View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Thèses en ligne de l'Université Toulouse III - Paul Sabatier

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)*

Présentée et soutenue le *24/06/2013* **par :**

SELMA DJEDDAI

Combining Formal Verification Environments and Model-Driven Engineering

---

JURY

LOUIS FÉRAUD	Université de Toulouse	Président du Jury
ALLAOUA CHAOUI	Université de Constantine, Algérie	Rapporteur
MOHAMED MEZGHICHE	Université de Boumerdès, Algérie	Examinateur
RALPH MATTHES	Université de Toulouse	Directeur de Thèse

École doctorale et spécialité :
MITT : Domaine STIC : Sureté de logiciel et calcul de haute performance

Unité de Recherche :
Institut de Recherche en Informatique de Toulouse (IRIT UMR 5505)

Directeur(s) de Thèse :
Ralph MATTHES et *Martin STRECKER*

Rapporteurs :
Allaoua CHAOUI et *Ousmane KONÉ*

À ceux que j'aime

Remerciement

En premier lieu, je souhaite remercier le ministère Algérien de l'enseignement supérieur et de la recherche scientifique d'avoir financé ma thèse durant ces années et de m'avoir permis de perfectionner mes connaissances au cours de cette thèse à l'IRIT.

J'adresse mes remerciements à mon encadrant Martin Strecker, maître de conférence à l'Université Paul Sabatier (Toulouse) pour le temps qu'il m'a accordé mais aussi pour tout ce qu'il m'a appris et permis d'apprendre. Je remercie aussi mon directeur de thèse, Ralph Matthes, chargé de recherche CNRS à l'IRIT, pour ses précieux conseils et sa bonne humeur.

Je souhaite remercier particulièrement Mohamed Mezghiche, professeur à l'université M'hamed Bougara de Boumerdès (Algérie), d'avoir cru en moi depuis le début et de m'avoir toujours encouragée. Je suis ravie d'intégrer son équipe l'année prochaine et espère être à la hauteur de ses attentes.

Un grand merci aux autres membres du jury d'avoir accepter et pris le temps d'évaluer ma thèse.

Je remercie chaleureusement l'ensemble de l'équipe ACADIE dont les membre m'ont accueillit parmi eux pendant quatre années. C'était très agréable de les rencontrer tout les jours, et de travailler parmi eux, en particulier: Mamoun, Jean-Paul, Bertrand, Mathieu, Manuel, Jean Baptiste, Yamine... etc. Je les remercie pour leurs gentillesse et leurs encouragements.

Je tiens également à remercier les membres avec qui j'ai partagé mon bureau pendant ces années : Nadezhda, Iulia et Elie. Merci pour tous les coup de gueule et les fous rires, je vous souhaite le meilleur.

Lorsque je suis arrivée à Toulouse, j'étais loin de ma famille, mais heureusement petit à petit je me suis reconstruit une petite famille ici. Ces personnes, je les garderai toujours

dans mon coeur. Je leur témoigne toute ma gratitude pour avoir été là pour moi dans les bons comme dans les mauvais moments : Mounira, Faten, Fatiha, Akila, Hajer, Anis, Rym et Jamel; sans oublier mes deux bouts de chou: Chahd et Rayane.

Un remerciement chaleureux à mes amis en France et en Algérie en particulier mes adorées: Sarah, Nabila et Ludivine.

Pour leur soutien indéfectible, je remercie toute ma famille, en particulier ma soeur Amina et mon frère Mehdi. Bien sûr, je remercie grandement mes chers parents sans qui je n'en serais pas là aujourd'hui. Merci pour leurs sacrifices pour assurer mon éducation, j'espère les rendre fiers.

Enfin, je remercie sincèrement mon fiancé Mebarek pour sa présence même dans les moments les plus difficiles et pour son soutien sans failles.

Selma Djeddai

Association d'environnements de vérification formelle et de l'Ingénierie Dirigée par les Modèles

Directeur de thèse : Ralph Matthes, *CNRS*
Co-directeur de thèse : Martin Strecker, *CNRS*

Résumé

Les méthodes formelles (comme les prouveurs interactifs) sont de plus en plus utilisées dans la vérification de logiciels (en particulier les logiciels critiques). Elles peuvent compter sur leurs bases formelles solides ainsi que sur leurs sémantiques précises. Cependant, elles utilisent des notations complexes qui sont souvent difficiles à comprendre pour un public non averti. Ce problème se pose particulièrement lors de collaborations entre des experts du domaine industriel et des professionnels de la preuve interactive. En effet, les experts du domaine industriel ont parfois du mal à voir précisément comment leurs systèmes sont représentés dans les assistants de preuves. D'un autre côté, ces experts sont souvent habitués à interagir avec les outils et formalismes que propose l'Ingénierie Dirigée par les Modèles comme les diagrammes de classes. Ces diagrammes utilisent des notations intuitives mais souffrent d'un manque de bases formelles. Aussi, ils ne permettent aucunement d'effectuer des vérifications sur les systèmes.

Dans cette thèse, nous proposons de faire interagir les deux domaines complémentaires que sont les méthodes formelles et l'ingénierie dirigée par les modèles. Nous proposons une approche permettant de traduire des types de données fonctionnels (utilisés dans les prouveurs interactifs comme Coq ou Isabelle) en diagrammes de classes et vice-versa. Afin d'atteindre ce but, nous utilisons une méthode de transformation dirigée par les modèles. Cette dernière consiste à définir des règles de transformation sur les éléments d'un méta-modèle source vers les éléments d'un méta-modèle cible. Dans ce cas, tout modèle source (conforme au méta-modèle source) donne automatiquement un modèle cible (conforme au méta-modèle cible) après application de la transformation.

Par conséquent, nous définissons dans cette thèse chacun des méta-modèles source et cible pour les types de données fonctionnels ainsi que pour les diagrammes de classes. Nous décrivons aussi les règles de transformation dans les deux sens de la transformation. Nous illustrons notre approche avec deux études de cas et combinons nos résultats avec la génération d'éditeurs graphiques ou textuels à partir de diagrammes de classes (en utilisant les outils GMF et Xtext). La première étude de cas porte sur les diagrammes de décision binaires, tandis que la seconde décrit la définition d'un langage spécifique à un domaine: Safety Critical Java.

Institut de Recherche en Informatique de Toulouse - UMR 5505
Université Paul Sabatier, 118 route de Narbonne, 31062 TOULOUSE cedex 4

Selma Djeddai

Combining Formal Verification Environments and Model-Driven Engineering

Thesis Advisor: Ralph Matthes, *C.N.R.S.*
Thesis Co-advisor: Martin Strecker, *C.N.R.S.*

Abstract

Formal methods (such as interactive provers) are increasingly used in software verification especially for critical software. This is so because they rely on their strong formal basis and precise semantics. However, they use complex notations that are often difficult to understand for unaccustomed users. This becomes a problem when a collaboration is needed between interactive proof professionals on the one hand and domain experts on the other hand. In fact, the latter may have trouble to see precisely how their system specifications are represented in proof assistants, because they are often used to interact with specific Model Driven Engineering tools and formalisms (such as class diagrams). These latter offer a more attractive syntax and use intuitive notations. However, they suffer from a lack of formal foundations and do not allow to perform verification on systems.

In this thesis, we are interested in combining these two complementary domains that are formal methods and Model Driven Engineering. We propose an approach allowing to translate functional data types (used in interactive provers like Coq or Isabelle) into class diagrams and vice versa. To achieve this goal, we use a model-driven transformation method. This method consists in defining transformation rules from the elements of a source meta-model into those of a target meta-model. Consequently, after processing the transformation, every source model (which conforms to the source meta-model) gives automatically a target model (which conforms to the target meta-model).

Therefore, we define in this thesis each of the source and target meta-models for each of the functional data types and the class diagrams. We also describe the transformation rules in both directions of the transformation. We illustrate our approach with two case studies and combine our results with the generation of graphical or textual editors out of class diagrams (using the tools Xtext and GMF). The first case study deals with Binary Decision Diagrams, while the second describes the definition of a domain specific language: Safety Critical Java.

Institut de Recherche en Informatique de Toulouse - UMR 5505
Université Paul Sabatier, 118 route de Narbonne, 31062 TOULOUSE cedex 4

Contents

List of Figures

List of Tables

Introduction

Context and Motivation for this Thesis

The PhD thesis described in this document has been carried out in Toulouse that has become the center of the European aerospace industry. This has created a strong demand for systems and software in a field which is often referred to as Embedded Software. This software is complex and is required to be failure-free. In contrast to software that we use in our everyday life (games, web, email ...), defects on critical software can lead to considerable financial loss or even endanger human life. This is why this software is called Safety Critical Software. To ensure the high quality requirements, industrials rely on formal methods to certify their applications.

In fact, formal methods (such as interactive proof assistants [73, 29]) are increasingly used in software engineering to verify the correctness of software. They have a solid formal basis and a precise semantics, but they use complex notations that might be difficult to understand for unaccustomed users. This becomes a problem when collaboration is needed between interactive proof professionals and experts of an applicative domain. In fact, these experts often have trouble to see precisely how their system specifications are represented in the proof assistants.

Instead, domain experts are often used to interact with the tools and formalisms proposed by Model Driven Engineering (MDE) [19, 84]. This method can rely on its visual specification languages such as class diagrams [45] that use intuitive notations. These diagrams allow to specify, visualize, understand and document software systems. However, they suffer from a lack of precise semantics. Also, they do not allow to perform any verification on systems. We are interested in combining these two complementary domains that are formal methods and MDE by translating the elements of the one into the other.

One possible scenario is to define the abstract syntax of a Domain Specific Language (DSL) [94] to be used in the context of a formal verification, and then to generate a corresponding meta-model. Inversely, the meta-model can then be modified by an application engineer and serve as basis for re-generating the corresponding data types. This operation may be used to find a compromise between the representation of the application engineer's wishes on the meta-model and functional data structures used in the proof. Furthermore, the meta-model can be used to easily generate a textual (or graphical) editor.

Overview of our Approach

In order to translate functional data types (used in interactive provers like Coq or Isabelle) into class diagrams and vice versa, we use an MDE-based transformation method. This method allows to define a generic transformation process from functional data types to meta-models and backwards.

Figure A.1 shows an overview of our approach. In the first direction of the translation, we derive a meta-model of data types starting from an EBNF representation of the data type definition grammar [73]. This meta-model is the source meta-model of our transformation. The class diagrams are represented using Ecore: the core language of the Eclipse Modeling Framework [51]. The latter is comparable to EMOF (the class diagram standard recommended by the OMG). We describe then a subset of the Ecore meta-model to be the target meta-model. The transformation rules are defined on the meta-level and map elements from the source meta-model to their counterparts in the target meta-model. The *DataTypeToEcore* function implements these rules in Java. It takes as input models which conform to the source meta-model and returns their equivalent in a model which conforms to the target meta-model. We use the mapping between the constructs of the two meta-models to define the reverse direction of transformation rules.

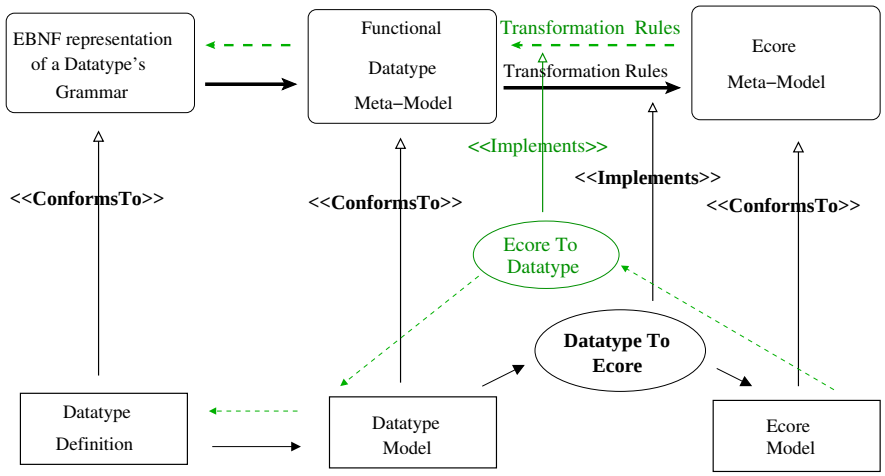


Figure 1: Overview of the Transformation Method

Bidirectionality [91] is one of the desired options of MDE-based transformations. Indeed, assuming we start from a source model M_S, then we perform a transformation using a function f to get a target model M_T. It is important to derive an equivalent model to M_S, as a result to the application of f^{-1} on M_T. In our case, such a feature requires more restrictions on the Ecore models. This property is only guaranteed when the source model is the data types model (for more details see discussion on page 102). The implementation of most of the transformation rules of the two sides has been successfully performed in an

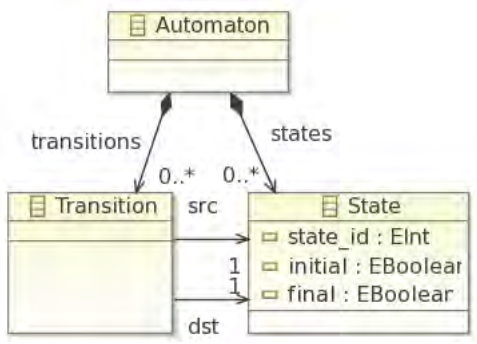
application.

Our work aims at narrowing the gap between interactive proof and meta-modeling by offering a way to transform data structures used in interactive provers to meta-models and vice-versa. Furthermore, the generated meta-model can be used to easily generate textual (or graphical) editors using Xtext (respectively GMF: Graphical Modeling Framework) facilities [51].

Example: Figure 2 shows an example of the application of our transformation approach to an automaton description. The left part of the figure (2a) represents a data type description of an automaton, in this case written in the Caml language. Each automaton is then composed of a list of states and a list of transitions. Every state is composed of an integer value (for identifying the state) and two Boolean values (defining whether a state is an initial state and/or a final state). A transition is then described by two states, a source and a target. The right part of the figure consists in the representation of the same automaton as a meta-model in Ecore. This meta-model represents the result of applying our transformation on the presented data types.

```
type automaton =
state list * transition list

type state = int * bool * bool

type transition = state * state
```

(a) Automaton Data Types (in Caml)



(b) Automaton Meta-model (in Ecore)

Figure 2: Example of the Application of our Approach

Contributions

The contributions of this thesis consist in several achieved goals that are presented in the following:

- We define a subset of data types descriptions which are common to functional languages (SML, Caml, Haskell) and the Isabelle proof assistant. This subset contains the essential elements needed to describe the shapes that data types take in every-day

practice, including the use of parameterized types. We then construct the corresponding meta-model representing this subset, starting from the EBNF grammar of the subset.

- We define a subset of the Ecore meta-model which at the same time is expressive enough to model the basic class diagrams and also contains elements that are translatable to functional data types. The meta-models are essential to apply an MDE-based transformation approach.
- We describe in a first direction a fully-automated MDE-based transformation process from functional data types to meta-models. We particularly pay attention to write transformation rules that cover the whole defined subset of data types. In order to ensure the validation of our generated meta-models in Ecore, we propose some well formedness constraints on the translated data types.
- We introduce the transformation in the opposite direction: from meta-models to data types. After studying all the possible patterns that may appear in the meta-model, we select the translatable ones. This step requires to postulate some well-formedness conditions insuring the correctness of the generated data types.
- Most of the transformations described in this thesis are implemented using Java and EMF as an application that is used in case studies. We couple this work with both the generation of graphical (and/or textual) editors and generation of certified object oriented code.
- We illustrate the feasibility of our approach with two case studies. The first consists in the construction of Binary Decision Diagrams with subtree sharing using certified code generation. The second defines a DSL named Safety Critical Java: a Java-like language enriched with timing annotations.

Publications Our work has resulted in the following publications:

- *Selma Djeddai, Mohamed Mezghiche, and Martin Strecker. A case study in combining formal verification and Model-Driven Engineering. In Vadim Ermolayev, Heinrich C. Mayr, Mykola Nikitchenko, Aleksander Spivakovsky, Grygoriy Zholtkevych, Mikhail Zavileysky, and Vitaliy Kobets, editors, ICTERI, volume 848 of CEUR Workshop Proceedings, pages 275–289. CEUR-WS.org, 2012.*
- After a second selection, an extension of our paper (accepted in the previous conference) has been published as post proceedings in :

  Selma Djeddai, Mohamed Mezghiche, and Martin Strecker. Combining verification and MDE illustrated by a formal Java development. In Vadim Ermolayev, Heinrich C. Mayr, Mykola Nikitchenko, Aleksander Spivakovsky, and Grygoriy Zholtkevych,

editors, ICT in Education, Research, and Industrial Applications, volume 347 of Communications in Computer and Information Science, pages 131–148. Springer Berlin Heidelberg, 2013.

- *Selma Djeddai, Martin Strecker, and Mohamed Mezghiche. Integrating a formal development for DSLs into meta-modeling. In Alberto Abelló, Ladjel Bellatreche, and Boualem Benatallah, editors, MEDI, volume 7602 of Lecture Notes in Computer Science, pages 55–66. Springer, 2012.* An extension of this paper has been submitted to the Journal of Data Semantics (JoDS).

Outline of the Thesis

This PhD document is organized as follows:

Chapter 1 describes the basic notions of MDE and model transformation. It also gives an overview of related work consisting in different approaches aiming at combining formal verification and MDE. Chapter 2 consists in the presentation of our chosen meta-modeling framework: Eclipse Modeling Framework. Chapter 3 fixes the formal framework used in the rest of the thesis. Here, we introduce some Caml and Isabelle constructs.

The core contributions of this PhD are given in Part II. Chapter 4 and Chapter 5 introduce respectively our two-way transformation from functional data types to class diagrams (represented in Ecore) and back from class diagrams to data types. As for Chapters 6 and 7, they illustrate our contributions by two case studies: Binary decision diagrams and Safety Critical Java. We finish this document by drawing a conclusion and citing the perspectives of this work.

Chapter 1

Scientific Context and Related Work

1.1 Introduction

This thesis consists mainly in the transformation of data types used in functional programming into class diagrams and vice versa in the context of Model Driven Engineering (MDE). This is why we need to clearly define the basic concepts of MDE.

In this chapter, we give an overview of the basic concepts of MDE. We start in Section 1.2 by introducing the notions of *Model* and *Meta-model* along with defining the *MDA* standard. Then, in Section 1.3 we define *Model Transformation* and some model transformation approaches. We illustrate these approaches with model transformation tools (that implement each approach). We finish the section with conclusions that explain the reasons that lead us to choose our implementation approach for the transformation process.

Section 1.4 consists in a presentation of a subset of the related work. At first, we introduce existing approaches aiming at the integration of MDE and formal methods in general. Then, we detail the most related approaches to our work: the transformations between class diagrams and formal frameworks. We finally position our approach in the summary (Section 1.5).

1.2 Model Driven Engineering

Model Driven Engineering (MDE) is a software development methodology where the models are the central elements in the development process. It is a particular kind of generative engineering in which all or parts of an application is engendered from a model.

In order to describe and develop a system, MDE embodies a stepwise refinement process and allows to describe the software architecture (in the jargon called business knowledge) independently of the technical platform. Basically, MDE was triggered by object technol-

ogy: in fact, after the "everything is an object" MDE prones the "everything is a model" [18]. It provides a large number of modeling views that allow to express separately the concerns of users, designers, developers .... Its main objective is to develop, maintain and evolve software by performing model transformations.

Before going further into the definition of principles related to the MDE, it seems necessary to introduce models and meta-models.

1.2.1 Model and Meta-Model

In MDE, the models are the primary artifacts of the development life-cycle. Despite this, there is no unique definition of what a *model* is. In fact, the MDA guide [74] defines a model as follows: "a model of a system is a description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text. The text may be in a modeling language or in a natural language"; while in the MDA Explained book [65], "A model is a description of a system written in a well-defined language." As for *Bézivin & Gerbé* [20] "A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system."

A *meta-model* defines elements of a language allowing to express models. It describes the different kinds of model components and the way they are arranged and related [19]. The instances of the meta-model elements are used to construct a model of the language.

Meta-models are used to define the abstract syntax of languages belonging to a particular domain: Domain Specific Languages (DSL). They constitute the heart of MDE.

1.2.2 Domain Specific Languages (DSL)

In the literature, there are several definitions for what is a DSL. According to [94], a Domain Specific Language (DSL) is: "a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain".

1.2.3 Model Driven Architecture

The Object Management Group (OMG) [75] defined the Model Driven Architecture (MDA) standard [65, 20], as specific incarnation of the MDE. MDA promotes the use of models in different phases of software development. The basic idea is to separate the business logic description from any technical platform. While this idea is not new, using models is preferred over classical programming languages.

The development cycle of MDA does not seem very different from the classical development cycle which is mainly based on the phases of analysis, design, coding, testing and deployment. The main difference lies in the nature of the artifacts created during the development phase: models.

MDA classifies models in three categories: Computation Independent Model (CIM), Platform Independent Model (PIM) and Platform Specific Model (PSM). In these models, the developed system is described at different levels of abstraction. According to the MDA guide [74], these models are defined as follows:

- CIM: "is a view of a system from the computation independent viewpoint. A CIM does not show details of the structure of systems. A CIM is sometimes called a domain model and a vocabulary that is familiar to the practitioners of the domain in question is used in its specification".
- PIM: "is a view of a system from the platform independent viewpoint. A PIM exhibits a specified degree of platform independence so as to be suitable for use with a number of different platforms of similar type".
- PSM: "is a view of a system from the platform specific viewpoint. A PSM combines the specifications in the PIM with the details that specify how that system uses a particular type of platform".

1.2.4 The Four-Layer MOF Meta-modeling Architecture

Simplification or abstraction is the essence of modeling. The OMG defines a modeling architecture called "The Four-Layer Meta-modeling Architecture" as presented in Figure 1.1. According to this architecture there are four levels of modeling:

- M0 level: the level of real (concrete) data one wants to model. These data provide an instance of the model level depending on the M1 level.
- M1 level: the model level that allows to write data at M0. Typically a UML model belongs to this level. Models belonging to this level are described by a meta-model level M2.
- M2 level: to this level belong meta-models of description languages, typically the UML meta-model or EMF.
- M3 level: the meta-meta-model level. The OMG defines a single language for defining meta-models called the Meta-Object-Facility (MOF) [75]. It allows to describe, extend or modify meta-models. MOF is self-describing, it can describe its semantics itself.

1.3 Model Transformation

Model transformation is one of the key techniques of the field of MDE. The principal motivation behind model transformation is the ability to automate routine aspects of processes in order to make software development and maintenance more efficient.

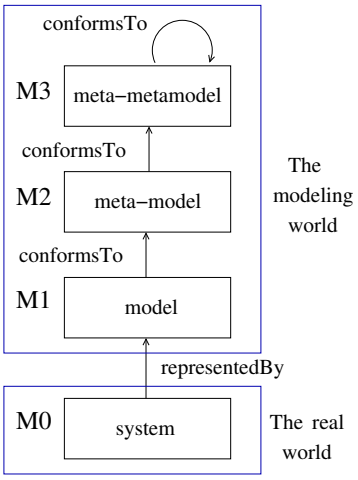


Figure 1.1: Meta-modeling Layers [18]

Kleppe et al [65] define a *transformation* as "the automatic generation of a target model from a source model, according to a transformation definition. A *transformation definition* is a set of transformation rules that together describe how a model in the source language can be transformed to a model in the target language. A *transformation rule* is a description of how one or more constructs in the source language can be transformed to one or more constructs in the target language".

Figure 1.2 represents a typical MDE transformation pattern. A transformation definition is then represented by a mapping from elements of the source meta-model (Language 1) to those of the target meta-model (Language 2). Consequently, each model conforming to the source meta-model can be automatically translated to an instance model of the target meta-model using a transformation tool that implements the transformation definition. The transformation definition (which is itself a model) is written in a meta-language that extends the standard meta-language (the meta-meta-model (MOF)).

Model transformation is often associated with the MDA approach. Indeed, the implementation of MDA is completely based on the definition of models and their transformations. To do this, the OMG proposed a formalization and standardization of techniques for the transformation of models to ensure compatibility between MDA tools. This standard is known as *QVT*.

QVT stands for Query/View/Transformation [77]. It is the standard language recommended by the OMG to specify Model transformation in the domain of MDA [66]. It defines three sublanguages for transforming models: the *Relations* language and the *Core* language which are declarative and the *Operational mappings* language which is specified as a standard way for providing imperative implementations.

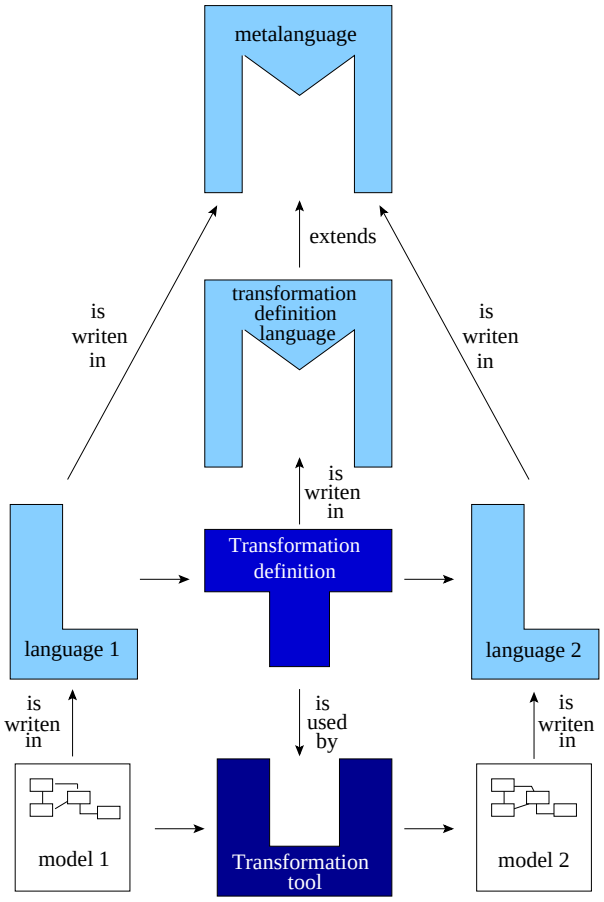


Figure 1.2: Pattern for MDE Transformations [65]

The Object Constraint Language (OCL) [76] is a language used to express constraints on class diagrams. It allows to describe invariants on MOF models/meta-models in a textual format. It is usually associated with UML. It constitutes an important part of the QVT standard for model transformation. It is not a transformation language in the strict sense but it is used by some transformation tools for the navigation in models.

1.3.1 Classification of Model Transformation Approaches

Model transformation approaches can be classified depending on several criteria. They can be classified according to the target meta-model of the transformation. In fact, when the source and target meta-models (languages) are identical, it is an *endogenous* transformation. When they are different, it is *exogenous*.

They can also be divided into two categories depending on type of the target of the

transformation: *Model-to-model* and *Model-to-text*. The model-to-model transformation gives a model as a result of the transformation, while model-to-text approaches generate code.

This section is mainly about the classification of Model-to-model approaches, based on this survey [31].

Direct Manipulation Approaches In this category, the implementation of transformation rules, scheduling (*i. e.* the sequencing of conflicting rules), tracing and other facilities is performed by the user in a programming language (as Java for example). They are then implemented as object oriented framework that provides only an internal model representation and some API to manipulate it.

Operational Approaches This kind of approaches is comparable to direct manipulation approaches with a more dedicated support for transformation. They are usually based on an existing meta-modeling formalism that is extended in order to allow modeling the behavior of meta-models. Several systems implement this solution for model transformation such as QVT Operational mappings [77] and Kermeta [72]. This latter is described more in detail in Section 1.3.2.5.

Relational Approaches As its name implies, this type of approaches is based on the mathematical notion of a relation and uses declarative transformation rules. The basic idea is to specify relations among the source and target model elements. This category of approaches supports multi-directional rules and usually requires to separate source and target models. Examples of implementations of relational approaches: QVT Relations [77] and AMW [35].

Graph Transformation Based Approaches They are based on graph transformation theory. The source and target meta-models are represented as graphs. The transformation rules are formed of a LHS and RHS. When applying a transformation rule, the LHS is matched in the source model to be replaced by the RHS in the target model. Examples of systems in this category: VIATRA [95], GReAT [3], AGG [92] (Section 1.3.2.1), AToM^3 [32] (Section 1.3.2.2) and Moflon [7] (Section 1.3.2.3).

There are also *hybrid approaches* that use a mix of different approaches for transforming models. We can cite the ATL tool [62] that implements a combination of imperative and declarative constructs in the transformation definition. This tool is described more in detail in Section 1.3.2.4.

1.3.2 Model Transformation Tools

In this section, we present different model transformation tools that implement some model transformation approaches presented in the previous section. We then explain the reasons that lead us to implement our transformation process using a direct manipulation approach in Java (see Section 1.3.2.6). For more in-depth discussions and comparisons, the reader can refer to the articles [41, 90, 97, 80].

1.3.2.1 The Attributed Graph Grammar (AGG)

The Attributed Graph Grammar system (AGG) [92, 2] is a graph transformation tool developed at Technische Universität Berlin. Unlike other transformation tools, AGG has solid formal bases on Category Theory and implements attributed graph grammar.

Meta-models are represented with graphs (called type graphs) allowing the inheritance mechanism and multiplicities while models are represented with attributed graphs.

It provides a graphical mode that is well adapted for expressing rules.

In AGG, there is no distinction between the source and target meta-models. The type graph defined merges the two. It therefore implements only endogenous transformations.

AGG offers two execution modes (for applying the transformation rules): interactive mode and interpreter mode. In the first mode, the user can choose the rules to be applied while in the second the transformation engine chooses the right rules to be applied. In this case, transformation rules can be classified by layers, depending on when they will be executed. In AGG, it is also possible to express priority between rules. In fact, if two rules can be executed on the same sub-graph, the one which has the highest priority is chosen to be run, else the choice is made in a non-deterministic manner. Also, it provides a way to express patterns preventing a rule application (Non-Applicability Conditions (NAC)). It shows when a particular rule has not to be executed for a special shape of the sub-graph.

AGG has formal foundations. It is based on category theory. It concentrates on structural analysis aspects. It offers type checking functionalities on graphs and rules. Actually, it checks whether the transformation rules and graphs correspond to the type graph description. It is possible to disable or enable this functionality (partly or totally). It also looks for conflicts between rules and checks termination criteria.

One of the disadvantages of AGG is the lack of model exchange formats. However, it is implemented in Java and thus its transformation engine is usable by a Java API.

Since 2010, it is possible to use a tool that is close to AGG named Henshin [21, 55]. The latter is also developed at the University of Berlin. However, it is implemented on Eclipse and uses models described in EMF. Henshin provides easy exchange of models with AGG.

1.3.2.2 AToM^3

AToM^3 (A Tool for Multi-formalism and Meta-Modeling) [32, 16] is a tool for designing domain specific languages and model transformation. It is developed at McGill University of Montreal, Canada. It allows to define the abstract and the concrete syntax of a DSL using meta-modeling in order to generate a customized modeling environment for the developed DSL. It also permits to perform model transformation by applying graph transformation. Moreover, it offers a way to generate environments for Multi-View Languages.

AToM^3 uses UML-like meta-models [33] and manipulates them as graphs. It provides a graphical mode that is suitable for model transformation. Attribute values can also be specified textually by Python code.

The model transformation approach that implements AToM^3 is triple graph grammar. This approach is cleaner than regular graph grammar since it is based on three graphs: one for the source model, one for target model and a third one for the correspondence graph. The latter is hidden to the users, it is simply used to maintain the consistency between the source and target models.

Updating models is particularly developed in AToM^3. Indeed, it can perform transformation in an incremental way. When adding elements in the source model, it is possible to execute again the transformation and update the target model (due to the persistence of the correspondence graph).

AToM^3 comes along with some exchange format supported as XMI.

AToM^3 is a complete tool: it permits to perform model transformation, as well as to define the abstract and concrete syntax of a DSL in order to use it in a graphical or textual editor. It also facilitates the update mechanism of the models and allows the reuse of rules. However, its verification capabilities are very low compared to those proposed by AGG for example (consisting only in checking the correct typing of the target model).

1.3.2.3 MOFLON

MOFLON [7, 70] is a meta-modeling environment developed in the Real-Time Systems Lab at Technische Universität Darmstadt. It provides full support for MOF 2.0 (modularization and refinement concepts). It thus allows MOF meta-modeling, graph transformation based on graph grammar; verifying some properties (OCL constrains) on models and generating Java code for models and transformations. The major goal of MOFLON is the conformity with standards as MOF 2.0.

MOFLON is based on the Fujaba Toolsuite [38] which uses graph transformation for UML-like graph schemata. In fact the TGG (Triple Graph Grammar) editor and the SDM (Story Driven Modeling) editor has been adapted from Fujaba. These two editors allow to perform respectively bidirectional and unidirectional transformations. The transformation process is supported by visual editors except for the OCL part (for describing constraints).

The MOFLON tool provides several ways to represent the meta-models. Indeed, they can be implemented using commercial tools like Rational Rose or directly specified with

MOF 2.0 editors. It also allows the exchange with other existing modeling environments by providing the ability to import and export meta-models as XMI.

MOFLON does not offer support for the representation of the concrete syntax of DSLs. Its developers claim that it is mainly dedicated to the adaptation of existing tools for DSLs (having existing concrete syntaxes) [8].

In 2011, MOFLON has been completely re-engineered into eMoflon [13]. The model transformation tool has been combined with the EMF and Eclipse technologies. The main reasons that led to re-engineer Moflon is the important role that EMF has taken these few last years in research. In fact the developers claim that EMF has become the de-facto standard in the meta-modeling community.

1.3.2.4 The ATLAS Transformation Language

ATL [62, 15] is a model to model transformation tool developed at Université de Nantes, France by the ATLAS team. The transformation process (rules and helpers) is expressed textually in the ATLAS language. It has been created as an answer to the OMG QVT language request. It is considered as a hybrid textual language, it mixes declarative and imperative style. This language is composed of Rules, Helpers, Queries and Libraries. The rules can be divided in two categories:

- Declarative rules consisting of *matched rules* and *lazy rules*. Lazy rules are applied only when they are called by another rule.
- Imperative rules represented by *called rules*.

Helpers can be viewed as an ATL equivalent to methods in Java. They are specified in OCL and are used to navigate in the source meta-model. They can be called from anywhere in the ATL program. Thanks to helpers, ATL is suitable for the navigation on the meta-model.

It is implemented as an Eclipse plug-in and is well adapted to all the Eclipse modeling tools and functionalities. ATL has a different execution mode for endogenous transformations (target and source meta-model are the same), it is called: Refining mode.

When using ATL, we realized that besides its advantages cited previously, there were some negative points. The first one is about its complex syntax. The existence of helpers in addition to different sorts of rules makes the syntax heavier. Moreover, these different parts of an ATL program may use common constructs but which are syntactically expressed differently depending in which part they are used. This may lead to confusions. (For example the syntax of the “if” instruction has different syntaxes in helpers and matched rules).

Also in ATL, the type checking of rules is relatively weak. It consists mainly on a syntactic verification. Actually, the user can generate wrong elements in the target meta-model.

