Thesis

A methodology for the development of complex domain specific languages

RISOLDI, Matteo

Abstract

The main goal of this thesis is tackling the domain of interactive systems and applying a DSML-based workflow which leads from a system specification to the prototyping of a Graphical User Interface. We chose to use the domain of Control Systems as an example of application for several reasons. Among others, it needs modularity, interactivity, property validation; it requires the development of a user interface; and the domain experts are not typically expert software engineers. The outcome of the thesis is the definition of a methodology that allows easy prototyping of a GUI for interactive systems. The take-away lesson is giving readers a concrete working example of how to build a similar methodology for their domain.

Reference

RISOLDI, Matteo. *A methodology for the development of complex domain specific languages*. Thèse de doctorat : Univ. Genève, 2010, no. Sc. 4230

UNIVERSITÉ DE GENÈVE
Centre Universitaire d'Informatique

FACULTÉ DES SCIENCES
Professeur D. Buchs, directeur
Professeur G. Falquet, codirecteur

A Methodology For The Development Of Complex Domain Specific Languages

THÈSE

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

par

Matteo RISOLDI
de
Terni (Italie)

Thèse No 4230

Genève
Atelier d'impression ReproMail
2010

UNIVERSITÉ DE GENÈVE

FACULTÉ DES SCIENCES

Doctorat ès sciences
Mention informatique

Thèse de *Monsieur Matteo RISOLDI*

intitulée :

" A Methodology for the Development of Complex Domain Specific Languages "

La Faculté des sciences, sur le préavis de Messieurs D. BUCHS, professeur associé et directeur de thèse (Département d'informatique, Centre Universitaire d'informatique), G. FLAQUET, professeur titulaire et codirecteur de thèse (Faculté des sciences économiques et sociales, Département de systèmes d'information, Centre Universitaire d'informatique), V. M. MOREIRA DO AMARAL, professeur (Departamento de Informatica, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Caparica, Portugal), H. VANGHELUWE, professeur (Department of Mathematics and Computer Science, University of Antwerp, Belgium) et Ph. DUGERDIL, professeur (Department of Information Systems, Haute Ecole de Gestion de Genève, Carouge, Suisse), autorise l'impression de la présente thèse, sans exprimer d'opinion sur les propositions qui y sont énoncées.

Genève, le 30 juin 2010

Thèse - 4230 -

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".

Acknowledgements

First of all I would like to thank Didier Buchs for giving me the opportunity to carry out this work in his group and for the support, advice and patience on several occasions when my goals were a bit fuzzy.

Thanks to the other members of the jury, Vasco Amaral, Philippe Dugerdil, Gilles Falquet and Hans Vangheluwe, for taking the time to read my work and to give me precious advice.

Thanks to the SMV group members, current and past, for always being available to provide a different point of view on what I was doing, and for creating a pleasant working environment.

Thanks to the CERN colleagues and friends, for making that time special.

Thanks to Leonello who started the events bringing to this thesis.

Thanks to all the colleagues in the BATIC^{3}S project, who provided expertise in fields different than mine, and to the Hasler foundation and the Department of Computer Science for financially supporting my work.

Thanks to Luís, who taught me a lot (and not only Portuguese curse words) and helped pushing and uplifting me at times when my motivation and morale were not so high.

Thanks to Silvia and Paolo for supporting me when I first emigrated and for giving me their constant friendship along these years.

Thanks to all of my friends, who helped making me the person I am. I can't name you all, but I have a place in my heart for each single one of you.

Thanks to my parents who encouraged me at times when it was hard to understand what I wanted to do. I owe you a lot, this would not have been possible without you.

Thanks to my Italian and Polish families. I am lucky to be loved, and I am thankful for this every single day.

Thanks to Majka and little Leon, who give me love every day. Your love is the achievement I am most proud of. I love you.

Sommaire

Le terme “Langages de modélisation spécifiques à un domaine” (LMSD) est utilisé dans le développement du logiciel pour indiquer un langage de modélisation (et parfois de programmation) dédié à un domaine de problèmes particulier, une technique particulière de représentation d'un problème, et/ou une technique de solution particulière. Le concept n'est pas nouveau; des langages de programmation pour des but spéciaux et un grand nombre de langages de modélisation/spécification existent depuis longtemps. Le terme LMSD a gagné en popularité grâce à la diffusion de la modélisation spécifique à un domaine. Les LMSDs sont considérés comme langages de programmation de 4ème génération (4GL).

Les techniques de modélisation spécifiques à un domaine ont été adoptés pendant quelques années. Toutefois, les techniques et les outils utilisés aujourd'hui souffrent encore des problèmes de complexité et fragmentation. Même si récemment des outils plus intégrées ont vu le jour, ce n'est toujours pas courant de voir des cas concrets où la modélisation spécifique à un domaine a été mise en jeu.

Le but principale de cette thèse est d'attaquer le domaine des systèmes interactifs et d'appliquer un flot de travail basé LMSD pour passer d'une spécification au prototypage d'une interface graphique. Nous avons choisi comme exemple d'application le domaine des systèmes de contrôle pour plusieurs raisons. Entre autres, ce domaine demande modularité, interactivité, validation des propriétés et le développement d'interfaces graphiques. En plus, les experts du domaine ne sont souvent pas des ingénieurs du logiciel.

Les Systèmes de Contrôle (SdC) peuvent être définis comme des mécanismes produisants des variables de sortie d'un système en manipulant ses entrées (provenants de capteurs ou commandes). Certains SdC peuvent être très simples (p.ex. un thermostat) et ne posent pas de défis particuliers à la modélisation avec des formalismes **general-purpose**, tandis que des autres

SdC peuvent être complexes par rapport au nombre de composants, taille, organisation physique et fonctionnelle et supervision.

Un SdC *complexe* aura en général une structure composée, dans laquelle chaque objet peut être groupé avec des autres; les objets composés peuvent être, à leur tour, composants (ou "enfants") d'objets plus grands, dans un arbre hiérarchique où la racine représenterait le système et les feuilles sont les dispositifs plus élémentaires. Contrôler et superviser un système si complexe demande le développement d'interfaces utilisateur graphiques complexes, qui doivent tirer profit d'une méthodologie spécifique à un domaine.

Le résultat concret de cette thèse est la définition d'une méthodologie qui permet le prototypage rapide d'une interface graphique pour des systèmes interactifs. La leçon à en tirer pour le lecteur est un exemple concret de moyen pour construire une méthodologie similaire pour leur domaine.

Abstract

The term Domain Specific Modeling Language (DSML) is used in software development to indicate a modeling (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 modeling. Domain-specific languages are considered $4GL$ programming languages.

Domain-specific modeling techniques have been adopted for a number of years now. However, the techniques and frameworks used still suffer from problems of complexity of use and fragmentation. Although in recent times some integrated environments are seeing the light, it is not common to see many concrete use cases in which domain-specific modeling has been put to use.

The main goal of this thesis is tackling the domain of interactive systems and applying a DSML-based workflow which leads from a system specification to the prototyping of a Graphical User Interface (GUI). We chose to use the domain of Control Systems (CSs) as an example of application for several reasons. Among others, it needs modularity, interactivity, property validation; it requires the development of a user interface; and the domain experts are not typically expert software engineers.

CSs can be defined as mechanisms that provide output variables of a system by manipulating its inputs (from sensors or commands). While some CSs can be very simple (e.g., a thermostate) and pose little or no problem to modeling using general-purpose formalisms, other CSs can be complex with respect to the number of components, dimensions, physical and functional organization and supervision issues.

A complex CS will generally have a composite structure, in which each

object can be grouped with others; composite objects can be, in their turn, components (or "children") of larger objects, forming a hierarchical tree in which the root represents the whole system and the leaves are its most elementary devices. Controlling and supervising such complex systems requires the development of complex GUIs, which can benefit from adopting a domain-specific methodology.

The outcome of the thesis is the definition of a methodology that allows easy prototyping of a GUI for interactive systems. The take-away lesson is giving readers a concrete working example of how to build a similar methodology for their domain.

Contents

List of Figures

List of Tables

List of Listings

Part I

Preliminary matters

Chapter 1

Introduction

1.1 Domain specific modeling languages

The term DSML is used in software development to indicate a modeling (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 modeling. 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).

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

- a set of components or language terms and constructs with which em-

bedded hardware / software can be modeled;

- 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 inter-relationships 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 DSML environments are not linked to a single technology and a good DSML development environment should be able to deal with diversity.

1.2 Motivation of this work

Domain-specific modeling techniques have been adopted for a number of years now. However, the techniques and frameworks used still suffer from problems of complexity of use and fragmentation. Although in recent times some integrated environments are reaching maturity, it is not common to see many concrete use cases in which domain-specific modeling has been put to use.

One way that domain-specific modeling can improve software development is by reducing the knowledge gap between a system expert and the specification. This makes specifications more usable and correct, and opens perspectives for a non-traditional way of prototyping software.

The domain we worked on for this thesis is GUI development for interactive systems. This is a domain where the competences of software engineers and system experts have to integrate to provide a satisfactory software product. *Satisfactory* in this context means correct with respect to requirements and quality. The knowledge gap here is concretized in the different point of view that a software engineer and a system expert can have on the domain. This can introduce bias, generate communication difficulties, and ultimately hurt the developing process.

1.3 Thesis

The thesis we defend is that domain-specific modeling techniques can improve the field of developing GUI for interactive systems. We propose a DSML-based workflow that leads from the specification phase to the implementation of a solution. The improvements we foresee are better reuse of domain-specific system knowledge, reduced development effort, and enabling of features like verification and simulation. We limit our thesis to the field of interactive systems, although it can be foreseeable to extend this work to similar domains.

We chose to test the proposed workflow in the domain of control systems. This is for several reasons. Among others, control systems need modularity, interactivity, property validation; they require the development of a user interface; and the domain experts are not typically expert software engineers.

1.4 Contribution of this thesis

The deliverable of the thesis is the definition of a methodology for DSML engineering, and a workflow that allows for generation of interactive user interfaces, system simulation, validation and verification. In this methodology, a language engineer makes the effort of creating the DSML for a specific domain and integrating it to the workflow, while the task of specifying the system and prototyping the software goes to a system expert. The latter can use the DSML to interact with the workflow using concepts and metaphors which are near to her view on the system, without requiring (to a certain degree) a deep computing science knowledge.

The specification phase focuses on specifying the system to interact with instead of the GUI. On one hand, this makes the methodology less general and only applicable to the domain of interactive systems. But on the other hand, the methodology becomes accessible by people who don't have a specific GUI development know-how, and allows rapid prototyping by reusing existing information. In this, our approach is different from several others which try to be general by focusing on GUI specification formalisms (see the Related Work section).

We exemplify our approach by illustrating a project, called BATIC^{3}S, which defines a technique for prototyping GUIs for control systems. We show the process of creating a DSML for specifying control systems, and the

process that can produce a GUI and simulator from it. We apply the methodology to two high-energy physics experiment at CERN. We will discuss one in this thesis, and reference the other in the final sections.

After discussing the BATIC^{3}S project we generalize the methodology by giving some advice and lessons learned about language engineering, transformations and property verification.

Although related to the field of research on human-machine interaction, this study does not focus primarily on effectiveness and ergonomy of user interfaces, nor on the search for a better paradigm for visualization of control systems. The case study we present here happens to use an interactive 3D visualization style; this was motivated by the main research interests of some of the collaboration members. However we will not delve here into an evaluation of effective this paradigm is for the specific domain.

1.5 Requirements for this work

To say that domain-specific software engineering can benefit from the proposed methodology, we defined a list of requirements that have to be satisfied. These belong to three groups: user-related requirements, technical requirements, and methodological requirements.

User-related requirements are those that influence the way the users interface with the methodology, and their expectations. These are:

- **Domain specificity:** the methodology should provide a way for engineering DSMLs;
- **Abstraction:** using the DSML, the user must be able to work at her level of knowledge, hiding the low-level details;
- **Evolution:** it must be possible, and relatively easy, for the workflow to evolve with changed or additional requirements from the user in terms of what can be specified;
- **Simulation:** the user must be able to simulate the behaviour of the system in order to evaluate the prototype GUI without having to actively code a simulator.

Technical requirements are those that influence how the methodology can evolve with time as new techniques appear and platforms evolve. These are:

- **Integration:** proposed technical solutions must be open and fully accessible via API calls with a variety of languages. A plug-in architecture is desirable so that functionalities can be added or removed on-demand;
- **Platform independency:** proposed technical solutions must not be tied to a platform, and be as little tied as possible to a specific version of an operating system/environment.

Methodological requirements are those that define what is expected by the methodology in terms of offered functionality. These are:

- **Transformation:** it must be possible to transform all languages and formalisms used in the methodology to some different but equivalent language. This is required both for evolution and for integrating possible future, more efficient way of representing specifications;
- **Validation and verification:** the methodology must include a way to enable V&V activities, both by the use of a formal representation of specifications, and by using frameworks which support simpler forms of validation (structural properties, constraint checking);
- **Automation:** the methodology should aim for automating the workflow that leads from the specification to the GUI and simulator as much as possible. This is also related with the technical requirement of Integration.

1.6 Structure of the document

Chapter 2 reviews related work in the field of DSMLs, Model-Driven Engineering and Model-Driven GUI development. Each of the reviewed techniques is evaluated with respect to a set of requirements we established to reach our goals.

Chapter 3 introduces the case study we adopted to exemplify our methodology. This is the description of a control apparatus, with its components, behaviours and properties. This chapter is useful as a reference for the examples that follow.

Chapter 4 illustrates the proposed methodology from an abstract point of view. It focuses more on principles and workflow than on the technologies used for implementation. It also contains some general considerations about validation and model checking.

Chapter 5 describes in more detail the DSML called Cospel that is part of the proposed methodology. Examples of use of the language are given using the case study in Chapter 3.

Chapter 6 describes the transformation process that achieves the prototyping and simulation goals of the methodology. It begins with a description of the target formalism and of the transformation framework. It then proceeds to an overview of the Cospel transformation. Finally, examples are given for the transformation of each part of Cospel.

Chapter 7 briefly describes the architecture and implementation of the GUI engine that renders the GUI prototype. It is rather focused on technical aspects of the engine.

Chapter 8 explains the approach of this methodology to model checking. Different types of model checking and validation are presented, using different techniques and checking different types of properties. A formalization of the CO-OPN to Algebraic Petri Net (APN) transformation is given. Finally, we present an example of reachability model checking using the AlPiNA tool.

Chapter 9 takes a more abstract point of view on the content of this thesis and extracts more general advice and lessons learned from this work. We discuss some key points and critical difficulties in achieving similar goals in different domains, and give suggestions and perspectives on how to approach certain delicate problems.

Chapter 10 overviews the context of the BATIC^{3}S project in which this work was developed. It relates parallel developments and other institutions that took part in the project.

Chapter 11 summarizes the thesis, draws conclusions and gives perspectives of this work. It also lists the concrete outcomes of this thesis in terms of publications and student projects.

The document ends with appendices containing acronyms (A) and some code examples (B), and the references.

Chapter 2

Related work

The work in this thesis encompasses several domains. We are going to present techniques and approaches in each of them, and evaluate them against a set of relevant criteria. We will start by presenting different approaches for creating DSMLs. Then we will speak about Model Driven Engineering, and list some tools and techniques to implement this methodology. Finally we will present some frameworks and tools for model-driven DSML development, which compare more directly to the work in this thesis.

2.1 DSML development approaches

The zoology of DSML approaches is vast and varied w.r.t. many aspects. Approaches reflect a different point of view on the essence of DSMLs, and have different levels of functionality and completeness. Tools range from compiler generators to complete frameworks with editor generation and visualization. The tools we will list go from the more low-level, compiler-oriented strategies to the more complete DSML ecosystems. We will evaluate them against the following set of criteria:

- Abstraction: the language must contain concepts at the abstraction level of the domain;
- Platform Independence: the approach must be easy to port to different platforms;
- Executability: it must be possible to define languages with executable semantics;

- Verification: it must be possible to perform verification (e.g., property checking) on models;

- Transformation and Integration with tools: it must be possible to transform models to different formalisms, and integrate with other modeling/transformation/verification frameworks.

2.1.1 Compiler compilers

Compiler compilers are tools able to generate a parser for a textual language from a given grammar. This is the lowest-level approach to DSML development. Generated parsers can be used for building a compiler or an interpreter. A rather exhaustive list is given in (Wikipedia, 2009), however it is worth mentioning some of the better-known, such as ANTLR (Parr & Quong, 1995), CoCo/R (Mössenböck, Löberbauer, & Wöß, 2010) and JavaCC (Viswanadha, 2010). The latter supports the Java language, whereas the others (especially CoCo/R) support a number of different languages, ranging from C to Ada, from Pascal to Ruby passing by Visual Basic .NET.

Compiler compilers mainly help with the coding of a compiler, by generating it instead of having it coded by hand. They are typically fairly complex to use and require a strong knowledge of language theory.

2.1.2 Language extension approaches

Another approach to DSML development is the extension of an already-existing language with domain-specific constructs. This is typically achieved by generating new compilers for a given language by extending the language's grammar, or by creating mappings to existing constructs. Examples are JTS (Batory, Lofaso, & Smaragdakis, 1998) and JLE (Van Wyk, Krishnan, Bodin, & Johnson, 2006) for Java, and EasyExtend (Schluehr, 2008) for Python. The metafront (Brabrand, Schwartzbach, & Vanggaard, 2003) framework allows to extend any user-provided grammar with another user-provided grammar, allowing not only the extension of existing languages but the creation of completely new ones.

2.1.3 DSML ecosystems

Other approaches try to go further and generate more than just a compiler or an interpreter. LISA (Henriques et al., 2002) is based on attribute grammars, and generates a syntax-aware editor and finite states automata visualization. Metaenvironment (Brand et al., 2001) uses ASF+DSF (Algebraic Specification Formalism and Syntax Definition Formalism) to define semantics and syntax of a DSML. It can generate executable specifications based on conditional equations and term rewriting, as well as interpreters, editors and pretty printers. SmartTools (Parigot et al., 2002), which defines a language using abstract syntax trees, also generates a Unified Modeling Language (UML) model.

2.1.4 Evaluation with respect to our criteria

Table 2.1 evaluates the DSML development approaches with respect to the criteria given in Section 2.1. It seems quite clear that the DSML ecosystems approaches satisfy more of the requirements, and in a more complete way. This becomes even more evident in the case of a particular family of ecosystems, those based on the MDE approach, that we will see in the next section.

	Abstraction	Platform Independ.	Execut.	Verification	Transform. & Integr.
Compiler Compilers	Low	Some	**Good**	Some	Low
Language Extensions	Some	Some	Low	Some	Low
DSML Ecosystems	**Good**	**Good**	Some	Some	**Good**

Table 2.1. Criteria satisfaction for DSML development approaches

2.2 Model-driven engineering

Model-driven engineering (MDE) is a point of view on systems engineering that sees the model at the center of the development process. It is based on

the principle that *"Everything is a model"* (Bézivin, 2005): models are no more only means of documentation or idea exchange, but can be any software artifact. This introduces the necessity of formal model specifications, where "formal" can be intended to various degrees of rigor, to capture information which can be informal in nature.

MDE is based on the key concepts of model, metamodel, domain-specific modeling languages and model transformation. While we already spoke about DSMLs, let us give a brief definition of the other concepts. A more formal definition of model, metamodel and model transformation has been given in (Pedro, 2009).

A **model**, according to a good definition in (Kleppe, Warmer, & Bast, 2003), is

> *[...] a description of a (part of) a system written in a well-defined language. A well-defined language is a language with well-defined form (syntax) and meaning (semantics), which is suitable for automated interpretation by a computer.*

This definition, which seems tailored for MDE, puts in evidence the need for a way to ensure that a modeling language is "well-defined". This is what a metamodel does.

A **metamodel** (literally, *after model* from the Greek μετά) is a model which describes the structure of another model. A metamodel is to a model what a language grammar is to a sentence in that language. The metamodel describes the constructs and rules that can be used to create a class of models. In other words,

> *A metamodel is a model that defines the language for expressing a model.* (Object Management Group, 2003)

The metamodel is itself a model, and is described in a specific language. Of course, this language too has to be described – which is done by another layer of abstraction, the **metametamodel**. This naturally raises the recursive question: what describes the structure of a metametamodel, and where does this pyramid stop? The answer is that metametamodels are self describing: the metamodel of a metametamodel is the metametamodel itself.

Methodologies that implement Model-Driven Engineering (MDE) generally reflect this layered structure, defining four levels of abstractions (data, model, metamodel, metametamodel). They generally differ in the metametamodel they use. We will now speak about some MDE implementations.

2.2.1 Different implementations of MDE, and tools

One of the most well-known implementations of MDE is Model Driven Architecture (MDA). MDA proposes to separate design and architecture by using different abstraction levels. The most abstract level is called Platform Indipendent Model (PIM), and is a conceptual model leaving out implementation details. The PIM is typically expressed using a DSML, in order to focus on a domain's concepts.

On the other hand, a Platform Specific model (PSM) is a model that includes specific aspects of the platform and technology used for implementation. A PSM might be made of code in some programming language, or by a low level-of-abstraction DSML. In the latter case, they may often be used for code generation. PSMs are typically obtained by transformation of PIMs, and one PIM might be used to obtain several PSMs using different transformations.

Several metametamodels can be used for Model Driven Architecture (MDA). Among these are OMG's Meta-Object Facility (MOF) (Object Management Group, 2003), the Kernel Metametamodel (KM3) (Jouault & Bézivin, 2006), EMF ECore (Steinberg, Budinsky, Paternostro, & Merks, 2009) and Kermeta (Muller, Fleurey, & Jézéquel, 2005). They all serve the same purpose although some differ in expressiveness.

The multiplicity of metametamodels is one of the causes of the fundamental fragmentation of this domain. To add to this situation, other MDE implementations exist, with respective metametamodels and tools: Software Factories is Microsoft's take on the subject, with its own DSL tools. Vanderbilt University has its implementation named Model Integrated Computing, based on the MetaGME metametamodel and the Generic Modeling Environment (GME) tool. The list could go on for long, but the important message here is that the universe of metamodeling is divided among several concurrent solutions. While the metamodeling stack is a shared notion among different implementation, each of these comes with its own set of implementations for each layer, which it might or might not share with other approaches. This "partitioned" view is based on the concept of *technological spaces*. According to (Kurtev, Bézivin, Jouault, & Valduriez, 2006),

> *A technological space is a model management framework with a set of tools that operate on the models definable within the framework.*

Figure 2.1, which is taken from (Karlsch, 2007), shows an example of some different technological spaces.

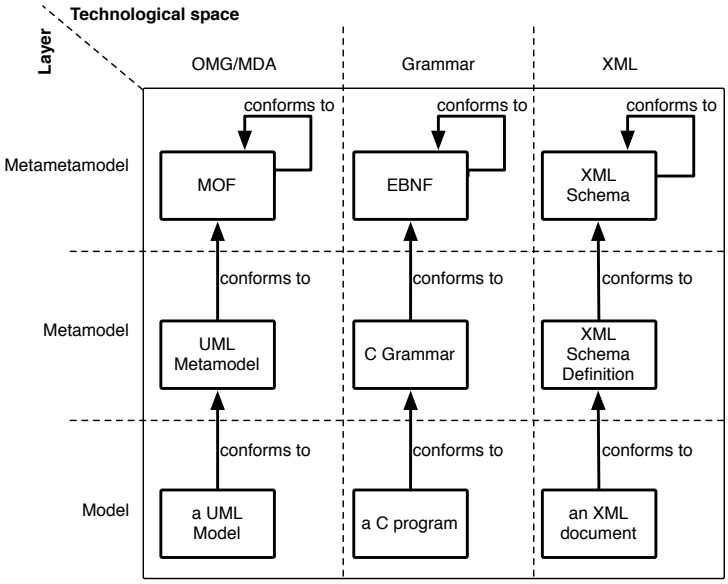

Figure 2.1. How different technological spaces implement the metamodeling stack

2.2.2 Tools for DSML creation: Eclipse/EMF and the others

In the context of this thesis, it is worth examining in more detail how the Eclipse/EMF framework implements MDA.

EMF is a framework for modeling and code generation which is part of the Eclipse platform since 2002. It includes three main components: **core** (metametamodel, persistence, serialization, validation and model tracing); **edit** (model viewing and editing); and **codegen** (API generation for ECore-based models).

EMF was originally supposed to be based on the MOF metametamodel. However, at the time MOF was very complex and did not fit the needs of the EMF developers. The ECore metametamodel was thus adopted. In the meantime, MOF continued its evolution until version 2.0, which has an Eclipse implementation called EMOF which is mostly similar to ECore. EMF today can use models based on either EMOF or ECore.

On top of EMF several other tools have been built that extend it with various functionalities, like Graphical Modeling Framework (GMF) (Graphical Modeling Framework) which allows to create visual syntax for model editing, or XText to define textual editors with syntax checking, autocompletion and other advanced features. EMF is very general and flexible; a drawback of this feature is that it is very complex to use, and often suffers compatibility problems with third-party extensions from one release to another.

In addition to EMF, several other frameworks for DSML creation exist. Without going to great detail, here is a brief list of the most well-known. For each, we mention between parenthesis and in italics the metametamodel they are based on:

- Generic Eclipse Modeling System (White, Schmidt, Nechypurenko, & Wuchner, 2007) *(ECore)* - a refinement of GMF, where many steps are automatized
- Rational Software Architect (IBM, 2010a) *(MOF/UML)* - DSML creation by use of UML 2.0 profiles. Limited graphical representation possible.
- Tau G2 (IBM, 2010b) *(MOF/UML)* - DSML creation by extension of the UML metamodel. Graphical representation possible for model elements (not for links).
- MetaEdit+ (MetaCase Consulting, 2010) *(GOPRR)* - DSML editor with diagrams. Support for constraints.
- Generic Modeling Environment (Vanderbilt University, 2005) *(MetaGME)* - Support for Object Constraint Language (OCL) and XMI. Graphical representation is very flexible.
- Microsoft DSL tools (Microsoft, 2008) *(no name)* - .NET integration. Graphical representation is very flexible.
- Kermeta (Triskell team, 2010) *(EMOF+behaviour)* - adds semantics to DSML definition. Textual and graphical representation. Exports to ECore.
- openArchitectureWare (openArchitectureWare Working Group, 2010) *(ECore)* - centered on model-to-text and text-to-model transformations. Some support for constraint checking.

- AToM^3 (A Tool for Multi-formalism and Meta-Modeling) (de Lara & Vangheluwe, 2002; de Lara, Vangheluwe, & Alfonseca, 2004; Vangheluwe & de Lara, 2004) *(UML Class Diagrams, previously ER)* - graphical syntax editor. Model properties store the concrete syntax. Model views are generated with triple graph grammars.

2.2.3 Model transformation

With such a varied panorama of formalisms, metametamodels and tools, one question that arises is how to transform a model of one kind to something different. Many model-to-model transformation techniques exist. Most are inter-technological space. They differ in many aspects. Some are imperative, other declarative. Some manage several input models. Some are graph-based, others algebraic. Some manipulate models directly. In (Mens & Gorp, 2006) a taxonomy is attempted on characteristics, success criteria and quality requirements. It concludes that *"[...] there is never a unique answer to the question which approach (or tool, or technique) to model transformation is the best"*, but identifies criteria that can be useful in choosing the right transformation technique.

In (de Lara & Guerra, 2005), model transformation approaches are classified according to two orthogonal axes: *source and target formalisms* and *abstraction level*. On the first axis, two groups are defined:

Inter-formalism: a model is transformed into a different formalism. Typical applications are database version migration or transformation for model analysis (which this thesis will propose at a given point). A common problem is preserving the semantics and properties of models through transformation.

Intra-formalism: a model is transformed into another model in the same formalism. Typically used for refactoring, optimization and similar applications.

On the abstraction level axis, two other groups are defined:

Horizontal: a model is transformed into another model at the same level of abstraction. For example, in the MDA context, a PIM-to-PIM transformation is horizontal. This can be used for refactoring (intra-formalism) or migration (inter-formalism).

Vertical: a model is trasformed into another model at a different level of abstraction, typically lower. Examples are PIM-to-PSM transformations and code generation. An example of low-to-high abstraction level transformation is reverse engineering.

As we will see in the next chapters, the work proposed in this document involves several steps of inter-formalism, vertical transformations.

Also in (de Lara & Guerra, 2005) three orthogonal characteristics are also given for classifying transformation languages: visual vs. textual; imperative vs. declarative; and formal vs. semi-formal. Transformation languages might sometimes be hybrid with respect to these characteristics.

A standard for model transformation proposed by Object Management Group (OMG) is Query-View-Transformation (QVT). This is a graph-based approach for which several implementations have been proposed through the years. However, no complete implementation of the Query / Views / Transformations (QVT) standard exists as of today. At the moment, the most complete implementation is the Atlas Transformation Language (ATL).

ATL has a large user community and an open source library of transformation. It is an hybrid of declarative and imperative rule-based language. With respect to the classification in (de Lara & Guerra, 2005), it is a textual, imperative/declarative hybrid, semi-formal transformation language. It can be used for both inter- or intra-formalism transformations, as well as for horizontal and vertical transformations.

ATL can use ECore or MOF metametamodels. It has been included in the Eclipse Modeling Project and is very well integrated with EMF.

Other transformation tools in the MDA technological space include Borland Together, SmartQVT, YATL. There are of course tools for different technological spaces, such as XSLT for the XML space, ANTLR for the BNF Grammar space, and GReaT, Viatra and FUJABA for the Graph space. A complete classification of transformation approaches would be long and out of scope for this document. An in-depth work on this subject has been done in (Czarnecki & Helsen, 2003). That article applied domain analysis to the field of model transformation approaches, and categorizes approaches with a rather detailed granularity.

2.2.4 Evaluation with respect to our criteria

Table 2.2 continues Table 2.1 and evaluates the MDE techniques with respect to the criteria in Section 2.1. We can see how this family of techniques fulfills best the criteria we need. In particular, the EMF-based techniques satisfy them all.

	Abstraction	Platform Independ.	Execut.	Verification	Transform. & Integr.
MDE techniques	**Good**	**Good**	Some[a]	**Good**[a]	**Good**

[a]By means of transformation

Table 2.2. Criteria satisfaction for DSML development approaches

2.3 Model-driven GUI development

Based on the philosophy and techniques we mentioned above, several approaches have gone to different lengths in using model-based techniques for GUI development. Some of these approaches are rather formalisms and/or methodologies, while others are full-fledged tools. Here too the panorama is too vast to include every possible related work; our criteria of choice for inclusion in this section is to cover the various different angles of attack to the problem. We will evaluate them with respect to the following criteria:

- Automation
- Rapid evolution
- Scalability
- Platform independence
- Support for different visual paradigms

2.3.1 Generative Programming

The Generative Programming (GP) paradigm (Czarnecki & Eisenecker, 2000) proposes to use feature diagrams to specify the contents of a dialog-based GUI. A library of about 200 features, which can be combined in several ways, establishes a mapping between feature diagrams and concrete GUI elements. Generative Programing (GP) can be used with the Angie-Based (ABA) GUI generator to transform the model to a Qt interface (Schlee & Vanderdonckt, 2004).

2.3.2 CTT

The work in (Nóbrega, Nunes, & Coelho, 2005) proposes to use ConcurTaskTrees (CTT) formalism for a runtime prototyper of multimodal user interfaces. The main assumption is that the task model is the central artifact around which a GUI should be prototyped. Abstract user interface elements are connected to task elements. A runtime system interprets the task model and assembles a concrete user interface in the form of a dynamic web page. Modifications at runtime are directly reflected in the interface. The main goal of this strategy is to facilitate the integration of system design, UI design and usability evaluation, making it easy to refine the prototype through intervention on the model.

2.3.3 Runtime adaptation

The problem of modeling dynamic, adaptive systems is tackled in (Fleurey & Solberg, 2009). This article does not directly focus on GUIs, but on dynamic systems in general. In it, a DSML is proposed that specifies runtime adaptation logic directly in the model, separating it from main system functionality. Constraints for adaptation can be specified, as well as property-based rules. It is also possible to establish priorities among rules, so that different context can apply different adaptation strategies. This work also tackles the validation of the specification through the use of invariants and simulation.

2.3.4 ICO

Interactive Cooperative Objects, or ICO (Bastide, Navarre, & Palanque, 2003), is a formal notation for modeling interactive systems, and the di-

alog part of interactive applications. It is an iterative process producing high-fidelity prototypes of the system. It uses an underlying high-level petri net model for describing interface logic, coupled with a visual environment for the design of the GUI.

2.3.5 Trident

Trident (Bodart et al., 1995) proposes knowledge-based techniques for automatic UI generation. It suggests interaction styles, selects interaction objects and layout, and produces windows based on the task model. It queries the developer for information about tasks, users and workplace, and uses tables of rules as a knowledge base for producing interface prototypes. It can also use fuzzy logic for more flexibility.

2.3.6 DIGBE

DIGBE (Penner & Steinmetz, 2002) concerns a specific domain, namely digital control systems for buildings. A domain model provides the semantic basis for interaction design. An interaction model provides design knowledge required for task and interaction design. A presentation model converts interaction design to UI. The approach has been generalized in MAID (Penner & Steinmetz, 2003).

2.3.7 ARTStudio

ARTStudio (Calvary, Coutaz, & Thevenin, 2001) researches *plastic* user interfaces, i.e., interfaces that can adapt to several platforms. There is a unifying reference framework for the specification of plastic UIs. A set of ontological models exist for multi-targeting: domain models, context models (users, platforms and environment) and adaptation models (reaction to context change). ARTStudio is an abstract work per se, but in (Calvary et al., 2003) there are examples of instantiation in PetShop, TERESA and other tools.

2.3.8 Ergoconceptor

Ergoconceptor (Moussa, Kolski, & Riahi, 2000) proposes automatic generation of 2D GUIs for control applications of industrial processes. It is based

on three models: a physical model of the industrial process; a structural model, describing the organization and data flow of subsystems; and a functional model, describing the relations and influences between variables using causality networks (a Petri net-like formalism). From this models, a UI specification is generated containing several implementation alternatives which take into account ergonomy and usability rules. The developer should choose the most appropriate alternatives. A final prototype is then produced, which the developer can further refine.

2.3.9 Envir3D

Envir3D (Vanderdonckt, Chieu, Bouillon, & Trevisan, 2004) presents model-based prototyping of 3D virtual interfaces. A library of Abstract Interaction Objects is used to abstract classes of Concrete Interaction Objects. A 3D modeler is then used to create a 3D world where the interface objects are to be instantiated. The virtual world is saved as an XML-based representation, and it is used to generate an interactive VRML scene. Model checking and usability evaluation is also possible on the basis of the XML definition.

2.3.10 Evaluation with respect to our criteria

Table 2.3 evaluates GUI development approaches with respect to the criteria in Section 2.3. We see that the Ergoconceptor approach satisfied best our set of criteria, except for the flexibility on visual paradigms supported, which is best satisfied by ArtStudio. The approach we will show takes inspiration from these two, streamlining the Ergoconceptor approach and reusing the multi-model idea of ArtStudio.

2.4 Summary and conclusion

In this chapter we reviewed several tools and techniques associated to the engineering domains tackled by this thesis. The two big families of techniques we reviewed were DSML engineering and model-driven GUI development. We evaluated them with respect to a set of criteria we gave, and found out that the most adapted DSML engineering technique was the MDE approach. Concerning GUI development, the approach which suited best our goals was Ergoconceptor.

	Automation	Rapid Evolution	Scalability	Platform Independ.	Diff. Vis. Paradigms
GP	**Good**	Some	Low	**Good**	Low
CTT	**Good**	**Good**	Low	**Good**	Low
Runtime Adapt.[a]	**Good**	**Good**	**Good**	**Good**	–
I.C.O.	**Good**	Low	**Good**	**Good**	Low
Trident	**Good**	**Good**	**Good**	**Good**	Low
DIGBE	Some	Some	Low	**Good**	Low
ArtStudio	Some	**Good**	Low	**Good**	**Good**
Ergocon-ceptor	**Good**	**Good**	**Good**	**Good**	Low
Envir3D	**Good**	Some	Low	**Good**	Some

[a]The referenced work (Fleurey & Solberg, 2009) is not specific on GUIs and the evaluation is only relative to the specification of adaptation.

Table 2.3. Criteria satisfaction for GUI development approaches

Based on this review, we were able to further concretize the methodology we wanted to propose. What was needed at this point was a concrete case study to serve as an application for our experiments. Chapter 3 will describe the system we chose as a case study. The purpose of the chapter is mostly for reference and documentation, so a reader who is rather interested in the methodology description should probably quickly skim through it before proceeding to Chapter 4 (page 49). Chapter 3 will be referred to as needed in the rest of this document.

Part II

Methodology

Chapter 3

Case study: CMS Cosmic Rack

As a reference for the examples in this thesis, we will use a case study from the European Laboratory for Particle Physics (CERN) in Geneva, published in (Risoldi, Masetti, Buchs, Barroca, & Amaral, 2008).

The *Compact Muon Solenoid* (CMS) experiment at CERN is a large particle detector installed along the Large Hadron Collider facility. Its Silicon Strip Tracker component is a complex system made of about 24000 silicon detectors, organized in 1944 *Power Groups*. These have several environmental and electric parameters to monitor. Tens of thousands of values and probes have to be controlled by the Tracker Control System (Dierlamm et al., 2008).

This system was chosen as a case study for a number of reasons. First of all, it has a complex hierarchical structure with a large number of components of different types. The states of the components are interdependent, and vary in a complex way with time as the state of the system evolves. Then, there was an interest on the part of the users of the system to achieve a different type of GUI development process than what was in place at the time. Also, the richness of information about the system posed a low risk of not finding complete enough data about the system structure. Finally, the type of users were not mainly computer scientists, which avoided introducing a positive bias in the way the information was gathered.

The CMS Tracker is a unique prototype which, at the start of this project, was being built for installation in the Large Hadron Collider (it has since been installed and is now operational). For strong reasons of safety and access rights we could not work on the actual tracker. We worked on an early dummy of the Silicon Strip tracker, called the **Cosmic Rack**. This is equiv-

alent to a section of the tracker's barrel, maintaining the same hierarchical complexity, but with a reduced total number of components. The Cosmic Rack has been used at CERN to test the hardware and software of the full tracker. It was used first as an engineering exercise to assemble the Tracker. Then it underwent irradiation, first by cosmic rays and then by man-made particles, to see if the physics data output would be correct and if all control subsystems were integrated in the right way. Figure 3.1 shows the Cosmic Rack mounted in a protective metal casing.

Figure 3.1. CMS Tracker Cosmic Rack

3.1 The domain of control systems

Control systems (CS) can be defined as mechanisms that provide output variables of a system by manipulating its inputs (from sensors or commands). While some CS can be very simple (e.g.,a thermostate) and pose little or no problem to modeling using general-purpose formalisms, other CS can be complex with respect to the number of components, dimensions, physical and functional organization and supervision issues.

A complex control system will generally have a composite structure, in which each object can be grouped with others; composite objects can be, in

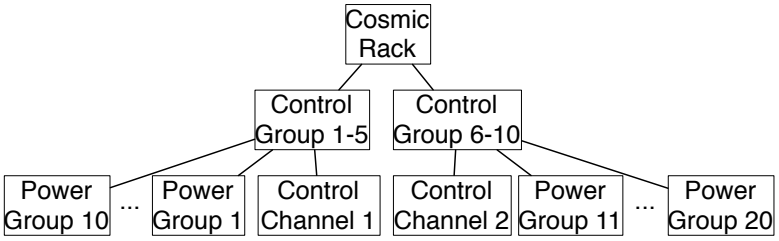

Figure 3.2. Cosmic Rack hierarchy

their turn, components (or "children") of larger objects, forming a hierarchical tree in which the root represents the whole system and the leaves are its most elementary devices. Typically this grouping will reflect a physical container-contained composition (e.g., an engine contains several cylinders), but it could reflect other kinds of relations, such as functional dependence or control logic. Elementary and composite objects can receive commands and communicate states and alarms. It is generally the case that the state of an object will depend both on its own properties and on the states of its subobjects.

Operators can access the system at different levels of granularity, and with possibly different types of views and levels of control, according to several factors (their profile, the conditions of the system, the current task being executed).

3.2 Structure of the system

The hierarchical structure of the Cosmic Rack is shown in Figure 3.2.

There are four types of components: *Partitions*, *Control Groups*, *Power Groups* and *Control Channels*. There is only one Partition, the *Cosmic Rack* object. There are two Control Groups; twenty Power Groups (ten per Control Group); and two Control Channels (one per Control Group). The shape of every component, and its position in space, was defined by mechanical engineers in the Cosmic Rack mechanical project. This information is stored in a database at CERN, and in mechanical drawings.

3.3 Component behaviour

Each component is characterized by a finite state machine (FSM). They are represented in Figures 3.3 (for Partitions and Control Groups, which have the same FSM), 3.4 (for Power Groups) and 3.5 (for Control Channels). No initial state is formally defined, and the control software will be initialized to whatever the state of the system is at startup.

