A TYPE SYSTEM FOR
GLOBAL MODEL MANAGEMENT

TESIS PARA OPTAR AL GRADO DE
DOCTOR EN CIENCIAS MENCION COMPUTACIÓN

ANDRÉS VIGNAGA

2011
A TYPE SYSTEM FOR
GLOBAL MODEL MANAGEMENT

TESIS PARA OPTAR AL GRADO DE
DOCTOR EN CIENCIAS MENCIÓN COMPUTACIÓN

ANDRÉS VIGNAGA

PROFESOR GUÍA:
MARÍA CECILIA BASTARRICA PIÑEYRO

MIEMBROS DE LA COMISIÓN:
PABLO BACELO BAEZA
ALEXANDRE BERGEL
JEFF GRAY
ÉRIC TANTER

SANTIAGO DE CHILE
DICIEMBRE 2011
A TYPE SYSTEM FOR GLOBAL MODEL MANAGEMENT

PHD THESIS IN COMPUTER SCIENCE

ANDRÉS VIGNAGA

ADVISOR:
MARÍA CECILIA BASTARRICA PIÑEYRO

EXAMINERS:
PABLO BACELO BAEZA
ALEXANDRE BERGEL
JEFF GRAY
ÉRIC TANTER

SANTIAGO DE CHILE
DECEMBER 2011
Resumen

La Ingeniería Dirigida por Modelos (IDM) se posiciona como un cambio de paradigma desde el desarrollo de software centrado en el código hacia el desarrollo basado en modelos. El enfoque de IDM se centra en artefactos que pueden ser procesados automáticamente, y sugiere basar los procesos de desarrollo y mantención de software en cadenas de transformaciones de dichos artefactos. Las aplicaciones de IDM a proyectos reales han mostrado resultados tanto interesantes como promisorios. Sin embargo, cuando el número de artefactos crece, su gestión se vuelve cada vez más compleja. A pesar de que unos pocos artefactos pueden ser manejados de manera sencilla, en casos de uso industriales los desarrolladores deben enfrentarse a grandes conjuntos de artefactos (e.g., modelos, metamodelos, transformaciones). Para poder utilizar esos artefactos en la práctica, se requieren mecanismos más sofisticados para crear, almacenar, visualizar, acceder, modificar y utilizar la información asociada a todas estas entidades. Este es el propósito de la Gestión Global de Modelos (GGM) introducida por el equipo AtlanMod. GGM propone representar artefactos, incluyendo composiciones y ejecuciones de transformaciones, en un modelo denominado megamodelo. La información de tipo de los artefactos puede ser utilizada para evitar errores de tipo durante la ejecución, como por ejemplo la ejecución de un elemento que no es una trasformación, o el uso de una transformación sobre argumentos para la cual no fue definida.

A pesar de que en GGM los tipos son utilizados extensivamente para evitar errores de tipo, el tipado no fue abordado en forma completa en su definición original. En particular, GGM asume que todos los artefactos gestionados son modelos que conforman con metamodelos precisos. El tipado se basa simplemente en la relación de conformidad, y los metamodelos son utilizados como tipos. Esta idea es suficiente para los casos más comunes, pero falla en otros al no lograr asignar un tipo completo a algunos elementos. Cuando esto sucede se puede introducir errores de tipo inadvertidamente. Los artefactos se relacionan entre sí mediante vínculos semánticos. Por ejemplo, una transformación refiere a sus metamodelos de origen y destino. En tales casos GGM adicionalmente utiliza una parte de dichos vínculos para el tipado. A pesar de ello, la manipulación de elementos cuyos tipos dependen de esos vínculos exceden los límites del enfoque de tipado actual. Ejemplos de ello son el caso de una transformación que recibe o produce otra transormación, o cuando el tipo del resultado de una transformación depende de un valor de entrada. Esta situación afecta el comportamiento de AM3, una herramienta que realiza GGM. Aquí identificamos limitaciones en el enfoque actual de tipado de GGM y presentamos formalmente cGMM, un sistema de tipos correcto y decidible que se enfoca en asegurar que una transformación reciba los argumentos correctos, y en tipar en forma consistente los resultados de transformaciones. Nuestro sistema de tipos fue desarrollado en coordinación con el equipo AtlanMod y constituye un primer esfuerzo por tipar los artefactos contenidos en un megamodelo. Soporta construcciones generales de IDM incluidas en la definición básica de GGM, y construcciones específicas incluidas en sus extensiones actuales. Expresando artefactos de IDM en un megamodelo como términos de cGMM permite el chequeo estático de tipos para dichos elementos e inferencia de tipos mecánica. Manipulaciones de artefactos que causarían errores de tipo pueden ser detectadas estáticamente para así evitar su ejecución. Ilustramos la aplicabilidad del sistema de tipos en casos de estudio reales. Un prototipo de cGMM fue implementado para efectuar validaciones, y fue empaquetado como un componente de AM3.
Abstract

Model-Driven Engineering (MDE) is positioned as a paradigm shift from code-centric software development to model-based development. The MDE approach is centered around machine processable artifacts that represent systems, and suggests basing software development and maintenance processes on chains of transformations on those artifacts. In the past years it has been increasingly used in real-world projects with interesting and promising results. However, when the number of machine processable artifacts rises, managing them becomes more complex. While a few artifacts are often quite easy to handle, in industrial use cases developers have to deal with large sets of MDE artifacts (e.g., models, metamodels, transformations) from which a solution has to be assembled. In order to effectively use those artifacts in practice, developers require more sophisticated ways of creating, storing, viewing, accessing, modifying, and using the information associated with all these modeling entities. This is the purpose of Global Model Management (GMM) introduced by the AtlanMod team. GMM proposes representing artifacts, including transformation composition and execution, within a model called a megamodel. Type information about artifacts can then be used for preventing type errors during execution, such as the attempted execution of a non-transformation, or the use of a transformation on arguments for which it is not defined.

Even if in GMM types are extensively used for preventing type errors, typing was not completely addressed in its original definition. In particular, GMM assumes that all managed artifacts are models conforming to precise metamodels. Typing is then simply based on the conformance relationship, and metamodels are used as types. This scheme suffices for most common cases, but it fails in others by not being able to assign a complete type to some elements. When this happens, type errors may be inadvertently introduced. Artifacts are related to each other by semantic links. For example, a transformation refers to its source and target metamodels. In such cases GMM additionally uses a part of those links for typing purposes. Despite that, manipulations of elements whose type actually involves their semantic links go beyond the limits of the current typing approach. Such manipulations include the case when a transformation is the input or output of another transformation, or when the type of the result of a transformation depends on an input value. This situation affects the behavior of AM3, a tool realizing GMM.

In this work we identify a number of limitations in GMM’s current type approach and formally present cGMM, a sound and decidable type system that focuses on ensuring that transformations receive the right arguments, and on consistently typing the results of transformations. Our type system was developed in coordination with the AtlanMod team and constitutes the first effort specifically oriented to typing artifacts within a megamodel. It supports general MDE constructs included in the core definition of GMM such as models, metamodels and metametamodels, and specific constructs included in its current extensions: ATL and composite transformations, TCS projectors and textual entities, and AMW weaving models. Expressing MDE artifacts within a megamodel as cGMM terms enables the static typechecking of these elements and type inference in a mechanical fashion. Artifact manipulations that would cause a type error can be statically detected and thus their execution can be prevented. We illustrate the applicability of the type system on real-world case studies. A prototypical implementation of cGMM was used for validation purposes and was packed as an AM3 plugin.
Acknowledgements

My PhD is reaching its end after a very long and hard journey. This thesis constitutes the results of a fair amount of the effort made by me in collaboration with many others during that journey. Now, the time to show my gratitude to those who contributed either directly or indirectly to this work has finally come.

First, I would like to thank the institution that made this thesis possible: the computer science department of Universidad de Chile was my second home for many years. CONICYT Chile provided me a generous grant for most of the program and funded many of my trips all these years. Universidad de Chile funded my first internship at the AtlanMod group from Ecole des Mines de Nantes, and the CONICYT-INRIA (Chile-France) Hot Mate project funded the second one. These two trips were key to this work. The Association for Computer Machinery (ACM) funded my expenses during two conferences. Finally, the Agencia Nacional de Investigación e Innovación (ANII) of my homeland Uruguay funded my visit to Universidad ORT Uruguay. This trip was key as well.

I am very thankful to my thesis advisor Maríα Cecilia Bastarrica. Her trust in me encouraged me when nights seemed darker than they really were. She willingly and timely reviewed this manuscript a zillion times. Thank you so much! I know it was a hard task for her being my supervisor, and there were times when I made it even harder, but I hope that at the end this experience turned out to be as good for her as it was for me. I cannot thank enough Alvaro Tasistro. He generously dedicated significant amounts of his personal time to me and my problems (I managed to become a real pain in his neck). I also thank his wife Nora Szasz and their kids for that. I cannot even imagine this work without his uncredited contribution.

I would like to thank the members of my jury Éric Tanter, Pablo Barcelo, Alexandre Bergel and Jeff Gray for their valuable comments and kindly contributing to improving this work. My apologies for the tough reading. Jeff reviewed the original proposal of my thesis and continuously showed interest in my work. I would also like to express my gratitude to those who reviewed the proposal of my thesis: Jean Bézivin, Jeff Gray again, Sergio Ochoa and Krzysztof Czarnecki. It felt awesome to get such a good feedback from all of them.
I also thank the members of the computer science department of Universidad de Chile in general, and specifically Pedro Rossel, Claudio Gutiérrez and Angélica Aguirre. Each of you played a special role in my work.

I am grateful to the past and present members of the AtlanMod team, starting with Jean Bézivin. Those two visits to Nantes literally changed my life for good for very disparate reasons. Frédéric Jouault once told me that expressing the type of the KM32ATLCopier transformation as a function posed some kind of challenge, and I naively replied “I can do that!” The rest is history. Hugo Brunelière, Guillaume Doux, Michaël Barbero, Benoît Combemale, Kelly García and Cauê Avila Clasen, in their own way, significantly contributed to this story.

Many other people comes to my mind when I look back. From a simple advice to just sympathy, this work, and myself, were positively (and in some cases inadvertently) affected by Antonio Vallecillo, Alfonso Pierantonio, Claudia Pons, Jordi Cabot, Didier Vojtisek, Jean-Marc Jézéquel and Jesús Sanchez Cuadrado. Thank you all.

My favorite uruguayan band used to sing “I only know that my friends are all what I am”. I thank all my personal friends around the world, especially Rogelio Gracia, Daniel Perovich, Carlos Burmester, José Villar and Alvaro Sordo.

This work was necessarily affected by the environment that affected me. I am forever grateful to Chile (despite its 8.1 earthquake!) and those chileans who left an indelible mark on me during all these years away from home, in a place that I came to call home.

Last but not least, I thank my family, especially my mother. I know that my late father would be proud of me. This thesis has given me a PhD degree. But it also has given me the chance to meet Irene one day after lunch while smoking a cigarette outside the front door of Ecole des Mines de Nantes. I will always thank you from the bottom of my heart for doing the craziest thing ever. I am looking at a sign of your encouragement right now. I love you.

“Hay manos capaces de fabricar herramientas con las que se hacen máquinas para hacer ordenadores que a su vez diseñan máquinas que hacen herramientas para que las use la mano.”

— Jorge Drexler, 2004
## Contents

1 Introduction .................................................. 1

2 Megamodelling and Typing ........................................ 6
   2.1 Model-Driven Engineering ...................................... 7
   2.1.1 Modeling .................................................. 7
   2.1.2 Model-Driven Approach ..................................... 8
   2.2 Technical Spaces ............................................. 13
   2.3 Global Model Management ...................................... 15
   2.3.1 Motivation .................................................. 15
   2.3.2 Metamodel Architecture ..................................... 16
   2.3.3 Megamodelling Conceptual Framework ...................... 18
   2.3.4 Realization and Functionalities ............................ 21
   2.4 Type Systems .................................................. 23
   2.4.1 Typing ..................................................... 23
   2.4.2 Formal Type Systems ....................................... 24
   2.4.3 Meta-Theoretic Properties ................................. 26
   2.5 Typing in GMM ................................................ 26
   2.5.1 Current Typing Approach .................................... 27
   2.5.2 Limitations ............................................... 29
   2.5.3 Example ................................................... 32
   2.6 Problem Statement ............................................ 34

3 Global Model Management Extensions ......................... 38
   3.1 The AtlanMod Transformation Language Extension .......... 40
   3.2 The Textual Concrete Syntax Extension ...................... 42
   3.3 The Composite Transformations Extension .................... 43
   3.4 The AtlanMod Model Weaver Extension ...................... 45
   3.5 Discussion .................................................... 46
CONTENTS

7.3 Discussion .................................................. 124

8 Conclusions and Further Work ... 126
  8.1 Summary .................................................. 126
  8.2 Contributions and Conclusions ................. 128
    8.2.1 Contributions ...................................... 128
    8.2.2 Conclusions ......................................... 129
  8.3 Further Work ............................................ 131
    8.3.1 AM3 Tool ........................................... 132
    8.3.2 cGMM Calculus .................................... 132
    8.3.3 GMM Approach ..................................... 134

Bibliography .................................................. 136

List of Figures ................................................. 146

Index .......................................................... 148
Chapter 1

Introduction

In the field of software development, the increasing use of Model-Driven Engineering (MDE) [Sch06] in recent years has led to more complex situations. MDE mainly suggests basing the software development and maintenance processes on models and chains of model transformations. A few MDE artifacts (e.g., models, metamodels, transformations) can be easily managed, but as soon as industrial use cases are tackled, we are faced with large sets of interrelated heterogeneous and complex artifacts that become unmanageable. In order to be able to use them, but keep the complexity of MDE under control, we need to count on sophisticated ways of creating, storing, viewing, accessing, modifying, and using the information associated with all these modeling entities. This is the purpose of Global Model Management (GMM) [BJRV04].

GMM is a solution for coping with complex management cases by modeling in-the-large. A megamodel [BJB08] is a special kind of model that stores references to artifacts and relationships among them. GMM defines a metamodel describing the MDE artifacts that can be contained in a megamodel. Such a metamodel is incrementally introduced [Mod09b]. A core metamodel defines basic and general constructs, and a primary extension introduces the notion of a model and its variants. Additionally, it presents a number of extensions targeting specific domains that currently include ATL [JK05], TCS [JBK06] and AMW [FBJ+05], as well as compositions of transformations. A megamodel not only stores and organizes artifacts, but it also enables their execution. A transformation may be applied to existing models for producing new ones, augmenting the contents of the megamodel. Execution of artifacts within a megamodel enables a number of powerful applications, but it also introduces the notion of execution error.

As the managed modeling resources may be of different nature, some support for classifying them is required. In order to cope with this heterogeneity, a GMM solution has to rely on a definition which allows precisely typing all the involved artifacts. This should prevent a form of execution error: a type error. In the context of this work,
type errors are caused by the attempted execution of a non-transformation, and the use of a transformation on arguments for which it is not defined. GMM currently lacks a type system, and its simplistic typing approach is only informally described. A precisely defined type system for GMM is required, especially when its implementation is to be included in a supporting tool.