In addition, to our knowledge, in ATL there is no predefined way to represent priorities on rule executions. It is even rather complicated to declare these priorities whenever there are different combinations of rules (Matched and Called rules).

1.3.2.5 Kermeta

Kermeta [72] (for a **Ker**nel **meta**modeling language) is a tool developed by the team Triskel at Université de Rennes 1, France. It is a meta-programming environment allowing to simulate the execution of meta-models. Usually, this tool is dedicated to represent the operational semantics of meta-models for testing and simulation purposes. To achieve this goal, the developers used aspect oriented modeling to build an executable meta-language by composing action meta-models with class diagram meta-languages. In other words, the meta-language of Kermeta is based on Essential MOF (EMOF) 2.0 [75] which was extended by classes allowing to describe the semantics of operations on meta-models by defining a support for actions. The action language of Kermeta is imperative and object-oriented.

The meta-model of Kermeta can be divided into two parts; structural and behavioral. The structural part is represented by the EMOF meta-model while the behavioral consists of a class named expression and its subclasses. This part is used to specify the semantics of operations [71].

In the model transformation point of view, the approach of Kermeta is classified among the operational approaches of model transformation [31]. The part in the meta-model of Kermeta allowing to define operations is used to define transformation functions while the structural part is used to define models. For example, in Kermeta a transformation rule *AToB* is defined as an operation of the model element *A* with *B* as its return type. Then, in Kermeta, instead of handling the source and target models, the user manipulates the objects of the transformation itself which is also a model.

In order to validate model transformations, Kermeta provides a unit test framework (KUnit) based on JUnit (the Java framework for performing unit tests).

Kermeta is a powerful and expressive language, but it lacks simplicity and clarity. Unlike rule based transformation languages, Kermeta does not separately define target and source models [90].

Kermeta is available as an Eclipse Plug-in and offers a bridge towards the Eclipse Ecore formalism.

1.3.2.6 Synthesis

In the part devoted to the bibliography, we tested and experimented with several transformation tools to choose the approach that we will adopt for the implementation of our transformation. For each of the approaches presented further in the section, we experimented at least one tool.

Graph transformation tools (here AGG, AToM^3 and Moflon) permit to define source

and target meta-models all along with a set of transformation rules and use graphical representations of instance models that ease the transformation process. AGG is best suited to endogenous transformations. Also, it offers more verification options than other tools (even if they are purely syntactic). AToM^3 is particularly useful when transforming DSLs. Indeed, it is the only tool that offers a way to describe the concrete syntax (whether textual or graphical) of the processed DSLs. Moflon is dedicated to the transformation of already existing tools. It offers various exchange formats with other modeling tools. Kermeta offers a rich and expressive but complex transformation environment. Finally, ATL is particularly well-suited for injective transformations (that use principally declarative rules).

In general, the tools presented above offer weak verification functionalities. In fact, they are often limited to syntactic aspects (such as confluence of transformation rules) and do not allow to model deeper semantic properties (such as an operational semantics of a programming language and proofs by bisimulation). Also, when using one of the tools presented previously, we do not always know precisely how transformations are performed.

So we ended up opting for an approach of direct manipulation (using the EMF framework and Java). The encoding of our transformation rules is carried out in Java. However, they are written formally in a functional style (as if they had been written in Caml). The underlying model structure is provided by EMF.

1.4 Related Work

In this thesis, we are interested in bridging the gap between Model Driven Engineering and formal methods. Our main contributions consist in transforming data type structures used in functional languages into class diagrams (represented in Ecore) and vice-versa. This section contains the essential related work. We start by spelling out some approaches aiming at integrating formal methods and MDE in different ways. Next we present closer work: the transformation of class diagrams to formal languages, allowing to perform proofs.

1.4.1 Formal Frameworks and Model Driven Engineering

Despite its advantages, MDE suffers from a lack of solid formal basis. To remedy this shortcoming, different research teams are working on integrating MDE with existing formal methods. These formalizations can occur on different parts of the MDE. Indeed, they may occur on models (or meta-models), model transformation [28, 67, 9] or directly at DSL level [22, 12, 43]. Here, we are particularly interested in the interaction of MDE and formal languages in terms of class diagrams.

In this category, we start by presenting the work of *Richters* [83]. The author presents a formal definition of UML Class diagrams and OCL semantics using set theory and first order logic. His approach has been used to detect inconsistencies and to validate well-formedness rules of the UML 2.0 Standard. It then has been implemented as a tool for

the specification and the validation of systems (UML/OCL) : the USE tool (UML-based Specification Tool) [50].

The following approaches are concerned with the description of modeling frameworks in proof assistant environments as Coq [29] and Isabelle [73].

1.4.1.1 Coq4MDE

Coq4MDE [93] is a proof environment for describing aspects of MDE with the aim of giving formal foundations to MDE. To do so, the developers define the notion of model and reference model (actually meta-model). They also introduce two fundamental relations for MDE: *conformsTo* and *promotion*. The *conformsTo* relation "indicates whether a model is valid with respect to a reference model" while *promotion* "builds a reference model from a model". The consistency of the approach is validated by proving (using the Coq proof assistant [29]) that EMOF is self defined.

In [63], Coq4MDE is extended to support composition of model elements. This work consists in the formalization of composable verification technologies in order to ease the integration of pre-verified components.

In a more general approach [82], *C.Picard* worked on interfacing MDE and Type Theory [30] by using the Coq interactive theorem prover. The author represents meta-models as graphs and performs some proofs on these graphs in order to certify that the application of transformations on these graphs ensures some properties regardless of the input model.

1.4.1.2 A Formal Proof Environment for UML/OCL

A.D. Brucker [23] has worked during his PhD thesis on the encoding of object oriented specifications in the Isabelle/HOL interactive prover.

The tool HOL-OCL [25] is an interactive proof environment for UML class models annotated with OCL [76] specifications. It is based on a repository for UML/OCL models and Isabelle/HOL [73]. This system provides a way to run proofs on UML meta-models. It consists of:

- a repository used to import UML models in an XMI format.
- a package allowing to encode object oriented components into HOL.
- a library providing the theorems needed for performing verification.
- a suite of automated proof procedures.

This system was later extended with the ability of processing models by model transformation and code generation [24].

To sum up, *A.D. Brucker* has defined a formal proof environment for the object-oriented world and MDE. To do this, he used a shallow embedding approach by encoding object oriented data structures and MDE in HOL. However, such a functional and formal environment is not well suited for representations of object-oriented components. This gives rise to a complex environment that makes the proof process more difficult in Isabelle.

The transformation is one of the means used to establish the links between the formal languages and the languages provided by the MDE. It allows to describe a system using a formal language, but also to represent the same system using a more intuitive description language. This topic is discussed in the next section.

1.4.2 From Class Diagrams to Formal Languages and Back Again

EMF (Eclipse Modeling Framework) [27] models are comparable to Unified Modeling Language class diagrams [45]. For this reason, we are interested in the mappings from other formal languages to UML class diagrams and back again. Some research is dedicated to establishing the link between these two formalisms.

The integration of formal and semi-formal notations has been investigated for several years [46]. Among the first experiments of transformations between class diagrams and formal languages were transformations into Z [60]. In fact, *Kim et al.* worked on using Z as underlying semantics for UML diagrams [64] while *Evans et al.* worked on the effective transformation of UML models to Object Z [42].

In the next subsections, we present some work related to the transformations between formal languages and Class diagrams. We begin with transformations between UML and B, then between Alloy and UML, and finally Focal and UML. To our knowledge, there is no work for the translation of data types (used in functional languages) into class diagrams (or in the opposite direction).

1.4.2.1 B To UML and Back Again

Much of the work done for the transformations between class diagrams and formal languages has chosen the B language as formal part. B [1] is a formal method that allows to construct a program by successive refinements, using abstract specifications.

In order to specify the semantics of UML and benefit from verification tools provided by the B method, several studies have been conducted to translate UML to B. This was achieved by the development of several tools of translation, including UML2B [54] and U2B [86, 87]. In this work, the translation rules are missing solid semantics because they are defined in natural language.

The transformation of UML to B has been a little less explored. We can cite the thesis *J-C. Voisinet* [96, 52] and work conducted by *H. Fekih* [44].

Ossami & al. [79] adopt a different approach to establish the link between UML and B. The approach consists in a joint and simultaneous development of two views of a system

(in UML and B) while maintaining the traceability and consistency between the two views.

Work of *Idani & al.* We present this work in a separate paragraph because it is different from the ones cited previously. It is the only translation from B to UML (and vice-versa) that uses meta-model based transformation while the others only use simple translations.

In his PhD thesis [56], *A.Idani* studies the derivation of UML diagrams starting from B specifications. He uses a generic process based on a structural and semantic mapping between UML and B constructs in order to perform a model driven transformation.

This approach consists in two steps. First, some transformations of static parts of the B specifications to UML class diagrams are presented with the purpose of documentation. Next, using defined pertinence criteria, the user selects the adequate transformation rules. Then, an algorithm is proposed to perform the transformation. This phase is semi-automated, a user intervention is still necessary for the selection of rules. The second step is about exploring the dynamic (behavioral) parts of a B specification for the derivation of UML state transition diagrams. To achieve this objective, the author uses graph abstraction techniques in order to build automatically a state transition diagram. The whole approach is spelled out in [57].

In [58], the authors focus on the reverse side of the transformation (only for the static parts). They propose an evolutive framework to assist the derivation of formal B specifications from UML class diagrams. This work is based on the structural and semantic mapping between UML and B static constructs along with a well-defined UML core with a set of safety commandments [59]. Starting from this safe UML model, a formal model of a B specification is built. A certifiable code can then be produced after a series of refinements and proofs using the Atelier-B tool [14].

This approach seems to be global and complete, however it includes drawbacks. In fact, the user is still solicited in order to choose the rules to be applied during the transformation process. Consequently, it is possible to generate two different UML diagrams starting from a single B description. Also, in the second direction of the transformation (from UML to B), it is not the usual UML version that is used but a modified one, Safe UML.

1.4.2.2 UML To Alloy and Back Again

Alloy [61] is a declarative textual modeling language based on first order logic that gives support for notions of object orientation. Its models are analyzed with a fully automated tool: the Alloy Analyzer.

Several studies have been performed to establish the correspondence between the UML class diagram (augmented with OCL [76]) and the components of Alloy. Some have established the mapping rules defined using natural language [69], while others have translated manually UML specification to Alloy [37, 47].

In his PhD thesis [10], the author worked on an MDE based transformation approach for automatically generating Alloy specifications from UML class diagrams [11]. The purpose

of the approach is to perform analysis using Alloy on UML models. The transformation is defined using a number of transformation rules from UML concepts to Alloy concepts. This work was realized by the development of a UML profile for Alloy.

In [85], the authors extended this work with a way to transform Alloy instance models to UML object models. This transformation is automatically generated from the transformation that maps UML to Alloy in the first place. The process is implemented as follows. When transforming UML class diagrams into Alloy models, a trace of the transformation is produced. At this point, it is possible to use the Alloy analyzer in order to automatically perform a verification of the produced model. This analysis consists of either a simulation or an assertion checking. In the case of an assertion checking, the user checks if a property is valid on a model. If this is not the case, Alloy generates a counter-example (which is an instance of the Alloy model). Using the trace of the transformation in the first direction, the module Alloy To UML instance converter translates the Alloy counter example to a UML object diagram.

This approach offers the possibility of transforming a class diagram (in UML) into a model used by an automatic analyzer in order to perform proofs. It is clear that using an automatic analyzer is simpler than using an interactive prover (which requires a particular expertise of the user). However, the interactive provers are more powerful. Also, this approach offers no way to translate a model to UML class diagrams, it does so only at an instance level.

1.4.2.3 Focal To UML

Focal [39] is a formal language allowing to build certified applications gradually starting from abstract specifications to concrete implementations. This language includes elements inspired from object oriented programming as inheritance and parameterization.

In [36], *Delahaye & al.* describe a formal and sound framework for transforming Focal specification into UML models. It consists in an automatic translation of Focal specifications into UML Class diagrams. This work aims at providing graphical documentation of formal models (described in Focal) to developers. To implement this approach, they started by defining a formal description for a subset of the UML Class diagram, represented by an EBNF grammar of the subset of UML. Then, this UML subset has been extended via the mechanism of profiles (provided by UML) in order to take into account the semantic characteristics of Focal. The transformation rules are then formally presented and used to establish the soundness of the approach. This proof is also presented in the cited paper.

This approach offers a formal and sound environment to transform Focal specifications into UML Class diagrams. This transformation is mainly used as documentation for the developers of the proven system. However, there is no transformation of UML class diagrams into Focal specifications. This would provide a way for developers to interact with the proof expert in order to agree on a solution that would suit both parts.

These methods enable to generate UML components from a formal description and

backwards but their formal representation is significantly different from our needs: functional data structures used in proof assistants.

1.5 Summary

In this chapter we have presented the basic concepts of MDE as well as some related work. To provide the translation of our data types to class diagrams (and vice versa) we use the MDE method, more particularly model transformation. Our transformation is exogenous and goes from a PSM to another PSM. Functional data types (like class diagrams) are self-descriptive. This implies that our transformation is situated on the M3 level of the four-layer meta-modeling architecture. Instances of data types and class diagrams (in Ecore) are located on the level below (M2).

In order to implement our transformation, we use a direct manipulation approach (of model-to-model approaches). This choice has been made after a study of several approaches and testing the tools that implement them.

Our work is situated in a general context of combining formal methods with the MDE. In order to perform this combination, we have chosen the transformation because this technique seems to be the best suited to take advantage of both frameworks for the purpose of defining a Domain Specific Language.

To the best of our knowledge, our study is the only one to propose a fully automatic bidirectional transformation from functional data types to meta-models. Moreover, it permits to use the resulting meta-model to easily generate a graphical or textual editor using Eclipse.

In the following part, we introduce the two frameworks that represent the two directions of the transformation starting by the formalism of our meta-models Eclipse Modeling Framework.

Part I

EMF/Functional Data structures

Chapter 2

Eclipse Modeling Framework

2.1 Introduction

The work we are presenting in this thesis is mainly about the transformation process from data structures used in functional programming into class diagrams. To represent these class diagrams, there exist several languages where UML class diagram is the most famous. We choose to work with Ecore, the core language of Eclipse Modeling Framework (EMF) which is close to UML.

This chapter is structured as follow : First, we start by a general presentation of Eclipse and EMF, before a more detailed definition of Ecore. Then, we focus on the subject used in the thesis. Further, we precise the relation between UML and EMF. Finally, we introduce tools provided by Eclipse Modeling Project (EMP).

2.2 Overview of the Eclipse Modeling Framework

Let us first introduce Eclipse. It is an open source, extensible and polyvalent software project. It provides a highly integrated tool platform for software development as coding activities, modeling, design, testing, etc. Its architecture is based on the notion of plug-in. Each plug-in provides a certain type and number of functionalities in the context of the Eclipse Workbench.

Eclipse Modeling Framework (EMF) is an Eclipse framework for building applications based on model definitions. It unifies three technologies: Java, XML and UML. It allows to describe a model as a class diagram, class interfaces in the Java programming language or in the form of an XML schema. Moreover, it is possible to describe a model in one of these formats and generate it in the two others.

`Ecore` is the model that is used to describe and handle models in EMF. It has been developed as a small and simplified implementation of UML. It is self-descriptive. Its components are further developed in Section 2.3.

To make the rest of the chapter clearer, and explain correctly the following sections, we need to introduce briefly the main components of `Ecore`. They consist of:

- The `EPackage` is the root element in serialized `Ecore` models. It encompasses `EClass`es and `EDataType`s.
- The `EClass` component represents classes in `Ecore`. It describes the structure of objects. It contains `EAttribute`s and `EOperation`s.
- The `EDataType` component represents the types of `EAttribute`s, either predefined (types: Integer, Boolean, Float, etc.) or defined by the user.
- `EReference`s are comparable to the UML Association links. It defines the kinds of the objects that can be linked together. The `containment` feature of `EReference`s is a Boolean value that makes a stronger type of relations.

2.3 Defining EMF Models

2.3.1 Packages

`EPackage` gathers all the `EClass`es and `EDataType`s via the `EReference eClassifiers` (see Figure 2.1). It is the root element in serialized Ecore models. Each `EPackage` has a name and two other values `nsURI` and `nsPrefix`. These values represent respectively the URI and the Prefix of the XML namespace, in the serialization of instance documents. It defines the `EOperation getEClassifier()` that returns an `EClassifier` contained in the current package.

2.3.2 Factories

The `EFactory` is used to create instances of `EClass`es and `EDataType`s contained in the package. It is possible to access an `EFactory` only from an `EPackage` via the `eFactoryInstance` reference.

2.3.3 Classifiers

`EClassifier` is a common base class for `EClass` and `EDataType`. It assembles the features that appear in these two classes. In fact, these classes have similarities: both are target `eType` references. This reference permits `EStructuralFeature`s to define classes or data types as their types. The operation `isInstance` provided by EMF checks whether a Java Object is an instance of the `EClass` or `EDataType` represented by the `EClassifier`.

2.3.4 Classes

EClass is the element that represents the UML class in Ecore. EClasses define the structure of the objects that make up instances of the model. Each EClass has a name and contains typed EAttributes. EClasses are linked via EReferences. It is possible to define EOperations on EClasses. They represent the behavioral features of the EClass (methods that can be defined on the class). For example the EOperation getEStructuralFeature() defined on the class EClass allows to retrieve structural features defined on it.

2.3.4.1 Generalization Link

The generalization link allows to define inheritance. Indeed, an EClass inherits all the structural and behavioral features from its super class. It is represented on the Ecore diagram with the EReference eSuperType. An EClass cannot inherit features from more than one EClass (no multiple inheritance).

2.3.5 Data Types

An EDataType represents a single block of simple data. It illustrates a Java type. It is generally used to model primitive types, or simple types that do not necessitate to be modeled as an EClass. It types the EAttributes that compose the EClasses. In the diagram presented in Figure 2.1, this relationship is modeled by the bidirectional reference between eAttributeType and EAttributes. It defines that each attribute has to be typed and several attributes can be typed with a EDataType.

2.3.6 Enumerated Types

Enumerated data types are a special kind of data types. They represent sets of enumerated values. Each data type is formed of a list of explicit values called literals. It is modeled in Ecore with the EClass EEnum. The EEnum class is composed of a list of EEnumLiterals (in Figure 2.1, the reference EEnumLiterals). Each EEnumLiteral contains a string value: it is the literal value.

2.3.7 Structural Features

EStructuralFeature is a common base for the two classes EReference and EAttribute that defines a state of an instance of EClass. The two classes have similarities: both have a name and a type (represented in Figure 2.1 by the reference eType that has a target to an EClassifier). Also, they count lower and upper bounds to determine the number of feature that compose the instance EClass.

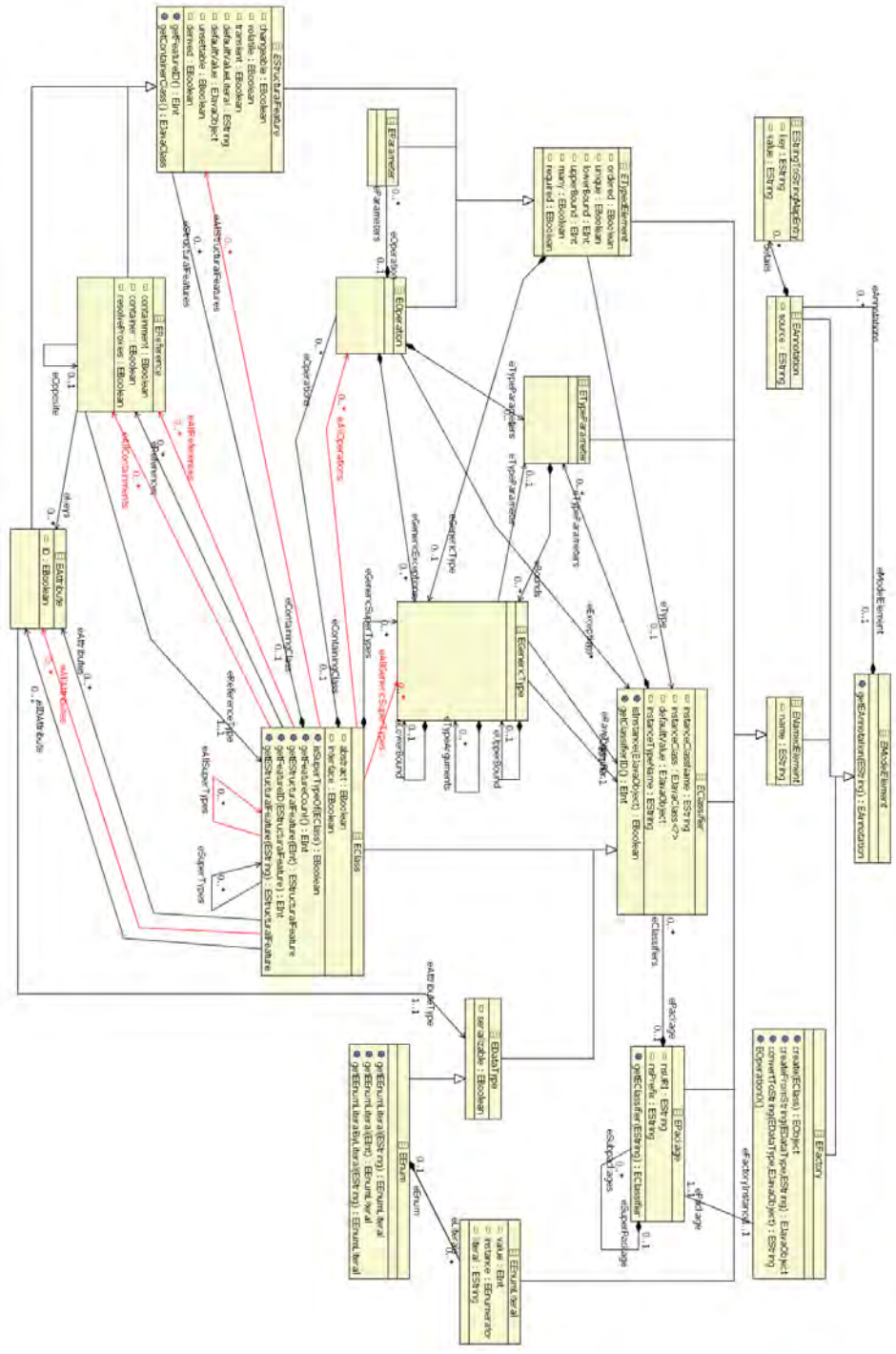


Figure 2.1: The Complete Ecore Meta-model [27]

2.3.7.1 Attributes

Attributes are structural components of a class. They are modeled in Ecore with the `EClass` `EAttribute`. Each attribute has a type; in Figure 2.1, `EAttribute` defines a derived reference `eAttributeType` that refers to an `EDataType`. It has to refer to the same `EClassifier` as the `eType` reference to the `EStructuralFeature`.

2.3.7.2 References

`EReference` adds an attribute to those inherited from `EStructuralFeature`: the containment. When this value is set to true, it becomes a stronger type of association. It represents a whole/part relationship known as "by-value aggregation" in UML. An object cannot contain its own container, it cannot have more than one container, and its life ends with that of its container. `EReference` defines also a reference named `eReferenceType` that target to an EClass. Like in the previous section, it has to refer to the same classifier as the `eType` reference.

2.3.8 Behavioral Features

2.3.8.1 Operations

The `EOperation` is the only way to represent the behavioral feature of an `EClass`. It models only the interface of an operation and does not provide the constructs to express its concrete behavior. Operations are contained by an `EClass` via the `eOperation` containment. `EOperations` may have parameters represented by the `EParameters`. Both `EOperation` and `EParameter` have names, multiplicity and can be typed by an `EClassifier`.

2.3.9 Generics Representation in Ecore

Since Java 5.0, it has become possible to use generics in Java. Ecore had been extended in the same way. This feature augments the flexibility and the re-usability of the code and models. Actually, parameterized types and operations can be specified, and types with arguments can be used instead of regular types. The changes are represented in the Ecore meta-model mainly in two new classes `EGenericType` and `ETypeParameter`. Each `ETypeParameter` has a name.

In Java, there are several ways to use generics (wildcards, type erasures ...), they are modeled as following in Ecore:

First, it is possible to model a Java *wildcard* by not specifying references of the `EGeneric-Type`. In order to express the *extends* construct (respectively *super*), we have to set an `eUpperBound` (respectively `eLowerBound`).

In Ecore, the `eRawType` reference is used to express the Java *erasure* (the mapping from parameterized types to regular types) for unbounded generic types.

For parameterized types, it is represented by a simple `EClass` that contains one or more `ETypeParameters`. An `EGenericType` represents an explicit reference to either an `EClassifier` or an `ETypeParameter` (but not both at the same time). The `eTypeArguments` reference is used to contain the `EGenericTypes` representing the type parameters. When modeling a type parameter that extends a class or a datatype, one uses the `eBounds` reference related to `ETypeParameter`. The reference `eGenericSuperTypes` replaces `eSuperTypes` when it comes to generic types (inheritance).

2.4 Eclipse Modeling Framework and UML

UML has become the de-facto language when it comes to model driven engineering. It supports the idea that complex systems have to be described in different views. UML encompasses several diagrams used to represent each view. Among them, we can mention class diagrams, use case diagrams, activity diagrams ....

The Meta-Object Facility (MOF) standardized by the OMG [75] defines a subset of UML class diagram [45]. It represents the meta-meta-model of UML. `Ecore` is comparable to MOF but quite simpler. We can find similarities in their ability to specify classes, structural and behavioral features, inheritance and packages. However, their difference appear in the data type structures, package relationships and complex aspects of association links. EMOF (Essential Meta-Object Facility) is the new core meta-model that is very close to Ecore [27].

2.5 Features of EMF Used in this Thesis

Figure 2.2 represents a subset of the `Ecore` Language. This subset contains essentially the elements that we needed for the translation to/from functional data structures. In this meta-model appear only basic classes features and operations allowing to keep the expressive power of Ecore. It is also important to note that this subset of Ecore allows us to define basic models that are validated by Ecore.

In Figure 2.2, the constructs that express genericity are distinguishable from the others by the green color. The white `EClasses` are the super classes, from where inherits other classes. It is easy to recognize in the figure the most important elements: `EClass`, `EDataType`, `EReference`, `EAttribute`, etc. This subset contains the main elements allowing to construct an EMF meta-model. Some optional elements are not included. Note that the EClass `ENamedElement` has no particular interest, it only serves to define the EAttribute `name` for EClasses that inherit from it.

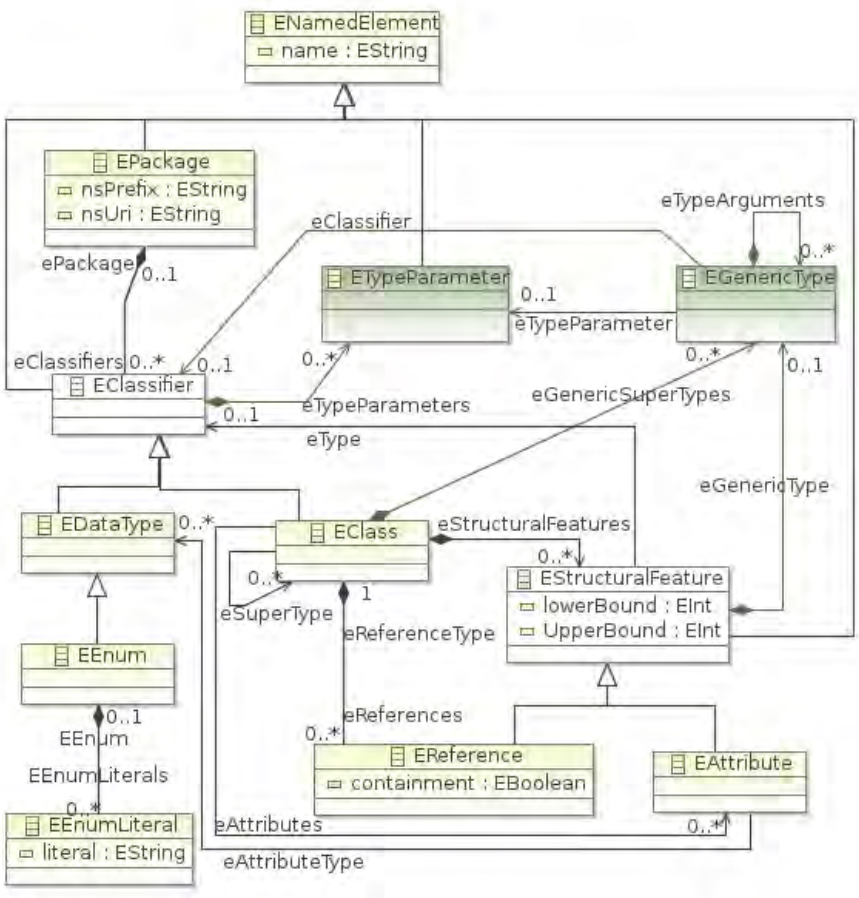


Figure 2.2: Simplified Subset of the Ecore Meta-model

2.5.1 Textual Representation for Meta-models

Additionally to this description, it seems important to represent these constructs in the style of an EBNF grammar. This grammar is presented in Figure 2.3. This description is essential for the rest of the thesis. In fact, in Chapters 4 and 5, we will describe transformation rules using a formal notation. It is therefore necessary to formally determine this subset.

In this grammar, each rule corresponds to an element of the subset we work with. The rule *EPackage* derives a package name followed by a set of `EClassifier`s. An `EClassifier` might give an `EClass` or an `EEnum`. Each `EClass` has a name and defines a set of `EStructuralFeature`s. These are either `EAttribute`s or `EReference`s. The `EEnum` rule provides a set of string values preceded by the keyword `literal`. We define in this grammar four possible predefined types: `int`, `float`, `string` and `boolean`.

EPackage	::=	`ePackage` *Name NsPrefix NsURI* {*EClassifier*} *
EClassifier	::=	{*EClass*|*EEnum*}{*ETypeParameter*} *
EClass	::=	`eClass` *Name* [*ESuperType*] [*EGenericSuperType*] {*EStructuralFeature*} *
EStructuralFeature	::=	{*EReference*|*EAttribute*} *Name LowerBound UpperBound* [*EType*][*EGenericType*]
EType	::=	`eType ="{`*PredefinedType*|`ident}"`
EReference	::=	`eReference` *Containment*
EAttribute	::=	`eAttribute`
Containment	::=	`containment ="`*BoolVal*`"`
EDatatype	::=	*EEnum*| *PredefinedType*
EEnum	::=	`eEnum` *Name* {*EEnumLiteral*} *
EEnumLiteral	::=	`literal=" ident "`
EGenericSuperType	::=	`eGenricSuperType` *Name* [*ETypeParameter*]
EGenericType	::=	`eGenericType` [*Name*] *ETypeParameter**
ETypeParameter	::=	`eTypeParameter` *Name*
PredefinedType	::=	`int`| `float`| `string`| `boolean`
ESuperType	::=	`eSuperType ="ident"`
Name	::=	`name = "ident"`
NsPrefix	::=	`nsPrefix= "ident"`
NsURI	::=	`nsURI= "ident"`
LowerBound	::=	`lowerBound ="{0|1}"`
UpperBound	::=	`upperBound ="{1|*}"`
BoolVal	::=	`true|false`

Figure 2.3: Grammar Allowing to Describe Ecore Models Textually

Example To illustrate the components shown in the previous sections, here is an example of a meta-model (see Figure 2.4). The latter describes simple arithmetic expressions. An Expression is represented by the `EClass` "Expr", it is the super class of 3 other `EClass`es: "Add" for the addition of two expressions (`EReference` link with containment value set to true), "Vars" for named variables and "Consts" for integer constants.

In Figure 2.5, we show the corresponding textual representation for the meta-model presented in this example. To write this description we used the rules of the grammar defined in Figure 2.3.

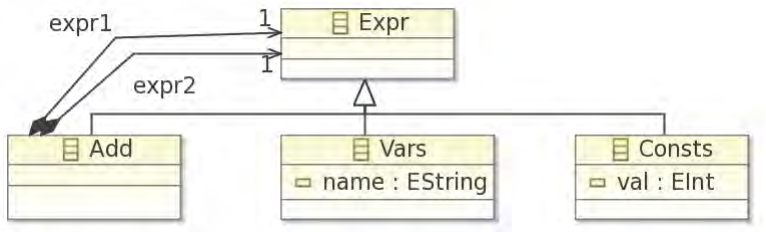


Figure 2.4: Ecore Meta-model for an Arithmetic Expression

```
ePackage name = "Expression" nsPrefix= "ident" nsURI= "ident"
 eClass name ="Expr"
 eClass name ="Add" eSuperType ="Expr"
  eReference containment ="true" name ="expr1" lowerBound ="1" upperBound
  ="1" eType ="Expr"
  eReference containment ="true" name ="expr2" lowerBound ="1" upperBound
  ="1" eType ="Expr"
 eClass name ="Vars" eSuperType ="Expr"
  eAttribute name ="name" lowerBound ="1" upperBound ="1" eType =string
 eClass name ="Consts" eSuperType ="Expr"
  eAttribute name ="val" lowerBound ="1" upperBound ="1" eType =int
```

Figure 2.5: Textual Representation of the Arithmetic Expression Model

2.6 Eclipse Modeling Project

Eclipse Modeling Project (EMP) is a project that has appeared following EMF. It has been created in order to ease handling domain specific languages. It offers tools that permit to define the textual/graphical concrete syntax of DSLs based on an Ecore meta-model. Then it is possible to generate a DSL editor as an Eclipse plug-in. In this section, we present two of these tools: GMF and Xtext.

2.6.1 Graphical Modeling Framework

Graphical Modeling Framework (GMF) [51, 6] is a framework created to facilitate the building of graphical Eclipse based editors. This technology constitutes a bridge between EMF and Graphical Editing Framework (GEF): a platform for building graphical editors based on Model-View-Controller (MVC) architecture.

The MVC architecture [34] is traditionally used in the development of applications with graphical interfaces. It aims to keep the essence of an application separate from its

interface. It consists of three main parts :

- Model part for containing data which supports the underlying problem.
- View part for displaying the data.
- Controller part for handling events and ensuring communication between the view and the model.

When an application is structured with an MVC architecture, its view part is represented by the graphical interface.

The aim of the MVC architecture is to not overload users of an application with information they do not need. For example, when developing an application, the model part contains all the data necessary for the application, while the view part (represented by the GUI manipulated by the user) is responsible for displaying information necessary for the user. The controller part is then is in charge of the exchange of information between the model and the view.

2.6.1.1 GMF Architecture

The MVC pattern separates the model elements (containing data) from the user interface. The communication between the two parts is made by the controller. In an editor generated with GMF, the Model part is represented by the *model code*, as the Controller part represented by the *edit code*. Both are carried out with the generation code facilities provided by EMF. The *diagram code* (view part) is generated thanks to the GMF facilities. The GMF generator model is built after selecting the domain (EMF) model and informing some definitions about the editor. It is important to note that the MVC architecture which GMF is based on, is completely transparent to the user of GMF. The generation and management of the three parts is completely supported by EMF and GMF.