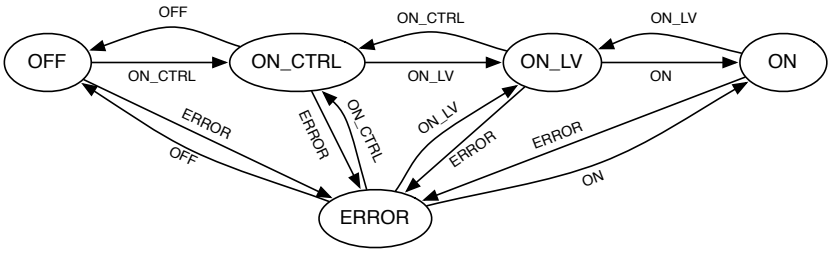

Figure 3.3. Partitions and Control Groups FSM

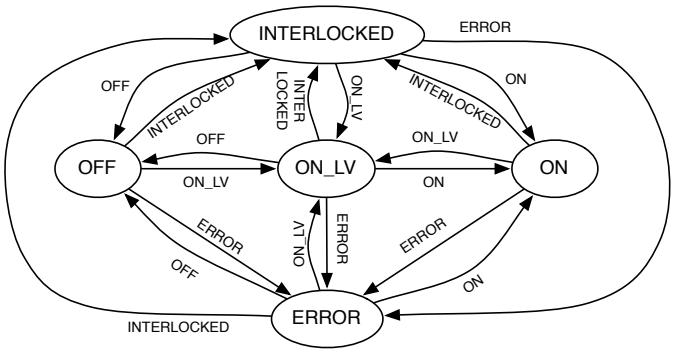

Figure 3.4. Power Groups FSM

Each component can receive commands. Commands trigger transitions in the FSM. A command will generally trigger a transition having the same name as the command. There are some transitions, however, which are not triggered by commands, but rather by internal system events. It is the case for all transitions going to the “ERROR” state of all FSMs (triggered by property values out of range) and those going to the ”INTERLOCKED” state of the Power Groups FSM. The INTERLOCKED state is generally reached when software reactions to problems are not sufficient, or are not

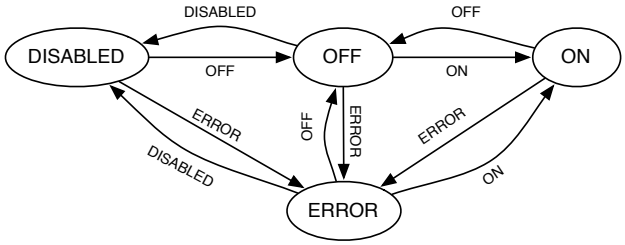

Figure 3.5. Control Channels FSM

put in place, and corresponds to components being disabled and locked. To exit the INTERLOCKED state, the only way is explicitly sending the Power Group a *Clear* command which, according to the circumstances, triggers one of the transitions leaving the "INTERLOCKED" state.

States are propagated up through the component hierarchy according to a set of rules. Power Group and Control Channel states are propagated to the Control Group above. Control Group states are in their turn propagated to the Partition. Table 3.1 shows the rule set for propagating Control Groups (CG) states to the Partition. The table also shows that the FSMs we saw are a slight simplification of reality - there exist "mixed" states, for example when going from OFF to ON_CTRL, where only some of the Control Groups have already switched state. These mixed states, however, are normally ignored as they are not part of the "ideal" behaviour of the machine – they are merely temporary states that either evolve into the next foreseen state or into an error. Control Groups in their turn also have a similar table of rules for propagating the states of their children components, Power Groups and Control Channels.

3.4 Alarms

Power Groups have a *temperature* property. Different levels of alarm are defined on the property's value intervals. Each interval corresponds to a diagnostic on the temperature (*normal*, *warning*, *alert* and *severe*). This is needed to detect temperature anomalies and take action before a hardware safety mechanism (based on Programmable Logical Controllers, or PLCs) intervenes to cut power to components in order to preserve them (which would bring the Power Group to the INTERLOCKED state - a situation better avoided as it implies stopping the machine and manually resetting of

Partition State	State of the CGs
OFF	All CGs OFF
CTRLMIXED	Some CGs ON_CTRL, others OFF
ON_CTRL	All CGs ON_CTRL
LVMIXED	Some CGs ON_LV, others ON_CTRL
ON_LV	All CGs ON_LV
HVMIXED	Some CGs ON, others ON_LV
ON	All CGs ON
ERROR	Some CGs ERROR

Table 3.1. State propagation rules for the Partition. CG stands for Control Groups.

the power groups, both complicate and expensive tasks).

3.5 Flow of operation

There are command sequences which constitute the normal operation of the Cosmic Rack. These are turning on the system, turning off the system, clearing errors and clearing interlock events.

- Turning on the system consists of turning on the Control Channels, then turning on the Power Groups.
  - Turning on the Control Channels means enabling (DISABLED → OFF) and then turning on (OFF → ON) the Control Channels (in any order).
  - Turning on the Power Groups means turning on low voltage (OFF → ON_LV) and then high voltage (ON_LV → ON) of Power Groups (in any order).
- Turning off the system consists of the inverse sequence as turning on the system.
- Clearing an error consists of sending commands to a Control Channel or Power Group in ERROR state, according to the situation at hand (most of the times, this will mean trying to power down a component and

power it up again with a turn off / turn on sequence on the concerned branch).

- Clearing an interlock state consists in sending a *Clear* command to an interlocked Power Group.

3.6 Important properties

From the point of view of the user, the following properties have to be satisfied by the Cosmic Rack:

1. An object should only be in one state at a given time;
2. Sending a command should trigger the desired state change, as long as the environmental conditions allow it. Conversely, a state change should never occur if the environmental conditions dictate otherwise, even if a command has been given in that sense;
3. An object should never get blocked in a given state;
4. A state change in a sufficient number of objects at a given level should always be reflected in the state of the next level object, according to state propagation rules;
5. A property value change should always trigger an alarm if the value is outside of fixed thresholds;
6. In the case of several parallel alarms, the ones with a highest priority should be shown over the lowest priority ones.
7. An interlock should occur independently of object state or user commands, when the conditions dictate it.

Risks are associated with the non-respect of the previous properties.

1. Inconsistent state (critical): if objects are considered to be in several states at once, inconsistent transition sequences can occur. The control interface cannot reliably show the state of an object which is in an undefined state. This leads to unforeseen and potentially catastrophic behaviour of the system.

2. Unresponsive system (serious): if the system does not react to commands, the operator is not in the condition of acting to solve a fault. This can potentially lead to the interlock of the system, with the associated cost in time and effort. An unresponsive system can be caused by communication failures, design bugs (like a deadlocked Finite State Machine (FSM)) or hardware faults.

3. "Fake" states (critical): an FSM which deadlocks in a "good" state gives the impression that the object is doing fine, while it is actually out of its operating range. This leads to lack of action to solve problems, or to wrong commands when alarms are triggered but the objects looks OK (e.g., ignoring alarms).

4. Invisible problems (serious): problems which are not propagated up in the object hierarchy's states can negatively influence the system by delaying the time of response to a problem, eventually leading to a possible interlock of the system.

3.7 Summary and conclusion

We described here the concrete case study we used as a guide throughout the development of the project. A couple more case studies were also adopted; these are discussed in Section 10.4 (page 180) but are not used in the examples in the remainder of this document.

Having some real-world case studies, instead of just a toy case study, helped us focusing some of the critical difficulties of adapting a methodology to a given domain. One important aspect that was clear, as it entailed several iterations in the definition of our techniques, was that a successful methodology must support modularity. Requirements for this case study came in several waves. This would have caused major overhead had we not designed the methodology in such a way that it was relatively easy to add, remove or refine steps or language features. In a way, not only our case studies were instrumental in evaluating the methodology *a posteriori*, but they were essential in defining the guiding principles from the very start.

Chapter 4 will first introduce the general ideas underlying our methodology, and discuss what advantages it presents. We then will talk about some methodological choices that constrained our work, and give an overview of the process as we designed it. The information model will be presented in an

abstract way. We will also briefly go over the transformation and verification goals. The focus of the chapter is on concepts and workflow rather than on implementation. We will concentrate on the more technical aspects in the chapters that follow Chapter 4.

Chapter 4

Methodology

This chapter illustrates the definition of the GUI prototyping methodology, the abstract model of the problem domain, the semantics definition and some considerations on what is needed for the validation activity.

4.1 Principles

There is a basic assumption on which the methodology is based, that is, that a lot of the information needed to develop a user interface for a control system can be inferred from the description of the control system itself. Consider the type of system under control: it is made of objects which typically have commands (inputs) and events (outputs). They will have parameters that can be read and/or set, and will be organized in a more or less hierarchical way. In the case of a physical system, objects will have a shape and location in space.

All of this information might seem a lot to specify for building a GUI. However, if we consider that most of it (if not all) is already present when designing a control system, one starts to see the value in reusing it for GUI development. For example, the names of the system components can be used to identify elements in the interface. The commands and events that are defined in the hardware can be reflected in the commands and events managed by the interface. Also, the hierarchy of the system can be represented by a hierarchical organization of the user interface (e.g., hierarchical panels, or a scene tree in a 3D interface).

4.1.1 The advantages of the BATIC^{3}S methodology

The methodology we discuss here is named BATIC^{3}S (an acronym for *Building Adaptive Three-dimensional Interfaces for Critical Complex Control Systems*, which was the name of the collaboration in the context of which it was developed). The idea it promotes is that if system design is done using an appropriate and coherent language, and provided that the system description is sufficiently detailed to solve GUI-related issues, then GUI development can follow with a relatively minor effort. For this reason, the methodology is centered around the definition of a DSML to specify control systems; the GUI development part is completely based on model transformation. In the examples that we will provide, it will be evident that we always focus on modeling the system, while the GUI generation is mostly a push-button operation.

Also, in our survey we noticed that control systems GUI developers often need a system simulator to evaluate the GUI against. This is necessary because the actual control system can often be not available (either because it is still being built while the GUI is developed, or because it is too expensive and delicate to be used by a prototype GUI). This led us to include in the methodology the automatic generation of a software executable system simulator. The simulator can be interfaced to the GUI in the prototyping phase to evaluate its functioning and/or train end users. This is the second advantage in generating the GUI from a system specification.

4.2 Methodological choices

There are a few *a priori* principles that guided our methodological choices in the development of this work.

First of all, we wanted to avoid the "yet another modeling framework" effect: rather than developing a brand new formalism and set of tools, we focused on reusing current and accepted technologies. In our point of view, the limitations of domain-specific modeling are not coming from intrinsic limitations of existing tools, but rather from the lack of integration between them.

We also chose to adopt a modular strategy. The methodology is organized as a workflow, in which each step can be accomplished by one or more technologies. There are often several ways to accomplish a certain task; we will illustrate the choices we made, and we will also give pointers to possible

extensions of our work in one direction or another.

Finally, we chose to use open source and free frameworks. One of the reasons is that domain-specific modeling is still mostly a research field, with much of the work being defined by education institutions - which inevitably brings to a large availability of free software. Also, we find that a contribution based on free software is easier to extend in the future for other collaborations.

4.3 Engineering process

The engineering process for prototyping a GUI for a control system is illustrated in Figure 4.1. There are four steps in this methodology. In step ①,

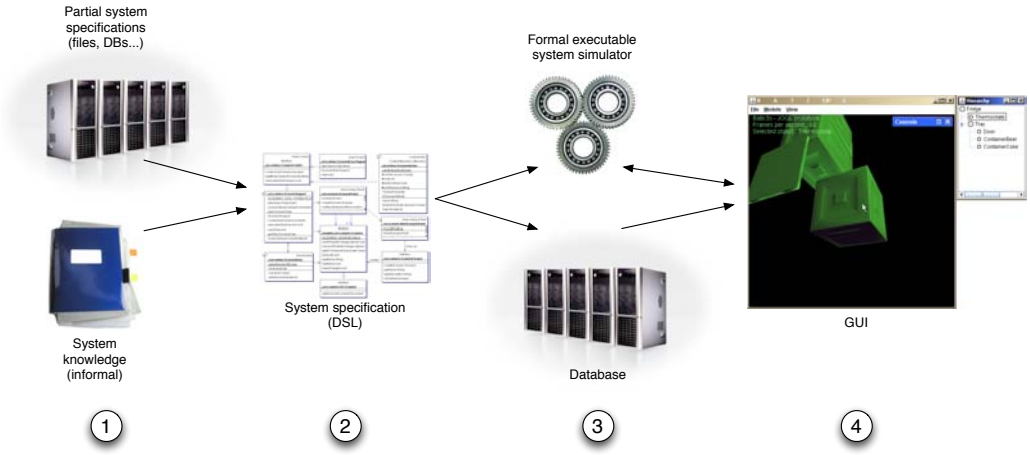

Figure 4.1. The engineering process

knowledge about the control system is gathered. This is usually present in the form of a collection of more or less formal documents. In the case of complex systems, it is often the case that many of the aspects of the system are modeled in electronic form for engineering purposes. These models can have various levels of reusability depending on the format they are in. Knowledge about the system is essential because the composition of the system, its inputs and outputs, and its behaviour in terms of state evolution are key information for automated GUI prototyping.

In step ②, this information is expressed using a DSML. This language models the domain of the information gathered in step one. Based on our

analysis of the domain, we found that there were three distinct groups of information we should represent: how the system is built and works; what is the visual aspect of the system as we want it to be represented in the GUI; and how users interact with the system. As a consequence, the domain model is comprised of four models: the *system model* describes the structure and internal behaviour of the system; the *visual model* describes the geometry of the system; and the *user model* and *task model* describe the interactive aspects of the system. An abstract overview of the models is shown in Figure 4.2. The various models are not completely separate, but are linked by several relationships, abstracted in Figure 4.2 by lines.

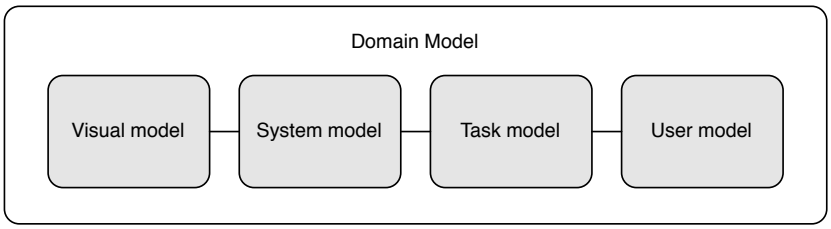

Figure 4.2. Packages of the domain model. Arcs are abstractions of existing relationships among classes in the packages.

Step ③ of Figure 4.1 sees the generation of deliverables from this specification: a database containing data used for GUI generation (the visual, user and task model and part of the system model), and an executable system simulator. This is done by automated tools.

Finally, step ④ is the dynamic generation of a GUI prototype built from the database data, which interacts with the system simulator.

Steps ② to ④ are those tackled by our model-based approach. We will now describe the features of the domain specific model, then the simulator generation and GUI prototyping activity.

4.4 Domain model

Figure 4.2 shows how the domain model is divided into four packages, representing different concerns of the domain. Following is a description of each model package. In this section we will give an abstract, conceptual description of the model. The following chapter will translate these concepts to concrete modeling formalisms.

4.4.1 The System model

The system model contains useful abstractions to describe the structure of the system and its behaviour. It includes the following concepts:

- Objects in the system
- Types, defining sets of similar objects
- A hierarchical composition relationship between objects
- Behaviour of objects in terms of states and transitions
- State dependency between components
- Properties of objects
- Commands and events of objects

Objects are identified by a name and represent components in the system.

Types define all features which are not specific to individual instances of objects. Typing objects enables quick definition of properties common to a large number of objects (a typical situation in control systems which are highly repetitive). A type is identified by a name.

The hierarchical composition is modeled as a tree of objects. Each object can have one parent and/or several children. The hierarchical relationship can semantically express either physical containment or logical groupings (e.g., electrical connection, common cooling pipes...). Also, the hierarchical composition is reflected in types: each type can be defined to contain a certain number of other types. This allows the definition of a hierarchy template against which the object hierarchy can be validated.

Behaviour of objects is modeled by finite state machines (FSMs). These are well-suited for control systems as they express expected behaviour in a clear way and are a standard in the control systems domain.

State dependency is expressed with conditional rules. Conditions can be of type "if at least one of the children (or – if all of the children) of an object is in state x, then go to state y".

Properties of objects are identified by names and data types. Their possible values can be divided in intervals corresponding to four diagnostic levels: normal, warning, alert or severe.

Commands: as control systems are asynchronous by nature, commands are defined by their name and parameters only (i.e., no return value). For the sake of simulation, the possibility of defining a simplified behaviour for a command is provided (e.g., it is possible to specify that a command changes a property of an object).

Events are defined by their name and parameters. An event can be triggered by state transitions (also in children objects), commands or property changes. An event can also trigger state transitions, commands and/or other events.

The behaviour of objects, their commands, events and properties are not directly associated in the model to individual objects, but rather to their types. This is because they are common features of all objects of a given type (e.g., all power supplies will have the ON and OFF commands). Individual objects instantiate types, inheriting all of the type's features without having to specify them for each object. Figure 4.3 shows an abstract overview of the relationships among concepts in the system model. Each edge in the figure sums up one or more relationships existing among the concepts. Individual relationships will be detailed in chapter 5, while discussing the language metamodel.

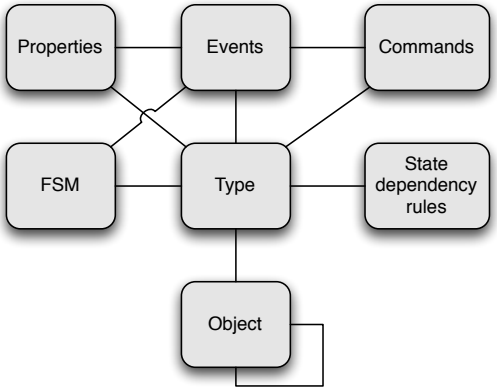

Figure 4.3. Concepts in the system model and their relationships

4.4.2 The Visual model

The visual model describes all aspects of the system which are visually relevant. This includes their geometrical space and their position in space.

Geometrical shape is defined for each type (and not per-object, for the same reasons mentioned earlier). The geometry is expressed by association to a geometry file URL. This is a file in the *Object* format (Burkardt, n.d.), a commonly used standard for 3D object description. The language also includes pre-defined common primitive shapes (box, sphere, cylinder...) that are mapped to pre-made object files. Geometrical shapes can be modified by scaling attributes. For this reason a single shape can be used to model different objects. For example, using the cube primitive, one can model different cuboids by modifying its scale along one or more axes.

Position in space can be defined in different ways. The simplest is expressing translation and rotation for each object. This requires defining coordinates for each object. Components of complex systems, however, are sometimes positioned according to repetitive patterns (arrays, circles...). Thus, a more efficient choice could be positioning an object by specifying its relationship with other objects. Relationships of type "x is parallel to y", or "x is concentric to y" can be repeatedly applied to rapidly express whole sets of coordinates in such cases.

Figure 4.4 shows an abstract overview of the concepts of the visual model and their relationships. Individual relationships will be detailed in chapter 5.

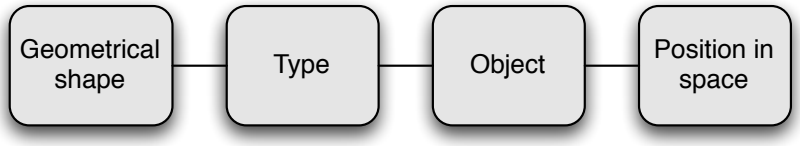

Figure 4.4. Concepts in the visual model and their relationships (`Type` and `Object` are those from the System model)

4.4.3 The User model

As a general definition, the user model is a knowledge source that contains a set of beliefs about an individual on various aspects, and these beliefs can be decoupled from the rest of the system (Kobsa & Wahlster, 1989).

Other participants to the BATIC^{3}S project built a user model based on the *Generic Ontology based User Model* (GenOUM) (Cretton & Le Calvé,

2007, 2008). This model is therefore not properly part of the author's personal work. It is nonetheless described for completeness.

GenOUM is a general-purpose user model ontology including information about users' personality and knowledge. The GenOUM ontology is quite rich and goes beyond the needs of this project. We used a subset of it, represented in Figure 4.5. The full GenOUM ontology is presented in (Cretton & Le Calvé, 2007). We model the users' profile (*Behaviour*) and their level of knowledge for tasks to accomplish. The object of a user's knowledge in the GenOUM ontology is generic (the *Thing* concept from OWL); we replaced *Thing* with the *Task* concept. The task model is described in section 4.4.4.

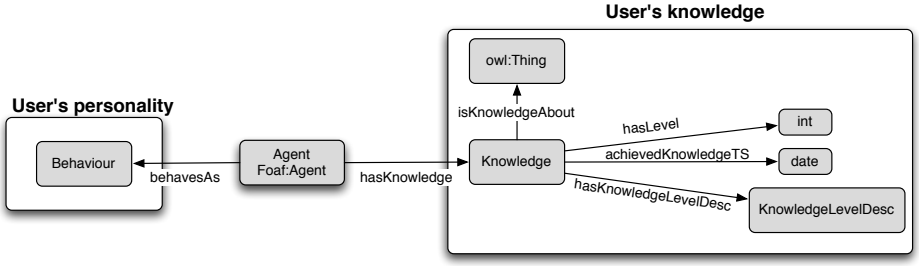

Figure 4.5. GenOUM concepts and properties

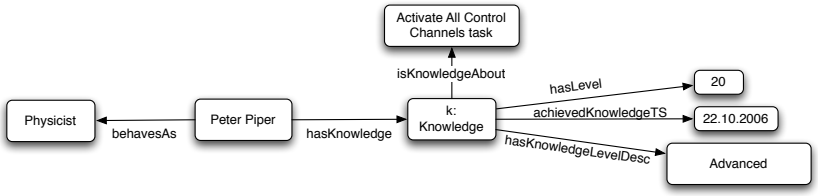

Figure 4.6. Example of User's knowledge level

As an example referring to our Cosmic Rack case study, Figure 4.6 states that user Peter Piper has a physicist profile. He knows how to perform the "Activate all control channels" task, his knowledge level about this task is 20, an advanced knowledge level, and he acquired this knowledge on October 22, 2006. The interpretation of the knowledge level '20' depends on the domain and is not defined *a priori* by GenOUM.

4.4.4 The Task model

A task defines how users can reach a goal in a specific application domain. The goal is a query about, or a modification of, the state of a system (Paternò, Mancini, & Meniconi, 1997). Tasks must be seen as structured entities. A task is a way to organize and coordinate the activities that users perform with the GUI. These activities might be at the interaction level (like sending a command) or at a more abstract level (like a concept of "sequence of commands"). Section 3.5 (page 44) showed a few examples of this. We said that Control Channels have an "enable" command, and a "turn on" command. We can then define a "turn on control channel" task, which is not directly corresponding to any single command, but rather to the goal of turning on the component, by sending in sequence the "enable" and "turn on" commands. As tasks represent the complex interaction between users and GUI, we should be able to deal with more than just atomic or linear tasks.

Based on the state of the art we reviewed earlier, we chose to use the ConcurTaskTree formalism (CTT) (Paternò et al., 1997) as a Task model. In CTT, a task is identified by a name, a type and an ordered list of subtasks, in a hierarchical composition structured like a tree. Causal relationships between subtasks are defined.

There are four task types: *abstract*, *user*, *application* and *interaction*. An *abstract* task is generally a complex task we can define in terms of its subtasks. A *user* task is something performed by the user outside the interaction with the system (e.g., deciding or reading something). An *application* task is something completely executed by the system (e.g., cashing a coin inserted in a drink vending machine). Finally, an *interaction* task is performed by the user interacting with the system (e.g., clicking a button).

Concurrency relationships between tasks are defined by a number of process algebra operators (called *temporal operators* in (Paternò et al., 1997)). A non-exhaustive list of these taken from (Paternò et al., 1997) follows as an example:

- `T1 ||| T2` is *interleaving*: the actions of the two tasks can be performed in any order
- `T1 [] T2` is *choice*: T1 or T2 can be performed
- `T1 >> T2` is *enabling*: when the first task is terminated then the second is activated

- `T1*` is *iteration*: the task is iterative

CTT has a visual syntax, representing a task as a tree (where the root is the highest abstraction of the task, and the leaves are the most elementary actions) and is supported by editing tools and libraries. For the four task types mentioned, the symbols in Figures 4.7 are used.

Figure 4.7. CTT task types: abstract, user, application and interaction

An example of CTT applied to our Cosmic Rack case study is shown in Figure 4.8.

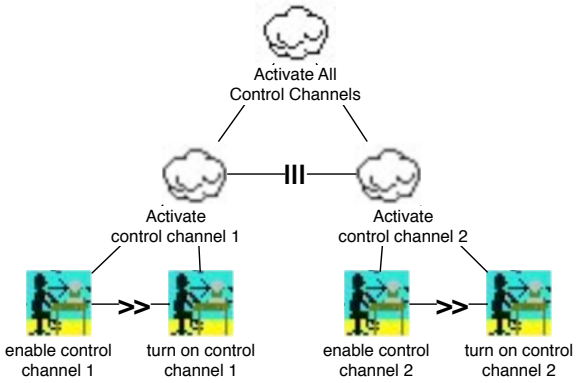

Figure 4.8. CTT for the turn on control channels *task*

Through the association of task and user models, we add to the CTT model the possibility to specify for which user profiles (the *behaviour* in GenOUM) each task is available. This will be clearer later, when we give the detailed implementation of the task model.

4.5 Transformations

The specification made using the DSL is used to create two artifacts: an executable system simulator, and a database containing GUI-related information.

The system simulator is generated so that the GUI prototype can be run agains a realistic system without having to use the actual production system. This is often a requirement in the field of interactive systems. The production system might be for example too delicate or expensive to allow testing an unfinished GUI prototype with it. In other cases, while the system specification might be complete, the production system might not be up and running yet. This happens for example with very complex systems that take a long time to build, where software development has to be performed while the system is still under construction.

The database is used to store all the information that will be used by an appropriate generic GUI engine to display the system under control. It contains all information relevant to the interface, as well as the definition of the communication between the interface and the system (commands and events).

The transformation that generates these artifacts is modular and based on rewriting patterns. The domain model is transformed into a corresponding executable model for the simulator, and into a relational schema for the database. Next chapter will give concrete examples of how transformation is performed, using actual models from our use case.

4.6 Model Checking and Validation

Model checking is

> [...] *an automatic technique for verifying finite state concurrent systems.*

(Clarke, Grumberg, & Peled, 1999). Model checking is part of the verification activity, defined in (IEEE, 1991) as

> *[t]he process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.*

Validation, still according to (IEEE, 1991), is

> *[t]he process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements.*

The activity of model checking and validation in the context of this framework is used for two main goals: validating structural constraints on the model, and checking the behaviour of the generated system simulator. The former can be of assistance in building the model; the latter serves confirming that the system simulator is actually mirroring a realistic system behaviour, and thus can be trusted when evaluating the GUI.

Model checking and validation require the developer to express *properties* that must be checked for validity. A valid property is one that is satisfied by a model. Properties that can be checked can be classified in two groups: static properties and dynamic properties.

Static properties are those which can be statically verified on the model without considering the evolution of its states. Dynamic properties are those which concern the evolution of states of the model. For these properties the state space must be calculated.

4.6.1 Static properties

A typical example of static properties are structural properties. These properties are satisfied if the structure of the model respects certain constraints.

For example, in an FSM model, a well-formed transition is always supposed to have a source and a destination state which belong to the same FSM as the transition. The well-formed transition constraint can be written as follows. Let be:

- F the set of FSMs in a model
- T the set of transitions belonging to all FSMs in F
- S the set of states belonging to all FSMs in F
- $\alpha : T \to S$ the function associating a transition to its source state
- $\beta : T \to S$ the function associating a transition to its destination state
- $\tau : T \to F$ the function associating a transition to the FSM to which it belongs
- $\sigma : S \to F$ the function associating a state to the FSM to which it belongs

a well-formed transition t satisfies the constraint

$$\sigma(\alpha(t)) = \sigma(\beta(t)) \ \wedge \ \sigma(\alpha(t)) = \tau(t)$$

Constraints of this kind are generally expressed using constraint languages like OCL (Object Management Group, 2006). Verification of constraints can be performed directly at design time, even on partial models, as it requires no model execution.

4.6.2 Dynamic properties

As a model is simulated, it evolves from an initial state to a number of possible following states. Each state represents the union of values of all variables of the model. According to the semantics of the model, it is possible (or not) to pass from a given state to another given state. The union of all possible states and transitions is called state space.

Dynamic properties are expressed on this evolution of states. For example, again in an FSM model, it might be desirable that all transitions are live. That means, for each transition, the model must always be able to reach a state in which the transition can be fired. This kind of properties can typically be expressed using some kind of temporal logic, like Computational Tree Logic (CTL) (Huth & Ryan, 2000) or Linear Temporal Logic (LTL) (Pnueli, 1977). Let *pre* be the source state for a given FSM transition, the liveness property for that transition can be written in CTL as follows:

$$AG(EF(pre))$$

In a few words, this formula says that for every possible state evolution of the system (*Always Globally*, AG) there exists eventually a state (*Exists Finally*, EF) where *pre* holds. Verification of this kind of properties can be performed by calculating of the state space of the model, and subsequently checking the truth value of properties on the state space. This is computationally a very complex task. Several theories and tools have been devised or are under study to tackle it in an efficient way.

4.7 Summary and conclusion

This section gave an abstract description of the guiding principles, the workflow and the models involved in the methodology. We described how the

information about the system is reunited in a domain-specific model. The latter is used to generate a runnable prototype of the system, and a database for the GUI generation. A GUI engine loads the database to build the interface, and communicates with the simulator in order to enable evaluation of the GUI.

We also discussed how different model checking and validation techniques can be applied to this methodology in order to validate the model and verify behavioural aspects of the simulator.

Part III of this document will discuss how we implemented the methodology, what technologies we chose, and how we integrated them. We will start with a chapter describing in detail the DSML we used for our case study. Then we will in turn talk about the transformation framework, the GUI engine, and finally the model checking and validation technique. Concrete examples based on the case study will be given for each phase of the workflow. Part III will be wrapped up by a chapter giving a more general point of view on this work.

Part III

Model and techniques

Chapter 5

The specification language: Cospel

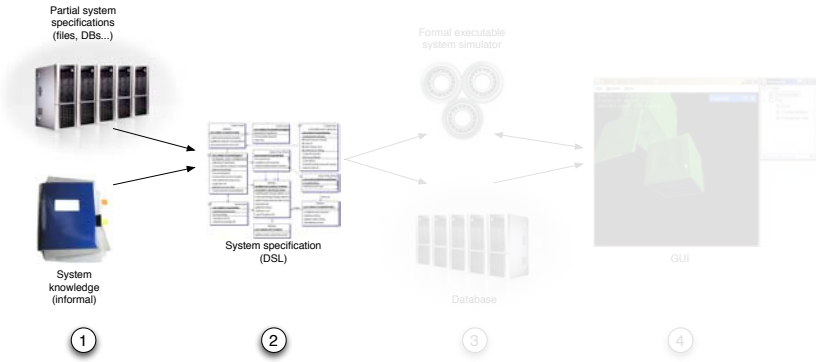

Figure 5.1. From knowledge to specification: the Cospel language

One of the purposes of the methodology we propose is to introduce a comprehensive DSL for the domain of control systems that is based on useful abstractions for domain experts. This constitutes the passage from step 1 to step 2 of the BATIC^{3}S methodology in Figure 5.1. We already saw in the previous chapter that the domain model is modular, made of different aspects which are interwoven.

This led us to design a DSL called **COntrol systems SPEcification Language (Cospel)** that is built out of different parts, or *packages*, modeling different aspects of the system. Here we give the abstract syntax of Cospel by EMF metamodels. The semantics are given by transformation to a different formalism, which will be described in section 6.

5.1 Reasons for choosing EMF

Our main criteria when choosing a technology for syntax definition were the following:

- Evolution: it should be easy to add, remove or change parts of the language as needed;
- Interfacing: it should be easy to access models made with our language either by hand or by software, to ease the transformation process;
- Automation: upon modifications to the language, editors and APIs should be modified automatically;
- Clarity: it should be easy to read and document the syntax definition.

After examining the state of the art for language design techniques at the time the framework choice was made, we found that our needs were best satisfied by approaches based on metamodeling techniques. Among the technologies available at that moment, the Eclipse Modeling Framework (Budinsky, Steinberg, Merks, Ellersick, & Grose, 2004) (EMF) was the one best satisfying our criteria for the following reasons:

- Evolution: metamodels can be easily modified or redefined. As long as certain constraints are respected, older specifications remain compatible, or are modified to be. Generated artifacts can be manually modified too.
- Interfacing: models are generated in XMI (XML Metadata Interchange), an XML standard for metadata representation, which is easy to parse. Artifacts are generated in Java, which is easily ported to various platforms and integrated with other technologies.
- Automation: EMF includes an automatic generator of editors and APIs, which are refreshed when the language metamodels are changed.
- Clarity: the metamodels can be seen as UML class diagrams, tree representations, and even XML; metamodels can be divided in packages, and can import other metamodels for greater modularity.

5.2 Cospel packages

Cospel specifications can be created via an editor, generated automatically by EMF from the abstract syntax. The concrete syntax used by this editor is a tree-like visualization of the specification, where each node is a syntactic element. The tree is structured according to the aggregations in the abstract syntax. Properties and associations other than aggregations can be specified by editing properties of tree nodes. Specifications are serialized in XMI. The editor runs as an Eclipse application.

Based on extensions of related work (Pedro, Risoldi, Buchs, Barroca, & Amaral, 2010), Cospel is composed of several packages (following the order in which models have been introduced in Section 4) which are integrated into a single metamodel. Packages concretize the abstract models (System, Visual, Task and User) we saw in the previous chapter, according to Figure 5.2.

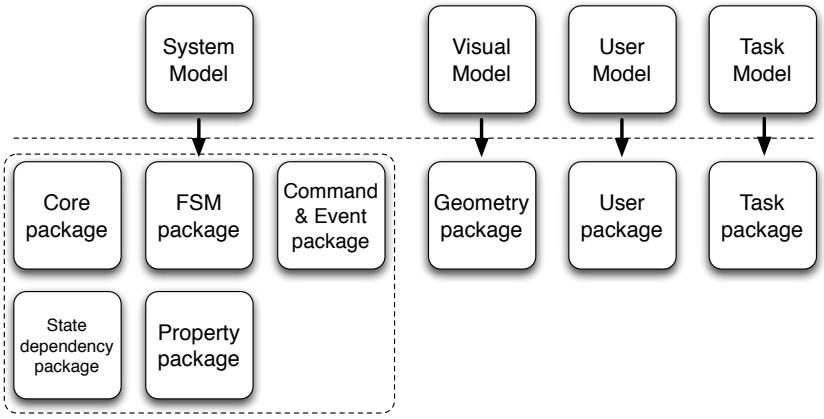

Figure 5.2. Cospel packages corresponding to the abstract models

For each package, we will see the metamodel, and give a few properties. Some of these properties will be enforced by the metamodel itself, for example by cardinalities. Others will be constraints on the model that may involve multiple metaclasses in different packages, and will be given as OCL constraints in Section 5.2.9. Finally, some properties will rather concern behaviour, and will be the object of model checking in Chapter 8.

5.2.1 The Cospel core package

The Cospel core package is part of the System model. It defines the hierarchical structure of the control system. Figure 5.3 shows the metamodel of this package. The `Specification` element is the top level element for any Cospel specification. It serves as the container for all specification elements. The `Type` element serves as a template for similar objects: all features which are common to a number of similar objects can be modeled only once in their type. This choice is motivated by the highly repetitive structure of control systems, and is supported by the common practice of using such a template system to reduce the workload of specification in the domain. The other key concept here is the `Object`. It represents an individual component, and it models the hierarchical structure of the control system (an object can be the child of another object). All other features of the system are modeled in other packages and associated with either the type (for features common to many objects) or the object (for object-specific features).

Note that there is only one level of typing, i.e., there is no "subtype" concept for the type. This comes from the domain analysis that led Cospel's design. However, it would be fairly easy to introduce a hierarchy of types by extending this core metamodel, for example by adding a recursive `subtypeOf` containment relationship to the `Type` metaclass.

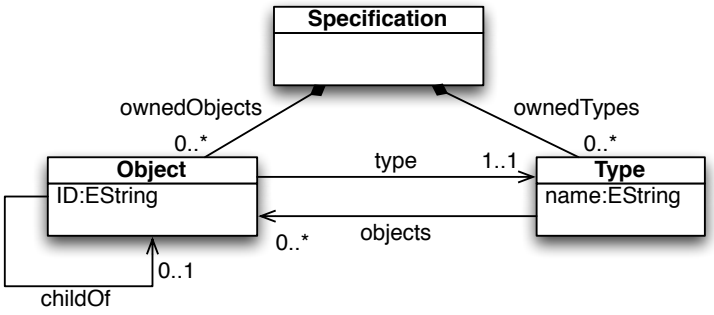

Figure 5.3. Cospel core metamodel

With respect to our Cosmic Rack example of Figure 3.2, we model here four types: `PARTITION`, `CTRL-GRP`, `CTRL-CHN` and `POWER-GRP`. Individual objects are then defined (and associated to their type): `CosmicRack`, `CG1-5`, `CG6-10`, and two `ControlPowerChannel`, plus all Power Groups from 1 to 20 (numbered according to a layer-and-index convention, e.g., the eigth Power

Group is the second of the fourth layer, thus `PG-Layer-4-Rod-2`). Figure 5.4 shows a screenshot of the Cospel editor with the definition of a partition, some objects and some types. The bottom panel shows the properties of the selected `CTRL-CHN` type, namely the association with the objects of that type. Associations with other elements are also shown, which will become clearer as we introduce further packages, like FSMs in the next paragraphs.

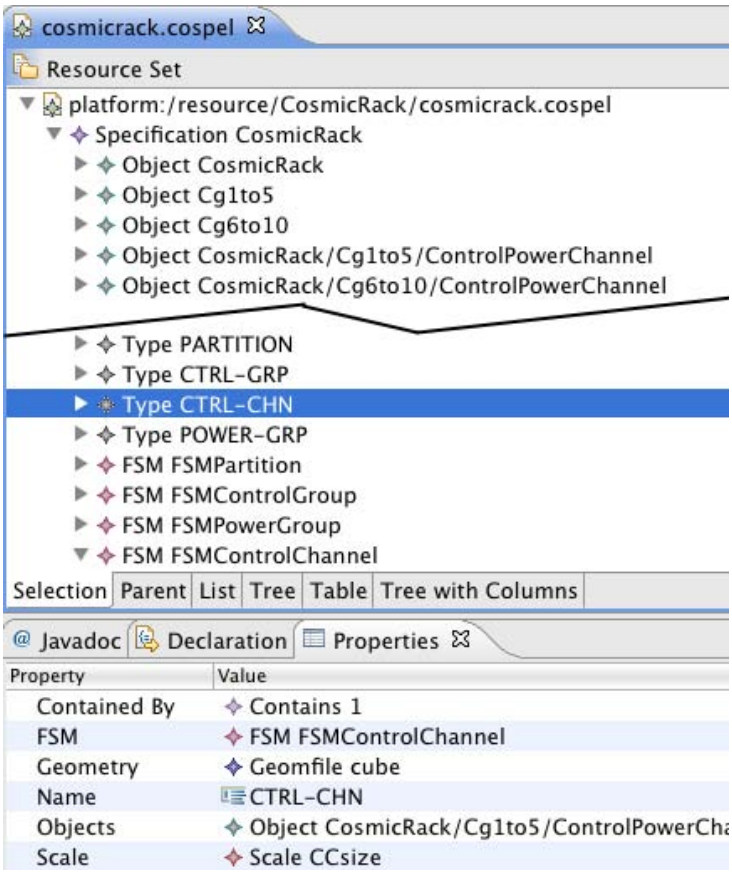

Figure 5.4. Cospel editor with core package elements (the image has been cut in the middle)

Properties of the core Package

Properties of models conforming to the Cospel core package are trivially satisfied by construction, as they are explicitly expressed in the metamodel. They are:

- An object may only have one parent
- An object always has one and only one type
- Object names are unique (EMF lets the designer set a `Unique` attribute property to enforce unicity on an attribute)

5.2.2 The Finite State Machine (FSM) package

The FSM package is part of the System model. The behaviour of objects in terms of states and transitions is modeled via a simple FSM model. This is supported by FSMs being well known and understood in the domain. An `FSM`, comprised of `States` and `Transitions`, is associated to a `Type` (from the core package). This means that all objects of that type will have that FSM. Figure 5.5 shows the FSM package metamodel.

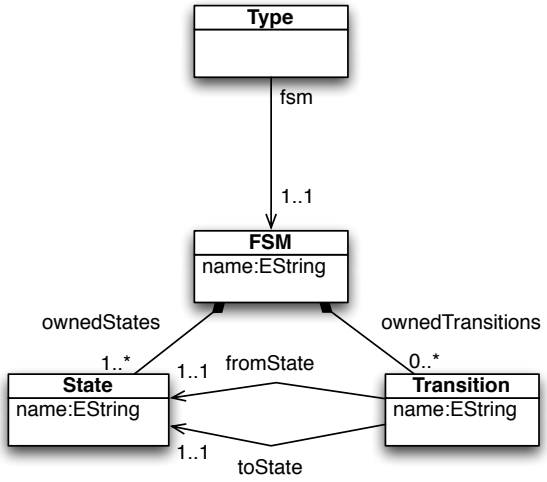

Figure 5.5. FSM package metamodel

Using this metamodel we can, for example, model the FSM of a Control Channel (from Figure 3.5). We create four states: `DISABLED`, `OFF`, `ON` and `ERROR`. Also, we define a transition for each arc in the FSM. These states and transitions are grouped in an FSM called `FSMControlChannel` which is associated to the `CTRL-CHN` type. Figure 5.6 shows the Cospel editor with the definition of `FSMControlChannel`. One of the states, `OFF`, is selected to show its properties in the bottom property panel (namely, its association with transitions).