Problem Statement

The GMM approach currently assumes that all managed artifacts are models conforming to precise metamodels. Model typing is then simply based on the conformance relationship, and metamodels are used as types. This straightforward approach is complemented by the fact that artifacts are also related by strong semantic links. For instance, a transformation refers to its source and target metamodels (i.e., its parameter and return types). Typing based on a combination of conformance and semantic links suffices for most common cases. However, in more elaborate cases this scheme notably fails by losing sensitive information such as the complete type of the result of a transformation. As a consequence, appropriate checks for guarding a megamodel from some harmful actions that lead to execution errors could not be performed. Moreover, it may not be possible to automatically assign a type to some elements resulting from the execution of transformations. One problematic situation occurs when the type of the result of a transformation depends on (a part of) the type of its parameters, like in the two following cases:

- When a metamodel is used as input to a transformation (i.e., a type is used as a value)
- When a transformation is used as input to another transformation (i.e., a function is used as a value)

Additionally, in GMM not every artifact is a model. In fact, textual entities (i.e., textual representations of artifacts) are not models and are left untyped since the typing-by-metamodel relation is not applicable to them. As a consequence, for transformations that involve textual entities it is not possible in general to know what kind of elements they safely accept or produce. In turn, a weaving model represents links among other models and conforms to a metamodel that specifies the semantics of those links. A subsequent usage of a weaving model may lead to an element whose type could depend on the type of any of its woven models, and thus such an element could not be properly typed.

In a concrete project, a megamodel would refer to all involved artifacts, and the manipulation of such a megamodel would encompass the progress of project-specific processes. For this reason, a megamodel-centric tool, such as AM3 [AM309], should prevent the occurrence of execution errors in general, and type errors in particular, when
its underlying megamodel is manipulated. A more complex typing approach for GMM is required, and more specifically, a type system [Mod09a].

Formal approaches provide unambiguous definitions which enable convincing proofs of their properties and provide a precise foundation for implementations. However, formalizing a language is a hard task. For example, only a subset of Java has been formally specified [Hui01]. Even the type system of Haskell is informally described [Jon99]. Applications of formal methods to MDE are still in their initial stage. Favre [Fav04] motivated a theory for MDE and presented a metamodel for core MDE concepts and their relationships only. Typing of those concepts was not addressed though. In [SJ07], Steel discusses the typing of (terminal) models and formalizes a type system for a basic transformation language. However, such an approach does not include metametamodels, higher-order transformations and transformation compositions. In a close relation to our approach, Poernomo [Poe06] used Constructive Type Theory for encoding the MOF metamodeling architecture. The single metametamodel formalism does not match our context, and although many concepts were considered, transformations were not. In that same direction, the Predicative Calculus of (Co)Inductive Constructions (pCIC), the underlying formal language of Coq [CPT09], provides elements that directly map to some GMM constructs, especially dependent types and higher-order functions, and thus notably influenced our work.

Overview

The goal of this work is to define a formal type system, specifically targeting GMM. A type system addresses a set of type errors $E_T$ and may be used for determining well-typing. A well-typed program, with respect to a consistent type system, then exhibits good behavior (i.e., does not cause type errors from $E_T$ upon execution, and thus program behavior is not unpredictable). In our context, a program corresponds to actions on a megamodel, and execution occurs in the context of a GMM tool such as AM3. Formality in the definition of a type system not only provides an unambiguous specification towards its implementations, but it also enables the precise proof of some of its fundamental properties such as soundness and decidability.

We claim that a formal type system for GMM provides a proper foundation for a correct use of megamodels. In this thesis we introduce cGMM, a dependently typed calculus dedicated to the core constructs of GMM and its extensions. Expressing GMM elements as terms of our calculus enables to statically typecheck these elements in a mechanical fashion as follows. Terms in a static typing environment mirror elements in a megamodel and handle their typing. Actions on the megamodel (e.g., adding a new element and executing a transformation) are then guarded by their counterpart in the static typing environment; if an action on the environment is valid, then it can be safely
performed on the megamodel. Associated proofs of soundness and decidability, and significant applications, validate our approach. The practical benefits of such a formal definition are demonstrated by a tool which is ready to be integrated with AM3. Our work makes the following contributions:

- A formal type system specific to GMM is defined. Such a type system is proven to be sound and decidable, and enables the prevention of the most common forms of type errors: the application of non-functions and the application of functions to values of the wrong type. The type system is also extensible for encompassing the evolution of GMM, and it can be implemented within any GMM tool.

- A formal semantics for megamodel manipulations is given. In combination with our calculus, such a semantics provides a basic formal language which can be evolved into a megamodel-based programming language.

- A complete application with practical impact of formal methods to MDE is provided. In particular, such an application involves moving beyond the borders of the MDE technical space to a domain where typing facilities better apply and their construction is more cost-effective.

- A solid foundation for typing MDE artifacts within GMM tools is developed. Results of this work are implemented as an AM3 plug-in.

Our work improves one basic aspect not stressed in the GMM original proposal, contributes to the applicability of MDE based on more reliable tool support, and helps bridge still separate domains such as formal methods and MDE. This thesis integrates and extends results that have been presented in [VB09a, VB09b, Vig09c, VJBB09, VJBB11].

**Structure of the Thesis**

The remainder of this thesis is structured as follows. Chapter 2 is dedicated to discussing the necessary background. We start from Model-Driven Engineering, through Technical Spaces, to Global Model Management. We also discuss Type Systems, and address how typing is approached in GMM. In addition, some of its current limitations are identified. Based on the discussed concepts, we close with a detailed account of the problem to be addressed in this thesis. In Chapter 3, we dive into the details of the GMM metamodel extensions. This sets the context for some of the key decisions in the design of our type system. Chapter 4 completely introduces a formal type system for GMM. We define the admissible terms, the notion of environment, and the type rules. A procedure for extending the type system for supporting future GMM extensions then closes. In Chapter 5, we prove the main properties of our type system: soundness and
decidability. To that end, we introduce the notions of principal and direct types for coping with type multiplicity introduced by subtyping, and a semantics for megamodel manipulations. Such a semantics is a key element in type soundness. Decidability follows from an algorithm that solves both typechecking and type inference. Chapter 6 addresses a number of real-world applications for demonstrating the operation of the type system. We choose simple but real-world MDE case studies which pose challenging typing situations involving higher-order transformations, dependently typed transformations, textual entities and projectors, weaving models, and composite transformations. In Chapter 7, the practical application of the type system is addressed. We discuss the typing of MDE artifacts within a GMM tool, and present an implementation of our type system. Such an implementation can be accessed by a command line tool that handles terms of our calculus directly for validation purposes, and conforms the core component of an Eclipse plug-in which is ready for integration with AM3. Chapter 8 concludes with a summary of our results, a discussion of the main contributions of our work, and an outline of some possible directions of future work.

About this Thesis

This thesis followed a long path before taking its final form. In fact, our original interest was set on developing a Software & Systems Process Engineering Metamodel [OMG08] (SPEM)-based methodology for developing model transformations [Vig07]. As the MDE community was still debating basic issues, we soon found that some fundamental concerns required a solution before such a general topic could be addressed effectively.

In our path to grasping model transformation development, we addressed different languages (Kermeta [VB07], QVT [VPB08] and ATL [Vig09a, Vig09b]) and approaches (transformation models [Vig08]). While experimenting with ATL higher-order transformations and composite transformations, a more basic problem in their development was identified: their typing. Moreover, we noticed that advanced theoretical concepts such as dependent types were applicable to model transformations, regardless of their internal complexity.

In that context, the goal of the thesis was revised. While investigating the typing of higher-order transformations and dependently typed transformations, we discovered that our findings could be extended to other constructs such as model-to-text and text-to-model transformations, and weaving models. The focus of our work then shifted from model transformation development to the problem of typing MDE artifacts within a global model management setting. The cGMM calculus, the proof of its main properties and the prototypical implementation which integrates with AM3 are the results of that progression.
Chapter 2

Megamodeling and Typing

Global Model Management (GMM) is about managing large sets of artifacts which, for example, are involved in a software development project. Model-Driven Engineering (MDE) promotes basing software development on precisely defined artifacts (i.e., models) expressed in precise domain-specific languages representing multiple views of the system under construction, where some of them are operational definitions of precise operations on other models. However, in some cases, artifacts are (also) defined outside of this scope where other approaches and facilities (better) apply. A megamodel is a special artifact where not only MDE-based artifacts can be stored for management. The contents of a megamodel evolves encompassing the progress of the process where such artifacts are involved. Such evolution is partially produced by the execution of operations (i.e., transformations) on the artifacts contained in the megamodel. Artifacts can be typed, and transformations both operate on and produce specific types of artifacts. Not checking, either by choice or by lacking information, if an artifact has the type a transformation expects may cause an execution error. Type systems provide the means for preventing such errors. Even though GMM is concerned with typing, it lacks a type system which ensures good behavior of a tool realizing the megamodeling approach.

In this chapter, we review the necessary concepts which lead us to a concrete definition of the problem addressed in this work and its solution. In Section 2.1 we set the foundations of the modeling world, the general context of this work, by summarizing the basic notions of the MDE approach and their organization. The world of human-readable specifications as an example of a foreign working context is addressed in Section 2.2 where technical spaces are treated. Once the sources of artifacts to be managed are introduced, we discuss in Section 2.3 the GMM or Megamodeling approach including its practical realization. The basic concepts of a formal type system, which is the essence of the contribution of our dissertation, are presented in Section 2.4. How GMM currently addresses typing and what the limitations and issues of such an approach are that motivates our
work are detailed in Section 2.5. Finally, in Section 2.6, we revisit the problem statement of this thesis based on the concepts, approaches and issues presented in this chapter.

2.1 Model-Driven Engineering

In this section we review the Model-Driven Engineering approach which encloses the general context of our work and introduces the basic concepts for Megamodelling. Models are first-class constructs in Model-Driven Engineering, thus we begin with a discussion on the role of modeling in software development. We then address model-based development by discussing its basic notions in terms of terminal models, metamodels and metametamodels, and model-related operations in terms of model transformation and model weaving.

2.1.1 Modeling

The complexity of computer-based systems, both in terms of functionality and quality of service, has been steadily increasing in the last decades. Software developers have relied on modeling for coping with such a complexity, as in other disciplines such as engineering or architecture. Modeling is essential to human activity as every action is preceded by the (implicit or explicit) construction of a model [Béz05]. Kühne [Küh06] concisely defined the notion of model as an abstraction of a (real or language-based) system allowing predictions or inferences to be made. More specifically, Stachowiak [Sta73] identified three features for models:

1. A model is based on an original (i.e., a system),
2. A model only reflects a relevant selection of an original’s properties, and
3. A model needs to be usable in place of an original with respect to some purpose.

According to Rothenberg [Rot89], modeling is the cost-effective use of something in place of something else for some cognitive purpose. It allows us to use something that is simpler, safer or cheaper than reality instead of reality for some purpose. As models are not intended to represent all aspects of reality, they allow us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality. Models may be either descriptive or prescriptive [Rot90]. A descriptive model is a representation of an existing system. A prescriptive model is a representation of a system which is intended to be built. Even though a descriptive model is built by observation of a system and a system is built by observation of a prescriptive model, in either case a system is represented by a model [Béz05].
2.1 Model-Driven Engineering

2.1.2 Model-Driven Approach

Prescriptive models are used as blueprints for building systems. Their main purpose is to support planning and early validation, i.e., finding errors as early as possible and partially evaluating a system before it is realized [Küh06]. However, from an historical perspective, a common practice in software development when dealing with a problem is to informally build a high-level model of the problem and to refine that model as understanding of the problem, and its solution, is gained. A prescriptive model is adjusted so that it represents the right system. This process usually ends when the level of abstraction is close enough to that of the implementation constructs, since the development effort is focused on the implementation level. Models are then usually put aside and left unmaintained, and the representation relationship breaks. In other words, models are a volatile means for getting to the lower level. In this context, the value obtainable from the prescriptive nature of models is limited. If additionally considering the lack of rigor with which models are expressed and the lack of preciseness with which they are interpreted, modeling has not delivered all its potential benefits to software development and as such has not been considered as a main activity.

Model-Driven Engineering (MDE) [Sch06], introduced by Kent in [Ken02], is positioned as a paradigm shift from code-centric software development to model-based development [Béz05]. MDE proposes to change the purpose of models in software development, seeking a raise in the level of abstraction in which developers reason about problems. With such an approach, it is presumed that the major effort will be devoted to generation, maintenance and synchronization of models, which constitute the fundamental assets in software development. In what follows we discuss a number of basic notions in MDE.

2.1.2.1 Basic Notions in MDE

Along with the widely accepted representation relationship between models and systems, Bézivin introduced in [Béz05] the conformance relationship. A model conforms to its metamodel, just like a map conforms to its legend. A map is written in the language defined by its legend, and the legend indicates how to interpret the map. Similarly, each model is written in the language of its metamodel. A metamodel is a formal specification of an abstraction, usually consensual and normative. From a given system we can extract a particular model which represents it in terms of the constructions defined by a specific metamodel. A metamodel acts as a precisely defined filter expressed in a given formalism.

A system involves a number of different aspects or concerns. For example, typical concerns for a building include electrical, plumbing, ventilation, and so on, which may be separately expressed using proper notations. A system may then be represented by different models, each of them focusing on a separate concern. Such models need
not be expressed in the same language. In fact, domain-specific languages defined by appropriate metamodels could be used for that purpose. Models that represent different views of the same system are naturally related and thus the concerns they address can be combined. This enables the coordination of models which are based on different metamodels. Model coordination can be achieved by means of model transformation and model weaving, which we shall discuss next. As stated before, metamodels need to be precisely defined. A *metametamodel* defines a language for expressing metamodels. Such metamodels conform to that metametamodel. Additionally, a metametamodel conforms to itself. In other words, the (domain-specific) language defined by a metametamodel is used for expressing metamodels and the metametamodel itself, which is nothing less than the metamodel of those metamodels. Concrete metametamodels include MOF [OMG06], ECore [EMF10] and KM3 [JB06]. To summarize, MDE introduces the following three different kinds of models organized in levels M1 to M3:

- **Terminal models** (level M1): conform to *metamodels* and are representations of real-world systems.
- **Metamodels** (level M2): conform to *metametamodels* and define domain-specific concepts.
- **Metametamodels** (level M3): conform to themselves and provide generic concepts for metamodel specification.

These kinds of models, a real-world system (considered at an M0 level), and the representation and conformance relations among them are illustrated in Figure 2.1.