The generated code (for model, edit and diagram) represents a GEF project. Figure 2.6 illustrates the GMF architecture. The green arrows represent an automatic generation. The bidirectional black ones define communication between the different parts.

In Chapter 6, we present a case study of our transformation process from functional data structures into EMF meta-models. This case study consists in the representation of Binary Decision Diagrams. The resulting meta-model is used with GMF to represent this DSL in a specific graphical editor. There is thus in Chapter 6, an example of the use of GMF.

2.6.1.2 Using GMF

Figure 2.7 presents the main components and models used on GMF and shows how they interact when using GMF.

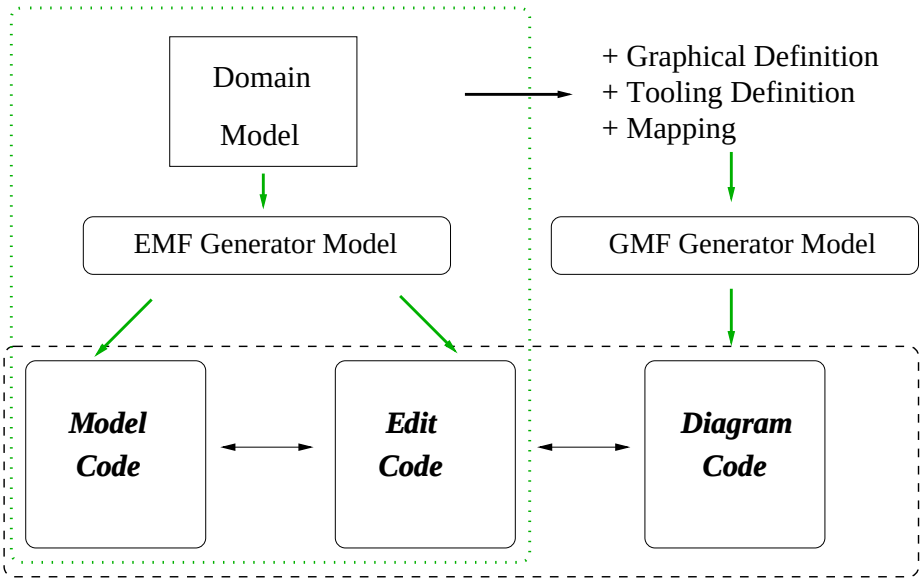


Figure 2.6: GMF Project Architecture

When a GMF project is created, it is referenced by a `domain model`. Then, one has to provide three different definitions, specific to GMF. The first one is the *graphical definition*, it is used to define the graphical elements that will appear in the generated DSL editor (nodes shapes, connections, labels, ...). It answers the question: How to shape the elements of our diagram and how to connect them. Then, the *tooling definition* enquires the information about the tools of the DSL editor (components of the palette). The *mapping definition* is used to map graphical (resp. tooling) definitions and the *domain model*. Once these elements correctly communicate together, we can generate the *diagram code*, using the *GMF generator model*. The latter generates diagram code automatically.

2.6.2 Xtext

Xtext [40] is a tool that supports the development of textual concrete syntax for DSLs. In the first versions of Xtext, it was only possible to create a DSL textual editor starting from an Extended Backus-Naur Form-like grammar and generating a corresponding Ecore-based meta-model. But since Xtext 2.0, it is possible to start from a meta-model and get the corresponding EBNF-like grammar. Starting from this grammar, the generator creates a parser as well as a functional Eclipse textual editor, complete with syntax highlighting, code assist and outline view [51].

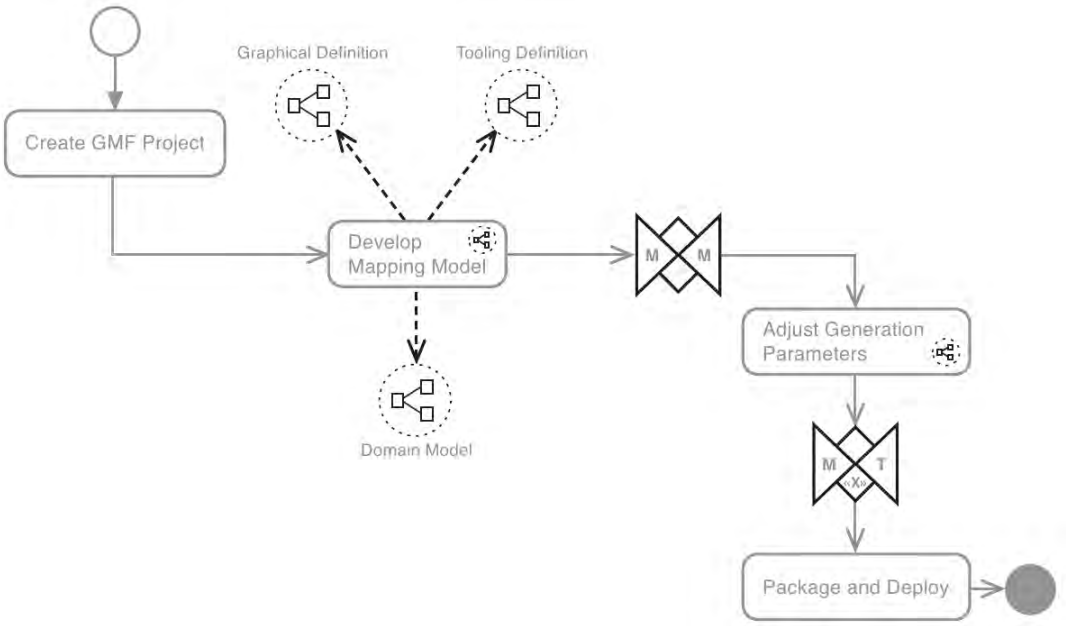


Figure 2.7: GMF Workflow [51]

2.7 Summary

In this chapter, we presented a detailed definition of the Eclipse Modeling Framework. Then, we selected the features used in our transformation process. We also introduced tools for defining and generating support for textual/graphical syntaxes to develop Domain Specific Languages based on an EMF meta-model. This framework is the entry point of the first direction of our transformation process (see Chapter 4). Next, we'll define the other part of the transformation process, consisting of the formal framework: functional programming.

Chapter 3

Formal Framework

3.1 Introduction

In this chapter, we define one of the two parts (source/target) of the transformation process. It consists of a formal framework allowing description of DSLs in order to represent them formally and perform verifications on them.

Figure 3.1 shows the global context we work in. The arrows represent automatic functions. The dashed ones show a transformation that could be implemented managing some restrictions on the language used to describe data types, whereas the basic arrows show an automatic transformation that is operationally implemented in this thesis. In this figure, we have on the one hand the abstract syntax of data type descriptions (presented in Section 3.3) which is common to several ML Languages (as Caml) and interactive provers like Isabelle. This syntax is an entry point for our transformation process. To be able to use it with different languages, it suffices to create a parser (and a serializer) that corresponds to the concrete syntax of each language. We have implemented these parsers (and serializers) for the Caml language and for the Isabelle prover. It could be possible to develop a parser (and a serializer) for the Coq proof assistant also, but its language would be restricted to a subset of the syntax of Coq, prohibiting the use of type dependency.

In the next sections, we adopt a convention for describing syntax. The latter is presented using grammars. Non-terminals are represented in *italic* style. Terminals, for their part, are represented in the `typewriter` style. The square brackets [] express an optional component. As for the curly braces, they contain elements that could be repeated several times.

This chapter is organized as follows: first, we start by presenting functional programming and clarifying its relation with interactive provers. Then, we define the abstract syntax common to ML languages and Isabelle. Next, we determine the subset we work with before constructing the corresponding meta-model for this subset.

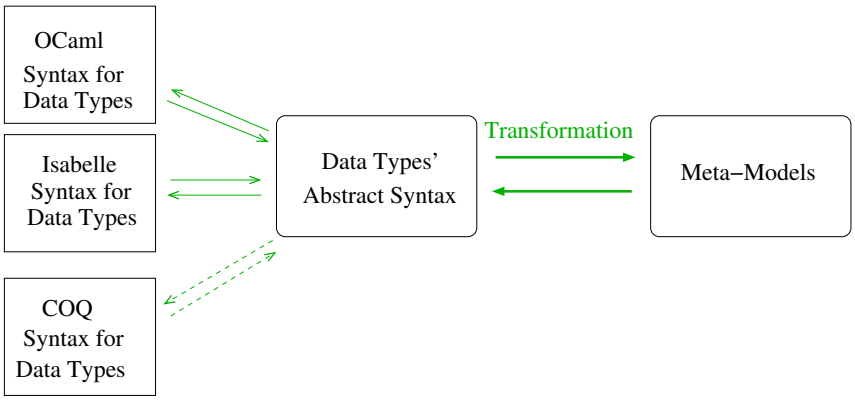


Figure 3.1: Overview of the Transformation

3.2 Functional Programming

Functional programming is a programming paradigm based on the mathematical notion of *function*. It implements λ-calculus: a formal language in mathematical logic that formalizes systems through the notion of function. A function, in functional programming, consists in the mapping of elements from a set to another. These sets are called *types*. Usually, they give indication about the correctness of programs. We can count among the languages implementing functional programming :

- *Lisp* [89] is the oldest family of functional languages. It is distinguishable by its simple syntax based on linked lists and prefixed notation. It is also characterized by its dynamic type system. From its inception, it has been widely used particularly in Artificial Intelligence. The best known dialects belonging to the Lisp family are *Common Lisp* and *Scheme*.

- *Haskell* [81] is a purely functional programming language. In contrast to *Lisp*, it has a strong static type system and can be used without type annotations. It is rather close to the ML languages. Its strength lies in its type classes: structures originally conceived to handle ad hoc polymorphism.

- *ML languages*: ML stands for Meta Language. It is based on a user-friendly syntax of λ-calculus augmented with polymorphism. It is known for its ability to automatically infer the types of most expressions without explicit type annotations. ML languages are considered as non-purely functional languages. In fact, they can use mutable data structures, features allowing to program in an imperative way. The most famous dialects of the ML family are SML (Standard ML) and OCaml (Objective Caml) [68].

3.2.1 Interactive Provers

An interactive theorem prover is a tool used to assist the user in order to perform some formal proof on a system. The proof checks whether the computer program meets some specification in a formal logic. It can also verify the nature of the relations between components (checking if a component refines another one, for example). These specifications ensure the correctness of software which is an important requirement for critical programs.

Isabelle [73] and *Coq* [29] are well-known specification and verification systems. These proof assistants are implemented in an ML language and consist in a combination of *functional programming* and *mathematical logics*.

The functions in these powerful verification programs are not directly executable, but an execution can be simulated by rewriting. To overcome this shortcoming, it is possible to automatically generate executable and certified code in Caml. Additionally, from an Isabelle *theory* (file gathering Isabelle code parts), it is possible to generate executable and certified object oriented code in Scala [78].

3.3 Abstract Syntax of ML Languages

In this section, we present the common abstract syntax for ML languages and Isabelle. Each of these languages has its own way to describe programs, but they have similarities in their abstract syntax. We illustrate the differences between ML and Isabelle when it seems necessary. In ML programs, it is possible to group portions of programs into a *module*. In Isabelle, it is called a *theory*.

3.3.1 Data Types

Historically, types were created to predict the amount of space needed to allocate a variable, nowadays they became essential elements in the definition of programs. In fact, they can be very useful in documenting programs, but they give also information about the correctness of programs (such as in functional programming). When writing a program, some of these types are predefined and implemented by default due to their frequent use, some others have to be defined by the user.

3.3.1.1 User Defined Data Types

Users can define their own data types by means of type definitions. These type definitions include records and variant types. They are introduced by the `type` declaration in OCaml, as in Isabelle by the `datatype` keyword.

Figure 3.2 depicts the detailed abstract syntax of type definitions that could be defined by the user. Every particular case presented next is taken into account in this syntax.

Type equation In the first case, a data structure description is represented by a *type equation*. It is the simplest aspect of a type definition. The equation consists of a type constructor defined by a type expression.

Records A record is a type containing different named and typed fields. It represents the shape of a particular data structure. In Isabelle, a record type declaration is introduced by the keyword `record`. In each field declaration, the field type follows the field name and is introduced by two successive colons (`::`). Record instances are presented in field declarations contained inside pairs of parentheses accompanied by vertical bars `(| |)`. The field declarations are separated with commas (`,`).

In Caml, a record type declaration is introduced with the same keyword as other user defined data types: `type`. In every field declaration, the type of a specific field is separated from its name with a colon `:`. When instantiating a record, fields are put between curly braces (`{}`) and separated by semi-colons (`;`). To access a specific field of the data structure, we use the `"data_type.field_name"` notation.

Variant Types Variant types are used to represent disjoint unions of types. They show all the possible shapes of values for a type. Each possible shape is identified by a *constructor*. These constructors are used to build a value of a type or to perform the inspection of a value with the pattern matching. Each constructor can take *type expressions* to construct the case of variant type. In these type expressions it is allowed to use either a predefined type or a user defined one.

It is also allowed to use the same type in the type definition that is currently described. It is then a recursive data structure. The most common usage of variant types is for this kind of data type. Moreover, it is possible to define mutually recursive data types. It is two datatype definitions separated by the keyword `and`, where each of them is used in the type expression of the other.

Additionally, there is a way to represent genericity by using parameterized data types. They are used to express generic data structures. They permit to build different data structures that accept any kinds of values. Each definition of a parameterized type is formed of a *type constructor* and a set of *type parameter*s. The type expressions then can contain a previously defined parameterized type or one of the specified parameters.

3.3.1.2 Predefined Data Types

ML languages and Isabelle offer the usual basic data types: integers, floating-point numbers, Booleans, characters and strings.

In addition to these primitive types, some frequently used data structures have been implemented and are by default present in the languages. They have particular keywords and notations. Among them, we mention lists, type option and the references.

type-definition ::= *typedef* {`and` *typedef*}
typedef ::= [*type-params*] *typeconstr-name* [*type-information*]
type-information ::= [*type-equation*] [*type-representation*]
type-equation ::= `=`*typexpr*
type-representation ::= `=`*constr-decl* { | *constr-decl* }
| `= (` *field-decl* {`;` *field-decl*} `)`
type-params ::= *type-param*
| `(`*type-param* { , *type-param* }`)`
type-param ::= `'ident`
constr-decl ::= *constr-name*
| *constr-name typexpr* { `*` *typexpr*}
field-decl ::= *field-name* : *typexpr*
| `mutable` *field-name* : *typexpr*
typexpr ::= *type-param*
| _
| `(`*typexpr*`)`
| *typexpr* → *typexpr*
| *typexpr* { *typexpr* }
| *typeconstr-name*
| *typexpr typeconstr-name*
| `(`*typexpr* {`,`*typexpr*}`)` *typeconstr-name*
typeconstr-name ::= `ident`
constr-name ::= `ident`
field-name ::= `ident`

Figure 3.2: Syntax of Type Definitions in Caml [68]

Lists Lists are predefined data structures which are very suitable for functional programming. Because of their definition by induction, they are particularly easy to handle. In fact, the structure of a list is implemented as an empty list (Nil) to which one can add elements (one at a time) using the `cons` (`::`) operator. To facilitate their utilization in programs, a more convenient syntax has been developed in the different languages. In Caml, they are represented as a bracketed list of semicolon-separated elements. The same in Isabelle, the elements are separated with commas.

References Pointers are represented in ML by references. As for all the imperative features, they are implemented using a mutable data structure. More precisely, their implementation consists of a one-field mutable record. It is introduced in Caml by the keyword `ref` and implemented as presented next.

```
type 'a ref = { mutable contents:  'a }
```

In Isabelle, references are not implemented but they can be defined in the same way as in Caml. In his thesis [48], *Giorgino* worked on the specification and refinement of pointer structures in Isabelle.

Arrays In the predefined data structures we can also find Arrays. These are mutable data structures borrowed from imperative programming. Actually, in imperative programming, it is natural to write a value into a memory location. This is a contrast to the vision of functional programming. They became essential elements in computer science programs. This is why, they were implemented, even if it is frequently more efficient to use lists instead of arrays in Caml. An Array in OCaml can be given between `[|` and `|]` brackets or initialized with the key word `Array` .

Type Option *Type option* is used to type a value that could be absent or present. Type option returns `None` when the value is absent and `Some` of the type otherwise. This notion is present in ML languages as in Isabelle. It is by default implemented in Caml as follows:

```
type 'a option = None |Some of 'a
```

3.3.2 Functions

The concept of function in functional programming is very similar to that of mathematical function. A function is considered as a mapping from a set to another one.

In the body of a function, it is possible to use the basic `if-then-else` statement and the `for` and `while` loops. Additionally, a particular feature is provided: the pattern matching. It allows to indicate a special action according to the shape of the value.

When a function is partly defined by itself and calls itself with other arguments, it is a recursive function. It is often used in functional programming and is introduced in Caml by the key words `let rec`. As example of recursive functions we can cite the function used to compute the factorial of a number. Indeed, the `fact` function calculates the factorial of a number `n` by performing the product of n and the factorial of its predecessor. The recursion stops when the `n` number equals `0`.

Example

```
 let rec fact n = match n with
 0 -> 1
|_ -> n * fact (n-1) ;;
```

We do not spend too much time on the description of syntax of functions because it does not enter into the translation process that we define, except for the case of accessor functions. The main focus of this thesis is about the structural part of programs.

3.4 Features of Functional Programming Used in this Thesis

Our translation does not treat all of the features typically present in functional programming languages such Isabelle and Caml. The primary reason is that some features which are specific to functional programming have no counterpart in Ecore. This is particularly true of higher-order constructors, *i. e.* constructors taking functions as arguments. This is why we defined a subset that both contains the essential elements composing data types and can be translated into class diagrams (and vice versa).

3.4.1 Part of Caml Grammar Used in this Thesis

Figure D.1 presents the concrete syntax of the data types used in our translation described in the Caml language. We define this grammar starting from the grammar presented in Figure 3.2 managing some revisions. The changes performed are spelled out in the following paragraphs.

First, we add to this syntax a way for the user to differentiate enumerations from other data type definition. This option is introduced as the `(**Enumerated*)` comment and concerns variant types having constructor declarations without type expressions. This comment has to be placed immediately before the concerned type definition. It could be possible to automatically translate this kind of type definitions without adding the comment, but we decided to leave the choice to the user about how he would treat this kind of data types: as enumerations or as basic data type definitions. We also remove the ability to define *type-equations* as *type-definitions*. It constitutes one particular case that is not essential to the data types description. In fact, this shape of data types can easily be replaced by a *type-representation* with a single constructor declaration (*constr-decl*).

For more clarity of the transformation rules presented in Chapter 4, we split the rule *type-representation* into two rules *variant-type-rep* and *record-rep* respectively for variant types representation and record representation.

We run some modifications on the rule defining type expressions. This rule allows a large number of combinations that can make the translation more complex. We broke this recursion in order to reduce the number of possible forms for type expression. This rule is replaced in our revised grammar by two sub rules: *comp-typexpr* and *typexpr*. The first rule allows to introduce compound type expressions represented in simple type expressions followed by keywords introducing lists, references and type option. The *typexpr* rule defines the shape that can take a simple type expression restricting among others primitive types to integers, Booleans, floats and strings.

In this subset we exclude some mutable data structures, in particular arrays. Also, for now, we have not implemented a treatment for mutually recursive types, except for the list, reference and option type constructors. Genuine mutual recursion considerably complicates the transformation procedure, but apart from the exceptions mentioned, only occurs rarely in practice.

module	::=	`Module` *module-name* {*type-definition*}
type-definition	::=	*typeDef*
	|	*typeEnum*
typeEnum	::=	`(**Enumerated*) type` *typeconstr-name* `=`*constr-name* { | *constr-name* }
typeDef	::=	`type` [*type-params*] *typeconstr-name* `=` *type-representation*
type-representation	::=	*variant-type-rep*
	|	*record-rep*
variant-type-rep	::=	*constr-decl* { | *constr-decl* }
record-rep	::=	`{` *field-decl* {`;` *field-decl*} `}`
constr-decl	::=	*constr-name*
	|	*constr-name* `of` *comp-typexpr* {`*` *comp-typexpr*}
field-decl	::=	*field-name* : *comp-typexpr*
comp-typexpr	::=	*typexpr*
	|	(*typexpr* `option`)
	|	(*typexpr* `list`)
	|	(*typexpr* `ref`)
type-params	::=	`(`*type-param* { , *type-param* }`)`
typexpr	::=	*type-params typeconstr-name*
	|	*typeconstr-name*
	|	*type-param*
	|	*prim-type*
prim-type	::=	`int` | `float` | `bool` | `string`
type-param	::=	`'`*lowercase-ident*
module-name	::=	*capitalized-ident*
typeconstr-name	::=	*lowercase-ident*
field-name	::=	*lowercase-ident*
constr-name	::=	*capitalized-ident*
lowercase-ident	::=	(`a...z`){`a...z`|`A...Z`|`0...9`}
capitalized-ident	::=	(`A...Z`){`a...z`|`A...Z`|`0...9`}

Figure 3.3: Caml Grammar of Data Types Used in this Thesis

In sum, the syntax presented in Figure D.1 starts with a *module* description. It states that a *module* has a name and encompasses several *type definition*s. Each *type definition* can be a type enumeration or basic type definition. Each type definition *typeDef* consists of a named *type constructor* that can be parameterized. Then, the data type representation consists of a variant type or a record. A variant type is formed of at least one *constructor declaration*. These declarations have a name (*constr-name*) which is the name of a partic-

ular type case. It takes as argument some (optional) *type expression*s. When it is a record, a type definition is presented as a list of separated *field declarations*. Every field is named and typed with a *type expression*. *Type expressions* can appear in a compound representation. A compound type is a simple type with one of the additional features: The *list* predefined data structure, the *type option* and the references *ref*. Simple *type expressions* can either be a *primitive type*, a user defined data type (that can be parameterized) or a type parameter.

3.4.2 Proposed Extension for Accessor Functions

In the abstract syntax presented in Figure 3.2, we can notice that elements composing type definitions are often unnamed and just expressed with *type expressions*. The unique exception is remarkable in the record case where the fields composing a type definition have a name and a type. However, for the rest of our work these typed elements have to be differentiable by their names. Therefore, we enriched the type definition grammar with a new element named *Accessor*. It is a function introduced by a special annotation `(*@accessor*)`. It allows to assign a name to a special part of the type declaration. These accessor functions are essential for the transformation process, their absence would lead to nameless `EStructuralFeature`s. The syntax of these functions in the Caml language is presented in Figure D.2.

In this syntax, *acc-name* is the name of the accessor and n is the number of type expressions composing the type for the constructor *constr-name*. The i^{th} value represents the position of the typed element to which the name is assigned. If the type definition is a *type equation*, there is no *constr-name*.

```
(*@ accessor *)
      let  acc_name_i  (constr-name  (x_1, ..., x_n) ) =  x_i      /  1 ≤ i ≤ n
```

Figure 3.4: Syntax of Accessor Functions in Caml

3.4.3 Example of a Data Type Definition

To illustrate the subset we work with, here is an example of a data type definition written in the OCaml language and that is accepted by our subset. To guarantee some coherence, we take the same example as in Section 2.5 and represent a simple arithmetic expression in the Caml language.

In Figure 3.5, the type of an arithmetic expression *"expr"* is represented by a recursive and variant data type. The expression can appear in three different shapes. The addition represented by the *"Add"* constructor can take two expressions. An expression could be also a variable *"Vars"* that has only one field typed as a string. The last shape is the constant *"Consts"* of integers.

The functions *expr1* and *expr2* are used to name respectively the first and the second expression of the addition *Add*. The same for *name* and *val* for respectively the *Vars* and *Consts*.

```
type expr = Add of expr * expr
          | Vars of string
          | Consts of int

(*@accessor*)  let expr1 (Add x y) = x
(*@accessor*)  let expr2 (Add x y) = y
(*@accessor*)  let name  (Vars x)  = x
(*@accessor*)  let val   (Consts x) = x
```

Figure 3.5: Data Type "expr" and its Accessor Functions in Caml

3.4.4 Part of Isabelle Grammar Used in this Thesis

In the introduction of this chapter, we state that our formal model is constructed from a subset of the abstract syntax which is common to ML languages and Isabelle. Figures D.1 and D.2 represent this subset of data types and accessors functions syntaxes in the Caml language. Figures 3.6 and 3.7 represent the equivalent syntaxes in Isabelle.

3.5 Meta-model of the Formal Framework

Because it is needed in the following chapters of this thesis, we define a meta model corresponding to the grammar of functional languages used in this thesis (and presented in Section 3.4). To do this, we are somewhat inspired by the work of [5] and [98]. They worked widely on defining generic processes to transform EBNF grammars into meta-models and vice-versa. We mainly focused on the definition of transformation rules and the correspondence between the elements of the two formalisms. However, we did not use any tools or algorithms developed.

We used the same intuition concerning the translation of repetition into `0...*` multiplicity. As for the notion of option, it is translated into `0...1` multiplicity. The alternative set up with the `|` operator gives inheritance in the target meta model. Regarding to the `::=` operator, it is established in the meta-model with the composition link. We then adapted the result in order to ensure that is corresponds exactly to the semantics of our syntax.

The resulting meta-model is encoded in Ecore. Figure 3.8 depicts the datatype meta-model constructed from a subset of data type's declarations grammar presented in Fig-

module	::=	`theory` *module-name* `begin` {*type-definition*} `end`
type-definition	::=	*typeDef*
	|	*typeEnum*
typeEnum	::=	`(**Enumerated*)` `datatype` *typeconstr-name* `=`*constr-name*
		{ | *constr-name* }
typeDef	::=	`datatype` [*type-params*] *typeconstr-name* `=` *variant-type-rep*
	|	`record` *typeconstr-name* `=` *record-rep*
variant-type-rep	::=	*constr-decl* { | *constr-decl* }
constr-decl	::=	*constr-name*
	|	*constr-name comp-typexpr* { *comp-typexpr*}
record-rep	::=	*field-decl* {*field-decl*}
field-decl	::=	*field-name* :: *comp-typexpr*
comp-typexpr	::=	*typexpr*
	|	(*typexpr* `option`)
	|	(*typexpr* `list`)
	|	(*typexpr* `ref`)
type-params	::=	`(`*type-param* { , *type-param* }`)`
typexpr	::=	*type-params typeconstr-name*
	|	*typeconstr-name*
	|	*type-param*
	|	*prim-type*
prim-type	::=	`int` | `float` | `bool` | `string`
type-param	::=	`'`*lowercase-ident*
module-name	::=	*capitalized-ident*
typeconstr-name	::=	*lowercase-ident*
field-name	::=	*capitalized-ident*
constr-name	::=	*capitalized-ident*
lowercase-ident	::=	(`a...z`){`a...z`|`A...Z`|`0...9`}
capitalized-ident	::=	(`A...Z`){`a...z`|`A...Z`|`0...9`}

Figure 3.6: Isabelle Grammar of Data Types used in this Thesis

```
(*@ accessor *)
      fun    acc_name_i :: typeconstr-name => typexpr_i
      where  acc_name_i (constr-name (x_1, ..., x_n)) = x_i      /  1 ≤ i ≤ n
```

Figure 3.7: Syntax of Accessor Functions in Isabelle

ure D.1.

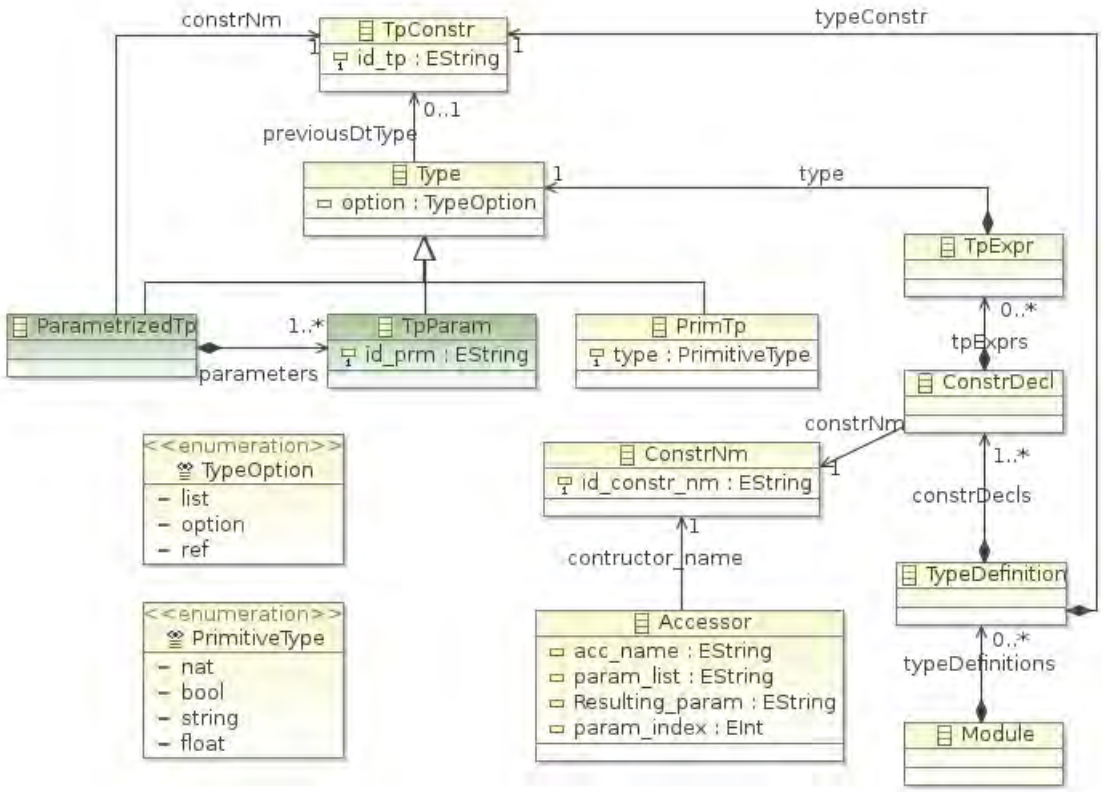


Figure 3.8: Datatype Meta-model

3.6 Summary

This chapter presents the formal framework we deal with in this thesis: *functional languages*. We have started by presenting the general framework of functional programming, we have then shown the relationship between functional programming and interactive provers. Subsequently, we have defined the subset that we work with in this thesis and finally have created a corresponding meta-model. Now, let's discuss the transformations performed between this formal framework and the Eclipse Modeling one.

Part II

From Functional Models to Meta-models and Back Again

Chapter 4

Functional Models to EMF

4.1 Introduction

In this chapter we specify the transformation process used to translate elements from the subset of functional data types (presented in Chapter 3) into class diagrams. First, we start by defining the constraints needed to guarantee the correctness of the transformations, then we present the shape of the transformation rules before detailing them one by one. We ensure that we give an example in each rule case.

4.2 Well-formedness Constraints for Input Data Types

In Section 3.4, we defined a subset of the Caml language that can be applied to our transformation rules. To guarantee the correctness of the transformed elements, we have to impose some well-formedness constraints on the input subset of data type definitions (and accessor functions). The main constraint consists in observing the well-formedness constraints of Caml programs. In fact, each input component must not generate any error when it is evaluated by the Caml interpreter. Among all these constraints, ones we are interested in are:

- Ordering of definition of data types: Any data type used to define another data type must appear before it (see Example 1). For mutually recursive data types, there is a particular description, however, the latter is not included in the subset of data types that we have defined.
- Using the same number of type parameters (in type expressions) as defined on the type constructor parameters (see Example 2).
- Using only type parameters (in type expressions) that are defined on the type constructor parameters (see Example 3).

Example 1 The following type definition is not accepted. The type definition tp_1 must be placed before the type definition tp_2.

$$\begin{array}{ll} \texttt{type } tp_2\texttt{=} & \text{Tp2 } \texttt{of } tp_1 \\ & |... \\ \texttt{type } tp_1\texttt{=} & ... \end{array}$$

Example 2 This definition is also unacceptable due to a type mismatch. The type tp_1 has three type parameters and in the type definition tp_2 it is used with only two.

$$\begin{array}{ll} \texttt{type } ('a,'b,'c)\ tp_1\texttt{=} & ... \\ \texttt{type } tp_2\texttt{=} & \text{Tp2 } \texttt{of } ('a,'b) tp_1 \\ & |\ ... \end{array}$$

Example 3 For this case, it is not acceptable because the type parameter $'d$ is undefined. It is only allowed to use $'a$, $'b$ or $'c$ parameters.

$$\begin{array}{ll} \texttt{type } ('a,'b,'c)\ tp_1\texttt{=} & \text{Tp1 } \texttt{of } ('d) * \ ... \\ & |\ ... \end{array}$$

4.3 Transformation Rules Representation

The global transformation method is named *Tr* and takes parts of data type descriptions and returns Ecore elements. The transformation rules are presented as sub-functions relatively to the component given as input. In each rule definition, we start by an informal description, then we present it formally and finally we show an effective example.

$$Tr : DataTypes \longrightarrow Ecore\ Meta\text{-}model$$

In the following sections, transformation rules are presented as sub-functions and given for a concrete syntax in the style of Caml [68]. Since most functional languages (including the language of proof assistants) have great similarities, the concrete syntax can be mapped to different functional languages.

In Table 4.1, we unfold the grammar presented in Section 3.4 in order to show that each element of the grammar is covered by a rule. In the first column of the table, we display the elements of the grammar whereas in the second, we show the applied rule. We also reference the section where the rules are defined.

We also set priorities for the rule application in order to avoid ambiguity. For example, we find this case for the part of the grammar allowing to translate *typeDef*. In fact, when *typeDef* is formed of a *typeconstr-name* and a single *constr-decl*, there are two possible rules that could be applied for this part of the grammar **DatatypeToEClass** and **DatatypeToEClasses**. This is why the rule **DatatypeToEClass** has a higher priority than **DatatypeToEClasses**.

