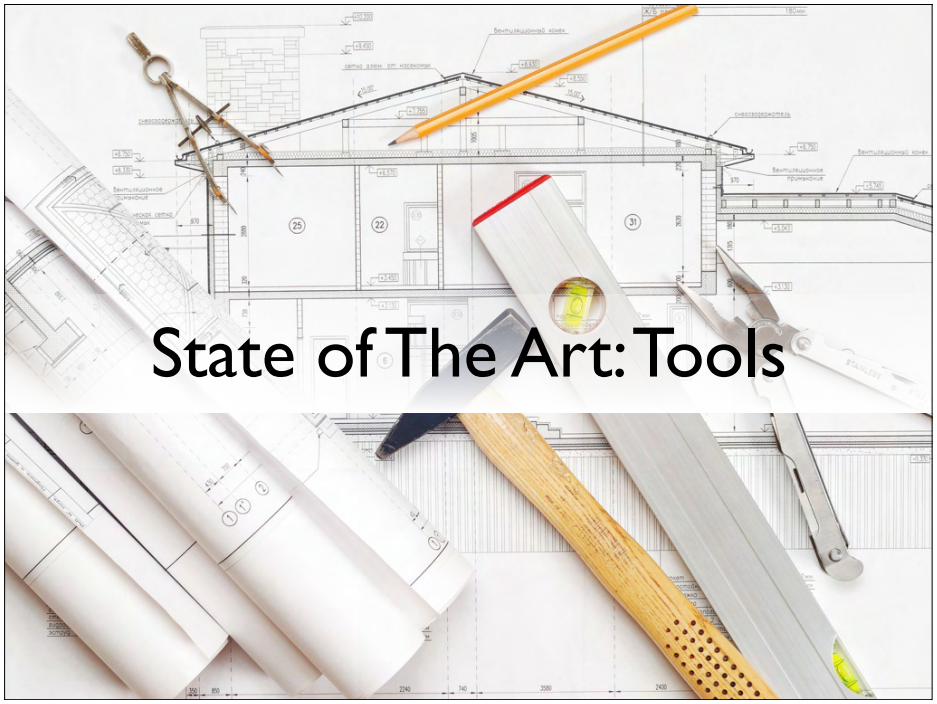

MOF and ECore
QVT
Metamodelling
Model Transformation
Metamodel Composition


MOF and ECore
QVT
Model Extension
Metamodelling
Model Transformation
Metamodel Composition


MOF and ECore
QVT
Model Extension
Metamodelling
Model Transformation
Metamodel Composition
Semantics Enrichment


MOF and ECore
QVT
Model Extension
Semantics Anchoring
Metamodelling
Model Transformation
Metamodel Composition
Semantics Enrichment


MOF and ECore
QVT
Model Extension
Semantics Anchoring
Metamodelling
Model Transformation
Metamodel Composition
Semantics Enrichment
Transformation Composition