### 2.1.2.2 Model Transformation

Models are built to be used. Using a model consists of exploiting the information contained in it (i.e., model elements and their relations) and even in its relations to other models. Actions on that information ranges from highly inventive to purely repetitive. The underlying development process strongly determines how, why and when to manipulate models. Model manipulation usually yields other models but also other work products such as reports, source code, configuration scripts, and so on. When models are just drawings on a piece of paper or a blackboard, or artifacts informally represented on a computer-based tool, their manipulation provides limited benefits even if it is performed manually. One important consequence of defining precise metamodels is that representations of systems become machine processable. A program that produces models from other models is called a *model transformation*. A model transformation is considered itself a model which conforms to a specific metamodel, the metamodel that defines the model transformation language in which the transformation is expressed. Figure 2.2 illustrates the elements involved in a model transformation. Transformation Mt conforms
2.1 Model-Driven Engineering

to metamodel $\text{MM}_t$ and operates on models conforming to $\text{MM}_a$ for producing models conforming to $\text{MM}_b$. $\text{MM}_a$ is the source metamodel and $\text{MM}_b$ is the target metamodel. As a concrete case, the application of $\text{M}_t$ to $\text{M}_a$ produces $\text{M}_b$. $\text{M}_a$ is the source model and $\text{M}_b$ is the target model. The operation of $\text{M}_t$ is conceptually simple: from some model elements contained in $\text{M}_a$, $\text{M}_t$ produces and links model elements contained in $\text{M}_b$. Since $\text{M}_t$ produces models from models, it is called a model-to-model (M2M) transformation. Other kinds of transformations, such as model-to-text (M2T) and text-to-model (T2M) will be discussed later. Metamodel $\text{MM}_t$ could be the metamodel of any of the various available model transformation languages, such as ATL (AtlanMod Transformation Language) [JK05, JABK08] or QVT [OMG11]. As $\text{M}_t$ is in fact a model, some interesting situations are enabled. There could be another model transformation which takes $\text{M}_t$ as its source model, or still another transformation where $\text{M}_t$ is its target model. A transformation which accepts or produces model transformations is called a higher-order transformation (HOT) [TJF+09]. Furthermore, as metamodels and metametamodels are models, they could also be manipulated by transformations. Note that if $\text{M}_t$ accepts metamodel $\text{MM}_a$ as its source model (the source metamodel in this case would be metametamodel $\text{MMM}$), a terminal model conforming to $\text{MM}_a$ could be involved in the result of $\text{M}_t$. In such a case, viewing $\text{M}_t$ as a function, $\text{MM}_a$ is both used as a value (in the source or domain) and as a type (in the target or codomain). By analogy to functions in type theory, we call a transformation of this kind a dependently-typed transformation (DTT). HOTs and DTTs pose interesting challenges for characterizing their source and target models and metamodels which lay at the very heart of the problem addressed in our work. Such kinds of transformations will be discussed in detail in a later section.

Figure 2.1: Three-layered organization of basic MDE notions
2.1.2 Model Weaving

As discussed before, different models representing different views of the same system are somehow related. In object orientation, associations relate classes and enable links among instances of those classes, and the meaning of such relations is defined by the user. Similarly, relations among models, and among model elements within them, may be defined. This kind of relation between models is called model weaving. Model weaving enables the definition of user-defined relationships between models. More specifically, a relationship is realized by a set of links which connect model elements contained in the
2.1 Model-Driven Engineering

Figure 2.3: Model weaving

models involved in the model weaving. Furthermore, such set of links is the contents of a model: a weaving model. As illustrated in Figure 2.3, for relating models Ma and Mb, a weaving model Mw is defined. Mw has the form \{<a1,b1>,<a2,b1>, ... \}, where <ai,bj> denotes a link between model elements ai and bj, and ai and bj are such that ai ∈ Ma and bj ∈ Mb. Model weavings may be n-ary in general. A model weaving is essentially a mapping, and as such, it enables a number of useful applications [FBV06, SME09]. In particular we could mention transformation specification and tracing. Tracing models are a particular usage of weaving models. In this case, a link between two elements means that one of them was created (during the execution of a transformation) from the other. In the example of Figure 2.3, model Mw relates models Ma and Mb. Links within Mw then would specify which model elements of Ma were used for generating which model elements of Mb. Links may have different meanings, which directly depend on the purpose of the model weaving where they participate. Being a model, a weaving model conforms to a (weaving) metamodel. Such a metamodel, in turn, defines the semantics of links. As a consequence, a separate weaving metamodel is required for each kind of relationship between models one wishes to define.

AtlanMod Model Weaver (AMW) [FBJ+05] is one possible realization of the notion of model weaving. It proposes a concrete approach to deal with the problem of multiple weaving metamodels. Such metamodels exhibit more commonalities than differences. Therefore, AMW defines a core weaving metamodel that factorizes those commonalities, from which several specific extensions may be produced. In particular, an abstract notion
of link is defined, which is expected to be specialized as needed by the weaving metamodel extensions.

2.2 Technical Spaces

Artifacts involved in a software development project can be organized into technical spaces \[\text{KBA02}\]. A technical space is a working context with a set of associated concepts, body of knowledge, tools, required skills and possibilities. A technical space may be organized according to the three-layer architecture illustrated in Figure 2.1. It includes a (self-defined) metalanguage, all languages produced by it, and all specifications expressed in those languages. The MDE technical space is the world where models live. It is actually an aggregation of a number of concrete model-based technical spaces, each of them induced by a different metametamodel (e.g., ECore, KM3 or MOF). By the MDE technical space we mean the modeling world in general, and we do not imply any of those specific model-based technical spaces. MDE-based projects focus on artifacts lying within the MDE technical space. However, other technical spaces are important. The Grammarware \[\text{KLV05}\] technical space is another example. Such a technical space enables the definition of language grammars. Similarly to the MDE technical space, there are a number of metalanguages within Grammarware, where EBNF is probably the most widely known. Very often domain-specific languages have textual concrete syntaxes; for this reason textual representations of artifacts are a common situation.

In many cases additional representations of systems within other technical spaces different from MDE could be desirable. It is possible to represent a view of a system \(s\) as a model \(a_m\) in the MDE technical space and as a non-model \(a_e\) in a foreign technical space \(T\). On the one hand, \(a_m\) can be manipulated via model transformations (i.e., MDE-based facilities), and on the other hand, the (external) \(a_e\) representation can be more appropriately manipulated through facilities provided by \(T\). Bridging those two worlds enables the generation of \(a_m\) from \(a_e\) and vice versa. This in turn enables the reuse of existing facilities. Moreover, facilities can be built within the technical space where their construction and operation is more cost-effective. Considering the case where external representations are textual, a model representation of a Java program could be more suitable for some refactorings via model-to-model transformations, while the textual representation is required for compilation, and it could be more appropriate for manual editing. When the external representation is textual (i.e., it is a textual entity), conversion is known as model-to-text and text-to-model transformations. Textual Concrete Syntax (TCS) \[\text{JBK06}\] is a concrete technology for realizing such kinds of transformations.

Bridging technical spaces involves the ability to convert a specification in a language within one technical space to a specification in a language within another technical space.
This is carried out by operations called *projectors*. A projector is an artifact that lives in a technical space $T$; if it transfers artifacts from $T$ to another technical space it is called an *extractor*, and it is called an *injector* if it transfers artifacts in the opposite direction. Figure 2.4 illustrates a bridge between the MDE and the Grammarware technical spaces. $M$ and $T$ are two representations of the same system in those technical spaces. Model $M_e$, which conforms to metamodel $MMe$, is an extractor that produces $T$ from $M$. Similarly, model $M_i$, which conforms to metamodel $MMi$, is an injector that produces $M$ from $T$. Note the implicit relation between metamodel $MM$ and grammar $G$: $G$ can be regarded as the grammar defining a textual concrete syntax for the language defined by $MM$.

One interesting case is that of TCS mentioned above. A TCS artifact is a model (MDE technical space) from which both an injector and an extractor are generated (models $M_i$ and $M_e$ in Figure 2.4). This means that in this case metamodels $MMi$ and $MMe$ are not two separate metamodels, they are actually the metamodel of TCS. Furthermore, TCS has a textual concrete syntax, and therefore textual TCS specifications (Grammarware technical space) may be handled. For example, the target of a transformation could be a model representation of a TCS artifact, which could be projected to the Grammarware technical space for manual editing. In turn, a developer could write a textual TCS specification by hand, which could be projected to an MDE technical space for being the...
source of another transformation. Finally, both projectors can be realized by TCS itself.

Our work mainly focuses on artifacts within the MDE technical space, however, textual representations of models along with TCS-based bridges to the Grammarware technical space will be also considered.

2.3 Global Model Management

In this section we discuss the basic elements and ideas of Global Model Management, or Megamodelling, which sets the context of our work. We start by motivating the approach and describing its overall architecture based on a core metamodel and several extensions. We then discuss some of the key elements that conform to the megamodelling conceptual framework. We also describe a tool that realizes the approach and some of its main functionalities. The core metamodel and its primary extension is also described. The remaining extensions are discussed in full detail in the next chapter.

2.3.1 Motivation

Boehm introduced in [BS92] the notion of megaprogramming as a technique for addressing the development of large-scale software systems. The notion of megamodelling introduced by Bézivin in [BJV04] by analogy addresses the development of large-scale systems from a modeling perspective. MDE-based projects are intensive in modeling artifacts such as terminal models, metamodels, metametamodels, transformations, weavings, as well as other artifacts from foreign technical spaces. For example, the Rational Unified Process (RUP) [SK08] introduces over a hundred different kinds of artifacts, each of which could yield several separate terminal models. Furthermore, ideally, parts of the development and maintenance processes could be encoded as compositions of transformations. One single transformation involves only a few artifacts: the transformation model itself, one or more models, one or more metamodels and at least one metametamodel. In turn, an industrial project could involve thousands of artifacts to be managed. Additionally, such artifacts also have the following qualities:

- **Distributed.** An artifact does not necessarily have to be located in the same computer where it is used, not only within an organization but also across different collaborating companies.

- **Heterogeneous.** Artifacts are of a different nature. Models can be terminal models, metamodels, metametamodels, transformations, and so on. Additionally there may be artifacts from foreign technical spaces. Each kind of artifact exhibits different properties and behavior. Furthermore, several languages may be involved. Naturally, a terminal model conforming to different metamodels are
expressed in different languages, but such metamodels may in turn conform to
different metametamodels. There could be transformations defined in different
transformation languages as well.

- **Interrelated.** Artifacts are related to each other through several kinds of relations.
  In addition to the basic conformance relation, a transformation is related to its
  source and target metamodels, the source and target models of a transformation
  are related to it, and those same source and target models may be related to a
  trace (weaving) model. As new kinds of artifacts are considered, new relations can
  be defined.

- **Large and complex.** Models are representations of systems. The increase in the
  size and complexity of systems is naturally encompassed by an increase in the size
  and complexity of models. Artifacts can easily become complex networks of a large
  amount of model elements.

- **Not closed.** A particular project may require the definition of specific kinds of
  artifacts which were never used in other projects before. Therefore, the set of
  manageable artifacts is not closed to a fixed definition.

Managing artifacts and their relations in this context constitutes a major challenge.
Global Model Management (GMM) or Megamodeling [BJRV04] is a model-based ap-
proach for coping with this complex situation by *modeling in the large*. GMM is centered
around the notion of a *megamodel* [BJB08]. A megamodel is a special kind of model that
logically contains manageable artifacts (i.e., models, and artifacts from foreign technical
spaces) and relationships among them. The principle is the following: a megamodel
may be used for representing all the artifacts involved in a real-world system or process.
Megamodels provide an environment for efficiently creating, storing, viewing, accessing,
modifying, executing and using large amounts of artifacts and their relations.

### 2.3.2 Metamodel Architecture

Megamodels are models, and as such they must conform to a metamodel. Such a meta-
model has to describe what megamodels are. In essence, megamodels are collections
of manageable *elements* which must be precisely defined by the metamodel. As stated
above, a taxonomy of manageable artifacts is not fixed as the management of new kinds
of artifacts may be called for at some point in time. For this reason, the metamodel of
megamodels is expected to grow in time and thus must support incrementality.

The metamodel of megamodels is modularly defined as illustrated in Figure 2.5. One
core metamodel introduces basic megamodelling constructs, and a number of domain-
specific extensions in turn introduce constructs for supporting different kinds of artifacts
specific to different modeling contexts. In particular:
2.3 Global Model Management

Figure 2.5: Modular structure of the metamodel of megamodels

- **AM3Core**: This metamodel includes the most basic constructs for megamodelling which are the notion of megamodel and two variants of elements: entities and relationships.

- **GlobalModelManagement**: This extension introduces the notion of a model as a variant of an entity with a categorization compatible with the three-layered organization of Figure 2.1. Variants of terminal models (e.g., transformation models and weaving models) are also included. Furthermore, the notions of model transformation and model weaving are also included as variants of unidirectional and bidirectional relationships respectively.

- **GMM4ASM**: This extension adds constructs related to the ATL virtual machine.

- **GMM4ATL**: This extension defines ATL-specific artifacts (see Section 3.1 for details).

- **GMM4TCS**: This extension defines TCS-specific artifacts (see Section 3.2 for details).

- **GMM4CT**: This extension defines artifacts for dealing with composite transformations (see Section 3.3 for details).

- **GMM4AMW**: This extension defines AMW-specific artifacts (see Section 3.4 for details).
2.3 Global Model Management

When artifacts which are not already supported by this metamodel are to be managed, new extensions can be created. For example, a GMM4QVT metamodel extension could extend GlobalModelManagement for specifically defining QVT-related artifacts. The metamodel architecture presented above directly maps the architecture of the tool which is discussed later in this section. Among others, such a tool enables the execution of transformation models. This explains the existence of the GMM4ASM extension, which is not actually relevant for our work, where elements related to the ATL virtual machine are defined. Furthermore, the implementation of the metamodel includes infrastructure-related elements. For example, the AM3Core metamodel includes facilities for locating artifacts (to be contained in megamodels) in a global environment. We refer the reader to [Mod09b] for further information.

The AM3Core metamodel and the GlobalModelManagement extension are briefly discussed next. In the next chapter, other extensions are discussed in more detail.

2.3.3 Megamodelling Conceptual Framework

The basic concepts of MDE already discussed at the beginning of this chapter (e.g., model, metamodel, metametamodel, terminal model, transformation, weaving) are defined as a number of general constructs of the GMM approach. Most of such constructs are defined in the AM3Core metamodel and in the GlobalModelManagement extension. In this section we discuss other basic constructs and how GMM relates all of them [Mod09a].

Figure 2.6 shows some selected basic constructs of GMM and their relations. A megamodel is a collection of manageable elements, and Element is their root class. An element may be either an Entity (i.e., an artifact) or a Relationship among them. These constructs are defined in the AM3Core metamodel\(^1\). An Entity can be a Model or a TextualEntity. A Model is an MDE artifact, while TextualEntity, defined in the GMM4TCS extension, is an M1-level specification from the Grammarware technical space.

The subhierarchy of Figure 2.6 having Model as its root is entirely defined in the GlobalModelManagement extension. It obviously introduces all manageable variants of the abstract notion of a model. TerminalModel corresponds to the homonymous concept at level M1 shown in Figure 2.1. A reference model is a model defining a language for expressing other models. Metamodels and metametamodels fall in this category; if we would merge levels M2 and M3 we would obtain the “level of reference models”. With the structure described so far, a Model in general conforms to a ReferenceModel. In particular, invariants at the bottom of Figure 2.6 express the constraints illustrated in

\(^1\)In GMM there are actually two megamodel constructs. One is Megamodel, which is associated to many Element and is defined in the AM3Core metamodel. The other is MegaModel, which is defined as a variant of TerminalModel in the GlobalModelManagement extension. Given the structure of Figure 2.5, one single megamodel construct cannot belong to AM3Core and extend TerminalModel at the same time. The Megamodel class in Figure 2.6 is a simplification of this situation.
2.3 Global Model Management

Figure 2.6: Selected constructs from the AM3Core metamodel, and the GlobalModel-Management and TCS extensions.

Figure 2.1: a terminal model conforms to a metamodel, a metamodel conforms to a metametamodel, and a metametamodel conforms to itself.