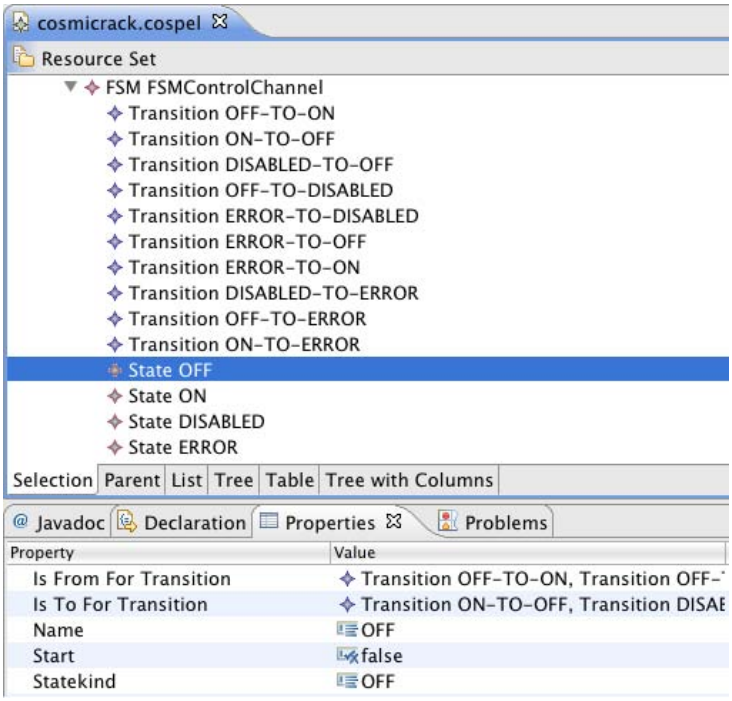

Figure 5.6. Cospel editor with FSM package elements

Properties of the FSM package

Some of the properties of models conforming to the FSM package are satisfied by construction; others are not explicit in the metamodel and must be expressed as additional constraints. Following is the list of properties:

- A type is always associated to an FSM
- An FSM always has at least one state
- A transition always has one `fromState` and one `toState`
- The `fromState` and `toState` of a transition belong to the same FSM as the transition itself (not explicit in the metamodel, checked by OCL constraint, see OCL formula 1 on page 82)
- An FSM is not deadlocking, and from any state it is possible to go to any other state (through a transition sequence) infinitely often (not explicit in the metamodel, needs model checking to be verified)

5.2.3 The State dependency rules package

The State dependency rules package is part of the System model. Recalling what was said in section 4, we want to be able to define rules in the form *"if one/all of the children of this object are in state x, set this object to state y"*. This is achieved by associating a state composition rule (`BooleanStateCompositionRule`) to a type. This rule can be of two kinds: `OneChildInStateRule`, which is triggered if at least one of the children is in a given state; and `AllChildrenInStateRule`, which is triggered if all the children are in a given state. All objects of the associated type will implement this rule. The rule is associated to the triggering state of the children via a *childrenState* relationship. It is also associated to a resulting state of the parent via a *resultingState* relationship. Figure 5.7 shows the metamodel of this package. The domain analysis showed these types of rules were sufficient. A possible refinement of this metamodel for slightly different systems could include rules that quantify a threshold number or percentage of children objects in a given state as a trigger.

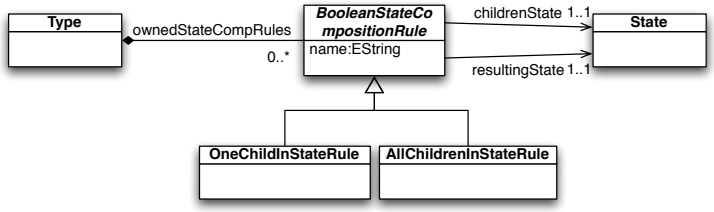

Figure 5.7. State composition rules package metamodel

For the Cosmic Rack example, referring to Table 3.1, we declare a `PartitionErrorDependency` rule to implement the last line of the table. The rule is of the `OneChildInStateRule` kind (having some Control Groups in error will trigger the rule). This rule is associated with the `PARTITION` type. Through the *childrenState* relationship, it is associated to the Control Groups' `ERROR` state. Through the *resultingState* relationship, it is associated to the Partition's `ERROR` state. Figure 5.8 shows the Cospel editor in the process of defining the mentioned `PartitionErrorDependency` rule.

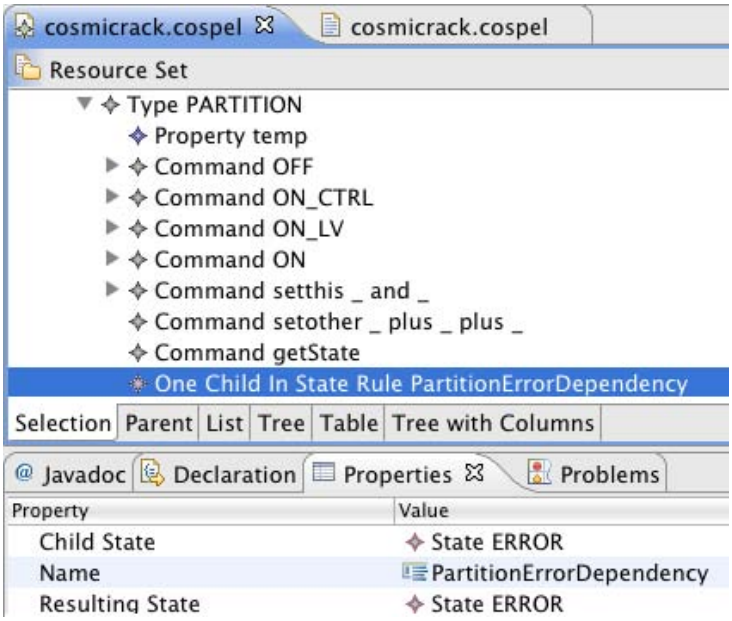

Figure 5.8. Cospel editor with state composition rule package elements

Properties of the state composition rule package

This package has two types of properties: local ones and cross-package ones. Local properties are easy to list, and again they are satisfied by construction as they are explicit in the metamodel:

- A `BooleanStateCompositionRule` will always have one `childrenState`
- A `BooleanStateCompositionRule` will always have one `resultingState`

Cross-package properties, however, are less trivial as evaluating them requires taking into account the composition of this package with the core and FSM packages. Such properties are:

- Let r be a `BooleanStateCompositionRule` associated to a type t. Let O_t be the set of objects of type t. Let T_c be the set of types of the children of the objects in O_t. The `childrenState` of rule r belongs to one of the FSMs associated to the types in T_c. Also, the `resultingState` of rule r belongs to the FSM associated to t. This is easy enough to express as an OCL constraint, but it is not expressible unless we compose

this package with the core and FSM ones. This property is expressed with the OCL formula 2 on page 83.

- A state change in a child object matching a state composition rule should put the parent object in state `resultingState`. This can be checked by model checking.

5.2.4 The Property package

The Property package is part of the System model. Its metamodel is shown in Figure 5.9. It defines the *Property* class, associated to a type. All objects of that type will have the property. The property has a data type. It also can have alarms defining upper and lower value limits associated to diagnostic alerts (`OK`, `warning`, `alert` and `severe`). Alarms can be simple (they disappear when the alarm condition is gone) or acknowledged (they stay visible even after the alarm condition is gone, until the user acknowledges them). An alarm also has a priority level, defining which alarm(s) should be displayed if the current conditions enable multiple simultaneous alarms (e.g., in the case of overlapping value intervals).

For the Power Groups in the Cosmic Rack example, we define the `temperature` property, an integer value. For each of the four diagnostic values, we define upper and lower temperature limits. Figure 5.10 shows the Cospel editor defining the `temperature` property. One of its alarms is selected, showing that a warning will be produced if the temperature value is between -30 and -20.

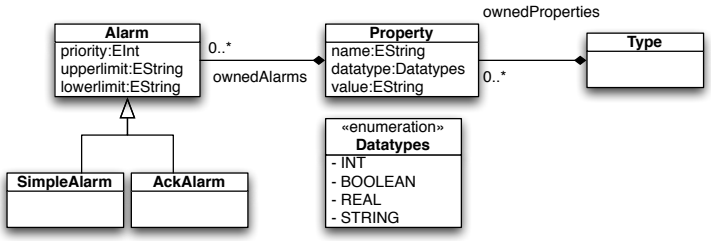

Figure 5.9. Property package metamodel

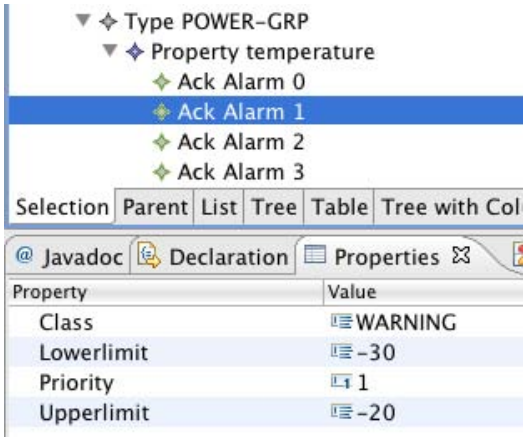

Figure 5.10. Cospel editor with property package elements

Properties of the property package

Apart from the structural properties expressed by the relationships making it up, the property package has only the following property:

- Value intervals of alarms can't overlap for the alarms associated to the same property. This is a well-formedness constraint that can only be verified at the model level (through constraint checking). The property is expressed by the OCL formula 3 on page 83.

5.2.5 The Command and Event package

The Command and Event package is part of the System model. The metamodel in Figure 5.11 describes events.

An `AbstractEvent` is associated to a `Type`, meaning that all objects of that type will have it. It can be either a simple `Event`, or a `Sequence` of events. The definition of `Sequence` is recursive: the `firstEvent` and `secondEvent` of the sequence may be `Sequence`s in their turn. A simple event has a name and a list of parameters. A `Command` is a specialization of an event. Based on the satisfaction of a `Condition`, an event can be executed, and can optionally trigger another event. The event can also be associated (through the `triggerTransition` relationship, not shown in Figure 5.11) to a transition of the type's own FSM; this means when the event is executed, the transition is triggered. Conditions are characterized by ex-

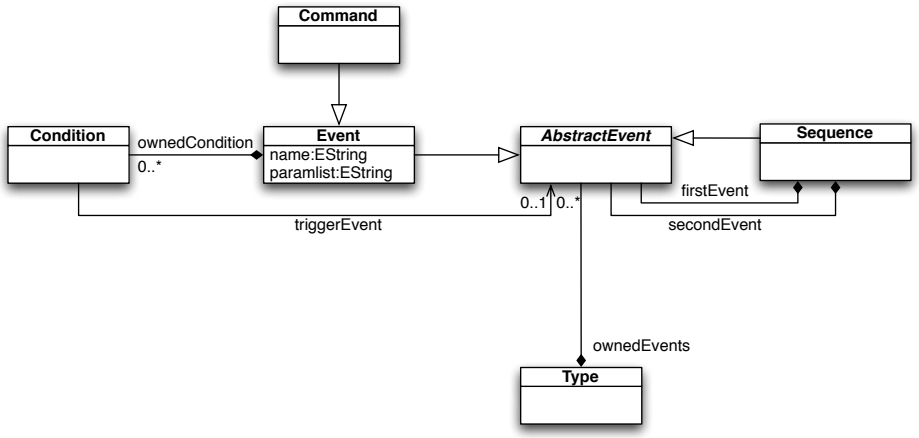

Figure 5.11. Event package metamodel

pressions, evaluated on properties and/or event parameters, and include the definition of pre-postcondition expressions. Multiple conditions can be used to axiomatize an event.

The Control Channels in our Cosmic Rack case study have an `ENABLE` command. This command has no parameters, thus its condition does not state any particular precondition for its execution. However, `ENABLE` is associated to the `DISABLED-TO-OFF` transition of the `FSMControlChannel` FSM, going from state `DISABLED` to state `OFF` (see Figure 3.5). This implies that the command is only available if the control channel is in state `DISABLED` (otherwise the transition would not be fireable). This is shown in Figure 5.12.

Properties of the event package

Apart from the structural properties expressed by the relationships making it up, the event package has the following cross-package property:

- Let e be an event with a `triggerTransition` relationship. Let t be the type associated to e. The transition referenced by `e.triggerTransition` must belong to the FSM associated to t.

- An event should only be raised when the associated condition is satisfied.

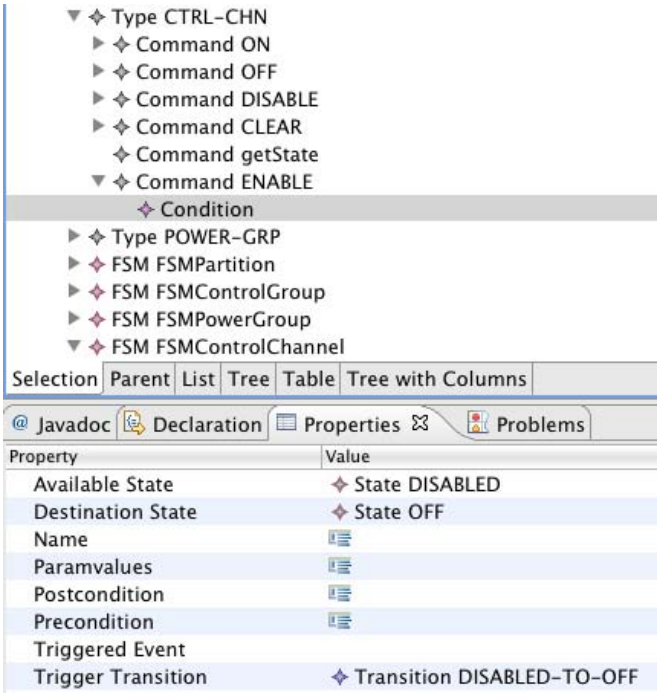

Figure 5.12. Cospel editor with event package elements

- If the `triggerEvent` of an event e points to an `Event` (or `Sequence`, or `Command`) $e2$, then $e2$ should be triggered if e is raised (as long as $e2$'s condition is satisfiable).

5.2.6 The Geometry package

The Geometry package corresponds to the visual model. A `Geometry` class (abstract, that generalizes several classes like `Box`, `Cylinder`, `GeomFile`...) is associated to a `Type`, defining the shape of all objects of that type. A `Scale` class is also associated to a `Type`, allowing the reuse of the same geometry at different scales for different types (e.g., one might have two types of screws, identical but for the length). Classes defining position in space and rotation are directly associated to the object, placing the object in space. For the position in space various possibilities exist: giving absolute coordinates, or placing the object with relationships to other objects (parallel surfaces, distance...). The metamodel of this package is in Figure 5.13, with a simplification on the *relationship* class (which generalizes a number

of possible relationships). Note that having a geometry/position is optional; this is because we could make models in which there are some objects which are only *logical* objects. They do not correspond to a physical object, but are only used to group other objects for diagnostic purposes.

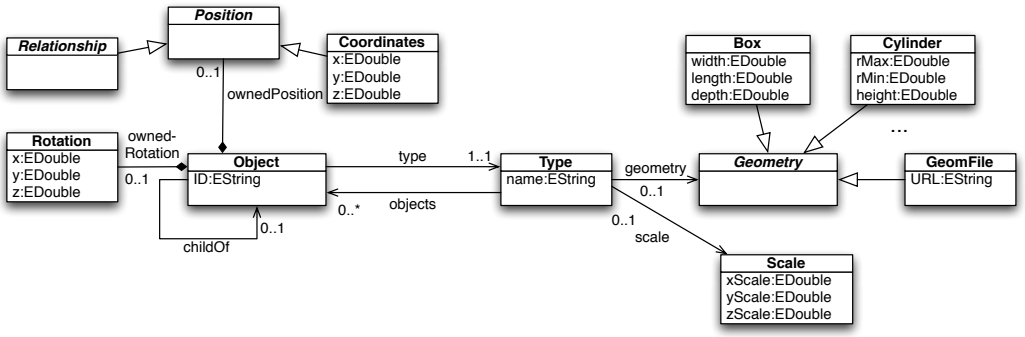

Figure 5.13. Geometry package metamodel

In the Cosmic Rack, all components are cuboids (although with different sizes, proportions and orientations). To model their geometry, we define a `Box` with unitary dimensions and associate it to all types. Then we associate a different `Scale` to each of the four types, giving the size in the three axes of the box. Also, all Power Groups with a name ending in "Rod_1" (i.e., every other Power Group) have a rotation on the X axis, so we define a `Rotation` and apply it to all those objects. Thus, to characterize all shapes, we only define one geometry, one rotation and four scales. We then position each object in space according to the mechanical drawings of the Cosmic Rack. Figure 5.14 shows these definitions. The `Shape` element is selected, showing the associated file (*cube.obj*) and types.

Properties of the geometry package

Apart from the structural properties expressed by the relationships making it up, the geometry package has the following property:

- If a type has a geometry, the associated objects must have a position. This is expressed by the OCL formula 4 on page 83.

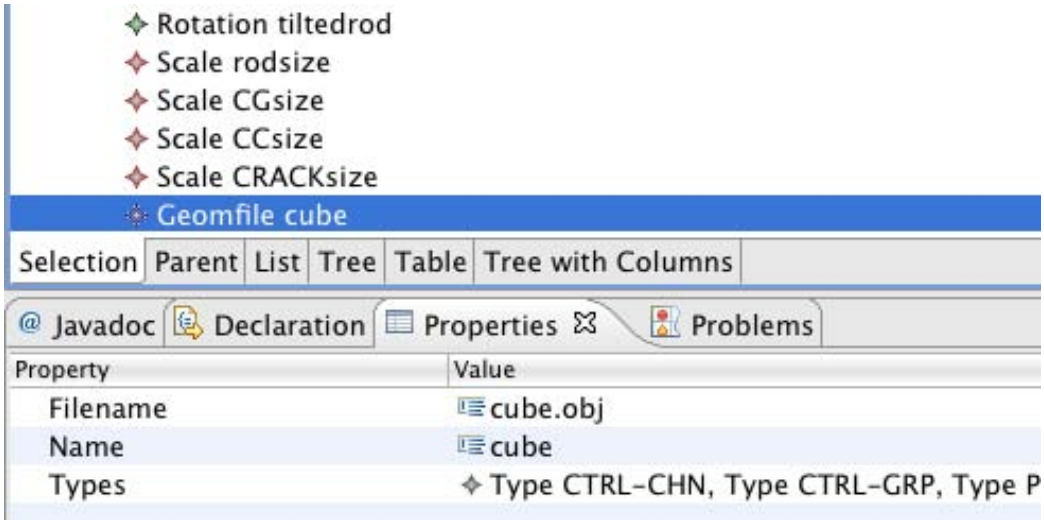

Figure 5.14. Cospel editor with geometry package elements

5.2.7 The Tasks package

The Tasks package corresponds to the Task model. Building the metamodel for the CTT formalism, we chose to use binary task trees to avoid the problem of defining a priority for the temporal operators, as suggested in (Paternò et al., 1997). The resulting metamodel is shown in Figure 5.15.

Tasks in our approach are a way to organize commands into more complex, structured semantic units. Tasks for the CTT leaves are commands associated to objects. Higher-level tasks however are not necessarily associated to a command and may simply represent a mental way for the user to organize elementary commands into a larger procedure. Figure 5.16 (on page 81) showed such a task: the leaves are actual commands on control channels, while the nodes on the first two levels of the task tree only represent the sequential organization for these tasks.

All `Tasks` are defined at the level of `Types`. Defining a task for a type means that it will be available for every object of that type. An example of a Type-level task could be turning on a single control channel. The subtrees of Figure 4.8 show that for every control channel, activating means enabling the channel and then turning it on. A task can be associated with a `Command` from the Event package, which means that performing the task involves sending that command to the objects associated with the task. Finally, a task

is associated to a `Profile`, a concept described in the user model in the following paragraph, meaning that only certain profiles are allowed to perform that task. Tasks are related in a tree structure by `Operator`s (here we show only three of them: `Interleaving`, `Choice` and `Enabling`).

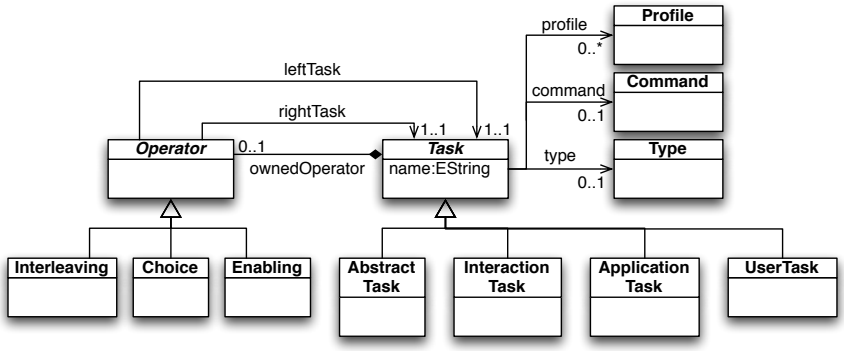

Figure 5.15. Task package metamodel

Type-level tasks make specification quicker, as for example one does not have to specify the turning on task for every control channel. The drawback is that they can't model fine grain tasks for which one might want to specify which exact objects are involved. For example the top-level task of Figure 4.8 specifies that the control channel 1 and 2 have to be activated. This can be accomplished by specifying which object a subtask applies to through the use of the operators' properties.

Figure 5.16 shows how the *Activate All Control Channels* task of Figure 4.8 is modeled using the Cospel editor. We started by creating the leaves, the `EnableCC` and `TurnOnCC` interaction tasks, associating them to the `CTRL-CHN` type, and to its `ENABLE` and `ON` commands respectively. We then created the `ActivateCC` abstract task which links the two interaction tasks in a sequence with the interleaving operator. Finally we modeled the top level `ActivateAllControlChannels` abstract task, which applies `ActivateCC` on an object-by-object basis to each control channel in the specification.

Note that the specification of tasks has been included in the language mainly for documentation purposes; however, for lack of resources the GUI engine (described in chapter 7) does not implement tasks.

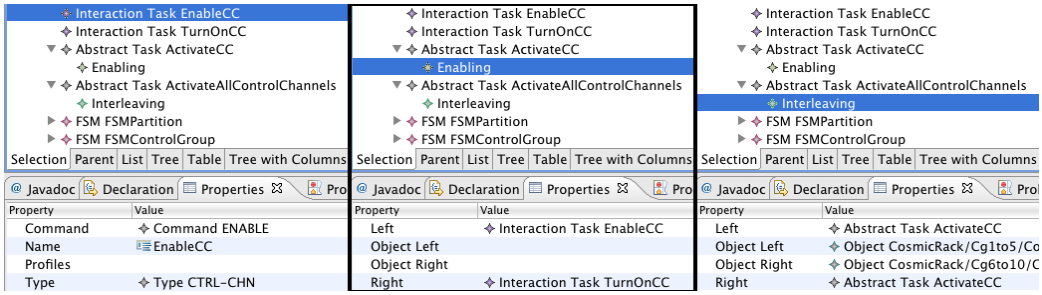

Figure 5.16. Cospel editor with elements of the Task package

Properties of the task package

The task package only has structural properties which are explicit in the metamodel, expressed by the metaclasses' relationships.

5.2.8 The User Package

The User Package corresponds to the user model. The metamodel corresponds to what has already been shown in Figure 4.5, with the *Thing* class replaced by the `Task` class from the task package. We also renamed *Behaviour* to *Profile* for clarity. The metamodel is shown in Figure 5.17.

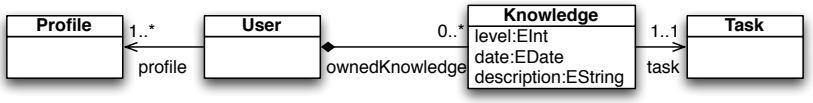

Figure 5.17. User package metamodel

Figure 5.18 shows how to define the example of Figure 4.6 (page 56) in the Cospel editor: we create a `Profile` called `Physicist`, and a `User` named `Peter Piper` associated to this profile. We associate Peter to the task model for the `ActivateAllControlChannels` task through an instance of `Knowledge`, which is characterized by a `level` of 20, and a `date` of 22.10.2006.

Note that, like tasks, user information has been included in the language for documentation purposes but not implemented in the GUI engine for lack of resources.

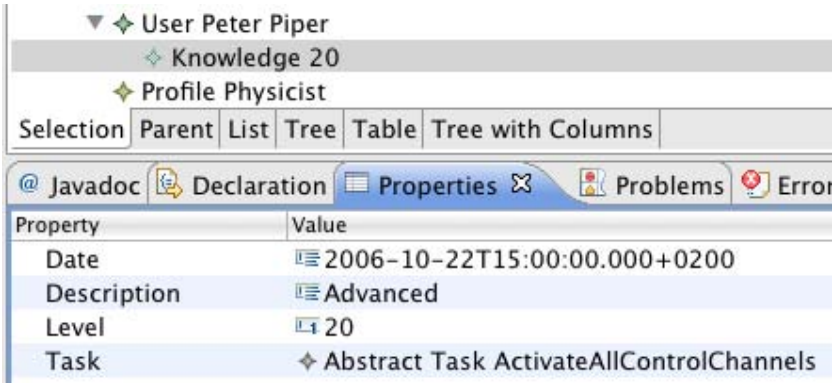

Figure 5.18. Cospel editor with elements of the User package

Properties of the user package

Apart from the structural properties expressed by the relationships making it up, the user package has the following property:

- A given user might have at most one knowledge related to the same task

5.2.9 OCL Constraints

Some of the properties enunciated for the various Cospel packages in the previous sections reference OCL constraints contained in this section. These constraints formally express those properties and can be used to validate models.

Property 1

The `fromState` and `toState` of a transition belong to the same FSM as the transition itself (property of the FSM package on page 71).

OCL formula 1

```
context Transition inv:
self.toState.FSM = self.fromState.FSM and
self.toState.FSM = self.FSM
```

Property 2

Let r be a `BooleanStateCompositionRule` associated to a type t. Let

O_t be the set of objects of type t. Let T_c be the set of types of the children of the objects in O_t. The `childrenState` of rule r belongs to one of the FSMs associated to the types in T_c. Also, the `resultingState` of rule r belongs to the FSM associated to t (property of the State composition rule package, page 73).

OCL formula 2

```
context BooleanStateCompositionRule inv::
Object.allInstances
   -> select (o | o.childOf.Type=self.Type)
   -> collect(o | o.Type)
   -> includes(self.childrenState.FSM.Type);
context BooleanStateCompositionRule inv::
self.resultingState.FSM.Type=self.Type;
```

Property 3

Value intervals of alarms can't overlap for the alarms associated to the same property (property of the Property package, page 75).

OCL formula 3

```
context Property inv:
self.ownedAlarms->forall(a | (
   self.ownedAlarms->excluding(a)->not exists(a2 |
   (
      (a2.upperlimit >= a.lowerlimit and
       a2.lowerlimit<=a.lowerlimit)
      or
      (a2.upperlimit >= a.upperlimit and
       a2.lowerlimit<=a.upperlimit)
   )
)))
```

Property 4

If a type has a geometry, the associated objects must have a position (property of the Geometry package, on page 78).

OCL formula 4

```
context Type inv:
(not(self.geometry.oclIsUndefined())) implies
self.objects ->
     forall(o | not(o.ownedPosition.oclIsUndefined()))
```

5.3 Summary and conclusion

This chapter described the Cospel language in detail. We explained how it is composed of several packages, which are composed to form the whole language. We saw each package's abstract syntax (defined by metamodels) in detail, and gave examples of use in relation to our case study.

The next chapter focuses on how Cospel is transformed into other formalisms, in order to provide its semantics and to generate the artifacts allowing GUI prototyping.

Chapter 6

Transformation

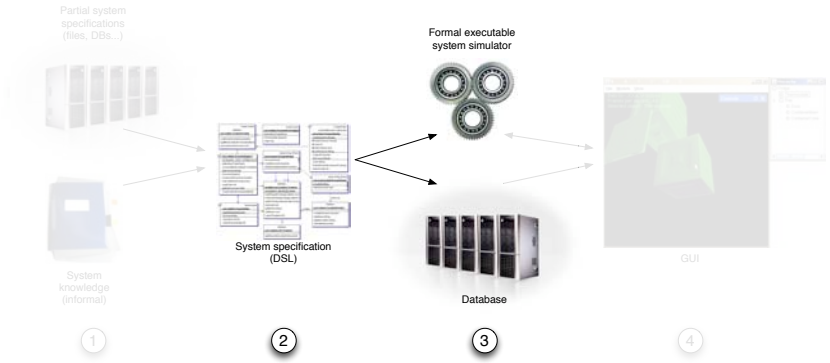

Figure 6.1. Transformation step of the specification

Transformation in the context of the BATIC^{3}S project has two purposes: giving semantics to Cospel models and producing deliverables. It represents the passage from step 2 to step 3 of the BATIC^{3}S process in Figure 6.1.

Instead of defining semantics for each element of the Cospel language, our approach establishes mapping rules between the Cospel metamodel and the CO-OPN (Biberstein, 1997) metamodel. CO-OPN is an object-oriented modeling language based on algebraic Petri nets, allowing the execution of specifications and providing tools for simulation, verification and test generation. CO-OPN supports concurrency and transactionality, and there are tools to generate executable Java code from a CO-OPN model. Since the semantics of CO-OPN are already defined (in formal terms), we obtain the Cospel semantics as a result of the transformation.

We chose to use the ATLAS Transformation Language (ATL) framework (ATLAS Group, 2008), a declarative, rule-based language and framework for specifying mapping rules in language transformations. ATL is particularly well suited for its declarative style and its modularity. The ATL framework runs as an Eclipse plugin, another advantage for our methodology which is mainly based on Eclipse-related tools.

We followed a modular approach, identifying the different packages of the Cospel language (e.g.,tasks, users, object hierarchy...) as sources for the transformation; for each module, we gave a transformation pattern with ATL rules. The patterns have then been composed, with syntactic and semantic composition techniques, to obtain a set of transformation rules able to transform whole Cospel models.

The next two sections (6.1 and 6.2) will describe CO-OPN's metamodel and give an overview of ATL. The purpose is providing a reference for the elements involved in Cospel's transformations. However, it is still possible to intuitively understand those transformations without knowing the deep details of CO-OPN and ATL semantics. A reader who is only interested in the methodological aspects would probably be better off skipping these sections and go straight to section 6.3.

6.1 CO-OPN's metamodel: the 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 combines Petri nets, object-orientation and abstract algebraic data types to provide a framework for the design and specification of concurrent and distributed systems.

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

a) the generation of a prototype for a given specification;

b) the enrichment of prototypes with user-written code;

c) its execution and the verification of implementations through testing;

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

In this sense CO-OPN is 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 Cospel concepts transformation. Among the most relevant are:

- It is a 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 of three types of modules: *ADT*, *Classes*, and *Contexts*:

- *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. Figure 6.2 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.

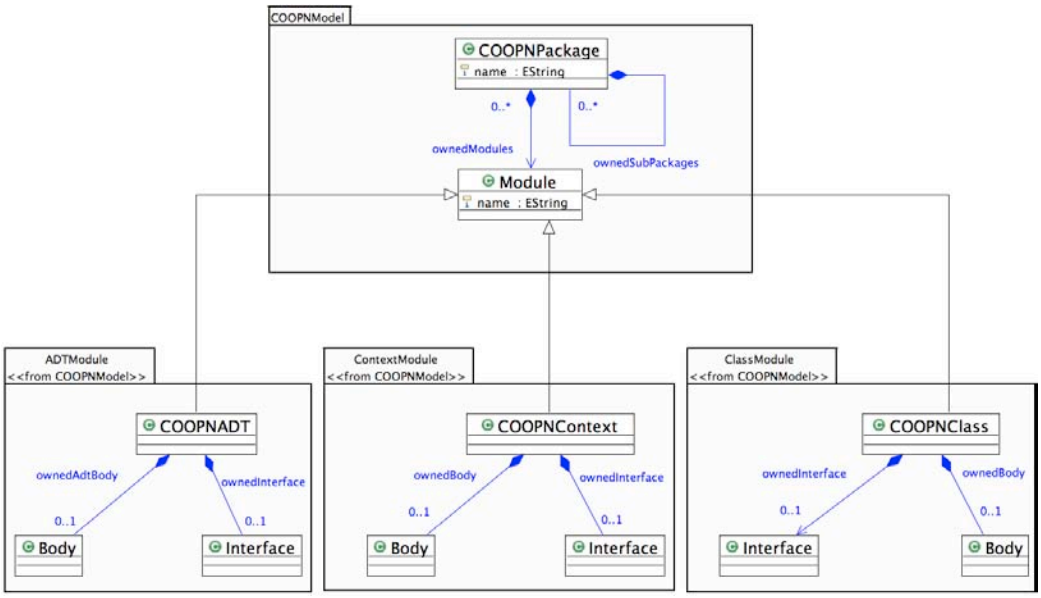

Figure 6.2. CO-OPN metamodel: focus on CO-OPN modules

Figure 6.2 presents the root part of the CO-OPN metamodel, which is the base of a CO-OPN specification.

COOPNPackage represents the CO-OPN element called *Package*. A `COOPNPackage` contains zero or more `subPackages` (through the `ownedSubPackages` composition). A Package can be defined as a set of related modeling 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 modeling languages. `COOPNPackage` is the outermost modeling element of the CO-OPN metamodel and contains zero or more modeling elements called `Module`s (through the `ownedModules` composition).

Module is an abstract CO-OPN element . It is concretized by three different type of modules:

a) Algebraic Abstract Data Type (ADT)s

b) Classes

c) Contexts

The following will describe each of the CO-OPN modules. Fragments of the modules' metamodels will be used in order to provide essential information regarding CO-OPN. This information will be later used in order to understand the Cospel-to-CO-OPN transformation.

a) Algebraic Abstract Data Type Module

The 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 constructs, `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.

Figure 6.3 shows the part of the CO-OPN metamodel that defines the ADT focusing on its `Interface` section.

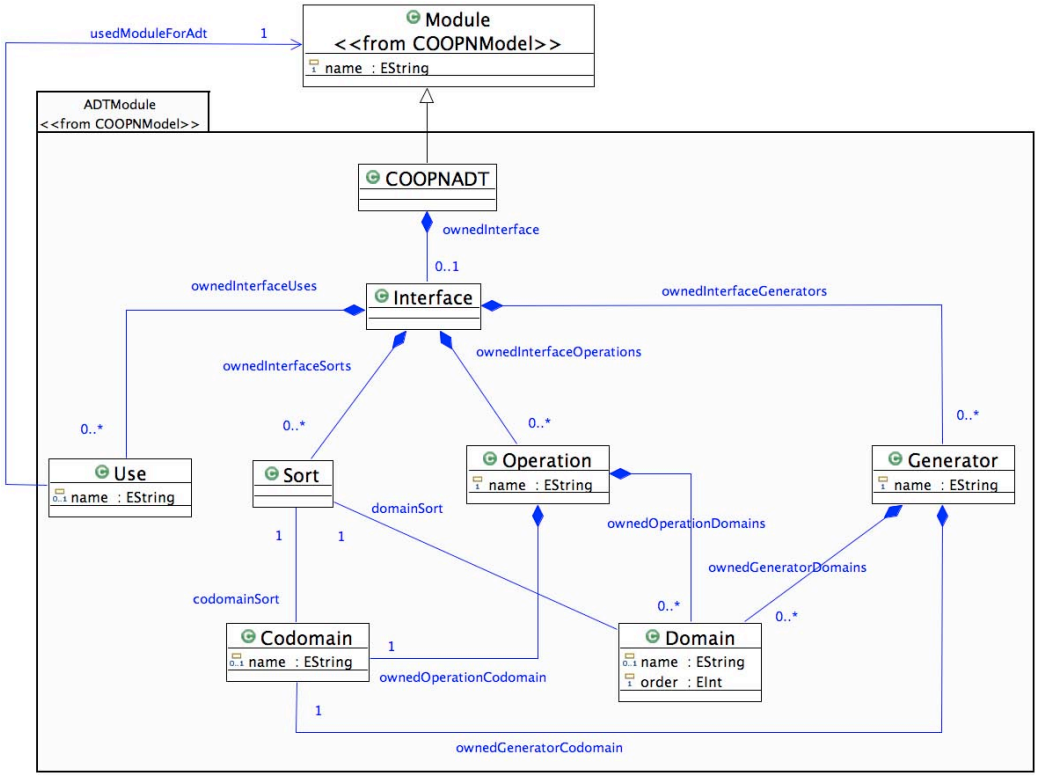

Figure 6.3. CO-OPN metamodel: focus on the ADT Interface

Interface The interface section specifies module components that are going to be accessible by other modules. An ADT interface aggregates zero or more `Use`, `Operation`, `Sort` and `Generator`;

Use represents a reference to another CO-OPN module. It is represented by the `Use EClass` element. A `usedModuleForAdt` association, not shown in the ADT metamodel excerpt for readability reasons, 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 a `sortName` attribute, representing the type name;

Operation defines the operations that can be performed on the data type under specification. An `Operation` is characterized by a `CoDomain` and zero or more `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 characterized by a `name` attribute. Since `Codomain` is an ADT type it also has a mandatory reference (`codomainSort` in the metamodel) to the `Sort EClass` element;

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 it will figure in the operation signature. Similarly to 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 zero or more `Domain`s.

The interface section of an ADT basically specifies the ADT's signature. Its semantics is defined in the `Body` section. Figure 6.4 depicts a subset of the ADT `Body` metamodel section focused on the definition of the ADT axioms.

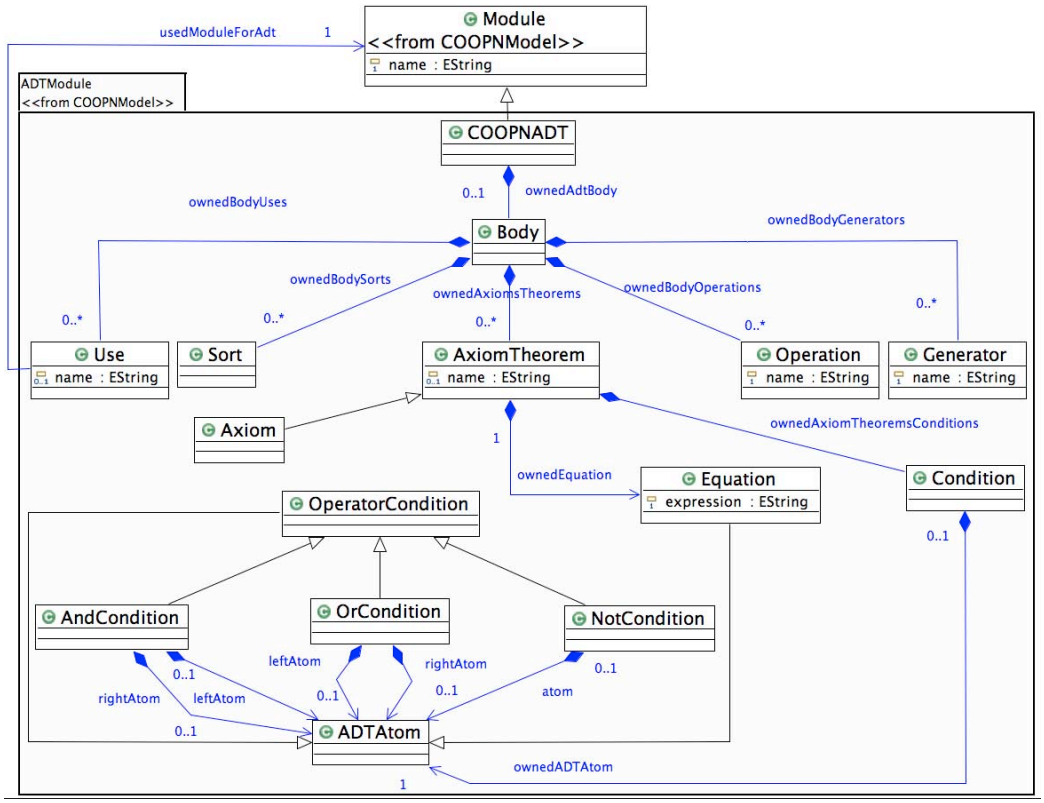

Figure 6.4. CO-OPN metamodel: focus on ADT Body and Axioms Definition

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

Sort specifies set of sorts that can be defined for ADT's 'internal' use. A `Sort` is identified by `sortName` attribute representing the type's 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, is the function that builds all the possible values of the ADT's sort. They are of private use of 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 specialization of the `OperatorCondition` meta-class which, in turn, specializes the `ADTAtom` meta-class. All this setup of meta-classes that specialize and are specialized define a recursive structure that allow to specify conditions of the type $(var1 = var2) = true \ \& \ (var3 > 0) = false$.

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

b) 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 components that are accessible by other modules. Figure 6.5 highlights some of the components that define a CO-OPN `Class` module. This figure focuses on its `Interface` part.