Grammar Parts	Rule
module::= `Module` *module-name* {*typeDef*|*typeEnum*}	ModuleToEPackage (Section 4.4)
typeEnum::= `(**Enumerated*)` `type` *typeconstr-name* `=`*constr-name* {|*constr-name*}	DatatypeToEEnum (Section 4.6)
typeDef::= `type` *typeconstr-name* `=` *constr-decl*	DatatypeToEClass (Section 4.5)
typeDef::= `type` *typeconstr-name* `=` *variant-type-rep*	DatatypeToEClasses (Section 4.7)
typeDef::= `type` *typeconstr-name* `=` *record-rep*	RecordToEClass (Section 4.8)
typeDef::= `type` *type-params* *typeconstr-name* `=` (*variant-type-rep* | *record-rep*)	Transforming Generics (Section 4.15)
typexpr::=*prim-type*	PrimitiveTypeToEAttribute (Section 4.9)
typexpr::=*typeconstr-name*	TypeToEReference (Section 4.10)
typexpr::= *type-params typeconstr-name*	Transforming Generics (Section 4.15)
typexpr::=*type-param*	Transforming Generics (Section 4.15)
comp-typexpr::=(*typexpr* `option`)	OptionToMultiplicity (Section 4.11)
comp-typexpr::=(*typexpr* `list`)	ListToMultiplicity (Section E.2.3)
comp-typexpr::=(*typexpr* `ref`)	RefToEReference (Section 4.13)

Table 4.1: Correspondence between Grammar and Transformation Rules

4.4 Rule ModuleToEPackage

In ML programs (respectively in the Isabelle proof assistant), it is possible to group portions of programs into *modules*. We decided to represent these modules by `EPackage`s in `Ecore`. They are used to gather `EDataTypes` and `EClasses`. Thus, the transformation process consists in creating an `EPackage` for each module. The name of the corresponding `EPackage` is the module name. We have also to specify the prefix and the URI of the XML namespace by instantiating the `NsPrefix` and `NsURI` values. To translate the data types contained in

the module, we call the function $Tr_{dtp}()$ for each type definition.

$$\begin{aligned} Tr_{module}(Module\ md\_name\ Dtp_1...Dtp_n) = \\ & createEPackage(); \\ & setName(md\_name); \\ & setNsPrefix(md\_name); \\ & setNsURI(\text{"}http://md\_name/1.0\text{"}); \\ & Tr_{dtp}(Dtp_i) \qquad / \ 1 \leq i \leq n \end{aligned}$$

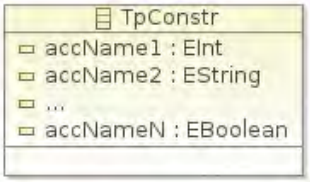
4.5 Rule DatatypeToEClass

The rule we are presenting, is the most simple case. It is applied when the data type is formed of only one constructor. The data type description is then transformed to an EClass. For this purpose we call the `createEClass()` function. The class name is set as the type constructor name of the type definition. Next to transform each type expression composing the constructor declaration, we call the $Tr_{type}()$ function. This function translates types corresponding to their nature using *PrimitiveTypeToEAttribute* or *TypeToEReference rule*.

$$\begin{aligned} Tr_{dtp}(tpConstr = cn\ t_1...t_n) \quad = \quad & createEClass(); \\ & setName(tpConstr); \\ & Tr_{type}(acc_i, t_i) \\ & / \ 1 \leq i \leq n \end{aligned}$$

Example:

```
type tpConstr =
Cn of int * string * ...*
bool
```



4.6 Rule DatatypeToEEnum

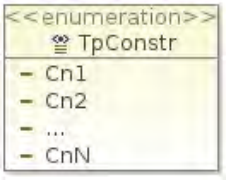
The second case that we deal with is when data types are formed of enumerations. This kind of type definitions is composed only of constructors without type expressions (*typexpr*) and are introduced by the comment `(** Enumerated*)`. It is translated to an `EEnum`. This concept is usually employed to model enumerated types in `Ecore`. The type definition's name becomes the `EEnum` name. Then, each constructor composing the type definition is

translated into a literal named `EEnumLiteral` of the created `EEnum`. The name of each constructor becomes the name of each a literal.

$$
\begin{array}{lcl}
Tr_{dtp}(tpConstr = cn_1|...|cn_p) & = & createEEnum(); \\
 & & setName(tpConstr); \\
 & & Tr_{constrNm}(cn_i) \quad / \ 1 \leq i \leq p \\
Tr_{constrNm}(cn_i) & = & EEnumLiteral(cn_i) \quad / \ 1 \leq i \leq p
\end{array}
$$

Example:

`type` *tpConstr*=
Cn1 |*Cn2*|... | *CnN*



4.7 Rule DatatypeToEClasses

This rule is the most important rule in our translation process. It consists in transforming type definitions in their general case representation into a hierarchy of classes (except the two special cases previously presented in Sections 4.5 and 4.6).

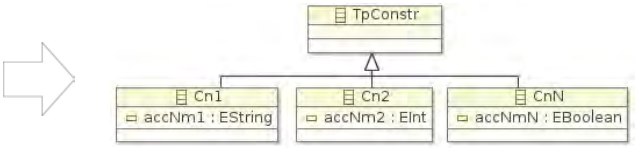
The input consists of a type definition composed of a set of constructor declarations containing type expressions. First, an `EClass` is created to represent the type constructor, its name is the type constructor name. Then, for each constructor declaration the Tr_{decl} function is called. This function produces `EClass`es starting from a constructor declaration. It permits to create an `EClass` that inherits from the previous one. The Inheritance link is set thanks to the Ecore function *setSupertype()*. The name of this second class is the name of a particular constructor.

To transform the types expressions of each constructor, we call the functions for translating the type expressions $Tr_{type}()$, where acc_j is the accessor name corresponding to type expression t_j (the transformation of accessors is detailed inSection 4.14).

$$
\begin{array}{lcl}
Tr_{dtp}(tpConstr = cd_1|...|cd_n) & = & createEClass(); \\
 & & setName(tpConstr); \\
 & & Tr_{decl}(cd_i, tpConstr) \\
 & & / \ 1 \leq i \leq n \\
Tr_{decl} : ConstructorDeclaration & \longrightarrow & EClass \\
Tr_{decl}(cn_i \ t_1...t_m, tpConstr) & = & createEClass(); \\
 & & setName(cn_i); \\
 & & setSuperType \ (EClass(tpConstr)); \\
 & & Tr_{type}(acc_j, t_j) \\
 & & / \ 1 \leq j \leq m
\end{array}
$$

Example:

```
type tpConstr =
Cn1 of string
|Cn2 of int
|...
|CnN of bool
```



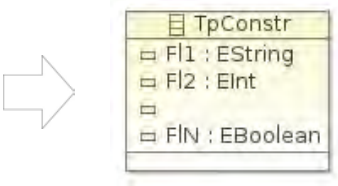
4.8 Rule RecordToEClass

When a type definition is presented as a record, it is translated intuitively to a class. The transformation function *Tr()* starts by creating an EClass. The EClass' name is the name of the type constructor. Then, the fields of the record are translated (using the Tr_{field} function) into structural features.The Structural Feature name is the name of a particular field and the type is translated to its equivalent in Ecore.

$$
\begin{array}{lcl}
Tr(tpConstr = Fl_1; Fl_2; ...; Fl_n) & = & createEClass(); \\
 & & setName(tpConstr); \\
 & & Tr_{field}(Fl_i) / \ 1 \leq i \leq n \\
Tr_{field} : FieldDeclaration & \longrightarrow & EStructuralFeatures \\
Tr_{field}(FlName_i : tp_i) & = & Tr_{type}(FlName_i, tp_i)
\end{array}
$$

Example:

```
type tpConstr =
{ Fl1 :string;
Fl2 : int;
... ;
FlN : bool }
```



4.9 Rule PrimitiveTypeToEAttribute

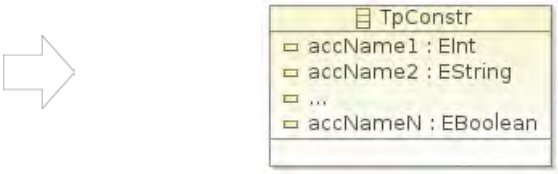
To translate type expressions, the transformation function Tr_{type} is called. If this type expression is formed of a primitive type, the translation function generates a new `EAttribute`. The type of the `EAttribute` is the EMF representation of the corresponding primitive type: `EInt` for *int*, `EBoolean` for *bool*, `EString` for *string*, etc.

On the other side, we notice that the `EAttribute` would be nameless, if we settle for the only information found in the type definition. We use then the corresponding accessor function's name (for accessor functions see Section 3.4.2).

$$
\begin{array}{lcl}
Tr_{type} : (accessor, type) & \longrightarrow & EStructualFeature \\
Tr_{type}(acc, primTp) & = & createEAttribute(); \\
 & & setName(acc); \\
 & & setType(primTp_{EMF});
\end{array}
$$

Example:

```
type tpConstr =
Cn of int * string * ...*
bool
```



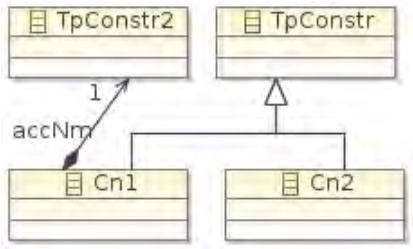
4.10 Rule TypeToEReference

Type expressions might contain user defined types, previously defined in the type definition (consequently previously translated). Then, in most cases an `EReference` is created. The target of the `EReference` (its type) is the `EClass` referenced by the type constructor. The containment feature represented in Ecore by a Boolean value is set to True. The name of the reference is given by the accessor rule. The multiplicity is left to `1`.

$$
\begin{array}{lcl}
Tr_{type} : (accessor, type) & \longrightarrow & EStructualFeature \\
Tr_{type}(acc, tpConstr) & = & createEReference(); \\
 & & setName(acc); \\
 & & setType\ (tp\_constr); \\
 & & setContainment\ (true); \\
 & & setLowerBound(1); \\
 & & setUpperBound(1);
\end{array}
$$

Example:

```
type tpConstr=
Cn1 of tpConstr2
| Cn2
```

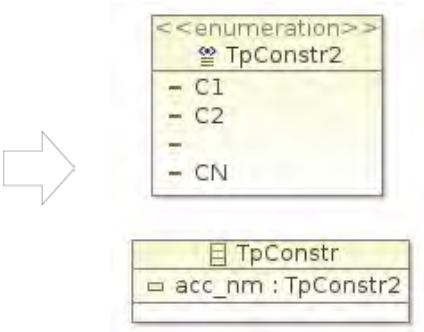


A particular case is detected in this rule. If the user-defined type was translated to an `EEnum`, it is no longer possible to translate the type expression into an `EReference`. Then, it is translated to an `EAttribute` instead of the `EReference`. An example of this particular case is given as following.

Example:

`type` *tpConstr2*`=`
Cn1 |*Cn2*|... | *CnN*

`type` *tpConstr*`=`
Cn `of` *tpConstr2*



4.11 Rule OptionToMultiplicity

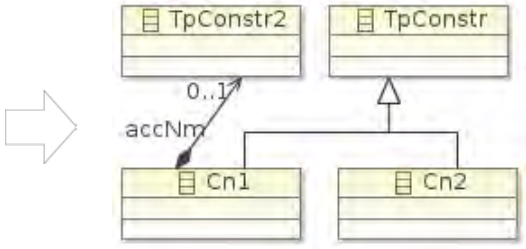
The following set of rules (presented in Sections E.2.3 and 4.13) concerns the translation options contained in a composite type of the form: *type* `ref`, *type* `list` or *type* `option`. In order to translate composite types, we start by performing a first translation of the type part applying on the *type* the function $Tr_{type}()$, then, performing the translation on the optional part (consisting in `ref`, `list` or `option`). This second translation is about changing features in the structural feature created (the multiplicity or the Boolean value of containment in the `EStructuralFeature`)

The type expression *type* `option` is used to express whether a value is present or not. It returns `None`, if it is absent and `Some` value, if it is present. This is modeled by changing the lower and upper bound of the `EStructuralFeature` to respectively `0` and `1`, allowing to create or not the element when instantiating the model.

$$\begin{array}{lcl} Tr_{type} : (accessor, type) & \longrightarrow & EStructualFeature \\ Tr_{type}(acc, t\ \texttt{option}) & = & Tr_{type}(acc, t) \\ & & setLowerBound(0); \\ & & setUpperBound(1); \end{array}$$

Example:

`datatype` *tpConstr*`=`
Cn_1 `of` *tpConstr2* `option`
$|\ Cn_2$



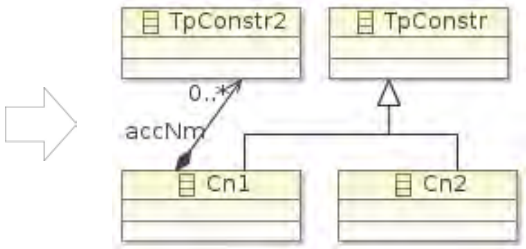
4.12 Rule ListToMultiplicity

To represent collections of values that have the same type we use a type expression of the form `type` *list*. The type expression's transformation consists in translating the type into a structural feature then to change the upper and lower bound respectively to `0` and `*` (`*` arbitrarily many).

$$\begin{array}{lcl} Tr_{type} : (accessor, type) & \longrightarrow & EStructuralFeature \\ Tr_{type}(acc, t\ \texttt{list}) & = & Tr_{type}(acc, t) \\ & & setLowerBound(0); \\ & & setUpperBound(*); \end{array}$$

Example:

`datatype` *tpConstr*`=`
Cn_1 `of` *tpConstr2* `list`
$|\ Cn_2$



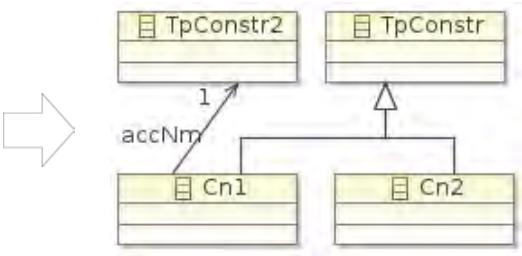
4.13 Rule RefToEReference

The last case that we deal with, is *type* `ref` which is used to represent pointers. In fact, a value of *type* `ref` is pointer to a location in memory, where this location contains a value typed with *type*. We decided to translate it into an `EReference` with containment flag set to `False`. This kind of references is used to model weak associations. Consequently the two classes are related but without a containment relation.

$$Tr_{type}(acc, t\ \texttt{ref}) \quad = \quad Tr_{type}(acc, t); \\ setContainment(False);$$

Example:

`datatype` *tpConstr*=
Cn_1 `of` *tpConstr2* `ref`
| Cn_2



4.14 Transforming Accessors to Structural Features Names

This section is spelled out to define how the *accessor_name* is selected for naming a particular `EStructuralFeature`. The accessor functions are small functions allowing to access a particular field of a type definition. In our source model, we regroup accessors in *accessor_lists*. Each accessor structure is formed of an *accessor_name*, a *constructor − name* (that defines to which constructor the current accessor is belonging) and an integer value named "*index*". This index *(i)* corresponds to the rank of the field that is accessed in a particular constructor declaration.

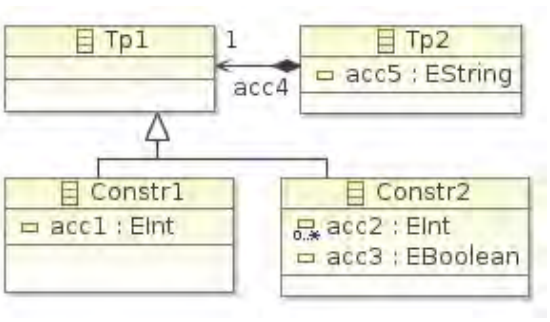
The translation function consists on giving a name to a structural feature. First, the *constructor_name* is used to select the corresponding `EClass` where `EStructuralFeature` has been created. In this `EClass`, we have then to select the concerned structural feature. The index value is compared to the value `FeatureID` given by Ecore to represent the rank of the `EStructuralFeature` creation in a particular `EClass`. When these values are equal, the corresponding accessor name is selected as the name of `EStructuralFeature`. The details of the process are given formally by the following representation.

$$\begin{array}{lcl} Tr_{acc} : Accessor & \longrightarrow & EStructuralFeature \\ Tr_{acc}(acc) & = & Tr_{acc}(acc\_name, constr\_name, i); \\ & & eClass\_list = package.getEClassifier;(); \\ & & selected\_eClass = eClass\_list.search\_by\_name(constr\_name); \\ & & eSF\_list = selected\_eClass.getEAllStructuralFeatures(); \\ & & selected\_eStructuralFeature.set\_Name(acc\_name); \end{array}$$

Here is an example of transforming a data type description together with accessor functions into a class diagram represented in `Ecore`.

Example:

```
type tp1= Constr1 of int
| Constr2 of (int list)* bool

type tp2 = Tp2 of tp1 * string

(*@accessor*) let acc1 (Constr1 (x)) =x ;;

(*@accessor*) let acc2 (Constr2 (x,y)) =x ;;

(*@accessor*) let acc3 (Constr2 (x,y)) =y ;;

(*@accessor*) let acc4 (Tp2 (x,y)) =x ;;

(*@accessor*) let acc5 (Tp2 (x,y)) =y ;;
```



4.15 Transforming Generics

In case the data type definition is a polymorphic data type, composed of a type parameter and a type constructor, it is translated using the representation of generic types in `Ecore` consisting of the handling of `ETypeParameters` together with `EGenericType`.

The transformation consists in the creation of an `EClass` to represent the *Type Constructor* and for each type parameter creating an `ETypeParameter` related to the `EClass` via the `eTypeParameters` reference. Notice that we have to create an `EGenericType` for

each class and type parameter (related to their `EGenericType` via `eTypeArguments`) each time we intend to use the `EClass` as a generic. Then, for each *Constructor Declaration*:

- Create an `EClass` to represent the *Constructor Declaration* which has the same `ETypeParameter`s as the *Type Constructor*.

- Setting its `eGenericSuperType` referring to the generic type representing the *Type Constructor* `EClass`.

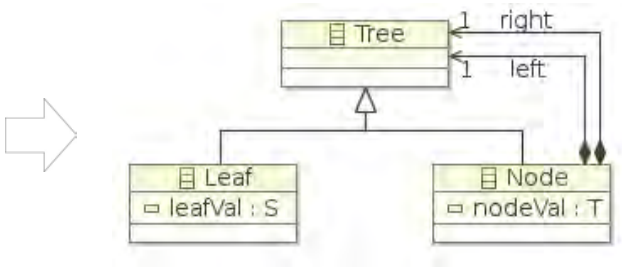
When it comes to use these Generics to type `EStructuralFeature`s, we are faced with two scenarios, the first one is when the type expression is in the form of a `Type Parameter`. The `EStructuralFeature` is then typed with an `EGenericType` referring to the `ETypeParameter` of the containing `EClass`. If instead the *type expression* corresponds to a *Parameterized Type* with *Type Parameters* it is typed with an `EGenericType` representing the `EClass` with `ETypeParameter`s.

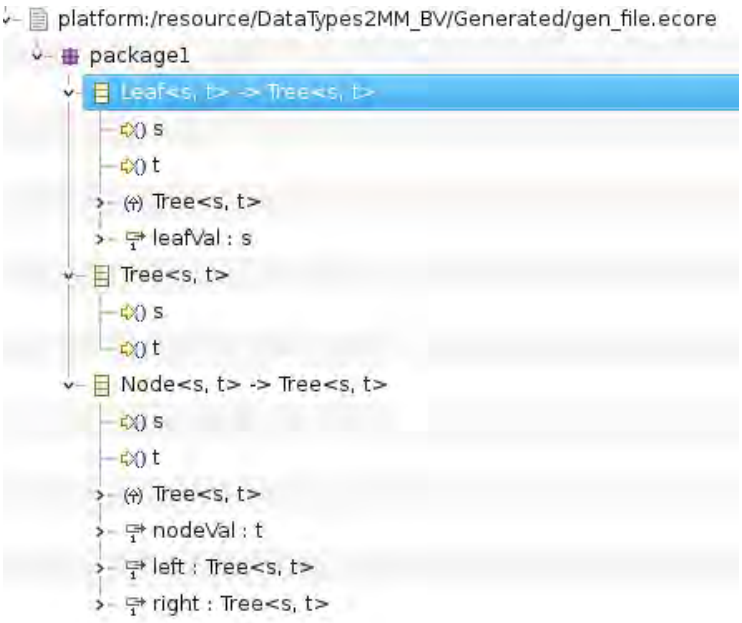
To clarify this process, we use the example below. It consists on the transformation of a simple data type of parameterized Tree. It has two parameters: the first corresponds to the type of leaves and the second to the type of values contained in a Node.

The result after performing the translation is displayed in an Ecore diagram and in the arborescent `Ecore` editor. The `EGenericType`s are not explicitly represented in the `EcoreDiagram`. Although it is correctly present in the `Ecore` file, the graphical interface does not enable to represent all the parameterized types.

Example:

```
type ('s,'t) tree =
Leaf of 's
|Node of 't * ('s,'t) tree *
('s,'t) tree
```





4.16 Summary

This chapter was principally about the detailed presentation of the transformation rules allowing to produce meta-models starting from data type description in functional programming. We took into account all possible cases that may appear in our entry model, and illustrated them with explicit examples. In the next chapter, we will focus on the second side of the transformation.

Chapter 5

EMF to Functional Models

5.1 Introduction

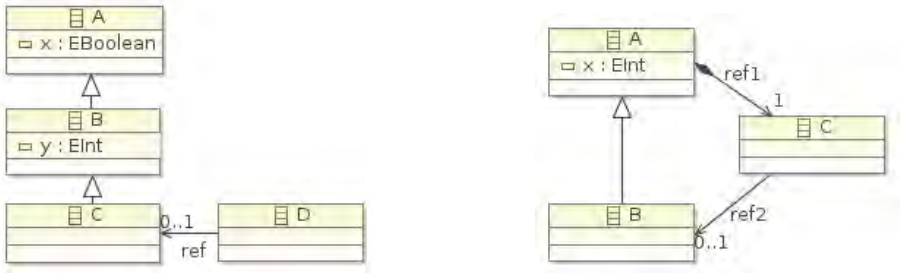
In this chapter, we present the second direction of the translation: from meta-models into data structures used in functional programming and interactive proof. We start by defining some well-formedness conditions on the entry meta-model. Next, we detail one by one the different transformation rules. We finish by discussing the transformation rules and a chapter summary.

5.2 Well-formedness Constraints for Input Meta Models

To perform the reverse direction of the transformation, we draw heavily on the mapping performed on the forward translation (Chapter 4). In our view, it is important to successfully implement a function that is the inverse of the one from datatype to meta-models. Indeed, the possibility of composing the two functions, apply them on a model and find an equivalent model is paramount. Even if it leads us to set some additional restrictions on the meta-model.

The first restriction concerns the depth of inheritance relations: the transformation of a meta-model containing inheritance of classes on more than one level (a class that inherits from a class that inherits from another one etc.) is not supported by our rules. For example, the model presented in Figure 5.1a is not translatable using our transformation functions. Such model would be rejected during the analysis phase because the class C inherits from another class named B which inherits again from the class A.

The second restriction aims at avoiding mutually dependent data types. We therefore define a partial order $\prec$ on classes for the transformation of `EClassifier`s contained in an `EPackage`. The `EEnum`s have to be translated first, because they don't depend on any other elements. The `EClass`es left in the `EPackage` have then to be ordered using two criteria:



(a) Non-compliance with the 1^{st} restriction (b) Non-compliance with the 2^{nd} restriction

Figure 5.1: Examples of Untranslatable Models

- **The Inheritance relation:** if an `EClass` C_1 is a `superType` (used in Ecore for determining a super class) of another `EClass` C_2, then C_1 has to be translated before C_2. We therefore add the constraint $C_1 \prec C_2$.
- **The reference relation:** if an `EClass` C_1 is a target (`eType` in Ecore) of an `EReference` belonging to another `EClass` C_2, then C_1 has to be translated before C_2, thus $C_1 \prec C_2$.

This order allows us to define the second well-formedness restriction: The order $\prec$ generated by the above two constraints has to be acyclic.

The example shown in Figure 5.1b illustrates a case where this restriction is not respected. Indeed, $A \prec B$ because A is the super class of B. Also, $B \prec C$ because C has a reference named $ref2$ whose type is B. Thus, $A \prec C$. But, $C \prec A$ due to the reference $ref1$ from A to C. Clearly, we find a cycle. This model is then untranslatable using our transformation function.

The last constraint we impose on the models is about inheritance and genericity. Indeed, if we have an inheritance relation between two generics (`EClass`es with `ETypeParameter`s), all the parameters used by the child class have to appear in the super class.

5.3 Representation of Transformation Rules

As in Chapter 4 , transformation rules are presented in natural language with a formal notation followed by an illustrative example. To avoid overloading the notation, we use again the notation $Tr()$ to represent the translation function (instead of writing $Tr()^{-1}$). The transformation method $Tr()$ then produces data type descriptions (gathered in modules) starting from a Meta Model represented by a set of `EPackage`s.

$$Tr : Ecore\ Meta\text{-}model \longrightarrow DataTypes$$

The entry point of the transformation is the rule *EPackageToModule*, which triggers recursively the other transformation rules.

5.4 Rule EPackageToModule

The elements composing Ecore models are gathered into `EPackage`s. When we perform the translation from Ecore models to functional data type descriptions, we transform these packages into modules. The name of a particular `EPackage` gives the name of the *module*. The additional elements `nsPrefix` and `nsURI` are specific features of Ecore. They are not translated and not used in the functional description. We call the rule Tr_{Cl} to translate the `EClassifier`s contained in the `EPackage`.

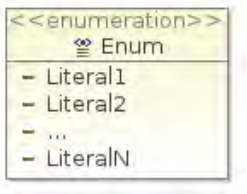
$$\begin{aligned} Tr\ (&\texttt{ePackage name} = p \\ &\texttt{nsPrefix} = pp \\ &\texttt{nsURI} = puri \\ &\{ECl_1\ ...\ ECl_n\}) = \quad createModule(); \\ &\qquad\qquad\qquad\qquad\quad setName(p); \\ &\qquad\qquad\qquad\qquad\quad Tr_{Cl}(ECl_i); \qquad / \ 0 \leq i \leq n \end{aligned}$$

5.5 Rule EEnumToDatatype

Enumerated types are represented in Ecore by `EEnum`s. To translate an `EEnum`, we first get all the `EClassifier`s contained in the `EPackage`, check if they are instances of `EEnum`s. In this case, they are transformed into data type definitions composed of *constructor declaration*s without type expressions. Each `EEnumLiteral` of the enumeration gives a *constructor* name.

$$\begin{aligned} Tr_{Cl}(&\texttt{eEnum name} = e \\ &\{ELit_1\ ...\ ELit_n\}) = \quad createDatatype(); \\ &\qquad\qquad\qquad\qquad\quad NewTp\_Constr(); \\ &\qquad\qquad\qquad\qquad\quad setName(e); \\ &\qquad\qquad\qquad\qquad\quad Tr_{Lit}(ELit_i); \qquad / \ 1 \leq i \leq n \\ Tr_{Lit}(&\texttt{literal} = l) = \quad createConstructor(); \\ &\qquad\qquad\qquad\qquad\quad setName(l); \end{aligned}$$

Example: The following example illustrates the transformation of an `EEnum` into a data type.



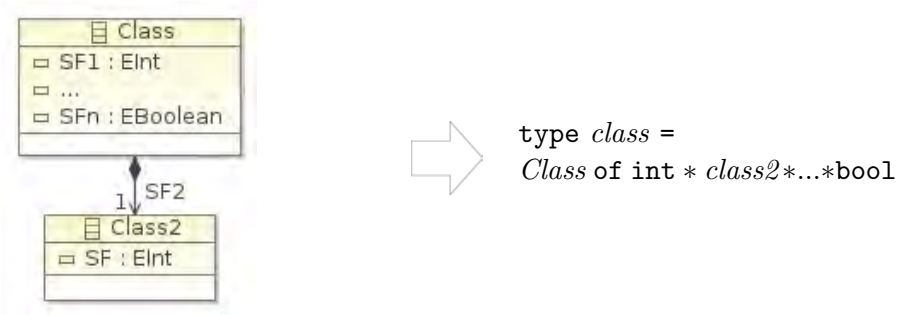
`type` *enum*=
Literal1
|*Literal2*
|...
|*LiteralN*

5.6 Rule EClassToDatatype

The simplest case that we deal with is the one consisting in transforming a simple `EClass` which is not related with other `EClass`es by any inheritance link. In such a case, the `EClass` is translated to a single *type definition* with a unique *constructor declaration*. The `EClass` name gives the *type constructor* and the *constructor* names (they take the same name). Then, for each `EStructuralFeature` contained in the `EClass`, we call the appropriate sub-function: Tr_{SF} that stands for Translate Structural Feature.

$$
\begin{aligned}
Tr_{Cl}(\texttt{eClass name} = c \\
\{ESf_1 \ldots ESf_n\}) = \quad & if(not(c.is\_superType())) \\
& createDatatype(); \\
& setName(c); \\
& NewTp\_Constr(); \\
& setName(c); \\
& Tr_{SF}(ESf_i); \qquad / \ 0 \leq i \leq n
\end{aligned}
$$

Example: This example shows how the `EClass` *Class* is tranformed into a type definition.



5.7 Rule EClassInheritanceToDatatype

This rule transforms an `EClass` hierarchy into a *type definition*. When we are faced with an `EClass` transformation, we first check if it is a `SuperType` of other classes. In such a case, we create a new *type definition* named with the `EClass` name. Then, we select all the classes that inherit from this super class. For each of them, we apply the rule *EClassToConstructor*.

If the super class is a generic type (an `EClass` augmented with `ETypeParameters`) we call the function $Tr_{prm}()$ for every `ETypeParameter`.

$$
\begin{aligned}
Tr_{Cl}(&\texttt{eClass name} = sClass \\
&\{ESf_1 \ldots ESf_n\} \\
&\{ETp_1 \ldots ETp_m\}) = \quad if(c.is\_superType()) \\
&\qquad createDatatype(); \\
&\qquad setName(sClass); \\
&\qquad class\_list = select\_child\_classes \\
&\qquad Tr_{ch\_cl}(class_i) \qquad / \; class_i \in class\_list \\
&\qquad Tr_{prm}(ETp_i) \qquad / \; 0 \leq i \leq m
\end{aligned}
$$

5.7.1 Rule EClassToConstructor

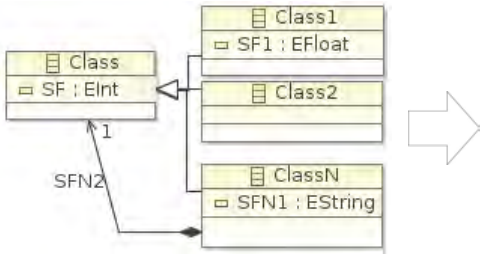
Thanks to this rule, each (child) `EClass` is translated to a constructor declaration in the corresponding type definition. First, a new *constructor* is created, the name of the *constructor* is the `EClass` name. Then for each `EStructuralFeature` contained in the `EClass` the function Tr_{SF} is called. The rules *EAttributeToType* and *EReferenceToType* are applied depending on the nature of the `EStructuralFeature`.

$$
\begin{aligned}
Tr_{ch\_cl}(&\texttt{eClass name} = c \\
&\texttt{eSuperType} = sClass \\
&\{ESf_1 \ldots ESf_n\}) \quad = \quad setName(); \\
&\qquad\qquad\qquad\qquad createConstructor(c); \\
&\qquad\qquad\qquad\qquad Tr_{SF}(ESf_i); \qquad / \; 1 \leq i \leq n
\end{aligned}
$$

The rule is applied in the same way when the super class is generic (in this case we have `eGenericSuperType` instead of `eGenericType`).

$$
\begin{aligned}
Tr_{ch\_cl}(&\texttt{eClass name} = c \\
&\texttt{eGenericSuperType} = sClass \\
&\{ESf_1 \ldots ESf_n\})
\end{aligned}
$$

Example: Here is an example of translating a hierarchy of `EClasses` into a *type definition.*



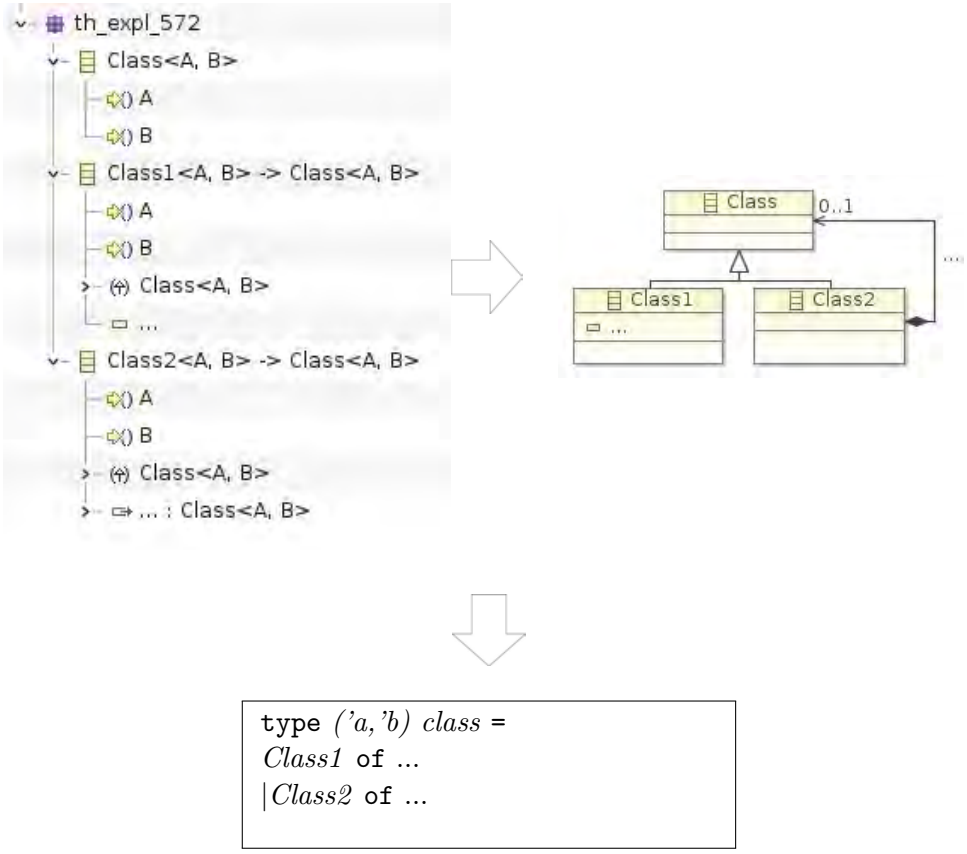

```
type class =
Class1 of int* float
|Class2 of int
|...
|ClassN of int* string *
class
```

5.7.2 Rule ETypeParameterToTypeParameter

Clearly, the `ETypeParameters` used in the representation of generics in Ecore are translated to their equivalents in functional programming: *type parameters*.

$$Tr_{prm}(\texttt{eTypeParameter name} = tp) = \\ createTypeParameter(); \\ setName(tp);$$

Example: This example shows how `ETypeParameters` of a generic type representation are translated into type parameters in a functional language. On the Ecore diagram generics do not appear explicitly, thus we added an image of our model in the arborescent Ecore editor.




```
type ('a,'b) class =
Class1 of ...
|Class2 of ...
```

5.8 Rule EAttributeToType

Transforming every `EAttribute` consists in creating a new *type expression* in the corresponding *constructor declaration* (and *type definition*). The corresponding *type definition* can be selected by name in the list of created data types. The created *type expression* is composed of the translation of `EAttribute`'s type and the bounds. The `eType` translation consists in giving an equivalent type in the functional language. This process is done by using the transformation function Tr_{Type}. To translate the upper and lower bounds, the function Tr_{Bnd} is called.

$$
\begin{aligned}
&Tr_{SF}(\texttt{eAttribute name} = a \\
&\qquad\qquad LowerBound \\
&\qquad\qquad UpperBound \\
&\qquad\qquad EType) = \\
&\quad createTypeExpression(Tr_{Type}(EType)); \\
&\quad Tr_{Bnd}(LowerBound, UpperBound);
\end{aligned}
$$

$$
\begin{aligned}
Tr_{Type}(\texttt{eType} = \texttt{EInt}) &= \texttt{int} \\
Tr_{Type}(\texttt{eType} = \texttt{EBoolean}) &= \texttt{bool} \\
Tr_{Type}(\texttt{eType} = \texttt{EFloat}) &= \texttt{float} \\
Tr_{Type}(\texttt{eType} = \texttt{EString}) &= \texttt{string} \\
Tr_{Type}(\texttt{eType} = \texttt{eenum e}) &= e
\end{aligned}
$$

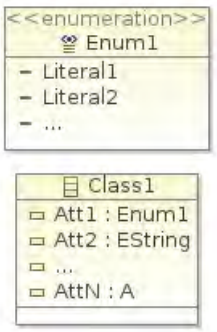
5.8.1 Rule EAttributeToTypeParameter

If the `EAttribute` is typed with an `ETypeParameter` (belonging to a generic type), it is translated (as the previous case) to a *type expression* consisting in a *type parameter*. The name of the `ETypeParameter` becomes the name of the *type parameter* in the *type expression*.

$$
\begin{aligned}
&Tr_{SF}(\texttt{eAttribute name} = a \\
&\qquad\qquad LowerBound \\
&\qquad\qquad UpperBound \\
&\qquad\qquad \texttt{eGenericType}\{ETp_1 \ ... \ ETp_n\}) = \\
&\quad createTypeExpression(Tr_{TpPrm}(ETp_i)); \qquad / \ 1 \leq i \leq n \\
&\quad Tr_{Bnd}(LowerBound, UpperBound);
\end{aligned}
$$

$$
\begin{aligned}
&Tr_{TpPrm}(\texttt{eTypeParameter name} = prm) = \\
&\quad createTypeExpression(prm)
\end{aligned}
$$

Example: Here is an example of how the `EAttribute`s are translated into *type expressions*. Three cases are taken into account: attributes typed with an `EEnum`, a predefined type then an `ETypeParameter`.