State of The Art: Tools

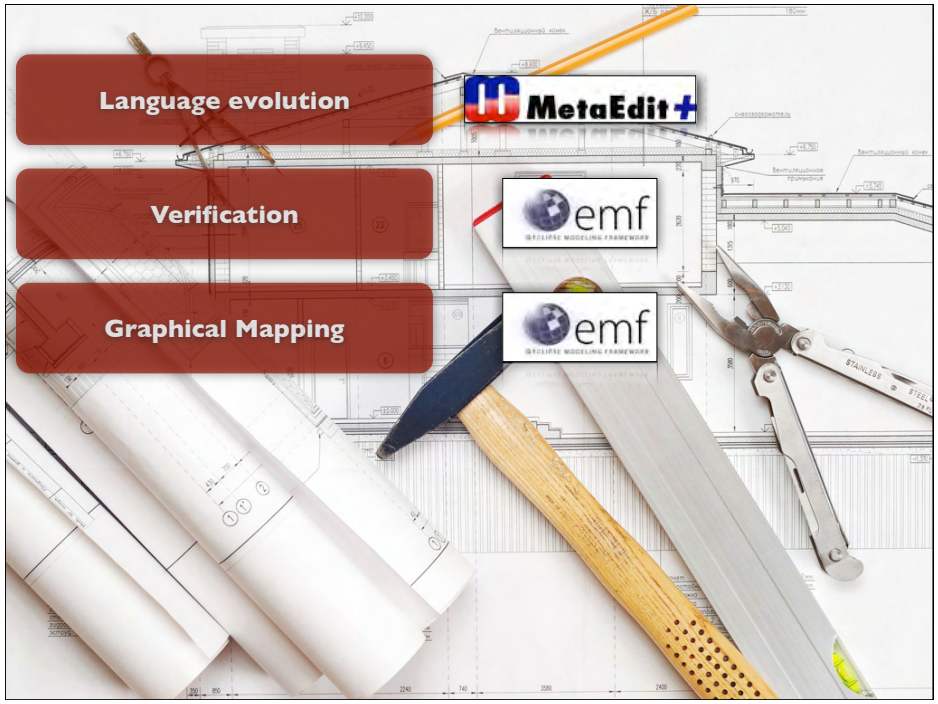

Language evolution

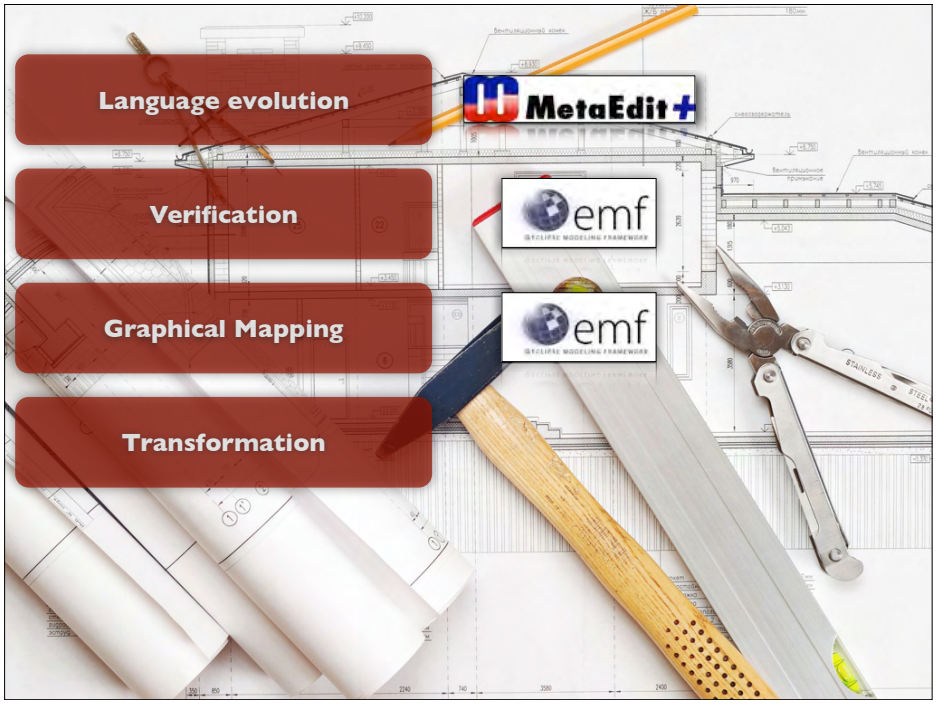

Language evolution
MetaEdit+

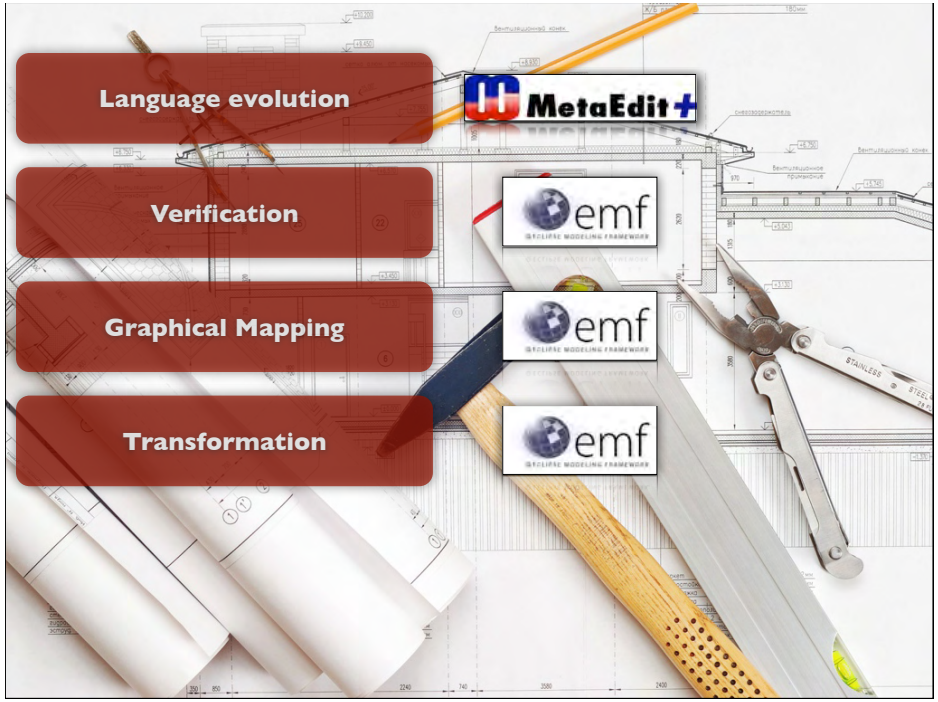

Language evolution
MetaEdit+
Verification

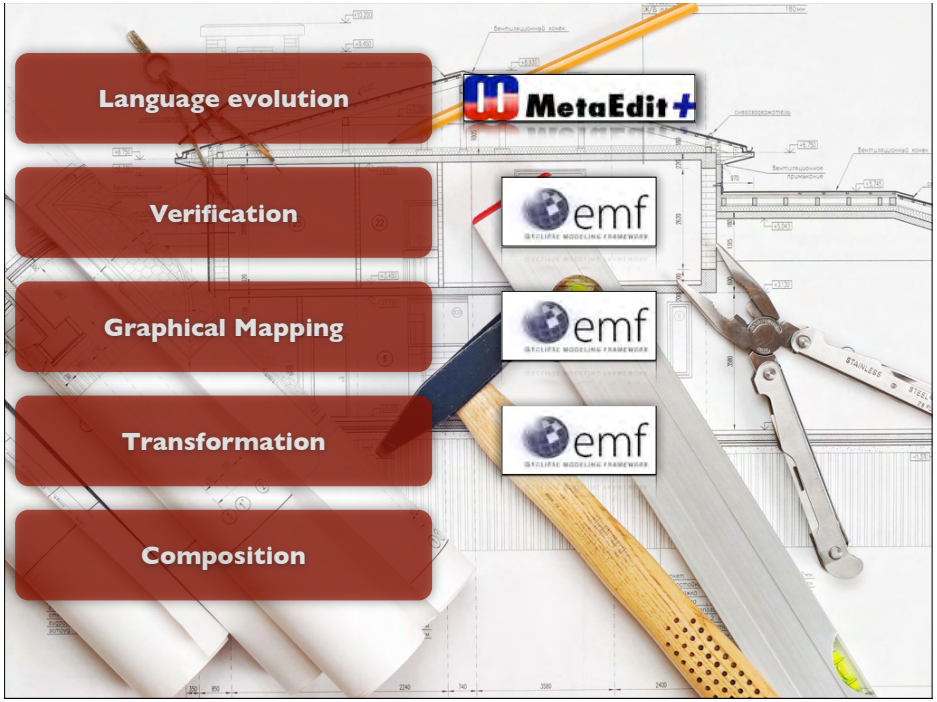

Language evolution
MetaEdit+
Verification
emf

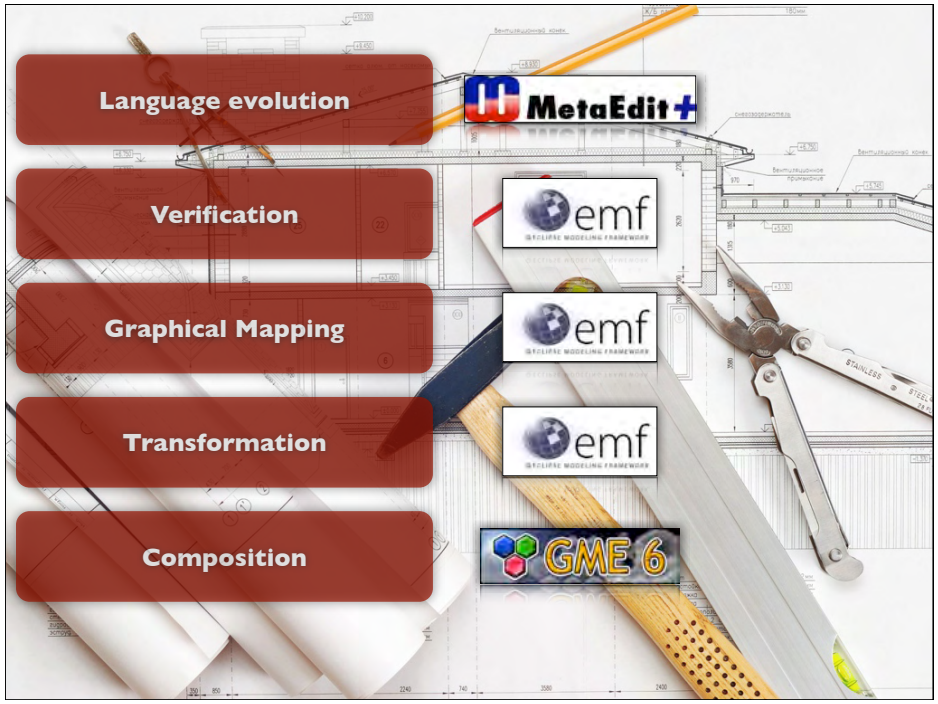

Language evolution
MetaEdit+
Verification
emf
Graphical Mapping

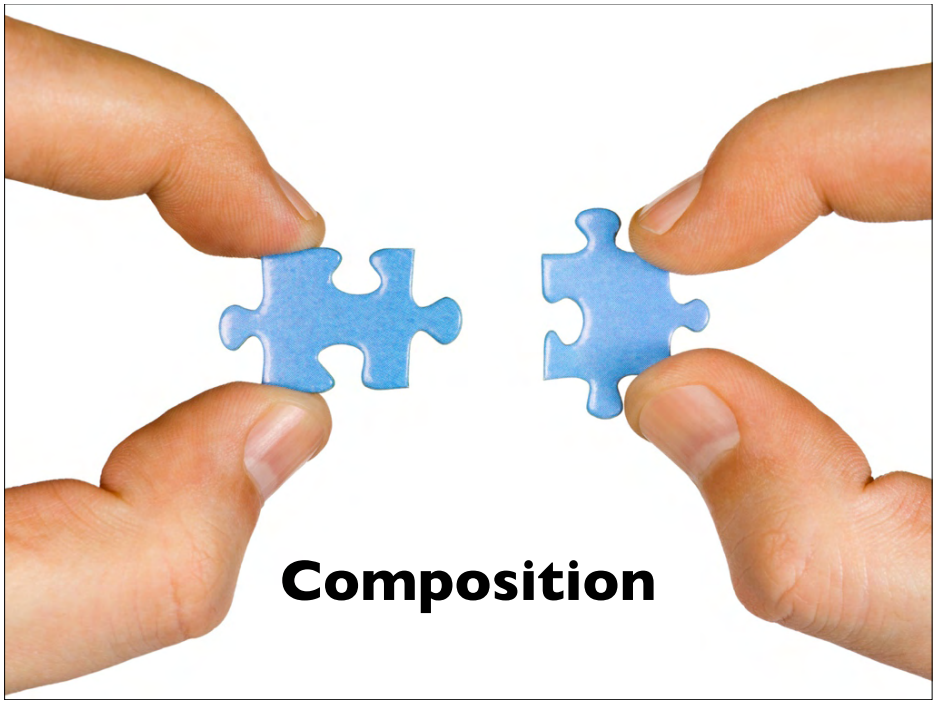

Language evolution
MetaEdit+
Verification
emf
Graphical Mapping
emf

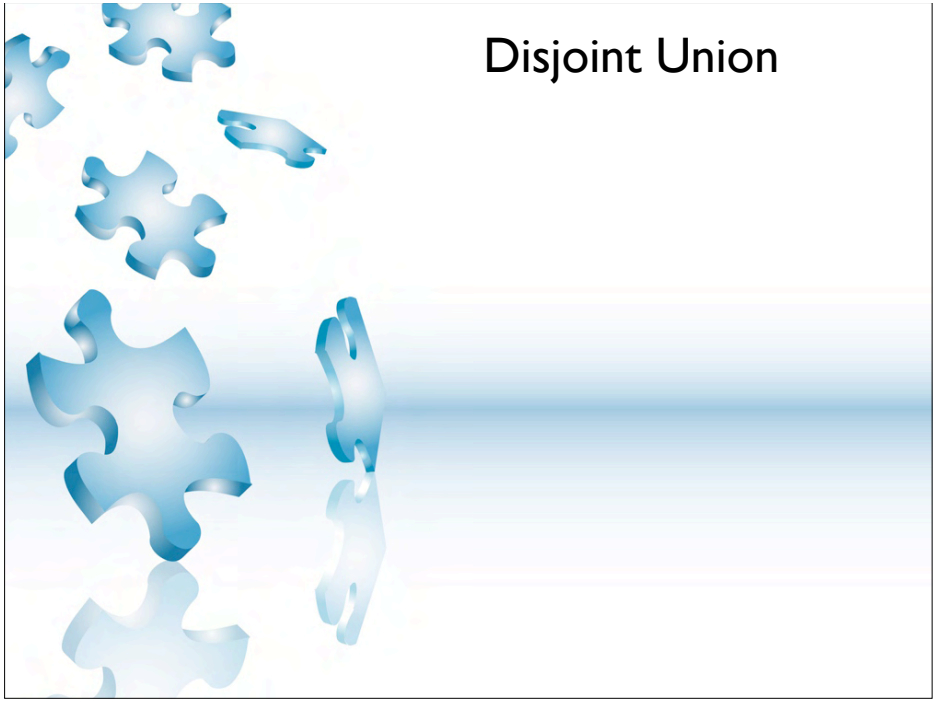

Language evolution
MetaEdit+
Verification
emf
Graphical Mapping
emf
Transformation

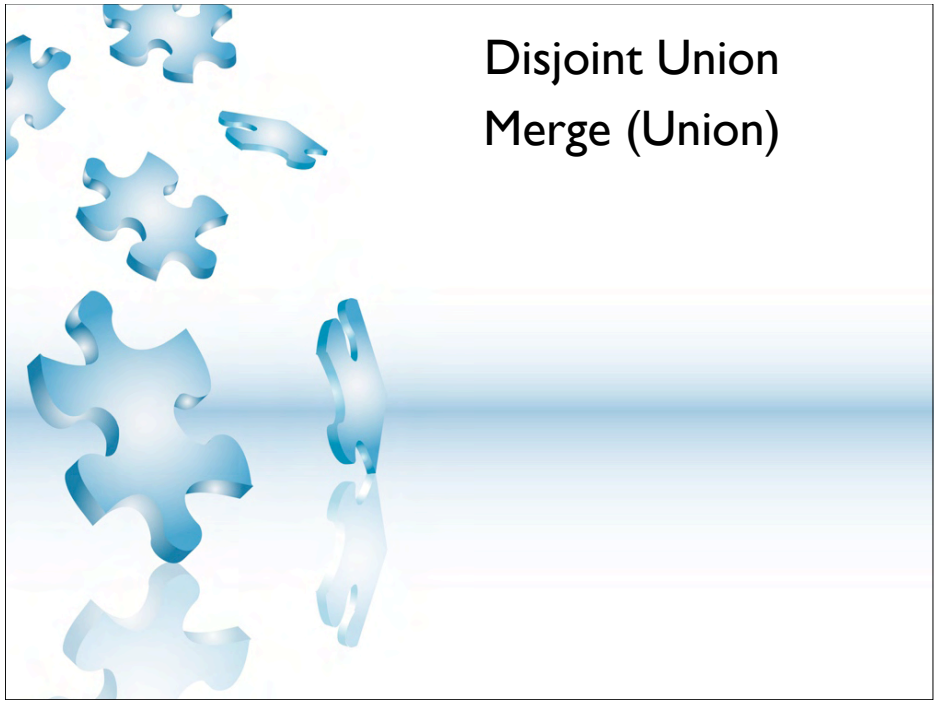

Language evolution
MetaEdit+
Verification
emf
Graphical Mapping
emf
Transformation
emf

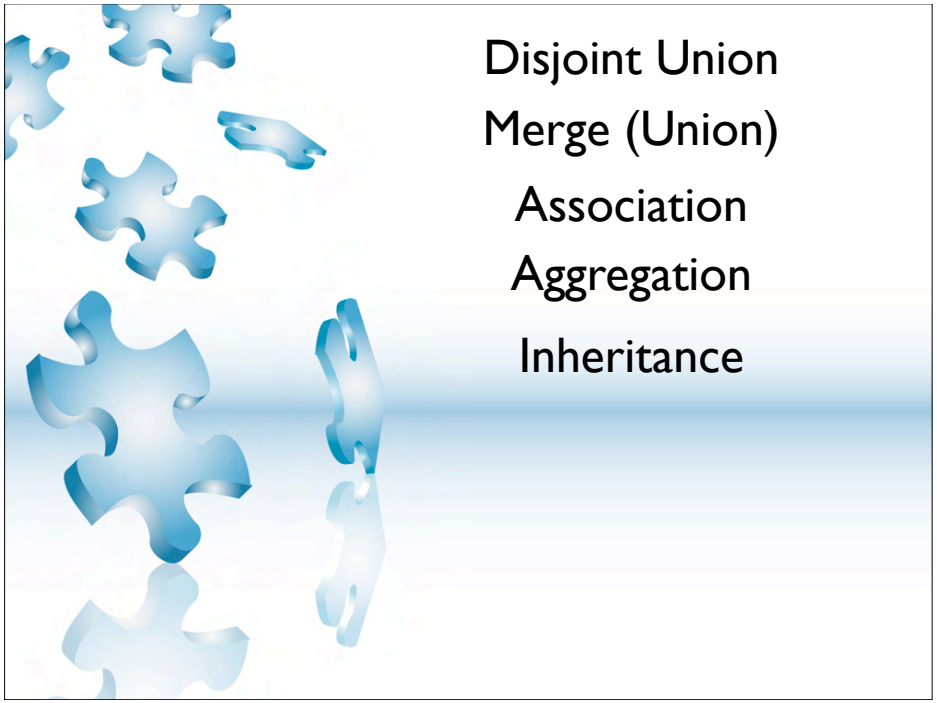

Language evolution
MetaEdit+
Verification
emf
Graphical Mapping
emf
Transformation
emf
Composition

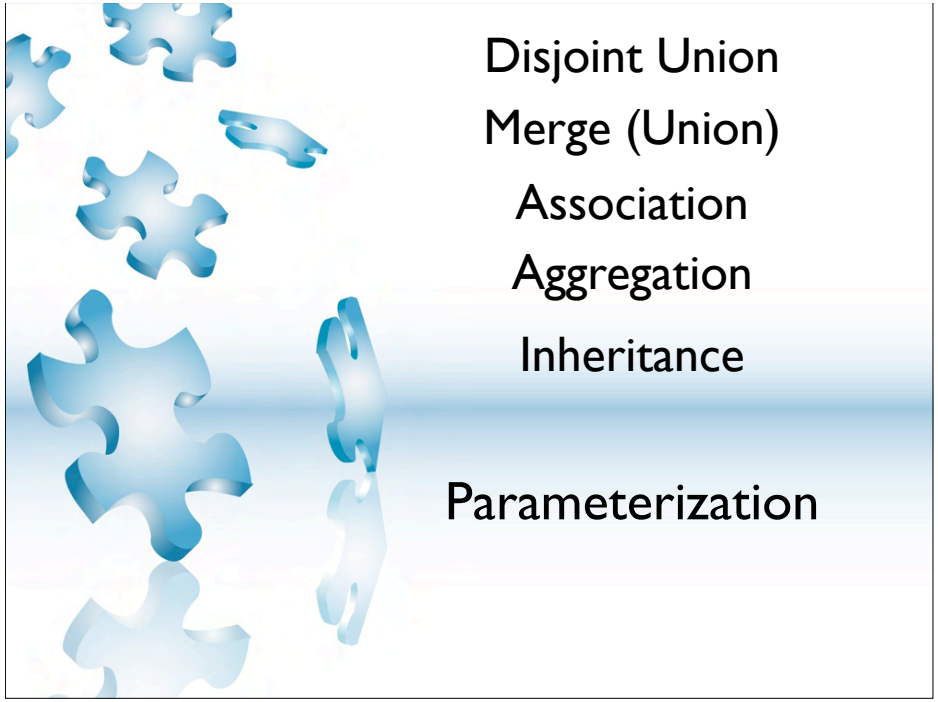

Language evolution
MetaEdit+
Verification
emf
Graphical Mapping
emf
Transformation
emf
Composition
GME 6