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

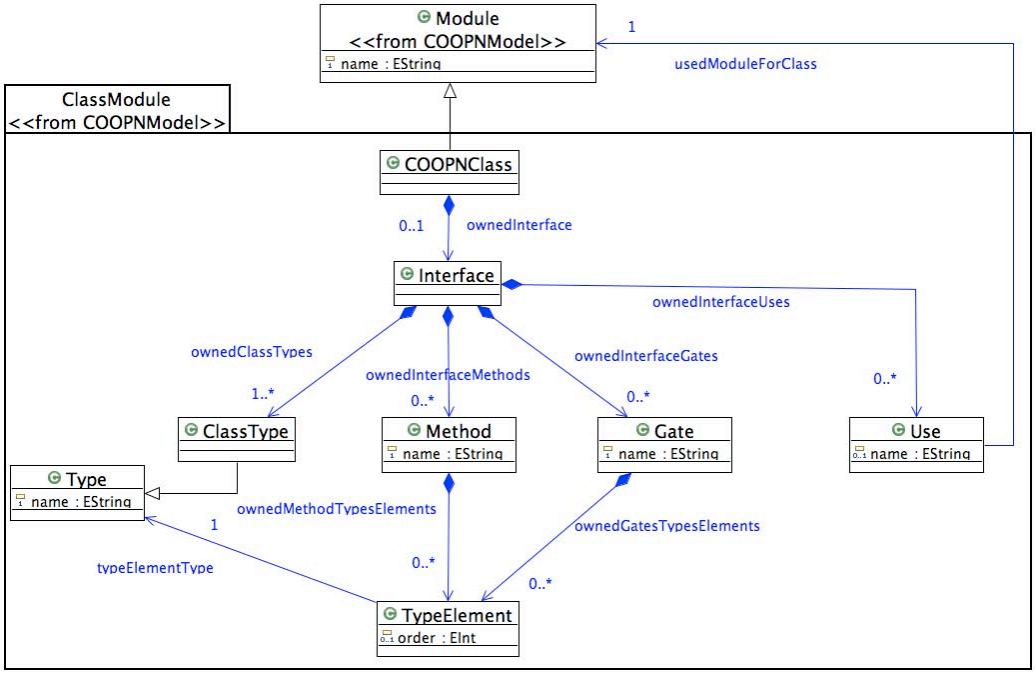

Figure 6.5. CO-OPN metamodel: focus on Class Interface

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

Interface A Class interface contains zero or more `Use`, `Gate` and `Method` represented respectively by the `ownedInterfaceUses`, `ownedInterfaceGates` and `ownedInterfaceMethods`. It also contains one ore more `ClassType`;

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

Use allows to create a reference to another module. This reference is represented 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` attribute serves as a unique identifier and also represents the signature

of the method. A CO-OPN `Class` method is composed of zero or more 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 `CO-OPN`, a method parameter is designated by an underscore sign ("_") that can be present in any place of the method's name. As an example, a method $sum(\ \_\ \_\ )$ that sums two parameters can also be written as $\_\ sum\ \_$ or $\_\ \_ sum$. 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 represents an outgoing event, in other words a requested service. Gates and methods are complementary (requesting a service generally involves calling a method and receiving a gate event) and have the same syntactical structure;

The other major section of a CO-OPN Class is its `Body`. The most important characteristic of the CO-OPN Class body section is that it encapsulates the underlying Petri net and axiom 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 Figure 6.6 we present a simplified metamodel of the body section of a CO-OPN Class.

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, but it is not accessible by other modules;

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

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

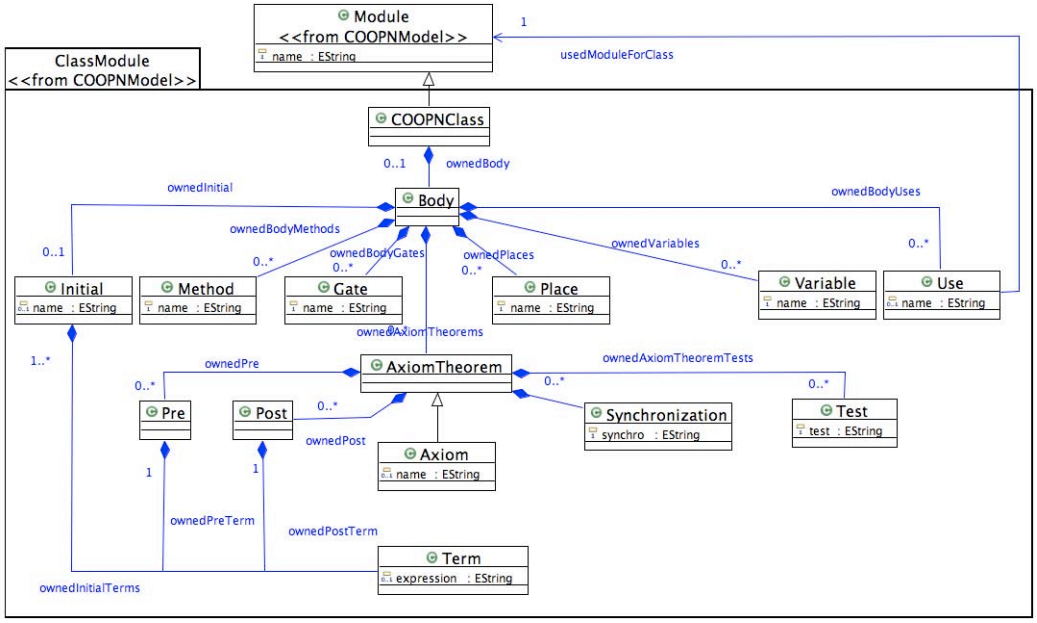

Figure 6.6. CO-OPN metamodel: focus on Class Body ***without*** *Axiom definitions*

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 is true for places, 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 initialize 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 initialized to the values defined in the `Initial` field;

Axiom As for ADTss, a class body is also composed by axioms and theorems. Again, they define the behaviour of the class. An axiom is composed of zero or more preconditions, postconditions, synchronizations, 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 Figure 6.7;

Pre The precondition that must be satisfied by the object's state when an axiom is evaluated;

Post The postcondition that must be satisfied by the object's state 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.

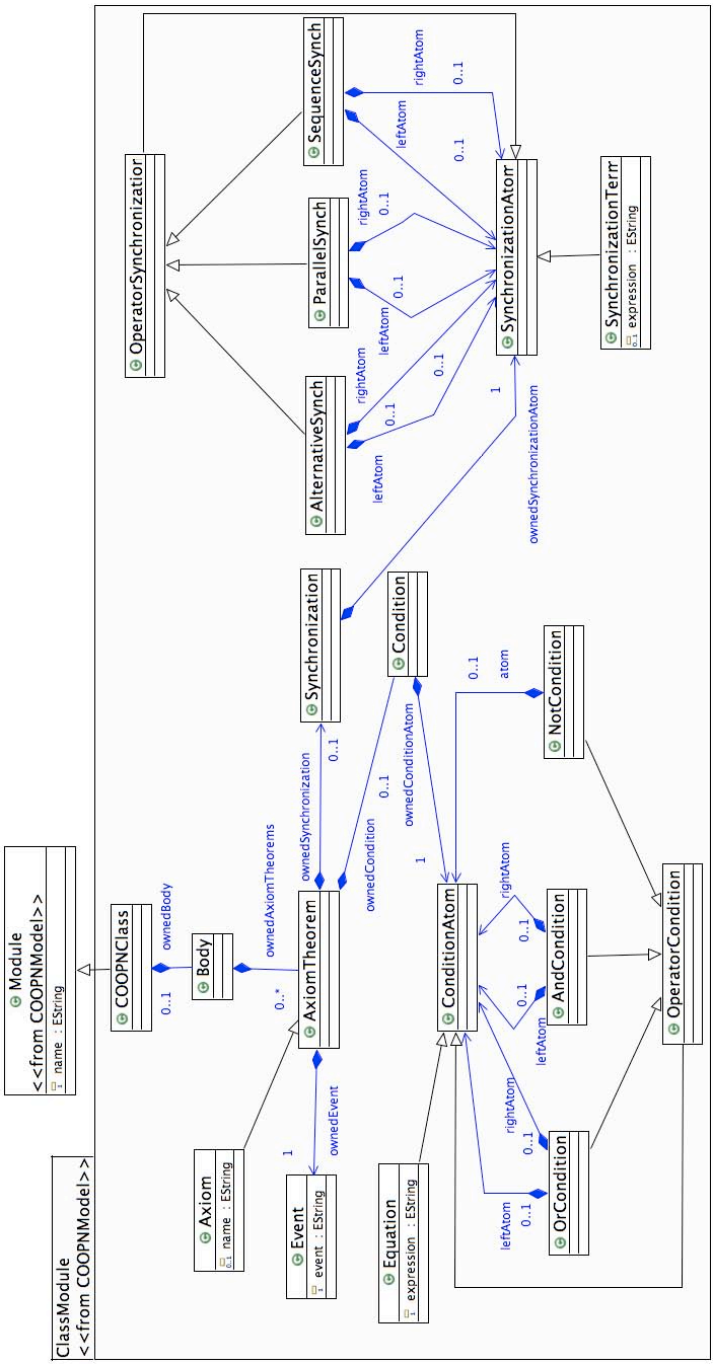

Figure 6.7. CO-OPN metamodel: focus on Class Body Axiom definition

Having described the generic aspects of CO-OPN body structure we proceed by providing a more detailed characterization of some aspects of it. Figure 6.7 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 under 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 specialized by the `Equation` and `OperatorCondition` meta-classes. The `OperatorCondition` is specialized by the `OrCondition`, `AndCondition` and `NotCondition` meta-classes, which are in turn composed by other condition atoms. This allows to have conditions as a composition of statements like $(a > b) = true\ and\ (a > 0) = false$.

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 synchronization 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 either in the same class as the axiom, or in other classes. A synchronization is a composition of a synchronization atom (`SynchronizationAtom` meta-class in the metamodel) that is specialized by the `SynchronizationTerm` and `OperatorSynchronization` meta-classes. This last meta-class is specialized by the `AlternativeSynch`, `ParallelSynch` and `SequenceSynch` meta-classes. These meta-classes are in turn a composition of `SynchronizationAtom` elements and they represent the types of synchronization available in CO-OPN:

- Alternative is represented as "+" in CO-OPN's textual concrete syntax. It allows to express services 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's textual concrete syntax. It allows to call services in parallel. The services expressed with this operator will be executed simultaneously. E.g.

o.methodA // o.methodB means that *methodA* and *methodB* of object *o* are executed in parallel;

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

c) Context Module

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 contain tokens.

Contexts are coordination entities. They are responsible for coordinating activities among classes. Coordination is done by means of behavioral axioms that connect methods and gates of the embedded entities.

As the previous two CO-OPN type of modules, contexts con be separated in interface and body sections. The metamodel's interface section is presented in Figure 6.8. 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 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 organized 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;

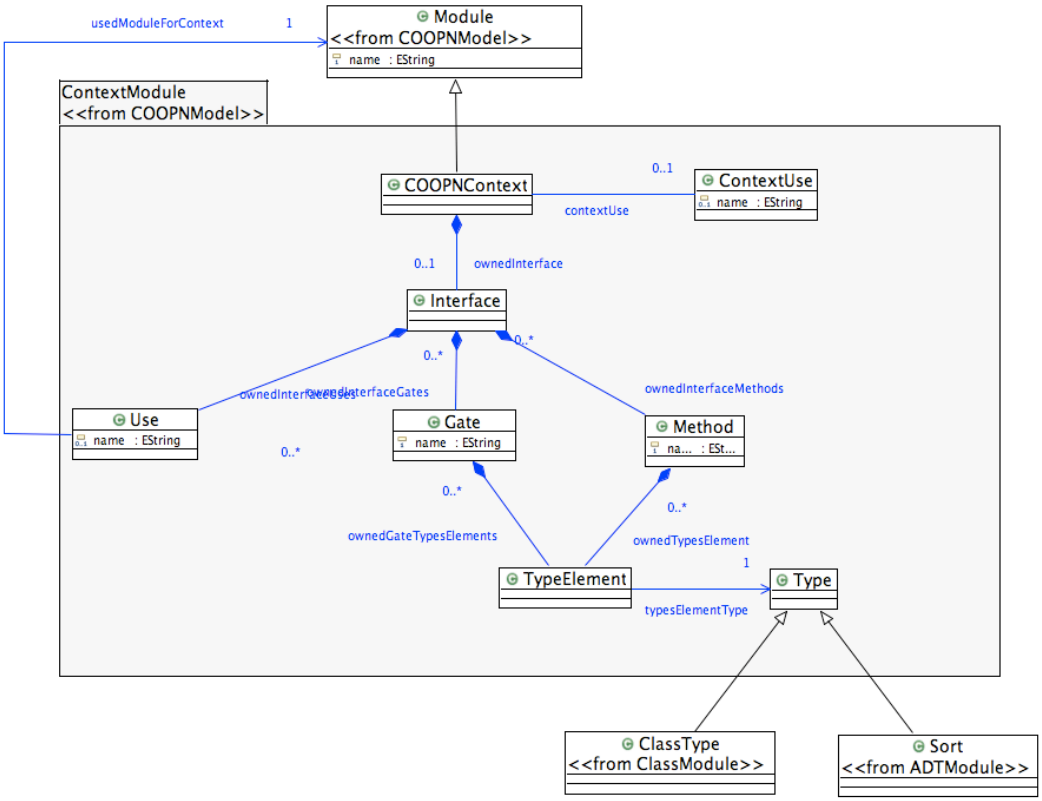

Figure 6.8. CO-OPN metamodel: focus on Context Interface

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 under which circumstances the axiom is going to be executed. A condition is a composition of equations possible separated by an operator. Operators can be 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 available synchronization operators. Its usage and semantics is similar to the one presented for CO-OPN classes in Sec. 6.1 on page 98. Again, the provided event is built with the sequential, parallel and alternative synchronization operators.

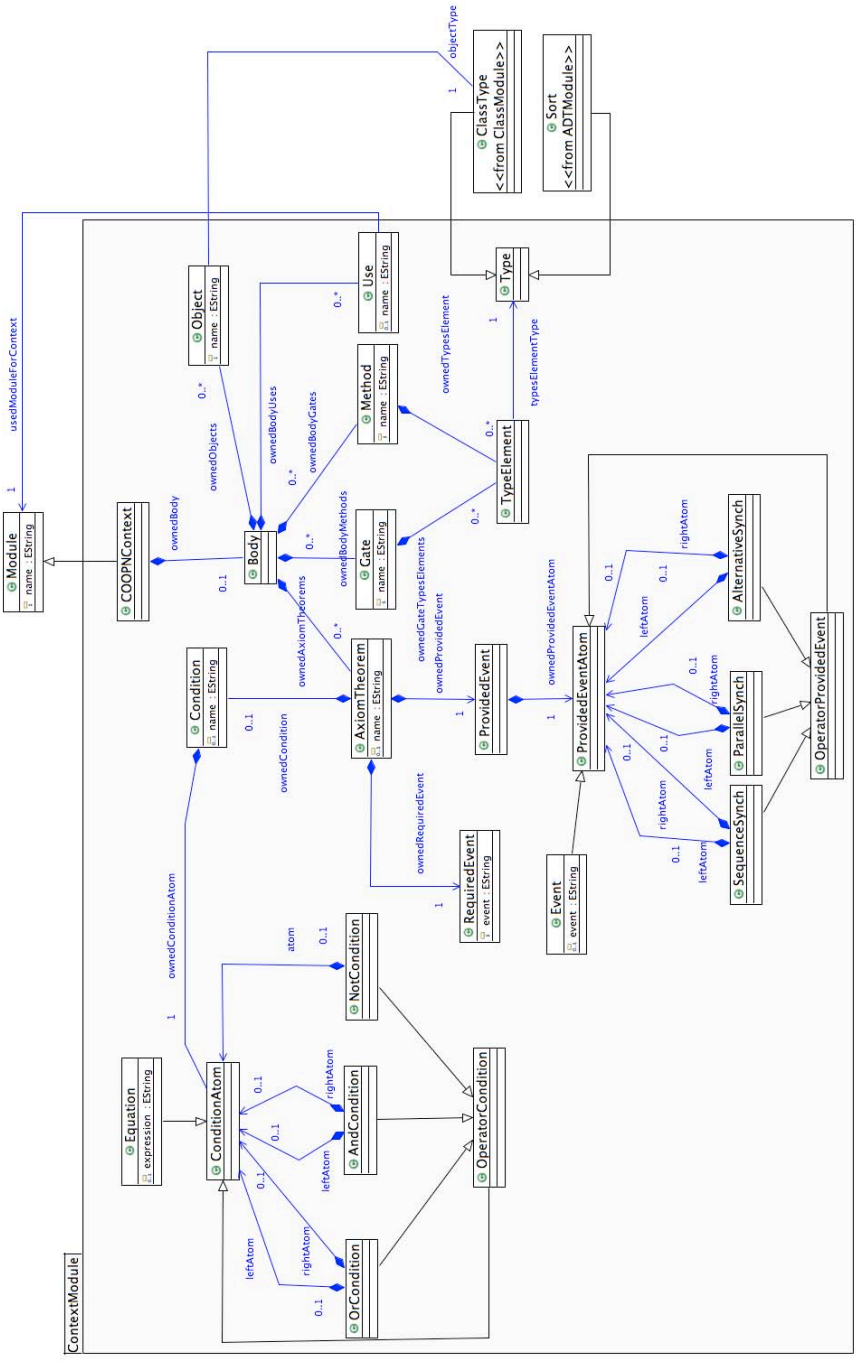

Figure 6.9. CO-OPN metamodel: focus on Context Body

6.2 ATL: the transformation framework

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.

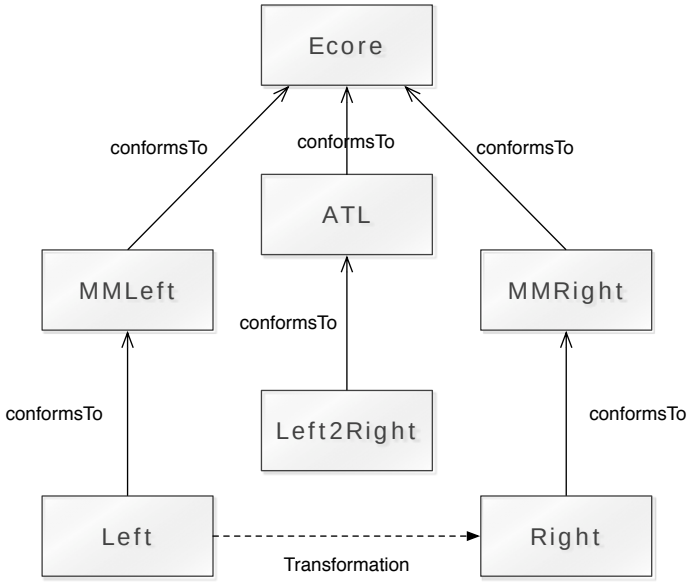

Figure 6.10. An overview of ATL Model Transformation

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 a response to the QVT RFP (Object Management Group, 2002). It is a model transformation language specified as both metamodel and textual concrete syntax with a hybrid of declarative and imperative programming approach. ATL mainly focuses on the 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.

Figure 6.10 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.

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

a) Header Section

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

Listing 6.1. ATL Header Definition

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

Listing 6.1 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 comma. The name of the declared models is used to identity them.

b) 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 6.2.

Listing 6.2. ATL Import Definition

```
uses extensionless_library_file_name;
```

c) 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 the helper is defined. This is similar to the way a method is defined in the context of a 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 6.3.

Listing 6.3. 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 supported programming modes: 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 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 6.4 provides the schema of an ATL rule.

Listing 6.4. ATL Rule Definition

```
rule rule_name {
from
in_var : in_type [(
condition
)]?
[using {
```

```
  var1 : var_type1 = init_exp1;
8 ...
  varn : var_typen = init_expn;
10 : = ; }]?
  to
12 out_var : out_type (
  bindings1
14 ),
  out_var : distinct out_type foreach(e in collection)(
16 bindings2
  ),
18 ...
  out_varn : out_typen (
20 bindingsn
  )
22 [do {
  statements
24 }]?
  }
```

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.

ATL Data Types

The ATL data type scheme is very similar to the one defined by OCL. Figure 6.11 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.

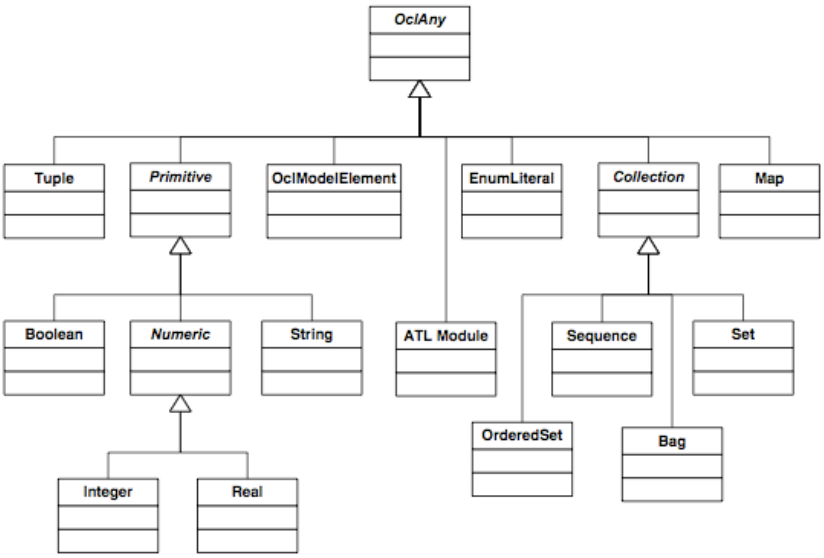

Figure 6.11. ATL Data Types metamodel

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' on collections is out of the scope of this document.

6.3 Different information for different results

As we said before, the methodology's outputs are twofold: a system simulator and a GUI. To produce them, the ATL transformation creates two models from the Cospel specification. The first one is a CO-OPN specification which represents the behavioural aspects of the system. This specification is then used to generate the system simulator. Since the simulation is purely behavioural, no visualization-related information is used for building the CO-OPN model.

The second model created by the ATL transformation is a database, which has to be used as a data source for creating the dynamic 3D scene in the GUI. Information including object hierarchy, geometrical data, FSMs (for representing states), tasks, properties and commands/events (to create interaction methods) are stored in this database. Figure 6.12 shows which Cospel packages contribute to which model.

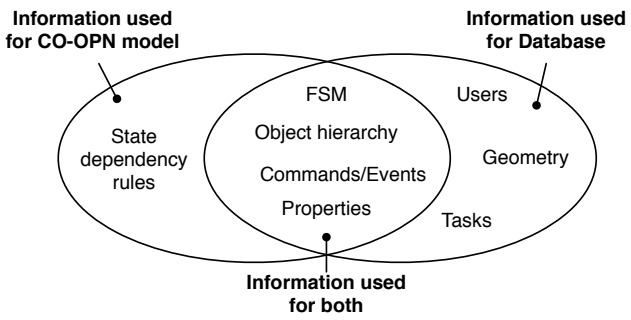

Figure 6.12. The CO-OPN model and the database are built by transformation of Cospel packages; some packages are used to produce both models, while others only contribute to one of them

An overview of the transformation process is given in Figure 6.13, showing which are the players in the transformation of the Cospel model to a CO-OPN+Database model. It is shown how the individual language packages and their transformation patterns are composed into a global metamodel and a global transformation. A detailed description of this composition and transformation process is in (Pedro, Risoldi, Buchs, Barroca, & Amaral, 2008).

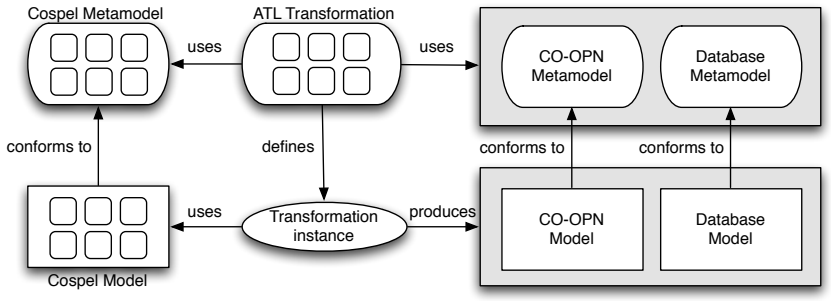

Figure 6.13. Transformation overview. Smaller rectangles inside Cospel metamodel and model and inside the ATL transformation represent individual language packages and their individual transformation patterns

6.4 The system simulator

Generating the system simulator starts with the Cospel transformation creating a CO-OPN specification. We will now discuss how elements of each Cospel package are transformed into corresponding CO-OPN elements. For the sake of readability, we will not go into deep details of the generated CO-OPN code or the ATL code that implements the transformation. We will rather show which parts of the Cospel metamodel are transformed into which parts of the CO-OPN metamodel by use of diagrams. We will show how single meta-classes and relationships of the Cospel metamodel are transformed into CO-OPN metaclasses and relationships. The diagrams will also contain grayed-out relationships that will show how the different CO-OPN metaclasses resulting from the transformation are related.

6.4.1 Cospel core package transformation

Figure 6.14 illustrates the transformation of the Cospel core package elements. A Cospel `Specification` is transformed into a CO-OPN `Package` of the same name (Figure 6.14 - a). A Cospel `Type`, which serves as a template for objects in the system, is transformed into a CO-OPN `Class` contained in the package (Figure 6.14 - b). Cospel `Objects`, which instantiate types, are transformed into CO-OPN `Contexts` in the package, which instantiate a CO-OPN `Object` of the corresponding class. The `childOf` hierarchical relationship among Cospel objects is translated to a hierarchy of contexts through the `Use` meta-class (Figure 6.14 - c).

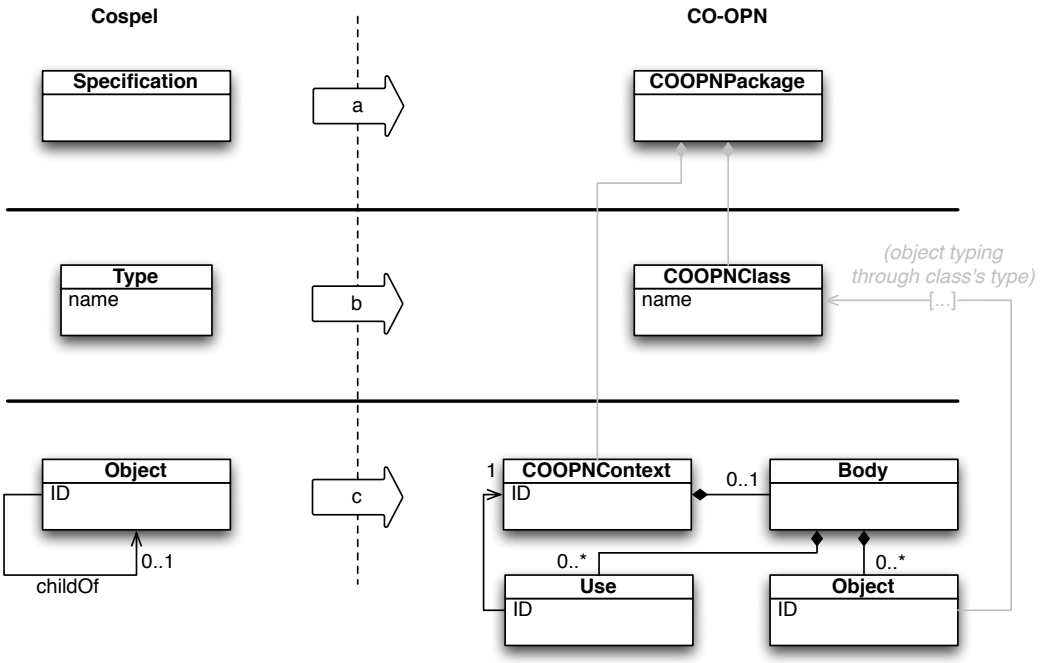

Figure 6.14. Transformation of the Cospel core package

6.4.2 FSM package transformation

Figure 6.15 illustrates the transformation of the Cospel FSM package elements. The `FSM` Cospel meta-class itself is not transformed. However, its relationship to the Cospel `Type` is used to identify which CO-OPN class should implement the FSM's behaviour. FSM `States` are transformed into `Places` in the respective CO-OPN class (Figure 6.15 - a). These are "black token" places. For example, an `OK` state will be translated to a place defined as `OK _ : blackToken`. An initial state will be initialized with a token.

FSM `Transitions` are translated to CO-OPN `AxiomTheorems` which implement the transition's semantics (Figure 6.15 - b). A transition `OK_TO_ERROR`, which goes from the `OK` state to the `Error` state, will be transformed into the following axiom: `OK_TO_ERROR::OK @ -> ERROR @;`.

6.4.3 State rules package transformation

Figure 6.16 illustrates the transformation of the Cospel State rules package elements. A Cospel `BooleanStateCompositionRule` is transformed to a se-

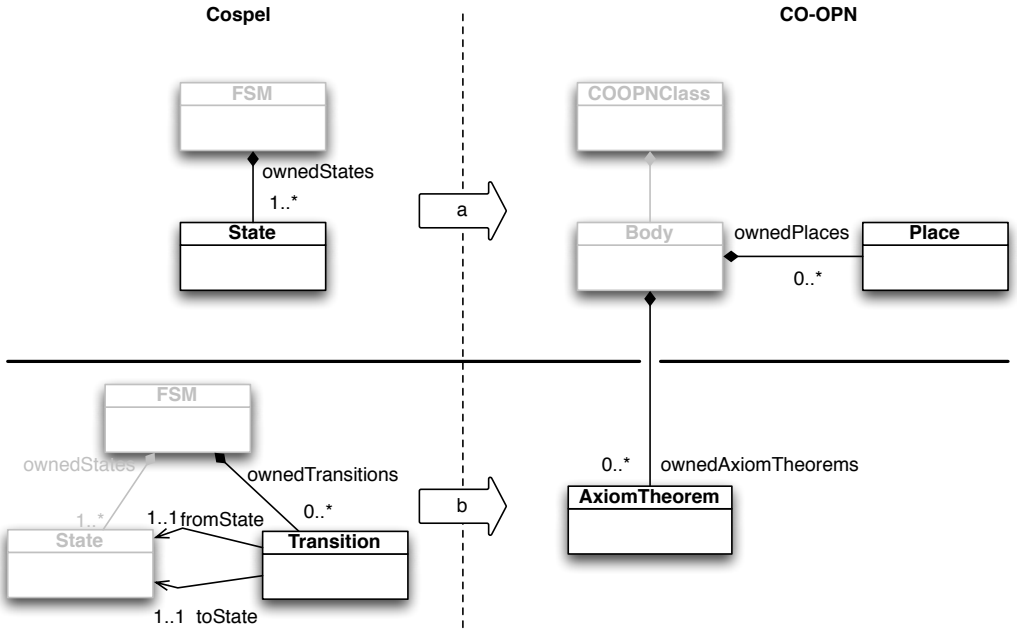

Figure 6.15. Transformation of the Cospel FSM package

ries of CO-OPN `Axioms` and `Places`. The transformation is a bit complex. A place is created for each child object which is used in the rule. These places are updated when children change their state, and are used to compute the resulting new state of the parent object. This task is done by an `Arecalculatestate` axiom, which implements the state composition rule(s) logic. For example, a rule might state that if children `c1` **and** `c2` are in state `ERROR`, then the object should go to state `ERROR`. Transforming this rule creates two places `c1` and `c2`, of type `statesort` (an ad-hoc ADT enumerating the possible states for an object). Then, an axiom is created implementing the rule:

```
this=Self & (s1 = ERROR & s2 = ERROR) =>
Arecalculatestate with this.OK_to_ERROR::c1 s1,c2 s2->c1 s1,c2 s2
```

To explain the axiom, it is easier to read it from the end. This axiom does not change the places storing the children states, as it has identical pre- and post-conditions:

```
c1 s1, c2 s2 -> c1 s1, c2 s2
```

Thus, `s1` (resp. `s2`) represents the contents of place `c1` (resp. `c2`) before and after the axiom is evaluated. Using this values, it is possible to evaluate the axiom's condition: both values must be equal to `ERROR`:

```
this=Self & (s1 = ERROR & s2 = ERROR)
```

On the satisfaction of this condition, the axiom can be evaluated to call the `OK_to_ERROR` transition:

```
Arecalculatestate with this.OK_to_ERROR
```

The axiom is triggered by the event of a child object changing state. This is implemented through a synchronization among the CO-OPN context of the child and that of the parent - however, we will skip describing this low-level plumbing of the CO-OPN model.

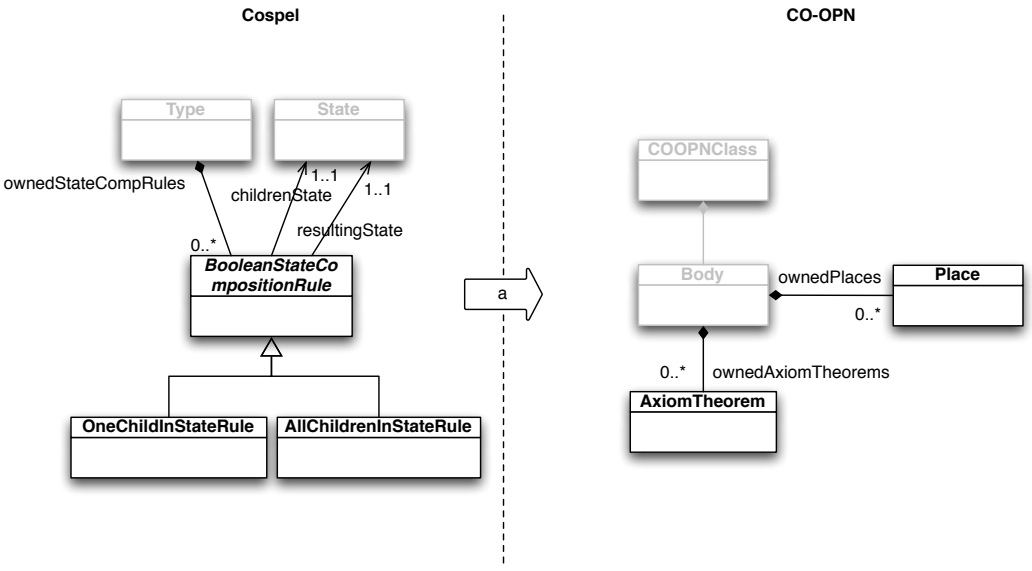

Figure 6.16. Transformation of the Cospel state rules package

6.4.4 Property package transformation

Figure 6.17 illustrates the transformation of the Cospel property package elements. A Cospel `Property` (attached to a `Type`) is transformed to a CO-OPN `Place` (attached to the corresponding `Class`). The type of the place

will reflect the property's type using CO-OPN's ADTs. For example, an integer property will be transformed to a `natural` place. It is in principle possible to use user-defined ADTs in this transformation, however this is not supported at the moment. It is worth noting that the part of the Cospel Property package dealing with alarms has no transformation at the CO-OPN level. That information is in fact rather used for the GUI engine.

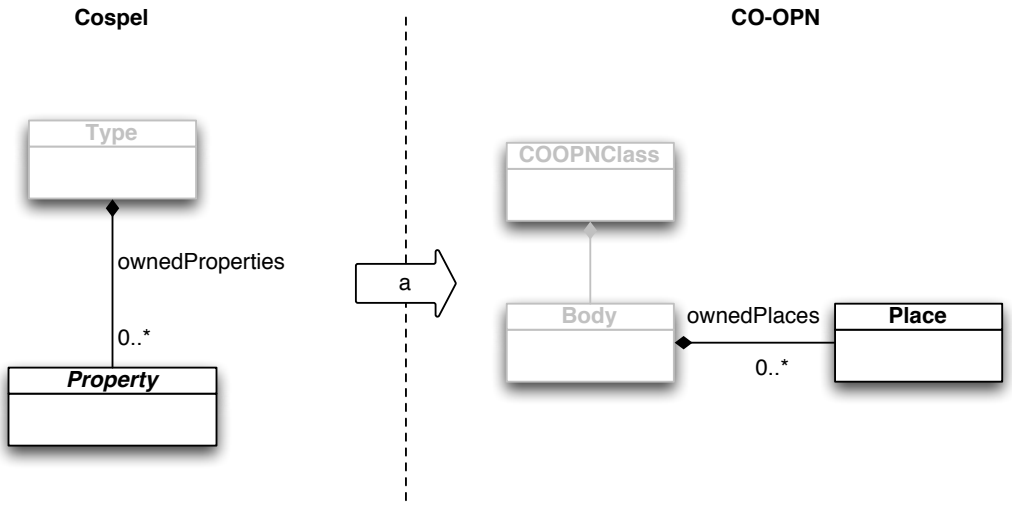

Figure 6.17. Transformation of the Cospel property package

6.4.5 Command and event package

Figure 6.18 illustrates the transformation of the Cospel event package elements. Cospel `Commands` (available services attached to a type) are transformed into CO-OPN `Methods` (attached to the respective class's interface). Cospel Events (requested services) are likewise transformed into CO-OPN `Gates`. Also, for each command/event, CO-OPN `Axiomtheorems` are added to the class to implement their behaviour (namely, to reflect the Cospel `Conditions` associated to the commands/events).

6.4.6 Result of the transformation

Figure 6.19 shows the structure of a part of the Cosmic Rack hierarchy using CO-OPN's visual syntax. It shows how objects are instantiated inside

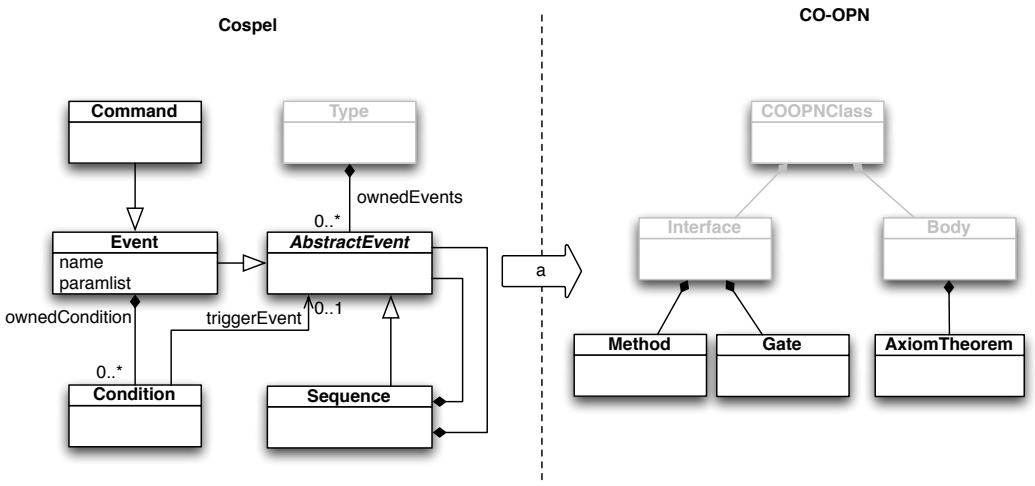

Figure 6.18. Transformation of the Cospel event package

imbricated contexts, how commands are routed (arrows) down the hierarchy until their destination object, and how events are routed up. The routes (called *synchronizations*) are decided based on conditions (e.g., the name of the destination object, or the parameters of the command/event), a part of which comes from the specification, and the rest are automatically generated in the transformation patterns. Appendix B, page 197, contains the full code of the CO-OPN class created for the Power Group objects.

The CO-OPN specification has two purposes. One one hand, it is used to generate Java code (using the CO-OPN tools). This code constitutes the executable system simulator mentioned in Figure 4.1. It can be run standalone, or called by API. It serves as the executable basis for evaluating the GUI prototype. The second purpose is running an analysis of the behaviour of the CO-OPN specification, to check that it is correct with respect to the original model. Since the generation of the GUI is based on the Cospel specification, it is important to verify that expected properties are satisfied. This can be accomplished by the model checking activity, described in the next chapter.

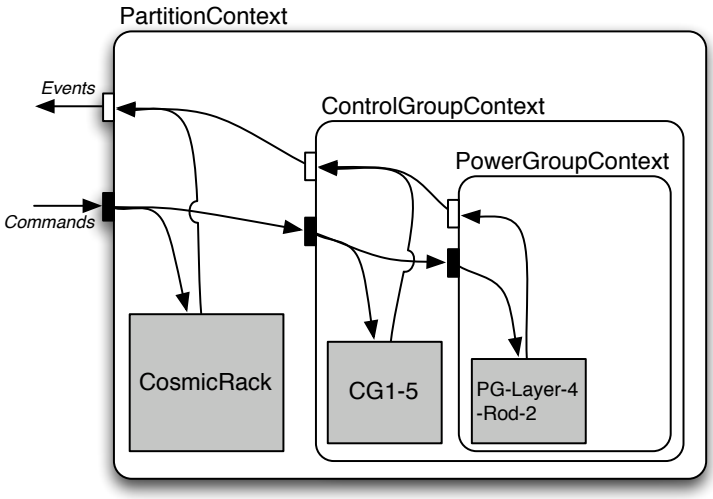

Figure 6.19. Hierarchical structure of CO-OPN model

6.5 The database

The database is the second artifact of the transformation. As Figure 4.1 shows, the DB is the source of information for the GUI engine to visualize the system. In the implementation of our project, the DB is a relational database made with MySQL. Its schema is shown in Figure 6.20. Only tables are shown for reading clarity; relationships simply implement the metamodel relationships between entities. Comparing this schema to the Cospel metamodel it is fairly easy to see that there is almost a 1:1 correspondence between the tables and the metaclasses.

The model transformations create an Structured Query Language (SQL) script. In this script, first an empty database is created, named after the Cospel specification. Then the specification is read, and the information about model entities are used to fill the tables. The final result is a database creation script which, when run, will create and populate a database for the Cospel specification that has been transformed. It is worth noting that the database is static in nature: the information it contains does not change in time, as it is merely structural information and does not reflect the runtime state of the system (that's what the system simulator is for).