`type` *enum*=
Literal1|*Literal2* |...

`type` *'a class1* =
Class1 `of` *enum1* ∗ `string` ∗
...∗ *'a*

5.9 Rule EReferenceToType

To translate an `EReference` (targeting to an `EClass` "c"), we first create, as in the previous rule, a new *type expression* in the corresponding *constructor declaration* (and *type definition*). This *type expression* is then represented by the name of the `EClass` to which is targeting the `EReference` (the `eType` value of the `EReference`). The name of this `EClass` corresponds to a previously translated EClass. The name of this EClass could be translated either to a type constructor name (using one of the rules presented in Sections 5.6 and 5.7) or to a constructor name (using the rule of the Section 5.7.1). For each of these two cases, there is a particular processing :

- if the `EClass` name was previously translated to a type constructor name (*typeconstr-name*): the type expression is then represented by the name of the EClass "c".
- if the `EClass` name was previously translated to a constructor name (*constr-name*): the type expression is then represented by the name of the SuperClass of the EClass "c".

It is guaranteed that the data type is previously translated thanks to our translation order given in Section 5.2.

To translate the multiplicities and the containment values we call respectively the functions Tr_{Bnd} and Tr_{Cont}.

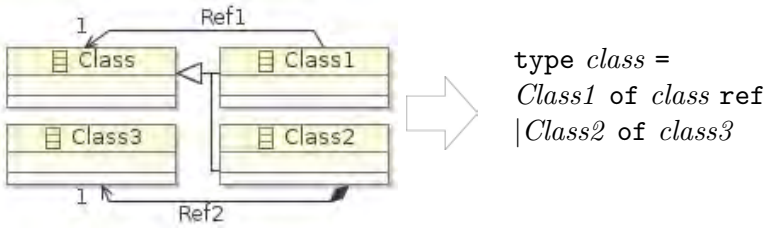
$$\begin{aligned} Tr_{SF}(&\texttt{eReference}\ Containment \\ &\texttt{name} = r \\ &LowerBound \\ &UpperBound \\ &\texttt{eType} = c) = \\ &createTypeExpression(get\_tpConstr\ (c)) \\ &Tr_{Cont}(Containment); \\ &Tr_{Bnd}(LowerBound, UpperBound); \end{aligned}$$

5.9.1 Rule ContainmentToRef

The function Tr_{Cont} is used to translate the containment Boolean value (to represent whole/part relationship) in the functional world. In case this value is set to false (the containment relationship doesn't hold), it is then translated by adding the `ref` option to the translated type expression (representation of pointers). On the contrary if the containment value is set to true, the translation into type expression is left unchanged (no pointers).

$$Tr_{Cont}(\texttt{containment=false}) = ref$$
$$Tr_{Cont}(\texttt{containment=true}) = \emptyset$$

Example: This is an example of the translation of `EReference`s into *type expressions*.



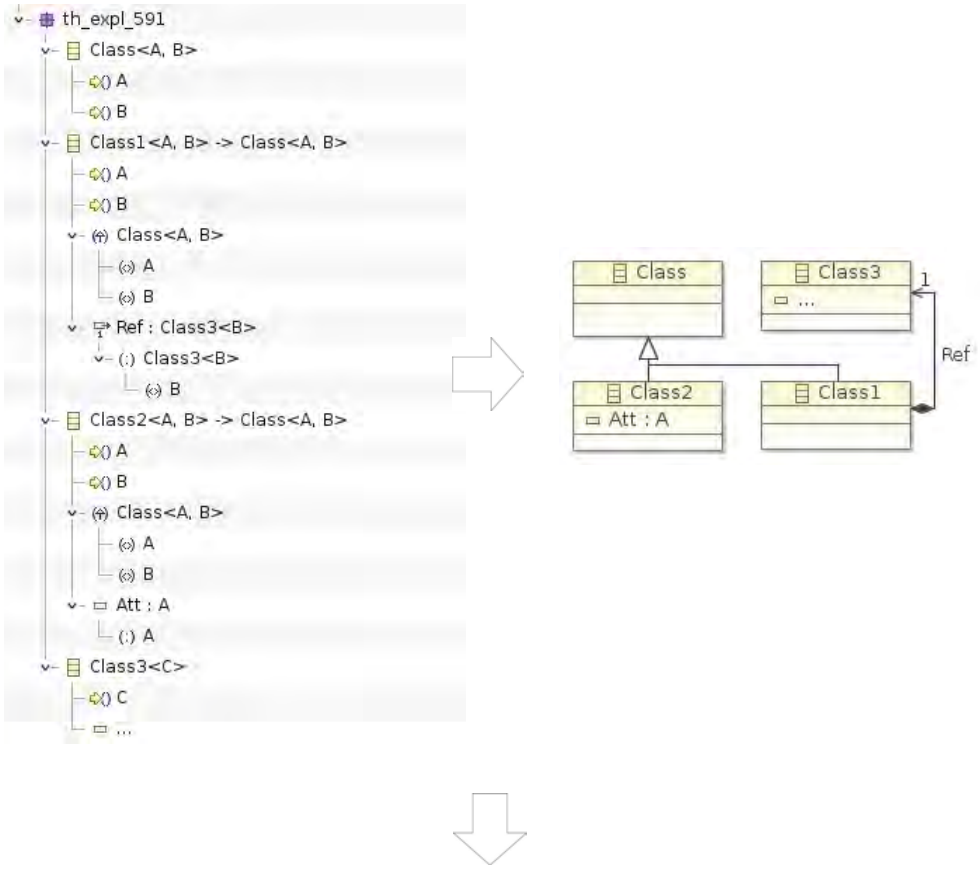
5.9.2 Rule EReferenceToParametrizedType

When the `EReference` target is a generic type (in the shape of an `EClass` augmented with type parameters), a new *type expression* in the corresponding *constructor declaration* (and *type definition*) is created to represent the `EClass`. Next, to each `ETypeParameter` related to the `EClass` a *type parameter* is created in the *type expression*.

$$\begin{aligned} Tr_{SF}(&\texttt{eReference name} = a \\ &\quad LowerBound \\ &\quad UpperBound \\ &\quad \texttt{eGenericType name} = genTp \\ &\qquad \{ETp_1 \ ... \ ETp_n\}) = \\ &createTypeExpression(genTp) \\ &createTypeParameters(prm_i) \ / \ 1 \leq i \leq n \end{aligned}$$

Where ETp_i has the form :
$ETp_i = \ (\texttt{eTypeParameter name} = \ prm_i), 1 \leq i \leq n$

Example: This example illustrates the transformation of the references to generic types, once again, so that the parameters appear more clearly, we show the representation of the model in Ecore, as well as Ecore diagram.




```
type 'c class3 =Class3 ...

type ('a,'b) class =
Class1 of 'b class3
|Class2 of 'a
```

5.10 Rule MultiplicitiesToTypeOptions

This rule permits to translate multiplicity values contained in structural feature definitions (represented by an upper and a lower bound). They are used to determine the number of features that compose an instance. When the `lowerBound` is set to `0` and the `upperBound` to `1`, this signifies that this `EStructuralFeature` might be present or absent (when instantiating). These values are translated to type `option` in the *type expressions*.

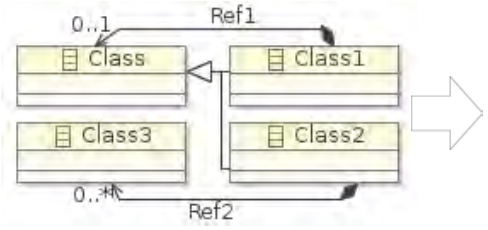
In case the `upperBound` is represented by a `*` this implies the ability of creating more than one instance of the concerned `EStructuralFeature`. It is translated to the type *list*

in the data type description. Notice that we don't make a distinction when the `lowerBound` is set to `0` or `1`. This is due to the impossibility to define the type of a non empty list in Caml.

If both upper and lower bounds are set to `1`, exactly one instance of the `EStructuralFeature` is expected. In this case no customization is performed. The type expression is left unchanged.

$$Tr_{Bnd}(\texttt{lowerBound="0"}, \texttt{upperBound="1"}) = \texttt{option}$$
$$Tr_{Bnd}(\texttt{lowerBound="0"}, \texttt{upperBound="*"}) = \texttt{list}$$
$$Tr_{Bnd}(\texttt{lowerBound="1"}, \texttt{upperBound="*"}) = \texttt{list}$$
$$Tr_{Bnd}(\texttt{lowerBound="1"}, \texttt{upperBound="1"}) = \emptyset$$

Example: This last example explains how to translate the multiplicities in type definitions.



`type` *class* `=`
Class1 `of` *class* `option`
|*Class2* `of` *class3* `list`

5.11 Summary

We defined in this chapter how we perform the translation of EMF models into data types used in functional languages. We first presented some restrictions on the shape of meta models provided as input, in order to avoid inconsistencies on the translated data types. Also, these constraints allow to ensure that the composition of the transformation function and its inverse gives the identity as we start the formal model. Next, we gave detailed rules allowing to transform elements of the meta models into those composing data structures. These rules were illustrated by simple examples. Table 5.1 summarizes the most important features taken into account in our transformation process and their translation in the functional world.

The following part concerns the illustration of our approach by applying it to different case studies, starting by Binary Decision Diagrams in Chapter 6.

`Ecore` Components	Functional Data Structures
`EClass`	*Type Constructor* + *Constructor*
`EAttribute`	*Type Expression* (*Primitive Type*)
`EReference`	*Type Expression*
`EEnum`	*Type Constructor* + *Constructor*s (without *Type Expression*s)
`EEnumLiteral`	*Constructor* (without *Type Expression*)
`Inheritance`	*Type Constructor* + *Constructor*s + *Type Expression*s
`EGenericType`	*Parameterized Datatype*
`ETypeParameter`	*Type Parameter*

Table 5.1: Table Summarizing the Transformation Rules from Ecore Meta-models to Data Types

Part III

Case Studies

Chapter 6

Binary Decision Diagrams

6.1 Introduction

This chapter presents a simple case study illustrating the transformation method defined in Chapter 4. The case study consists in applying our transformation method on a verified Isabelle theory allowing to construct Binary Decision Diagrams. We couple the extracted code also with the use of tools provided by Eclipse Modeling Project [51].

The chapter is structured as follows: we start by presenting Binary Decision Diagrams, then we explain the way our transformation method is applied to this particular example. We continue by showing the creation of graphical and textual editors.

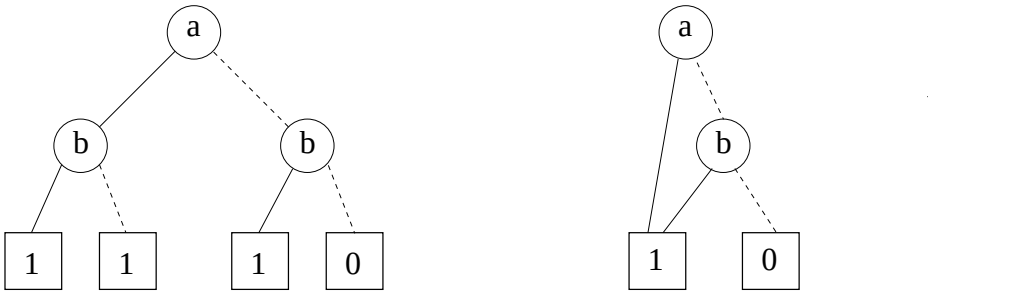
6.2 Binary Decision Diagrams

Binary Decision Diagrams (BDDs) [26, 4] are particular oriented graphs representing Boolean expressions in a concise and canonical form. BDDs can be understood as decision trees with maximal sharing.

They are represented as binary trees composed of two sorts of nodes: *decision nodes* and *terminal nodes*. Each decision node is labeled by a Boolean variable and has two child nodes. The link from a decision node to its left (respectively right) child represents an assignment of the variable to True (respectively False). The terminal nodes are the Boolean constants True and False.

The compact representation of reduced BDDs consists in merging isomorphic subgraphs and joining common subtrees in order to eliminate redundancies. The representation of Boolean formulas in this compact way makes validity checking very efficient. BDDs are frequently used in model checking and digital circuit development. They can also be applied to the solution of large systems of linear equations [88].

Example: In Figure 6.1, we present two BDDs built starting from the Boolean formula "a∨b". The sub-figure 6.1b shows a reduced BDD by merging common subtrees whereas in the 6.1a one, no reduction is performed.



(a) Decision Tree Representing "a∨b" (b) Compact BDD Representing "a∨b"

Figure 6.1: BDDs Representing the Formula "a∨b"

6.3 Verified BDD Construction

In [49] *Giorgino et al* carried out, in the Isabelle proof assistant [73], some proofs about the correctness of an algorithm constructing BDDs. The algorithm is mainly about converting a Boolean expression to a canonical representation of the expression : a BDD. Moreover, it consists of two main functions *build()* and *app()*. The first one allows to build recursively BDDs of subexpressions. The second simply combines the built subtrees.

Starting from the proven Isabelle theories, it is possible to generate the corresponding object oriented code in Scala. Since Scala [78] compiles to Java Virtual Machine, it is possible to combine it with Java code extracted from our framework. We used also these Isabelle theories as entry point to our case study. The approach is further spelled out in the next section.

6.4 Presentation of the Case Study

In this case study, we aim at bringing readability to verified components for those who are not used to handle interactive proofs and formal languages.

The global process we perform is described in Figure 6.2. In this diagram, non-dashed arrows represent automatic code generation whereas dashed ones state for hand-written code. The entry point of the diagram is an Isabelle *theory* defining data types and functions. Here also are set the properties to be checked. Starting from these Isabelle theories (containing data types and accessor functions), we generate using our translation function the corresponding meta model. Also, thanks to generation facilities of Isabelle we get the corresponding Scala code for data types and functions.

Starting from the generated meta model in Ecore, we use the GMF or the Xtext tools allowing to easily create a graphical or a textual tool for a domain specific language. Once the tool is created (and its code generated in Java), we write some adapting code making the link between the generated code in Scala and the one in Java (belonging to the graphical or the textual tool for the DSL). Consequently, it is possible to access the dynamical part (represented by generated Scala functions) with the tools provided by EMF.

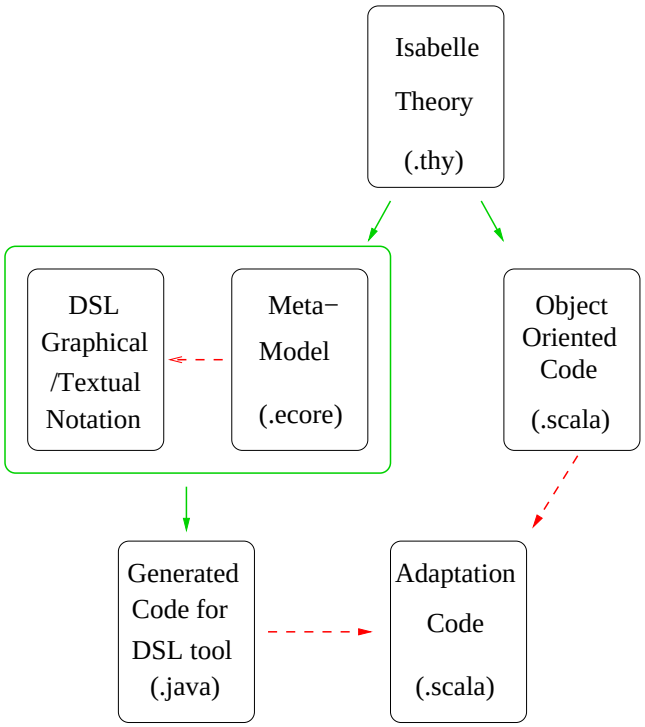


Figure 6.2: Adopted Approach in the Implementation of the BDD Case Study

The process described above is applied on the Isabelle *theories* written for both Boolean expressions and Binary Decision Diagrams. The type definitions used to describe them will be transformed using our transformation function. They will be also translated together with the algorithm building BDDs to an object oriented program in Scala. Then it will be easy to handle Boolean values in a textual editor, build the corresponding BDD using the *build()* and *app()* functions, then display the result in a graphical editor.

Figure G.1 explains the procedure used for the case study. The code carried out consists in converting the data recovered from Xtext to Expressions defined in Scala to make them usable by the *Build()* function. Then building the BDD, and finally converting the Scala BDD to EMF models and serialize the result to GMF models.

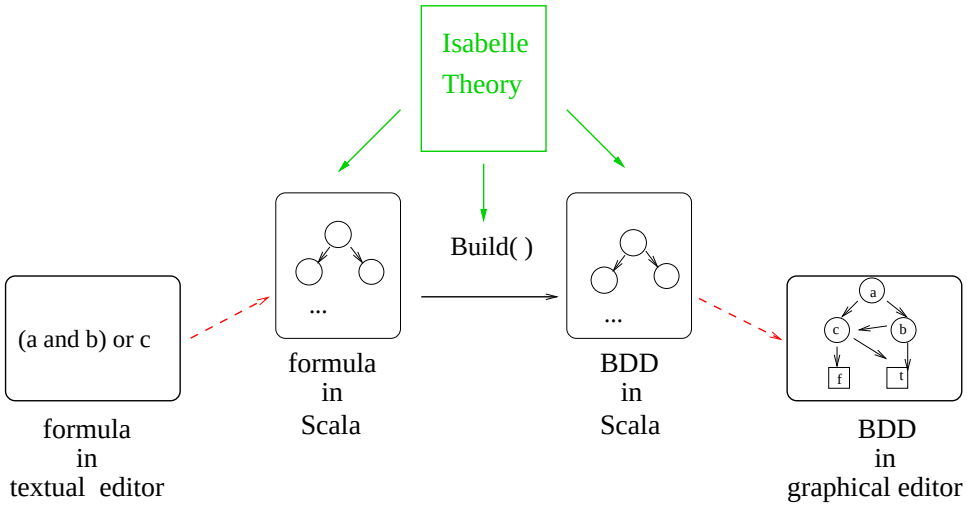


Figure 6.3: Architecture of the Case Study

6.5 Generating Ecore Diagrams from Data Types

6.5.1 Applying the Transformation on the Formula Type Definition

Figure G.2 shows the Isabelle data type used in the formal development. For our purposes, we defined accessor functions that are necessary to perform adequately our transformation. In this type definition, each Boolean expression *bexpr* can appear in the form of a variable *BVar* represented by a string value, or in the form of a constant *BConst*; while in its compound form, an expression can be shaped as a binary operation *bbinop* along with two Boolean expressions. The data type *bbinop* is preceded by the term *(**Enumerated*)* to specify how it has to be translated. Binary operations are *OR*, *AND*, the implication *IMP* and equivalence *IFF*.

The following accessor functions are used to name a specific field of the different type expressions, *var* for the variable, *const* for the constant, *op* for the operation and also *rexpr* (respectively *lexpr*) for the right (respectively left) part of a binary expression.

This part of the Isabelle theory was given as input to our translation function presented in Chapter 4. The Ecore diagram resulting from the transformation is shown in Figure G.3.

As it appears in the diagram, the data type *bbinop*, commented as enumerated has been transformed to an `EEnum` in Ecore. The *bexpr* data type has been converted to a hierarchy of `EClass`: A super class `Bexpr`, with three child classes `BVar`, `BConst` and `BBExpr`. The type expression in the *BVar Constructor* yields a new `EAttribute var`, named as specified in the corresponding accessor function. The same is performed to obtain the `const` and `op`

$$
\begin{array}{lrl}
(**Enumerated*) & datatype\ bbinop & = OR \mid AND \mid IMP \mid IFF \\
 & datatype\ bexpr & = BVar\ string \\
 & & \mid BConst\ bool \\
 & & \mid BBExpr\ bbinop\ bexpr\ bexpr \\
(*@accessor*) & fun\ var & :: bexpr\ =>\ string \\
 & where & var(BVar\ x\ ) = x \\
(*@accessor*) & fun\ var & :: bexpr\ =>\ bool \\
 & where & const(BConst\ x\ ) = x \\
(*@accessor*) & fun\ op & :: bexpr\ =>\ bbinop \\
 & where & op(BBExpr\ x\ \_\ \_) = x \\
(*@accessor*) & fun\ lexpr & :: bexpr\ =>\ bexpr \\
 & where & lexpr(BBExpr\ \_\ x\ \_) = x \\
(*@accessor*) & fun\ rexpr & :: bexpr\ =>\ bexpr \\
 & where & rexpr(BBExpr\ \_\ \_\ x) = x
\end{array}
$$

Figure 6.4: Data Type for Boolean Formulas in Isabelle and its Accessors Functions

`EAttributes`. The left and right `EReferences` (from `BBExpr` to `Bexpr`) represent the type expressions `bexpr` in the *BBExpr* `Constructor`.

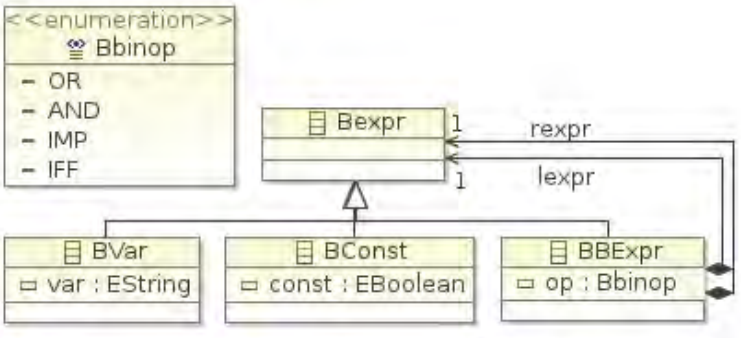


Figure 6.5: Formula-Generated Meta Model

6.5.2 Applying the Transformation on the BDD Type Definition

The type definition presented in Figure G.4 is given to define BDDs in Isabelle. It states that each *BDD tree* can be represented by either a *Node*, formed of two subtrees together with a string value, or a *Leaf* containing a Boolean value. The function *nodeVal* is used to access the string value contained in the node. The same way the functions *left* and *right* access the subtrees. The last accessor function *leafVal* returns the Boolean value of a *Leaf*.

```
                                datatype bddTree =  Node string bddTree bddTree
                                                   |Leaf bool
(*@accessor*)                        fun nodeVal  :: bddTree => string
                                      where  nodeVal(Node x _ _) = x
(*@accessor*)                        fun left     :: bddTree => bddTree
                                      where  left(Node _ x _) = x
(*@accessor*)                        fun right    :: bddTree => bddTree
                                      where  right(Node _ _ x) = x
(*@accessor*)                        fun leafVal  :: bddTree => bool
                                      where  leafVal(Leaf x) = x
```

Figure 6.6: BDD Type Definition and its Accessors Functions in Isabelle

As previously, Figure G.5 shows the generated meta-model of BDDs starting from the Isabelle description of data types presented in Figure G.4. In the resulting diagram, the type definition *bddTree* is translated to a `Superclass` with two child classes: `Node` and `Leaf`. The first one contains an `EAttribute` named `nodeVal` which is typed as an `EString`, and two references `left` and `right` to the class `BddTree` (representing subtrees). The `Leaf EClass` has only one `EAttribute leafVal` typed with a `Boolean`.

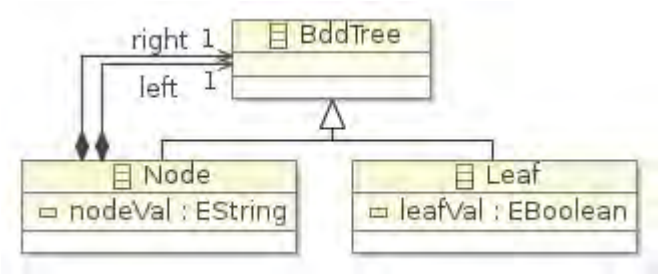


Figure 6.7: Translated Meta-model for BDDs

6.6 Using Xtext Facilities to define a DSL Textual Editor: Application to the Boolean Formula Example

Xtext (see Section 2.6.2) is a component of TMF (Textual Modeling Framework) in the Eclipse Modeling Project. Briefly, it is a tool allowing to develop text editors for DSLs starting from an Ecore meta-model (or an EBNF grammar). The editor is automatically generated as a Eclipse plug-in containing a corresponding parser based on ANTLR.

The generated Xtext project (from a meta-model) contains among others a file which defines the grammar of the DSL. It is possible to make some modifications on generated grammar to adapt it to the needs of the user. In fact, it is possible to choose the most appropriate keywords and how an element of the DSL is shaped in the editor.

From the generated Ecore meta-model for Boolean expressions (see Figure G.3), it is relatively easy to automatically generate an Xtext project (it only takes a few clicks). In the generated grammar we run some changes. We add opening and closing parentheses in order to get a better view of the Boolean expression. We also define the way the expression is shaped: the right expression `rexpr` then an operation `op` before the left expression `lexpr`. For the binary operation `Bbinop`, we choose the `->` and `<->` symbols for respectively the `IMP` and `IFF` operations. Figure 6.8 shows the Xtext grammar used to define a textual editor for Boolean formulas after running the changes.

```
Bexpr returns Bexpr:
    Bexpr_Impl | BVar | BConst | BBExpr;

BVar returns BVar:
    {BVar}
        ( var=EString)? ;

BConst returns BConst:
    {BConst}
    (const?='true')? ;

BBExpr returns BBExpr:
    '('  rexpr=Bexpr
         op=Bbinop
         lexpr=Bexpr
    ')';

EString returns ecore::EString:
    STRING | ID;

EBoolean returns ecore::EBoolean:
    'true' | 'false';

enum Bbinop returns Bbinop:
    OR = 'OR' | AND = 'AND' | IMP = '->' | IFF = '<->';
```

Figure 6.8: Xtext Grammar for Boolean Formulas

6.7 Using GMF Facilities to define a DSL Graphical Editor: Application to the BDD Example

Graphical Modeling Framework (GMF) [51] is a framework for building graphical Eclipse based editors starting from an Ecore meta-model (more details in Section 2.6.1).

Each generated code using GMF is composed of three parts: a *model* part, an *edit* part and a *diagram* part. The *model* and *edit* parts are carried out with the code generation facilities given by EMF, while the *diagram* part is generated thanks to GMF one's.

When using GMF, we have to define different models which are specific to GMF. The first one is the *graphical model*, it is used to define the graphical elements that will appear in the generated editor (node shapes, connections, labels...). The second model is the *tooling model*. The latter contains the tools allowing to create the elements in the generated editor (components of the palette). The correspondence between the graphical model, the tooling model and the base Ecore model is ensured by the *mapping model*.

In our BDD example, starting from our generated Ecore model for BDDs (presented in Figure G.5), we follow the previous steps in order to generate an editor for BDD diagrams. We first create a graphical model where we define :

- Two sorts of nodes: `Node` and `Leaf`
- Each `Node` has a String label
- Each `Leaf` has a Boolean label
- Two types of connections : `Left` and `Right`

In the tooling model we introduce :

- A tool for creating a new `Node` node, `Leaf` node, `Left` connection and `Right` connection.

In the mapping model, we make the correspondence between elements of the two previous models with the Ecore meta-model for BDDs. for example :

The Leaf `EClass` in the Ecore model $\leftrightarrow$ The `Leaf` node in the graphical model $\leftrightarrow$ The `Leaf` creation tool in the tooling model.

Once the different models instantiated, and additionally to the *model* and the *edit* part generated using EMF facilities, we can generate the *diagram code*. The editor is then created and usable as an Eclipse plug-in.

6.8 A Complete Execution of the Case Study

Now we have on one side a Scala program generated from Isabelle theories allowing to build BDDs from logical formulas, and on the other side, two editors enabling the representation of logical formulas and BDDs. It is thus to make the correspondence between the code generated from Isabelle and the one generated using Xtext and GMF in order to display the input and output data in editors. More precisely, this code consists in converting the recovered data from the formulas editor to the Scala representation of these formulas, in order to make them usable by the *Build()* function. Then, proceed to construction of BDD, and finally convert the BDD data in Scala to display it as a diagram in the generated editor.

An example of our case study is presented, Figure 6.9 shows the logical formula $((A \wedge B) \vee C)$ displayed in the generated Xtext editor. After executing the *Build()* function, the corresponding BDD is displayed in the generated GMF graphical editor as shows Figure 6.10.

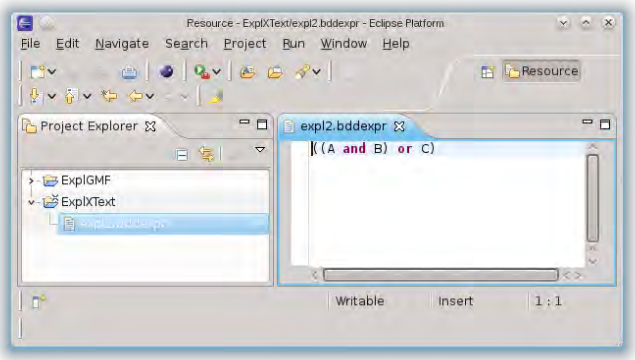


Figure 6.9: Logical Formula Displayed in a Generated Textual Editor

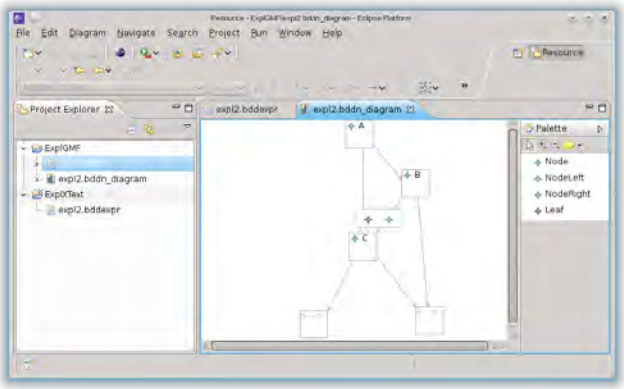


Figure 6.10: Resulting BDD Tree Displayed in a Generated Graphical Editor

6.9 Summary

In sum, this chapter is presented to illustrate by a case study our transformation technique allowing to generate meta models from a data type description in Isabelle. We started by a description of our case study consisting in using a verified program for building a BDD for the compact representation of Boolean formulas. We then applied our transformation method and used the generated meta models to create a textual and a graphical editor. Consequently, we showed a possible way to use the generated meta models in Ecore.

We worked on the automatization of the whole approach adopted in this case study, by automating the generation of the adaptation code carried out manually in the presented case study. Unfortunately this part was not successful. This is due to the incompatibility of the code generated by Isabelle in Scala and the one generated by EMF in Java.

The mechanism used to generate code using EMF (generation of model code) is correctly defined. For each EClass of the model, a Java class and a Java interface are generated. For the Isabelle mechanism of generation it is a bit more complicated. This part is not entirely stable and it is difficult to know exactly what are the data structures that would be generated.

Chapter 7

Safety Critical Java

7.1 Introduction

In the previous chapter, we chose to test our approach in a case study with simple data types. This case study is very useful to show the different possible solutions for the use of our generated meta model. It also reveals the possible compositions with object oriented code generation in Scala. The purpose of this chapter is quite different. Indeed, this case study consists in the transformation of data types that define a Domain Specific Language (DSL). This example is an exercise sufficiently complex to demonstrate the effectiveness of our approach.

In this chapter, we start by the DSL definition, then we spell out the implementation before finishing with the effective results of the transformation of the DSL components using our transformation method.

7.2 Defining Safety Critical Java

Baklanova et al. are currently working on a real-time dialect of the Java language allowing to carry out specific static analyses of Java programs. We only sketch this language here; details are described in [17].

This language is not a genuine subset of Java, since some annotations characterizing timing behavior of program parts were added and inserted in particular comments into the program. Neither is the language a superset of Java, because syntactic restrictions on the shape of the program are imposed, and also static restrictions on the number of objects that are allocated.

This leads them to write their own syntax analysis, which is integrated into the Eclipse Xtext environment [40]. After syntax analysis and verification of the above-mentioned static restrictions, the program together with its timing annotations is translated to Timed Automata (TA) for model checking. The language is currently not entirely stable and will

be modified while the translation from Java to TA is improved, and while the formal model evolves. The formal aspect comes into play when developing a real-time semantics of the DSL in the proof assistant Isabelle, based on an execution semantics using inductive relations.

7.2.1 Elements of the Language

Safety Critical Java is a subset of the Java language augmented with timing annotations. In this section, we present the elements composing the language referring to the data type description presented in Figures 7.3 and 7.4.

Binary Operators The data type *binop* consists of arithmetic operators *BArith*, comparison operator *BCompar* and logical operators *BLogic*. Each of them defines a list of possible operators.

Expressions Expressions are defined within the type definition *Expr*. It is a variant type permitting different shapes for expressions. Constants are introduced by the constructor *Const*. The possible constant values are typed as Booleans, integers or strings. Variables are defined by a binding and a string. Binary operations (*BinOp*) take two expressions and a binary operator whereas ternary operations (*TernOp*) three expressions. Arrays (*ArrayE*) are specified by an array name followed by an expression (*expr*).

Statements The *stmt* datatype describes the statements allowed in the method body. It consists of the skip statement, assignments, conditions, while loops, sequence of statements, return, the annotation statement (in order to represent timing annotations) and a synchronization statement.

Declarations In Safety Critical Java, it is possible to declare variable (using the constructor *VarDecl*), arrays (using the constructor *ArrayDecl*) and objects (using the constructor *ObjDecl*). These are contained in the type definition *declaration*.

The data type *methodDecl* defines how to declare a method whereas *constrDecl* and *classDecl* introduce respectively constructor and class declarations.

Method definition The data type *methodDefn* offers two cases for a method definition. When it is a constructor method, it is built using a constructor declaration followed by a list of declarations and a statement composing the body of the constructor. When it is a basic method, it starts by a method declaration before a list of declarations and a statement.