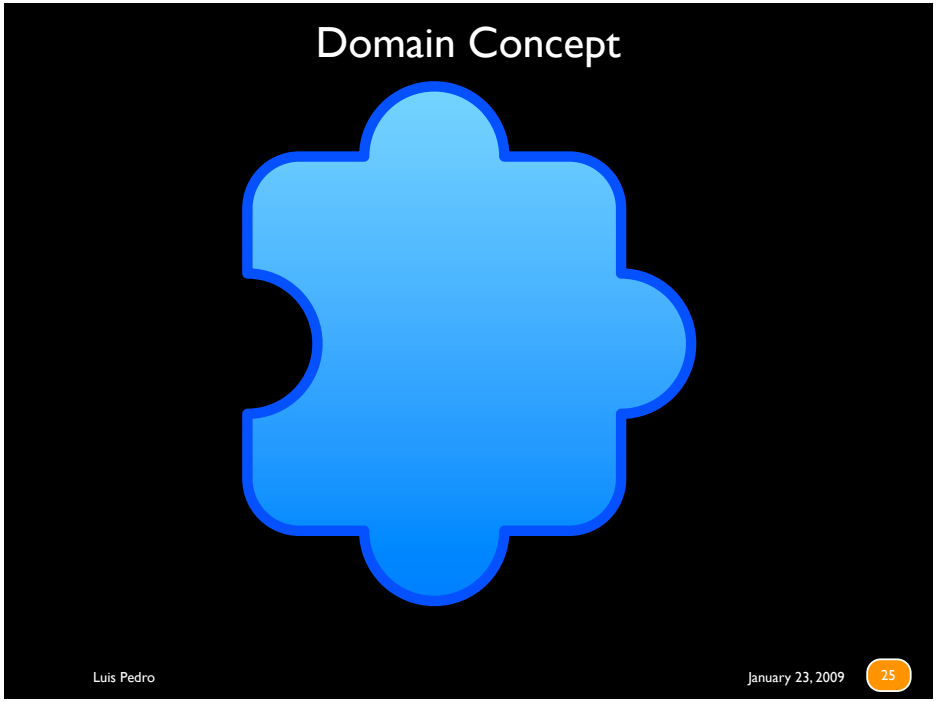

Composition


Disjoint Union

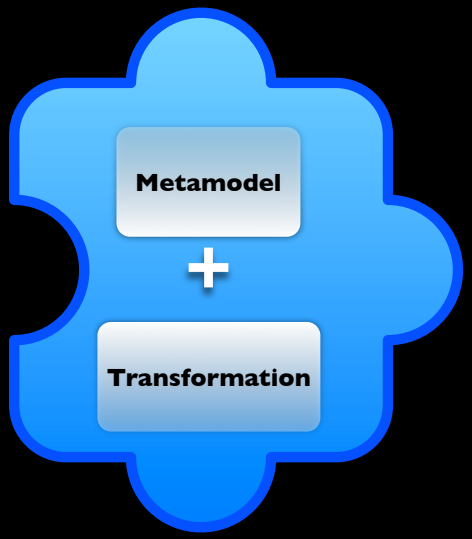

Disjoint Union
Merge (Union)


Disjoint Union
Merge (Union)
Association
Aggregation
Inheritance

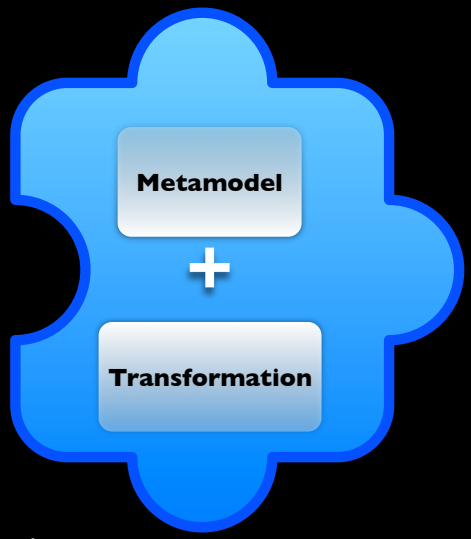

Disjoint Union
Merge (Union)
Association
Aggregation
Inheritance
Parameterization


Domain Concept
Luis Pedro
January 23, 2009
25

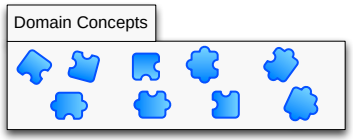
Domain Concept

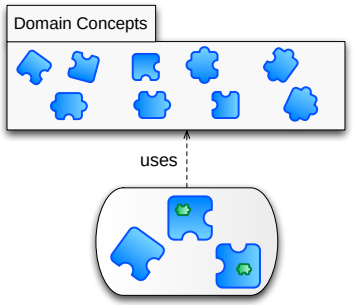


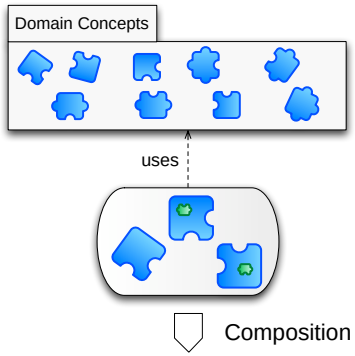
Domain Concept

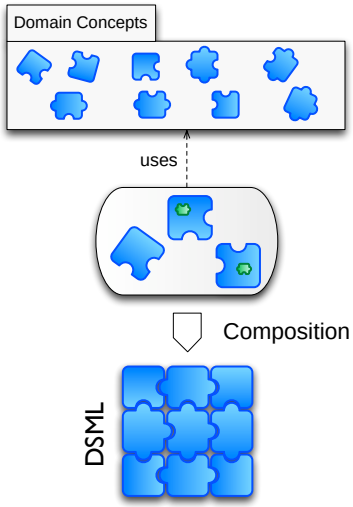


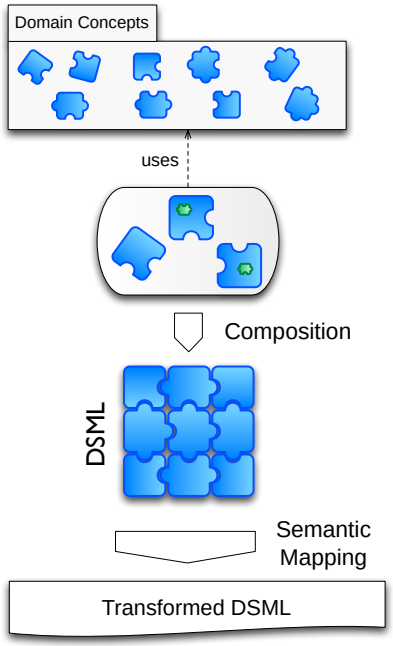
A semantic block capturing domain knowledge



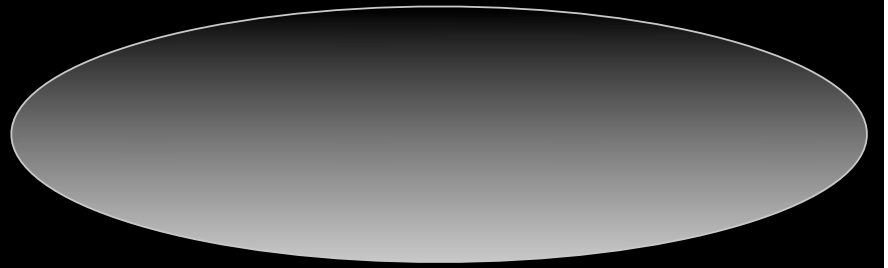

Domain Concepts
uses


Domain Concepts
uses
Composition


Domain Concepts
uses
Composition
DSML

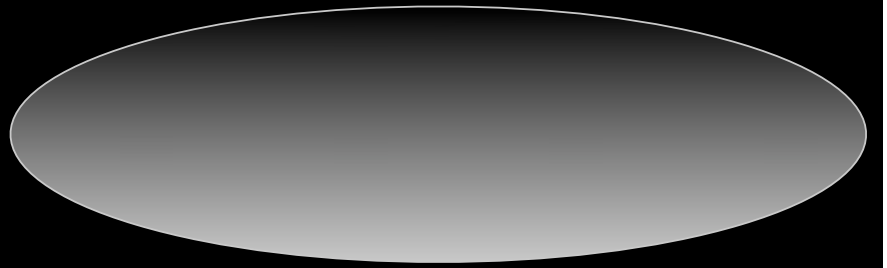

Domain Concepts
uses
Composition
DSML
Semantic Mapping
Transformed DSML

Semantics, Prototyping and Animation



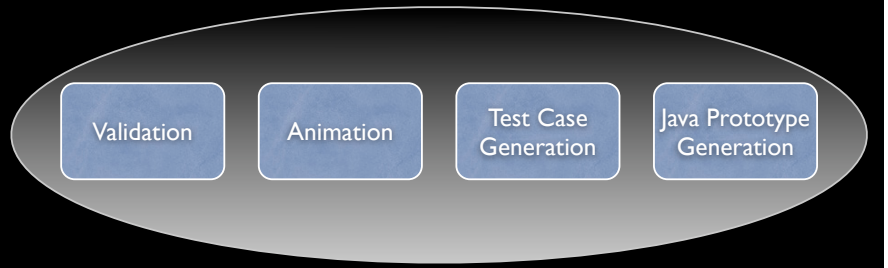
Semantics, Prototyping and Animation

Chosen Platform for Semantic Mapping:
CO-OPN Builder



Semantics, Prototyping and Animation

Chosen Platform for Semantic Mapping:
CO-OPN Builder

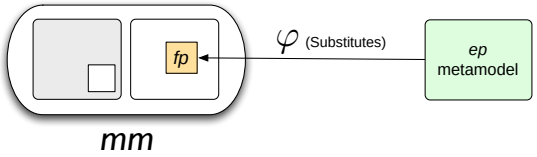


fp formal parameter

ep effective parameter

fp formal parameter

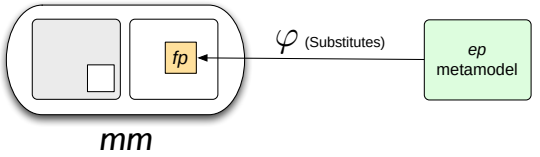
Before Parameterization



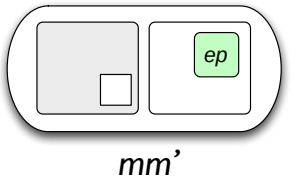
ep effective parameter

fp formal parameter

Before Parameterization



After Parameterization



ep effective parameter

Metamodel Composition

$$mm' = mm[fp \overset{\varphi}{\leftarrow} ep, F_{fp}]$$

Metamodel Composition

$$mm' = mm[fp \overset{\varphi}{\leftarrow} ep, F_{fp}]$$

fp is the formal parameter

Metamodel Composition

$$mm' = mm[fp \overset{\varphi}{\leftarrow} ep, F_{fp}]$$

fp is the formal parameter

ep is the effective parameter

Metamodel Composition

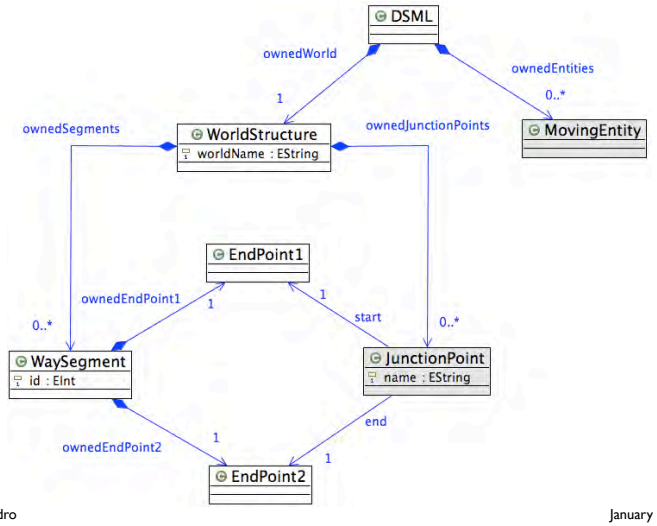
$$mm' = mm[fp \overset{\varphi}{\leftarrow} ep, F_{fp}]$$

fp is the formal parameter

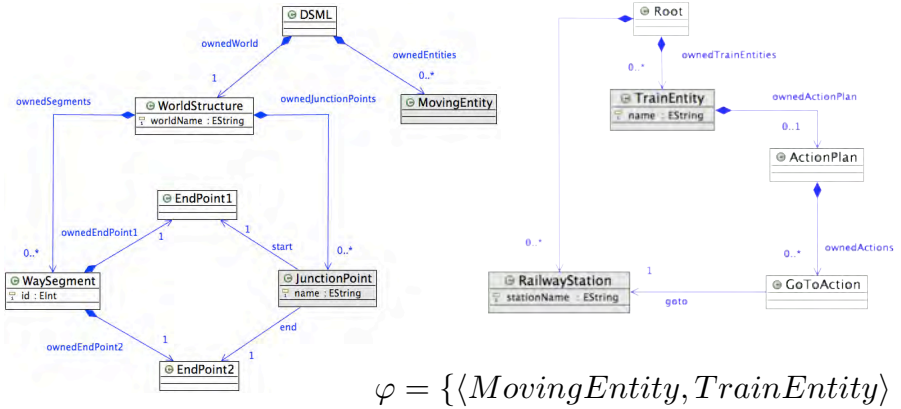
ep is the effective parameter

φ is a total function creating a map between elements of *fp* and *ep*

Moving Entity DSML Metamodel

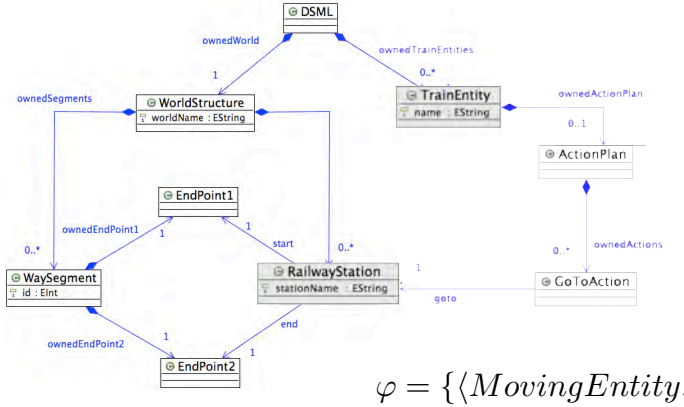


Train Entity DSML Metamodel



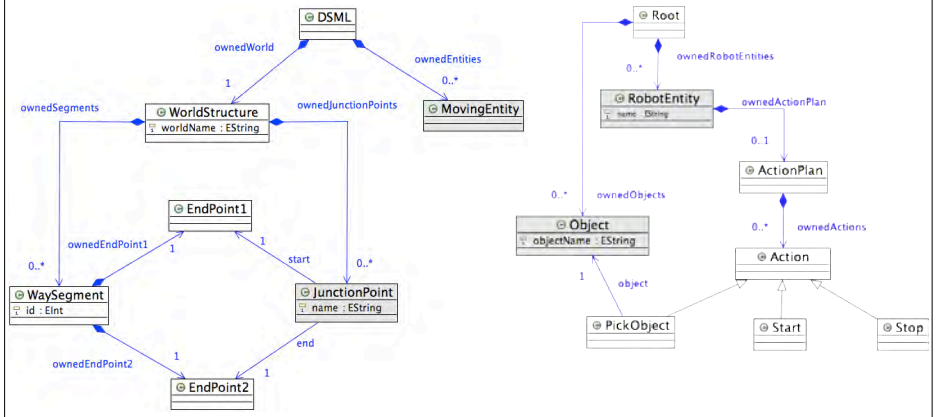
$$\varphi = \{\langle MovingEntity, TrainEntity\rangle$$
$$\langle JunctionPoint, RaiwayStation\rangle\}$$