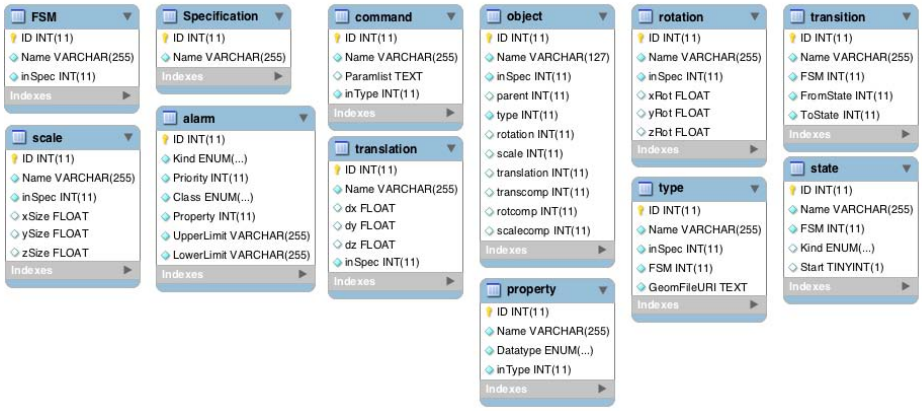

Figure 6.20. Schema of the produced database

6.6 Summary and conclusion

This chapter illustrated the transformation step of the methodology. We started by introducing the target formalism for transforming Cospel, called CO-OPN, and the transformation language, called ATL. We saw how the information in a Cospel specification is used to create two artifacts: a CO-OPN specification used to create the system simulator, and a database used for building the GUI prototype. We then saw in detail how each Cospel package is translated.

Next, we will talk about the GUI engine. This is the piece of software that uses the two artifacts created in the transformation step. It loads the database to build the GUI content, and communicates with the system simulator in order to allow a realistic user interaction.

Chapter 7

The GUI engine

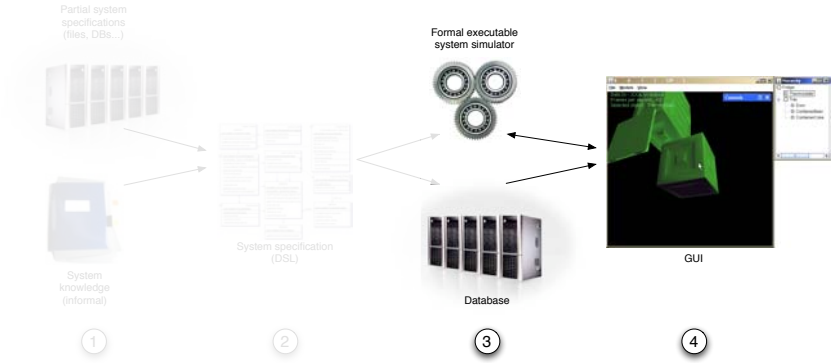

Figure 7.1. The GUI concretization steps of the BATIC^{3}S process

The GUI engine, which creates the GUI for the user to interact with, is written in Java. It constitutes step 4 of Figure 7.1 and uses the artifacts from step 3.

7.1 Technical description

The engine presents a 3D rendering of the system which allows spatial navigation and interaction with system components. Figure 7.2 shows the data flow of the GUI engine. Its source of information for rendering the scene are the database (DB) produced in the transformation phase, and the set of Object files containing the shapes of the system components. By load-

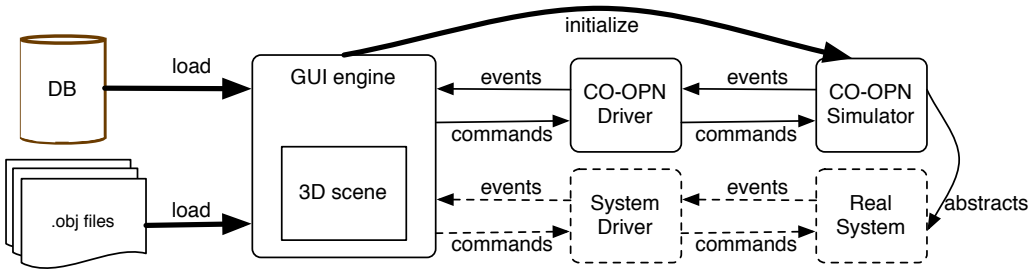

Figure 7.2. GUI engine communication

ing these resources, a 3D scene is initialized and shown to the user. At the same time, the CO-OPN system simulator produced in the transformation is instantiated and initialized.

To minimize memory usage (a serious problem for scaling to systems with a large number of components), features common to sets of objects (i.e., those features associated in the specification language to types rather than to individual objects, like geometrical shape) are loaded only once and stored in instances of a `Object3DStructure` class. Individual objects are obtained by instantiating an `Object3D` class which applies individual features (e.g., position in space) to the associated `Object3DStructure` instances.

Objects are organized in a tree, according to the hierarchy in the specification. Each of them keeps an instance of FSM, and the current state of an object is represented in the scene by colors, according to common practices in the field (green=ok, yellow=warning, red=error, blue=powered, gray=off or unknown).

Upon selecting objects, users are shown clickable controls corresponding to the commands defined for that object. These, when clicked, send a command message to the driver, which transmits it in the appropriate format to the CO-OPN simulator. Events coming from the simulator are also translated by the driver and can be interpreted by the GUI to update the scene (e.g., an object state change will modify its color, or trigger an error message).

After the development phase, when the prototype can be considered final, the CO-OPN driver can be substituted by a driver for the real system (see Figure 7.2), so that the interface can be evaluated in a real-life environment.

Rendering is done by using the JoGL API (JoGL expert group, n.d.), a Java binding of the OpenGL libraries developed by Sun and Silicon Graph-

ics. JoGL provides all necessary methods for 3D rendering and navigation. In addition to that, it has built-in support for stereoscopic visualization, which allowed us to experiment with how stereoscopic perception can affect interaction and navigation in a control system. We also used the FengGUI API (FengGUI developer group, n.d.), which allows drawing 2D components, like windows and buttons, in a 3D stereoscopic scene.

There are two navigation modes: spatial and hierarchical. Different controllers can be used for spatial navigation: keyboard, mouse (via click and drag gestures) or 3D controllers (e.g., six degrees of freedom knobs like 3D Connexion's SpaceNavigator (3DConnexion, n.d.)). Hierarchical navigation is done via a tree-like representation of the system.

The two navigation modes answer different types of tasks. When investigating a faulty component, the user may be interested in checking the nearby components' values to prevent further alarms, especially when the faults are of environmental nature (temperature, humidity). These components might well be "far" in terms of system hierarchy (i.e., belonging to a whole different branch of the hierarchy), but will be at a short distance in the 3D scene. The spatial navigation system provides easy access by moving the camera and directly clicking on them. On the other hand, if a user wants to quickly jump to another part of the system following an error, spatial navigation is not efficient as it requires several zooming, panning and rotating operations. In these cases, a few clicks on the tree representation provide a quick focus switch to another region of the system. Also, the tree representation can help selecting objects which are difficult to pick (very small or hidden by other objects).

Figure 7.3 shows a screenshot of the GUI prototype. A 3D representation of the Cosmic Rack is shown, and Power Groups are visible. Objects containing Power Groups (i.e., the Control Groups and the Cosmic Rack external shell) have been made invisible, in order to show one of the Power Groups which is having an error (it can be seen in a different color). The 3D scene is built using the information about the system hierarchy, FSMs and geometry.

The user middle-clicked on the component, which brought up possible commands in the bottom panel. The temperature property is also shown here. The engine knows what commands and properties to show because they had been specified in the model.

The right panel marks the name of the currently selected component in the hierarchical tree view of the system. Since the model says that the current user has access to the *off*, *on_lv*, *on* and *clear* commands of the Power

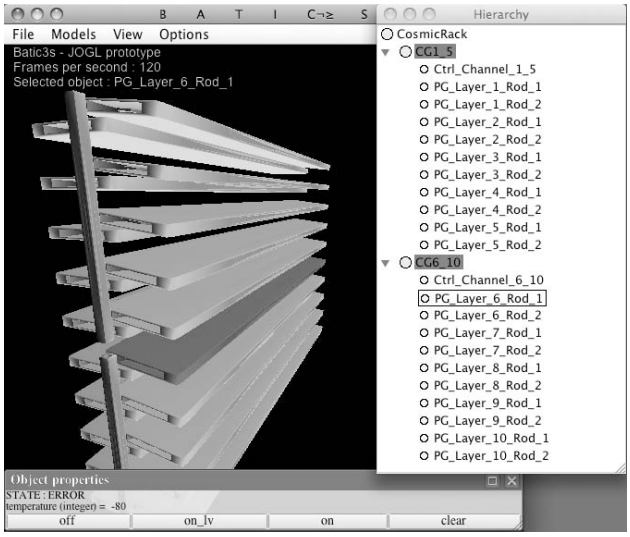

Figure 7.3. GUI prototype screenshot. The Power Groups of the Cosmic Rack can be seen in the 3D scene

Group, all these buttons appear in the bottom panel. A user with monitoring responsibilities only would not see the buttons.

7.2 Field test: the Cosmic Rack

After several iterations on specifying the Cosmic Rack model and refining the way the engine lets the user interact with objects, we evaluated how the interface fared in controlling the real system. Ideally we would have liked to test it on the actual Cosmic Rack hardware. However, the hardware was not available at the time of testing. This incidentally validated our use case in which users want to evaluate the GUI with a simulator when the system is not available.

Luckily, CERN was able to provide us with a "virtual" Cosmic Rack. It turned out they had their own software simulator, based on the actual drivers used for the hardware Cosmic Rack. It was made exactly of the same code that controlled the real system, only instead of CPUs answering to command there were software entities. From the outside, and for the purpose of the exercise of integrating our GUI engine with a real system, the fact of being a virtual system was completely transparent.

We adapted the drivers of the GUI engine to adopt the communication

protocol of the Cosmic Rack, which was SOAP messaging. We should probably remind now that our case study concerned what is called "slow controls", which is to say a scenario in which response times do not have to be strictly short. A network-mediated, SOAP-based communication protocol is thus acceptable in this case. Commands in the GUI engine were translated to appropriately formatted SOAP messages. Events from the machine were also received as SOAP messages and parsed to interpret them.

As expected, the interaction was a bit different from what we had experienced with the CO-OPN simulator. Mainly, the transactional nature of CO-OPN transitions meant that local simulation never showed intermediate states when giving commands that brought the system through a series of states (e.g., turning on the Power Groups, which meant turning on their low-voltage channels, followed by turning on their high-voltage channels). Interacting with the CERN system gave a more realistic feedback, in which the interface clearly showed the state changes of the system in succession, with objects going from gray to blue (the colour coding for intermediate states), then turning green after about a second. A high-definition video of the prototype running with the CERN system is available at `http://bit.ly/BATICSGUI` .

7.3 Summary and conclusion

This chapter gave a description of the GUI engine that concretizes the GUI prototype from the database produced from Cospel. We talked about how in interfaces with the system simulator, and how its drivers can be adapted to communicate with a real system. We reported our experience of interfacing with the Cosmic Rack system at CERN.

Since the focus of this thesis is on the methodology for GUI development, and not on the visualization paradigms themselves, we did not perform an evaluation of effectiveness and ergonomy of this kind of 3D interfaces for control systems. The choice of the 3D paradigm came from an interest expressed by the domain experts. The literature has some evaluation of 3D interfaces for other domains, like software analysis (Alam et al., 2009), document management (Cockburn & McKenzie, 2001) and search results visualization (Sebrechts, Cugini, Laskowski, Vasilakis, & Miller, 1999). In most cases, conclusions are tentative, and are mostly made of perspectives. This is motivated by the relative youth of the domain and the fact that only recently technology brought the possibility of having complex 3D interactive

interfaces with a reasonable investment. We could not find any complete, in-depth evaluation of 3D GUIs for control systems. The reference we found that was closest to this domain was (Wolf, Mofor, & Rode, 2007), in which 3D GUIs for manufacturing operations are presented. However, even this article has open conclusions about the evaluation of this type of GUIs.

The next chapter illustrates the different types of model checking and validation techniques that we applied to the models in this methodology. We give concrete examples of verification of different properties, and introduce the formal framework we used to achieve the goal of verifying the system simulator.

Chapter 8

Model checking and validation

Section 4.6 explained that there are two types of properties we want to check on our model. The first is checking that the Cospel specification is structurally valid; the second is checking that the system simulator generated from the Cospel specification behaves correctly. This section explains what tools and techniques are used to accomplish model checking on a Cospel model.

8.1 Validating the model's structure: static properties

To say that a model is valid, certain constraints have to be satisfied. Properties like "a model contains only one specification", "an object is only assigned to one type", "a type only has one geometry" are constraints that are expressed already in the metamodel, e.g., through the cardinality of the metaclass relationships. The metamodel itself can enforce these constraints, either by providing enough information (e.g., the cardinalities) for the editor to signal errors, or because in syntax-directed editing it is impossible to build a specification that violates them (correctness by construction).

On the other hand, some properties are not decidable by the metamodel information alone. For example, we know that in an FSM, a transition goes from a state to another state. But for the FSM to be valid, these two states and the transition itself must belong to the same FSM. We had expressed this property in abstract terms in section 4.6.1 (page 60). Now we can express it in terms of the metamodel elements (using terms from Figure 5.5) as the

following constraint, that has to hold for all transitions in all FSMs:

$$\begin{aligned} &\forall f \in m.ownedFSMs, \\ &\forall t \in f.ownedTransitions, \\ &\qquad (t.toState.FSM = t.fromState.FSM) \wedge \\ &\qquad (t.toState.FSM = t.FSM) \end{aligned} \tag{8.1}$$

where m is a Cospel model. To verify this constraint, several ways are possible. We explored two: OCL and Java.

8.1.1 OCL constraint verification

The constraint can be expressed using the Object Constraint Language (OCL) (Object Management Group, 2006). OCL has a syntax for specifying constraints over models. It supports quantifiers, collections, iterations and attributes expression. Using OCL syntax, we can write for example the above property as follows:

```
context Transition inv:
self.toState.FSM = self.fromState.FSM and
self.toState.FSM = self.FSM
```

This constraint is automatically applied to all matching contexts (thus, to all instances of `Transition`) and is treated as an invariant (keyword `inv`). Therefore, whenever a transition definition in an FSM violates this invariant, an error is signaled.

The EMF framework supports the use of OCL as a constraint language on models. This is accomplished by specifying the OCL constraints in an XML file. Specific libraries parse and evaluate the constraints during the modeling phase. For example, let's consider the above constraint on FSM transitions. This constraint applies to all instances of the Cospel metaclass `Transition` (see Figure 5.5). To add this constraint, we must add the following XML code to the validation configuration of the project:

```
<constraint statusCode="13" severity="ERROR" lang="OCL"
mode="Batch" name="Incoherent states in a transition"
id="transitionfsmcheck">
  <description>
```

```
      A transition from and to states must be in the
      same fsm as the transition
  </description>
  <message>
      One of your transitions has a from or to state
      in a different fsm
  </message>
  <target class="Transition"/>
  <!-- the \ac{OCL} expression -->
  <![CDATA[
      (fromState.FSM = toState.FSM) and (fromState.FSM = FSM)
  ]]>
</constraint>
```

The EMF-generated editor for Cospel integrates the check of this constraint in its code, via a dedicated "Validate" command (available through a right click on the specification elements), see Figure 8.1.

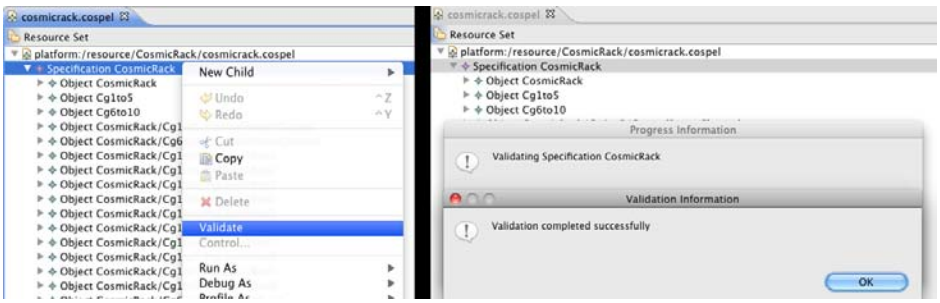

Figure 8.1. Screenshot of the EMF Validate tool

Model checking via OCL constraints has one drawback. The Validate command checks if the model is correct, but does not *prevent* the user from building an invalid model. A user can still define an invalid transition and save the model without performing validation. In some cases it would be desirable for some constraints to be applied at modeling time. This would reduce the possibility for modeling errors. In our transition example, it would be practical that the user could only choose states from the same FSM when defining a transition. This possibility is given by the Java constraint verification.

8.1.2 Java constraint verification

As we saw, EMF is capable of generating a model editor from a metamodel definition. This editor is generated as a collection of Java classes that are then compiled and executed. The Java code defines, among other things, what are the values that the user can select in the editor when defining an entity. For example, when the user defines a Transition, he has to choose the source and destination states in a combo box; the values appearing in this box are the result of a query on the model, giving all the names of the states in the model. The code providing the state names for the source state is shown in listing 8.1.

Listing 8.1. Original generated code snippet for cospel.FSMPackage.provider.TransitionItemProvider: query returning possible values for the source state

```
protected void addFromStatePropertyDescriptor(Object object)
   {
2  itemPropertyDescriptors.add
     (createItemPropertyDescriptor
4      (((ComposeableAdapterFactory)adapterFactory).
                                 getRootAdapterFactory(),
6       getResourceLocator(),
        getString("_UI_Transition_fromState_feature"),
8       getString("_UI_PropertyDescriptor_description",
            "_UI_Transition_fromState_feature",
10          "_UI_Transition_type"),
        FSMPackagePackage.Literals.TRANSITION__FROM_STATE,
12      true,
        false,
14      true,
        null,
16      null,
        null));
18 }
```

The source code for this value generator can be modified so that the query only returns the names of the states belonging to the same FSM as the transition. This avoids completely the possibility that the user chooses a state from a different FSM, thus ensuring the property by construction. Listing 8.2 shows the modified code; the `getComboBoxObjects` method at the end (lines 19–25) is the one providing the correct values.

Listing 8.2. Modified code snippet for `cospel.FSMPackage.provider.TransitionItemProvider`: query returning only states from the transition's FSM

```
protected void addFromStatePropertyDescriptor(Object object)
    {
  itemPropertyDescriptors.add
    (new ItemPropertyDescriptor
      (((ComposeableAdapterFactory)adapterFactory).
                              getRootAdapterFactory(),
        getResourceLocator(),
        getString("_UI_Transition_has_from_state_feature"),
        getString("_UI_PropertyDescriptor_description",
           "_UI_Transition_has_from_state_feature",
           "_UI_Transition_type"),
        FSMPackagePackage.Literals.TRANSITION__FROM_STATE,
        true,
        false,
        true,
        null,
        null,
        null)
        {
          protected Collection<State>
            getComboBoxObjects(Object object)
          {
            FSM fsm=((Transition)object).getFSM();
            return fsm.getOwnedStates();// This is where
                                        // state values
                                        // are retrieved
          }
        });
}
```

Modifying the generated Java code has a disadvantage: if at a later time the metamodel is modified, the Java code of the editor has to be re-generated to reflect the metamodel modifications. This entails that the Java code modifications we made to the code might be lost. Although it is possible to put some markers in the code to specify that the methods we modified should not be overwritten in subsequent code generation steps, this is not always possible (the metamodel modifications could require re-generation of the modified methods). Because of this, the Java constraint verification is somehow fragile and difficult to track. Our experience suggests using it with extreme caution (and regular backups of the modifications) and only when it is absolutely necessary (e.g., when one really wants constraints to be enforced at modeling time).

8.2 Dynamic properties model checking

Checking dynamic properties implies being able to generate the state space of a model, i.e., calculating all possible states the model can assume and verify if the property holds for some or all of them. A CO-OPN specification is, at its foundations, a Petri net with an algebra defining data types. It is possible in principle to calculate its state space. However, currently there are no tools that achieve this goal directly on a CO-OPN specification. One of the reasons for this lack is the extremely complex semantics of CO-OPN, and the fact that model checking tools typically operate on much lower-level formalisms (and even then, with some limitations). It is thus necessary to bridge this semantic gap by transforming once again the CO-OPN specification into an equivalent, lower level model.

We will now give the definition of a possible approach to transform CO-OPN models to equivalent specifications in a different Petri-net based formalism, Algebraic Petri Nets (APNs), which is a step in the direction of model checking CO-OPN specifications. The implementation of a model checking framework for CO-OPN is out of the scope of this work. It is worth noting that in the same group where this thesis was developed, there is ongoing work on the implementation of a model checking tool for APNs, called AlPiNA (Buchs & Hostettler, 2009; Buchs, Hostettler, Marechal, & Risoldi, 2010), a first release of which already exist.

8.2.1 Definition of CO-OPN and APN Specifications

Given the following:

- an algebraic specification ADT
- a set of places P
- a set of transitions T
- a set of axioms $Axioms_{COOPN}$ denoted by $t\ with\ sync : pre \rightarrow post$

Definition 8.1. A CO-OPN specification is defined as

$$Spec_{COOPN} \stackrel{\text{def}}{=} \langle P, T, Axioms_{COOPN}, ADT \rangle$$

Given a set of axioms $Axioms_{APN}$ denoted by $t : pre \rightarrow post$,

Definition 8.2. an APN specification is defined as

$$Spec_{APN} \stackrel{\text{def}}{=} \langle P, T, Axioms_{APN}, ADT \rangle$$

8.2.2 Creating composable APN boxes

CO-OPN synchronizations can be related through concurrency operators, namely a *simultaneous* operator "//", a *sequential* operator ".." and a *choice* operator "+". Fig. 8.2 shows an example of such a synchronization, where two simple transitions $t1$ and $t2$ are related via the simultaneous operator. This impies that firing $t1$ will cause a simultaneous firing of $t2$. More details on the implications of this semantics are in the following section.

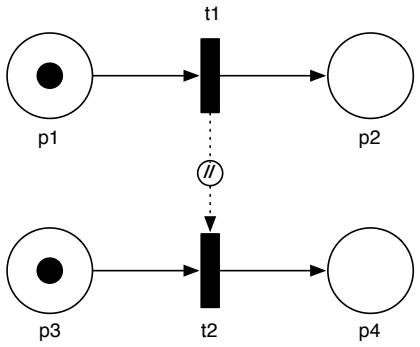

Figure 8.2. Example of a simultaneous synchronization

One can see the elementary synchronizations as "boxes" that connect their behaviour through these operators. The approach we follow is to translate the simpler CO-OPN synchronizations (the "boxes") to APN fragments. We then use a strategy to compose these fragments in a way that reflects the semantics of the concurrency operators of the more complex CO-OPN synchronizations in the original model. Fig. 8.3 shows a schematized overview of the approach. The "boxes" of the transitions from Fig. 8.2 are transformed to two APN "boxes", containing an APN fragment each. The information about the simultaneous operator is translated in a way to link the APN fragments so that the semantics are respected.

This approach is similar to the *versatile boxes* strategy introduced in (Pommereau, 2007). The author of that article defines an algebra that allows composing high-level Petri nets, using operators for sequence, choice, iteration and traps (interruptions). With respect to that work, we might say

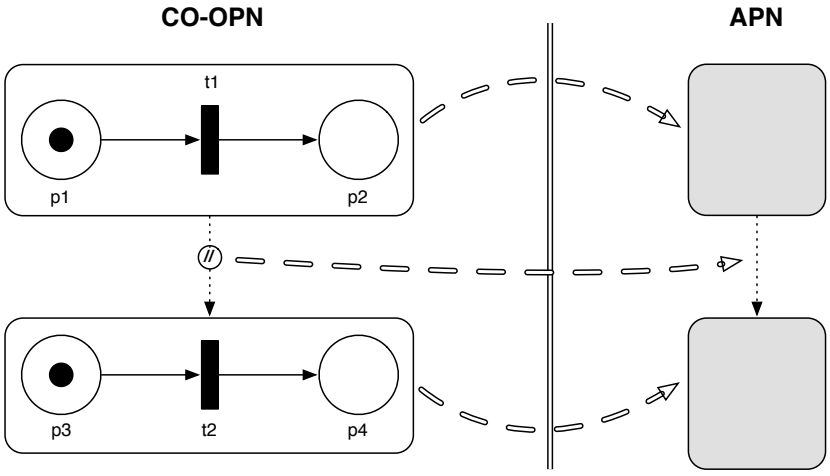

Figure 8.3. Overview of the "boxes" transformation approach

that what we propose here sacrificed generality to simplicity. Our approach is simpler, meaning that we don't need it to be as general as what Pommereau proposes – e.g., we don't need traps – so we can eliminate some complexity from the approach. It is also more adapted to our models, meaning that we do have, for example, nested parallel composition and transitions synchronization – although arguably these features can be reproduced with versatile boxes as well.

8.2.3 Synchronizations as transactions

A CO-OPN synchronization is considered as a transaction. The first step to translating a CO-OPN specification to a APN one is translating the synchronization concept to a basic Petri net fragment.

Given a transition $t \in T$, we can *split* it into a starting transition t_{start} and an ending transition t_{end} (Figure 8.4).

If t has pre- and postconditions, the preconditions become preconditions of t_{start} and the postconditions become postconditions of t_{end}. Also, the preconditions of t_{start} are cached and made available to t_{end} (Figure 8.5), since the postconditions' value might be calculated by t_{end} based on the preconditions' value.

This is the basic building brick for translations. Before seeing how to translate synchronizations of transitions, we will introduce transition fusion.

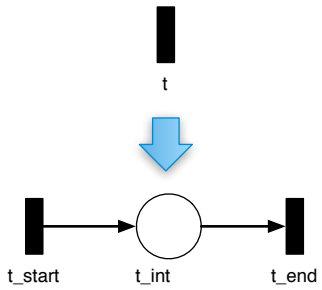

Figure 8.4. Splitting a transition to its start and end transitions

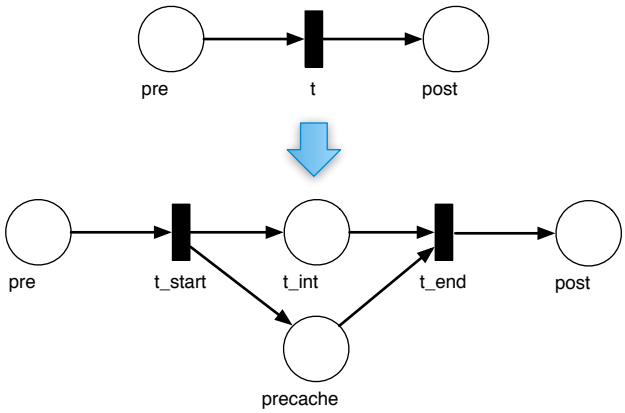

Figure 8.5. Splitting a transition to its start and end transitions (with pre- and postconditions)

8.2.4 Definition of transition fusion

We define the syntactic fusion operator, denoted by $FusionOpSet$, with the following profile:

$$name\#name_1, ..., name_n \in T^{n+1} \tag{8.2}$$

The semantics of this consist in the union of all the sets of pre- and postconditions of transition $name_i$ involved in the fusion operation. This produces a new set of pre- and postcondition associated to the new resulting transition $name$.

Let $t, s \in T$ be two transitions, where t is defined by the axiom $t : pre \rightarrow post$ and s is defined by the axiom $s : pre' \rightarrow post'$, the **fusion** of t and s, denoted by

$$t\_s\#t, s \tag{8.3}$$

creates a t_s transition with a precondition set which is the union of the preconditions of t and s. Figure 8.6 shows the graphical representation of the fusion process.

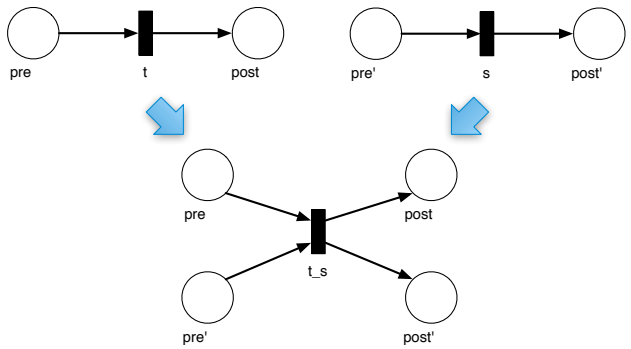

Figure 8.6. Transition fusion example

More generally, we can define a $Fusion$ operator with the following profile:

$$\wp(Axioms_{APN}) \times FusionOpSet \to \wp(Axioms_{APN}) \tag{8.4}$$

defined as follows:

$$Fusion(Axioms_{APN}, name\#name_1 \ldots name_n) \stackrel{\text{def}}{=} Axioms_{APN} \cup \{name : pre \to post\} \tag{8.5}$$

where

$$pre = \bigcup_{i=1}^{n} Pre(Axioms_{APN}, name_i) \tag{8.6}$$

and

$$post = \bigcup_{i=1}^{n} Post(Axioms_{APN}, name_i) \tag{8.7}$$

Proposition 8.3. *If two transitions t and s, defined by the axioms $t : pre \to post$ and $s : pre' \to post'$ are fireable concurrently, then their fusion $t\_s\#t, s$ is fireable.*

Proof. *Since t and s are fireable concurrently, it means that $pre \cup pre'$ is satisfied. Since the precondition for t_s is $pre \cup pre'$, t_s is fireable.*

8.2.5 Translation of synchronization expressions

We will now start a series of definitions for operations translating CO-OPN synchronizations to APN axioms. We follow an "incremental" approach: we start with definitions working for basic cases, then we enrich these definitions to involve more complex and general cases.

We define the $TrBox$ translation operation, acting on synchronization expressions:

$$TrBox : Sync \rightarrow \mathbb{N} \times Axioms_{APN} \times \wp(FusionOpSet) \times T \times T \times T \times T \tag{8.8}$$

This translation produces axioms, fusion operators and transitions. The result can be seen as a "box", with an entry point T_{start} and an exit point T_{end} (the fourth and fifth element of $TrBox$'s codomain). The semantics is: the synchronization is fireable if, in the translated net, it exists a fireable sequence of transitions starting with T_{start} and ending with T_{end}.

Simple coercions

We start by defining $TrBox$ with a rule for the simplest case, the splitting of a coercion t without synchronization operators, defined by a single axiom, with a precondition pre, a postcondition $post$ and a guard $cond$:

$$t : cond \Rightarrow pre \rightarrow post \tag{8.9}$$

The rule for simple coercions, called $Trans$, is the following:

$$\frac{}{TrBox(t) = \langle i, newAx, \oslash, t_{start}, t_{end}, t_{substart}, t_{subend} \rangle} \quad Trans \tag{8.10}$$

where $i = 0$ and:

$$\begin{aligned} newAx \quad = \quad & \{t_{substart} : temp1 \rightarrow\} \cup \\ & \{t_{subend} :\rightarrow temp2\} \cup \\ & \{t_{start} : cond \Rightarrow pre \rightarrow temp, temp1, precache\} \cup \\ & \{t_{end} : cond \Rightarrow temp2, temp, precache \rightarrow post\} \end{aligned} \tag{8.11}$$

with $precache$ being a marking where each value taken from a place p in pre is put in a corresponding cache place p_{cache}. The purpose of the i index, as

well as of the $t_{substart}$ and t_{subend} transitions, will be clearer when we tackle synchronization axioms and recursion in sections 8.2.7 and 8.2.8 respectively.

An example of application of the $Trans$ rule to a coercion:

$$t : a = 3 \Rightarrow p1\ a \rightarrow p2\ (a + 1)$$

is shown in Figure 8.7.

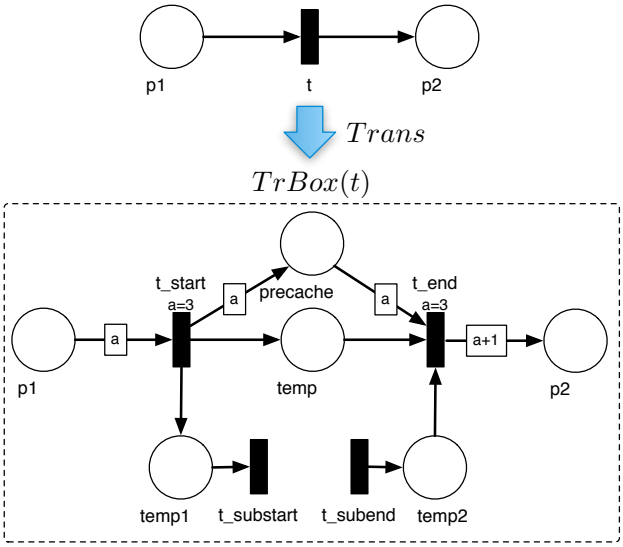

Figure 8.7. Application of the Trans rule to a simple coercion

Lemma 8.4. *The result of a $TrBox$ operation on a synchronization expression s, $TrBox(s) = \langle i, Ax, Fus, s_{start}, s_{end}, s_{substart}, s_{subend} \rangle$ is said to be* fireable *if $s_{start}, s_{end}, s_{substart}$ and s_{subend} are fireable in a sequence $s_{start}, s_{substart}, s_{subend}, s_{end}$, if concurrent modifications of resources are ignored.*

Proposition 8.5. *Given a CO-OPN coercion axiom $t : cond \Rightarrow pre \rightarrow post$, and the result of $TrBox(t) = \langle i, newAx, \oslash, t_{start}, t_{end}, t_{substart}, t_{subend} \rangle$, if t is fireable, then $TrBox(t)$ is fireable.*

Proof. *If t is fireable, it means that pre is available and $cond$ is true.*

Since t_{start} has the same preconditions and guard as t, then t_{start} is fireable.

Since t_{start} creates the preconditions for $t_{substart}$, then $t_{substart}$ is fireable after t_{start}.

Since t_{subend} has no preconditions, it is fireable regardless of the other transitions in $newAx$.

Finally, since t_{start} and t_{subend} create the preconditions for t_{end}, then t_{end} is fireable after t_{start} and t_{subend}.

Simultaneous synchronizations

Recursively, we define the rule for a simultaneous synchronization $s \,//\, s'$, called Sim. Each iteration produces new T_{start} and T_{end} transitions, as well as a new $temp$ place and new $Sseq1$ and $Sseq2$ transitions (the latter are used to get into and out of the intermediate state):

$$\frac{\begin{array}{c} TrBox(s) = \langle i, Ax, Fus, S_{start}, S_{end}, S_{substart}, S_{subend}\rangle \\ TrBox(s') = \langle i, Ax', Fus', S'_{start}, S'_{end}, S'_{substart}, S'_{subend}\rangle \end{array}}{TrBox(s//s') = \langle i, newAx, newFus, T_{start}, T_{end}, T_{substart}, T_{subend}\rangle} Sim \tag{8.12}$$

where:

$$newAx = Ax \cup Ax' \cup \{Sseq1 :\rightarrow temp\} \cup \{Sseq2 : temp \rightarrow\} \tag{8.13}$$

and:

$$\begin{array}{rcl} newFus & = & Fus \cup \\ & & Fus' \cup \\ & & \{T_{start}\#S_{start}, S'_{start}, Sseq1\} \cup \\ & & \{T_{end}\#S_{end}, S'_{end}, Sseq2\} \cup \\ & & \{T_{substart}\#S_{substart}, S'_{substart}\} \cup \\ & & \{T_{subend}\#S_{subend}, S'_{subend}\} \end{array} \tag{8.14}$$

An example is shown in Figure 8.8. It shows the application of the Sim rule to a simultaneous synchronization $s//s'$, where s and s' are defined respectively by the following axioms:

$$\begin{array}{l} s :: a = 3 \Rightarrow p1\ a \rightarrow; \\ s' :: c = 1 \Rightarrow p1\ c \rightarrow; \end{array} \tag{8.15}$$

The top part of Figure 8.8 shows the expansion of s and s', while the bottom part shows the result of the application of the Sim rule. A certain number of

redundant duplicate places are omitted in the figure (for example, the multiple *temp* places, one for each sub-synchronization and one for the resulting synchronization), both for readability and because we can assume that an optimized implementation of this transformation process would fuse them anyway.

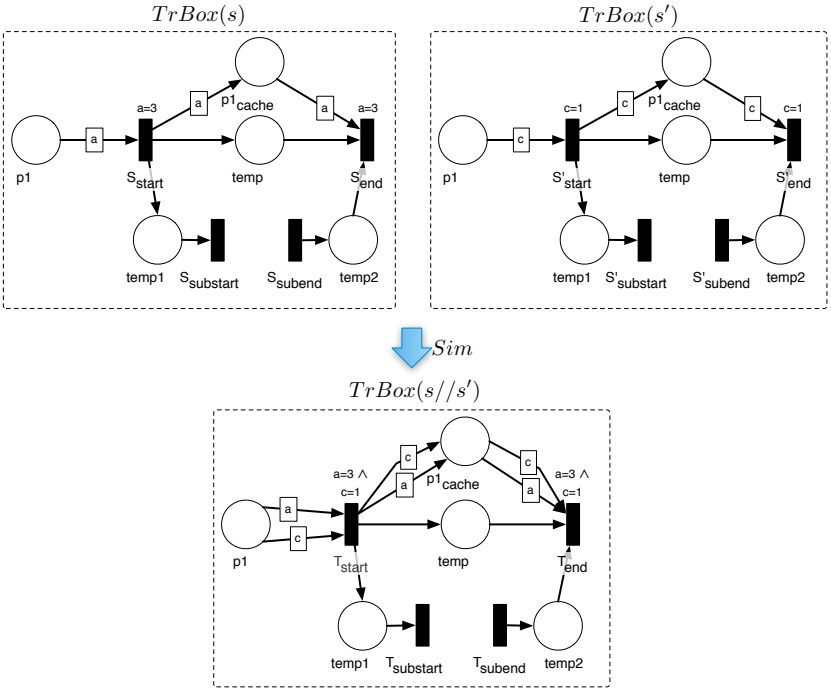

Figure 8.8. Example of transformation of $s//s'$

Proposition 8.6. *If $TrBox(s)$ and $TrBox(s')$ are fireable, and a precondition $pre = Pre(Ax, s) \cup Pre(Ax', s')$ is available, then $TrBox(s//s')$ is fireable when pre is available.*

Proof. *The following points prove the proposition:*

- *S_{start} and S'_{start} are fireable, and there are enough resources to satisfy the precondition of both. $Sseq1$ is fireable as it has no preconditions. Thus, $T_{start}\#S_{start}, S'_{start}, Sseq1$ is fireable.*
- *Since $Sseq1$ (and therefore, T_{start}) creates the precondition for $Sseq2$, then the latter is fireable after T_{start}.*

- *Since also* S_{end} *and* S'_{end} *are fireable, then* $T_{end}\#S_{end}, S'_{end}, Sseq2$ *is fireable after* T_{start}.
- *Fireability of* $T_{substart}$ *and* T_{subend} *can be proved in the same way as in the proof of Proposition 8.5.*

Sequential synchronizations

For the case of synchronizations that use the sequentiality operator, like $s..s'$, we define the following rule Seq. Each iteration produces new T_{start} and T_{end} transitions, as well as $temp1, temp2, temp3$ places (marking intermediate states, respectively during the s transaction, between s and s', and during the s' transaction) and $Sseq1, Sseq2, S'seq1$ and $S'seq2$ transitions. Assuming ε as a generic symbol for unnamed transitions:

$$\frac{\begin{array}{c} TrBox(s) = \langle i, Ax, Fus, S_{start}, S_{end}, S_{substart}, S_{subend}\rangle \\ TrBox(s') = \langle i, Ax', Fus', S'_{start}, S'_{end}, S'_{substart}, S'_{subend}\rangle \end{array}}{TrBox(s..s') = \langle i, newAx, newFus, T_{start}, T_{end}, T_{substart}, T_{subend}\rangle} Seq \quad (8.16)$$

where:

$$\begin{array}{rcl} newAx & = & Ax \cup \\ & & Ax' \cup \\ & & \{Sseq1 :\rightarrow temp1, subtemp1\} \cup \\ & & \{Sseq2 : temp1 \rightarrow temp2\} \cup \\ & & \{S'seq1 : temp2 \rightarrow temp3\} \cup \\ & & \{S'seq2 : temp3, subtemp2 \rightarrow\} \cup \\ & & \{T_{substart} : subtemp1 \rightarrow\} \cup \\ & & \{T_{subend} :\rightarrow subtemp2\} \end{array} \quad (8.17)$$

and:

$$\begin{array}{rcl} newFus & = & Fus \cup \\ & & Fus' \cup \\ & & \{T_{start}\#S_{start}, Sseq1\} \cup \\ & & \{\varepsilon\#Sseq2, S_{end}\} \cup \\ & & \{\varepsilon\#S'_{start}, S'seq1\} \cup \\ & & \{T_{end}\#S'seq2, S'_{end}\} \end{array} \quad (8.18)$$

Figure 8.9 shows an example of application of the *Seq* rule to a sequential synchronization $s..s'$, where s and s' are defined by the same axioms in 8.15.