Classes and Programs Finally, a Safety Critical Java program (*Prog*) consists of a list of class definitions. Each class definition begins by a class declaration followed by a list of declarations and method definitions.

7.3 Presentation of the Case Study

Figure H.1 shows the architecture of our application. There, green arrows represent model transformations or code generation. The base element is an Isabelle *theory* where data types, functions and proofs are defined. The corresponding meta-model is generated using the translation function described in Chapter 4. As was the case in Chapter 6, we use the Xtext tool to define a textual concrete syntax starting from the generated `Ecore` meta-model.

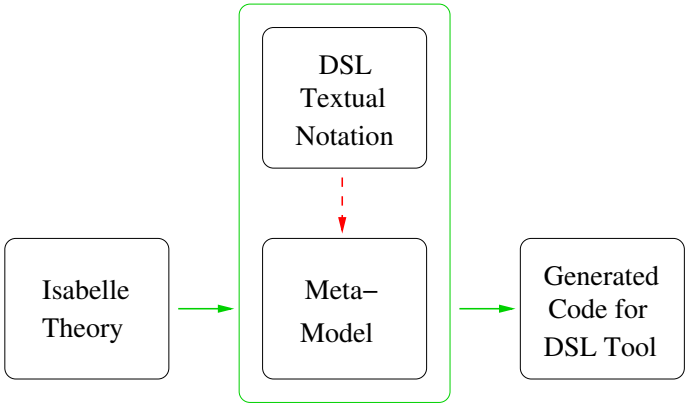


Figure 7.1: Datatype to Ecore Implementation Architecture

7.4 Generating an Ecore Diagram from Data Types

Figures 7.3 and 7.4 show data type definitions taken from the Isabelle *theory* where the verifications were performed. Figure 7.5 represents the examples of the corresponding accessor functions for our DSL Safety Critical Java. These parts of the Isabelle *theory* were given as input to the implementation of our translation rules presented in Chapter 4. The resulting `Ecore` diagram is presented in Figure 7.6.

As it is shown on Figure 7.6, data type definitions built only of type constructors and preceded by the comment `(**Enumerated*)` (*binding*, *barith*, *blogic*, *bcompar*, *tp*, *accModifier*) are treated as enumerations in the meta-model. Whereas data types composed of only one constructor derive a single class (as *var*, *superCl*, *interface*, *methodDecl*, *constrDecl*, *classDef* and *prog*).

The result of type definitions containing more than one constructor and at least a type expression (like *binop*, *value*, *expr*, *callExp*, *genType*, *stmt*, *declaration* and *methodDefn*) is modeled as a number of classes inheriting from a main one (a detailed description of the translation is provided at the end of this section).

For type expressions that represent lists of types (like *accModifier list* in datatype *declaration*), they generate a structural feature in the corresponding class and their multiplicities are set to *(0...*)*. The `EClass ClassDecl` contains an `EReference` named `superClass` whose multiplicities are `0...1`. This is due to the presence of the property *option* in the corresponding type expression defined in the datatype *ClassDecl*.

Finally, the translation of the *int*, *bool* and *string* types is straightforward. They are translated to respectively `EInt`, `EBoolean` and `EString`.

We submitted this generated meta-model to a validation tool provided by EMF. The meta-model has been successfully validated.

When we apply the inverse transformation (from meta-models to data types) on this generated meta-model, the result is the same data type description used to generate the meta-model in the first transformation.

Detailed Example: Transformation of Statements

To illustrate clearly the transformation of type expressions and variant types definitions, we focus on the translation of a particular data type: *stmt*. This data type is composed of a type constructor and several constructor definitions. It is then translated to a hierarchy of `EClasse`s. The type constructor is represented by the super class `Stmt` and for each constructor definition a child class is created. Type expressions in each constructor definition are translated into either `EAttribute`s when they consist of predefined types (such as the `EAttribute annot` of the `EClass AnnotStmt`); or to `EReference`s when it is a user defined type (such as the `EReference sBody` in `SyncStmt EClass`...).

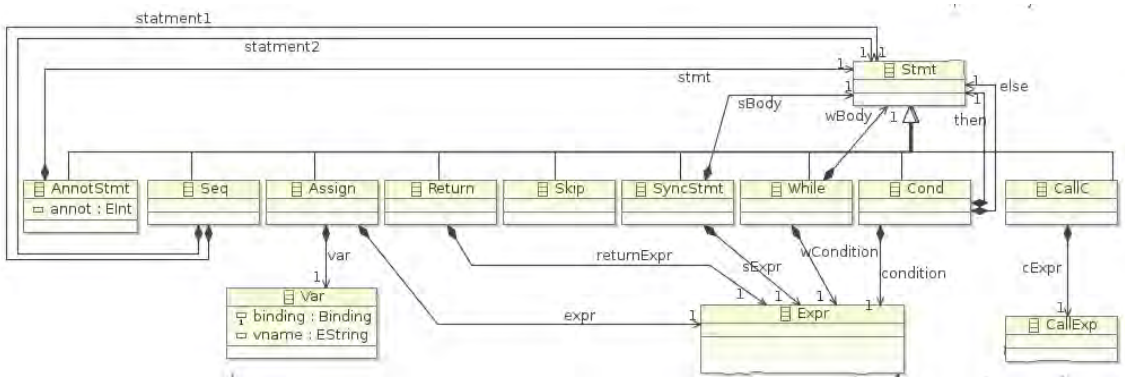


Figure 7.2: Statements Meta-model

7.5 Summary

In this second case study, we evaluated the efficiency of our transformation approach in a concrete Isabelle theory. The data types and accessor functions are described for a DSL described in [17]. This DSL is a Java-like language enriched with assertions which no off-the-shelf definition exists. The generated meta-model (of the DSL) can be easily used to generate a textual editor using tools like Xtext. This meta-model respects the well-formedness conditions set by Ecore and is successfully validated using the validation facilities of Ecore (on Eclipse).

```
(** Enumerated*)
datatype binding    = Local| Global
(** Enumerated*)
datatype barith     = BAadd| BAsub| BAmul| BAdiv| BAmod
(** Enumerated*)
datatype blogic     = BLand|BLor
(** Enumerated*)
datatype bcompar    = BCeq| BCge| BCgt| BCle| BClt| BCne

datatype binop      = BArith barith
                    |BCompar bcompar
                    |BLogic blogic

datatype value      = BoolV bool
                    |IntV int
                    |StringV string
                    |VoidV

datatype var        = Var binding string

datatype expr       = Const value
                    |VarE var
                    |BinOp binop expr expr
                    |TernOp expr expr expr
                    |ArrayE string expr
                    |This
                    |Null

datatype callExp    = CallObject string
                    |CallField callExp string
                    |CallMethod callExp string (expr list)
                    |CallConstructor string (expr list)

(** Enumerated*)
datatype tp         = BoolT| IntT| VoidT| StringT

datatype genType    = PrimitiveType tp
                    |CustomType string
```

Figure 7.3: Data Types for Safety Critical Java in Isabelle (1)

```
datatype stmt = Skip
              |Assign var expr
              |Seq stmt stmt
              |Cond expr stmt stmt
              |While expr stmt
              |CallC callExp
              |Return expr
              |AnnotStmt int stmt
              |SyncStmt expr stmt

(**Enumerated*)
datatype accessModifier = Public|Private|Abstract|Static|Protected|Synchronized

datatype declaration =
  VarDecl (accessModifier list) tp string
  |ArrayDecl (accessModifier list) genType string
  |ObjDecl (accessModifier list) string string

datatype superCl = SuperCl string

datatype interface = Interface string

datatype methodDecl =
  MethodDecl (accModifier list) genType string (declaration list)

datatype classDecl =
 ClassDeCl (accessModifier list)string(superCl option)(interface list)

datatype constrDecl =
  ConstrDeCl (accessModifier list) string (declaration list)

datatype methodDefn =
  MethodDefn methodDecl (declaration list) stmt
  |ConstrDefn constrDecl (declaration list) stmt

datatype classDefn =
  ClassDefn classDeCl (declaration list) (methodDefn list)

datatype prog = Prog (classDefn list)
```

Figure 7.4: Data Types for Safety Critical Java in Isabelle (2)

```
(**Accessor*)  fun bVal :: value => bool
               where bVal (BoolV (x)) = x;;
(**Accessor*)  fun iVal :: value => int
               where iVal (IntV (x)) = x;;
(**Accessor*)  fun sVal :: value => string
               where sVal (StringV (x)) = x;;

(**Accessor*)  fun binding :: var => binding
               where binding (Var (x, y)) = x;;
(**Accessor*)  fun vname :: var => string
               where vname (Var (x, y)) = y;;

(**Accessor*)  fun val :: expr => value
               where val (Const (x)) = x;;
(**Accessor*)  fun varExpr :: expr => var
               where varExpr (VarE (x)) = x;;
(**Accessor*)  fun bOp :: expr => binop
               where bOp (BinOp (x, y, z)) = x;;
```

Figure 7.5: Examples of Accessor Functions for Safety Critical Java in Isabelle

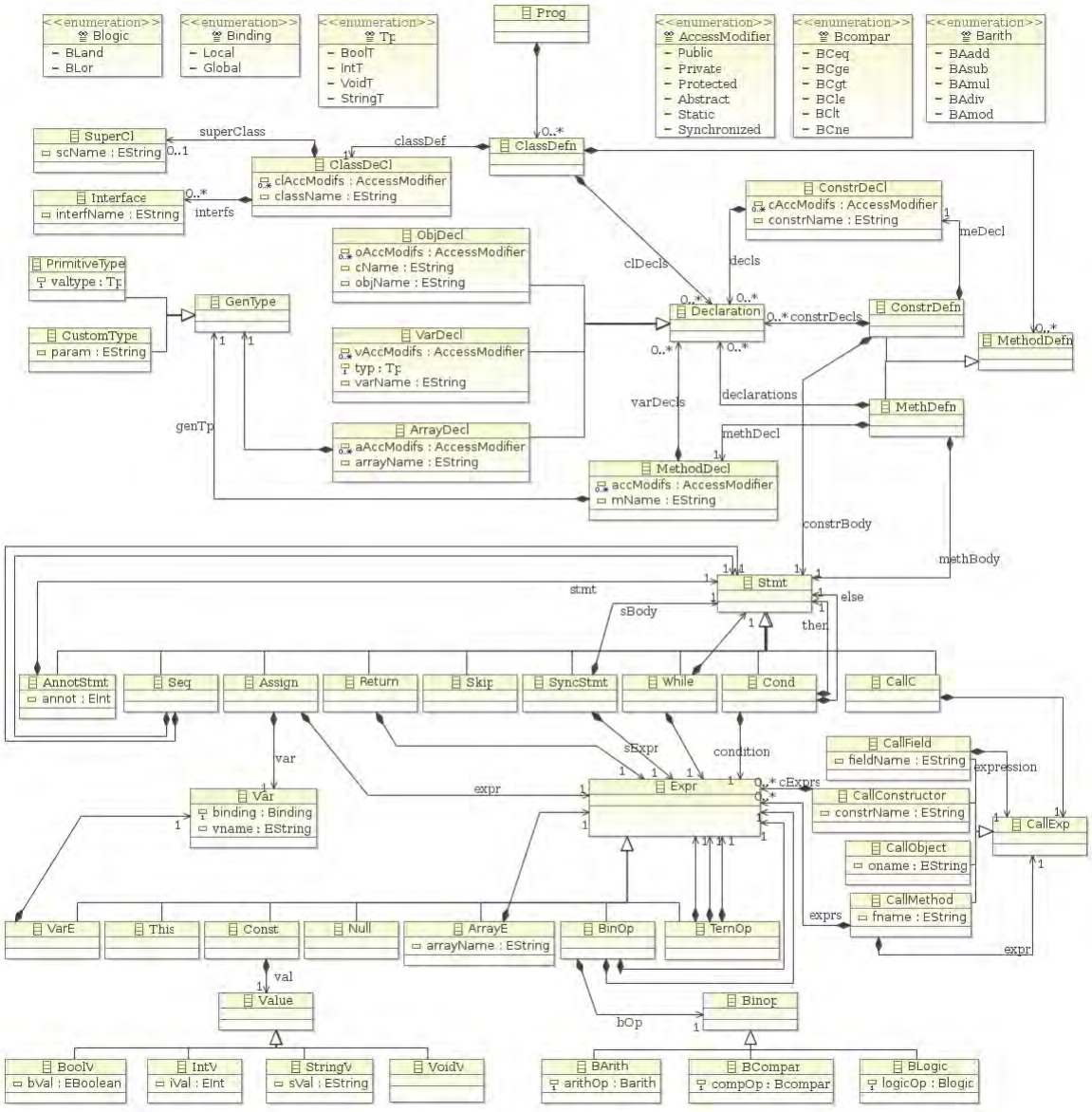


Figure 7.6: Resulting Ecore Diagram

Part IV

Conclusion

Conclusion and Perspectives

Thesis Summary

In this PhD thesis, we have presented a fully-automated MDE-based transformation process from functional data types into class diagrams and vice-versa. This work constitutes a first step towards the combination of interactive proof and Model Driven Engineering. This approach is dedicated to the simplification of the communication between professionals of formal proof and industrial experts, in the context of formally developing safety critical software.

In order to achieve this goal, we first have defined a subset of data types starting from an EBNF grammar of functional data type descriptions. This subset contains the essential elements needed to describe basic data type definitions including parameterized types. We also have conceived a particular syntax to write accessor functions that are further used for the transformation. From this grammar we have constructed a meta-model for data type descriptions which constitutes an essential part of an MDE-based transformation process.

After that, we have presented a subset of the Ecore meta-model which is sufficiently expressive to model basic class diagrams (including generic types). It constitutes the other part needed to perform the transformation.

We next have spelled out a set of transformation rules for the first side of the transformation: From data types to meta-models. We particularly have taken into account to completely cover the constructs defined by our data types grammar in order to avoid having untranslated constructs.

Based on this transformation, we derived the second side of the transformation: from meta-models to data types. For this side of the transformation, we had to define some well-formedness conditions to guarantee the correctness of the generated data types. These conditions are also used to ensure that the composition of our transformation gives identity as we start from the formal model.

Most transformation rules described in this thesis have been implemented in Java under the Eclipse platform. To represent the meta-models we used the Eclipse Modeling Framework and the Ecore plug-in. We also developed parsers for our subset of data types for both Caml and Isabelle. We have used this environment in order to test the approach with smaller examples and two major case studies described in the following.

The first consists in a verified Binary Decision Diagram (BDD) construction with sharing of subtrees. In this case study, we generated a meta-model starting from a data type description. Moreover, we used this meta-model to define a textual editor (to describe Boolean formulas) and graphical editor (to describe BDDs). We also couple this work with the generation of object oriented code from Isabelle theories.

The second constitutes a Domain Specific Language: Safety Critical Java. It is a Java-like language enriched with annotations. This case study constitutes an example of application which is complex enough to show the usability of our approach. Starting from data type definitions, set up for the semantic modeling of the DSL, we have been able to generate an EMF meta-model. The generated meta-model is used for documenting and visualizing the DSL, it can also be manipulated in the Eclipse workbench to generate a textual editor as an Eclipse plug-in.

Work in Progress

Besides the previously accomplished tasks, some work is still under development. It is presented in the following paragraphs:

Validating properties of our transformation rules

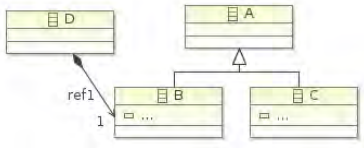
We are currently working on validating two properties of our transformation rules: Reflexivity and Bidirectionality.

After the implementation of the basic transformation rules, we put to the proof our approach by applying our transformation process on our own Ecore subset. The purpose behind such an action is to validate our approach by giving it a property of reflexivity. The expected result is to be able to generate in Caml a data type description for Ecore, which is close to the representation of the Ecore grammar (given in this thesis in Chapter 2). Applying our transformation rules on this subset allows us to analyze our transformation rules and identify particular cases that have been added later to the transformation rules.

The other property on which we are working is Bidirectionality. In fact, the composition of our two transformation functions gives the identity as we start with the formal model. We have detected this property when experimenting our transformations on different examples. In fact, when we apply our transformation function ($f()$) on a source model (M_s) of data types, we generate a class diagram M_T. If we apply to this generated M_T the opposite direction function ($f'()$) we get a model which is the same as M_s (the source model of the first direction of the transformation).

On the contrary, if we start by generating a data type model starting from a class diagram (using ($f'()$)) and we apply $f()$ to the resulting model, this model can be different from the source model. After analyzing our transformation rules we detect the rule that is responsible for this. This rule is used to translate a pattern that never appears in the

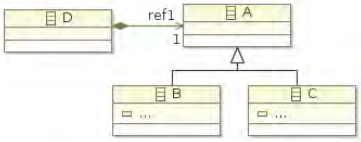
generated class diagrams. Figure 7.7 shows an example of this pattern. In the first sub-figure we can see the source meta-model. After performing the transformation the result is shown in the middle sub-figure. Finally, the last sub-figure is the result of applying the generated data type to the inverse direction of the transformation. The difference between the sub-figures 7.7a and 7.7c is the `EReference` *ref1* that is typed with the `EClass A` instead of the `EClass B`.



(a) Starting Source Model

```
type a =
B of ...
| C of ...

type d = D of a
```

(b) Generated Data Type



(c) Generated Model from Generated Data Type

Figure 7.7: Counter-Example for Bidirectionality

Implementing the transformation process as an Eclipse plug-in

Most transformation rules presented in this PhD thesis have been successfully implemented in Java using EMF as underlying representation of models. This implementation has been used to illustrate our approach with case studies. However, the transformation rules that have been defined during a second phase have not been implemented. We are currently working on implementing these remaining transformation rules. Next, we intend to gather all the transformation rules and wrap the whole application as an Eclipse plug-in. In such a context, it would be available and accessible to interested parties.

Perspectives

Many possibilities can be considered to complete this work:

Extending the subset of data types

In this thesis we couldn't handle all the constructs provided by functional languages. We have worked on a subset of the functional data types descriptions presented in Chapter 3. Even if we think that our subset is sufficiently expressive to construct basic data types, it could be extended to include more constructs in future work. This extension could be performed by adding a translation for mutually recursive data types. This kind of variant

types consists of two data types separated by the key word `"and"`, where each of them is present in the type expression of the other.

Also, formal languages and object-oriented languages have divergent objectives and for this reason are built differently. Therefore some components of functional languages are not translatable to the object-oriented languages. In this context, we mention the example of the higher-order types in Caml that cannot find an equivalent in Java. We must therefore exclude such types from our extension plans.

Moreover, we plan to broaden the scope of our transformations to other functional languages and interactive provers that have an abstract syntax for data types close to the subset on which we work. We mention among these systems the Coq proof assistant [29].

Extending the subset of Ecore

In Chapter 2, we have described a subset of Ecore allowing to define basic class diagrams. Although this subset contains a sufficiently rich set of components to represent the majority of Ecore diagrams, it would be desirable to extend it in order to propose transformations for other components, in particular `EAnnotation`s and `EOperation`s.

In addition, we would like to reduce the well-formedness constraints imposed to the user in order to increase the number of translatable patterns. This could be done for example by allowing the user to translate deeper class hierarchy.

Performing proofs on the transformations

The transformations described in Chapters 4 and 5 have been developed in a mathematical language (in a functional way as if we had implemented them in Caml). It would be interesting to prove some properties on these transformations using manual or mechanized proof. Such a process requires to firstly attribute a formal semantics for both parts of the transformation: our subsets of class diagrams and functional data types. Then, to define the properties to be checked as for example the preservation of the semantics after performing transformation.

Transforming the dynamic part of functional languages

In Part II, we defined transformations to and from the structural part of the functional programs: data types. In future work, we propose to transform the dynamic part of ML languages represented by functions. It would be useful to model these functions on class diagrams using Ecore `EOperation`. This attribute allows to represent functions on `EClass`es as interfaces (function name and parameters) but it does not allow to model their behavior.

In another approach, it would be interesting to prospect a way to represent the behavior of functions. In other words, find a target processing that would be adapted to the representation of the behavior of functions even if it is not implemented by EMF, for

example state/transition diagrams [53]. These functions could be written in a specific style which is close to state transition diagrams.

Giving indications about proofs

In the context of collaborations with the industrial field, the domain experts of the industrial field may have trouble to see the representation of their systems in the proof assistants but also the proofs performed there. For this problem, the documentation of proofs seems to be a good solution. It would be interesting to conduct proofs on a system using an interactive proof environment, then to transform data types in class diagrams together with documentation about some proofs. This documentation can be represented as annotations on the generated meta-models. We could use as annotation language the OCL [76] language in order to take advantage of its structure which is adapted to the description of formalization.

Bibliography

[1] Jean-Raymond Abrial. *The B-book - assigning programs to meanings*. Cambridge University Press, 2005.

[2] AGG Development Team. AGG website, 2013. Available from: http://user.cs.tu-berlin.de/~gragra/agg/.

[3] Aditya Agrawal, Gabor Karsai, Sandeep Neema, Feng Shi, and Attila Vizhanyo. The design of a language for model transformations. *Software and System Modeling*, 5(3):261–288, 2006.

[4] Sheldon B. Akers. Binary Decision Diagrams. *IEEE Trans. Computers*, 27(6):509–516, 1978.

[5] Marcus Alanen and Ivan Porres. A relation between context-free grammars and meta object facility metamodels. Technical report, Turku Centre for Computer Science (TUCS), March 2003.

[6] Jesús Almendros-Jiménez, Luis Iribarne, José Asensio, Nicolás Padilla, and Cristina Vicente-Chicote. An Eclipse GMF tool for modelling user interaction. In Miltiadis Lytras, Ernesto Damiani, John Carroll, Robert Tennyson, David Avison, Ambjörn Naeve, Adrian Dale, Paul Lefrere, Felix Tan, Janice Sipior, and Gottfried Vossen, editors, *Visioning and Engineering the Knowledge Society. A Web Science Perspective*, volume 5736 of *Lecture Notes in Computer Science*, pages 405–416. Springer Berlin / Heidelberg, 2009.

[7] Carsten Amelunxen, Alexander Königs, Tobias Rötschke, and Andy Schürr. MOFLON: A standard-compliant metamodeling framework with graph transformations. In Arend Rensink and Jos Warmer, editors, *ECMDA-FA*, volume 4066 of *Lecture Notes in Computer Science*, pages 361–375. Springer, 2006.

[8] Carsten Amelunxen and Andy Schürr. Formalising model transformation rules for UML/MOF 2. *IET Software*, 2(3):204–222, 2008.

[9] Moussa Amrani, Levi Lucio, Gehan M. K. Selim, Benoît Combemale, Jürgen Dingel, Hans Vangheluwe, Yves Le Traon, and James R. Cordy. A tridimensional approach for studying the formal verification of model transformations. In Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche, editors, *ICST*, pages 921–928. IEEE, 2012.

[10] Kyriakos Anastasakis. *A Model Driven Approach for the Automated Analysis of UML Class Diagrams*. PhD thesis, The University of Birmingham, May 2009.

[11] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. UML2Alloy: A challenging model transformation. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *MoDELS*, volume 4735 of *Lecture Notes in Computer Science*, pages 436–450. Springer, 2007.

[12] Suzana Andova, Mark van den Brand, and Luc Engelen. Prototyping the semantics of a DSL using ASF+SDF: Link to formal verification of DSL models. In Francisco Durán and Vlad Rusu, editors, *AMMSE*, EPTCS, pages 65–79, 2011.

[13] Anthony Anjorin, Marius Lauder, Sven Patzina, and Andy Schürr. eMoflon: Leveraging EMF and professional CASE Tools. In H.U. Heiß, P. Pepper, H. Schlingloff, and J. Schneider, editors, *INFORMATIK 2011*, volume 192 of *Lecture Notes in Informatics*, page 281, Bonn, October 2011. Gesellschaft für Informatik. Extended abstract.

[14] Atelier-B Development Team. Atelier-B website, 2013. Available from: http://www.atelierb.eu.

[15] ATL Development Team. ATL website, 2013. Available from: http://www.eclipse.org/atl/.

[16] AToM3 Development Team. AToM3 website, 2013. Available from: http://atom3.cs.mcgill.ca/.

[17] Nadezhda Baklanova and Martin Strecker. Abstraction and verification of properties of a real-time java. In Vadim Ermolayev, Heinrich C. Mayr, Mykola Nikitchenko, Aleksander Spivakovsky, and Grygoriy Zholtkevych, editors, *ICT in Education, Research, and Industrial Applications*, volume 347 of *Communications in Computer and Information Science*, pages 1–18. Springer Berlin Heidelberg, 2013.

[18] Jean Bézivin. In search of a basic principle for model driven engineering. *Upgrade*, 5(2):21–24, April 2004.

[19] Jean Bézivin. Model driven engineering: An emerging technical space. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, pages 36–64. Springer Berlin / Heidelberg, 2006.

[20] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the OMG/MDA framework. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01, pages 273–280, Washington, DC, USA, 2001. IEEE Computer Society.

[21] Enrico Biermann. EMF model transformation based on graph transformation: formal foundation and tool environment. In Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, and Andy Schürr, editors, *Proceedings of the 5th International Conference on Graph Transformations*, volume 6372 of *ICGT'10*, pages 381–383, Berlin, Heidelberg, 2010. Springer-Verlag.

[22] Jean-Paul Bodeveix, Mamoun Filali, Julia Lawall, and Gilles Muller. Formal methods meet domain specific languages. In Judi Romijn, Graeme Smith, and Jaco van de Pol, editors, *Proceedings of the 5th international conference on Integrated Formal Methods*, volume 3771 of *Lecture Notes in Computer Science*, pages 187–206, Berlin, Heidelberg, 2005. Springer-Verlag.

[23] Achim D. Brucker. *An Interactive Proof Environment for Object-Oriented Specifications*. PhD thesis, ETH Zurich, March 2007.

[24] Achim D. Brucker, Jürgen Doser, and Burkhart Wolff. An MDA framework supporting OCL. *ECEASST*, 5:1–18, 2006.

[25] Achim D. Brucker and Burkhart Wolff. HOL-OCL: A formal proof environment for UML/OCL. In José Luiz Fiadeiro and Paola Inverardi, editors, *FASE*, volume 4961 of *Lecture Notes in Computer Science*, pages 97–100. Springer, 2008.

[26] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[27] Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[28] Jordi Cabot, Robert Clarisó, Esther Guerra, and Juan de Lara. Verification and validation of declarative model-to-model transformations through invariants. *J. Syst. Softw.*, 83(2):283–302, February 2010.

[29] Coq Development Team. *The Coq Proof Assistant Reference Manual. Version 8.31*, 2010. http://coq.inria.fr/refman/.

[30] Thierry Coquand and Peter Dybjer. Inductive definitions and type theory an introduction (preliminary version). In P.S. Thiagarajan, editor, *Foundation of Software Technology and Theoretical Computer Science*, volume 880 of *Lecture Notes in Computer Science*, pages 60–76. Springer Berlin Heidelberg, 1994.

[31] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, July 2006.

[32] Juan de Lara and Hans Vangheluwe. AToM^3: A tool for multi-formalism and meta-modelling. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 174–188. Springer Berlin / Heidelberg, 2002.

[33] Juan de Lara and Hans Vangheluwe. Using AToM^3 as a meta-case tool. In *Proceedings of the 4st International Conference on Enterprise Information Systems (ICEIS)*, pages 642–649, Ciudad Real, Spain, April 2002.

[34] John Deacon. *Model-View-Controller (MVC) Architecture*, 2009. Training Material.

[35] Marcos D. Del Fabro, Jean Bézivin, Frédéric Jouault, Erwan Breton, and Guillaume Gueltas. AMW: a generic model weaver. In *Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM05)*, 2005.

[36] David Delahaye, Jean-Frédéric Étienne, and Véronique Viguié Donzeau-Gouge. A Formal and Sound Transformation from Focal to UML: An Application to Airport Security Regulations. In *UML and Formal Methods (UML&FM)*, Innovations in Systems and Software Engineering (ISSE) NASA Journal, Kitakyushu-City (Japan), October 2008. Springer.

[37] Greg Dennis, Robert Seater, Derek Rayside, and Daniel Jackson. Automating commutativity analysis at the design level. In George S. Avrunin and Gregg Rothermel, editors, *ISSTA*, pages 165–174. ACM, 2004.

[38] Fujaba development team. Fujaba toolsuite website, 2012. Available from: http://www.fujaba.de/.

[39] Catherine Dubois, Thérèse Hardin, and Véronique Donzeau-Gouge. Building certified components within Focal. In Hans-Wolfgang Loidl, editor, *Trends in Functional Programming*, volume 5 of *Trends in Functional Programming*, pages 33–48. Intellect, 2004.

[40] Eclipse Community. *Tutorials and Documentation for Xtext 2.0*, 2011. http://www.eclipse.org/Xtext/documentation/.

[41] Karsten Ehrig, Esther Guerra, Juan de Lara, Laszló Lengyel, Tihamér Levendovszky, Ulrike Prange, Gabriele Taentzer, Dániel Varró, and Szilvia Varró-Gyapay. Model transformation by graph transformation: A comparative study. In *Proceedings of MTiP 2005, International Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005)*, October 2005.

[42] Andy Evans, Robert France, and Emanuel Grant. Towards Formal Reasoning With UML Models. In *OOPSLA'99 Workshop On Behavioral Semantics*, 1999.

[43] Elie Fares. *Real-Time Systems Refinement - Application to the Verification of Web Services*. PhD thesis, Université de Toulouse, IRIT, March 2013.

[44] Houda Fekih, Leila Jemni Ben Ayed, and Stephan Merz. Transformation of B specifications into UML class diagrams and state machines. In Hisham Haddad, editor, *SAC*, pages 1840–1844. ACM, 2006.

[45] Robert B. France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19(7):325–334, 1998.

[46] M.D. Fraser, K. Kumar, and V.K. Vaishnavi. Informal and formal requirements specification languages: bridging the gap. *Software Engineering, IEEE Transactions on*, 17(5):454–466, 1991.

[47] Geri Georg, Jores Bieman, and Robert B. France. Using Alloy and UML/OCL to specify run-time configuration management: A case study. In Andy Evans, Robert B. France, Ana M. D. Moreira, and Bernhard Rumpe, editors, *pUML*, volume 7 of *LNI*, pages 128–141. GI, 2001.

[48] Mathieu Giorgino. *Inductive Specifications and Refinement of Pointer Structures*. PhD thesis, University of Toulouse, 2013.

[49] Mathieu Giorgino and Martin Strecker. BDDs verified in a proof assistant (preliminary report). In *Proceedings TAAPSD'2010*, Univ. Taras Shevchenko, Kiev, 2010.

[50] Martin Gogolla, Fabian Büttner, and Mark Richters. USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming*, 69(1 - 3):27 – 34, 2007. Special issue on Experimental Software and Toolkits.

[51] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley, Upper Saddle River, NJ, 2009.

[52] Ahmed Hammad, Bruno Tatibouët, Jean-Christophe Voisinet, and Weiping Wu. From a B specification to UML statechart diagrams. In Chris George and Huaikou Miao, editors, *ICFEM*, volume 2495 of *Lecture Notes in Computer Science*, pages 511–522. Springer, 2002.

[53] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[54] Lotfi Hazem, Nicole Levy, and Rafael Marcano-Kamenoff. UML2B: un outil pour la génération de modèles formels. In *J.Julliand (ed.), AFADL'2004 - Session Outils*, 2004.

[55] Henshin Development Team. Henshin website, 2013. Available from: http://www.eclipse.org/henshin/.

[56] Akram Idani. *B/UML : Mise en relation de specifications B et de descriptions UML pour l'aide à la validation externe de developpements formels en B*. PhD thesis, Université Joseph Fourier, November 2006.

[57] Akram Idani. UML models engineering from static and dynamic aspects of formal specifications. In Terry A. Halpin, John Krogstie, Selmin Nurcan, Erik Proper, Rainer Schmidt, Pnina Soffer, and Roland Ukor, editors, *BMMDS/EMMSAD*, volume 29 of *Lecture Notes in Business Information Processing*, pages 237–250. Springer, 2009.

[58] Akram Idani, Jean-Louis Boulanger, and Laurent Philippe. A generic process and its tool support towards combining UML and B for safety critical systems. In Gongzhu Hu, editor, *CAINE*, pages 185–192. ISCA, 2007.

[59] Akram Idani, Dieu Donné Okalas Ossami, and Jean-Louis Boulanger. Commandments of UML for safety. In *Proceedings of the Second International Conference on Software Engineering Advances (ICSEA 2007)*, page 58, Cap Esterel, French Riviera, France, 2007. IEEE Computer Society.

[60] ISO. Information technology – Z formal specification notation – Syntax, type system and semantics. Technical Report ISO/IEC 13568, International Organization for Standardization, 2002.

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

[62] Frédéric Jouault and Ivan Kurtev. Transforming models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138, Berlin, 2006. Springer Verlag.

[63] Mounira Kezadri, Benoît Combemale, Marc Pantel, and Xavier Thirioux. A proof assistant based formalization of MDE components. In Farhad Arbab and Peter Csaba Ölveczky, editors, *FACS*, volume 7253 of *Lecture Notes in Computer Science*, pages 223–240. Springer, 2011.

[64] Soon-Kyeong Kim and David A. Carrington. Formalizing the UML Class Diagram Using Object-Z. In Robert B. France and Bernhard Rumpe, editors, *UML*, volume 1723 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 1999.

[65] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained : The Model Driven Architecture : Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[66] Ivan Kurtev. State of the art of QVT: a model transformation language standard. In A. Schürr, M. Nagl, and A. Zündorf, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 5088 of *Lecture Notes in Computer Science*, pages 377–393, Berlin, October 2008. Springer Verlag.

[67] K. Lano, S. Kolahdouz-Rahimi, and T. Clark. Comparing verification techniques for model transformations. In *Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation*, MoDeVVa '12, pages 23–28, New York, NY, USA, 2012. ACM.

[68] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system release 3.12. Documentation and user's manual*, July 2011. http://caml.inria.fr/pub/docs/manual-ocaml/index.html.

[69] Tiago Massoni, Rohit Gheyi, and Paulo Borba. A UML class diagram analyzer. In *In 3rd International Workshop on Critical Systems Development with UML, affiliated with 7th UML Conference*, pages 143–153, 2004.

[70] MOFLON Development Team. MOFLON website, 2013. Available from: http://www.moflon.org/.

[71] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In Lionel C. Briand and Clay Williams, editors, *International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278, Montego Bay, Jamaica, October 2005. Springer.

[72] Pierre-Alain Muller, Franck Fleurey, Didier Vojtisek, Zoé Drey, Damien Pollet, Frédéric Fondement, Philippe Studer, and Jean-Marc Jézéquel. On executable meta-languages applied to model transformations. In *Model Transformations In Practice Workshop*, Montego Bay, Jamaïque, October 2005.

[73] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, May 2012.

[74] Object Management Group. *MDA Guide Version 1.0.1*, June 2003.

[75] Object Management Group. *Meta Object Facility (MOF) Core 2.0*, 2006.