Train Entity DSML Metamodel



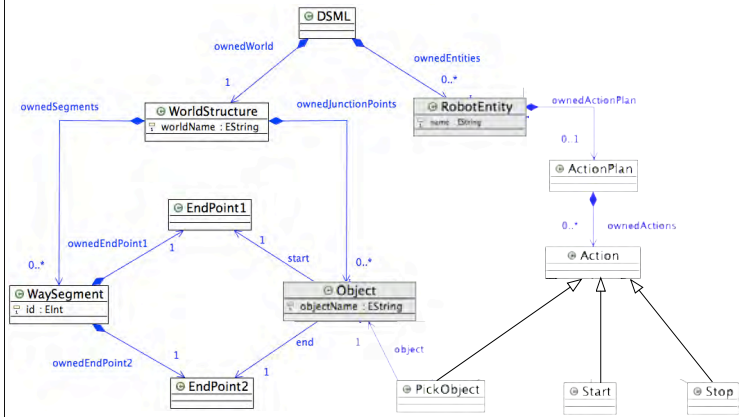
$$\varphi = \{\langle MovingEntity, TrainEntity\rangle \langle JunctionPoint, RaiwayStation\rangle\}$$

Robot Entity DSML Metamodel



$$\varphi = \{\langle MovingEntity, RobotEntity\rangle \langle JunctionPoint, Object\rangle\}$$

Robot Entity DSML Metamodel



$$\varphi = \{\langle MovingEntity, RobotEntity\rangle \langle JunctionPoint, Object\rangle\}$$

Transformation

Transformation

Semantics Mapping

Transformation

Semantics Mapping

Set of Rules Describing a Transformation

Transformation

Semantics Mapping

Set of Rules Describing a Transformation

Model Transformation Language

Transformation Composition

$$Tr_{mm'} = Tr_{mm}[Tr_{fp} \xleftarrow{\varphi,\psi} Tr_{ep}]$$

Metamodel Composition

$$mm' = mm[fp \xleftarrow{\varphi} ep, F_{fp}]$$

Transformation Composition

$$Tr_{mm'} = Tr_{mm}[Tr_{fp} \xleftarrow{\varphi,\psi} Tr_{ep}]$$

Tr_{fp} template transformation for *fp*

Metamodel Composition

$$mm' = mm[fp \xleftarrow{\varphi} ep, F_{fp}]$$

Transformation Composition

$$Tr_{mm'} = Tr_{mm}[Tr_{fp} \xleftarrow{\varphi,\psi} Tr_{ep}]$$

Tr_{fp} template transformation for *fp*
Tr_{ep} template transformation for *ep*

Metamodel Composition

$$mm' = mm[fp \xleftarrow{\varphi} ep, F_{fp}]$$

Luis Pedro January 23, 2009 34

Transformation Composition

$$Tr_{mm'} = Tr_{mm}[Tr_{fp} \xleftarrow{\varphi,\psi} Tr_{ep}]$$

Tr_{fp} template transformation for *fp*
Tr_{ep} template transformation for *ep*
φ: $Dom(Tr_{fp}) \longrightarrow Dom(Tr_{ep})$

Metamodel Composition

$$mm' = mm[fp \xleftarrow{\varphi} ep, F_{fp}]$$

Luis Pedro January 23, 2009 34

Transformation Composition

$$Tr_{mm'} = Tr_{mm}[Tr_{fp} \xleftarrow{\varphi,\psi} Tr_{ep}]$$

Tr_{fp} template transformation for *fp*
Tr_{ep} template transformation for *ep*
φ: $Dom(Tr_{fp}) \longrightarrow Dom(Tr_{ep})$
ψ: $Cod(Tr_{fp}) \longrightarrow Cod(Tr_{ep})$

Metamodel Composition

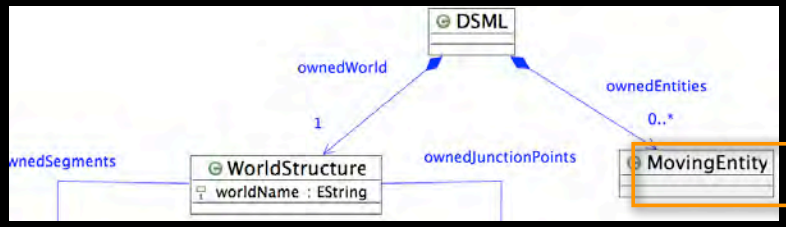
$$mm' = mm[fp \xleftarrow{\varphi} ep, F_{fp}]$$

Luis Pedro January 23, 2009 34

What Happens to Transformations

Transformation Element is a Leaf

Transformation Element is a Leaf



Transformation Element is a Leaf

Tr_{mm}

Tr_{fp1}

Tr_{fp2}

...

Tr_{fpn}

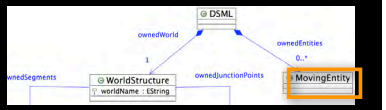
Tr_{ep}

Tr_{ep1}

Tr_{ep2}

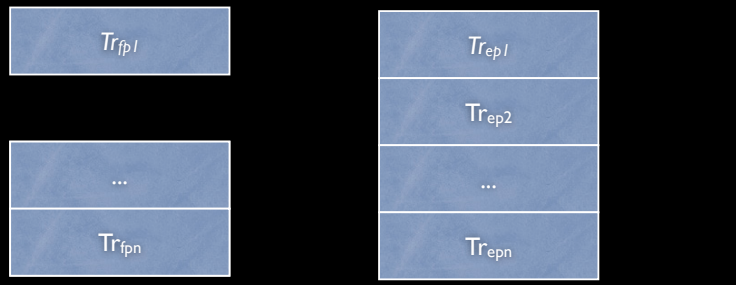
...

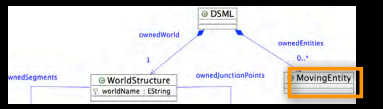
Tr_{epn}



Transformation Element is a Leaf

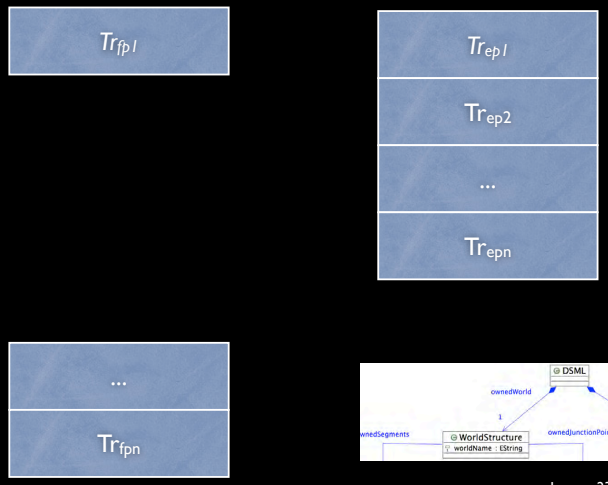
Tr_{mm} Tr_{ep}





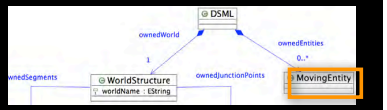
Transformation Element is a Leaf

Tr_{mm} Tr_{ep}



Transformation Element is a Leaf

$$Tr_{fp} \xleftarrow{\varphi,\psi} Tr_{ep}$$
$$\psi(Tr_{fp2}, Tr_{ep})$$



Transformation Element is a Leaf

Tr_{fp} $ep = Tr_{TrainEntity}$

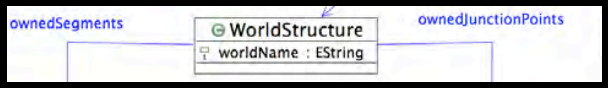
```
rule ruleJunctionPoint {
   from
      jp : MovingEntity!JunctionPoint
   to
      cl : COOPNMetaModel!COOPNClass(...)
}

rule ruleMovingEntity {
   from
      me : MovingEntity!MovingEntity
   to
      cl : COOPNMetaModel!COOPNClass(...)
}
```

```
rule ruleRailWayStation {
   from
      rs : TrainEntity!RailWayStation
   to
      cl : COOPNMetaModel!COOPNClass,
      pl: COOPNMetaModel!"COOPNClass!Place"
      (. . .)
}

(. . .)
```