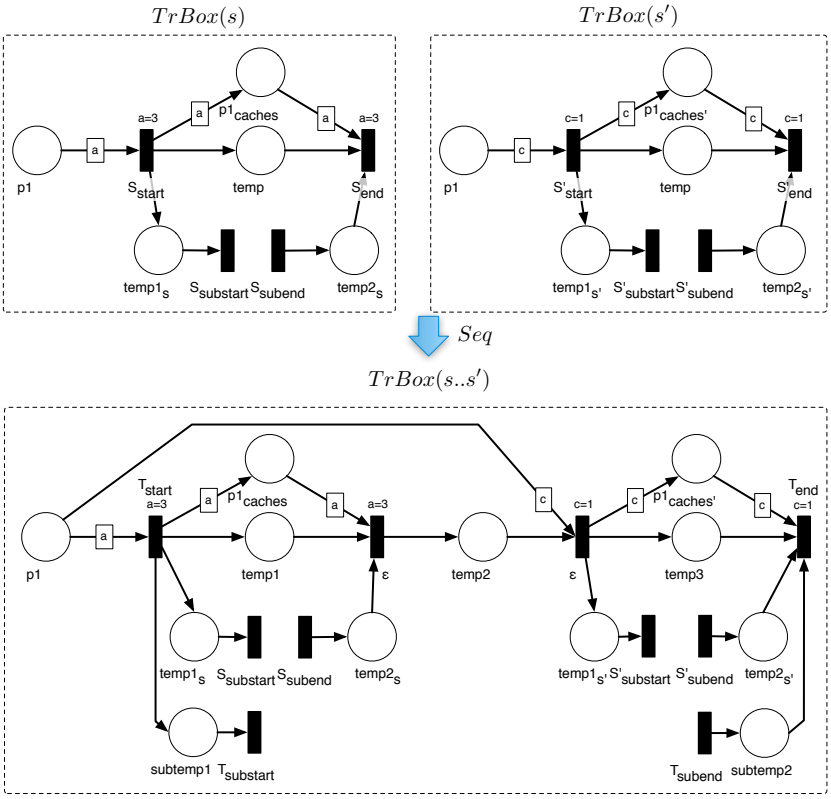

Figure 8.9. Example of application of the Seq rule

Proposition 8.7. *If $TrBox(s)$ is fireable, and after the firing $TrBox(s')$ is fireable, then $TrBox(s..s')$ is fireable.*

Proof. *The following points prove the proposition:*

- *S_{start} is fireable. $Sseq1$ is fireable as it has no preconditions. Thus, $T_{start}\#S_{start}, Sseq1$ is fireable.*
- *S_{end} is fireable. $Sseq1$ creates the preconditions for $Sseq2$. Thus, $\varepsilon\#Sseq2, S_{end}$ is fireable after T_{start}.*
- *S'_{start} is fireable, with a precondition that was either available from the start, or created (in part or in whole) by S_{end}. $Sseq2$ creates*

the preconditions for $S'seq1$. *Thus,* $\varepsilon\#S'_{start}, S'seq1$ *is fireable after* $\varepsilon\#Sseq2, S_{end}$.

- S'_{end} *is fireable.* $S'seq1$ *creates the preconditions for* $S'seq2$. *Thus,* $T_{end}\#S'seq2, S'_{end}$ *is fireable after* $\varepsilon\#S'_{start}, S'seq1$.

- *Fireability of* $T_{substart}$ *and* T_{subend} *can be proved in the same way as in the proof of Proposition 8.5.*

This thesis does not give rules for the "non-deterministic-choice" synchronizations of CO-OPN as they are not used in this work. They are given in (Buchs, Buffo, & Kordon, 2000).

8.2.6 Managing multiple axioms

The above definitions work when the transactions taking part in synchronizations are described by only one CO-OPN axiom each (thus, applying the $Trans$ rule produces only one $TrBox$ per transaction). This is not however the general case. A transaction s can have multiple behaviours, each described by a different axiom. Thus, when it takes part in a synchronization, the resulting behaviour depends on which of the axioms is chosen. To manage this multiplicity of axioms, the idea is calculating all possible combinations of axioms for the transactions involved in a synchronization, and creating a translation for each synchronization (as a side note, this could entail a very large number of combinations).

We thus redefine the $Trans$ rule to take into account multiple axioms. If a simple cohercion t has n axioms $\{t_1, .., tn\}$, then $TrBox(t)$ must be applied to each of these axioms, effectively producing a set of result (i.e., a set of "boxes"):

$$\frac{}{TrBox(t) = \bigcup_{c=1}^{n} \{\langle i, newAx_c, \oslash, t_{start_c}, t_{end_c}, t_{substart_c}, t_{subend_c}\rangle\}} Trans \quad (8.19)$$

It is clear that if t only has one axiom, then this definition gives the same result as the previous one which did not treat multiplicity. Properties

concerning fireability are maintained for each axiom: if a CO-OPN axiom is fireable, the equivalent $TrBox$ is fireable.

In a synchronization $s//s'$, where s and s' may have multiple axioms, the Sim rule should be applied to each possible couple b, b', where $b \in TrBox(s), b' \in TrBox = (s')$. Consequently, the Sim rule must be rewritten as follows to account for multiple axioms:

$$\frac{\begin{array}{c} TrBox(s) = \bigcup_{c=1}^{n} \{\langle i, Ax_c, Fus_c, S_{start_c}, S_{end_c}, S_{substart_c}, S_{subend_c}\rangle\} \\ TrBox(s') = \bigcup_{d=1}^{m} \{\langle i, Ax'_d, Fus'_d, S'_{start_d}, S'_{end_d}, S'_{substart_d}, S'_{subend_d}\rangle\} \end{array}}{TrBox(s//s') = \bigcup_{e=1}^{n \times m} \{\langle i, newAx_e, newFus_e, T_{start_e}, T_{end_e}, T_{substart_e}, T_{subend_e}\rangle\}} Sim \tag{8.20}$$

The same is true for the Seq rule applied to a $s..s'$ synchronization with multiple axioms. The Seq rule must be rewritten as follows:

$$\frac{\begin{array}{c} TrBox(s) = \bigcup_{c=1}^{n} \{\langle i, Ax_c, Fus_c, S_{start_c}, S_{end_c}, S_{substart_c}, S_{subend_c}\rangle\} \\ TrBox(s') = \bigcup_{d=1}^{m} \{\langle i, Ax'_d, Fus'_d, S'_{start_d}, S'_{end_d}, S'_{substart_d}, S'_{subend_d}\rangle\} \end{array}}{TrBox(s..s') = \bigcup_{e=1}^{n \times m} \{\langle i, newAx_e, newFus_e, T_{start_e}, T_{end_e}, T_{substart_e}, T_{subend_e}\rangle\}} Seq \tag{8.21}$$

Properties concerning fireability are maintained for each possible couple b, b': if b and b' are fireable concurrently (resp. one after the other), then $TrBox(s//s')$ (resp. $TrBox(s..s')$) is fireable.

For example, let's consider a synchronization $s//s'$, where s and s' are defined by the following axioms:

$$\begin{array}{l} s :: a = 3 \Rightarrow p1\ a \rightarrow; \\ s :: b = 2 \Rightarrow p1\ b \rightarrow; \\ s' :: c = 1 \Rightarrow p1\ c \rightarrow; \end{array} \tag{8.22}$$

Figure 8.10 shows an example of translation of this synchronization. The top left part shows the two possible expansions for s, the bottom part shows the two possible combinations of $TrBox(s)$ with $TrBox(s')$.

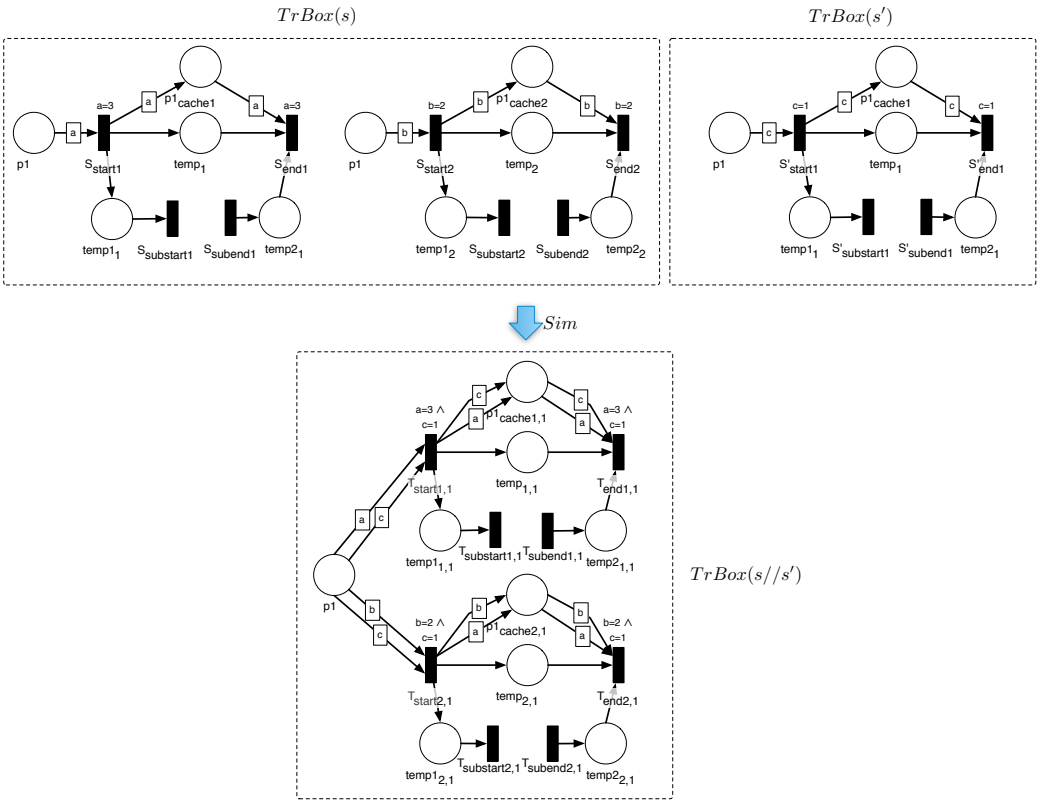

Figure 8.10. Example of application of the Sim rule with multiple axioms

8.2.7 Translation of synchronization axioms

Once the expressions used in synchronizations are translated as explained above, the synchronization axioms themselves need to be translated. CO-OPN synchronizations are in the form:

$$t\ [with\ syn] : [cond \Rightarrow][pre] \rightarrow [post]; \tag{8.23}$$

where t is a simple coercion, syn is a synchronization (i.e., either a simple coercion or something in the form $s..s'$ or $s//s'$), $cond$ is t's guard, pre is

the precondition and *post* is the postcondition. The parts between square parentheses are optional. The semantics are that when t is invoked, it not only has to accomplish its condition and pre- and postconditions, but also *syn* has to do the same (and, recursively, all synchronizations participating in *syn*). The behaviour is "all-or-nothing": if t or any part of *syn* is not fireable, the whole transaction is not fireable:

Lemma 8.8. *A CO-OPN synchronization axiom:*

$$t\ with\ syn : [cond \Rightarrow][pre] \rightarrow [post];$$

is fireable if pre is available, cond is true, and syn is fireable.

The $With$ rule defines the $TrBox$ operation for axioms which include a *with* synchronization:

$$\frac{\begin{array}{c} TrBox(t) = \bigcup_{c=1}^{n} \{\langle i, Ax_{1_c}, Fus_{1_c}, t_{start_c}, t_{end_c}, t_{substart_c}, t_{subend_c}\rangle\} \\ TrBox(syn) = \bigcup_{d=1}^{m} \{\langle i, Ax_{2_d}, Fus_{2_d}, syn_{start_d}, syn_{end_d}, syn_{substart_d}, syn_{subend_d}\rangle\} \end{array}}{TrBox(t\ with\ syn) = \bigcup_{c=1}^{n}\bigcup_{d=1}^{m} \{\langle j, newAx_{c,d}, newFus_{c,d}, t_{start_{c,d}}, t_{end_{c,d}}, t_{substart_{c,d}}, t_{subend_{c,d}}\rangle\}}\ With \tag{8.24}$$

where $j = i + 1$ and:

$$newAx_{c,d} \quad = \quad Ax_{1_c} \cup Ax_{2_d} \tag{8.25}$$

and:

$$\begin{array}{rcl} newFus_{c,d} & = & Fus_{1_c} \cup \\ & & Fus_{2_d} \cup \\ & & \{t_{substart_{c,d}} \# t_{substart_c}, syn_{start_d}\} \cup \\ & & \{t_{subend_{c,d}} \# t_{subend_c}, syn_{end_d}\} \end{array} \tag{8.26}$$

Figure 8.11 shows an example of translation of a synchronization $t\ with\ s//s$, where s and s' are described by the same axioms as in 8.15. This example has only one axiom for each synchronization, however in a similar way as we saw before, this rule can be applied to synchronizations defined by multiple axioms as well.

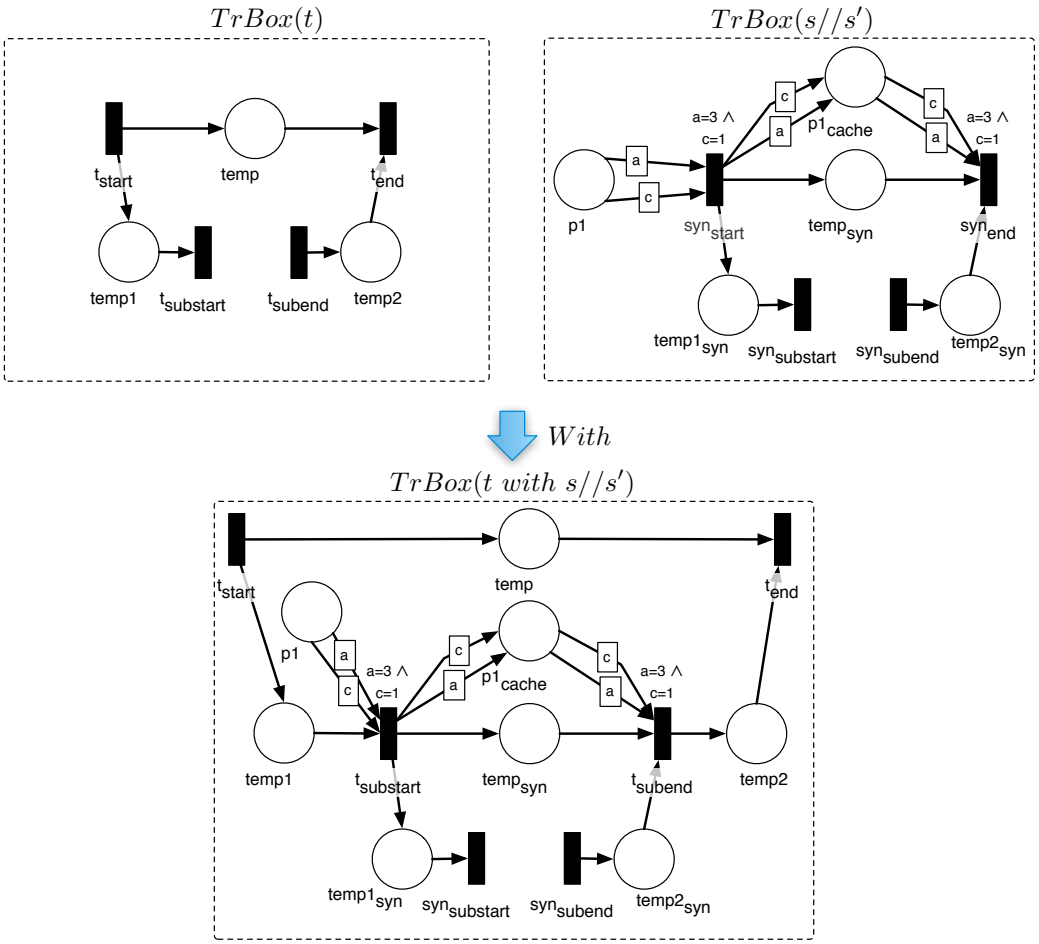

Figure 8.11. Example of application of the With rule

8.2.8 Recursive synchronizations

Synchronizations can involve recursive behaviour. For example, in the following axiom set:

$$\begin{aligned} t(succ(n))\ with\ t(n) &: p1\ @ \rightarrow p2\ @; \\ t(0) &: p3\ @ \rightarrow p4\ @; \end{aligned} \tag{8.27}$$

the t transition has two axioms defining two behaviors: a recursive case, and a terminal case. It is worth noting that a well designed CO-OPN model will normally have a terminal case for recursive synchronizations (otherwise, infinite recursion would cause livelocks in the model execution).

Recursion is a problematic pattern to translate. A complete translation involves calculating transition parameters for all possible iterations. However, even if we exclude infinite recursion *a priori*, the number of iterations can be very high. The strategy to treat recursion is to bound the application of the $With$ rule to a certain number of iterations. Recursions can't be deeper than this bound; if a terminal case can not be reached within the allowed number of iterations, the recursion will fail. The bound can be set as high or low as appropriate. The $With$ rule is rewritten to accomodate a guard on this bound (here the bound is set to 10 levels of recursion):

$$\frac{\begin{array}{c} i < 10 \\ TrBox(t) = \bigcup_{c=1}^{n} \{\langle i, Ax_{1_c}, Fus_{1_c}, t_{start_c}, t_{end_c}, t_{substart_c}, t_{subend_c}\rangle\} \\ TrBox(syn) = \bigcup_{d=1}^{m} \{\langle i, Ax_{2_d}, Fus_{2_d}, syn_{start_d}, syn_{end_d}, syn_{substart_d}, syn_{subend_d}\rangle\} \end{array}}{TrBox(t\ with\ syn) = \bigcup_{c=1}^{n}\bigcup_{d=1}^{m} \{\langle j, newAx_{c,d}, newFus_{c,d}, t_{start_{c,d}}, t_{end_{c,d}}, t_{substart_{c,d}}, t_{subend_{c,d}}\rangle\}} With \tag{8.28}$$

When the bound is reached, a forced termination of recursion has to be produced. This is accomplished by the $WithTerm$ rule. It is very similar to the $With$ rule, except that instead of the recursive sub-synchronization, it produces a transition t_{stop} with a false guard:

$$\frac{\begin{array}{c} i = 10 \\ TrBox(t) = \bigcup_{c=1}^{n} \{\langle i, Ax_{1_c}, Fus_{1_c}, t_{start_c}, t_{end_c}, t_{substart_c}, t_{subend_c}\rangle\} \\ TrBox(syn) = \bigcup_{d=1}^{m} \{\langle i, Ax_{2_d}, Fus_{2_d}, syn_{start_d}, syn_{end_d}, syn_{substart_d}, syn_{subend_d}\rangle\} \end{array}}{TrBox(t\ with\ syn) = \bigcup_{c=1}^{n}\bigcup_{d=1}^{m} \{\langle j, newAx_{c,d}, \oslash, t_{stop_{c,d}}, \oslash, \oslash, \oslash\rangle\}} WithTerm \tag{8.29}$$

where:

$$\begin{aligned} newAx_{c,d} \;=\; & Ax_{1_c} \cup Ax_{2_d} \cup \\ & \{t_{stop_{c,d}} : false \Rightarrow\} \end{aligned} \tag{8.30}$$

It is worth noting that the firing of the t_{stop} transition is not a successful termination of the recursion, but rather expresses that the recursion is too deep for the established bound. Thus, it symbolizes a failed firing of the whole transaction, as it does not terminate all the recursive calls.

To illustrate the application of these rules, let's consider the translation of the set of axioms in 8.27, which we repeat here for convenience:

$$\begin{aligned} t(succ(n))\ with\ t(n) &: p1\ @ \rightarrow p2\ @; \\ t(0) &: p3\ @ \rightarrow p4\ @; \end{aligned} \tag{8.31}$$

An application of the $Trans$ rule for $t(0)$ gives the following result:

$$TrBox(t(0)) = \{\langle 0, Ax_{t0}, \oslash, t0_{start}, t0_{end}\rangle\} \tag{8.32}$$

For $t(succ(n))$ the application of $Trans$ produces:

$$TrBox(t(succ(n))) = \{\langle 0, Ax_{tsuc}, \oslash, tsuc_{start}, tsuc_{end}\rangle\} \tag{8.33}$$

For the translation of the axiom

$$t(succ(n))\ with\ t(n)$$

the $With$ rule is applied iteratively, each time translating $t(n)$ to the $TrBox$es of the multiple possible axioms: the terminal case $TrBox(t(0))$ or the recursive case $TrBox(t(succ(n)))$. The iteration is repeated until the bound is reached, and the $WithTerm$ rule is applied.

Figure 8.12 shows the result with a bound $i = 2$. The top part of the figure shows the elementary boxes for the two possible behaviours of t, and the large schema in the bottom part shows the final result of applying the $With$ and $WithTerm$ rules to the $t(succ(n))\ with\ t(n)$ axiom. Different shade of grey are used to highlight nested levels of recursion.

8.2.9 Comparing transition systems

The original goal of the CO-OPN to APN transformation is to perform model checking on the resulting APN model, which is arguably easier as it is a lower-level model. However, in order for the model checking results to have a meaning for the original CO-OPN specification, we must ask ourselves what is the relationship between the transition systems of the two models. As we saw, the APN model has additional states and transition which do not exist in the CO-OPN model. It is a safe bet to say that the properties we are expecting from the model will probably not hold in these additional APN

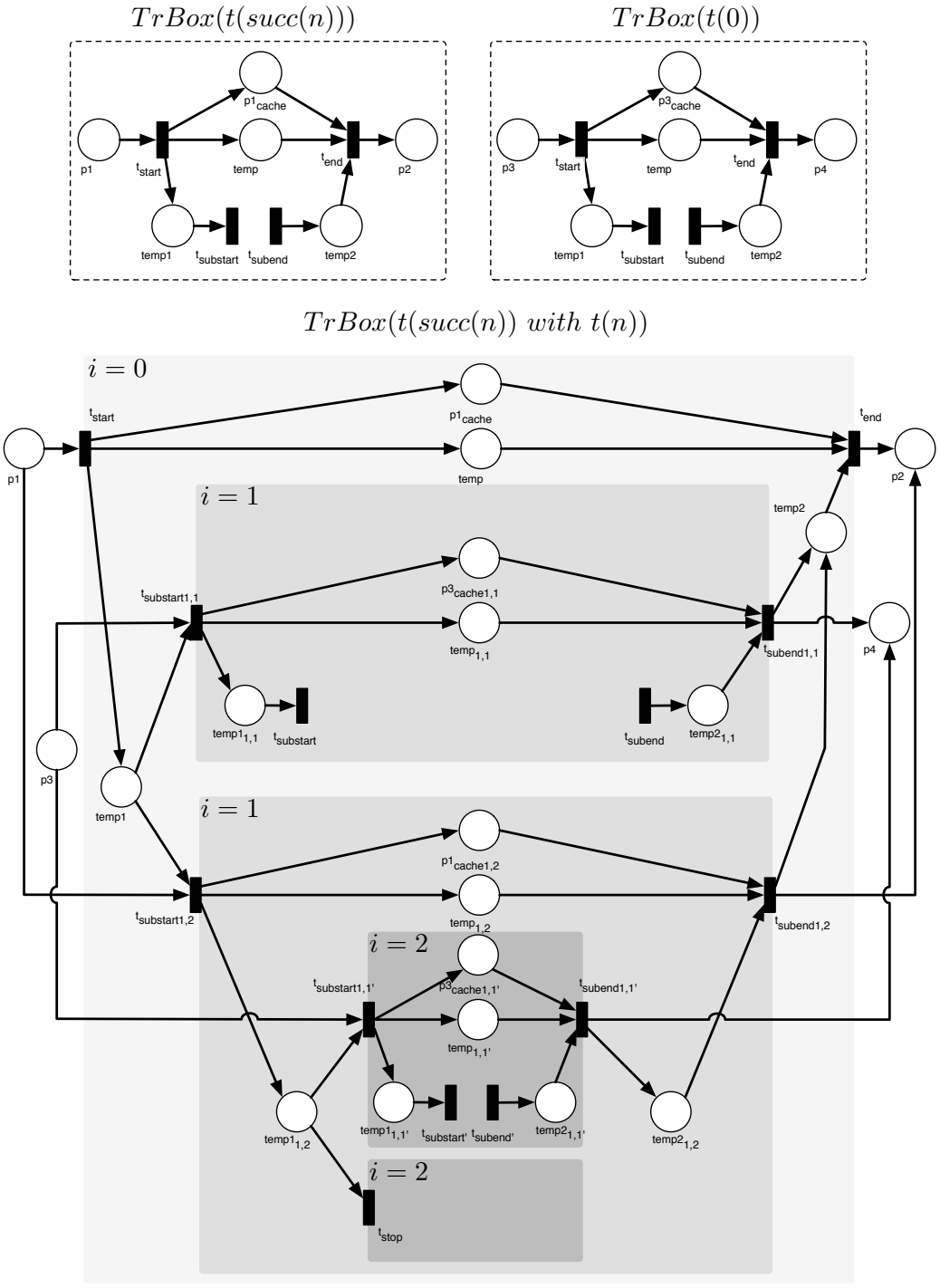

Figure 8.12. Example of application of the rules for recursion

states. The model checking activity has to ignore the satisfaction values of properties in these states, and only consider them in the states which have a corresponding CO-OPN state. It is thus necessary to define what does it mean to satisfy a property in terms of the transformed model.

A labeled transition system LTS is defined as:

$$LTS = \{S, T, \alpha, \beta, \lambda\} \tag{8.34}$$

Where:

- S is a set of states
- T is a set of transitions
- $\alpha : T \to S$ is the function giving the source state of a transition
- $\beta : T \to S$ is the function giving the destination state of a transition
- λ is a set of transition labels

Let $LTS_{COOPN} = \{S, T, \alpha, \beta, \lambda\}$ and $LTS_{APN} = \{S', T', \alpha', \beta', \lambda'\}$. We can define a mapping between states τ_S:

$$\tau_S : S \to S' \tag{8.35}$$

The τ_S mapping is defined on the basis of firing sequences:

Definition 8.9. Given the following:

- $s_1, s_2 \in S$ two states in the CO-OPN model
- $t \in T$ a transition such that $\alpha(t) = s_1$ and $\beta(t) = s_2$
- $s'_1, s'_2 \in S'$ a couple of APN states
- $t'_1 ... t'_n$ a transition sequence such that $\alpha'(t'_1) = s'_1$ and $\beta'(t'_n) = s'_2$
- $TrBox(t) = \langle i, Ax, Fus, t_{start}, t_{end}, t_{substart}, t_{subend} \rangle$ and $t'_1 = t_{start}$ and $t'_n = t_{end}$

then $\tau_S(s_1) = s'_1$ and $\tau_S(s_2) = s'_2$.

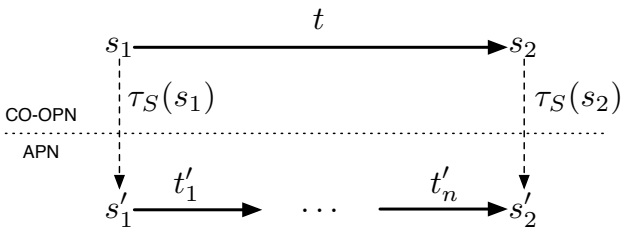

Figure 8.13. Mapping between LTS's

Figure 8.13 illustrates the τ_S mapping.

Let $\mathcal{P}$ be a set of properties on the CO-OPN model, and $\mathcal{P}'$ a set of properties on the APN model. We can define a mapping $\tau_{\mathcal{P}}$:

$$\tau_{\mathcal{P}} : \mathcal{P} \rightarrow \mathcal{P}' \tag{8.36}$$

If model checking reveals that a property $p' \in \mathcal{P}'$ holds in a set of states $S'_{p'} \subseteq S'$ in the APN model, we can conclude that a property $p \in \mathcal{P}$ holds in a set of states $S_p \subseteq S$ in the CO-OPN model if:

- $\tau_{\mathcal{P}}(p) = p'$ and
- $\forall s \in S_p, \tau_S(s) \in S'_{p'}$

8.2.10 Example: model checking an FSM

To illustrate the workflow of performing verification on the CO-OPN specification, we will now give an example. Since the full model would not fit in these pages and would be unreadable, we will perform property verification on just one of the objects of the system, a Power Group.

The original CO-OPN specification

Appendix B, on page 197, contains the full code of the CO-OPN class created for the Power Group type in the Cosmic Rack specification. This class is instantiated twenty times in the Cosmic Rack by as many CO-OPN contexts. Let's consider one of the instances, the one corresponding to the second Power Group of the fourth layer (`PG-Layer-4-Rod-2`). Listing 8.3 presents a few snippets of code from the class specification, namely, those that implement the FSM of the Power Group. The `Places` section contains a series of places

that model the FSM states. These are of the `blackToken` sort (a kind of non-coloured token similar to what you can find in a P/T net). The `OFF` place is initialized with a black token (`@`). The `Methods` section defines a list of methods that represent the FSM transitions. The transition behaviour is defined in the `Axioms` sectiojn: each transition consumes a black token from one state place and produces one into another place.

Listing 8.3. FSM part of the CO-OPN specification for the Power Groups class

```
Class PowerGroup

[...]

Body

  Methods
    OFF_TO_ON_LV;
    ON_LV_TO_ON;
    ON_TO_ON_LV;
    ON_LV_TO_OFF;
    ERROR_TO_OFF;
    ERROR_TO_ON_LV;
    OFF_TO_INTERLOCKED;
    ON_LV_TO_INTERLOCKED;
    ON_TO_INTERLOCKED;
    ERROR_TO_INTERLOCKED;
    INTERLOCKED_TO_OFF;
    [...]

  Places
    ON _ : blackToken;
    ON_LV _ : blackToken;
    OFF _ : blackToken;
    ERROR _ : blackToken;
    INTERLOCKED _ : blackToken;
    [...]

  Initial
    OFF @;
    [...]

  Axioms
    OFF_TO_ON_LV ::
      OFF @ -> ON_LV @;
    ON_LV_TO_ON ::
      ON_LV @ -> ON @;
    ON_TO_ON_LV ::
      ON @ -> ON_LV @;
    ON_LV_TO_OFF ::
      ON_LV @ -> OFF @;
    ERROR_TO_OFF ::
      ERROR @ -> OFF @;
    ERROR_TO_ON_LV ::
      ERROR @ -> ON_LV @;
    OFF_TO_INTERLOCKED ::
```

```
        OFF @ -> INTERLOCKED @;
48    ON-LV-TO-INTERLOCKED ::
        ON-LV @ -> INTERLOCKED @;
50    ON-TO-INTERLOCKED ::
        ON @ -> INTERLOCKED @;
52    ERROR-TO-INTERLOCKED ::
        ERROR @ -> INTERLOCKED @;
54    INTERLOCKED-TO-OFF ::
        INTERLOCKED @ -> OFF @;
56    [...]

58 End PowerGroup;
```

The transformed APN specification

The places in the CO-OPN class are transformed to places in the APN. The axioms for the FSM transitions are simple coercions. Thus, we apply the *Trans* rule (8.19 on page 141). We obtain the result shown in Figure 8.14, using the graphical representation provided by the AlPiNA model checker (SMV Group, 2010). Figure 8.14 only shows the translation for three of the axioms for readability - there are eleven in total. The ones shown in the figure are the axioms for the `ONLV-TO-OFF`, `OFF-TO-ONLV` and `ONLV-TO-ON` methods. Moreover one should not forget that the full code of the class is also translated. However, that would make for a quite unreadable APN on a printed page. This underlines the aspect that low level formalisms are good for machines but not for humans.

The property to verify

AlPiNA, in its current iteration, supports a property language to express reachability properties about the markings of places and their cardinality. This allows us to check, for example, that the FSM of the Power Group object respects the mutual exclusion of states: one object can only be in one state at a given time. Since states are represented by places with black tokens in the APN, this means that the cardinality of the union of all state places contents must be one. This is written in Listing 8.4 using AlPiNA's property language.

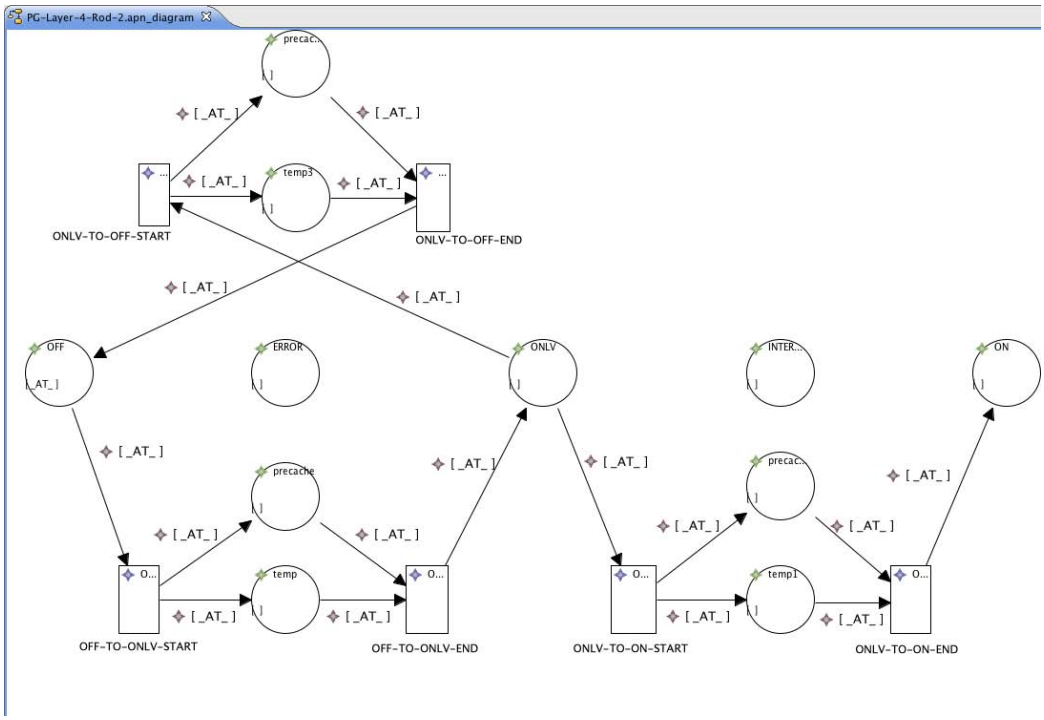

Figure 8.14. Transformed APN for the `PG-Layer-4-Rod-2` *Power Group (partial)*

Listing 8.4. Property: mutual exclusion of states

```
Expression

MUTUAL_EXCLUSION :
    (card(ON) +
     card(ONLV) +
     card(OFF) +
     card(ERROR) +
     card(INTERLOCKED) = 1;
```

Also, the translation to APNs introduces intermediate states (the ones where there are tokens in the temporary places `temp`, `temp1`, etc). These states are not really part of our system simulator's state space, and should be ignored while checking most reachability properties. This implies that the previous mutual exclusion property must hold only in all non-intermediate states, i.e., when no tokens are in the temporary places. The condition of being in an intermediate state is written in Listing 8.5.

Listing 8.5. Intermediate states

```
   TEMP : card(temp)=1;
   TEMP1 : card(temp1)=1;
   TEMP2 : card(temp2)=1;
   TEMP3 : card(temp3)=1;
   TEMP4 : card(temp4)=1;
   TEMP5 : card(temp5)=1;
   TEMP6 : card(temp6)=1;
   TEMP7 : card(temp7)=1;
   TEMP8 : card(temp8)=1;
   TEMP9 : card(temp9)=1;
   TEMP10 : card(temp10)=1;
   INTERMEDIATE_STATE : @TEMP | @TEMP1 | @TEMP2 |
                        @TEMP3 | @TEMP4 | @TEMP5 |
                        @TEMP6 | @TEMP7 | @TEMP8 |
                        @TEMP9 | @TEMP10;
```

Finally, we build the property to check by saying that for all states of the APN, not being in an intermediate state implies having exactly one token in the conjunction of state places. This is written in Listing 8.6.

Listing 8.6. Property: FSM mutual exclusion for relevant states

```
Check
  (!(@INTERMEDIATE_STATE) => @MUTUAL_EXCLUSION);
```

Having done this, we can let AlPiNA calculate the state space and verify that the property is satisfied for all states. This is indeed the case, and AlPiNA is able to provide an answer in a few milliseconds (as shown in Figure 8.15).

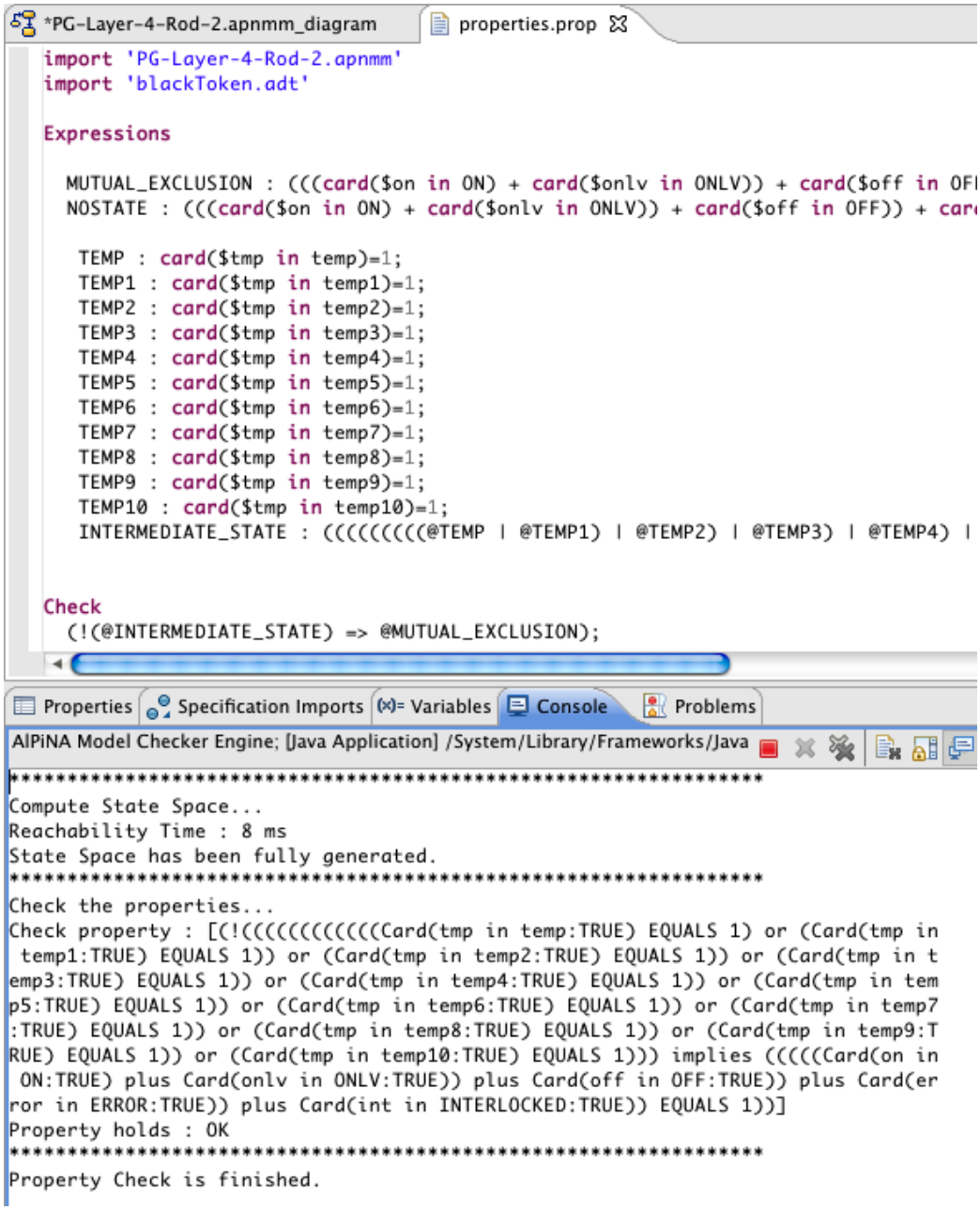

Figure 8.15. AlPiNA's property checking result

With the same method we can test all other properties (at least, all those that are expressible as reachability properties). For example, we can check that all FSM states are reachable:

Listing 8.7. States reachability

```
Expressions

  ON_REACHABLE :
    exists($on in ON : $on = _BT_); // _BT_ represents a black token
    // repeat for all other states

Check

  ! @ON_REACHABLE;  // this should fail and give a counterexample
  // repeat for all other states
```

Future extensions of AlPiNA will allow checking more complex temporal properties, like the ability to fire an FSM transition infinitely often (Buchs et al., 2010).

8.3 Summary and conclusion

This chapter first introduced and gave concrete examples for two different techniques we employed to verify static properties on Cospel models. Then we explained the formal framework we use to achieve verification of dynamic properties on the system simulator. We illustrated the transformation of CO-OPN to APN, and gave an example of verification of FSM properties on the transformed APN model.

The next chapter will abstract from the case study at hand and put the work in this thesis into a more general perspective of how to conceive and develop DSMLs. We will discuss what problems arise when property verification is an issue, and when transformations and compositions are involved. We will also give research perspectives for an improvement of the DSML development methodology.

Chapter 9

Generalization

The BATIC^{3}S workflow includes many technologies, and we gave an overview of its application to a particular case study for control systems. In this chapter we will start by resuming a couple of overview concepts about user roles and formalisms. Then we will abstract general principles for a language engineering-based methodology such as BATIC^{3}S. These are lessons learned, suggestions, or simply generalizations of the BATIC^{3}S approach.

9.1 Human roles in the methodology

With such a structured process, one natural question to ask is, who does what. Each step of the process illustrated in Figure 4.1 on page 51 requires a different set of competences, and may involve one or more different types of users to perform the corresponding activities.