Terminal models conform to metamodels and are representations of systems. However, there may be terminal models with specific features. Transformation models and weaving models are possible cases of terminal models. TransformationModel and WeavingModel are the base classes for all variants of transformation models (e.g., ATL and TCS models) and weaving models (AMW models), respectively. Those constructs are extended in the GMM4ATL, GMM4TCS and GMM4AWM extensions as discussed in the next chapter. Another case of terminal model is that of megamodels. In GMM, a megamodel is considered as a representation of a working zone [Mod09a]. For an MDE application developer, the working zone is the complete set of artifacts that he or she may potentially use to build an application. There are actually many events that may change a megamodel like the creation or deletion of a model or metamodel, or the execution of a given transformation, since it produces target models. In Figure 2.7 we show an example megamodel containing entities that represent artifacts in a sample working zone.
2.3 Global Model Management

Figure 2.7: Some MDE artifacts in a sample working zone and their representation in a concrete megamodel (τ denotes the typing relationship among megamodel elements). The GMM metamodel stands for the complete GMM metamodel shown in Figure 2.5. Dashed arrows relate megamodel elements and the artifacts they represent.

In addition, the GMM framework proposes different kinds of relationships between entities: unidirectional and bidirectional relationships. Such relationships are user-defined and should not be mistakenly considered as relations such as c2 or sourceMM in Figure 2.2 or wovenL in Figure 2.3. A ModelTransformation (unidirectional) relationship allows specifying source and target metamodels of a given TransformationModel, and can thus be regarded as its signature. In GMM some elements have dual representations. More specifically, some entities have their dual in relationships. A model transformation is the dual of a transformation model [BBG+06].

Finally, a ModelWeaving (bidirectional) relationship is the dual of a WeavingModel and allows specifying the metamodels of the woven models. As with terminal models, the model transformation and model weaving relationships are further specialized in other
2.3 Global Model Management

metamodel extensions. From an execution point of view, the TransformationRecord relationship (not shown in Figure 2.6) allows specifying actual source and target models concerning a specific execution of a given transformation. For example, considering the execution of Mt in Figure 2.2, a transformation record would specify that Ma is the sourceM and Mb is the targetM.

2.3.4 Realization and Functionalities

The current version of the Eclipse.org AtlanMod MegaModel Management (AM3) solution implements the GMM approach. It is a project which is part of the GMT subproject, which is itself part of the top-level Eclipse Modeling project. As an Eclipse project, the AM3 prototype is fully open-source and thus all its source code is freely available from its Eclipse website and download server [AM309].

The AM3 tool, built on top of the Eclipse environment, provides an underlying megamodel where manageable artifacts and their relationships can be registered. When an artifact is registered, the corresponding construct within GMM’s metamodel is instantiated and linked to the underlying megamodel. Furthermore, metadata associated to such an artifact (e.g., its name, location, the metamodel it conforms to, and so on) may also be registered. This involves setting the appropriate properties of the created construct, linking such a construct to already existing constructs (e.g., when specifying that the newly created terminal model conforms to an already registered metamodel), as well as instantiating associated constructs and setting their properties (e.g., when specifying the formal parameters of a newly created transformation).

The underlying megamodel of AM3 is manipulated by a user through a graphical user interface (see Figure 2.8). Such an interface provides a standard megamodel navigator and a number of editor pages. The megamodel navigator enables the user to access the contents of the underlying megamodel, dynamically recognizing all the available extensions. In turn, there are two different extensible editor pages: the entity editor page and the relationship editor page. The former page is used for any construct directly or indirectly extending the Entity construct. The latter page is used for any construct directly or indirectly extending the Relationship construct. For a given element, which is either an entity or a relationship, the corresponding editor page is generated with a number of tabs; some tabs are dedicated to the metadata specific to the element while the rest of the tabs show metadata inherited by the element on the way up in the hierarchy to Entity or Relationship.

Structurally, AM3 is composed of three distinct sets of Eclipse plug-ins, where the first two directly map the metamodel structure shown in Figure 2.5:

- The core plug-ins provide the basic metamodel of GMM, the core runtime environment, the main APIs and associated generic navigator and editors.
2.3 Global Model Management

Figure 2.8: The AM3 tool. Megamodel Navigator in the left pane, and Editor Page in the right pane.

- The **extension** plug-ins provide different extensions to the metamodel of GMM (e.g., GlobalModelManagement, GMM4TCS, and so on). Each plug-in additionally provides related specific APIs and corresponding extensions to the user interface (e.g., specific tabs for editor pages, contextual actions, etc).

- The **utility** plug-ins provide basic model-repository solutions and extraction facilities for ANT script models and Eclipse launch configuration models.

With AM3, users can build their customized megamodeling solution by creating new plug-ins either extending the core plug-ins or other already existing extension plug-ins. This involves the creation of a corresponding metamodel extension as discussed before, and the provision of facilities for properly executing and manipulating the newly defined artifacts.
AM3 functionalities enable users to manipulate the underlying megamodel. They include repository-related actions such create/retrieve/update/delete (CRUD), but also the execution of executable artifacts (typically model transformations). Note that transformation execution involves the creation of a number of models (i.e., target models) and their registration to the megamodel.

Executability, in this context, introduces the notion of an execution error within a megamodel. One form of execution error is a type error. In a concrete project, a megamodel would refer to all involved artifacts, and the manipulation of such a megamodel would encompass the progress of project-specific processes. A megamodel-centric tool, such as AM3, should prevent the occurrence of execution errors in general, and type errors in particular, when its underlying megamodel is manipulated. Type systems are a standard means for preventing type errors. For this reason, typing is critical in GMM too. In the next section we review the basic concepts of type systems, and after that we discuss the current typing approach of GMM. Such presentations finally enable a precise definition of the problem addressed in this work.

2.4 Type Systems

The fundamental purpose of a type system is to prevent the occurrence of type errors during the execution of a program. The absence of type errors is a nontrivial property. When such a property holds for all of the program runs that can be expressed within a programming language, we say that the language is type sound. The formalization of type systems requires the development of precise notations and definitions, and the detailed proof of formal properties that give confidence in the appropriateness of the definitions. In this section we review the elements of formal type systems based on the comprehensive overview by Cardelli [Car97].

2.4.1 Typing

A program variable can assume a range of values during the execution of a program. An upper bound of such a range is called a type of the variable. Languages where variables can be given (nontrivial) types are called typed languages. A type system is that component of a typed language that keeps track of the types of variables and, in general, of the types of all expressions in a program. Type systems are used to determine whether programs are well-behaved. Only program sources that comply with a type system should be considered real programs of a typed language; the other sources should be discarded before they are run.

A definition of a type error depends on a specific language and type system, but it always includes the use of a function on arguments for which it is not defined and the attempted application of a non-function [WF94]. A program fragment is said to have
2.4 Type Systems

_good behavior_, or to be _well-behaved_, if it does not cause type errors during execution. A language in which all of the (legal) programs have good behavior is called _strongly checked_. Typed languages can enforce good behavior by performing static (i.e., compile time) checks to prevent ill behaved programs from ever running. Those kind of languages are _statically checked_. Such a checking process is called _typechecking_, and the algorithm that performs it is called the _typechecker_. A program that passes the typechecker is said to be _well-typed_; otherwise, it is _ill-typed_. Well typing is a static approximation (i.e., is a stronger notion) of good behavior; there could exist ill typed programs with good behavior. But the important thing is that well-typed programs do not go wrong.

Typed languages are not the only available option for assuring good behavior, and the choice between typed and untyped languages depends on several factors. However, typed languages exhibit a number of advantages over untyped languages. From an engineering point of view, arguments in favor of typed languages include:

- **Economy of execution.** Accurate type information at compile-time leads to the application of the appropriate operations at run-time without the need of expensive tests.

- **Economy of small-scale development.** When a type system is well-designed, type-checking can capture a large fraction of routine programming errors, eliminating lengthy debugging sessions.

- **Economy of language features.** In ML [Mil84], a procedure with a single argument that is a tuple of $n$ parameters models a procedure of $n$ arguments. Thus, type systems promote orthogonality of language features, question the utility of artificial restrictions, and thus tend to reduce the complexity of programming languages.

The above list is not intended to be exhaustive. It regards some of the benefits of typing in the context of our work. A more complete discussion on the advantages of typing and further applications of type systems may be found in [Pie02].

2.4.2 Formal Type Systems

Formal type systems are the mathematical characterizations of the informal type systems that are described in programming language manuals. More precisely, a (formal) type system is a _calculus_, or equivalently, a _formal system_, which classifies expressions in types. A calculus or formal system is an uninterpreted (i.e., no interpretation or semantics is provided) symbolic system which includes axioms and rules for carrying out deductions. Formality here means that deducibility is purely syntactical. This is due to a decidable syntax, which enables deciding whether an expression is a valid derivation in the system or not. In summary, a type system is a system which classifies expressions in types relying on syntax only. Once a type system is formalized, a _type soundness_ theorem can
be proved. To that end, a semantics must be defined. When a semantics is provided, a
formal type system becomes a formal language. A soundness theorem states that well-
typed programs are well-behaved. If such a soundness theorem holds, we say that the type
system is sound.

In order to formalize a type system and prove a soundness theorem we must in essence
formalize the whole language in question. The first step in formalizing a programming
language is to describe its syntax. The next step is to define the scoping rules of the
language, which unambiguously associate occurrences of identifiers to their binding loca-
tions. Scoping can be formally specified by defining the set of free variables of a
program fragment (which involves specifying how variables are bound by declarations). The
associated notion of substitution of types or terms for free variables can then be
defined.

Once the above elements are settled, one can proceed to define the type rules of the
language. These describe a has-type relation of the form $M : A$ between terms $M$ and
types $A$. Some languages also require a relation subtype-of of the form $A <: B$ between
types, and often a relation equal-type of the form $A = B$ of type equivalence. If the type
names $X$ and $Y$ match because they are associated to similar types, we have structural
equivalence. If they fail to match just because they are distinct type names (without
looking at the associated types), we have by-name equivalence. The collection of type
rules of a language forms its type system.

When the language include identifiers, the type rules cannot be formalized without
first introducing another fundamental ingredient that is not reflected in the syntax of the
language: static typing environments. These are used to record the types of free variables
during the processing of program fragments. The type rules are always formulated with
respect to a static environment for the fragment being typechecked. For example, the
has-type relation $M : A$ is associated with a static typing environment $\Gamma$ that contains
information about the free variables of $M$ and $A$. The relation is written in full as $\Gamma \vdash
M : A$, meaning that $M$ has type $A$ in environment $\Gamma$.

The final step in formalizing a language is to define its semantics as a has-value
relation between terms and a collection of results. The form of this relation strongly
depends on the style of semantics that is adopted. In any case, the semantics and the
type system of a language are interconnected: the types of a term and of its result should
be the same (or appropriately related); this is the essence of the soundness theorem.

In a given type system, a term $M$ is well-typed for an environment $\Gamma$, if there is a
type $A$ such that $\Gamma \vdash M : A$ is a valid judgment; that is, if the term $M$ can be given some
type. The discovery of a derivation (and hence of a type) for a term is called the type
inference problem. Type inference is closely related to typechecking, and in some cases,
one depends on the other. The type inference problem for a given term is very sensitive
to the type system in question. An algorithm for type inference may be very easy, very
hard, or impossible to find, depending on the type system.
2.5 Typing in GMM

Type annotations associated to a term can always be physically written down by a user or programmer, and depending on the type system, in occasions it could be inferred. A well-designed statically typed language will never require large amounts of type information to be explicitly maintained by the programmer. There is some disagreement, though, about how much explicit type information is too much [Pie02].

2.4.3 Meta-Theoretic Properties

A type system is a collection of type rules. If chosen arbitrarily, type rules would probably yield a useless type system. An effective type system should satisfy at least the following properties:

- **Type systems should be decidable.** The purpose of a type system is to actively capture type errors before they happen. It should be possible to decide whether a program is well-typed or not. The existence of a correct typechecking algorithm implies that the typechecking process is decidable.

- **Type systems should be sound.** Well typing is meant to correspond to a semantic notion of good program behavior. It is customary to check the internal consistency of a type system by proving a type soundness theorem. This is where type systems meet semantics. For denotational semantics we expect that if $\emptyset \vdash M : A$ is valid, then $[M] \in [A]$ holds (the value of $M$ belongs to the set of values denoted by the type $A$), and for operational semantics, we expect that if $\emptyset \vdash M : A$ and $M$ reduces to $M'$ then $\emptyset \vdash M' : A$. In both cases the type soundness theorem asserts that well-typed programs compute without execution errors.

Proving a soundness theorem is commonly a hard task. Proofs of more basic, however non-trivial properties, such as confluence, strong normalization and type preservation are usually required. Proofs of soundness rely on the style of the semantics of the language in question, but also on its features. A survey on the different approaches followed in proofs for different languages can be found in [WF94].

2.5 Typing in GMM

One possible way a user can manipulate a megamodel is by executing a transformation contained in the megamodel, on artifacts also contained in the megamodel. A transformation explicitly specifies its source (and target) metamodels and produces an execution error if it is applied to models not conforming to such source metamodels. Although typing artifacts is not widespread in MDE, this mechanism shares its intent with typing the arguments and result value of functions in programming languages. Typing artifacts
has a key role in preventing execution errors in megamodel manipulations. In this section we focus on the typing of artifacts in GMM. We discuss how typing is currently addressed, we identify a number of limitations of such an approach and illustrate the main issues with a concrete example. Such a discussion leads us to the precise definition of the problem addressed in this work, which is presented at the end of this chapter.

2.5.1 Current Typing Approach

GMM currently lacks a formal type system (its definition is the main purpose of our work), but also lacks an informal description of its typing approach. As a consequence, the information discussed in this section was mainly obtained from two different sources: GMM-related documentation [Mod09a, Mod09b] and the AM3 tool.

GMM-related documentation actually includes very little information on the typing approach. In fact, the following text is the only explicit reference to typing:

The simplest idea that comes to mind is to consider that all artifacts are models. Then we have a simple solution which consists in stating that each one is typed by its metamodel. We will mainly follow this conjecture in the present work that all managed artifacts are models, conforming to a precise metamodel. [Mod09a, pp. 8-9]

According to the above description, every artifact in GMM is a model. This is not completely accurate since textual entities are manageable artifacts but they are not models. In addition, every artifact is said to be typed by its metamodel, or should we say, by its reference model. Such an approach is aligned with the specific work on this subject by Steel [SJ07]. In this way, the has-type relation is identified with the conformsTo, or c2, relationship. We may call this typing relation type-by-reference model and denote it as ‘:\c2’. Thus, for model m and reference model M, we have that \(m:\c2 M\) if and only if \((m,M) \in \text{conformsTo}\). Note that this definition does still not apply to textual entities. However, this approach fails when some models (e.g., transformations) are also executable entities. This is simply because the type of a transformation model is the reference model it conforms to (e.g., ATL), which says nothing about the source and target reference models. As a consequence, from the type of a transformation it is not possible to know the type of the models it accepts or produces. This prevents a megamodel user from executing even the simplest transformation in a typed context.