$$\psi = \{\langle Tr_{fp}, Tr_{TrainEntity} \rangle)$$

General Case

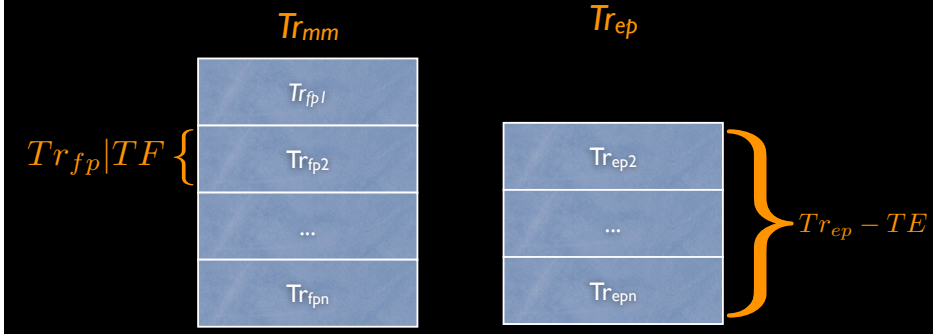
General Case



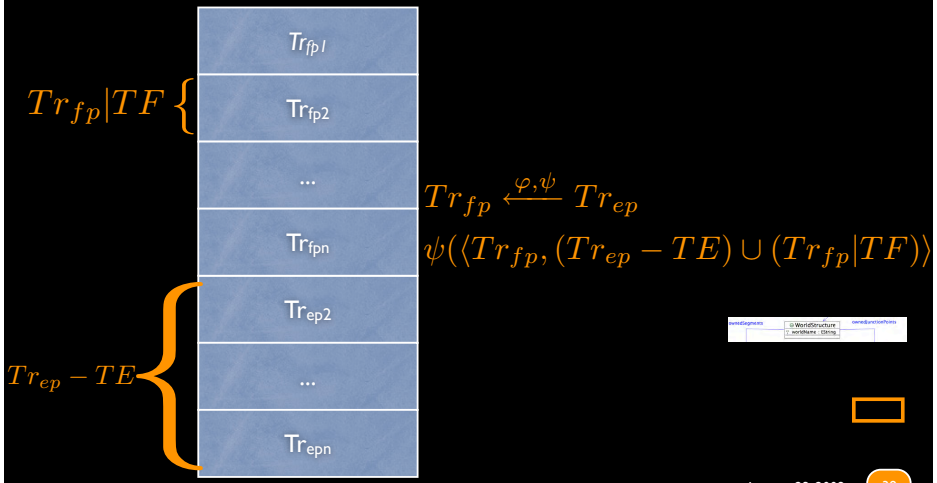
General Case

General Case

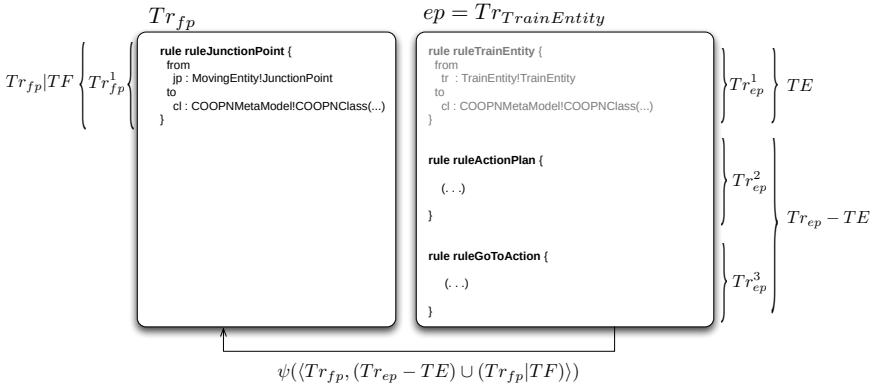
General Case

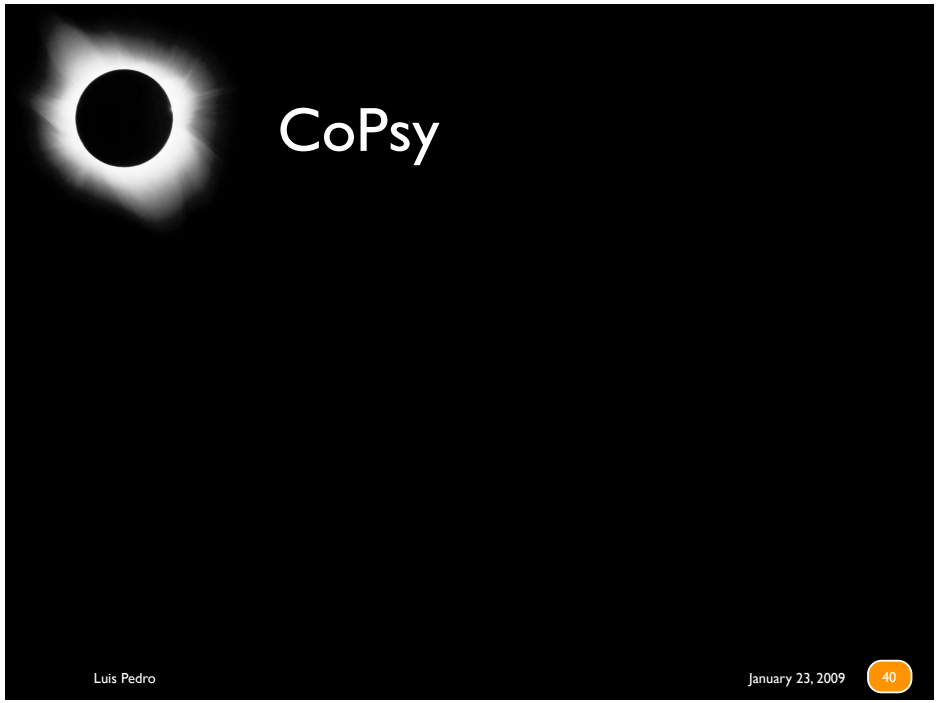


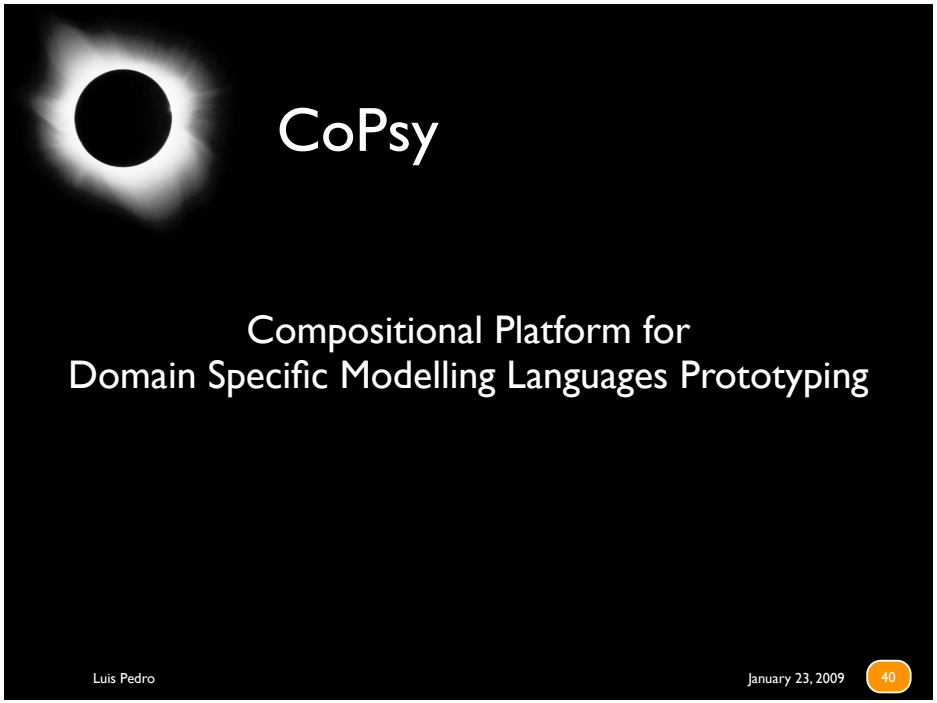
General Case

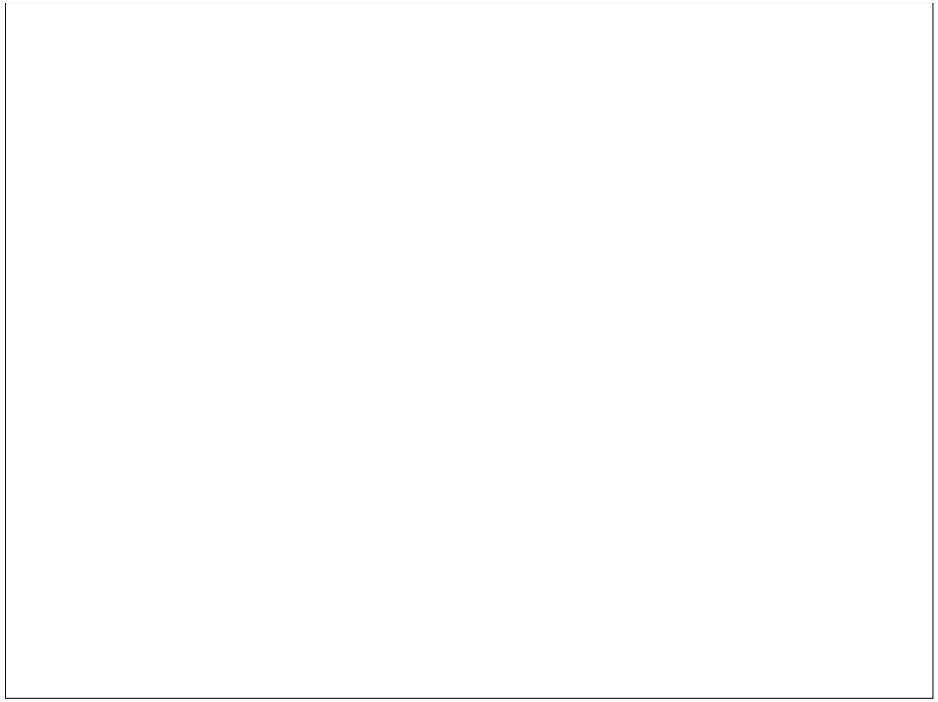


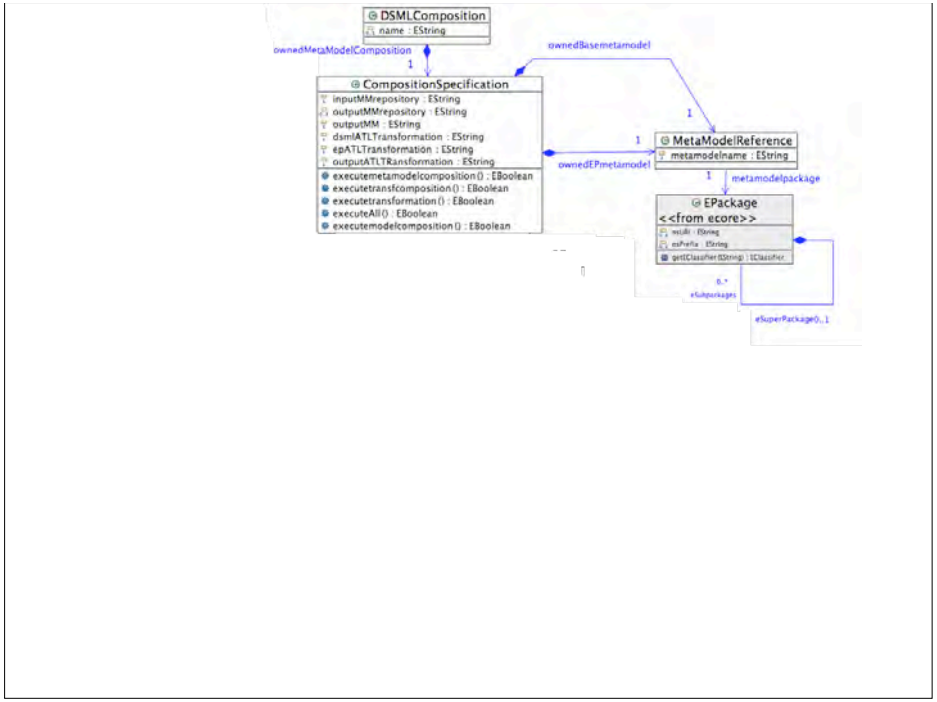
General Case

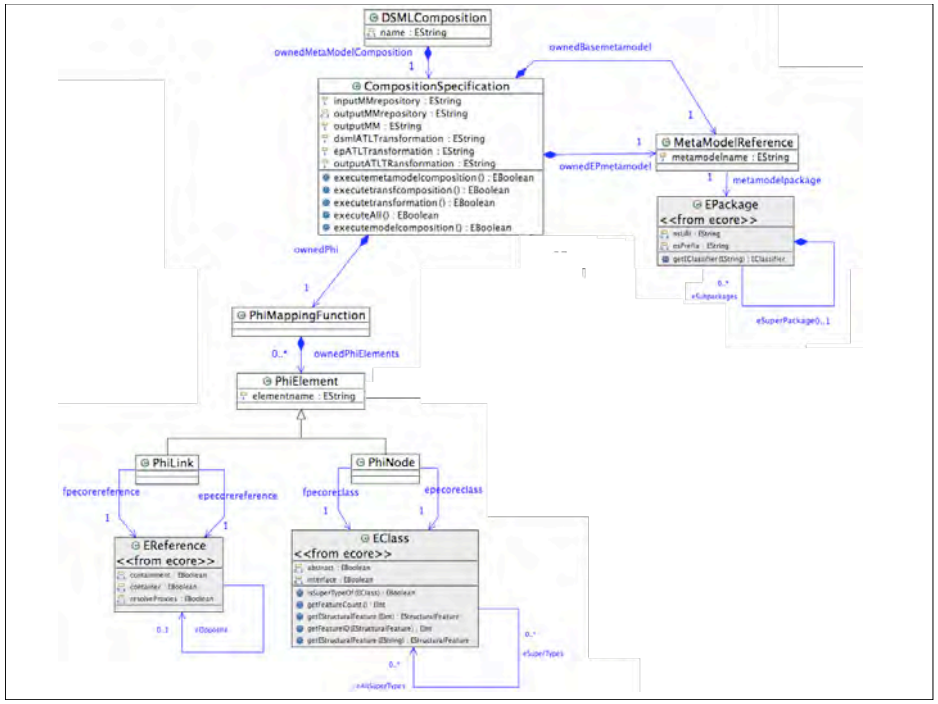


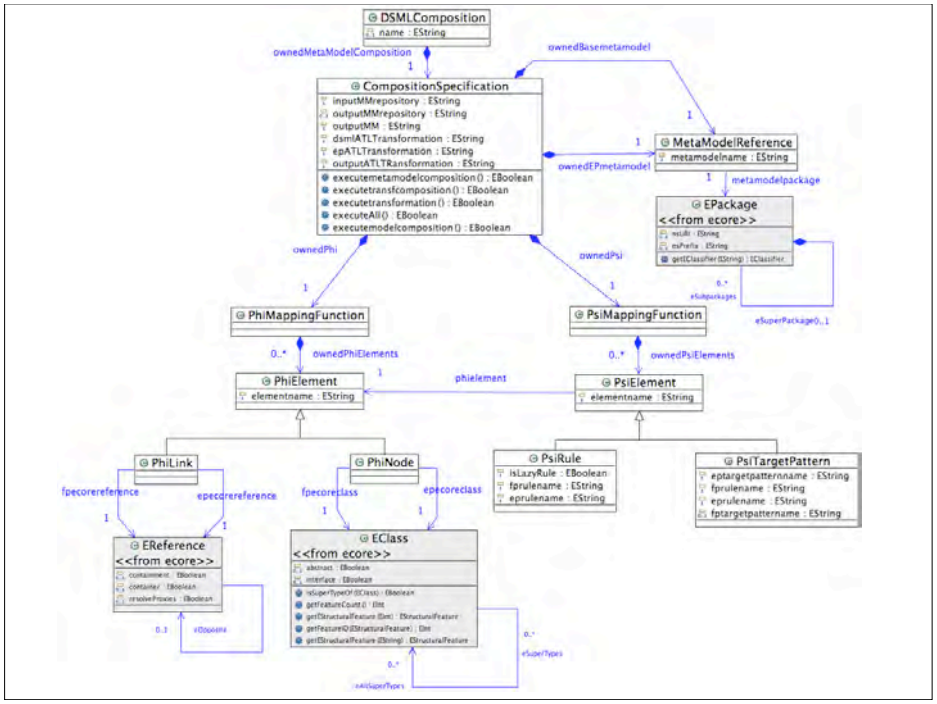

CoPsy
Luis Pedro
January 23, 2009
40

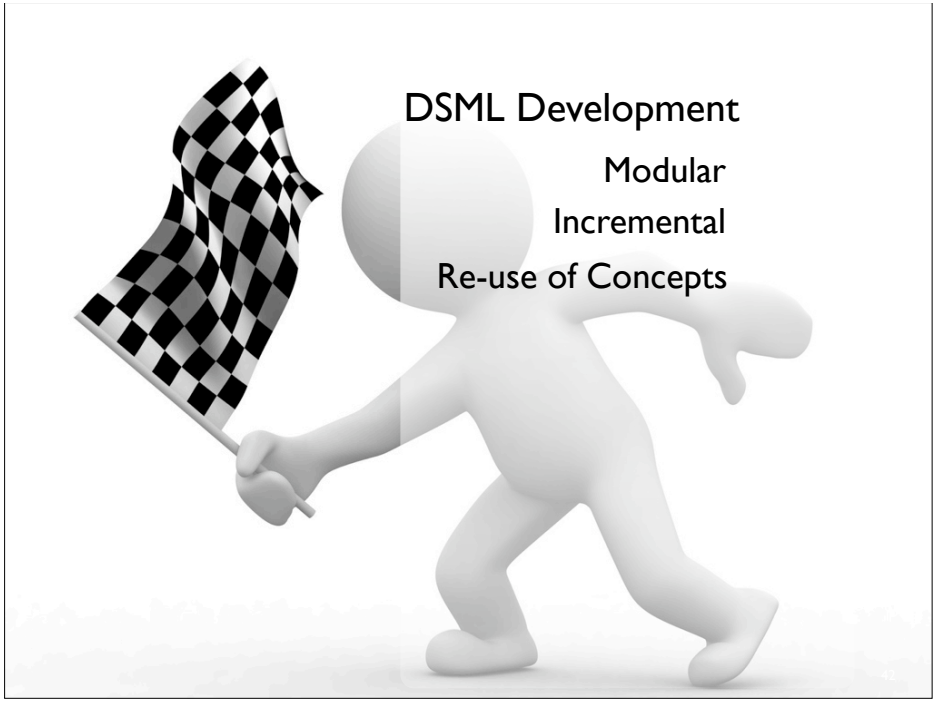

CoPsy
Compositional Platform for
Domain Specific Modelling Languages Prototyping
Luis Pedro
January 23, 2009
40