[76] Object Management Group. *Object Constraint Language (OCL) 2.2 Specification*, February 2010.

[77] Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification 1.1*, January 2011.

[78] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2007.

[79] Dieu Donné Okalas Ossami, Jean-Pierre Jacquot, and Jeanine Souquières. Consistency in UML and B multi-view specifications. In Judi Romijn, Graeme Smith, and Jaco van de Pol, editors, *IFM*, volume 3771 of *Lecture Notes in Computer Science*, pages 386–405. Springer, 2005.

[80] Sven Patzina and Lars Patzina. A case study based comparison of ATL and SDM. In Andy Schürr, Dániel Varró, and Gergely Varró, editors, *Proceedings of the 4th international conference on Applications of Graph Transformations with Industrial Relevance, AGTIVE 2011*, volume 7233 of *Lecture Notes in Computer Science*, pages 210–221, Berlin, Heidelberg, 2012. Springer.

[81] Simon Peyton-Jones. *Haskell 98 language and libraries : the revised report*. Cambridge University Press, Cambridge U.K. New York, 2003.

[82] Celia Picard. *Representation coinductive des graphes*. PhD thesis, Université Toulouse III Paul Sabatier, June 2012.

[83] Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universitaet Bremen, Logos Verlag, Berlin, BISS Monographs, No.14, 2002.

[84] Bran Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003.

[85] Seyyed M. A. Shah, Kyriakos Anastasakis, and Behzad Bordbar. From UML to Alloy and back again. In Sudipto Ghosh, editor, *MoDELS Workshops*, volume 6002 of *Lecture Notes in Computer Science*, pages 158–171. Springer, 2009.

[86] Colin Snook and Michael Butler. Using a graphical design tool for formal specification. In *Proceedings of the 13th Annual Workshop of the Psychology of Programming Interest Group (PPIG)*, pages 311–321, 2001.

[87] Colin Snook and Michael Butler. UML-B: Formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122, January 2006.

[88] Fabio Somenzi. Binary Decision Diagrams. In *Calculational System Design, volume 173 of NATO Science Series F: Computer and Systems Sciences*, pages 303–366. IOS Press, 1999.

[89] Guy L. Steele. *Common LISP*. Digital Press, 2nd edition, 1990.

[90] Matthew Stephan and Andrew Stevenson. A comparative look at model transformation languages. Queen's University, Kingston, Canada, 2009.

[91] Perdita Stevens. A landscape of bidirectional model transformations. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *GTTSE*, volume 5235 of *Lecture Notes in Computer Science*, pages 408–424. Springer, 2007.

[92] Gabriele Taentzer. AGG: A graph transformation environment for modeling and validation of software. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *AGTIVE*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer, 2003.

[93] Xavier Thirioux, Benoît Combemale, Xavier Crégut, and Pierre-Loïc Garoche. A framework to formalise the MDE foundations. In Richard Paige and Jean Bézivin, editors, *International Workshop on Towers of Models (TOWERS)*, Zurich, Switzerland, 2007.

[94] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[95] Dániel Varró, Gergely Varró, and András Pataricza. Designing the automatic transformation of visual languages. *Sci. Comput. Program.*, 44(2):205–227, August 2002.

[96] Jean-Christophe Voisinet. *Contribution au processus de développement d'applications specifiées à l'aide de la methode B par validation utilisant des vues UML et traduction vers des langages objets*. PhD thesis, Université de Franche-Comté, September 2004.

[97] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, Wieland Schwinger, Dimitris Kolovos, Richard Paige, Marius Lauder, Andy Schürr, and Dennis Wagelaar. A comparison of rule inheritance in model-to-model transformation languages. In Jordi Cabot and Eelco Visser, editors, *Proceedings of the 4th international conference on Theory and practice of model transformations (ICMT 2011)*, volume 6707 of *Lecture Notes in Computer Science*, pages 31–46, Berlin, Heidelberg, 2011. Springer-Verlag.

[98] Manuel Wimmer and Gerhard Kramler. Bridging grammarware and modelware. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Doctoral Symposium, Educators Symposium, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers*, volume 3844 of *Lecture Notes in Computer Science*, pages 159–168. Springer, 2005.

Part V

Résumé de la thèse en Français

Appendix A

Introduction

Contexte et motivation

La thèse décrite dans ce document a été effectuée à Toulouse. Cette ville est devenu le centre de l'industrie aérospatiale européenne, ce qui a créé une forte demande pour les systèmes et logiciels du domaine aérospatial. Ces logiciels sont appelés : logiciels embarqués. Ces sont des logiciels complexes qui requièrent le zéro défaut. Contrairement aux logiciels que nous utilisons dans notre vie quotidienne (jeux, web, e-mail ... etc.), des défauts sur ce type de logiciels peuvent entraîner des pertes financières considérables ou même mettre en danger des vies humaines. C'est pourquoi ces logiciels sont qualifiées de "critiques". Pour garantir les hautes exigences requises par ces logiciels, les industriels s'appuient sur des méthodes formelles pour certifier leurs applications.

En effet, les méthodes formelles (tels que les assistants de preuves [73, 29]) sont de plus en plus utilisées dans le développement et la vérification de code d'application. Elles ont de solides bases formelles ainsi que des sémantiques précises, cependant elles utilisent des notations complexes qui pourraient être difficiles à comprendre pour les utilisateurs non habitués à ce type de notations. Cela devient un problème quand une collaboration est nécessaire entre les professionnels de la preuve interactive et les experts d'un domaine applicatif. Ces experts ont souvent du mal à voir précisément comment leurs systèmes ont été spécifiés dans les assistants de preuve.

D'un autre côté, les experts du domaine applicatif sont souvent habitués à interagir avec les outils et formalismes proposés par l'ingénierie dirigée par les modèles (IDM). Cette méthode peut compter sur ses langages de spécification visuels tels que les diagrammes de classes qui utilisent des notations intuitives. Ces diagrammes permettent de spécifier, visualiser, comprendre et documenter des systèmes. Cependant, ces langages souffrent d'un manque de sémantique précise. En outre, ils ne sont pas adaptés à la vérification des systèmes. Nous nous intéressons dans cette thèse à l'association de ces deux domaines complémentaires que sont des méthodes formelles et l'IDM en effectuant des traductions

entre les éléments de l'un dans l'autre.

Survol de l'approche

Afin de traduire les types de données fonctionnels (utilisés dans les prouveurs interactifs comme Coq ou Isabelle) en diagrammes de classes et vice versa, nous utilisons une méthode de transformation basée sur l'IDM. Cette méthode permet de définir un processus de transformation générique des types de données fonctionnels vers les méta-modèles ainsi que dans le sens contraire (des méta-modèles vers les types de données).

Figure A.1 montre un aperçu de notre approche. Dans le premier sens de la traduction, nous dérivons un méta-modèle de types de données à partir d'une représentation sous forme de grammaire EBNF permettant la définition de types de données [73]. Ce méta-modèle représente la source de notre processus de transformation. Les diagrammes de classes sont quant à eux représentés par Ecore: le langage de base de l'Eclipse Modeling Framework [51]. Nous décrivons ensuite un sous-ensemble du méta-modèle de Ecore afin d'en faire la cible de notre transformation. Les règles de transformation sont définies sur le méta-niveau et font correspondre les éléments du méta-modèle source et ceux du méta-modèle cible. La fonction*DataTypeToEcore* implémente ces règles de transformations en Java. Elle prend en entrée un modèle conforme au méta-modèle source et retourne son équivalent dans un modèle conforme au méta-modèle cible. Nous utilisons la correspondance entre les éléments des deux méta-modèles afin de définir la seconde direction de la transformation.

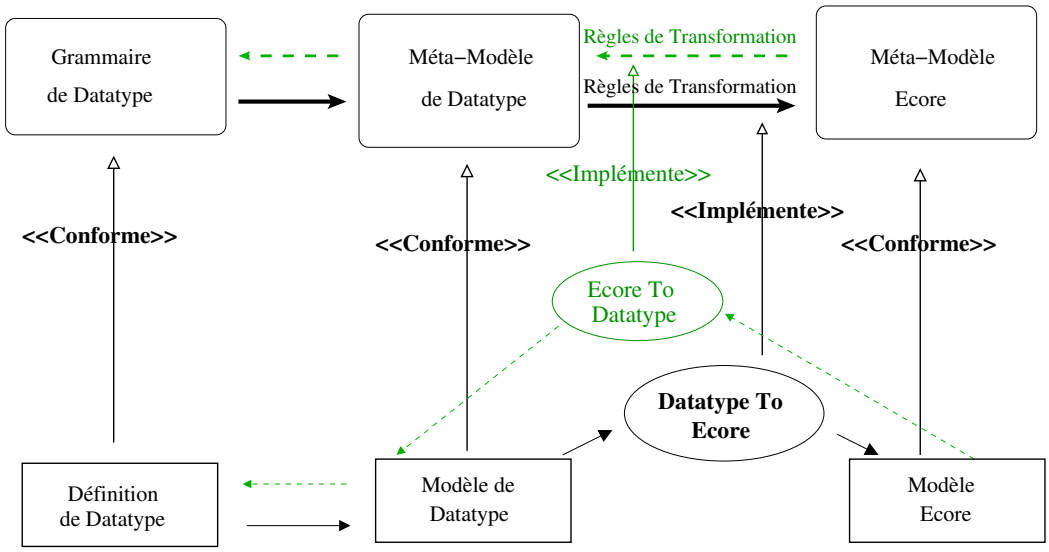


Figure A.1: Vue d'ensemble de la méthode de transformation

Contributions

Les contributions de cette thèse consistent en plusieurs objectifs atteints qui sont présentés dans ce qui suit:

- Nous définissons un sous-ensemble de types de données (commun à Isabelle et Caml) qui contient les éléments essentiels nécessaires à la description de la majorité des formes que prennent les types de données. Nous construisons par la suite le méta-modèle représentant ce sous-ensemble.
- Nous définissons aussi un sous-ensemble du méta-modèle de Ecore qui en même temps est suffisamment expressif pour modéliser les diagrammes de classes de base mais aussi qui contient des éléments traduisibles vers des types de données fonctionnels.
- Nous décrivons dans un premier sens, un processus de transformation entièrement automatisé basé sur l'IDM permettant de transformer des types de données fonctionnels en méta-modèles. Afin d'assurer la validité de nos méta-modèles générés en Ecore, nous proposons des contraintes de bonne formation sur les types de données.
- Nous présentons aussi une transformation dans le sens inverse : à partir des méta-modèles vers les types de données. Nous sélectionnons aussi les patterns de méta-modèles non-traduisibles. Cette étape nécessite de poser certaines conditions de bonne formation assurant la validité des types de données générés.
- Les transformations décrites dans cette thèse sont implémentées en utilisant Java et EMF sous forme d'une application qui est utilisée dans les études de cas. Nous avons aussi couplé ces travaux avec à la fois la génération d'éditeurs graphiques (et/ou textuels) ainsi qu'avec la génération de code orienté objet certifié.
- Nous illustrons la faisabilité de notre approche avec deux études de cas : la construction de diagrammes de décision binaires et la définition d'un DSL nommé Safety Critical Java.

Plan de la thèse

Cette thèse est organisée comme suit :

Le chapitre 1 décrit les notions de base de l'IDM ainsi que celles de la transformation de modèles. Il donne également un aperçu des travaux connexes qui consistent principalement en différentes approches visant à combiner la vérification formelle et l'IDM. Le chapitre 2 consiste en la présentation du cadre de méta-modèlisation que nous avons choisi : Eclipse Modeling Framework. Le chapitre 3 quant à lui fixe le cadre formel utilisé dans le reste de la thèse, nous y introduisons quelques notions des langages formels tels que Caml et Isabelle.

Les contributions fondamentales de cette thèse sont données dans la Partie II. Les chapitres 4 et 5 introduisent respectivement notre transformation bidirectionnelle des types de données fonctionnels vers les diagrammes de classes (représenté en Ecore) et dans le sens contraire à partir des diagrammes de classes vers les types de données.

Les chapitres 6 et 7 à leur tour illustrent nos contributions par deux études de cas: les diagrammes de décision binaires et le langage Safety Critical Java. Nous cloturons le document par tirer une conclusion de nos travaux et présentons quelques perspectives.

Appendix B

Contexte scientifique et travaux connexes

Cette thèse consiste principalement en la transformation des types de données utilisés dans la programmation fonctionnelle vers les diagrammes de classes et vice versa dans le cadre de l'ingénierie dirigée par les modèles (IDM). Dans ce chapitre, nous donnons un aperçu des concepts de base de l'IDM et de la transformation de modèles. Par la suite, nous présentons les approches existantes visant à l'intégration de l'IDM avec les méthodes formelles en général. Nous clôturons le chapitre en positionnant notre approche par rapport à ces dernières.

B.1 Contexte scientifique

B.1.1 Ingénierie Dirigée par les Modèles (IDM)

L'ingénierie des modèles (IDM) est une méthodologie de développement de logiciels où les modèles sont des éléments centraux dans le processus de développement. Il s'agit d'un type particulier de l'ingénierie générative dans laquelle tout ou partie d'une application est engendrée à partir d'un modèle.

En IDM, les modèles sont les principaux artéfacts du cycle de vie de développement. Malgré cela, il n'existe pas de définition unique de ce qu'est un *modèle*. Parmi les définitions disponibles dans la littérature [65, 74], *Bézivin & Gerbé* [20] le définissent comme suit : Un modèle est une simplification d'un système construit avec un objectif précis. Le modèle doit être en mesure de répondre aux questions à la place du système actuel.

Un *méta-modèle* définit les éléments d'un langage permettant d'exprimer des modèles. Il décrit les différents types de composants du modèle et la façon dont ils sont disposés et reliés [19]. Les instances des éléments du méta-modèle sont utilisés pour construire un modèle du langage.

Les méta-modèles sont utilisés pour définir la syntaxe abstraite des langues appartenant à un domaine particulier : Domain Specific Languages (DSLs). Ils constituent le cœur de l'IDM.

B.1.2 Transformation de Modèles

La transformation de modèles est l'une des techniques essentielles de l'IDM. La principale motivation derrière la transformation de modèles est la possibilité d'automatiser les aspects de routine des processus afin de rendre le développement et la maintenance des logiciels plus efficace. *Kleppe et al* [65] définit une *transformation* comme la génération automatique d'un modèle de la cible à partir d'un modèle de source, selon une définition de la transformation.

B.2 Travaux connexes

Malgré ses avantages, l'IDM souffre d'un manque de base formelle solide. Pour remédier à cette lacune, les différentes équipes de recherche travaillent sur l'intégration des méthodes formelles existantes et de l'IDM. Ici, nous nous intéressons particulièrement à l'interaction des langages formels et de la modélisation.

B.2.1 Méthodes formelles et IDM

Dans cette catégorie, nous commençons par présenter le travail de *Richters* [83]. L'auteur présente une définition formelle des diagrammes de classes UML et la sémantique de OCL en utilisant la théorie des ensembles et la logique du premier ordre. Son approche a été utilisée pour détecter les incohérences et de valider les règles de bonne formation du standard UML 2.0. Il a ensuite été mis en place comme un outil pour la spécification et la validation des systèmes (UML / OCL): l'outil de USE (UML-based Specification Tool) [50].

Les approches suivantes sont concernées par la description de frameworks de modélisation ans des assistants de preuve comme Coq [29] et Isabelle [73].

Coq4MDE Coq4MDE [93] est un environnement de preuve permettant de décrire des aspects de l'IDM dans le but de lui fournir des bases formelle. Pour ce faire, les développeurs définissent la notion de modèle et de modèle de référence (méta-modèle). Ils introduisent également deux relations fondamentales pour l'IDM : *conformsTo* et *promotion*. La cohérence de l'approche est validée par la preuve que EMOF est auto défini (en utilisant l'assistant de preuve Coq [29]).

Dans [63], Coq4MDE est étendu pour supporter la composition d'éléments de modèle. Ce travail consiste en la formalisation des technologies de vérification composables afin de faciliter l'intégration de composants pré-vérifiées.

Un environnement de preuve formelle pour UML/OCL : HOL/OCL L'outil HOL-OCL [25] est un environnement interactif de preuves pour les diagrammes de classe UML annoté avec des spécifications OCL[76]. Il est basé sur un repository pour les modèles UML/OCL et l'assistant de preuve Isabelle/HOL [73]. Ce système fournit un moyen d'exécuter des preuves sur les méta-modèles décrits en UML. Pour ce faire, les structures de données orienté objet ainsi que les concepts de l'IDM ont été encodés grâce a un plogement peu profond en Isabelle. Selon notre avis, un tel environnement fonctionnel et formel n'est pas bien adapté à la représentations de composants orientés objet. Cela donne lieu à un environnement complexe qui rend le processus de preuve plus difficile en Isabelle.

La transformation est l'un des moyens utilisés pour établir les liens entre les langages formels et les langages fournis par l'IDM. Elle permet de décrire un système utilisant un langage formel, mais aussi de représenter le même système en utilisant un langage de description plus intuitif. Ce sujet est abordé dans la section suivante.

B.2.2 Transformations des diagrammes de classes aux langages formels et vice versa

Les modèles EMF s'apparentent fortement aux diagrammes de classe en UML (Unified Modeling Language) [45]. Plusieurs travaux ont tenté d'établir des transformations entre les méthodes formelles et UML, parmi elles celles présentées dans ce qui suit.

B To UML and Back Again B [1] est une méthode formelle qui permet de construire un programme par raffinements successifs, en utilisant les spécifications abstraites.

Dans sa thèse de doctorat [56], *A.Idani* à étudié la dérivation de diagrammes UML à partir de spécifications en B. Pour ce faire, il utilise un processus générique basé sur un ensemble de correspondances structurelles et sémantiques entre les méta-modèles de B et d'UML. Cette approche consiste en deux étapes. Une première transformation est effectuée pour traduire les parties statiques des spécifications B en diagrammes de classes UML. La deuxième étape permet d'explorer les parties dynamiques (comportementales) d'une spécification B pour la dérivation de diagrammes UML état/transition.

Dans [58], les auteurs se concentrent sur le côté opposé de la transformation (seulement pour les parties statiques). Ils proposent un cadre évolutif pour aider la dérivation des spécifications formelles en B à partir de diagrammes de classes UML. Ce travail est basé sur le mapping structurel et sémantique entre parties statiques de UML et B (établies dans la thèse) avec la définitions d'un noyau d'UML bien défini avec un ensemble de commandements sécurité (Safe-UML). A partir de ce modèle Safe-UML, un modèle formel d'une spécification B est construit. Un code certifiable peut alors être produit après une série d'améliorations et de preuves à l'aide de l'outil Atelier-B [14].

Cette approche semble être globale et complète, mais elle comprend quelques inconvénients. En effet, l'utilisateur est toujours sollicité afin de choisir les règles à appliquer

lors du processus de transformation. Par conséquent, il est possible de générer deux diagrammes UML différents à partir d'une seule description de B. Aussi, dans le deuxième sens de la transformation (de UML vers B), ce n'est pas la version UML habituelle qui est utilisée mais une version modifiée, Safe-UML.

UML To Alloy and Back Again Alloy [61] est un langage de modélisation textuel déclaratif basé sur la logique du premier ordre qui offre un support pour les notions de l'orienté objet. Ses modèles sont analysés avec un outil entièrement automatisé: Alloy Analyzer.

Dans sa thèse de doctorat [10], l'auteur a travaillé sur une approche de transformation basée sur l'IDM afin de générer automatiquement des spécifications en Alloy à partir de diagrammes de classes UML [11].

Le but de la démarche est d'effectuer une analyse en utilisant Alloy sur des modèles UML. La transformation est définie en utilisant un certain nombre de règles de transformation de concepts UML des concepts en Alloy. Par la suite, des travaux ont été menées [85] afin d'étendre ce travail en proposant un moyen de transformer les instances de modèles en Alloy vers des diagrammes objets en UML. En utilisant la trace de la transformation dans la première direction, le module convertisseur Alloy To UML traduit des contres exemples produits par l'analyseur Alloy en diagrammes d'objets en UML.

Cette approche offre la possibilité de transformer un diagramme de classes (en UML) en un modèle utilisable sur un analyseur automatique afin d'effectuer les preuves. Il est clair que l'utilisation d'un analyseur automatique est plus simple que d'utiliser un prouveur interactif (qui nécessite une expertise particulière de l'utilisateur). Cependant, les démonstrateurs interactifs sont plus puissants. En outre, cette approche n'offre aucun moyen de traduire un modèle en Alloy vers les diagrammes de classes UML, il le fait seulement au niveau de l'instance.

Focal To UML Focal [39] est un langage formel permettant de construire des applications certifiées progressivement à partir des spécifications abstraites jusqu'à arriver à des implémentations concrètes. Ce langage comprend des éléments inspirés de la programmation orientée objet tels que l'héritage.

Dans [36], *Delahaye & al.* décrivent un cadre formel et sûr permettant de transformer des spécifications en Focal vers des modèles UML. Ce travail vise à fournir une documentation graphique des modèles formels (décrit en Focal) pour les développeurs. Pour mettre en œuvre cette approche, les auteurs ont commencé par définir une description formelle pour un sous-ensemble du diagramme de classes UML. Ensuite, ce sous-ensemble UML a été étendu via le mécanisme des profils (fourni par UML) afin de prendre en compte les caractéristiques sémantiques de Focal. Les règles de transformation sont par la suite formellement présentées et utilisées pour établir la validité de la démarche. Cette preuve est également présentée dans le document cité.

Cette transformation est principalement utilisé comme documentation pour les développeurs du système prouvé. Cependant, il n'y a pas de transformation des diagrammes de classes UML en spécifications Focal. Ce serait un moyen pour les développeurs d'interagir avec l'expert de la preuve afin de convenir d'une solution qui conviendrait aux deux parties.

Ces méthodes permettent de générer des diagrammes de classes à partir de descriptions formelles et vice versa, mais leur représentation formelle est significativement différente de nos besoins : Les types de données fonctionnels utilisés dans les langages ML et les assistants de preuve.

B.3 Résumé

Dans ce chapitre, nous avons présenté les concepts de base de l'IDM ainsi que certains travaux connexes. Pour assurer la traduction de nos types de données vers les diagrammes de classes (et vice versa), nous utilisons une méthode basée sur l'IDM, plus particulièrement la transformation de modèles.

Pour mettre en œuvre notre transformation, nous utilisons une approche de manipulation directe (une des approches de transformation model-to-model). Ce choix a été fait après avoir étudié plusieurs approches et testé les outils qui les implémentent.

Notre travail se situe dans un contexte général de combinaison de méthodes formelles avec l'IDM. Afin d'effectuer cette combinaison, nous avons choisi de procéder par transformation des composants de l'une dans l'autre. Cette technique nous semble être la mieux adaptée pour tirer parti de ces deux framework dans le but de définir un Domain Specific Language.

À notre connaissance, notre étude est la seule à proposer une transformation bidirectionnelle entièrement automatique des types de données fonctionnelles vers méta-modèles (et vice versa).

Dans la partie suivante, nous introduisons les deux cadres qui représentent les entrées de nos transformation en commençant par le langage dans lequel sont définis nos méta-modèles : Eclipse Modeling Framework.

Appendix C

Eclipse Modeling Framework

Le travail que nous présentons dans cette thèse porte principalement sur des transformations entre des structures de données utilisées dans la programmation fonctionnelle et les diagrammes de classes. Pour représenter ces diagrammes de classes, il existe plusieurs langages où UML est le plus célèbre. Nous avons choisi de travailler avec Ecore, le langage de base d'Eclipse Modeling Framework (EMF) qui est proche de UML.

C.1 Eclipse Modeling Framework

Eclipse Modeling Framework (EMF) est un framework pour développer des applications basées sur définitions de modèles. En effet, il est possible de décrire un modèle dans un de ces formats et de générer dans les deux autres.

`Ecore` est le modèle qui est utilisé pour décrire et manipuler les modèles en EMF. `Ecore` est auto-descriptif et a été développé comme une implementation simplifiée des diagrammes de classes d'UML. Nous présentons ici brièvement les principales classes composant de `Ecore`.

- Le `EPackage` est l'élément racine dans modèles `Ecore` sérialisés. Il permet de regrouper les `EClass`es et `EDataType`s.
- La composante `EClass` représente les classes `Ecore`. Elle décrit la structure des objets. Elle peut contenir des `EAttribute`s, `EReference`s, ainsi que des `EOperation`s.
- Un `EDataType` représente les types de `EAttribute`s, qu'ils soient pré-définis (types: entier, booléen, flottant, ... etc.) ou encore définis par l'utilisateur.
- Les `EReference`s sont comparables aux liens d'association d'UML. Elles définissent les types d'objets qui peuvent être liés ensemble. La composition est un type de références plus fort où l'attribut booléen `containment` prend la valeur true.

C.2 Sous-ensemble de EMF Utilisé dans cette thèse

Le sous-ensemble du langage `Ecore` que nous avons utilisé dans notre travail contient essentiellement les éléments dont nous avions besoin pour la traduction de/vers les types de données fonctionnels. Dans ce méta-modèle apparaissent uniquement les classes et opérations de base permettant de garder la puissance expressive de Ecore. Il est également important de noter que ce sous-ensemble nous permet de définir des modèles qui sont validées par Ecore.

Dans Figure C.1, les constructions qui permettent d'exprimer la généricité sont distingables des autres par leur couleur verte. Aussi, il y est simple de reconnaître les éléments les plus importants: `EClass`, `EDataType`, `EReference`, `EAttribute` ... etc. Ce sous-ensemble contient les principaux éléments permettant de construire un méta-modèle en EMF. Certains éléments optionnels n'y sont pas inclus. Notez que le `EClass` `ENamedElement` n'a pas d'intérêt particulier, elle sert principalement à définir le `EAttribute` `name` pour les classes qui en héritent.

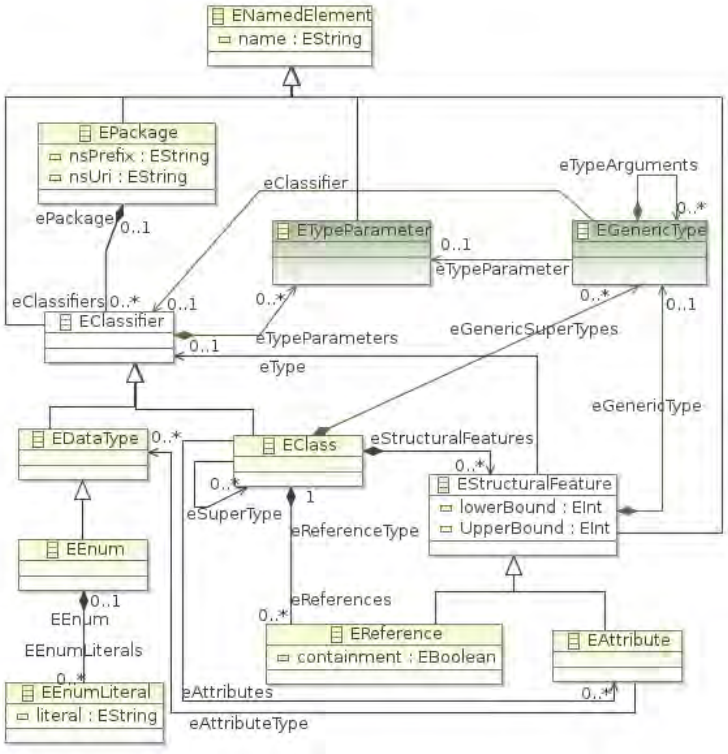


Figure C.1: Sous-ensemble simplifié du méta-modèle de Ecore

C.3 Résumé

Dans ce chapitre, nous avons présenté une définition de Eclipse Modeling Framework, puis, nous avons sélectionné les parties utilisées dans notre processus de transformation. Ce méta-modèle est le point d'entrée de la première direction de notre processus de transformation (voir chapitre 4). À présent, nous définissons l'autre partie essentiel au processus de transformation, le cadre formel: la programmation fonctionnelle.

Appendix D

Framework Formel

D.1 Programmation fonctionnelle

La programmation fonctionnelle est un paradigme de programmation basé sur la notion mathématique de *fonction*. Elle implémente le λ-calcul : un langage formel de la logique mathématique qui formalise les systèmes à travers la notion de fonction. Une fonction, dans la programmation fonctionnelle, consiste en la mise en correspondance des éléments d'un ensemble à un autre. Ces ensembles sont appelés *types*. Habituellement, ils donnent une indication sur la correction des programmes. Nous pouvons compter parmi les langages qui implémentent la programmation fonctionnelle : *Lisp* [89], *Haskell* [81], ainsi que les *langages ML* comme OCaml (Objective Caml) [68].

D.1.1 Les prouveurs interactifs

Un prouveur interactif est un outil servant à assister l'utilisateur à effectuer une preuve formelle d'un système. La preuve vérifie si le programme informatique répond à certaines spécifications dans une logique formelle. Il peut également vérifier la nature des relations entre les composants (vérifier si un élément raffine l'autre, par exemple). Ces caractéristiques garantissent la correction de logiciel qui est une exigence importante pour les programmes critiques.

Isabelle [73] and *Coq* [29] font partie des systèmes de vérification et de spécification les plus connus. Ces assistants de preuve sont implantés dans un langage ML et consistent en une combinaison de *programmation fonctionnelle* et de *logique mathématique* .

D.2 Sous-ensemble des langages fonctionnels utilisé dans cette thèse

Une déclaration de types en programmation fonctionnelle consiste en une séquence de définitions de types de données. Pour le moment, il n'est pas possible de prendre en compte toutes les composantes syntaxiques qu'offre la programmation fonctionnelle, entre autres parce que Ecore n'offre pas de fonctionnalités correspondante, par exemple des constructeurs de type d'ordre supérieur. C'est pour cela que nous avons basé notre travail sur le sous ensemble décrit par la grammaire présentée en Figure D.1. Les types de données y sont regroupés en *Module*s. Les types de données sont soit des types variant soit des enregistrements (records). Chaque type variant est composé d'une ou plusieurs déclarations de constructeurs. Ces déclarations sont formées d'un nom de constructeur et d'une successions de paramètres. Les paramètres peuvent se présenter sous forme de types primitifs ou de types déjà définis précédemment. Notons aussi que nous avons pris en compte dans ce sous ensemble la possibilité de définir les types polymorphes, donc des types avec des paramètres de type (type-param dans la grammaire de la Figure D.1).

La notation `type option` permet de typer un élément qui peut être présent ou absent. Elle retourne `None` s'il est absent et `Some x` dans le cas contraire. Les références `ref` quant à elles servent à représenter les pointeurs.

Nous ajoutons à cette syntaxe un moyen pour l'utilisateur de différencier les énumérations des autres définitions de type de données. Cette option est présentée en introduisant le commentaire `(**Enumerated*)` dans le code. Ceci concerne les types variant ayant déclarations de constructeur sans expressions de type.

Cette grammaire est enrichie avec de nouveaux éléments que nous avons nommé accesseurs. Ce sont des fonctions que l'on introduit par l'annotation `(*@accessor*)`. Elles permettent d'attribuer des noms à des champs particuliers dans une déclaration de type, comme il apparaît dans la définition présentée en Figure D.2.

D.3 Résumé

Ce chapitre présente le cadre formel que nous traitons dans cette thèse : *les langages fonctionnels*. Nous avons commencé par présenter le cadre général de la programmation fonctionnelle, nous avons ensuite montré la relation entre la programmation fonctionnelle et les prouveurs interactifs. Ultérieurement, nous avons défini le sous-ensemble sur lequel nous basons notre travail dans cette thèse. Maintenant, nous allons définir les règles de transformations qui permettent de traduire des types de données fonctionnels en métamodèles.

module ::= `Module` *module-name* {*type-definition*}
type-definition ::= *typeDef*
| *typeEnum*
typeEnum ::= `(**Enumerated*)` `type` *typeconstr-name* `=`*constr-name*
{ | *constr-name* }
typeDef ::= `type` [*type-params*] *typeconstr-name* `=` *type-representation*
type-representation ::= *variant-type-rep*
| *record-rep*
variant-type-rep ::= *constr-decl* { | *constr-decl* }
record-rep ::= `{` *field-decl* {`;` *field-decl*} `}`
constr-decl ::= *constr-name*
| *constr-name* `of` *comp-typexpr* {`*` *comp-typexpr*}
field-decl ::= *field-name* : *comp-typexpr*
comp-typexpr ::= *typexpr*
| (*typexpr* `option`)
| (*typexpr* `list`)
| (*typexpr* `ref`)
type-params ::= `(`*type-param* { , *type-param* }`)`
typexpr ::= *type-params typeconstr-name*
| *typeconstr-name*
| *type-param*
| *prim-type*
prim-type ::= `int` | `float` | `bool` | `string`
type-param ::= `'`*lowercase-ident*
module-name ::= *capitalized-ident*
typeconstr-name ::= *lowercase-ident*
field-name ::= *lowercase-ident*
constr-name ::= *capitalized-ident*
lowercase-ident ::= (`a...z`){`a...z`|`A...Z`|`0...9`}
capitalized-ident ::= (`A...Z`){`a...z`|`A...Z`|`0...9`}

Figure D.1: Grammaire de types de données utilisées dans cette thèse (en Caml)

```
(*@ accessor *)
```
`let` acc_name_i (*constr-name* $(x_1, ..., x_n)$) = x_i / $1 \leq i \leq n$

Figure D.2: Syntaxe des fonctions d'accesseur (en Caml)

Appendix E

Des modèles fonctionnels vers EMF

Dans ce chapitre est détaillée la transformation automatique des types de données fonctionnels en méta-modèles. Tout d'abord, nous commençons par définir les contraintes permettant de garantir un bon résultat après transformations, ensuite nous présentons les règles de transformation et détaillons certaines d'entre elles. Nous veillons à donner un exemple pour règle détaillée.

E.1 Conditions de bonne formation pour les types de données en entrée

Dans le chapitre précédent, nous avons défini un sous-ensemble du langage Caml auquel peuvent être appliquées nos règles de transformation. Afin de garantir la correction des éléments transformés, nous devons imposer certaines contraintes de bonne-formation sur le sous-ensemble en entrée. La contrainte principale consiste à observer les contraintes de bonne-formation des programmes Caml. En effet, chaque composant source ne doit pas générer d'erreur lors de son évaluation par l'interpréteur Caml. Parmi toutes ces contraintes, celles qui nous intéressent sont:

- L'ordre d'apparition des définitions des types de données: Tout type de données utilisé pour définir un autre type de données doit comparaître devant lui.
- L'utilisation du même nombre de paramètres de types (dans les expressions de type) que ceux définis dans les paramètres du constructeur de type.
- Utiliser uniquement des paramètres de type (dans les expressions de type) qui sont définies sur les paramètres du constructeur de type.

E.2 Règles de Transformations

La fonction de traduction *Tr()* prend des structures de données de datatypes en entrée, et produit en sortie un méta-modèle Ecore correspondant. Les règles de transformation sont présentées comme sous-fonctions, pour les différents composants des types de données définis sur la grammaire.

$$Tr : DataTypes \longrightarrow Ecore\ Meta\text{-}model$$

Dans cette partie du document il ne nous est pas possible de présenter toutes les règles de transformation à titre d'exemple nous vous en proposons trois présentées dans les prochaines sous-sections.