From an observation of AM3’s behavior, we find that, in simple cases, the situation described above does not occur, since the elements resulting from the execution of a transformation are correctly typed. Therefore, we can conclude that the actual typing approach goes beyond the pure type-by-reference model idea. In fact, by examining the metamodels defining GMM, the real approach comes to surface. As discussed in a previous section, some artifacts have a dual representation in GMM: as an entity and as a
relationship. This is particularly the case for transformations, where a transformation is represented as one transformation model (entity) and as one model transformation (relationship). From this perspective, we find useful the following fragment of documentation:

> All these relationship types [model transformation is one of such relationship types] act as abstract specifications of transformations. They may be linked with transformation models which provide concrete implementations of these specifications. [Mod09b, p. 17]

A transformation \( t \) is then represented by a pair \((t_{tm}, t_{mt})\), where \( t_{tm} \)'s metadata includes its type through \( c_2 \) and \( t_{mt} \)'s metadata includes its source and target reference models. In this way, such \( t_{mt} \)'s metadata can be regarded as a sort of signature for \( t \) and is thus more appropriate for being considered as its type. However, this form of typing is only applied when a transformation is used as an operation (i.e., when it is executed). When used as a value, a transformation is typed as a “regular model” (i.e., following type-by-reference model). As a consequence, transformations are dually typed. Moreover, transformation arguments and results are also typed as regular models. In conclusion, the typing approach of GMM is implicitly based on the following scheme:

- Artifacts in general, excluding textual ones, are considered as models and are thus typed through the type-by-reference model relation.
- Executable artifacts (i.e., transformations) are considered as relationships and are typed by a subset of their metadata when being applied. Additionally, arguments and results are typed as models as in the item above.
- Textual artifacts are considered as textual entities and they are all of the same type, regardless of their internal structure.

As mentioned above, this approach (i.e., type-by-reference model extended with the use of metadata) suffices for handling most of the common situations in megamodelling. However, the AM3 implementation does not currently exploit all the available information. The example in Figure 2.9 indicates the existence of a bug, but more importantly, it shows that typing has not been granted a primary role. In the editor pane, the Execution (History) tab allows selecting the actual parameters for the Families2Persons transformation [ATL09]. At the top area, 0 denotes a source formal parameter of type Families and 1 denotes a target formal parameter of type Persons. At the center area, both actual parameters are not set. By clicking the corresponding Set button, the Model Element Selection Dialog shows and any model can be selected, even ECore which is a metametamodel. In such a case, after clicking the Execute button an execution error should clearly occur.
2.5 Typing in GMM

In addition to that, even conceptually simple cases may lead to highly complex typing problems which the current approach fails to handle properly. Next, we discuss in detail such cases where the result of a transformation cannot be typed. Additionally, we also discuss cases where stronger (and necessary) checks cannot be performed.

2.5.2 Limitations

The limitations of GMM’s original typing approach are in essence the inability to express type information for some specific kinds of artifacts. In some critical cases, missing information plainly leads to untyped artifacts. In a model management context, an untyped artifact is of little use. This is simply because we would not know what actions on such an artifact are safe or unsafe (e.g., no type matching can be performed in order to safely use the artifact as an actual parameter for a given transformation). The use of
HOTs or DTTs, introduced in a previous section, may cause such situations [VJBB09]. In others cases, richer type information would enable stronger checks which could prevent unexpected results and even execution errors. Artifacts involved in such situations are textual entities and projectors [Vig09c], and weaving models [VB09b]. In what follows we discuss all of these cases.

2.5.2.1 Higher-Order Transformations

Consider a higher-order transformation \( H \) such that when applied to some model \( s \) produces another transformation \( t \) (i.e., \( H(s) = t \)). According to the current typing approach, both \( s \) and \( t \) are considered as models and typed as such, even though \( t \) is actually a transformation. We may assume that the \( H_{mt} \) component indicates that \( H \)'s source reference model is \( S \) and its target metamodel is \( T \). As a consequence, \( s \) must conform to \( S \) (i.e., \( s : c_2 S \)) and \( t \) must conform to metamodel \( T \) (i.e., \( t : c_2 T \)), where \( T \) defines the transformation language in which \( t \) is expressed. The problem is that \( t \) ends up partially typed. This is because we have no means to infer the metadata of the \( t_{mt} \) component concerning \( t \)'s source and target reference models. Such information is mandatory for checking an application of the form \( t(x) = y \). First, we have to make sure \( x \) is of the right type, and second, we must infer a type for \( y \). None of such things are possible. Even if metadata included in \( H_{mt} \) is regarded as a signature or function type for \( H \), its argument and result are still typed as values, but at least result \( t \) is a function.

As a remark, this typing approach presents an interesting feature though: some form of genericity is enabled. If metamodel \( S \) is a transformation metamodel, just like \( T \) is, then \( H \) accepts any transformation expressed in the language defined by \( S \), regardless of the number and type of their source and target elements. This capability is something we would like to preserve.

2.5.2.2 Dependently Typed Transformations

Another situation where the current typing approach may lead to a loss of type information is when a transformation \( D \) operates on metamodels, or more precisely, on reference models. In fact, the type of any reference model \( R \) accepted by \( D \) is necessarily a metametamodel \( M \) (i.e., \( R : c_2 M \)). Then, from the metadata included in the \( D_{mt} \) component it is possible to know that a reference model conforming to \( M \) is involved, but not that it is \( R \). If \( R \) occurs in the target type of \( D \) (this may happen because \( R \) is itself a type), then it is not possible to express such a target type. The problem is that GMM is actually dependently typed (i.e., types may depend on values) but the current typing approach is not. In the next subsection we discuss a conceptually simple and well-known transformation which simultaneously illustrates both of the issues presented so far.
2.5.2.3 Textual Entities

Now we address the cases where richer type information would enable stronger checks. As discussed before, textual artifacts are typed as textual entities. As in GMM, textual entities are not models (see Figure 2.6), they do not have a conformsTo relationship. Even though metadata associated to textual entities may include some useful information (e.g., a Uniform Type Identifier (UTI) [Mac09]), from a practical typing perspective, all textual entities are of the same type, say Text. A textual entity is an element \( t \) within a megamodel which is a textual representation of some model \( m \). In particular, \( t \) conforms to some grammar \( G \), which is related to a reference model \( A \), such that model \( m \) conforms to \( A \). If \( m \) existed beforehand, \( t \) is normally the extraction of \( m \), otherwise, \( m \) is an injection of \( t \). In the former scenario, the extractor goes from \( A \) to \( Text \) (a pretty printer), and in the latter scenario, the injector goes from \( Text \) to \( A \) (a parser). On the one hand, the type approach prevents such an extractor from classifying its result in a more precise way. On the other hand, any textual entity can be passed to such and injector, even a wrong one, and thus the execution of the injector may fail. The problem is that the way textual entities are typed affects how extractors are typed, and may enable injectors to cause execution errors.

2.5.2.4 Weaving Models

A weaving model contains links to other models and conforms to a weaving metamodel. A weaving metamodel is just a regular metamodel (GMM does not support a specific variant of metamodels for this purpose) specifying the semantics of the links. If weaving model \( m_w \) conforms to weaving metamodel \( M_w \) (i.e., \( m_w :c2 M_w \)) the following may occur. For simplicity, let us assume that models woven by \( m_w \) are \( m_a \) and \( m_b \), such that \( m_a :c2 M_a \) and \( m_b :c2 M_b \). Now let us assume another weaving model \( m_w' \) which conforms to \( M_w \) as well (i.e., \( m_w' :c2 M_w \)), but linking elements within \( m_c :c2 M_c \) and \( m_d :c2 M_d \). The current typing approach says that \( m_w :c2 M_w \) and \( m_w' :c2 M_w \). In other words, both weaving models would appear to have the same type. Although they are similar, both weaving models are not exactly of the same nature, since one links models of types \( M_a \) and \( M_b \) while the other links models of types \( M_c \) and \( M_d \). A transformation accepting a weaving model conforming to \( M_w \), but relying on \( M_a \) and \( M_b \), then accepts both \( m_w \) and \( m_w' \) and may fail to execute in the second case.

Additionally, a subsequent usage of a weaving model may lead to an element whose type could depend on the type of the woven models. For example, a transformation \( T \) operating on model \( m_w \) may produce an element \( e \) such that its type depends either on \( M_a \) or \( M_b \) (i.e., the woven types). Moreover, since \( m_a \) and \( m_b \) (i.e., the woven models) could be types themselves (e.g., metamodels), the type of element \( e \) could even depend on \( m_a \) and \( m_b \). Since this information is not captured in the type of \( m_w \), in some situations
Similarly to transformations, weaving artifacts have a dual representation in GMM. A weaving artifact $w$ is represented by $\langle w_{\text{wm}}, w_{\text{mw}} \rangle$. Weaving model $w_{\text{wm}}$ is typed through $:_{c2}$ by a weaving metamodel, and model weaving relationship $w_{\text{mw}}$’s metadata refers to the reference models involved in the weaving (see Chapter 3 for more details). This means that the required type information is actually available in GMM, but not used for typing purposes. Therefore, the problem is that the way weaving artifacts are currently typed, prevents stronger checks when they are passed to a transformation, or even typing transformations involving them.

### 2.5.3 Example

For concluding our presentation on the limitations the current typing approach, we use a concrete example, which is a very simple transformation case, but simultaneously illustrates the complex issues discussed for HOTs and DTTs. The $\text{KM32ATLCopier}$ transformation [ATL09], defined in ATL, receives a reference model $M$ conforming to KM3 and produces an ATL identity transformation (called a copier). Such a copier is specifically applicable to models conforming to $M$. Such a transformation produces a transformation, and thus it is a HOT, and the type of the resulting copier transformation clearly depends on $M$, and thus it is a DTT.

The type of $\text{KM32ATLCopier}$ as an entity (i.e., the $\text{KM32ATLCopier}_{\text{tm}}$ component) is just the metamodel the entity conforms to:

\[
\text{KM32ATLCopier} :_{c2} \text{ATL}
\]  

(T$_e$)

In turn, the type-related information from the metadata of $\text{KM32ATLCopier}$ as a relationship (i.e., from the $\text{KM32ATLCopier}_{\text{mt}}$ component), may be extracted from the header of its ATL definition:

\[
\text{create OUT} : \text{ATL} \text{ from IN} : \text{KM3}
\]

The transformation receives a $\text{KM3}$ reference model and produces an $\text{ATL}$ transformation (see Figure 2.8). Using for clarity the usual notation for function types, and for this purpose introducing the alternate $\text{has-type}$ relation denoted as $:_{md}$, this type information can be expressed as:

\[
\text{KM32ATLCopier} :_{md} \text{KM3} \rightarrow \text{ATL}
\]  

(T$_r$)

Figure 2.10 illustrates the artifacts involved in a sample application of the $\text{KM32ATLCopier}$ transformation model. The $c2$ relationship from $\text{KM32ATLCopier}$ to $\text{ATL}$ denotes type
Figure 2.10: The KM32ATLCopier transformation applied to SQL produces SQLCopier. Its source and target reference models (colored in red) cannot be inferred with the current typing approach.

(T_e) above. In turn, the sourceRM and targetRM relationships from KM32ATLCopier to KM3 and ATL respectively denote type (T_r). Assuming that an application of transformation KM32ATLCopier on (meta) model SQL produces as a result (transformation) model SQLCopier, the available type information, that is, (T_e) and (T_r) should enable us to determine two separate things. First, we should be able to decide if SQL is an appropriate parameter for KM32ATLCopier, and second, we should be able to infer at least type information (T'_e) and (T'_r) for SQLCopier. Since we know that:

\[ SQL :_{c2} KM3 \]

from (T_r) we positively know that SQL is an appropriate parameter for KM32ATLCopier. However, when it comes to typing SQLCopier the situation is different. From (T_r) again we can conclude:

\[ SQLCopier :_{c2} ATL \]  \( (T'_e) \)

This tells us that SQLCopier is a transformation and that it is defined in ATL. But nothing in the combination of (T_e) and (T_r) provides information for determining (T'_r). In Figure 2.10, sourceRM and targetRM for SQLCopier remain undefined, even though we
know that both of them should refer to SQL, which is the parameter of the application. As a consequence, we have:

\[
\text{SQLCopier} : \text{md} ? \rightarrow ?
\]

This shows that \((T_e)\) and \((T_r)\) are not sufficient for inferring both \((T'_e)\) and \((T'_r)\). In fact, \((T'_e)\) could be inferred, but \((T'_r)\) could not. Note that \((T_e)\) has no relevant information concerning the check and the inference performed above (in Chapter 4 we will show its relevance). In turn, if a first-order transformation such as Class2Relational [ATL09] was applied instead, information like \((T_r)\) alone would have sufficed. This means that \((T_r)\) is critical for these purposes. It seems that if the co-domain of \((T_r)\) was a function type, as \((T_r)\) itself is, then \((T'_r)\) could have been inferred. However, note that the original co-domain of \((T_r)\) was used for inferring \((T'_e)\), thus \((T_r)\) cannot be simply modified without introducing another loss of type information.

To summarize, \((T'_r)\) could not be inferred, and as a consequence the newly created SQLCopier transformation cannot be safely used. First, we cannot decide whether a given model is an appropriate parameter for SQLCopier, and second, we cannot infer a type for the result of such an application. Under these circumstances, GMM lacks a solid typing approach. A direct effect of this issue is that the AM3 tool cannot provide its users the necessary means to prevent the occurrence of type errors during their manipulation of its implemented megamodel.

Based on the concepts and discussions presented so far, we close this chapter with a concrete statement of the problem addressed in this work.

### 2.6 Problem Statement

The main motivation of our work is to define a formal static type system for Global Model Management. This fits in the ultimate and larger objective of realizing its practical benefits within the AM3 tool. To that end, the type system needs to be implemented and integrated with such a tool.

A megamodel represents a working zone by containing (references to) all artifacts involved in such a working zone. The contents of a megamodel then evolves encompassing processes occurring in that working zone. Such an evolution may be expressed in terms of a sequence of some primitive actions performed by a user on the megamodel. One form of primitive actions on megamodels is CRUD operations with respect to the elements they may contain. Another form of primitive actions is the execution of transformations, which produce new artifacts as a result. Sequences of such actions interactively performed by megamodel users may be regarded as programs of a basic megamodel-based programming language. The purpose of our work is to provide a type system which prevents the
occurrence of type errors, and in particular avoids the problems identified in this chapter. Then the benefits we refer to are an improved megamodelling environment with enhanced safety, reliability and usability. The reason for choosing a formal approach for defining the type system is twofold. First, a formal definition enables the proof of critical properties of the type system with a high degree of confidence. Second, a formal definition is a valuable specification in itself. But most importantly, as type systems are intended for being used in practice, an implementation for them is expected, and a formal specification is an even more valuable asset for that matter. As Cardelli pointed out, informal language descriptions often fail to specify the type structure of a language in sufficient detail to allow unambiguous implementations \cite{Car97}.

The task at hand represents a major challenge. Regarding type system definitions, Pierce observes that retrofitting a type system onto a language not designed with type-checking in mind can be tricky; ideally, language design should go hand-in-hand with type system design \cite{Pie02}. Even though GMM considered a form of typing, the lack of thorough typechecking mechanisms for advanced constructs (e.g., HOTs and DTTs) is apparent. According to Pierce, languages without type systems tend to offer features that make typechecking difficult or even unfeasible. As an example of this issue we can mention the dual representation of weaving artifacts, and especially, transformations.

In what follows we provide an account of the main issues with varying degrees of difficulty that our type system should take into consideration:

- The set of managed metametamodels should be open. This means that a user should be allowed to register and use (e.g., defining metamodels conforming to them) as many metametamodels as desired.
- Circularity of metametamodel definition should be directly supported. That is, a metametamodel should be typed by itself.
- The type system should support user-defined metamodels and their corresponding terminal models, but also artifacts from the metamodel extensions discussed in this chapter.
- Higher-order transformations should be supported, and their related issues which were discussed in Section 2.5.2.1 should be addressed.
- The form of genericity described in Section 2.5.2.1 should be preserved.
- Dependent-typed transformations should be supported, and the issues discussed in Section 2.5.2.2 should be addressed.
- The typing of textual entities should be revised for avoiding the problems discussed in Section 2.5.2.3. As a consequence, the typing of model-to-text and text-to-model transformations, and particularly of TCS artifacts, should also be revised.
• The typing of weaving artifacts, and particularly of AMW artifacts, should be revised for avoiding the problems discussed in Section 2.5.2.4.