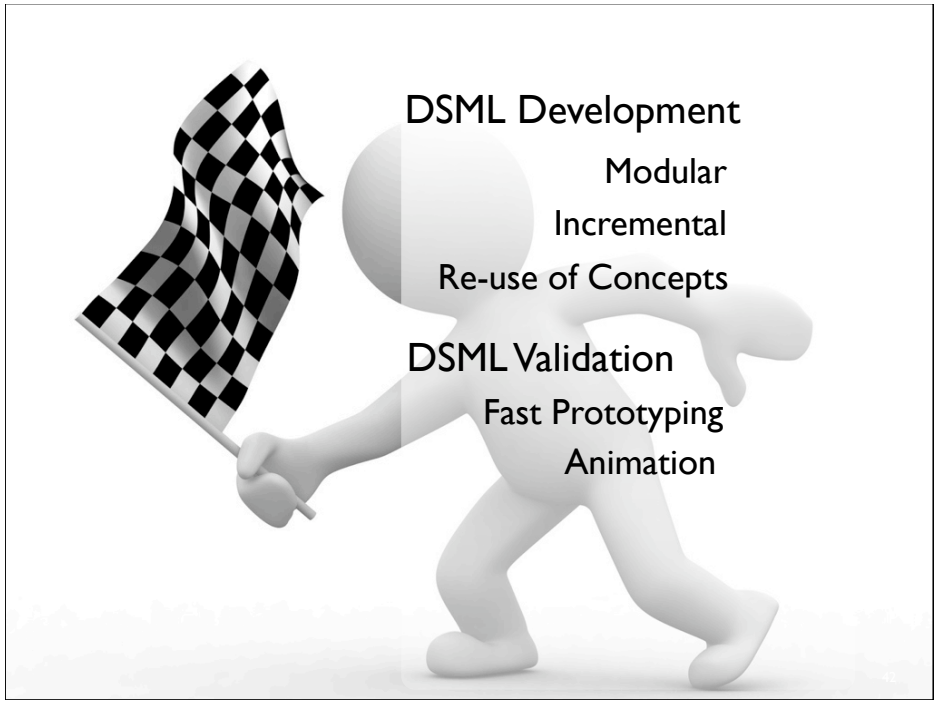


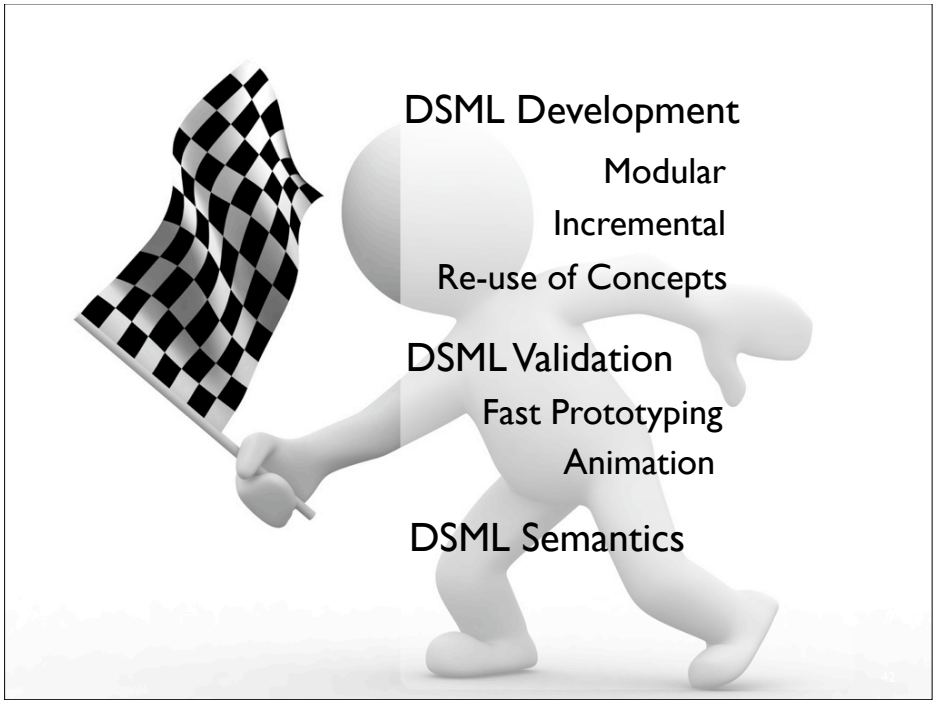

DSMLComposition
name : EString
ownedMetaModelComposition
1
ownedBasemetamodel
CompositionSpecification
inputMMrepository : EString
outputMMrepository : EString
outputMM : EString
dsmlATLTransformation : EString
epATLTransformation : EString
outputATLTransformation : EString
executemetamodelcomposition() : EBoolean
executetransfcomposition() : EBoolean
executetransformation() : EBoolean
executeAll() : EBoolean
executemodelcomposition() : EBoolean
1
1
MetaModelReference
metamodelname : EString
ownedEPmetamodel
1
metamodelpackage
EPackage
<<from ecore>>
nsURI : EString
nsPrefix : EString
getEClassifier(EString) : EClassifier
0..*
eSubpackages
eSuperPackage[0..1]


DSMLComposition
name : EString
ownedMetaModelComposition
ownedBasemetamodel
CompositionSpecification
inputMMrepository : EString
outputMMrepository : EString
outputMM : EString
dsmlATLTransformation : EString
epATLTransformation : EString
outputATLTransformation : EString
executemetamodelcomposition() : EBoolean
executetransfcomposition() : EBoolean
executetransformation() : EBoolean
executeAll() : EBoolean
executemodelcomposition() : EBoolean
MetaModelReference
metamodelname : EString
ownedEPmetamodel
metamodelpackage
EPackage
<<from ecore>>
ownedPhi
PhiMappingFunction
ownedPhiElements
PhiElement
elementname : EString
PhiLink
PhiNode
fpecorereference
epeccrereference
fpecoreclass
epecoreclass
EReference
<<from ecore>>
EClass
<<from ecore>>
eSuperTypes

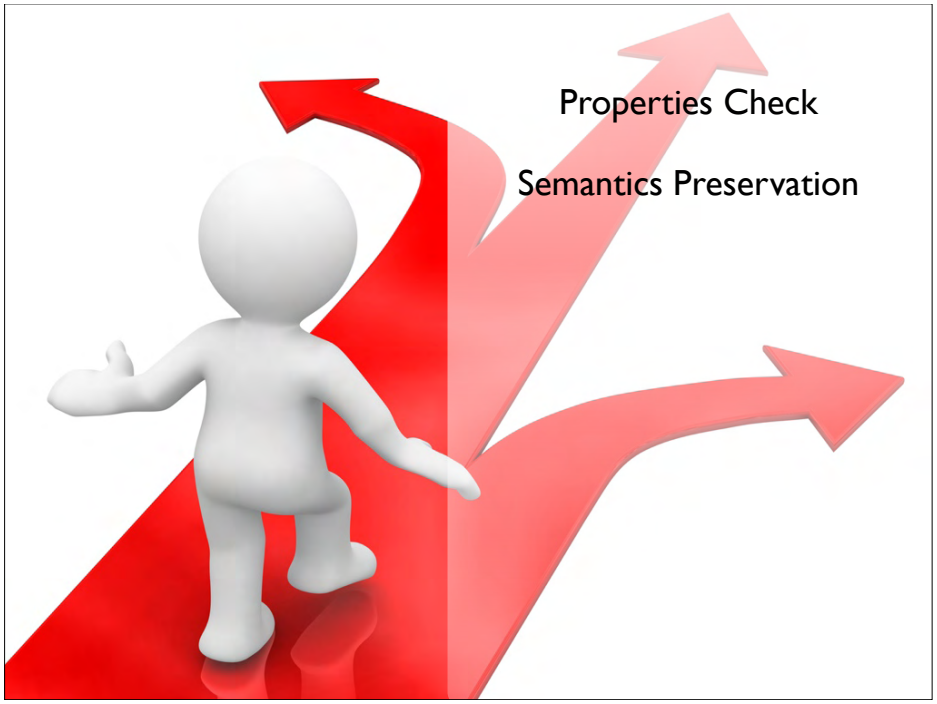

DSMLComposition
name : EString
ownedMetaModelComposition
ownedBasemetamodel
CompositionSpecification
inputMMrepository : EString
outputMMrepository : EString
outputMM : EString
dsmlATLTransformation : EString
epATLTransformation : EString
outputATLTransformation : EString
executemetamodelcomposition() : EBoolean
executetransfcomposition() : EBoolean
executetransformation() : EBoolean
executeAll() : EBoolean
executemodelcomposition() : EBoolean
MetaModelReference
metamodelname : EString
ownedEPmetamodel
metamodelpackage
EPackage
<<from ecore>>
ownedPhi
ownedPsi
PhiMappingFunction
PsiMappingFunction
ownedPhiElements
ownedPsiElements
PhiElement
elementname : EString
phielement
PsiElement
elementname : EString
PhiLink
PhiNode
PsiRule
isLazyRule : EBoolean
fprulename : EString
eprulename : EString
PsiTargetPattern
eptargetpatternname : EString
fprulename : EString
eprulename : EString
fptargetpatternname : EString
fpecorereference
epeccrereference
fpecoreclass
epecoreclass
EReference
<<from ecore>>
EClass
<<from ecore>>
eSuperTypes