Figure 9.1 shows the engineering process (including the transformation for model checking step) and indicates the different users involved in each activity.

There are three types of users involved: the language engineer (represented as a stick figure with a thick trait); the domain expert (represented with a thin trait); and the final user of the GUI (represented with a black head), who might or not be the same person as the domain expert.

The process starts with the analysis of the domain (control systems, in our case) by the language engineer and the domain expert, which leads to the definition of the key domain concepts. These are used by the language engineer to design a domain-specific modeling language (Cospel, in our case).

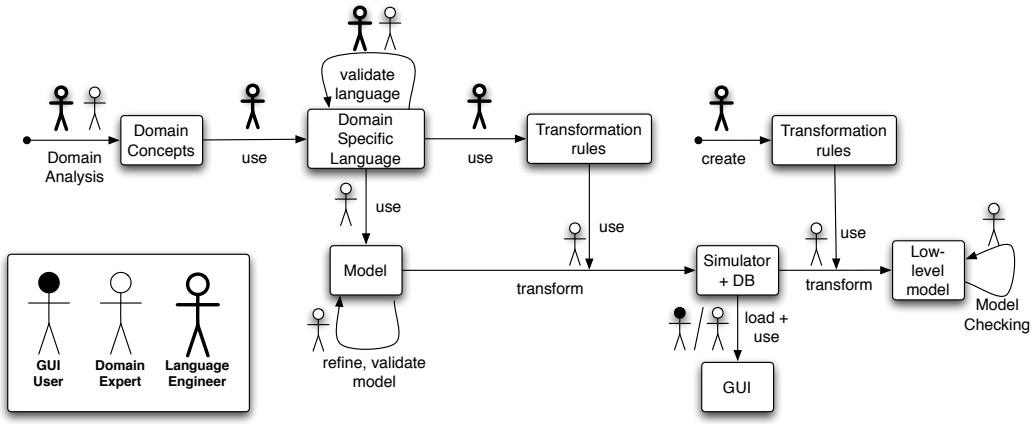

Figure 9.1. Roles in the engineering process

The language is then validated with the domain expert, and refined if necessary.

Then the language engineer has to define the transformation rules (ATL rules in our case) which should transform the domain-specific model to the formalism(s) used for the simulator and the DB (CO-OPN and MySQL in our case). These transformation rules are defined on the base of the DSML. Finally, the language engineer has to define the transformation rules to transform the simulator into a lower-level model apt for model checking. This step might be not necessary if the simulator already allows the type of dynamic checks needed; in our case, the CO-OPN simulator had to be transformed into an APN in order to perform model checking. This completes the toolkit that is provided to the domain expert.

The latter uses the DSML to create a model, which is then validated and further refined. He then transforms the model to the simulator, and if needed transforms the simulator to a lower-level model to perform model checking. In parallel, the GUI user (which may or may not be the domain expert himself) can load the executable specification and use the GUI to control it.

9.2 Multi-paradigm modeling

By looking at this work, one will notice that models undergo a certain number of transformations in order to perform different activities. These models are

expressed using different formalisms, described by different metamodels, and at different levels of abstraction. This way of modeling systems is called *multi-paradigm modeling* (MPM), and was introduced in (Vangheluwe, de Lara, & Mosterman, 2002) and developed in further articles by the same working group. The philosophical point of view behind MPM is that, as long as models are complete and transformations are sound and do not lose too much information, working with different formalisms to achieve different goals is more efficient than building a single model in a single formalism which tries to cover all possible activities.

In our case, the model of a system undergoes a number of incremental formalism transformations to reach goals which would be hard without this approach. For example, model checking is easier on a low-level formalism like APNs, but quickly becomes intractable on a more abstract formalism like CO-OPN (or even impossible on the original Cospel model, where no semantics are yet defined). Figure 9.2 resumes the different formalisms and transformations used in the context of this work.

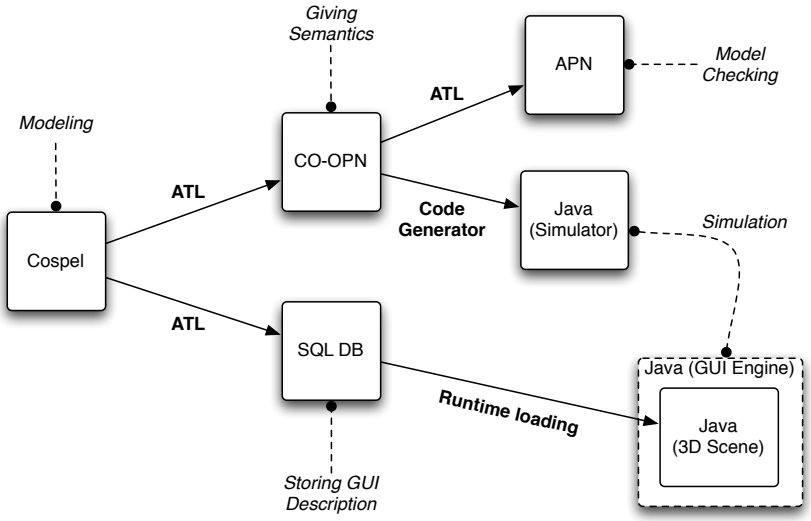

Figure 9.2. Transformations and formalisms used in this thesis

The rounded boxes represent the models, with their formalism indicated by labels. Arrows with bold labels indicate transformations. Ball-tipped dashed lines with italics labels indicate what activities the model is best-suited for.

This kind of approach, however, introduces a delicate problem. If one reasons about the topmost path in Figure 9.2, it is apparent that in order to perform model checking on the APN model we need to have some properties to check. These properties must be expressed in an appropriate formalism at the level of abstraction of the APN model. However, they are properties that concern the original Cospel model (and in a way that also concern the intermediate CO-OPN model). One cannot reasonably expect the user to build a model in Cospel and express its properties in an APN-specific format. In principle, there should be a way to express properties at the highest level of abstraction, and keep them through transformations down the line. Section 9.6.2 analyzes this specific problem in more detail. Figure 9.3 shows an abstract view of this approach. Properties expressed at the domain specific level are transformed into equivalent property sets at each level of abstraction with each transformation. The transformation rules from one property specification to another are a subset of the transformations rules from one model to another.

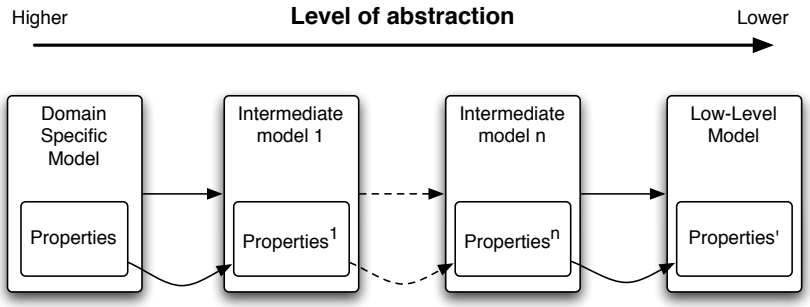

Figure 9.3. Expressing properties at all levels

In general, every time a methodology includes transformation steps like we show here, a guiding principle is that the original model should be as complete as possible, including all available information. This may as well be at a high level of abstraction, but the key point is that these information should be usable in some way later down the line in another phase of the methodology. It is however worth noting that in some cases part of this information might be lost in some of the transformations, especially when going from a high level of abstraction to a lower one. It is however important to keep the information that is lost in every step in some appropriate format. This enables going back in the transformation chain, which in turn allows, for

example, interpreting the APN model checking results in terms of the original Cospel model. Figure 9.4 is an abstract example of how this should work. Information lost in the transformations is kept together with the transformed models, and follows the transformation chain. Again, Section 9.6.2 deals with this problem in a more detailed way.

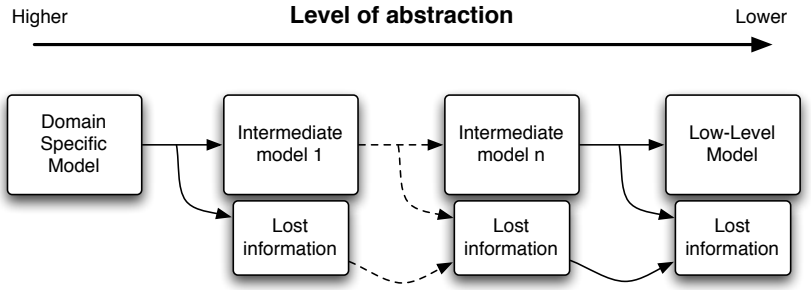

Figure 9.4. Keeping all information throughout transformations

9.3 Conceiving a DSML

A DSML, by definition, is made of concepts specific to a given domain. Collecting these concepts is a delicate task. On one hand, the metamodel of the DSML has to be precise enough to allow further treatment of models. On the other hand, one has to avoid being biased by what the methodology needs to do with the models. One example of a DSML which is biased by the use of the models is the Petri Nets Markup Language (PNML). This was originally conceived as a portable format to describe Petri net models. As such, it should have included concepts proper of the Petri nets domain (places, transition, arcs, tokens, weights, and so on). However, PNML is being used in some visual editors for Petri nets. As such, it includes things like the layout of the net in the editor, which in itself have no relationship to the domain of Petri nets. The result is that this feature bloats the language and adds a further layer of complexity in managing information in a PNML model.

Conceiving a DSML is an activity which should focus on the domain and on the users for the language. Sources of information which are useful for deciding what concepts to include are for example domani analysis, ontologies, dictionaries, even database schemas. A review of literature is generally

a good idea, as it can clarify what the current use of terminology can be. The experience in this project made clear that the use of terminology is a key factor in the acceptance of a language by the user, and it should not be underestimated. At one point in the initial phases of the work on Cospel, a strong opposition to the concept of "component" was resolved by renaming that concept to "object".

In any case, we feel we can strongly assert that language engineering, despite the name, is far from being simply an engineering problem. Any proposed solution will have to be based not only on technical bases, but on user feedback and an iterative and interactive process.

9.4 Modular development of languages

One of the long-term goals of the metamodel-based approach to language engineering is to improve reuse and speed up the engineering process. In this perspective, a good idea is to think in a modular way. Very often, it is possible to identify relatively independent aspects in a language's metamodel. In Cospel we saw that the metamodel can be seen as the composition of simpler metamodels representing different aspects of the language (FSMs, hierarchies, events, etc). By reusing such simpler blocks, one can quickly concretize partial or whole new languages for different domains.

9.4.1 Composing metamodels

Still, creating a metamodel by composition of smaller parts is all but a trivial job. Previous work (Pedro, 2009) investigated various types of composition. The approach is based on refining part of a metamodel (named *formal parameter*, or *fp*) with another metamodel (named *effective parameter*, or *ep*). Figure 9.5 shows a schematization of the composition.

In the simplest case, the fp is empty, thus the result is a union with *ep*. More in general, fp can be made of one of several metaclasses of the metamodel, which could or could not be edge metaclasses. Hence the need to apply more advanced composition techniques. We will not get too deep into the details of this, as it is not in the scope of this thesis. Let's just say that composition of metamodels has a few base cases:

1. Composition (union) of disjoint metamodels

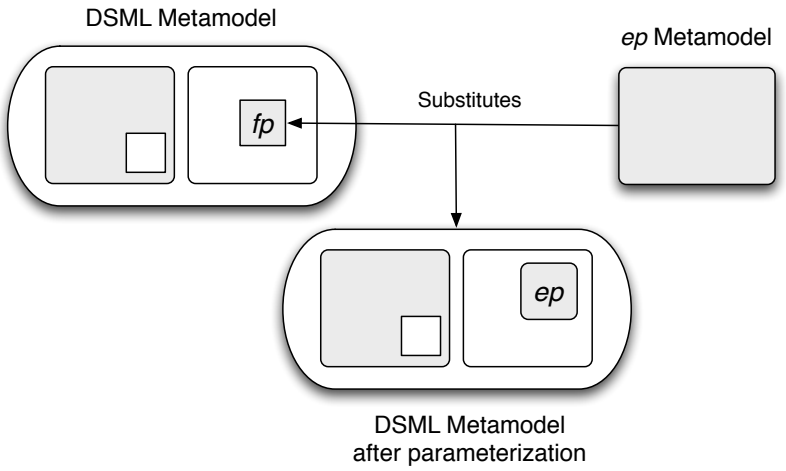

Figure 9.5. Schematization of metamodel composition

2. Composition of metamodels with refinement of edge metaclasses

3. Composition of metamodels with refinement of non-edge metaclasses

The first case is trivial. The second case is relatively simple, as long as one takes care of not causing naming conflicts and keeping relationships intact. The third case is more delicate, as there are a certain number of cases in which composition can introduce ambiguities or contradictions. Just for the sake of an example, Figure 9.6 shows the metamodel for the core part of Cospel, and Figure 9.7 shows a simplified metamodel for the events part. Composing this two metamodels involves substituting the `Event` and `Transition` metaclasses of the Cospel core metamodel (acting as fp) with the `Event` and `Transition` metaclasses of the events metamodel, which bring along the `Condition` metaclass (all acting as ep). Figure 9.8 shows the result of the composition. This is a simple example of the third type of composition (as the `Transition` metaclass is a non-edge metaclass) and represents the actual technique we used to build the complete Cospel metamodel. This and other case studies are present in (Pedro, 2009), where other types of compositions are discussed.

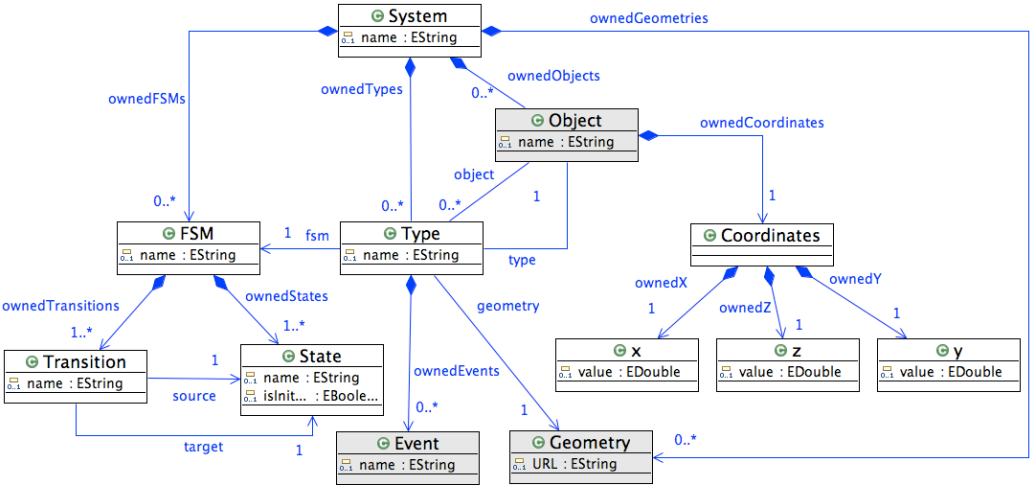

Figure 9.6. Generic Cospel metamodel

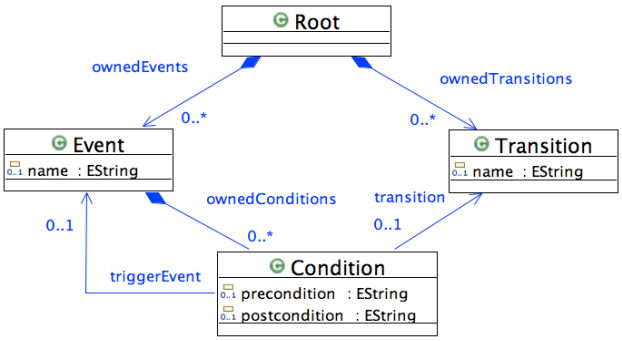

Figure 9.7. Cospel Event metamodel

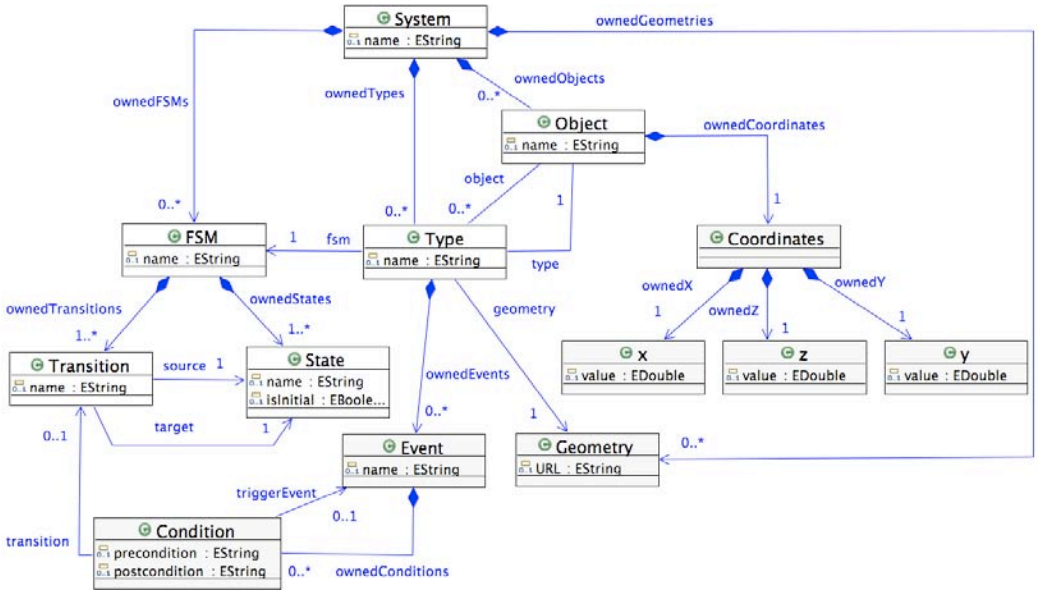

Figure 9.8. Event Extension of Cospel metamodel

9.5 Properties in a language

A central point when using models is establishing what properties we expect from the model. A property is a proposition generally expressed in some kind of logic or constraint language, for which we can assess a satisfiability value with respect to a model. We then say that the model *satisfies* the property.

Properties can be of many different kinds. In the case of Cospel we saw how there were "structural" properties, like "a type is always associated to an FSM" for the FSM package (page 71). These are sometimes explicit in the metamodel; in this example, the `fsm` association between the `Type` and `FSM` metaclasses has a cardinality of `1..1`, which explicitly states that the association is mandatory.

However, other properties are not directly derived from the metamodel, and might have to do with additional structural constraints we impose, or on the semantics we want to associate to our language. An example of additional structural constraint is "The `fromState` and `toState` of a transition belong to the same FSM as the transition itself" (page 71). No relationship in the metamodel states this, and according to the metamodel we could as well make transitions going from a state in one FSM to a state in another FSM.

An example of semantics-related property is "An FSM is not deadlocking" (page 71). This is not a property we can verify at the syntactic level, and requires examining the state space of the FSM to assess satisfiability.

Collecting properties of a language is fundamental for several activities. First, it allows us to establish well-formedness rules for our models. This enables checking if the model is built correctly. Then, it allows enumerating properties for dynamic model checking. As a matter of fact, choosing the right technique for model checking is a consequence of knowing what kind of properties we wish to verify. In our case study we checked reachability properties, which lead us to a tool which is very efficient for this kind of checks. However, should we wish to check temporal logic formulae, a different choice would be warranted.

The activity of listing the properties is best lead in the language engineering phase. The reason is that, just like use cases, properties are information that should only be concerned with the domain's level of abstraction, and not be biased by low-level technological aspects. However, the process of establishing desired properties can (and indeed could often be) iterative. Technological choices might not turn out to be able to verify some of the properties; or additional properties might become evident at later times. This is a very delicate aspect, as it could entail the insufficiency of the whole framework with fundamental properties verification.

The lesson learned here is that adequate time and resources should be spent collecting properties, to avoid expensive later corrections. A good source for properties are the experts of the type of systems we want to model. They are, in the end, those who have to be satisfied, so their point of view on the properties is the one which has to be taken into account the most. For example, the list of properties for the Cosmic Rack in Section 1 (page 45) are the main ones to take into account, and they all have to be covered in some way by the properties of the language being engineered. For example, Cosmic Rack property "An object should never get blocked in a given state" is covered by the FSM package properties "An FSM is not deadlocking" and "a type is always associated to an FSM", which through the object-type-FSM architecture apply to every object.

9.5.1 Properties and composition

A less-than-trivial aspect of establishing properties in a language is how to deal with composition. We saw how in Cospel some of the properties are

local to each language package, but some others only apply if we consider the composition of packages. We could not say anything about state propagation properties (page 73) if we consider the State dependency rules package in isolation. So, basically, these properties don't apply to models conforming to the State dependency rules package, but do apply when to the composition of this package with the Core and FSM packages.

The Cospel case study was relatively easy under this point of view; cross-package properties were simply added to local properties. There might be cases however in which the composition of two metamodels creates difficult situations. Figure 9.9 shows an example for this.

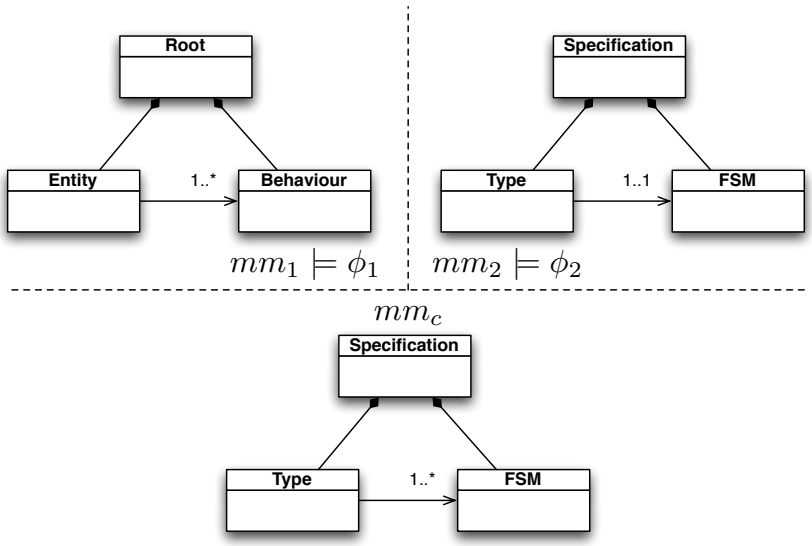

Figure 9.9. Property problem with metamodel composition

Metamodel mm_1 represents generic entities having generic behaviour. A property ϕ_1 (satisfied by construction in this metamodel) states that an entity might have one or several possible behaviours. Metamodel mm_2 represents the type-FSM architecture we used in Cospel. A property ϕ_2 states that each type must have one and one only associated FSM. Now, if we compose metamodels mm_1 and mm_2, depending on the composition strategy we use we might end up with mm_c (a list of possible composition strategies are given in (Pedro, 2009)). Note the cardinality of the association relationship. Now, even *mutatis mutandis*, we see that property ϕ_2 does not hold for models conforming to mm_c. This could or could not be acceptable according to the

specific case, and it is something that should be studied carefully whenever a metamodel composition is envisaged.

9.6 Transforming metamodels

The BATIC^{3}S methodology relies on transformation to give semantics to Cospel and to obtain an executable simulator as well as the database for the GUI.

Metamodel-based transformations are a rather powerful tool in which they allow reducing the quantity of work to build a complete framework. The general principle is the well-know "not reinventing the wheel". We found that the best practice in this case is that, even before starting to engineer the DSML, one should look for a low-level formalism that is flexible and powerful enough to achieve the desired goals. It does not have to be easy to use - in our case, we adopted CO-OPN (and APNs at a lower level), which are definitely not languages for a generic end user. The important point is that their semantics should satisfy one's needs.

Once this choice is done, an efficient way of working is defining the abstract syntax of a DSML and use transformation to achieve a twofold objective: giving semantics to the DSML, and hiding from the user the actual complexity of the low level formalism. This means that at the same time, the user can use the power of the low-level formalism while being constrained in its use by the DSML syntax (thus eliminating a large number of possibilities to make a incorrect specification, which is very easy with a complex formalism).

9.6.1 Transformation and composition

Given the fact that the methodology we propose in this thesis uses transformation to give semantics to our languages, a natural consequence of metamodel composition is that transformations have to be composed too. If a metamodel composition of metamodels $mm1$ and $mm2$ takes care of giving a unified abstract syntax, nothing is done at the level of the semantics unless we compose transformations as well.

This is actually the most delicate part, as even more ambiguities can happen at this level than at the syntactical level. Again, the work in (Pedro, 2009) did a study on composing transformations. This is based on refining

transformation rules associated to the metaclasses of fp with those associated to the metaclasses of ep. The complete formalization of this is rather complex, and again is out of the scope of this thesis. We will only give an abstract definition here. We call $Tr_{fp \to mmt}$ the set of transformation rules which transform the fp metamodel to a target metamodel mmt. This is made of several indexed rules $Tr^i_{fp \to mmt}$, each of which takes care of transforming one or more of the metaclasses in fp. Likewise, $Tr_{ep \to mmt}$ is the set of rules transforming metaclasses of ep to mmt (and again, it is made of indexed classes $Tr^i_{ep \to mmt}$). A function ψ can be defined that maps a (set of) rules in $Tr_{fp \to mmt}$ to a (set of) rules in $Tr_{ep \to mmt}$, meaning that the ep rules will refine the fp rules. Figure 9.10 shows this for the simplest case of edge metaclasses composition: the second fp rule, $Tr^2_{fp \to mmt}$, is substituted by the set of ep rules. Figure 9.11 shows the result of the transformation composition. More use cases and concrete code example are found in (Pedro, 2009).

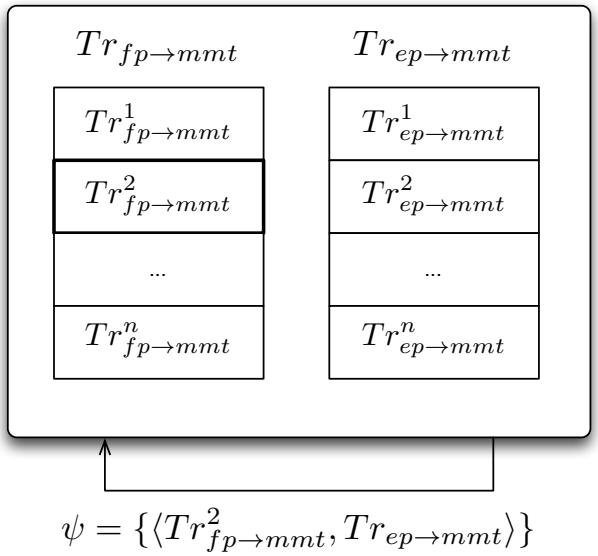

Figure 9.10. Transformation Composition for Edge Elements - Before

Aspects that have to be taken care of when composing transformations mainly stem from a potential loss of information when composing rules for non-edge metaclasses. In the Cospel event composition example we saw earlier, we can't just replace the transformation rules for the `Transition` metaclass in fp with the rule for the `Transition` metaclass in ep. The reason is that the latter does not process the `source` and `target` relationships that

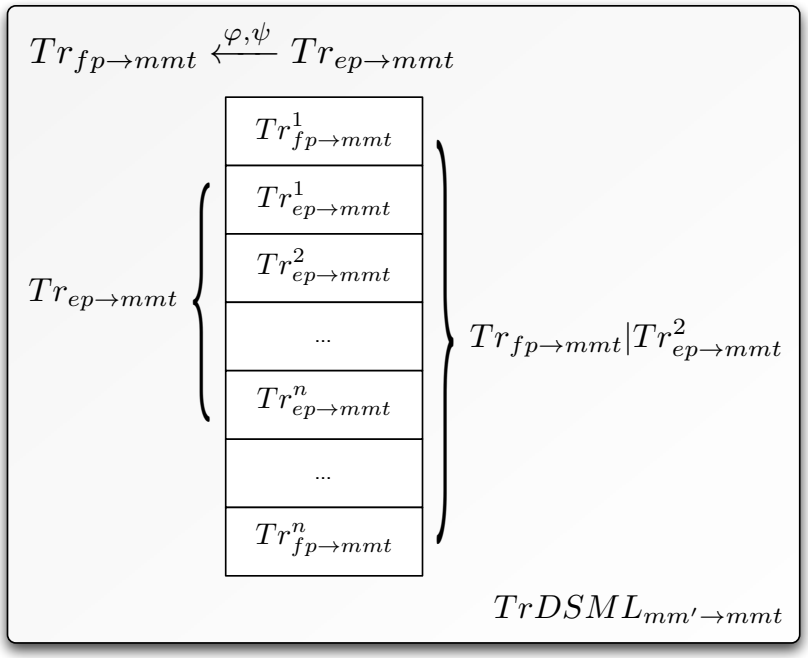

Figure 9.11. Transformation Composition for Edge Elements - After

the former does. If we simply substitute the *ep* for the fp, these relationships are lost and the transformation is broken. Strategies exist (and are detailed in (Pedro, 2009)) for solving (or at least mitigating) this problem. However, we feel the technique is still not completely automatic, in that a language engineer should probably intervene to solve a few conflict that cannot be automatically tackled. Since the language composition however is not an activity that has to be carried out too often in the course of the methodology, this is still an acceptable limit.

9.6.2 Transformation and properties

The impact of transformation on properties is another delicate subject.

Let m be a model conforming to a metamodel mm, such that $m \models \phi$, where ϕ is a property.

Likewise, let m' be a model conforming to a metamodel mm', such that $m' \models \phi'$, where ϕ' is a property.

Let also $t_{mm \to mm'}$ be a transformation such that $t_{mm \to mm'}(m) = m'$.

The problem consists in answering the following questions:

- is there a way (that can be automated) to establish a transformation relationship $\tau_{\mathcal{P}}$ between ϕ and ϕ' s.t. $\tau_{\mathcal{P}}(\phi) = \phi'$, given the $t_{mm \to mm'}$ transformation?

- even if a transformation relationship $\tau_{\mathcal{P}}$ can be found, and one manages to prove that $m' \models \phi'$, what conclusions does this draw about $m \models \phi$? In other words, how can one ensure that $m' \models \phi' \implies m \models \phi$?

In the Cospel case study, we encountered this problem when comparing model checking on the transformed APN models against the original CO-OPN model. The solution we found is indeed rather ad-hoc: properties have been hand-translated, and the problem of interpreting verification results on transformed models has been solved by studying the relationship between the CO-OPN and APN transition systems. However, establishing such a relationship could be very hard or even unfeasible, so the transformation approach must be used with a lot of caution when tackling model checking.

We can give a perspective suggestion of a general idea to try and improve the results in this field. The methodology illustrated in Figure 9.12 can be imagined. Once the domain has been studied (and thus domain concepts and properties are clear), one should find a decomposition of the concepts in

different semantic domains (time, probability, object-orientation, ...). This should be accompanied by the design of a property language that can tackle all the semantic domains that are being used. So, the user should be able to express the model *and* the properties using the provided DSML.

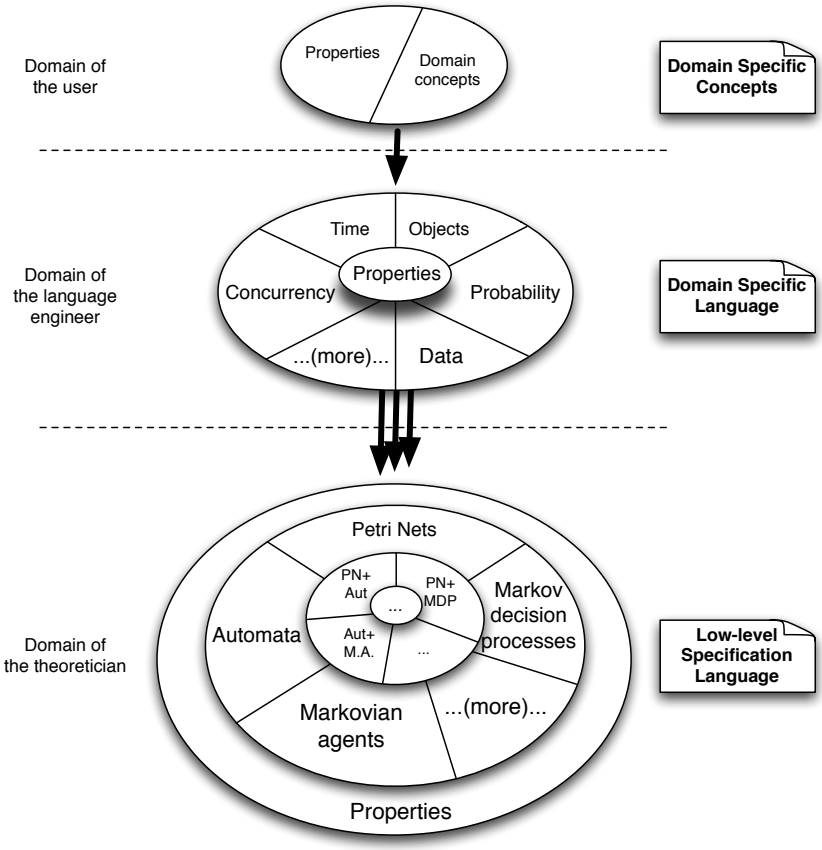

Figure 9.12. From concepts to low-level specfication

Then, the transformation phase should probably work in a modular way so that appropriate low-level formalisms are used to translate the different semantic aspects. This should include transforming properties in a respective appropriate formalisms: for example, properties concerning time can be transformed into timed logic, concurrency properties can be translated to Petri nets invariants, and so on. Also, if information is lost while transforming from a higher level to abstraction to a lower one, this information should be kept in an appropriate usable format. In this way, a multi-formalism low-

level specification can be envisaged that allows verifying different semantic aspects of the specification. Thanks to the information kept with each transformation step, the results of the verification can then be back-translated into the DSML domain. This is an open research point for the moment, and one of the most interesting perspectives of this work.

9.7 Summary and conclusion

This chapter concludes part III of this document. We went through the technical details of our specification language, and saw how it is transformed to achieve GUI prototyping and system simulation. We then gave an overview of the model checking and verification techniques that we applied to the methodology. Finally, this last chapter abstracted the methodology to a more general point of view on DSML development and the related issues and difficulties.

The next and final part of the document starts with an overview of the BATIC^{3}S project, to give a wider panorama of where the project came from and how it was put together. After that, a chapter will draw conclusions about how the methodology illustrated in this document satisfied the initial requirements, and make some final remarks and considerations. Future perspectives for this work and a list of publications and outcomes of this thesis wrap up the document.

Part IV

Conclusion

Chapter 10

BATIC^{3}S project overview

The work in this thesis was developed in the context of the BATIC^{3}S project. This project was funded by the Hasler Foundation[1] as a part of their MMI (Man Machine Interfaces) initiative [2].

We will now give a brief overview of the project development in all its aspects.

10.1 Participating institutions

Figure 10.1 shows the geographic distribution of the institutions that took part to the BATIC^{3}S project. Participants were distributed in three places:

- **Geneva, Switzerland**: The University of Geneva and the University of Applied Sciences for Landscaping, Engineering and Architecture (HEPIA – formerly Geneva School of Engineering);
- **Sierre, Switzerland**: The University of Applied Sciences of Valais;
- **Lisbon, Portugal**: Universidade Nova de Lisboa.

A total of five research groups were involved (one per institution, two at the University of Geneva) with different competences.

[1] `http://www.haslerstiftung.ch`
[2] `http://www.haslerstiftung.ch/eng/ausschreibung_mmi.html`

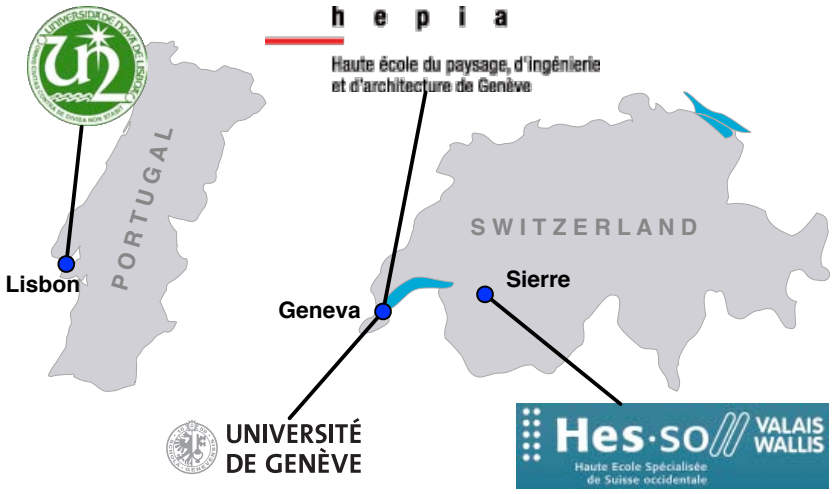

Figure 10.1. Participating institutions

10.2 Origins of the project

The original project stemmed from the author's previous work as an employee at CERN. He was directly involved in the development of a user interface for the construction database of the CMS tracker. This database contained the whole hierarchy of components, with many of their parameters and test data. During this period, he could see that a certain number of databases existed, each with different information: geometry, control, engineering, and so on. Much of the information was duplicated, and most of the databases were incompatible with each other. Users were suffering this fragmentation, as it seemed natural to integrate this information in a unified data source. The author helped coordinating an initiative to do an *a posteriori* integration of all the data. This unfortunately failed for two reasons: the most apparent was the difficulty of convincing people to switch from their own representation of data to a new, more general one. The real reason however was lack of resources, as an integration required a substantial development effort for translating the databases, adapting the applications and training the users.

10.2.1 Information fragmentation

The consideration that started the BATIC^{3}S project was that if a unified model could be produced from the start, this would facilitate the interaction

among groups working on different aspects of the system. The most evident problem was that most groups wanted to use their own jargons when speaking about the system. Proposals for a neutral, high-level model of the system were rejected on the basis of the formalisms they used – namely, most proposals were database schemas. This did not take into account the fact that physicists, control engineers and operators did not want (or need) to understand how to model their problem in a database schema. This representation was way too low-level for them. It failed to express their understanding of the concerns they were treating.

The observation of this status quo led to the realization of the need for a user-oriented representation of the problem domain. This led to the idea of Domain Specific Languages coming into play.

10.2.2 Developing interfaces

Another observation was that the development of control user interfaces for the Tracker was being performed by hand, with a lot of repeated work and wasted time following system and requirements evolution. Prospective users were also complaining about the fact that it was difficult to visualize the system using the available interfaces. These were in fact based on panels and buttons, and had no visual representation of the system in its entirety, let alone in the three-dimensional aspects. A development effort was made for integrating a 3D visualization in the system; however this did not reduce the development effort. Also, a lacking feature of the control system was an automated generation of a simulator to evaluate interfaces without the availability of the actual system. Simulators were coded manually, which added to the effort.

10.3 Integrating competences

The BATIC^{3}S workforce was chosen for their expertise around several axis of research.

- **Modeling and Verification:** the SMV group of Prof. Didier Buchs at the University of Geneva, where this thesis was developed, has several years of experience with modeling concurrent reactive systems using formal methods. This led in recent years to the development of

techniques for test generation, model transformation and automated model checking. This group provided the knowledge for developing the DSL-based methodology and the transformation workflow. In a second phase of the project SMV joined by the group of Prof. Vasco Amaral at the Universidade Nova de Lisboa, with previous work on domain specific languages for physics experiments.

- **User and Task representation:** the ISI group of Prof. Gilles Falquet at the University of Geneva is active in the field of man-machine interaction and knowledge representation. Their research focused, among other interests, on users and usability, especially in 3D. This brought a vast knowledge on how to model the interaction tasks in a three-dimensional interface. On the other hand, Prof. Anne Le Calvé from the University of Applied Sciences of Valais provided previous experience on ontologies for representing user knowledge, like GenOUM.

- **Adaptation:** this was also another original goal of the project. An interface can evolve and change its behaviour when the circumstances evolve. For example, it can react to errors by changing its point of view to help the user take action. In the end, not much was done in this regard, as priorities led the project in other directions; however, the ISI group provided some perspective suggestions on how to implement a rule-based adaptation system for 3D interfaces.

- **3D visualization and stereoscopy:** Prof. Stéphane Malandain from the University of Applied Sciences for Landscaping, Engineering and Architecture provided the expertise on 3D modeling, visualization and navigation. The responsibility of his group was the development of the 3D engine and the interaction techniques. Also he took care of deploying a stereoscopic version of the interface.

10.4 Other case studies

During the BATIC^{3}S project, we applied several iterations of the methodology to other case studies. We will discuss here the two main ones, a Drink Vending Machine and the ATLAS Trigger/DAQ system.