E.2.1 Règle DatatypeToEClasses

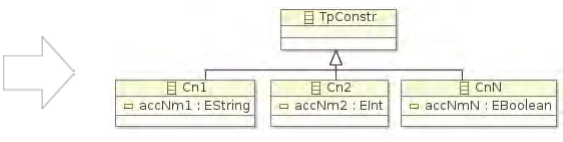
Cette règle est appliquée dans le cas où le type de données est un type variant avec plusieurs déclarations de constructeurs composées de noms de constructeurs et de types d'expressions.

Tout d'abord, une première `EClass` pour représenter le type de base (*tpConstr*) est créé. Ensuite, pour toute déclaration de constructeur, une seconde `EClass` qui hérite de la première est créée.

$$\begin{array}{lcl} Tr_{dtp}(tpConstr = cd_1|...|cd_n) & = & createEClass(); \\ & & setName(tpConstr); \\ & & Tr_{decl}(cd_i, tpConstr) \\ & & / \ 1 \leq i \leq n \\ Tr_{decl} : ConstructorDeclaration & \longrightarrow & EClass \\ Tr_{decl}(cn_i\ t_1...t_m, tpConstr) & = & createEClass(); \\ & & setName(cn_i); \\ & & setSuperType\ (EClass(tpConstr)); \\ & & Tr_{type}(acc_j, t_j) \\ & & / \ 1 \leq j \leq m \end{array}$$

Exemple:

```
type tpConstr =
Cn1 of string
|Cn2 of int
|...
|CnN of bool
```



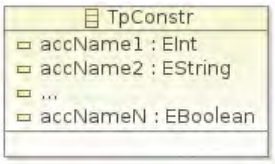
E.2.2 Règle PrimitiveTypeToEAttribute

Concernant la traduction des expressions de types, plusieurs cas sont à prendre en compte. Commençons par le plus simple, lorsque le type y est primitif. La traduction consiste à créer un `EAttribute` dont le type est le type Ecore correspondant et dont le nom est le nom de l'accesseur à cette valeur.

$$\begin{array}{lcl} Tr_{type} : (accessor, type) & \longrightarrow & EStructualFeature \\ Tr_{type}(asc, primTp) & = & createEAtrribute(); \\ & & setName(acc); \\ & & setType(primTp_{EMF}); \end{array}$$

Exemple:

`type` *tpConstr* `=`
Cn `of int * string * ...*`
`bool`



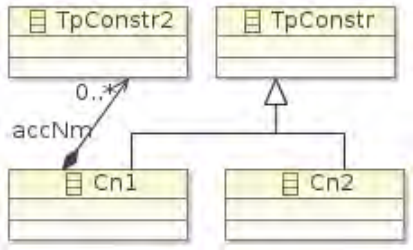
E.2.3 Règle ListToMultiplicity

Les expressions de types peuvent aussi apparaître de la forme d'une liste de type : *type* `list`. La traduction se fait par modification des cardinalités des `EStructuralFeature`s en `0...*`.

$$\begin{array}{lcl} Tr_{type}(asc, t\ \texttt{list}) & = & Tr_{type}(asc, t) \\ & & setLowerBound(0); \\ & & setUpperBound(*); \end{array}$$

Example:

`datatype` *tpConstr*`=`
Cn$_1$ `of` *tpConstr2* `list`
| *Cn*$_2$



E.3 Résumé

Ce chapitre porte principalement sur la présentation détaillée des règles de transformation permettant de produire des méta-modèles à partir d'une description du type de données utilisé dans la programmation fonctionnelle. Nous avons pris en compte tous les cas possibles qui pourraient apparaître sur notre grammaire en entrée, et illustré les règles de transformation par des exemples explicites. Dans le chapitre suivant, nous allons nous concentrer sur le sens opposé de la transformation.

Appendix F

De EMF vers les modèles fonctionnels

Dans ce chapitre, nous présentons le deuxième sens de la traduction: des méta-modèles vers les structures de données utilisées dans la programmation fonctionnelle. Nous commençons par définir quelques conditions de bonne formation sur le méta-modèle d'entrée. Ensuite, nous détaillons quelques règles de transformation.

F.1 Contraintes de bonne formation pour le méta-modèle source

Pour effectuer le sens inverse de la transformation, nous nous basons fortement sur le mapping réalisé lors de la précédente transformation (chapitre 4). À notre avis, il est important de mettre en place une fonction qui est l'inverse de celle permettant de passer des type de données aux méta-modèles. En effet, la possibilité de composer les deux fonctions, les appliquer sur un modèle et de trouver un modèle équivalent est primordiale, même si cela nous amène à poser certaines restrictions sur la méta-modèle source.

La première restriction concerne la profondeur des relations d'héritage : la transformation d'un méta-modèle contenant une hiérarchie de classes sur plus d'un niveau (une classe qui hérite d'une classe qui hérite d'une autre une, ...etc) ne sont pas pris en charge par nos règles.

La seconde restriction vise à éviter les types de données interdépendants. Nous définissons donc un ordre partiel *prec* sur les classes pour la transformation de `EClassifiers` contenus dans un `EPackage`. Les `EEnums` doivent être traduits en premier parce qu'ils ne dépendent pas d'autres éléments. Les autres `EClasses` contenues dans le `EPackage` doivent ensuite être ordonnées à l'aide de deux critères :

- **La relation d'héritage :** si une `EClass` C_1 est un `superType` d'une autre `EClass`

C_2, alors C_1 doit être traduite avant C_2. Nous ajoutons donc la contrainte $C_1 \prec C_2$.

- **La relation `EReference` :** si une `EClass` C_1 est la cible (`eType` en Ecore) d'une `EReference` appartenant à une autre `EClass` C_2, alors C_1 doit être traduite avant C_2, par conséquent $C_1 \prec C_2$.

Cette commande permet de définir le deuxième critère de bonne formation : l'ordre $\prec$ générés par les deux contraintes ci-dessus doit être acyclique.

La dernière contrainte que nous imposons sur les modèles porte sur l'héritage et la généricité. En effet, si nous avons une relation d'héritage entre deux types génériques (représentés par des `EClass`es avec `ETypeParameter`s) tous les paramètres utilisés par une classe fille doivent apparaître dans le super-classe.

F.2 Règles de transformation

Comme dans le chapitre précédent, les règles de transformation sont présentées en langage naturel accompagnés d'une notation formelle. Pour éviter de surcharger la notation, nous utilisons encore une fois la notation $Tr()$ pour représenter la fonction de traduction (au lieu d'écrire $Tr()^{-1}$). La méthode de transformation $Tr()$ produit alors des descriptions de types de données (rassemblées dans des *modules*) à partir d'un méta-modèle.

F.2.1 Règle EEnumToDatatype

Les énumération sont représentées dans Ecore par des `EEnum`s. Ils sont transformés selon notre algorithme en définitions de types de données composées de *déclarations de constructeurs* sans expressions de types. Chaque `EEnumLiteral` de l'énumération donne un *nom de constructeur*.

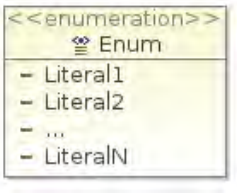
$$
\begin{aligned}
Tr_{Cl}(&\texttt{eEnum name} = e \\
&\{ELit_1 \ldots ELit_n\}) = \quad createDatatype(); \\
&\qquad\qquad\qquad\quad NewTp\_Constr(); \\
&\qquad\qquad\qquad\quad setName(e); \\
&\qquad\qquad\qquad\quad Tr_{Lit}(ELit_i); \qquad / \ 1 \leq i \leq n
\end{aligned}
$$

$$
\begin{aligned}
Tr_{Lit}(\texttt{literal} = l) = \quad & createConstructor(); \\
& setName(l);
\end{aligned}
$$

F.2.2 Règle EAttributeToType

La transformation de chaque `EAttribute` consiste à créer une nouvelle *expression de type* dans la *déclaration de constructeur* correspondante. Les *définitions de type* correspondantes peuvent être sélectionnées par leurs noms dans la liste des les types de données déjà créés.

Exemple:



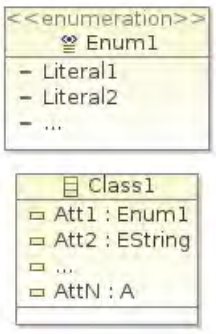
`type` *enum*`=`
Literal1
|*Literal2*
|...
|*LiteralN*

L'*expression de type* créée est composée de la traduction du type du `EAttribute` et de ses bornes dans le langage fonctionnel. Ce processus se fait en utilisant la fonction de transformation Tr_{type}. Pour traduire les bornes supérieures et inférieures, la fonction Tr_{Bnd} est appelée.

$$\begin{aligned} Tr_{SF}(&\texttt{eAttribute name} = a \\ &\quad LowerBound \\ &\quad UpperBound \\ &\quad EType) = \\ &createTypeExpression(Tr_{Type}(EType)); \\ &Tr_{Bnd}(LowerBound, UpperBound); \end{aligned}$$

$$\begin{aligned} Tr_{Type}(\texttt{eType} = \texttt{EInt}) &= \texttt{int} \\ Tr_{Type}(\texttt{eType} = \texttt{EBoolean}) &= \texttt{bool} \\ Tr_{Type}(\texttt{eType} = \texttt{EFloat}) &= \texttt{float} \\ Tr_{Type}(\texttt{eType} = \texttt{EString}) &= \texttt{string} \\ Tr_{Type}(\texttt{eType} = \texttt{eenum e}) &= e \end{aligned}$$

Exemple:



`type` *enum*`=`
Literal1|*Literal2* |...

`type` *'a class1* `=`
Class1 `of` *enum1* ∗ `string` ∗
`...`∗ *'a*

F.3 Règle MultiplicitiesToTypeOptions

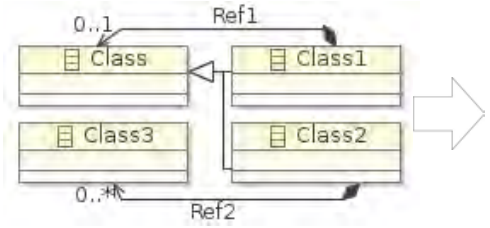
Cette règle permet de traduire les valeurs de multiplicités contenues dans les structural features (représentées par des bornes supérieures et inférieures). Ces multiplicités sont utilisées pour déterminer le nombre d'éléments qui composent une instance.

Lorsque la borne inferieure (`lowerBound`) prend la valeur `0` et que la borne supérieure (`upperBound`) prend la valeur `1`, cela signifie que cette `EStructuralFeature` peut être présente ou absente (lors de l'instanciation). Ces valeurs sont converties au type `option` dans les *expressions de type*.

Dans le cas où la `upperBound` est prend la valeur `*`, elle est traduite vers le type *list* dans la description du type de données.

$$Tr_{Bnd}(\texttt{lowerBound="0"}, \texttt{upperBound="1"}) = \texttt{option}$$
$$Tr_{Bnd}(\texttt{lowerBound="0"}, \texttt{upperBound="*"}) = \texttt{list}$$
$$Tr_{Bnd}(\texttt{lowerBound="1"}, \texttt{upperBound="*"}) = \texttt{list}$$
$$Tr_{Bnd}(\texttt{lowerBound="1"}, \texttt{upperBound="1"}) = \emptyset$$

Exemple:



`type` *class* `=`
Class1 `of` *class* `option`
|*Class2* `of` *class3* `list`

F.4 Résumé

Nous avons défini dans ce chapitre comment nous effectuons la traduction de modèles EMF en types de données utilisés dans les langages fonctionnels. Nous avons d'abord présenté quelques restrictions sur la forme des méta-modèles fournis en entrée, afin d'éviter l'apparition d'incohérences sur les types de données traduits. Ensuite, nous avons présenté quelques règles de transformation. Ces règles ont été illustrées par des exemples simples.

La partie suivante concerne l'illustration de notre approche en l'appliquant à différentes études de cas, en commençant par les diagrammes de décision binaires dans chapitre 6.

Appendix G

Diagrammes de décision binaires

Dans ce chapitre, nous présentons une étude de cas simple afin d'illustrer notre approche. Nous allons, tout d'abord présenter l'étude de cas choisie: La construction de diagrammes de décision binaires. En suite, nous expliquerons étape par étape comment nous avons mis en oeuvre cette etude de cas.

G.1 Diagrammes de décision binaires

Les diagrammes de décision binaires (BDD, Binary Decision Diagrams) [26] sont des arbres binaires dont les feuilles sont des valeurs Booléennes. Ils permettent de représenter les formules Booléennes de façon compacte afin de faciliter la vérification de leur correction. Leur spécificité réside dans leur capacité à joindre les sous-arbres communs et ainsi d'éliminer les redondances. Ils sont fréquemment utilisés dans le Model Checking ainsi que dans le développement de circuits numériques.

Des travaux antérieurs [49] ont été réalisés pour la vérification de la correction d'un algorithme qui construit un BDD à partir d'une expression Booléennes en utilisant l'assistant de preuve Isabelle [73]. L'essence de la construction du BDD est contenu dans la fonction *Build()*. Nous avons utilisé ces théories prouvées en Isabelle pour générer le code orienté objet correspondant en Scala, grâce au mécanisme de génération fourni par Isabelle. Nous avons, par la suite, utilisé ces mêmes théories pour générer notre méta-modèle en Ecore.

G.2 Mise en place de l'étude de cas

G.2.1 Présentation de l'étude de cas

Le point d'entrée, de l'étude de cas, est représenté par des *théorie*s Isabelle définissant les types de données ainsi que les fonctions. A partir de ces théories, nous générons en utilisant notre fonction de traduction le méta-modèle correspondant. Aussi, nous obtenons le code

Scala correspondant aux types de données et fonctions. À partir du méta-modèle Ecore généré, nous utilisons l'outil Xtext pour génerer du code pour un éditeur textuel pour les Formules Booléennes et l' outil GMF pour génerer du code pour un éditeur graphique permettant afficher les BDDs.

Le résultat souhaité est de pouvoir construire, en utilisant la fonction *Build()* générée en Scala, un BDD à partir d'une formule logique écrite dans un éditeur textuel et de l'afficher le BDD résulatant dans un éditeur graphique. A cet effet, nous avons effectué une correspondance entre les éléments du code générés par Xtext (resp. GMF) et les éléments du code générés en Scala (*c. f.* Figure G.1).

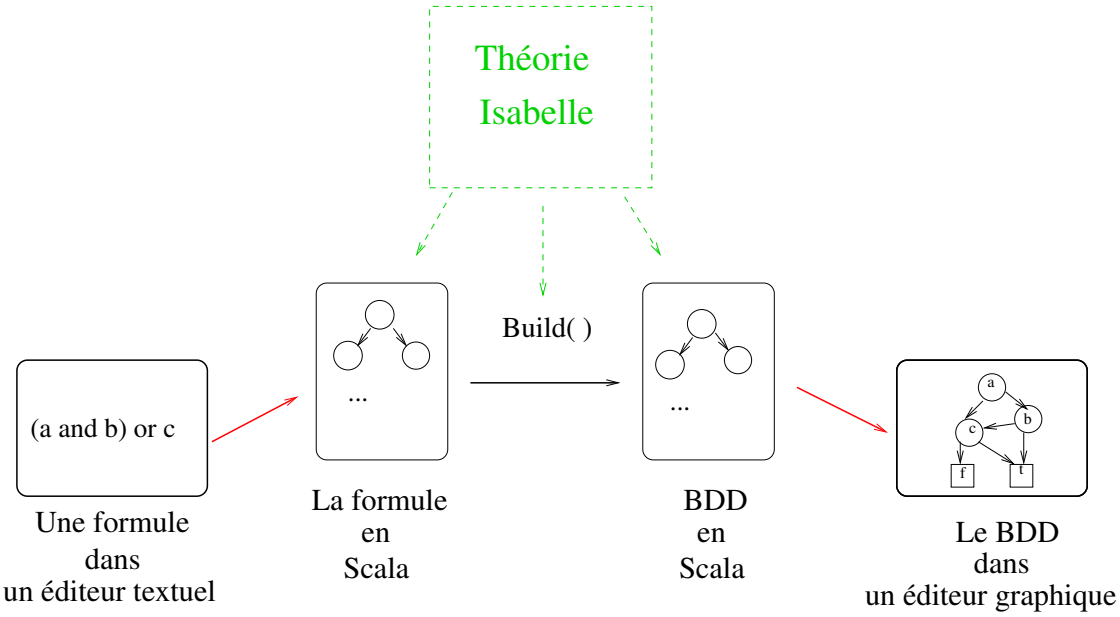


Figure G.1: Exécution de l'étude de cas

G.2.2 Génération de diagrammes Ecore à partir de types de données

Figure G.2 (Resp.Figure G.4) montre un type de données en Isabelle sur lequel ont été mises en place les vérifications ainsi que les fonctions d'accesseurs correspondantes. Cette partie de la théorie Isabelle a été donnée en entrée à notre fonction de traduction, présentée dans le chapitre 4. Le diagramme Ecore résultant est présenté dans la Figure G.3 (Resp.Figure G.5).

$$
\begin{array}{lrl}
(**Enumerated*) & datatype\ bbinop & = OR \mid AND \mid IMP \mid IFF \\
 & datatype\ bexpr & = BVar\ string \\
 & & \mid BConst\ bool \\
 & & \mid BBExpr\ bbinop\ bexpr\ bexpr \\
(*@accessor*) & fun\ var & :: bexpr\ =>\ string \\
 & where & var(BVar\ x\ ) = x \\
\ldots & &
\end{array}
$$

Figure G.2: Type de données correspondant à la formule logique et ses accésseurs en Isabelle

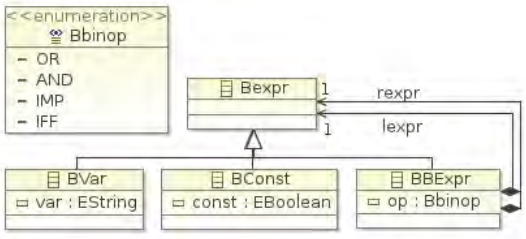


Figure G.3: Résultat de la traduction du type de donnée pour les formules logiques

$$
\begin{array}{lrl}
 & datatype\ bddTree = & Node\ string\ bddTree\ bddTree \\
 & & \mid Leaf\ bool \\
(*@accessor*) & fun\ nodeVal & :: bddTree\ =>\ string \\
 & where & nodeVal(Node\ x\ \_\ \_) = x \\
\ldots & &
\end{array}
$$

Figure G.4: Type de données correspondant à la définition de BDD et ses accésseurs en Isabelle

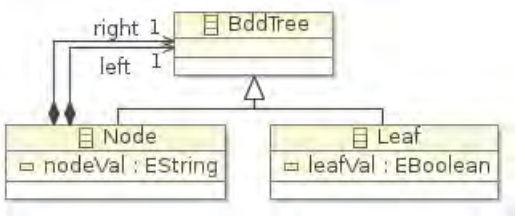


Figure G.5: Résultat de la traduction du type de donnée pour les BDDs

G.3 Résumé

En somme, ce chapitre est présenté dans le but d'illustrer par une étude de cas, notre technique de transformation permettant de générer des méta-modèles à partir d'une description de type de données en Isabelle. Nous avons commencé par une description de notre étude de cas consistant à utiliser un programme vérifié pour construire un BDD. Nous avons ensuite appliqué notre méthode de transformation et utilisé les méta-modèles générés pour créer un éditeur graphique et un autre textuel. Ainsi, nous avons montré une voie possible d'utiliser des méta-modèles générés en Ecore.

Appendix H

Safety Critical Java

Dans le chapitre précédent, nous avons choisi de tester notre approche sur une étude de cas avec des types de données simples. Cette étude de cas est très utile pour montrer les différentes solutions possibles pour l'utilisation de notre méta-modèle généré. Elle révèle également les compositions possibles avec la génération de code orienté objet en Scala. Le but de ce chapitre est différent. En effet, cette étude de cas consiste en la transformation pour types de données qui définissent un Domain Specific Language. Cet exemple est un exercice suffisamment complexe pour démontrer l'efficacité de notre approche.

Dans ce chapitre, nous commençons par définir ce DSL. En suite, nous expliquons la mise en œuvre de l'étude de cas, avant de terminer avec les résultats effectifs de la transformation des types de données définissant le DSL en utilisant notre méthode de transformation.

H.1 Définition de Safety Chritical java

C'est un langage de type Java enrichi avec des assertions (annotations temporelles). Il représente un dialecte temps réel du langage Java qui permet d'effectuer des analyses statiques spécifiques de programme Java (les détails sont décrits dans [17]). Nous présentons les éléments qui composent le langage se référant à la description des types de données présentés dans Figures 7.3 et 7.4. Ce langages permet de définir : des classes ainsi que des programmes et des méthodes qui peuvent contenir des opérateurs binaires, des expressions, des statements et des déclarations.

H.2 Présentation de l'étude de cas

L'élément de base est une *théorie* Isabelle où les types de données, les fonctions et les preuves sont définies. Le méta-modèle correspondant est généré en utilisant la fonction de traduction décrite dans le chapitre 4. Comme ce fut le cas dans le chapitre 6, nous utilisons

l'outil Xtext de définir une syntaxe textuelle concrète à partir de la `Ecore` méta-modèle généré.

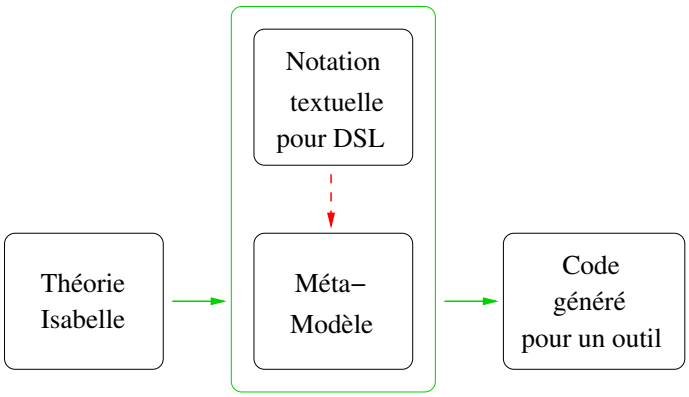


Figure H.1: Architecture de l'implementation de al fonction Datatype to Ecore

H.3 Génération d'un diagramme Ecore à partir de types de données

Figures 7.3 and 7.4 montrent les définitions de types de données provenant de la *théorie* Isabelle. Figure 7.5 représente quant à elle des exemples des fonctions d'accesseur correspondantes. Ces parties de la *théorie* Isabelle ont été données en entrée à la fonction implementant nos règles de transformation présentées dans le chapitre 4. Le diagramme `Ecore` résultant est présenté dans Figure 7.6. Nous avons soumis ce méta-modèle généré à l'outil de validation fournis par EMF. Le méta-modèle a été validé avec succès. Lorsque nous appliquons la transformation inverse (à partir des méta-modèles de types de données) sur ce méta-modèle généré, le résultat donne une même description de type de données que celle utilisée pour générer le méta-modèle dans la première transformation.

H.4 Résumé

Dans cette deuxième étude de cas, nous avons évalué l'efficacité de notre approche de transformation sur une théorie Isabelle concrète. Les types de données et les fonctions d'accesseur sont décrits pour le DSL Safety Critical Java [17]. Ce DSL est un langage de type Java enrichi avec des assertions. Le méta-modèle généré (pour le DSL) peut facilement être utilisé pour générer un éditeur textuel en utilisant des outils comme Xtext. Ce méta-modèle respecte les conditions de bonne formation fixées par Ecore et a été validé avec succès en utilisant l'outil de validation fourni par Ecore (sur Eclipse).

Appendix I

Conclusion

Résumé de la thèse

Dans cette thèse, nous avons présenté un processus de transformation entièrement automatisée basée sur l'IDM permettant de passer des types de données fonctionnels vers des diagrammes de classes et vice versa. Ce travail constitue un premier pas vers la combinaison de preuve interactive et ingénierie des modèles. Il est dédié à la simplification de la communication entre les professionnels de preuve formelle et des experts industriels, dans le cadre du développement formel de logiciels critiques.

Pour atteindre cet objectif, nous avons défini un premier sous-ensemble de types de données à partir d'une grammaire EBNF décrivant des types de données. Ce sous-ensemble contient les éléments essentiels nécessaires pour décrire les définitions de types de données de base y compris les types paramétrés. Nous avons également conçu une syntaxe particulière permettant de décrire des fonctions accesseurs qui sont aussi utilisés lors de la transformation. De cette grammaire nous avons construit un méta-modèle pour la description des types de données qui constitue une partie essentielle d'un processus de transformation basée sur l'IDM.

Après cela, nous avons présenté un sous-ensemble du méta-modèle Ecore qui est suffisamment expressif pour modéliser les diagrammes de classes de base (y compris les types génériques). Il constitue une autre partie nécessaire pour effectuer la transformation.

Ensuite, nous avons défini un ensemble de règles de transformation pour le premier côté de la transformation: Des types de données vers les méta-modèles. Nous nous sommes assuré d'avoir couvert intégralement notre grammaire de types de données par les règles de transformation afin d'éviter d'avoir des constructions non traduites.

Sur la base de cette transformation, nous avons dérivé le deuxième sens de la transformation: à partir des méta-modèles vers les types de données. Pour ce côté-ci de la transformation, nous avons dû définir des conditions de bonne formation pour garantir l'exactitude des types de données générés. Ces conditions sont également utilisés pour faire en sorte

que la composition de notre transformation donne l'identité si nous commençons à partir du modèle formel.

La plupart des règles de transformation décrites dans cette thèse ont été mis en œuvre en Java sous la plate-forme Eclipse. Pour représenter les méta-modèles, nous avons utilisé l'Eclipse Modeling Framework et le plug-in Ecore. Nous avons également développé des analyseurs syntaxiques pour notre sous-ensemble de types de données pour chacun des langages Caml et Isabelle. Nous avons utilisé cet environnement afin de tester l'approche avec les petits exemples ainsi que deux études de cas principales décrites ci-après.

La première consiste en la construction vérifiée d'un diagramme de décision binaire (BDD) avec partage de sous-arbres. Dans cette étude de cas, nous avons généré deux méta-modèles à partir de descriptions de types de données. De plus, nous avons utilisé ces méta-modèles pour définir un éditeur textuel (pour décrire les formules Booléennes) et éditeur graphique (pour décrire les BDDs). Nous avons également couplé nos travaux avec la génération de code orienté objet à partir de théories Isabelle.

La seconde représente un Domain Specific Language: Safety Critical Java. C'est un langage de type Java enrichi avec annotations. Cette étude de cas constitue un exemple d'application qui est assez complexe pour montrer l'utilisabilité des notre approche. A partir de définitions de types de données, mis en place pour la modélisation sémantique du DSL, nous avons été en mesure de générer un méta-modèle EMF. Le méta-modèle généré est utilisé pour la documentation et la visualisation du DSL. Il peut également être manipulé dans le workbench d'Eclipse pour générer un éditeur textuel en tant que plug-in Eclipse.

Travaux en cours

Outre les tâches précédemment accomplies, d'autres travaux sont encore en cours de développement. Ils sont présentés dans les paragraphes suivants:

Propriétés de nos règles de transformation

Nous travaillons actuellement sur la validation de deux propriétés sur nos règles de transformation : la réflexivité et la bidirectionnalité.

Après la mise en œuvre des règles de transformation de base, nous avons mis à l'épreuve notre approche en appliquant notre processus de transformation à notre propre sous-ensemble Ecore. Le but derrière une telle action est de valider notre approche en lui donnant une propriété de réflexivité. Le résultat escompté est d'être en mesure de générer en Caml une description du type de données pour Ecore, qui est proche de sa représentation sous forme de grammaire (donné dans cette thèse dans le chapitre 2).

L'autre propriété sur laquelle nous travaillons est la bidirectionnalité. En d'autres termes, il s'agit d'évaluer si la composition de nos deux fonctions de transformation donne l'identité. Tel est le cas lorsque le modèle source est un modèle formel. Nous avons détecté

que cette propriété lors de l'expérimentation nos transformations sur différents exemples. En effet lorsque nous appliquons notre fonction de transformation ($f()$) sur un modèle de source (M_s) des types de données, nous générons un diagramme de classes M_T. Si nous appliquons à ce diagramme généré M_T la fonction de transformation dans la direction opposée ($f'()$), on obtient un modèle qui est le même que M_S (le modèle de source de la première direction de la transformation).

Au contraire, si nous commençons par générer un modèle de type de données à partir d'un diagramme de classes (en utilisant ($f'()$)) et nous appliquons $f()$ pour le modèle qui en résulte, ce modèle peut être différent du modèle source. Après avoir analysé nos règles de transformation nous avons réussi à détecter l'une des règles qui est responsable de cela. Cette règle est utilisée pour traduire un pattern qui n'apparaît jamais dans les diagrammes de classes générées.

Implementation du processus de transformation en tant que plug-in Eclipse

La plupart des règles de transformation présentés dans cette thèse ont été implementées avec succès en Java en utilisant EMF comme représentation sous-jacente de modèles. Cette implémentation a été utilisée pour illustrer notre approche des études de cas. Toutefois, les règles de transformation qui ont été définies au cours d'une deuxième phase n'ont pas encore été implentées. C'est sur quoi nous travaillons actuellement. Ensuite, nous avons l'intention de rassembler les règles sous la forme d'un plug-in Eclipse. Dans un tel contexte, il serait disponible et accessible aux parties intéressées.

Perspectives

Plusieurs possibilités peuvent être envisagées pour compléter ce travail:

L'extension du sous-ensemble de types de données

Dans cette thèse, il n'était pas possible de prendre en compte toutes les constructions fournies par les langages fonctionnels. Nous avons travaillé sur un sous-ensemble des types de données présentées dans le chapitre 3. Même si nous pensons que notre sous-ensemble est suffisamment expressif pour construire des types de données de base, il pourrait être étendu à d'autres constructions dans les travaux futurs. Cette extension peut être réalisée en ajoutant une traduction pour les types de données mutuellement récursifs.

L'extension du sous-ensemble de Ecore

Dans le chapitre 2, nous avons décrit un sous-ensemble de Ecore permettant de définir les diagrammes de classes de base. Bien que ce sous-ensemble contient un ensemble assez riche de composants pour représenter la majorité des diagrammes Ecore, il serait souhaitable de

l'étendre afin de proposer des transformations pour d'autres composants, en particulier `EAnnotation`s et `EOperation`s.

En outre, nous voudrions réduire les contraintes de forme imposées à l'utilisateur afin d'augmenter le nombre de motifs traduisibles. Cela pourrait se faire par exemple en permettant à l'utilisateur de traduire des hiérarchies des classes plus profondes.

Mise en place de preuves sur les transformations

Les transformations décrites dans les chapitres 4 et 5 ont été élaborées dans un langage mathématique (de manière fonctionnelle comme si nous les avions mises en place en Caml). Il serait intéressant de prouver certaines propriétés de ces transformations en effectuant une preuve manuelle ou mécanisée. Un tel processus exige d'attribuer tout d'abord une sémantique formelle aux deux parties de la transformation: les sous-ensembles de diagrammes de classes et types de données fonctionnelles. Ensuite, pour définir les propriétés à vérifier par exemple la préservation de la sémantique après avoir effectué la transformation.

Transformer la partie dynamique des langages fonctionnels

Dans la partie II, nous avons défini les transformations de et vers la partie structurelle des programmes fonctionnels: les types de données. À l'avenir, nous proposons de transformer la partie dynamique des langues ML représentées par des fonctions. Il serait utile tout d'abord de modéliser ces fonctions sur les diagrammes de classe à l'aide des `EOperation`s sur Ecore (sous forme d'interfaces).

Aussi, il serait intéressant de prospecter une façon de représenter le comportement des fonctions. En d'autres termes, trouver un traitement ciblé qui serait adaptée à la représentation du comportement des fonctions, par exemple les diagrammes état/transition [53].

Donner des indications sur des preuves

Dans le cadre de collaborations avec le secteur industriel, les experts du domaine du secteur industriel peuvent avoir du mal à voir la représentation de leurs systèmes dans les assistants de preuve, mais aussi les preuves qui y sont effectuées. Pour cela, la documentation de preuves semble être une bonne solution. Il serait intéressant de mener des preuves sur un système utilisant un environnement de preuve interactive, puis de transformer les types de données en diagrammes de classes ainsi que la documentation de certaines preuves. Cette documentation peut être représentée sous forme d'annotations sur les méta-modèles générés. Nous pourrions utiliser le langages OCL [76] comme langage d'annotations afin de profiter de sa structure qui est adapté à la description de la formalisation.

Selma Djeddai

Association d'environnements de vérification formelle et de l'Ingénierie Dirigée par les Modèles

Directeur de thèse : Ralph Matthes, *CNRS*
Co-directeur de thèse : Martin Strecker, *CNRS*

Thèse soutenue le 24 juin 2013 à l'IRIT-Université Paul Sabatier

Résumé

Les méthodes formelles (comme les prouveurs interactifs) sont de plus en plus utilisées dans la vérification de logiciels critiques. Elles peuvent compter sur leurs bases formelles solides ainsi que sur leurs sémantiques précises. Cependant, elles utilisent des notations complexes qui sont souvent difficiles à comprendre. D'un autre côté, l'Ingénierie Dirigée par les Modèles nous propose des langages de descriptions, comme les diagrammes de classes, utilisant des notations intuitives mais qui souffrent d'un manque de bases formelles. Dans cette thèse, nous proposons de faire interagir les deux domaines complémentaires que sont les méthodes formelles et l'ingénierie dirigée par les modèles. Nous proposons une approche permettant de transformer des types de données fonctionnels (utilisés dans les prouveurs interactifs) en diagrammes de classes et vice-versa. Afin d'atteindre ce but, nous utilisons une méthode de transformation dirigée par les modèles.

Mots-clés Transformation de modèles, Ingénierie Dirigée par les Modèles (IDM), Méthodes formelles, Vérification.

Informatique-Sûreté du logiciel et calcul de haute performance

Institut de Recherche en Informatique de Toulouse - UMR 5505
Université Paul Sabatier, 118 route de Narbonne, 31062 TOULOUSE cedex 4

Selma Djeddai

Combining Formal Verification Environments and Model-Driven Engineering

Thesis Advisor: Ralph Matthes, *C.N.R.S.*
Thesis Co-advisor: Martin Strecker, *C.N.R.S.*

PhD defended June 24, 2013 at IRIT- Université Paul Sabatier

Abstract

Formal methods (such as interactive provers) are increasingly used in the verification of critical software. This is so because they rely on their strong formal basis and precise semantics. However, they use complex notations that are often difficult to understand. On the contrary, the tools and formalisms provided by Model Driven Engineering offer more attractive syntaxes and use intuitive notations. However, they suffer from a lack of formal foundations. In this thesis, we are interested in combining these two complementary domains that are formal methods and Model Driven Engineering. We propose an approach allowing to translate functional data types (used in interactive provers) into class diagrams and vice versa. To achieve this goal, we use a model-driven transformation method.

Keywords Model transformation, Model Driven Engineering (MDE), Formal Methods, Verification

Informatique-Sûreté du logiciel et calcul de haute performance

Institut de Recherche en Informatique de Toulouse - UMR 5505
Université Paul Sabatier, 118 route de Narbonne, 31062 TOULOUSE cedex 4