• Subtyping should be supported for metamodels or other types when applicable and possible.

• A typechecking mechanism for Composite Transformations (i.e., checking the composition of the individual transformations included in the composite) should be defined.

• Elements contained in a megamodel can be either (a) created outside the scope of the megamodel and then registered to it, or (b) created and registered as the result of an action within the megamodel. The types of elements corresponding to (a) should be provided by the user, while the types of elements corresponding to (b) should be inferred.

• The type system should be extensible as GMM itself is. That is, it should be able to be extended for encompassing further GMM extensions (e.g. for a currently unsupported transformation language) without a complete redesign.

• Last but not least, the type system should be sound and decidable.
In this context, our contribution is a calculus, named \texttt{cGMM} and presented in Chapter 4, which defines a formal type system for Global Model Management. Since \texttt{cGMM} is dedicated to Megamodelling, we regard it as a type of domain-specific calculus, in contrast with “general-purpose calculi” such as \texttt{pCIC}, the underlying formal language of \texttt{Coq} [CPT09]. In Chapter 5, when dealing with soundness, we introduce a semantics for our formal system. In conjunction, such a formal system and its semantics result in a formal language: the megamodel-based programming language mentioned above.

If a Type Theory (TT) technical space can be conceived, then we are bridging the MDE and TT technical spaces. This alternate view of our work is illustrated in Figure 2.11. A megamodel containing elements is a terminal model which is projected to an environment containing \texttt{cGMM} terms. Megamodelling and \texttt{cGMM} would thus be the languages involved in the projection. In this context, the primary operation within the TT technical space is typechecking. Once typechecking is performed, either a term (which is actually a type) or a type error are projected back to the MDE technical space. Megamodels live in the technical space they belong, and typing is delegated to the technical space, or more specifically, to the village [Val08] where it is efficiently carried out.
Chapter 3

Global Model Management
Extensions

The metamodel of Global Model Management defines how megamodelling is approached. It introduces a classification of manageable elements and the notion of a megamodel as a container of such elements and relations among them. More specifically regarding the definition of elements, the metamodel describes information associated to such elements, and specifies how they are represented. From our specific point of view, the associated information, which includes type information, enabled the identification of the limitations discussed in Section 2.5.2 and the structure of elements drives the definition of the type system that will be presented in Chapter 4. The purpose of this chapter is to dive into some of the details of GMM’s metamodel from this perspective. As discussed in Section 2.3.2, the metamodel is modularized and the AM3Core metamodel and the GlobalModelManagement extension were addressed in Section 2.3.3.

Figure 2.6 of Section 2.3.3 shows that entities and relationships are specializations of general GMM elements. In turn, entities are further specialized, where several kinds of models are among possible variants. Relationships include information which is relevant to the typing of entities. However, variants of relationships were not illustrated in Figure 2.6. In Figure 3.1 specializations of GMM relationships introduced by the GlobalModelManagement extension are shown. Such variants, together with the specializations of TerminalModel of Figure 2.6, will occur in the metamodel extensions discussed in this chapter. IdentifiedElement is a subclass of Element which is the actual superclass of Entity and Relationship, and in our context, an identified element can be simply regarded as an element. A Relationship then involves a collection of identified elements. A relationship may be unidirectional and is represented as a DirectedRelationship. Such a relationship is a regular binary relation, by means of its source and target, where both components can be n-ary cartesian products. These relations usually denote n-to-m
functions (e.g., transformations). Additionally, a symmetric relation is represented as a ModelWeaving. The subsets constraint in Figure 3.1 indicates that for both DirectedRelationship and ModelWeaving, the inherited association from Relationship does not actually introduce new information for them (i.e., the association which is the source of the constraint can be regarded as a refinement of the association which is the target). Classes at the bottom of Figure 3.1 represent all variants of transformations, and their names explain their intent. Model stands for an element within the MDE technical space, while External stands for an element within a foreign technical space. For example, an ExternalToModelTransformation is an injector, while ModelToExternalTransformation is an extractor.
3.1 The AtlanMod Transformation Language Extension

The AtlanMod Transformation Language Extension (GMM4ATL) introduces a number of constructs specifically intended for representing and executing ATL artifacts. Such constructs are defined in the metamodel.
extension illustrated in Figure 3.3. The main constructs introduced by such an extension are ATLModel, a specific kind of terminal model (i.e., an entity), and ATLTransformation, a specific kind of model-to-model transformation (i.e., a relationship). Such constructs inherit the association between TransformationModel and Transformation of Figure 3.2, and are dual representations of ATL artifacts.

An ATLModel can be either an ATLModule or an ATLLibrary. An ATLModule represents an ATL module which is a transformation artifact that can be independently executed. An ATL module contains a set of transformation rules and possibly helpers. However, such a module is treated as a black box and thus its contents is not represented in megamodels. For this reason the metamodel extension of Figure 3.3 does not include constructs for notions like rule or helper, unlike the ATL metamodel. ATL modules have input (sourceParameters) and output (targetParameters) parameters which are specified in the header of the module. For example, considering the header of module KM32ATLCopier discussed in Section 2.5.3:

```plaintext
create OUT : ATL from IN : KM3
```

induces two instances of ATLParameter which are associated to the ATLModule representing KM32ATLCopier. The first has its formalModelName attribute set to IN and its formalReferenceModelName set to KM3, and is the only parameter in the sourceParameters collection of the module. Analogously, the second has its formalModelName attribute
set to OUT and its formalReferenceModelName attribute set to ATL, while being the only parameter in the targetParameters collection. An ATLLibrary represents an ATL library which is an artifact containing helpers intended to be used by other ATL models. An ATL library cannot be independently executed. ATL libraries are not addressed in this work and their case is discussed at the end of this chapter.

If an ATL module can be regarded as an operation, then its associated ATLTransformation can be regarded as its signature. Such a signature has a number of formal parameters (ATLTfParameter, a subclass of TransformationParameter) which not only represent input parameters but also results (the association between ATLTransformation and ATLTfParameter subsets the association between Transformation and TransformationParameter shown in Figure 3.2). For executing a transformation, such formal parameters are bound to the corresponding module’s parameters. The type of a formal parameter is specified by the association to ReferenceModel that ATLTfParameter inherits from ModelParameter (not shown in Figure 3.3). This implies that ATL modules may accept or produce models (i.e., any variant of terminal models, but also metamodels and metametamodels), thus enabling dependent types. Furthermore, the limitation identified in the previous chapter directly relies on this structure as it was used for determining type (T_r) in Section 2.5.3.

To summarize, an ATL module is represented by an ATLModule and requires an ATLTransformation for its execution. An ATL module can be understood as an operation which accepts a non-empty set of models and produces a non-empty set of models. In both occurrences, by model we mean an artifact represented by any variant of the Model construct.

3.2 The Textual Concrete Syntax Extension

The GMM4TCS extension introduces a number of constructs specifically intended for representing and executing TCS artifacts. Such constructs are defined in the metamodel extension illustrated in Figure 3.4. The main constructs introduced by such an extension are TextualEntity, a specific kind of entity for representing textual entities, TCSModel which is a specific kind of terminal model (i.e., an entity), and TCSInjection and TCSExtraction, which are specific kinds of external-to-model and model-to-external transformations (i.e., relationships).

A TCSModel represents an executable TCS projector. A TCS projector contains a set of templates. Like ATL modules described before, a projector is treated as a black box and the notion of a template is not included in the metamodel extension of Figure 3.4. TCS projectors always involve a terminal model m conforming to some metamodel MM, and a textual entity t representing m, which conforms to some grammar G associated to MM. For such a projector, its referringMM is in fact MM.

As with ATL artifacts, TCS projectors have dual representations in GMM. A TC-
3.3 The Composite Transformations Extension

The GMM4CT extension introduces a number of constructs specifically intended for defining and executing composite transformations. Such constructs are defined in the meta-

Figure 3.4: Partial metamodel of GMM4TCS extension

SModel, seen as an operation, is associated to a TCSInjection or a TCSExtraction depending on its execution direction. Both TCSInjection and TCSExtraction relationships are ExternalTransformation and ModelTransformation at the same time (see Figure 3.1). From ExternalTransformation they inherit an ExternalParameter to be bound to a textual entity such as \( t \), and from ModelTransformation they inherit a ModelParameter to be bound to a terminal model such as \( m \). For a TCSInjection, the external parameter is the source parameter and the model parameter is the target parameter. Conversely, for a TCSExtraction, the model parameter is the source parameter and the external parameter is the target parameter. Unlike ATL artifacts, the (function) type of TCS projectors is not explicitly represented. If such a type was to be expressed, it would have the form \( MM \rightarrow \text{TextualEntity} \) or \( \text{TextualEntity} \rightarrow MM \), depending on whether the projector is seen as an extractor or as an injector, provided that \( MM \) is the referringMM metamodel associated to the projector (note that TextualEntity is fixed for all cases and independent from metamodel \( MM \) and thus independent from grammar \( G \)). This is the source of the limitations identified in the previous chapter.

To summarize, a TCS projector is represented as a TCSModel. For being executed as a model-to-external transformation, it is associated to a TCSExtraction relationship, and for being executed as an external-to-model transformation, it is associated to a TCSInjection relationship. A TCS projector can be understood as an operation which transforms a terminal model to its textual representation back and forth.

3.3 The Composite Transformations Extension

The GMM4CT extension introduces a number of constructs specifically intended for defining and executing composite transformations. Such constructs are defined in the meta-
model extension illustrated in Figure 3.5. The main construct introduced by such an extension is CompositeTransformation which is a specific kind of transformation (i.e., a relationship). Composite transformations exhibit some interesting differences with the transformations discussed so far. First, composite transformations are not treated as black boxes in GMM and thus their contained elements are indeed described in the metamodel extension shown in Figure 3.5. Second, composite transformation are not dually represented, and more specifically they are represented as relationships and not as models. An immediate consequence of this situation is that composite transformations cannot be used as the source or the target of other transformations.

By being a Transformation, a CompositeTransformation has parameters (cParameters) which in this context are specifically CompositeParameter elements (see the subset constraint in Figure 3.5). A composite transformation additionally has connectors among the sub-transformations which configure the concrete composition, and calls to such sub-transformations. A Connector has a number of connector ends (ConnectorEnd). A SimpleConnector is a binary connector, while a MultiConnector has several ends and enables
3.4 The AtlanMod Model Weaver Extension

connecting a sub-transformation to many others (either for receiving multiple inputs or for delivering its output to multiple sub-transformations). Although restrictions apply (e.g., a sub-transformation may not be fed back with its own output), this structure enables compositions similarly structured as Pipe and Filters architectures [BMR+96]. A DirectConnector is used for directly referencing entities and it is thus intended for defining composite parameters (CompositeParameter is a subclass of ConnectorEnd). A TransformationCall represents a call to a sub-transformation and thus refers to a Transformation. Additionally, a transformation call refers to the parameters of such a transformation by means of parameter references (ParamRef). Transformation calls relate to connected sub-transformations by making ParamRef a variant of ConnectorEnd.

From an execution perspective, a concrete application of a composite transformation to actual parameters is represented by a CompositeTransformationRecord. In turn, such a record contains the transformation record associated to each application of sub-transformations. The binding to actual parameters of sub-transformations is handled similarly as in Figure 3.2: a TransformationParameterRefValue inherits from TransformationParameterValue its association to Entity (i.e., the value), and is also associated to a ParamRef.

To summarize, a composite transformation, including its definition, is represented by a CompositeTransformation relationship. Such a definition is a functional composition of other transformations (i.e., any variant of Transformation). Note that composite transformation parameters (both source and target) are entities. This means that even if a transformation model (which is executable) is received, then it cannot be used as a sub-transformation since it is not a variant of the Transformation relationship. In addition, either models or textual entities are accepted or produced by a composite transformation. Therefore, a composite transformation, depending on its definition, could be either classified as model-to-model, model-to-external, external-to-model, or even as external-to-external. Finally, composite transformations do not support recursion. As a consequence, a composite transformation may not include itself as a sub-transformation.

3.4 The AtlanMod Model Weaver Extension

The GMM4AMW extension introduces a number of constructs specifically intended for representing AMW artifacts. Such constructs are defined in the metamodel extension illustrated in Figure 3.6. The main constructs introduced by such an extension are AMWModel, which is a variant of a weaving model (i.e., an entity), and AMWWWeaving, which is a variant of model weaving (i.e., a relationship). Such constructs are related by the association between WeavingModel and ModelWeaving, and are dual representations of AMW artifacts.

An AMWModel represents an AMW weaving model relating a number of models. An
Figure 3.6: Partial metamodel of GMM4AMW extension

AMW weaving model contains links among model elements within the woven models. Again, such a weaving model is treated as a black box and its contained links are not represented in a megamodel and thus are not specified in the metamodel extension. Woven models are specified by name, similarly to parameters of an ATLModule, in the modelParameters attribute of AMWModel. Even though a weaving model is not an executable operation, an AMWWaving can still be regarded as its signature. Each woven model is associated to an AMWWaving through a ModelParameterBinding element. In turn, the relatedModels collection refer to the types of the woven models. Each of those types is a Model; we would have expected a ReferenceModel there instead.

To summarize, an AMW weaving model is represented by an AMWModel entity. The binding of such a weaving model to woven models, together with their expected types, is specified by its associated AMWWaving relationship.

3.5 Discussion

The analysis presented in this chapter is based on the prototypical implementation of the metamodel extensions [AM309] and on documentation related to GMM [Mod09b]. However, such documentation has very little information concerning the typing approach (e.g., how type checks are concretely performed), and it does not address the GMM4AMW metamodel extension.

In this chapter we did not discuss the GMM4ASM extension. This is because such an extension does not introduce any constructs for representing artifacts. Rather, it introduces a number of concepts which are required for using the ATL virtual machine as a transformation execution engine. In particular, it provides the ParameterBinding and ModelParameterBinding constructs enabling parameter binding used in GMM4ATL and GMM4AMW extensions.

ATL libraries are tools for modularizing ATL transformations by grouping together ATL helpers which can be reused in other libraries or modules, or at least help reducing
the size of ATL modules. However, only a rather small amount of transformations within the ATL Transformation Zoo [ATL09] use such a tool. In [VB09a] we discussed ATL libraries and even proposed a typing approach for them which is completely compatible with our present work. In such a discussion, we argued that a library should not be considered as a transformation (TransformationModel is an ancestor of ATLLibrary; see Figure 3.3) because it contains only helpers and not transformation rules. Interestingly, in AM3 it is not possible to relate an ATLLibrary to any variant of Transformation, even though an ATLLibrary is currently a legal TransformationModel and as such it is associated to Transformation (see Figure 3.2). After the publication of our work we were informed\(^1\) by the member of the AtlanMod team in charge of the GMM project that the approach to ATL libraries will undergo a deep modification process. For this reason we excluded our results on the subject from the present work.