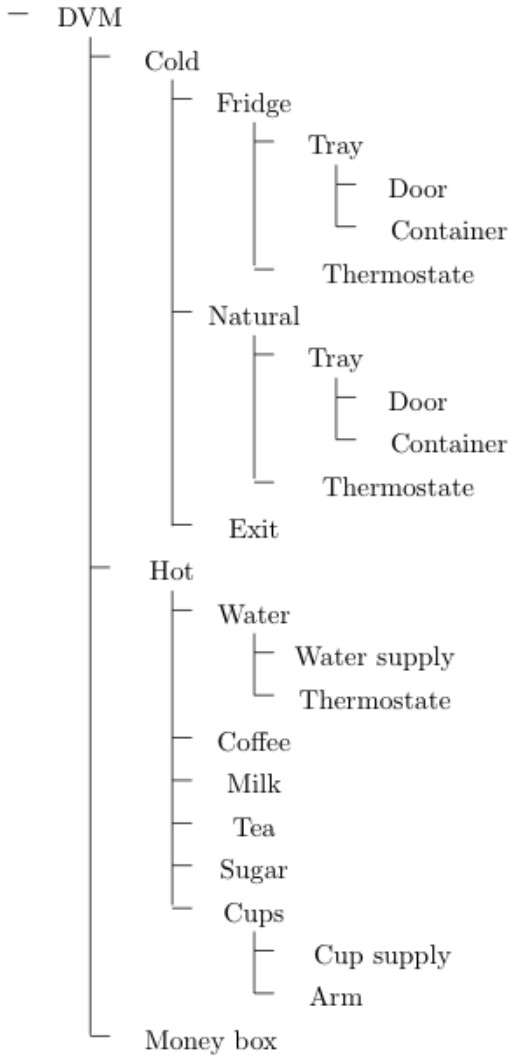

Figure 10.2. DVM hierarchy

10.4.1 Drink Vending Machine

While developing the methodology, we looked for a reasonable system model to run our tests. We thought that it would be better to start with a different model than the Cosmic Rack. This was for two reasons: we wanted to first concentrate on the methodology rather than use a lot of time to faithfully model an actual complex system; and we wanted to avoid being too focused on the high-energy physics field, by working on a different type of system. We defined a specification for a Drink Vending Machine (DVM) (chapter 1 of (BATIC^{3}S collaboration, 2006)). Figure 10.2 shows its object hierarchy.

This case study had all the features we wanted. A hierarchy of objects with individual behaviors, the state of which were dependent from the states of their sub-objects; inputs and outputs for many of the objects; and a classification of users and use cases. We also defined a geometrical model for it.

We initially built the model with a then unnamed textual specification language, then redid the work with Cospel when the language conception was refined. Then we explored the problem of how to transform it to CO-OPN. We first established what the structure of the CO-OPN specification should be. Four iterations of the CO-OPN model were hand coded, until we were satisfied that the model was reflecting the intended semantics for the Cospel model. Finally, we wrote the transformation rules that took the Cospel DVM specification (or any Cospel specification) and transformed it to its CO-OPN equivalent.

Once this was done, we tackled the filling of the database for the GUI. We wrote another ATL transformation which extracted relevant information from the Cospel specification and created a list of SQL statements to create and fill a database. Running these statements on a MySQL server gave us the database artifact.

All along this development phase, the GUI engine was also being created at HEPIA. A codebase was created to render the object hierarchy with associated states and geometry using Java OpenGL bindings (JoGL expert group, n.d.). A generic driver for communicating with a CO-OPN simulator was written, as well as a skeleton driver for facilitating communication with an hypotetical real system (we did not have an actual DVM to test this).

As part of the research on 3D interfaces, we built a setup for stereoscopic projection of the interface. This used dual overhead projectors and different polarized filters in front of each projector. The engine created two different

points of view for the interface, one for each eye, and sent them to the two projectors. These were aligned to be superimposed on the same screen, which viewed through polarized spectacles gave a perception of depth and distance. However, we found that this particular setup was not adding much in terms of intuitive manipulation. One of the main reasons was probably that the engine did not support free hand gestures and had to be controlled with traditional input devices like mice. Current experimentation is undergoing to add intuitive multiple touch interaction to the engine to see if this improves the immersive experience.

This first case study produced our first complete prototype, shown in Figure 10.3. This was followed by the Cosmic Rack modeling, during which

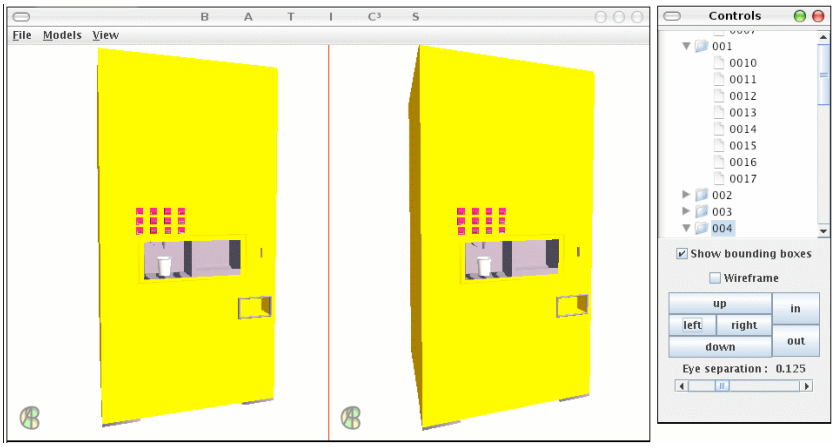

Figure 10.3. Screenshot of the DVM prototype

we found out that some system concerns were missing from the language. This led us to the development of a modular structure for the language. We thus joined other researchers in studying how to achieve modularity for metamodel-based DSL definition and transformation (Pedro, 2009; Pedro, Risoldi, Buchs, Barroca, & Amaral, 2009).

10.4.2 The ATLAS Trigger/DAQ system

One aspect of our methodology that we wanted to validate was how it would apply to control systems different from the one we were mainly tackling. Namely, we asked ourselves how the methodology would fare if the system did not have a physical reality, but were instead a software system. Also,

we wanted to try and apply the same methodology to the development of classical 2D interfaces.

This was achieved by working with another CERN experiment called ATLAS. This particle detector has a software system which controls the data acquisition applications for the machine. Applications have different states (stopped, running, error...) and are organized into a hierarchy. An operator can monitor and control the tree of active applications using a panel with a 2D representation of the tree. Colours and labels are used to represent states.

The specification for the system was done using the (H)ALL language (Barroca et al., 2008). This is another approach to the modeling activity led at the Universidade Nova de Lisboa, including in the language visual and behavioural aspects that Cospel abstracts from.

The resulting prototype is shown in Figure 10.4 and reproduces quite faithfully the hand-coded panel that was being used in production at ATLAS.

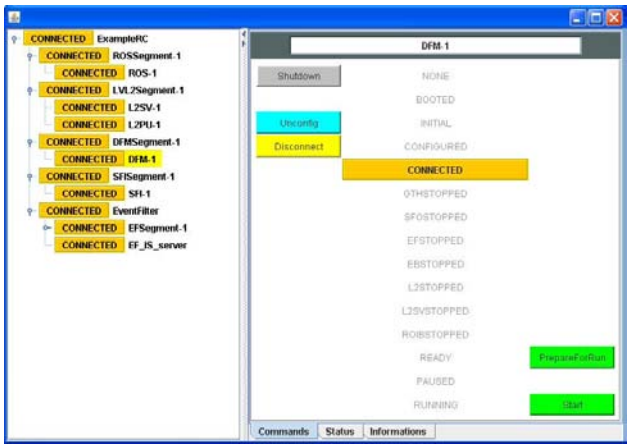

Figure 10.4. Screenshot of the ATLAS Trigger/DAQ prototype

Chapter 11

Conclusions

11.1 Satisfying requirements

In Section 1.5 (page 20) we outlined the requirements we had for the methodology proposed in this thesis. Table 11.1 resumes them and states how they have been satisfied.

Type of requirements	**Requirement**	**Satisfied by**
User-related requirements	Domain Specificity	EMF, Metamodeling techniques
	Abstraction	DSL approach
	Evolution	metamodeling + transformation, modular approach
	Simulation	ATL+CO-OPN, code generation, GUI engine
Technical requirements	Integration	Eclipse, EMF and associated ecosystem
	Platform independency	Being based on Eclipse, Java
Methodological requirements	Transformation	ATL+EMF
	Validation & Verification	CO-OPN, ATL, OCL, Java checks
	Automation	EMF, ATL, GUI engine

Table 11.1. Satisfied requirements

11.2 Summary and final considerations

This thesis presented a methodology to prototype GUIs for interactive systems using DSMLs and model transformations. To illustrate the methodology, we applied it to the control systems domain. A DSML called Cospel, based on useful and understandable abstractions for control systems experts, was designed for specifying systems. Transformation techniques and semantic mapping to a formal model allowed for simulation, validation, verification and automatic GUI prototype generation.

The main advantages of the presented work with respect to classical techniques for developing GUIs for interactive systems are:

- a more comprehensible language for system specification and data integration
- the possibility to generate a system simulator almost for free to evaluate GUIs
- allowing validation and verification of the system specification
- a rapid prototyping of the GUI, allowing development effort to focus on more specific issues

More in general, we think that the contribution of this work was a clearer and more concrete definition of the role DSLs and formal models can play in model-based development. In our multiple case studies, the experts we worked with found that this development methodology was cleaner and richer than the way things were being done (Masetti, Hartmann, Shah, & Stringer, 2008), and offered good perspectives in order to improve current practices.

Of course, there were a few aspects of this work in which we found shortcomings. The resources we had did not allow us to deliver a polished, commercial-grade framework. Although effort had been done to hide most of the low-level details from the user, some of the formal underpinnings are still visible. This is solvable with a further development effort which should focus on the user experience.

Another limit we found is more conceptual in nature, and lies in the fact that model-driven development is still not as widely accepted in industry as it is in academia. With deadlines, strict requirements and limited workforce, the average project manager still has a penchant for the traditional development techniques she knows, rather than investing in training for a new

paradigm. This is especially true for the small and medium sized companies. Although this is slowly changing, we think more concrete case studies and visible collaborations between academia and industrial subjects are needed to promote these approaches.

More on the formal side of things, there is still a lot of work to do with respect to the model checking activity. When a Cospel model is transformed to a CO-OPN model, *and then again* to an APN to perform model checking, one naturally might ask what is the significance of the model checking results with respect to the original Cospel model. In fact, it is not trivial to ensure that verifying that a property is satisfied by the APN model automatically implies that it is satisfied by the CO-OPN model, much less by the Cospel specification. One one hand, one has to completely rely on the correctness of the model transformations to assert that the original model is correct when the transformed model is. On the other hand, let us suppose that one discovers a state in the transformed model that violates the property; how can one track that state back to the original model? In other words, what is the state of the Cospel model that constitutes the counterexample of that property check?

Finally, talking more about the tools and techniques, Eclipse has been both empowering and limiting for this work. On one hand, it is a completely open framework, and the fact that most technologies related to MDE exist as an Eclipse plugin facilitated immensely the task of integrating tools. On the other hand, Eclipse is a very complex creature. Its architecture sometimes changes from one release to another, and we found ourselves having to do a lot of additional work to make our tools compatible with a new Eclipse version (we started working with Eclipse 3.2, and the current release is 3.5). At several points in time, our framework stopped being operational because of a change in architecture, an incompatibility of a plugin, or a previously silent bug which was unleashed by an update. This, coupled with the inevitable evolution of development platforms, makes an Eclipse-based framework difficult to call reliable, and demands a certain amount of expertise for maintenance. Which, in some way, contradicts some of the original goals of this work, which were to simplify development. As a matter of fact, while writing this lines, the framework is not able to run on the same machine we mainly used for its development, because of a change in architecture from 32 to 64 bits (and JOGL is not compatible with this architecture).

11.3 Perspectives

As is the case for many academic works, there are things which were not completed, left undone, or unexplored. Some of the perspectives that could define the future of this work are:

- **Improving the model checking methodology:** this includes solving the problem of interpretation of properties checking on low-level with respect to the original model, and achieving something similar to what was shown in Figure 9.12 (page 172).

- **Improving the language:** the modular structure we used for metamodels and transformations allows for a relatively easy extension of Cospel. Features can be added, such as defining a type-based template system for the hierarchy or specifying the behaviour of commands in a more complex way, or again modeling the interaction between the GUI and the system.

- **Enriching user information:** the user model we have defined is actually richer than what we currently use. By implementing metrics in the GUI engine, we could measure factors like user mood, learning style and cognitive style based on how the user interacts with the GUI. This information could be used for evaluation of the GUI prototype.

- **3D adaptation:** apart from open issues of defining procedural versus declarative 3D adaptation, work is ongoing on defining ontology-based 3D adaptation. We are currently investigating a case study for building 3D adaptive GUIs based on ontologies for urban planning communication (Métral, Falquet, & Vonlanthen, 2007). Urban planning is concerned with assembling and shaping the urban, local or municipal environment by deciding on the composition and configuration of geographical objects in a space-time continuum. The main characteristics of the ontology-based model are the semantic integration in a knowledge base of the urban knowledge coming from various sources (geographical information systems databases, master plans, local plans); and the modeling of the centre of interest of an urban actor. These models can be then used to generate adapted GUIs to present the project's data

and knowledge according to each actor's background and interests.

- **3D stereoscopy impact:** we want to evaluate if, in this domain, stereoscopy helps navigation; if immersion is relevant to knowledge representation; if there are unexpected side effects of using a 3D interactive environment; and if there is an advantage in using haptic devices or multitouch interaction.

- **Introducing ergonomy and usability criteria:** while our current work is mainly interested in the semantic and methodological aspects of GUI generation, one should not forget that usability and ergonomy of a GUI are capital factors in its success. Existing approaches for applying standard usability metrics to GUIs should be integrated in the prototyping process.

11.4 Outcome of this work

Apart from this document, the work produced a number of achievements in terms of publications and student projects.

11.4.1 Articles in international peer-reviewed venues and book chapters

1. Ang Chen, Didier Buchs, Levi Lucio, Luis Pedro, and Matteo 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, MOCA'06, ed. by Department of computer science, University of Hamburg, vol. FBI-HH-B-272, pp. 103-122. (Chen et al., 2006)

2. Matteo Risoldi and Vasco Amaral, (2007) *Towards a formal, model-based framework for control systems interaction prototyping* In: Rapid Integration of Software Engineering techniques, ed. by Nicolas Guelfi and Didier Buchs, vol. 4401, pp. 144-159, Springer-Verlag. LNCS. (Risoldi & Amaral, 2007)

3. Matteo Risoldi and Didier Buchs (2007) *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), 23-27 September 2007, USA, pp. 179-182. (Risoldi & Buchs, 2007)

4. Matteo Risoldi, Lorenzo Masetti, Didier Buchs, Bruno Barroca, and Vasco Amaral (2008) *A Methodology for Control Systems GUI Prototyping - a case study* In: Proceedings of the PCAPAC 2008 workshop. (Risoldi et al., 2008)

5. Bruno Barroca, Vasco Amaral, Pedro Calado, Matteo Risoldi, Mihai Caprini, Ana Moreira, and João Araújo (2008) *Towards the Application of a Model Based Design Methodology for Reliable Control Systems on HEP Experiments* In: Nuclear Science Symposium Conference Record, pp. 809-816, IEEE. (Barroca et al., 2008)

6. Matteo Risoldi, Vasco Amaral, Bruno Barroca, Kaveh Bazargan, Didier Buchs, Fabian Cretton, Gilles Falquet, Anne Le Calvé, Stéphane Malandain, and Pierrick Zoss (2009) *A language and a methodology for prototyping user interfaces for control systems* In: Human Machine Interaction, ed. by Denis Lalanne and Jürg Kohlas, vol. 5440/2009, pp. 223-252, Springer Berlin/Heidelberg. (Risoldi et al., 2009)

7. Luis Pedro, Matteo Risoldi, Didier Buchs, Bruno Barroca, and Vasco Amaral (2009) *Composing Visual Syntax for Domain Specific Languages* In: Human-Computer Interaction, ed. by Jacko, Julie A., vol. 5611, pp. 889-898, Heidelberg, Springer. Lecture Notes in Computer Science. (Pedro et al., 2009)

11.4.2 Student projects

1. **Bachelor thesis** (2005)
   CUI, Université de Genève
   Gaëlle Ribordy
   Drink Vending Machine model-based GUI prototyping

2. **Bachelor thesis** (2005)
   CUI, Université de Genève

Yanick Vuille
Drink Vending Machine model-based prototyping by Lego

3. **Bachelor thesis** (2008)
   CUI, Université de Genève
   Pierre Gianni
   Etude des relations topologiques pour une interface 3D

4. **Master thesis** (2008)
   CUI, Université de Genève
   Yassin Boudjenane
   Conception d'un langage de description d'interface utilisateur

5. **Bachelor thesis** (2010)
   CUI, Université de Genève
   Michael Gumowski
   Multi-touch control for 3D user interfaces

Part V

Appendices and references

Appendix A

Acronyms

ATL ATLAS Transformation Language

ADT Algebraic Abstract Data Type

APN Algebraic Petri Net

CO-OPN Concurrent Object-Oriented Petri Nets

CS Control System

Cospel COntrol systems SPEcification Language

DSML Domain Specific Modeling Language

DVM Drink Vending Machine

EMF Eclipse Modeling Framework

FSM Finite State Machine

GME Generic Modeling Environment

GMF Graphical Modeling Framework

GP Generative Programing

GUI Graphical User Interface

IDE Integrated Development Environment

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

QVT Query / Views / Transformations

SLE Software Language Engineering

SQL Structured Query Language

SVG Scalable Vector Graphics

UML Unified Modeling Language

Appendix B

CO-OPN example: a Power Group

Listing B.1 is the result of the transformation from Cospel to CO-OPN of the Power Group object in our Cosmic Rack case study.

Listing B.1. CO-OPN specification for the Power Group object of the Cosmic Rack

```
Class PowerGroup

Interface

  Use
    BlackTokens;
    States;
    Naturals;
    ObjectList;
    MethodDef;

  Type
    powergroup;

  Gates
    Astate _ : statesort;
    Acommandreq _ _ : objectname, methoddef;

  Methods
    OFF;
    ON-LV;
    ON;
    CLEAR;
    Achangestate _ : statesort;
    Agetstate;
    Ainitializestate;

Body

  Methods
    OFF-TO-ON-LV;
    ON-LV-TO-ON;
```

```
    ON_TO_ON_LV;
34  ON_LV_TO_OFF;
    ERROR_TO_OFF;
36  ERROR_TO_ON_LV;
    OFF_TO_INTERLOCKED;
38  ON_LV_TO_INTERLOCKED;
    ON_TO_INTERLOCKED;
40  ERROR_TO_INTERLOCKED;
    INTERLOCKED_TO_OFF;
42  AimplementationON;
    AimplementationOFF;
44  AimplementationON_LV;
    AimplementationCLEAR;
46


48
  Places
50  ON _ : blackToken;
    ON_LV _ : blackToken;
52  OFF _ : blackToken;
    ERROR _ : blackToken;
54  INTERLOCKED _ : blackToken;
    temperature _ : natural;
56
  Initial
58  OFF @;
    temperature 0;
60
  Axioms
62  OFF_TO_ON_LV ::
      OFF @ -> ON_LV @;
64  ON_LV_TO_ON ::
      ON_LV @ -> ON @;
66  ON_TO_ON_LV ::
      ON @ -> ON_LV @;
68  ON_LV_TO_OFF ::
      ON_LV @ -> OFF @;
70  ERROR_TO_OFF ::
      ERROR @ -> OFF @;
72  ERROR_TO_ON_LV ::
      ERROR @ -> ON_LV @;
74  OFF_TO_INTERLOCKED ::
      OFF @ -> INTERLOCKED @;
76  ON_LV_TO_INTERLOCKED ::
      ON_LV @ -> INTERLOCKED @;
78  ON_TO_INTERLOCKED ::
      ON @ -> INTERLOCKED @;
80  ERROR_TO_INTERLOCKED ::
      ERROR @ -> INTERLOCKED @;
82  INTERLOCKED_TO_OFF ::
      INTERLOCKED @ -> OFF @;
84  this = Self =>
      setTemperature n with
86      this.AimplementationsetTemperature n ..
        this.Arecalculatestate :: -> ;
88  this = Self =>
      Agetstate With this . Astate ON ::
```

```
90                  ON @ -> ON @;
          this = Self =>
92        Agetstate With this . Astate ON-LV ::
                    ON-LV @ -> ON-LV @;
94          this = Self =>
          Agetstate With this . Astate OFF ::
96                  OFF @ -> OFF @;
            this = Self =>
98        Agetstate With this . Astate ERROR ::
                    ERROR @ -> ERROR @;
100         this = Self =>
          Agetstate With this . Astate INTERLOCKED ::
102                 INTERLOCKED @ -> INTERLOCKED @;
            this = Self =>
104             Ainitializestate With this . Agetstate:: -> ;
            this = Self & ((newstate=ON)=true) =>
106                 Achangestate newstate With this.ON-LV-TO-ON // this . Astate
                        newstate::
                     -> ;
108     this = Self & ((newstate=OFF)=true) =>
                    Achangestate newstate With this.ON-LV-TO-OFF // this .
                        Astate newstate::
110                  -> ;
        this = Self & ((newstate=OFF)=true) =>
112                 Achangestate newstate With this.ERROR-TO-OFF // this .
                        Astate newstate::
                     -> ;
114     this = Self & ((newstate=OFF)=true) =>
                    Achangestate newstate With this.INTERLOCKED-TO-OFF // this .
                         Astate newstate::
116                  -> ;
        this = Self & ((newstate=ON-LV)=true) =>
118                 Achangestate newstate With this.OFF-TO-ON-LV // this .
                        Astate newstate::
                     -> ;
120     this = Self & ((newstate=ON-LV)=true) =>
                    Achangestate newstate With this.ON-TO-ON-LV // this . Astate
                         newstate::
122                  -> ;
        this = Self & ((newstate=ON-LV)=true) =>
124                 Achangestate newstate With this.ERROR-TO-ON-LV // this .
                        Astate newstate::
                     -> ;
126     this = Self & ((newstate=INTERLOCKED)=true) =>
                    Achangestate newstate With this.OFF-TO-INTERLOCKED // this .
                         Astate newstate::
128                  -> ;
        this = Self & ((newstate=INTERLOCKED)=true) =>
130                 Achangestate newstate With this.ON-LV-TO-INTERLOCKED // this
                         . Astate newstate::
                     -> ;
132     this = Self & ((newstate=INTERLOCKED)=true) =>
                    Achangestate newstate With this.ON-TO-INTERLOCKED // this .
                        Astate newstate::
134                  -> ;
        this = Self & ((newstate=INTERLOCKED)=true) =>
136                 Achangestate newstate With this.ERROR-TO-INTERLOCKED // this
```

```
                    . Astate newstate::
                  -> ;
138     this = Self =>
            OFF With this.AimplementationOFF .. this.Achangestate OFF:: -> ;
140     this = Self =>
            ON-LV With this.AimplementationON-LV .. this.Achangestate ON-LV:: ->
                ;
142     this = Self =>
            ON With this.AimplementationON .. this.Achangestate ON:: -> ;
144     this = Self =>
            CLEAR With this.AimplementationCLEAR .. this.Achangestate OFF:: -> ;
146     this = Self =>
            AimplementationOFF :: -> ;
148     this = Self =>
            AimplementationON-LV :: -> ;
150     this = Self =>
            AimplementationON :: -> ;
152     this = Self =>
            AimplementationCLEAR :: INTERLOCKED @ -> INTERLOCKED @;
154

156   Where
        newstate : statesort;
158             prevstate : statesort;
                this : powergroup;
160     n : natural;
                m : natural;
162
End PowerGroup;
```

References

3DConnexion. (n.d.). *Spacenavigator product web page.* http://www.3dconnexion.com/3dmouse/spacenavigator.php. (visited in 2008)

Alam, S., Boccuzzo, S., Wettel, R., Dugerdil, P., Gall, H., & Lanza, M. (2009). EvoSpaces - Multi-dimensional Navigation Spaces for Software Evolution. In D. Lalanne & J. Kohlas (Eds.), *Human Machine Interaction* (Vol. 5440, p. 167-192). Springer.

ATLAS Group. (2008). *Atlas transformation language.* (http://www.eclipse.org/m2m/atl/)

Barroca, B., Amaral, V., Calado, P., Risoldi, M., Caprini, M., Moreira, A., et al. (2008). Towards the application of a model based design methodology for reliable control systems on HEP experiments. In *Nuclear Science Symposium Conference Record* (p. 809-816). IEEE. Available from http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4774652&isnumber=4774073

Bastide, R., Navarre, D., & Palanque, P. A. (2003). A tool-supported design framework for safety critical interactive systems. *Interacting with Computers*, *15*(3), 309-328.

BATIC^{3}S collaboration. (2006). *BATIC^{3}S project document collection 2005-2006* (Tech. Rep.). Université de Genève. (url: http://smv.unige.ch/research-projects/batic3s/files/YearlyReport.pdf.zip)

Batory, D., Lofaso, B., & Smaragdakis, Y. (1998). JTS: Tools for Implementing Domain-Specific Languages. *International Conference on Software Reuse*, *0*, 143.

Bézivin, J. (2005, May). On the unification power of models. *Software and Systems Modeling*, *4*(2), 171–188. Available from http://www.metapress.com/content/XN50242535640K10

ATLAS Group. (2006, February). *ATL user manual* (Tech. Rep.). Nantes, France: LINA & INRIA.

Object Management Group. (2002, October). *OMG/RFP/QVT MOF 2.0 Query/Views/Transformations RFP* (Tech. Rep.). Needham, Massachusetts, USA: OMG.

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.

Bodart, F., Hennebert, A.-M., Leheureux, J.-M., Provot, I., Sacré, B., & Vanderdonckt, J. (1995). Towards a Systematic Building of Software Architectures: the TRIDENT Methodological Guide. In P. A. Palanque & R. Bastide (Eds.), *Design, Specification and Verification of Interactive Systems '95, Proceedings of the Eurographics Workshop in Toulouse, France June 7-9, 1995* (p. 262-278). Springer.

Brabrand, C., Schwartzbach, M. I., & Vanggaard, M. (2003). The metafront system: Extensible parsing and transformation. *Electronic Notes in Theoretical Computer Science*, *82*(3), 592 - 611. Available from `http://www.sciencedirect.com/science/article/B75H1-4G6932F-6K/2/8dd9a17a4108e320df37d8082b2ef523` (LDTA'2003 - Language descriptions, Tools and Applications)

Brand, M. van den, Deursen, A. van, Heering, J., Jong, H. A. de, Jonge, M. de, Kuipers, T., et al. (2001). The ASF+SDF meta-environment: A component-based language development environment. In R. Wilhelm (Ed.), *Compiler Construction, 10th International Conference, CC 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings* (Vol. 2027, p. 365-370). Springer.

Buchs, D., Buffo, M., & Kordon, F. (2000, January). *Object-oriented modeling and analysis capabilities* (Tech. Rep. No. 2000/1). LIP6, Université Pierre et Marie Curie.

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).

Buchs, D., & Guelfi, N. (2000, july). A formal specification framework for object-oriented distributed systems. *IEEE Transactions on Software Engineering*, *26*(7), 635-652.

Buchs, D., & Hostettler, S. (2009). Managing Complexity in Model Checking with Decision Diagrams for Algebraic Petri Net. In D. Moldt (Ed.), *Pre-proceedings of the International Workshop on Petri Nets and Software Engineering* (p. 255-271). (Available at `http://www.informatik.uni-hamburg.de/TGI/events/pnse09/pnse09-proceedings-firstpages.pdf`)

Buchs, D., Hostettler, S., Marechal, A., & Risoldi, M. (2010). AlPiNA: An Algebraic Petri Net Analyzer. In J. Esparza & R. Majumdar (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings* (Vol. 6015, p. 349-352). Springer.

Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., & Grose, T. J. (2004). *Eclipse modeling framework*. Addison Wesley. Available from `http://www.eclipse.org/emf/`

Burkardt, J. (n.d.). *Object file format specification.* `http://people.scs.fsu.edu/~burkardt/txt/obj_format.txt`. (visited in 2008)

Calvary, G., Coutaz, J., & Thevenin, D. (2001). A unifying reference framework for the development of plastic user interfaces. In M. R. Little & L. Nigay (Eds.), *Engineering for Human-Computer Interaction, 8th IFIP International Conference, EHCI 2001* (Vol. 2254, p. 173-192). Springer.

Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., & Vanderdonckt, J. (2003). A unifying reference framework for multi-target user interfaces. *Interacting with Computers*, *15*(3), 289-308.

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, MOCA 2006.*

Clarke, E. M., Grumberg, O., & Peled, D. A. (1999). *Model checking*. Cambridge, MA, USA: MIT Press.

Cockburn, A., & McKenzie, B. (2001). 3D or not 3D?: evaluating the effect of the third dimension in a document management system. In *CHI '01: Proceedings of the SIGCHI conference on Human factors in computing systems* (pp. 434–441). New York, NY, USA: ACM.

Cretton, F., & Le Calvé, A. (2007). *Working paper: Generic ontology based user modeling - GenOUM* (Tech. Rep.). HES-SO Valais.

Cretton, F., & Le Calvé, A. (2008, June). *Generic ontology based user model: GenOUM* (Tech. Rep.). Université de Genève. (`http://smv.unige.ch/technical-reports`)

Czarnecki, K., & Eisenecker, U. (2000). *Generative programming: Methods, tools, and applications*. Addison-Wesley Professional.

Czarnecki, K., & Helsen, S. (2003). Classification of model transformation approaches. Available from `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=?doi=10.1.1.122.8124`

de Lara, J., & Guerra, E. (2005). Formal support for model driven development with graph transformation techniques. In A. Estévez, V. Pelechano, & A. Vallecillo (Eds.), *Actas del Taller sobre Desarrollo Dirigido por Modelos, MDA y Aplicaciones, Granada, España* (Vol. 157). CEUR-WS.org.

de Lara, J., & Vangheluwe, H. (2002). AToM3: A Tool for Multi-formalism and Meta-modelling. In *FASE '02: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering* (pp. 174–188). London, UK: Springer-Verlag.

de Lara, J., Vangheluwe, H., & Alfonseca, M. (2004). Meta-modelling and graph grammars for multi-paradigm modelling in AToM3. *Software and System Modeling*, *3*(3), 194-209.

Dierlamm, A., Dirkes, G. H., Fahrer, M., Frey, M., Hartmann, F., Masetti, L., et al. (2008). The CMS tracker control system. *Journal of Physics: Conference Series*, *119*(2), 022019 (9pp). Available from `http://stacks.iop.org/1742-6596/119/022019`

FengGUI developer group. (n.d.). *FengGUI: Java GUIs with OpenGL.* `http://www.fenggui.org`. (visited in 2008)

Fleurey, F., & Solberg, A. (2009). A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems. In A. Schürr & B. Selic (Eds.), *Model driven engineering languages and systems, 12th international conference, MODELS 2009* (Vol. 5795, p. 606-621). Springer.

Greenfield, J., Short, K., Cook, S., & Kent, S. (2004). *Software factories: Assembling applications with patterns, models, frameworks, and tools*. Wiley. Paperback.

Henriques, P. R., Pereira, M. J. V., Mernik, M., Lenic, M., Avdicausevic, E., & Zumer, V. (2002). Automatic generation of language-based tools. *Electr. Notes Theor. Comput. Sci.*, *65*(3).

Huth, M. R. A., & Ryan, M. (2000). *Logic in computer science: modelling and*

reasoning about systems. New York, NY, USA: Cambridge University Press.

IBM. (2010a). *Rational Software Architect for WebSphere Software.* (`http://www-01.ibm.com/software/awdtools/swarchitect/websphere/` (Visited in June 2010))

IBM. (2010b). *Rational Tau.* (`http://www-01.ibm.com/software/awdtools/tau/` (Visited in June 2010))

IEEE. (1991). IEEE Standard Computer Dictionary. A Compilation of IEEE Standard Computer Glossaries. *IEEE Std 610*, 1.

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`

JoGL expert group. (n.d.). *JSR 231: Java^{TM} binding for the OpenGL® API.* `http://jcp.org/en/jsr/detail?id=231`. (visited in 2008)

Jouault, F., & Bézivin, J. (2006). KM3: A DSL for Metamodel Specification. In R. Gorrieri & H. Wehrheim (Eds.), *Formal Methods for Open Object-Based Distributed Systems, 8th IFIP WG 6.1 International Conference, FMOODS 2006, Bologna, Italy, June 14-16, 2006, Proceedings* (Vol. 4037, p. 171-185). Springer.

Karlsch, M. (2007). *A model-driven framework for domain specific languages.* Unpublished master's thesis, Hasso-Plattner-Institute of Software Systems Engineering, Potsdam, Germany.

Kleppe, A. G., Warmer, J., & Bast, W. (2003). *MDA explained: The model driven architecture: Practice and promise.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Kobsa, A., & Wahlster, W. (Eds.). (1989). *User models in dialog systems.* New York, NY, USA: Springer-Verlag New York, Inc.

Kurtev, I., Bézivin, J., Jouault, F., & Valduriez, P. (2006). Model-based DSL frameworks. In P. L. Tarr & W. R. Cook (Eds.), *OOPSLA companion* (p. 602-616). ACM.

Masetti, L., Hartmann, F., Shah, S. Y., & Stringer, R. (2008, 10). The CMS Tracker Detector Control System. In *11th IEEE Nuclear Science Symposium, Proceedings.* IEEE. (URL `http://www.nss-mic.org/2008/NSSMain.asp`)

Mens, T., & Gorp, P. V. (2006). A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, *152*, 125 - 142. Avail-

able from http://www.sciencedirect.com/science/article/B75H1-4JFR8K3-B/2/cd561440d16d44082d7b63da61c70252 (Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005))

MetaCase Consulting. (2010). *Metaedit+.* (http://www.metacase.com (visited in 2010))

Métral, C., Falquet, G., & Vonlanthen, M. (2007). An ontology-based model for urban planning communication. In J. Teller, J. R. Lee, & C. Roussey (Eds.), *Ontologies for urban development* (Vol. 61, p. 61-72). Springer.

Microsoft. (2008). *Domain-Specific Language Tools.* Available from http://msdn2.microsoft.com/en-us/vs2005/aa718368.aspx (http://msdn2.microsoft.com/en-us/vs2005/aa718368.aspx (Visited in 2008))

Moussa, F., Kolski, C., & Riahi, M. (2000). A model based approach to semi-automated user interface generation for process control interactive applications. *Interacting with Computers*, *12*(3), 245-279.

Mössenböck, H., Löberbauer, M., & Wöß, A. (2010). *The Compiler Generator Coco/R.* Available from http://www.ssw.uni-linz.ac.at/Coco/ ([Online; accessed 13-January-2010])

Muller, P.-A., Fleurey, F., & Jézéquel, J.-M. (2005). Weaving executability into object-oriented meta-languages. In L. C. Briand & C. Williams (Eds.), *Models* (Vol. 3713, p. 264-278). Springer.

Nóbrega, L., Nunes, N. J., & Coelho, H. (2005). Mapping ConcurTaskTrees into UML 2.0. In S. W. Gilroy & M. D. Harrison (Eds.), *Interactive systems, design, specification, and verification, 12th international workshop, DSVIS 2005* (Vol. 3941, p. 237-248). Springer.

Object Management Group. (2003, October). *Meta-Object Facility 2.0 core specification.* (url: http://www.omg.org/docs/ptc/04-10-15.pdf)

Object Management Group. (2006, May). *Object constraint language specification, version 2.0* (Tech. Rep.). OMG.

openArchitectureWare Working Group. (2010). *openArchitectureWare Working Group Homepage.* (http://www.eclipse.org/workinggroups/oaw/ (Visited in 2010))

Parigot, D., Courbis, C., Degenne, P., Fau, A., Pasquier, C., Fillon, J., et al. (2002). Aspect and xml-oriented semantic framework generator: Smarttools. *Electr. Notes Theor. Comput. Sci.*, *65*(3).

Parr, T. J., & Quong, R. W. (1995). Antlr: A predicated- *ll(k)* parser generator. *Softw., Pract. Exper.*, *25*(7), 789-810.

Paternò, F., Mancini, C., & Meniconi, S. (1997). ConcurTaskTrees: A diagrammatic notation for specifying task models. In *Interact '97: Proceedings of the ifip tc13 interantional conference on human-computer interaction* (pp. 362–369). London, UK, UK: Chapman & Hall, Ltd.

Pedro, L. (2009). *A systematic language engineering approach for prototyping domain specific languages*. Unpublished doctoral dissertation, Université de Genève. (Thesis # 4068)

Pedro, L., Risoldi, M., Buchs, D., Barroca, B., & Amaral, V. (2008). Developing domain specific modeling languages by metamodel semantic enrichment and composition: a case study. In *Submitted to the ieee software special issue on domain-specific languages & modeling.*

Pedro, L., Risoldi, M., Buchs, D., Barroca, B., & Amaral, V. (2009). Composing visual syntax for domain specific languages. In J. A. Jacko (Ed.), *Human-computer interaction* (Vol. 5611, p. 889-898). Heidelberg: Springer. Available from `http://smv.unige.ch/events/ruip09`

Pedro, L., Risoldi, M., Buchs, D., Barroca, B., & Amaral, V. (2010). *Developing domain specific modeling languages by metamodel semantic enrichment and composition: a case study* (Technical Report No. 215). Centre Universitaire d'Informatique, Université de Genève. (Available at `http://smv.unige.ch/technical-reports/techreportreference.2010-03-09.8949650056`)

Penner, R. R., & Steinmetz, E. S. (2002). Model-based automation of the design of user interfaces to digital control systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, *32*(1), 41-49.

Penner, R. R., & Steinmetz, E. S. (2003). Implementation of automated interaction design with collaborative models. *Interacting with Computers*, *15*(3), 367-385.

Pnueli, A. (1977). The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, 31 October-2 November, Providence, Rhode Island, USA* (p. 46-57). IEEE.

Pommereau, F. (2007). Versatile boxes: a multi-purpose algebra of high-level petri nets. In G. A. Wainer (Ed.), *Proceedings of the 2007 Summer Computer Simulation Conference, SCSC 2007, San Diego, California, USA, July 16-19, 2007* (p. 665-672). Simulation Councils, Inc.

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). Springer-Verlag.

Risoldi, M., Amaral, V., Barroca, B., Bazargan, K., Buchs, D., Cretton, F., et al. (2009). A language and a methodology for prototyping user interfaces for control systems. In D. Lalanne & J. Kohlas (Eds.), *Human machine interaction* (Vol. 5440/2009, p. 223-252). Springer Berlin/Heidelberg. Available from `http://archive-ouverte.unige.ch/unige:5104`

Risoldi, M., & Buchs, D. (2007). 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), 23-27 september 2007, usa* (p. 179-182). IEEE Computer Society.

Risoldi, M., Masetti, L., Buchs, D., Barroca, B., & Amaral, V. (2008). A model-based methodology for control systems gui design prototyping. In *Proceedings of the PCAPAC 2008 workshop, available at `http://archive-ouverte.unige.ch/unige:5102`.*

Schlee, M., & Vanderdonckt, J. (2004). Generative programming of graphical user interfaces. In *Avi '04: Proceedings of the working conference on advanced visual interfaces* (pp. 403–406). New York, NY, USA: ACM.

Schluehr, K. (2008). *Easyextend.* Available from `http://www.fiber-space.de/EasyExtend/doc/EE.html` ([Online; accessed 13-January-2010])

Sebrechts, M. M., Cugini, J. V., Laskowski, S. J., Vasilakis, J., & Miller, M. S. (1999). Visualization of search results: a comparative evaluation of text, 2D, and 3D interfaces. In *SIGIR '99: Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval* (pp. 3–10). New York, NY, USA: ACM.

SMV Group. (2010). *Algebraic Petri Nets Analyzer.* (`http://alpina.unige.ch`)

Steinberg, D., Budinsky, F., Paternostro, M., & Merks, E. (2009). *EMF: Eclipse Modeling Framework 2.0.* Addison-Wesley Professional.

Triskell team. (2010). *Kermeta - breathe life into your metamodels.* Available from `http://www.kermeta.org/` (`http://www.kermeta.org/` (Visited in 2010))

Vanderbilt University, I. f. S. I. S. (2005). GME 5 User's Manual [Computer software manual].

Vanderdonckt, J., Chieu, C. K., Bouillon, L., & Trevisan, D. (2004). Model-based design, generation, and evaluation of virtual user interfaces. In *Web3d* (p. 51-60).

Vangheluwe, H., & de Lara, J. (2004). Computer Automated Multi-Paradigm Modelling for Analysis and Design of Traffic Networks. In *Winter Simulation Conference* (p. 249-258).

Vangheluwe, H., de Lara, J., & Mosterman, P. (2002). An introduction to multi-paradigm modelling and simulation. In *Proceedings of AI, Simulation and Planning AIS 2002, Lisbon, Portugal* (p. 9-20). Available from `http://hdl.handle.net/1854/LU-158059`

Van Wyk, E., Krishnan, L., Bodin, D., & Johnson, E. (2006). Adding domain-specific and general purpose language features to Java with the Java language extender. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on object-oriented programming systems, languages, and applications* (pp. 728–729). New York, NY, USA: ACM.

Viswanadha, S. (2010). *JavaCC project home.* Available from `https://javacc.dev.java.net/` ([Online; accessed 13-January-2010])

White, J., Schmidt, D., Nechypurenko, A., & Wuchner, E. (2007). Introduction to the Generic Eclipse Modeling System. *Eclipse Magazine*(6), 11-18.

Wikipedia. (2009). *Comparison of parser generators — Wikipedia, The Free Encyclopedia.* Available from `http://en.wikipedia.org/w/index.php?title=Comparison_of_parser_generators&oldid=333680501` ([Online; accessed 13-January-2010])

Wolf, B., Mofor, G., & Rode, J. (2007). Use Cases and Concepts for 3D Visualisation in Manufacturing. In R. Koschke, O. Herzog, K.-H. Rödiger, & M. Ronthaler (Eds.), *INFORMATIK 2007: Informatik trifft Logistik. Band 2. Beiträge der 37. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 24.-27. September 2007 in Bremen* (Vol. 110, p. 345-352). GI.