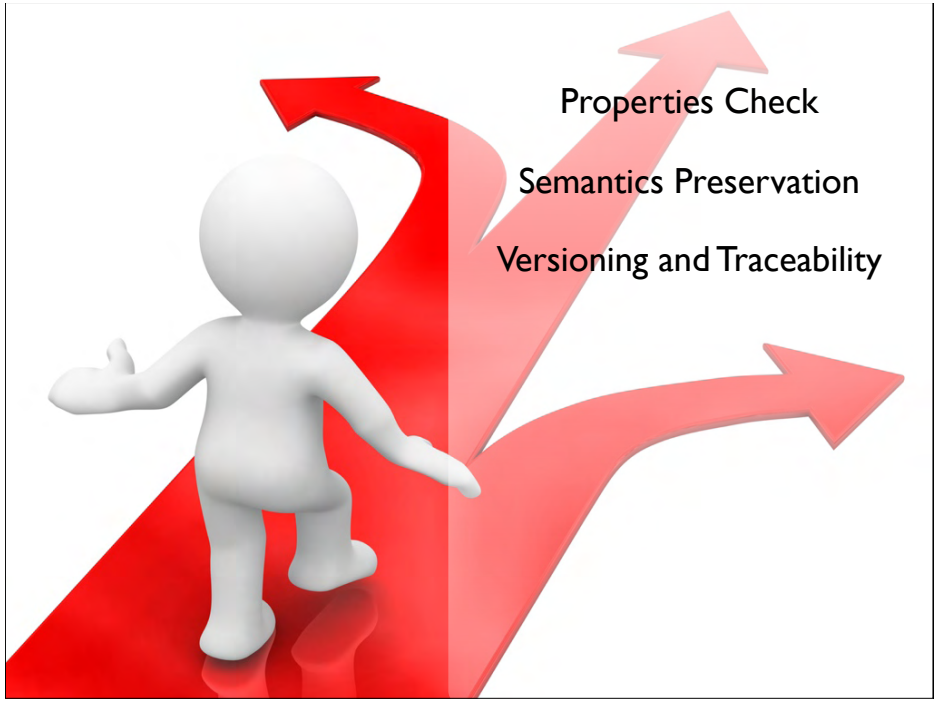

DSML Development
Modular
Incremental
Re-use of Concepts

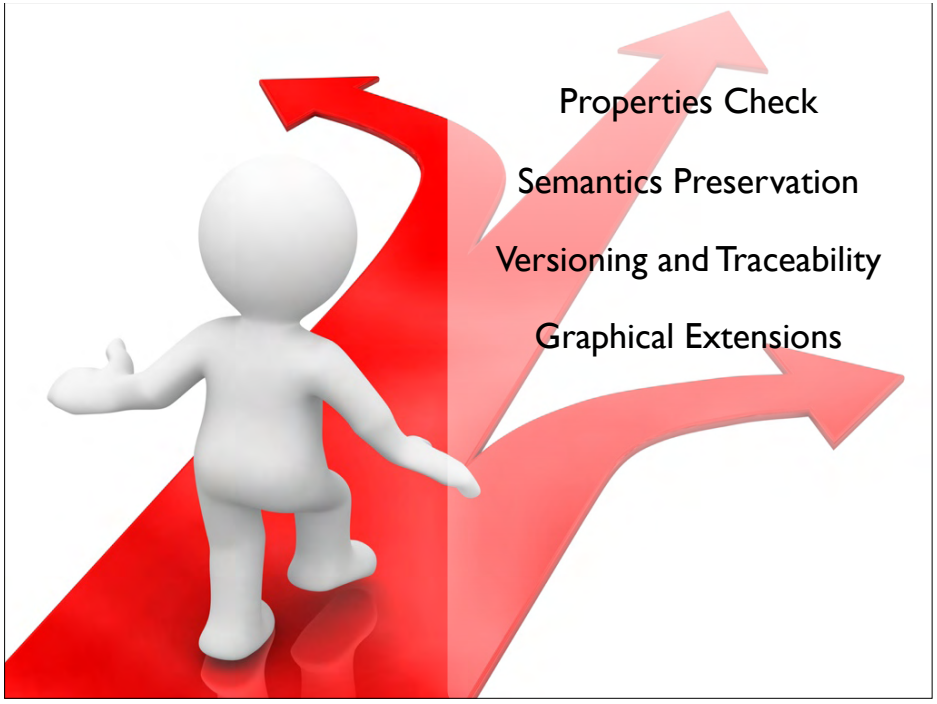

DSML Development
Modular
Incremental
Re-use of Concepts
DSML Validation
Fast Prototyping
Animation

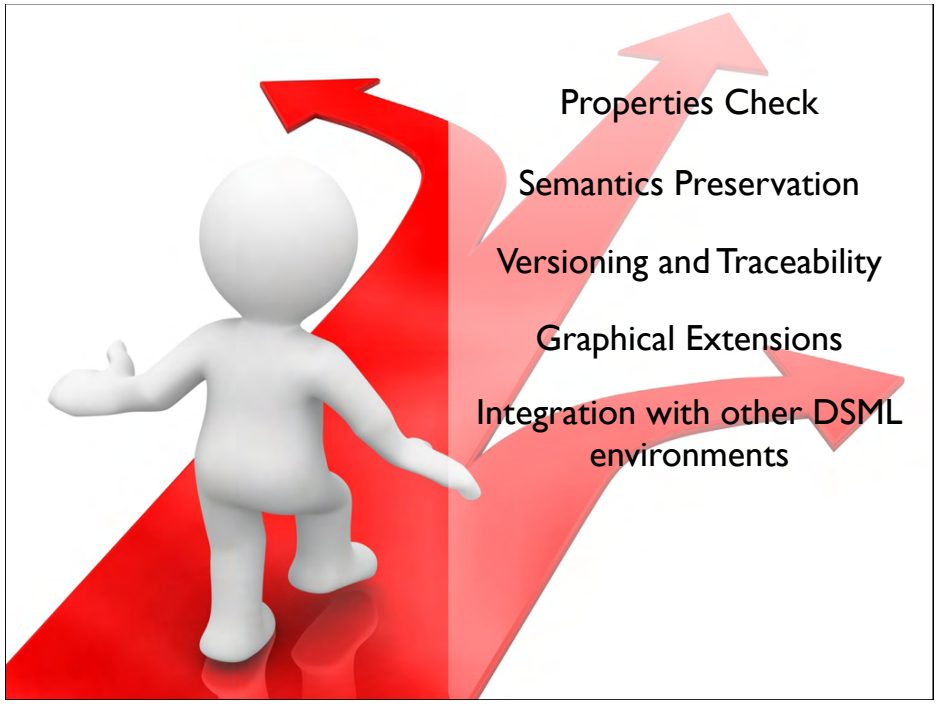

DSML Development
Modular
Incremental
Re-use of Concepts
DSML Validation
Fast Prototyping
Animation
DSML Semantics


Properties Check

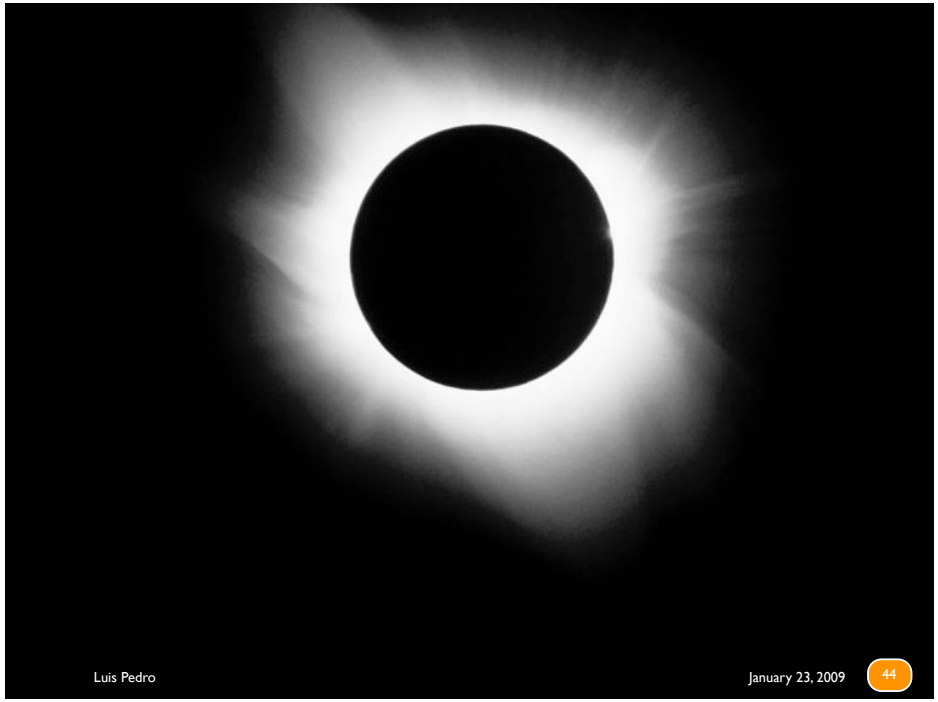

Properties Check
Semantics Preservation


Properties Check
Semantics Preservation
Versioning and Traceability


Properties Check
Semantics Preservation
Versioning and Traceability
Graphical Extensions


Properties Check
Semantics Preservation
Versioning and Traceability
Graphical Extensions
Integration with other DSML environments


$\varphi \quad a \Leftrightarrow b$
$\{\{true, false\} :\rightarrow Bool\}$
$f(x) \quad \Sigma_\omega$
$\leq \bigcup (S_{\Xi_i}, S_{\Xi_i})$
$1 \leq i \leq n \quad \langle v, s_i \rangle$
$\cup$
Luis Pedro
January 23, 2009
44


Luis Pedro
January 23, 2009
44

Articles

(Pedro, Risoldi, Amaral, Barroca, & Buchs, 2009) *Composing Visual Syntax for Domain Specific Languages Prototyping*, Human-Computer Interaction 2009

(Pedro, Amaral, & Buchs, 2008) *Foundations for a Domain Specific Modeling Language Prototyping Environment: A compositional approach,* in Proceedings of the 8th OOPSLA ACM-SIGPLAN Workshop on Domain-Specific Modeling (DSM), October 2008;

(Pedro, Lucio, & Buchs, 2007) *System Prototype and Verification Using Metamodel-Based Transformations*, in IEEE Distributed Systems Online, 2007;

(Pedro, Buchs, & Lucio, 2007) *Model and Metamodel Semantics Enrichment Using Transformations and Domain Composition*, in Rapid Integration of Software Engineering techniques 2007 (to be published);

(Pedro, Lucio, & Buchs, 2006a) *Principles for System Prototype and Verification Using Metamodel Based Transformations*, in Proceedings of IEEE International Workshop on Rapid System Prototyping, 2006;

(A. Chen, Buchs, Lucio, Pedro, & Risoldi, 2006) *Modeling Distributed Systems using Concurrent Object Oriented Petri Nets*, in Proceedings of the Fourth International Workshop on Modelling of Objects, Components and Agents, 2006

(Pedro, Lucio, & Buchs, 2006b) *Prototyping Domain Specific Languages With CO-OPN*, in Proceedings of Springer-Verlag Rapid Integration of Software Engineering techniques, 2005;

Technical Reports

(Pedro, 2008) *Metamodeling with Eclipse*, Centre Universitaire D'Informatique, Université de Genève, 2008;

(Pedro, 2006) *UML2 to CO-OPN transformation: State Machines and Class Diagrams*, Centre Universitaire D'Informatique, Université de Genève, 2006;

Author Index

References

Akehurst, D. H., & Patrascoiu, O. (2004). Ocl 2.0 - implementing the standard for multiple metamodels. *Electr. Notes Theor. Comput. Sci.*, *102*, 21-41.

Barbero, M., Jouault, F., Gray, J., & Bezivin, J. (2007). A practical approach to model extension. In D. H. Akehurst, R. Vogel, & R. F. Paige (Eds.), *Ecmda-fa* (Vol. 4530, p. 32-42). Springer. Available from `http://dblp.uni-trier.de/db/conf/ecmdafa/ecmdafa2007.html#BarberoJGB07`

Bezivin, J. (2005). On the unification power of models. *Software and System Modeling*, *4*(2), 171-188. Available from `http://dx.doi.org/10.1007/s10270-005-0079-0`

Bezivin, J., Jouault, F., & Palies, J. (2005). Towards model transformation design patterns. In *Proceedings of the first european workshop on model transformation (ewmt 2005).* Rennes, France. Available from `http://www.sciences.univ-nantes.fr/lina/atl/www/papers/DesignPatterns05.pdf`

ATLAS Group. (2006, February). *Atl user manual* (Tech. Rep.). Nantes, France: LINA & INRIA.

ATLAS Group. (2007). *Atlas transformation language.* (`http://www.eclipse.org/m2m/atl/`)

Eclipse Project. (2008). *Eclipse modeling framework project.* Available from `http://www.eclipse.org/modeling/emf/`

ikv++ technologies. (2008). *medini qvt.* (`http://projects.ikv.de/qvt`)

Institute for Software Integrated Systems, Vanderbilt University. (n.d.). *Gme 2000 and metagme 2000.* (`http://www.isis.vanderbilt.edu/projects/gme`)

MetaCase Consulting. (2008). *Metaedit+.* (`http://www.metacase.com`)

Object Management Group. (2002a, April). *Meta-Object Facility specification* (Tech. Rep.). OMG. (`http://www.omg.org/technology/documents/formal/mof.htm`)

Object Management Group. (2002b, October). *Omg/rfp/qvt mof 2.0*

query/views/transformations rfp (Tech. Rep.). Needham, Massachusetts, USA: OMG.

Object Management Group. (2003, June). *Mda guide version 1.0.1* (Tech. Rep.). OMG.

Object Management Group. (2004). *Uml profile for enterprise distributed object computing.*

Object Management Group. (2005, August). *Unified Modeling Language version 2.0.* (http://www.omg.org/technology/documents/formal/uml.htm)

Object Management Group. (2006, May). *Object constraint language specification, version 2.0* (Tech. Rep.). OMG.

Object Management Group. (2007a, January). *Meta-Object Facility 2.0 core specification* (Tech. Rep.). OMG. (http://www.omg.org/cgi-bin/doc?formal/2006-01-01)

Object Management Group. (2007b, December). *Mof qvt final adopted specification* (Tech. Rep.). OMG. (http://www.omg.org/cgi-bin/apps/doc?ptc/05-11-01.pdf)

Object Management Group. (2007c, January). *Omg/mof meta object facility (mof) 1.4. final adopted specification document* (Tech. Rep.). OMG. (http://www.omg.org/docs/formal/02-04-03.pdf)

Object Management Group. (2007d, March). *Uml 2.0 superstructure specification - version 2.1.1* (Tech. Rep.). OMG. Available from http://www.omg.org/cgi-bin/doc?formal/07-02-03

SMV Group. (2008). *Batic3s: Building adaptive three-dimensional interfaces for critical complex control systems.* (http://smv.unige.ch/tiki-index.php?page=BATICS)

Triskell team. (2008). *Kermeta - breathe life into your metamodels.* Available from http://www.kermeta.org/

Biberstein, O. (1997). *Co-opn/2: An object-oriented formalism for the specification of concurrent systems.* Unpublished doctoral dissertation, University of Geneva.

Biberstein, O., Buchs, D., & Guelfi, N. (1997). CO-OPN/2: A concurrent object-oriented formalism. In *Proc. second ifip conf. on formal methods for open object-based distributed systems (fmoods), canterbury, uk, july 21-23 1997* (pp. 57–72). Chapman and Hall, Lo.