\(^1\)Private communication.
Chapter 4

A Type System for Megamodeling

In this chapter, we introduce a formal type system for Global Model Management. It is presented as a formal calculus, called cGMM, which is a symbolic system that classifies its terms in types. We start by formalizing the syntax of such terms in Section 4.1. GMM’s original type approach assumes that the only types are metamodels. However, taking into account the conformsTo relationship it is apparent that metametamodels are types as well. Furthermore, our discussion in Section 2.5.2 suggests that other types are also required. In Section 4.2 we introduce a type for all cGMM types. Such a type is further partitioned into a number of subtypes, that are organized into a type hierarchy, that specifically type each kind of cGMM’s types: metamodels, metametamodels, transformation types, textual types and so on. Classification of terms in types is deduced by means of rules and axioms that describe the typing and subtyping relations. Such axioms and rules are type rules which are presented in Section 4.2. Using the syntax we can determine if an expression is a term of the calculus (well-formedness), and using type rules we can deduce a type for it (well-typing). In the next chapter we address the problem of showing that well-typing is decidable and that it implies good behavior of the associated term. The GMM approach is intended to be extended as new kinds of artifacts are to be managed. Our calculus was designed with this issue in mind, and how to deal with extensions is discussed in Section 4.3. We close this chapter with a discussion of related work.

4.1 Terms

Every cGMM term has a type. Unlike most type theories, we do not make a syntactic distinction between types and terms because the type-theory itself forces terms and types to be defined in a mutually recursive way. We therefore define both types and terms in the same syntactical structure. We denote by type the semantic subclass of types inside
the syntactic class term.

**Definition 4.1.1 (sorts)** Types in cGMM are typed objects, and the type of a type is a constant called a sort. Sorts are also called type universes or kinds. The set of sorts of cGMM is (initially) defined by: \( S_0 \equiv \{ \text{Type} \} \).

Type plays the role of the type of all types. In general, set \( S \), the working set of sorts of cGMM, is an extensible set, and is expected to be extended in the future with new sorts for encompassing further extensions to GMM. Later on, set \( S \) will be defined by augmenting \( S_0 \) with a number of other predefined sorts which will be used for supporting current GMM extensions. Such sorts will be the type of “some” types, and their relation to Type will be detailed.

Objects in cGMM are manipulated as constants declared (i.e., either assumed or defined) in an environment. This approach is borrowed from pCic and nicely fits GMM’s context. Such constants then represent objects which could be reused in the declaration of other objects. The environment containing the declaration of a constant \( c \) is responsible for providing the information about the object \( c \) refers to.

**Definition 4.1.2 (terms)** Terms are inductively defined by the following clauses:

1. The sorts in \( S \) are terms;
2. Names for the constants in the environment (see definition 4.2.1) are terms;
3. Variables are terms;
4. If \( x \) is a variable and \( M_1 \) and \( M_2 \) are terms then \( \Pi x: M_1. M_2 \) and \( \Pi_\alpha x: M_1. M_2 \) are terms, where \( \alpha \in \{ \text{ATL}, \text{CT}, \text{TCS}, \text{EXE} \} \). These terms are dependent products. The non indexed variant denotes a classical dependent product representing a parametric type. The indexed variant denotes a functional product, either dependent or non dependent, representing function types;
5. If \( x \) is a variable and \( M_1 \) and \( M_2 \) are terms then \( \Lambda x: M_1. M_2 \) and \( \lambda_{\text{CT}} x: M_1. M_2 \) are terms. These terms are abstractions. The \( \Lambda \)-based variant is used for parameterizing types. The \( \lambda \)-based variant is used for defining functions which map elements of \( M_1 \) to \( M_2 \);
6. If \( M_1 \) and \( M_2 \) are terms then \( (M_1 \ M_2) \) is a term. These terms are applications and both denote type instantiations and functional applications;
7. If \( M_1 \) and \( M_2 \) are terms then \( M_1 \times M_2 \) is a term. These terms are cartesian products, which are a non dependent form of \( \Sigma \)-types;
4.2 Type Rules

8. If $M_1$ and $M_2$ are terms then $\langle M_1, M_2 \rangle$ is a term. These terms are pairs;

9. If $M$ is a term then $\pi_1(M)$ and $\pi_2(M)$ are terms. These terms are projections of the first and second component of pairs respectively.

10. If $M$ is a term then $\text{Text}(M)$ is a term. These terms are used for typing textual entities;

11. If $M, N_1, \ldots, N_n, M_1, \ldots, M_n$ are terms then $\text{AMWWieving}(M, N_1::M_1, \ldots, N_n::M_n)$ is a term. These terms are used for typing AMW weaving models.

We use $T$ to denote the set of all terms.

The notion of free variables is defined in the usual way. In $\Pi x: M_1, M_2, \Pi_\alpha x: M_1, M_2, \Lambda x: M_1, M_2$ and $\lambda_{\text{CT}} x: M_1, M_2$, the occurrences of $x$ in $M_2$ are bound. $\text{FV}(M)$ denotes the set of free variables of term $M$. In turn, the substitution of term $M_1$ to free occurrences of variable $x$ in a term $M_2$ is denoted by $M_2\{x/M_1\}$. Terms which are the same up to $\alpha$-conversion, that is, they are the same term except for a change in bound variables, are considered as identical. In what follows, we assume that $\alpha$-conversion is automatically performed when required.

Remark As in GMM, type equivalence in cGMM is by-name. This means that two types are equivalent when they are referred to by the same constant. As a consequence, type equivalence is decidable. Furthermore, it does not involve reductions. For example, if two terms $t$ and $t'$ are typed by two other terms that are applications $T$ and $T'$, then for deciding whether $t$ and $t'$ are of the same type terms $T$ and $T'$ must be computed for checking whether they reduce to the same value. In cGMM, types are already in normal form. A term is in normal form if and only if it cannot be further reduced. This has a deep impact on some key aspects of the calculus, as we shall discuss next.

4.2 Type Rules

Type rules enable the classification of terms in types. A judgment asserts that such a classification was deduced. Thus, the validity of a judgment is derived by means of type rules. However, information about some fragments of the term to be classified may be required for a deduction. Such information is recorded in a static typing environment, which is defined next, so a judgment relies on an environment for being declared valid or invalid.

Definition 4.2.1 (environment) An environment $\Gamma$ is a finite sequence of declarations of constants $\Gamma_c$, followed by a finite sequence of declarations of subtyping assertions $\Gamma_<$. A declaration of constant $c$ is either:
4.2 Type Rules

- an assumption \( c : T \), where \( T \) is a type, or
- a definition \( c := t : T \), where \( t \) is a term of type \( T \).

In turn, a declaration of a subtyping assertion has the form:
- \( A <: B \), where \( A \) and \( B \) are metamodels.

Constants declared in an environment must be distinct. The empty environment is denoted by \( \emptyset \). If \( \Gamma \) declares constant \( c \) in \( \Gamma_c \), then \( c \in \Gamma \). If \( (c : T) \in \Gamma \), then either \( c \) is declared by \( \Gamma \) or \( c \) is defined in \( \Gamma \) by \( t \) of type \( T \). When convenient, a comma is used as an operator for both appending declarations to environments and for concatenating environments, as in \( \Gamma, c : T, \Gamma' \) or \( \Gamma, A <: B, \Gamma' \). As the form of both kinds of declarations are different, there should not be ambiguities on whether the declaration is inserted in \( \Gamma_c \) or \( \Gamma_\prec \). The set of free variables \( \text{FV}(\Gamma) \) of an environment \( \Gamma \equiv c_1 : T_1, \ldots, c_n : T_n \mid A_1 <: B_1, \ldots, A_m <: B_m \) is defined as \( \bigcup_{1 \leq i \leq n} (c_i \cup \text{FV}(T_i)) \).

Remark For \( A \) to be a metamodel, \( \Gamma_c \) must include a constant \( A \) of type \( \text{Metamodel} \) (see rule (Metamodel Formation) in Section 4.2.5).

Remark Note that metamodels are atomic types and thus no free variable ever occurs in them. More specifically, any \( A \) and \( B \) involved in a subtyping assertion in \( \Gamma_\prec \) are constants already declared in \( \Gamma_c \).

Definition 4.2.2 (judgement) A judgement has the form \( \Gamma \vdash A \), where \( \Gamma \) is an environment, \( A \) is an assertion, and the free variables of \( A \) are declared in \( \Gamma \). For eGMM, the following judgements will be needed:
- \( \Gamma \vdash \emptyset \) \( \Gamma \) is a well-formed environment
- \( \Gamma \vdash M : A \) \( M \) is a well-formed term of type \( A \) in \( \Gamma \)
- \( \Gamma \vdash A <: B \) \( A \) is a subtype of \( B \) in \( \Gamma \)

Judgements are valid or invalid. The validity of a judgement is formally proved by type derivations.

Definition 4.2.3 (derivation) A derivation of a judgement \( \mathcal{J} \) is a finite sequence of judgments \( \mathcal{J}_1, \ldots, \mathcal{J}_n \), where \( \mathcal{J}_n \equiv \mathcal{J} \), such that, for \( 1 \leq i \leq n \), \( \mathcal{J}_i \) is the conclusion of some instance of an inference rule whose premises are in \( \{ \mathcal{J}_j \mid j < i \} \).

A judgement \( \mathcal{J} \) is derivable if there is a derivation of it. A derivation of judgement \( \mathcal{J} \) embodies a proof of its validity.
### 4.2 Type Rules

**Definition 4.2.4 (well-typing)** A term \( M \) is **well-typed** in an environment \( \Gamma \) if and only if there exists a term \( A \) such that the judgement \( \Gamma \vdash M : A \) is derivable.

**Definition 4.2.5 (terms and types)** Let \( \Gamma \) be an environment.

- A term \( M \) is called a \( \Gamma \)-term if it is well-typed under \( \Gamma \).
- A term \( A \) is called a \( \Gamma \)-type if \( \Gamma \vdash A : \text{Type} \).

In the previous chapter, we showed that megamodels may contain either entities or relationships. In most cases, a relationship element is just an additional representation of one same artifact which is also represented as an entity (i.e., the relationship among such elements is one-to-one). As a consequence, we can regard artifacts as represented by “extended entities” which are in some cases just entities and (e.g., metamodels) and in other cases the aggregation of their dual representation. In this way, we cope with the complexity introduced by managing two separate elements which refer to the same artifact. The only exception so far are composite transformations, which are represented as relationships only. Even if a composite transformation is not explicitly represented as a model in GMM, its definition actually exists and conforms to the metamodel introduced by the GMM4CT extension. Therefore, we argue that, regardless of how such artifacts are called, the required information is available and fits our approach.

### 4.2.1 Environment

The following rules formalize the notion of well-formedness of environments, and the (type) information that can be extracted from declarations within an environment.

<table>
<thead>
<tr>
<th>Rule</th>
<th>premises</th>
<th>conclusions</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Env Empty)</td>
<td>( \emptyset \vdash \odot )</td>
<td></td>
</tr>
<tr>
<td>(Env Assumption)</td>
<td>( \Gamma \vdash T : \text{Type} \quad c \notin \Gamma )</td>
<td>( \Gamma, c : T \vdash \odot )</td>
</tr>
<tr>
<td>(Env Definition)</td>
<td>( \Gamma \vdash t : T \quad c \notin \Gamma )</td>
<td>( \Gamma, c := t : T \vdash \odot )</td>
</tr>
<tr>
<td>(Assumption)</td>
<td>( \Gamma \vdash \odot \quad (c : T) \in \Gamma )</td>
<td>( \Gamma \vdash c : T )</td>
</tr>
<tr>
<td>(Definition)</td>
<td>( \Gamma \vdash \odot \quad (c := t : T) \in \Gamma ) for some ( t )</td>
<td>( \Gamma \vdash c : T )</td>
</tr>
<tr>
<td>(Sub Declaration)</td>
<td>( \Gamma \vdash A : T \quad \Gamma \vdash B : T \quad \Gamma \vdash T : \text{Metametamodel} \quad A, B \neq T \quad (A &lt; : C) \notin \Gamma ) for some ( C )</td>
<td>( \Gamma, A &lt; : B \vdash \odot )</td>
</tr>
</tbody>
</table>
4.2 Type Rules

\[(\text{Subtyping})\]
\[
\Gamma \vdash (A <: B) \in \Gamma \\
\Gamma \vdash A <: B
\]

The notion of environment with declarations directly corresponds to megamodels containing elements. An element externally defined with respect to a megamodel is assumed in an environment, and an element internally defined within a megamodel is defined in a context. The information contained in a subtyping assertion can be extracted from the metadata associated to the extending metamodel within the megamodel.

**Remark** Rule (Sub Declaration) requires that both \(A\) and \(B\) be metamodels and conform to the same metametamodel (see rules (Metamodel Formation) and (Metamodel Circ) in Section 4.2.5). In addition, a metamodel may only be subtype of at most one metamodel.

4.2.2 Subtyping

While types are seen as collections of elements, subtyping captures the notion of inclusion between types. An element of one type may be also considered as an element of any of its supertypes. This property is called subsumption, and it enables the use of an element in different typed contexts. The subtyping relation (denoted as \(<:\) ) is defined as a partial order on types. This means that such a relation is reflexive, antisymmetric and transitive. Furthermore, the subtyping relation has a top element (i.e., an upper bound or greatest element) which is Type.

Besides the basic rules shown next that define the subtyping relation, it is necessary to define subtyping rules for each type constructor in the system which admits subtyping, especially for the structured types. Such rules determine the meaning of subtyping for each type constructor based on the structure of the type, and will be presented when needed. In turn, for atomic types (i.e., non structured types) subtyping is declared as subtyping assertions in the environment, since it cannot be derived as for structured types.

\[(\text{Sub Introduction})\]
\[
\Gamma \vdash A : \text{Type} \\
\Gamma \vdash A <: A
\]

\[(\text{Sub Antisym})\]
\[
\Gamma \vdash A <: B \quad \Gamma \vdash B <: A \\
A = B
\]

\[(\text{Sub Trans})\]
\[
\Gamma \vdash B <: A \\
\Gamma \vdash C <: B \\
\Gamma \vdash C <: A
\]

\[(\text{Sub Elimination})\]
\[
\Gamma \vdash M : B \\
\Gamma \vdash B <: A \\
\Gamma \vdash M : A
\]
For metamodels, the notion of subtyping is closely related to metamodel extension. Metamodel extension was addressed in [JSZ+04] specifically for the UML case, and in [SJ07] in a broader MDE context. The latter, identified a number of variants to metamodel extension, some of which do not induce a subtyping relation. The concrete conditions two metamodels must satisfy for being considered in the relation are still not consensual, and they certainly depend on the metametamodel the metamodels conform to. Some specific technologies, such as ECore, support an explicit notion of metamodel extension. In turn, for example, KM3 does not. However, [RRDV07] proposed an algorithm for determining whether two given KM3 metamodels are in the relation. Such an algorithm essentially checks a number of conditions that elements within both metamodels should fulfill. In cGMM we do not deal with this issue. As metamodels are abstract elements, it is not possible to derive whether two metamodels satisfy the conditions for being in the relation, and therefore it is the user’s responsibility to make such an assertion. In fact, in GMM, that information is part of the metadata associated to the extending metamodel, and is thus available beforehand. The situation where a model transformation produces a metamodel which is intended to be an extension of another one, requires further investigation. Finally, metametamodel extension has not been directly discussed in the bibliography, and we are not aware of practical applications of it. For these reasons, cGMM provides support to metamodel extension only.