Bichler, L. (2003, October). Tool support for generating implementations of mof-based modeling languages. In *Proceedings of the 3th oopsla workshop on domain-specific modeling.* Anahiem, California: ACM Press.

Booch, G. (2004). *Object-oriented analysis and design with applications* (3rd ed.). Redwood City, CA, USA: Addison Wesley Longman Publishing

Co., Inc.

Booch, G., Rumbaugh, J., & Jacobson, I. (1999). *The unified modeling language user guide*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc.

Borland. (n.d.). *Borland® together - visual modeling for software architecture design.* Available from `http://www.borland.com/us/products/together/index.html` (`http://www.borland.com/us/products/together/index.html`)

Buchs, D., & Guelfi, N. (1991). A concurrent object oriented petri nets approach for system specification. In *12th international conference on application and theory of petri nets* (p. 432-454). Gjern, Denmar: Springer.

Buchs, D., & Guelfi, N. (2000, july). A formal specification framework for object-oriented distributed systems. *IEEE Transactions on Software Engineering*, *26*(7), 635-652.

Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., & Grose, T. J. (2004). *Eclipse modeling framework* (E. Gamma, L. Nackman, & J. Wiegand, Eds.). Addison Wesley. Available from `http://www.eclipse.org/emf/`

Caporuscio, M., Ruscio, D. D., Inverardi, P., Pelliccione, P., & Pierantonio, A. (2005, June). Engineering mda into compositional reasoning for analyzing middleware-based applications. In *Second european workshop on software architecture (ewsa 2005).* Pisa, Italy.

Chachkov, S. (2004). *Generation of object-oriented programs from co-opn specifications.* Unpublished doctoral dissertation, Ecole Polytechnique Federal de Lausanne.

Chen, A., Buchs, D., Lucio, L., Pedro, L., & Risoldi, M. (2006). Modeling distributed systems using concurrent object oriented petri nets. In *Fourth international workshop on modelling of objects, components and agents.* Turku, Finland.

Chen, K., Sztipanovits, J., Abdelwalhed, S., & Jackson, E. (2005, November July-October). Semantic anchoring with model transformations. In *Model driven architecture* (Vol. 3748 / 2005). Springer Berlin / Heidelberg.

Clark, T., Sammut, P., & Willans, J. (2008). *Applied metamodelling - a foundation for language driven development* (Second ed.; Ceteva, Ed.). CETEVA. (`http://www.ceteva.com/book.html`)

Combemale, B., Rougemaille, S., Crégut, X., Migeon, F., Pantel, M., Maurel, C., et al. (2006). Towards rigorous metamodeling. In L. F. Pires & S. Hammoudi (Eds.), *Model-driven enterprise information systems* (p. 5-14). Paphos, Cyprus: INSTICC Press.

Cook, S. (2000, unknown). The UML Family: Profiles, Prefaces and Packages. In S. K. A Evans & B. Selic (Eds.), *Uml2000 conference proceedings.* LNCS 1939. Available from `http://www.cs.ukc.ac.uk/pubs/2000/1255`

Cook, S., Jones, G., Kent, S., & Wils, A. C. (2007). *Domain-specific development with visual studio dsl tools* (A.-W. Professional, Ed.). Addison-Wesley Professional.

Corporation, C. (2007). *Optimalj.* (`http://www.compuware.com/products/optimalj/`)

Czarnecki, K., & Eisenecker, U. W. (2000). *Generative programming: methods, tools, and applications.* New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.

Deursen, A. van, Klint, P., & Visser, J. (2000). Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, *35*(6), 26-36. Available from `citeseer.csail.mit.edu/article/vandeursen00domainspecific.html`

Drey, Z., Faucher, C., Fleurey, F., Mahé, V., & Vojtisek, D. (2008, January). Kermeta language - reference manual [Computer software manual].

DUPE, G., BELAUNDE, M., PERRUCHON, R., BESNARD, H., GUILLARD, F., & OLIVERES, V. (n.d.). *Smartqvt.* (`http://smartqvt.elibel.tm.fr/`)

Ehrig, K., Guerra, E., Lara, J. de, Lengyel, L., Levendovszky, T., Prange, U., et al. (2005). Model transformation by graph transformation: A comparative study. In J.-M. Bruel (Ed.), *Mtip 2005, international workshop on model transformations in practice (satellite event of models 2005).* Montego Bay, Jamaica: Springer. Available from `http://www.inf.mit.bme.hu/FTSRG/Publications/varro/2005/mtip05.pdf`

Emerson, M., & Sztipanovits, J. (2006, October). Techniques for metamodel composition. In *Oopsla - 6th workshop on domain specific modeling* (p. 123-139). Portland, Oregon: ACM Press. Available from `http://chess.eecs.berkeley.edu/pubs/289.html`

Gerber, A., & Lawley, M. (2004). Generating model-specific editors for mda. In *Proceedings of the 4th oopsla workshop on domain-specific modeling.* Vancouver, British Columbia, Canada: ACM Press.

Gerber, A., & Raymond, K. (2003). Mof to emf: there and back again. In *eclipse '03: Proceedings of the 2003 oopsla workshop on eclipse technology exchange* (pp. 60–64). New York, NY, USA: ACM Press.

Greenfield, J., Short, K., Cook, S., & Kent, S. (2004). *Software factories: Assembling applications with patterns, models, frameworks, and tools.* Wiley. Paperback.

IBM. (2007). *Ibm model transformation framework.* (`http://www`

.alphaworks.ibm.com/tech/mtf)

Jackson, E. K., & Sztipanovits, J. (2006, October). Towards a formal foundation for domain specific modeling languages. In W. Y. Sang Lyul Min (Ed.), *Proceedings of the sixth acm international conference on embedded software (emsoft 06)* (p. 53-63). Seoul, Korea: ACM Press. Available from http://chess.eecs.berkeley.edu/pubs/286.html

Jacobson, I. (2004). *Object-oriented software engineering: A use case driven approach*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc.

Jouault, F., & Bézivin, J. (2006). Km3: A dsl for metamodel specification. In *Fmoods* (p. 171-185).

Jouault, F., & Kurtev, I. (2006). Transforming models with atl. In J.-M. Bruel (Ed.), *Models satellite events* (p. 128-138). Montego Bay, Jamaica: Springer. Available from http://sosym.dcs.kcl.ac.uk/events/mtip/submissions/jouault_kurtev__transforming_models_with_atl.pdf

Kleppe, A., Warmer, J., & Bast, W. (2003). *MDA explained. the model driven architecture: Practice and promise*. Addison-Wesley.

Kosayba, B., Marvie, R., & Geib, J.-M. (2004). Model driven production of domain-specific modeling tools. In *Proceedings of the 4th oopsla workshop on domain-specific modeling.* Vancouver, British Columbia, Canada: ACM Press.

Ladd, D. A., & Ramming, J. C. (1994). Two application languages in software production. In *Proc. of the 19994 usenix symposium on very high level languages(vhll)* (p. 169-177). Santa Fe, NM.

Ledeczi, A., Karsai, G. N. G., Volgyesi, P., & Maroti, M. (2001, September). On metamodel composition. In *Control applications, 2001. (cca '01). proceedings of the 2001 ieee international conference on* (pp. 756–760). Mexico City, Mexico: IEEE Computer Society. Available from http://dx.doi.org/10.1109%2FCCA.2001.973959

Ledeczi, A., Maroti, M., & Volgyesi, P. (2001). *The generic modeling environment* (Tech. Rep.). Institute for Software Integrated Systems, Vanderbilt University.

Luoma, J., Kelly, S., & Tolvanen, J.-P. (2004, October). Defining domain-specific languages: Collected experiences. In *Proceedings of the 4th oopsla workshop on domain-specific modeling.* Vancouver, British Columbia, Canada: ACM Press.

Marvie, R. (2004). *A transformation composition framework for model driven engineering* (Tech. Rep. No. 2004-10). Lille, France: Laboratoire dÍnformatique Fondamentale de Lille.

Microsoft. (n.d.). *Microsoft .net.* (http://www.microsoft.com/net/)

Moss, A., & Muller, H. (2005, September). Efficient code generation for a domain specific language. In *Generative programming and component engineering* (p. 47-62). Tallinn, Estonia: Springer Verlag. Available from http://www.cs.bris.ac.uk/Publications/Papers/2000404.pdf

Nordstrom, G., Sztipanovits, J., Karsai, G., & Ledeczi, A. (1999). Metamodeling - rapid design and evolution of domain-specific modeling environments. In *Ecbs* (p. 68-74). Los Alamitos, CA, USA: IEEE Computer Society.

Pedro, L. (2006, September). *UML2 to CO-OPN transformation: State machines and class diagrams* (Tech. Rep.). Centre Universitaire D'Informatique, Université de Genève. (http://smv.unige.ch/tiki-download_file.php?fileId=683)

Pedro, L. (2008, May). *Metamodeling with eclipse* (Tech. Rep.). Centre Universitaire D'Informatique, Université de Genève.

Pedro, L., Amaral, V., & Buchs, D. (2008, October). Foundations for a domain specific modeling language prototyping environmen: A compositional approach. In *Proc. 8th oopsla acm-sigplan workshop on domain-specific modeling (dsm).* University of Jyvaskylan. (http://www.dsmforum.org/events/DSM08/)

Pedro, L., Buchs, D., & Lucio, L. (2007). Model and metamodel semantics enrichment using transformations and domain composition. In N. Guelfi (Ed.), *Rise 2007.*

Pedro, L., Lucio, L., & Buchs, D. (2006a). Principles for system prototype and verification using metamodel based transformations. In *Ieee international workshop on rapid system prototyping* (p. 10-17). Crete, Greece: IEEE Computer Society.

Pedro, L., Lucio, L., & Buchs, D. (2006b). Prototyping Domain Specific Languages with CO-OPN. In *Rapid integration of software engineering techniques* (Vol. LNCS 3943). Springer-Verlag.

Pedro, L., Lucio, L., & Buchs, D. (2007). System prototype and verification using metamodel-based transformations. *IEEE Distributed Systems Online*, *8*(4). (art. no. 0704-o4001, http://dsonline.computer.org/portal/site/dsonline/menuitem.9ed3d9924aeb0dcd82ccc6716bbe36ec/index.jsp?&pName=dso_level1&path=dsonline/2007/04&file=o4001.xml&xsl=article.xsl&;jsessionid=HGXLCh5sx9rbmBQTnMRQlj5bzfTdL1vWKQwh6nDWmG0yCnJy29gL!1539871550)

Pedro, L., Risoldi, M., Buchs, D., Barroca, B., & Amaral, V. (2009). Composing visual syntax for domain specific languages. In J. A. Jacko (Ed.), *Hci (2)* (Vol. 5611, p. 889-898). Springer.

Risoldi, M., & Amaral, V. (2007). Towards a formal, model-based framework for control systems interaction prototyping. In N. Guelfi & D. Buchs (Eds.), *Rapid integration of software engineering techniques* (Vol. 4401, p. 144-159). Geneva, Switzerland: Springer-Verlag.

Risoldi, M., Amaral, V., Barroca, B., Bazargan, K., Buchs, D., Cretton, F., et al. (2009, to appear). A language and a methodology for prototyping user interfaces for control systems. In D. Lalanne, M. Stolze, & J. Kohlas (Eds.), *Human machine interaction.* Springer-Verlag.

Risoldi, M., & Buchs, D. (2007, September). A domain specific language and methodology for control systems gui specification, verification and prototyping. In *2007 ieee symposium on visual languages and human-centric computing (vl/hcc 2007)* (p. 179-182). USA: IEEE Computer Society.

Rozenberg, G. (Ed.). (1997). *Handbook of graph grammars and computing by graph transformation: volume i. foundations.* River Edge, NJ, USA: World Scientific Publishing Co., Inc.

Selonen, P., Koskimies, K., & Sakkinen, M. (2001). How to make apples from oranges in uml. In *Hicss '01: Proceedings of the 34th annual hawaii international conference on system sciences (hicss-34)-volume 3* (p. 3054). Washington, DC, USA: IEEE Computer Society.

Sendall, S. (2003). Combining generative and graph transformation techniques for model transformation: An effective alliance? In *Oopsla 03 workshop - generative techniques in the context of mda.*

Sendall, S., & Kozaczynski, W. (2003). Model transformation: The heart and soul of model-driven software development. *IEEE Software*, *20*(5), 42-45.

Software, I. O. (2007). *Arcstyler web site, interactive objects software.* (`http://www.arcstyler.com/`)

Software Integrated Systems, I. for. (2005). Gme 5 user's manual [Computer software manual].

Sommerville, I. (1995). *Software engineering* (5th ed.). Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc.

Sun. (2008). *Java platform, enterprise edition.* (`http://java.sun.com/javaee/`)

Thibault, S. (1998). *Domain-specific languages: Conception, implementation and application.* Unpublished doctoral dissertation, University of Rennes.

Vangheluwe, H., & Lara, J. de. (2004). Domain-specifc visual modelling in atom3. In J. P.Tolvanen, J. Sprinkle, & M. Rossi (Eds.), *Proceedings of the 4th oopsla workshop on domain-specific modeling.* Vancouver, Canada: University of Jyväskylä.

Warmer, J., & Kleppe, A. (2003). *The Object Constraint Language, Getting Your Models Ready For MDA* (2nd ed.). Boston, MA, USA: Addison Wesley.

Zito, A., Diskin, Z., & Dingel, J. (2006). Package merge in uml 2: Practice vs. theory? In *Models* (p. 185-199).