### 4.2.3 Sorts

The main sort in cGMM is Type, which is the type of all types. Therefore, every type in cGMM, including all sorts and Type itself, is of type Type. This is captured in the following rule.

\[
\frac{\Gamma \vdash \phi}{\Gamma \vdash \text{Type} : \text{Type}}
\]

**Remark** The Type:Type property has been seen as a dangerous feature in type theory [MR86] due to Girard’s paradox [Gir72]. In formal systems where such a property holds, the existence of normal forms is not guaranteed in general. This means that the system may not be strongly normalizable. Non normalization has two major consequences. First, as a logic system, the system is inconsistent. However, even if a dependent type system where Type:Type holds is inconsistent as a logic system, it is not inconsistent as a computation system [Car86a]. Calculi with an embedded logic system, such as Calculus of Constructions [CH88], deal with propositions and proofs by means of a type usually called Prop. cGMM does not have an embedded logic and is thus safe from this problem. Second, typechecking may diverge. The reason for this is that
reductions may fail to terminate because the existence of normal forms is not guaranteed. However, examples leading to divergence are extremely hard to reproduce, and do not have any impact on programming practice [Car86b]. Moreover, Cardelli claims that the \texttt{Type:Type} property makes perfect sense and can be incorporated in useful and semantically understood tools [Car86a]. In the next chapter, a correct typechecking algorithm will be presented, which embodies an ultimate proof of typechecking decidability, which in turn implies termination. Second, one property that cannot be enjoyed in \texttt{cGMM} is the identification of types with sets. Since \texttt{Type:Type} holds, \texttt{Type} would be both a set and an element of itself, which is a concept that is not supported by ordinary set theory. However, as in [Car86a] or in calculi with impredicative types, this leads to less intuitive but yet consistent meanings of the notion of a type.

Types in \texttt{cGMM} are of varied nature. There are sorts, families of dependent products, cartesian products, types representing reference models, and textual and weaving types. This categorization naturally introduces the idea of organizing types into collections of elements of similar structure and intention. More precisely, such an organization is a subtyping-based hierarchy of sorts with \texttt{Type} as its root. To this end, we augment the set of sorts \( \mathcal{S}_0 \) with a number of additional (sub)sorts. These sorts are introduced separately, according to the GMM extension they match. \( \mathcal{S}_{\text{GMM}} \) introduces basic sorts corresponding to the AM3Core metamodel and the GlobalModelManagement extension discussed in Chapter 2. In turn, \( \mathcal{S}_{\text{ATL}}, \mathcal{S}_{\text{TCS}}, \mathcal{S}_{\text{CT}} \) and \( \mathcal{S}_{\text{AMW}} \), introduce sorts corresponding to GMM4ATL, GMM4TCS, GMM4CT and GMM4AMW extensions, respectively.

\[
\mathcal{S}_{\text{GMM}} \equiv \{ \text{Type}, \text{Product}, \text{ReferenceModel}, \text{Metametamodel}, \text{Metamodel}, \text{Transformation}, \text{ModelTransformation}, \text{ExternalTransformation}, \text{Projector}, \text{T2TTransformation}, \text{M2ETransformation}, \text{E2MTransformation}, \text{M2TTransformation}, \text{T2MTransformation} \}
\]

\[
\mathcal{S}_{\text{ATL}} \equiv \{ \text{ATLTransformation} \}
\]

\[
\mathcal{S}_{\text{TCS}} \equiv \{ \text{Textual}, \text{TCSExtraction}, \text{TCSInjection}, \text{Program} \}
\]

\[
\mathcal{S}_{\text{CT}} \equiv \{ \text{CompositeTransformation} \}
\]

\[
\mathcal{S}_{\text{AMW}} \equiv \{ \text{AMWWeaving} \}
\]

The working set of sorts for \texttt{cGMM} is then fully defined as:

\[
\mathcal{S} \equiv \mathcal{S}_{\text{GMM}} \cup \mathcal{S}_{\text{ATL}} \cup \mathcal{S}_{\text{TCS}} \cup \mathcal{S}_{\text{CT}} \cup \mathcal{S}_{\text{AMW}}
\]
4.2 Type Rules

Figure 4.1: Hierarchy of sorts. Generalizations among sorts denote subtyping relationships. Abstract sorts imply that their elements are the disjoint union of the elements of their subsorts. Associations among sorts denote typing information related to them. This hierarchy is extensible as new sorts can be considered when new extensions are incorporated to GMM.

The hierarchy organizing sorts in $S$ illustrated in Figure 4.1 is inspired by the internal organization of the AM3Core metamodel and its extensions [Mod09b]. A number of rules sharing a similar structure are derived from this hierarchy and axiomatized in our system. First, every sort in $S$ is a type. Second, every generalization relationship in the hierarchy induces a subtype relationship between the corresponding sorts. Therefore for every sort $S$ in $S$, and for every pair of sorts $S_1$ and $S_2$, such that $S_2$ is a subclass of $S_1$ in the hierarchy, rules conforming to the following schema are introduced:

$$(S \text{ Type}) \quad \text{(Sub } S_2)$$

$$\Gamma \vdash \diamond \quad \Gamma \vdash \diamond \tag{*}$$

$$\Gamma \vdash S : \text{Type} \quad \Gamma \vdash S_2 <: S_1$$
4.2 Type Rules

For example, for the case of sort ModelTransformation, the schema above is instantiated to the following rules:

\[
\begin{align*}
\Gamma \vdash ModelTransformation : Type \\
\Gamma \vdash ModelTransformation : Transformation
\end{align*}
\]

Then, by virtue of rule (Sub Trans) a number of derived rules can be produced, e.g.,

\[
\Gamma \vdash ModelTransformation : Type
\]

Here the less intuitive notion of type discussed above can be appreciated. ModelTransformation, as a type (i.e., as a collection of elements), is included in Type by rule (Sub ModelTransformation-2). At the same time, ModelTransformation is also a element of Type by rule (ModelTransformation Type).

4.2.4 Cartesian Products

Cartesian products are primarily introduced for enabling \( n \)-ary functional results. For uniformity reasons, the same approach is adopted for function arguments. In principle, cartesian products are binary, however, generalized products can be achieved by iterating binary products. In practical situations, such a generalized version will be used. Therefore, \( m \)-to-\( n \) functions, with \( m,n > 2 \), will accept one single \( m \)-uple of arguments and produce one single \( n \)-uple of results. Sort Product is the type of all cartesian product types.

\[
\begin{align*}
\Gamma \vdash A : Type & \quad \Gamma \vdash B : Type \\
\Gamma \vdash A \times B : Product \\
\Gamma \vdash M : A_1 \times A_2 & \quad i \in \{1,2\} \\
\Gamma \vdash \pi_i(M) : A_i \\
\Gamma \vdash B_1 \times B_2 : A_1 \times A_2
\end{align*}
\]

In the GMM metamodel, tuples and cartesian products are induced by associations with an unbound multiplicity and are handled implicitly using multiple links to several
objects. As a result, no explicit notion of a tuple or cartesian product is represented. In turn, a projection \( \pi \) is not represented as an explicit operation on tuples either. However, our formal system requires the explicit representation of all these concepts.

Rule (Product Elimination) is a type rule and not a reduction rule. Therefore it focuses on the type of an application of \( \pi_i \). Even though argument \( M \equiv \langle M_1, M_2 \rangle \) for some \( M_1 : A_1 \) and \( M_2 : A_2 \), and \( \pi_i(M) \) reduces to \( M_i \), our type system does not perform computations. Reductions are addressed in the next chapter for proving fundamental properties. Rule (Sub Product) enables the derivation of subtyping judgements on cartesian products based on the structure of two product types. That is, it is based on relations among component types \( A_i \) and \( B_i \). Each of the two components of a cartesian product are covariant (i.e., a component of the subtype product is a subtype of the corresponding component of the supertype product).

### 4.2.5 Reference Models

A reference model can be either a metametamodel or a metamodel. The type of a metametamodel element is that element itself. In turn, a metamodel is an element which is typed by a metametamodel, and is not that metametamodel itself. In rule (Metamodel Formation) the rightmost premise is required, since \( A \), which is not a metamodel, satisfies the leftmost premise by (Metametamodel Circ).

\[
\text{(Metametamodel Circ)} \quad \Gamma \vdash A : \text{Metametamodel} \quad \Gamma \vdash A : A
\]

\[
\text{(Metamodel Formation)} \quad \Gamma \vdash M : A \quad \Gamma \vdash A : \text{Metametamodel} \quad M \neq A \quad \Gamma \vdash M : \text{Metamodel}
\]

Metamodel has an associated formation rule. Rule (Metametamodel Formation) can be intuitively interpreted as “if an element satisfies some specific conditions, then we can conclude that it is a metamodel”. In turn, this is not the case for Metametamodel. An element is just assumed as being a metametamodel. Rule (Metametamodel Circ) enables situations such as a transformation which accepts a source element of type \( A \), where \( A : \text{Metametamodel} \), to also accept \( A \) itself as a source element.

As discussed above, metamodel subtyping cannot be derived as it was the case for cartesian products. Metamodel subtyping is assumed in the environment instead. Therefore, no specific rules are provided here for that matter. In turn, in this version we decided not to support metametamodel subtyping. As a result, any derivation, involving a subderivation of a subtyping judgement for metametamodels, will not be able to be completed.
4.2.6 Textual Types

As discussed in Chapter 2, when textual entity \( t \) conforming to grammar \( G \) corresponds to model \( m \) conforming to reference model \( A \), either \( t \) is an extraction of \( m \) or \( m \) is an injection of \( t \). For avoiding the problem already discussed, we need a type for \( t \) such that it keeps track of reference model \( A \). To this end in [Vig09c] we introduced and detailed a type constructor \( \text{Text} \), where \( \text{Text}(A) \) is a textual type typing all textual entities conforming to some grammar \( G \) associated to \( A \). Therefore, \( \text{Text}(A) \) can be regarded as a synonym of \( G \). As a result, two textual entities typed by \( \text{Text}(A) \) and \( \text{Text}(B) \) respectively, are still textual entities, but not of the same type. In turn, sort \( \text{Textual} \) is the type of all textual types. This is captured in the following formation rule. The transference of type information during projections is further formalized when TCS projectors are addressed.

\[
\text{(Textual Formation)}
\begin{align*}
\Gamma \vdash A : \text{ReferenceModel} \\
\Gamma \vdash \text{Text}(A) : \text{Textual}
\end{align*}
\]

\[
\text{(Sub Textual)}
\begin{align*}
\Gamma \vdash \text{Text}(A) : \text{Textual} & \quad \Gamma \vdash \text{Text}(B) : \text{Textual} & \quad \Gamma \vdash B <: A \\
\Gamma \vdash \text{Text}(B) <: \text{Text}(A)
\end{align*}
\]

In rule \( \text{(Sub Textual)} \) we used a covariant approach. As grammar \( G_A \) associated to \( A \) may be regarded as smaller than grammar \( G_B \) associated to \( B \), this is sound in conjunction with subsumption [Car97]. Note that since \( A \) and \( B \) are reference models, the premise \( B <: A \) makes that both of them are, in particular, metamodels.

4.2.7 Transformations

In what follows, product types are introduced. They are intended to be used as functional products, both dependent and non dependent. However, later in this section, products will be used for type parameterization as well. Functional products are used for typing transformations. A product \( \Pi x : M_1.M_2 \) denotes a function mapping a value \( x \) of \( M_1 \) to a value of \( M_2(x) \). The type of the result depends on argument \( x \), with \( M_2 \) specifying the dependence [Mac86]. The degenerate case where \( M_2 \) is a constant function (i.e., \( M_2 \) does not depend on \( x \)) corresponds to classic function types, usually denoted as \( M_1 \rightarrow M_2 \). Transformations are usually typed by functional non dependent products. The dependent case only applies when a transformation accepts a reference model (which is a type) as a source element, and the type of its result somehow depends on that type. This is exactly the case of DTTs introduced in Section 2.1.2.2. The result type of a
transformation which accepts terminal models only, may not depend on such elements. The latter, is the most usual case in transformations.

A product $\Pi x : M_1, M_2$ cannot be used for typing any transformation from $M_1$ to $M_2$. In [VB09a] we realized that, in a HOT $T$, a source transformation $t$ may be used as a value instead of being used as a function. In other words, $T$ processes the definition of $t$, meaning that to that end $T$ needs to know the syntax of $t$. The definition of a transformation of type $M_1 \rightarrow M_2$ in ATL is very different from the definition of a transformation of the same type but in QVT; such transformations are models conforming to different metamodels. Transformation $T$ may be defined such that it accepts only ATL transformations of some type $M_1 \rightarrow M_2$ (i.e., it does not accept transformations of that type but defined in other transformation languages). For this reason, we argue that the transformation language in which a transformation is defined (i.e., the type of the transformation seen as an entity) should be a part of its type. With this extended notion of transformation types, a HOT may constrain the transformations it is ready to process, and also express the kind of transformations it is able to produce, in both cases in terms of the syntax of the transformations. We realize this mechanism by indexing functional products with the name of the language $\alpha$ used for defining the transformations they type. This means that $cGMM$ actually handles a family of functional products. The practical consequence of this is that we require a separate set of rules for every possible value of $\alpha$. Such sets are analogous, however, as we shall see next, significant differences may occur. For example, elimination rules may be similar across all sets, but formation rules may not. Furthermore some sets may have an introduction rule, while others may not.

**Remark** In this version of $cGMM$, possible values of $\alpha$ for indexing product types are closed to $\mathcal{I} \equiv \{ATL, CT, TCS, EXE\}$. This means that $cGMM$ currently provides rules for ATL transformations, Composite Transformations, and TCS projectors. The index EXE is used for typing text-to-text transformations. Although such elements are not currently supported in GMM, we believe that this is a nice extension, as text-to-text transformations often occur as a part of transformation chains in interesting practical situations.

When GMM is extended for supporting a new transformation language, such as QVT, set $\mathcal{I}$ should also be extended with a new index, and the corresponding set of rules for that index must be introduced. More details are provided in Section 4.3.

For clarity reasons, functional products are denoted using the classic $\rightarrow$ notation as detailed next, where its concrete form varies from the dependent case to the non dependent case. However, we still use the indexed $\Pi$ notation in situations where the case is not known (e.g., in type rules). The non indexed $\Pi$ notation is reserved for the other form of dependent products (which is independent of transformation languages):
products used for type parameterization, discussed later on this section. This scheme should avoid ambiguities.

\[ \Pi_{\alpha} x : T . U \equiv \begin{cases} 
  x : T \xrightarrow{\alpha} U & \text{if functional dependent product } (x \in \text{FV}(U)) \\
  T \xrightarrow{\alpha} U & \text{if functional non dependent product } (x \notin \text{FV}(U)) 
\end{cases} \]

\textbf{cGMM} is a domain specific calculus, and not a general purpose one. For this reason, it must encompass GMM. The source and the target types of transformations may not be any type. Furthermore, admissible types depend on whether the transformation is a model-to-model transformation or a projector. While the case of projectors is easier to characterize, the most delicate case is that of model-to-model transformations (i.e., transformations typed by a type \( T \), such that \( T \) is of type \text{ModelTransformation} and composite transformations. The former kind of transformations may only accept or produce GMM models, and the latter may handle textual entities as well.

**Definition 4.2.6 (models in cGMM)** \textit{In cGMM, a GMM model corresponds to a term } \( M \text{ such that } \Gamma \vdash M : A \text{, where } A \text{ either satisfies:} \)

1. \( \Gamma \vdash A : \text{ReferenceModel} \)
2. \( \Gamma \vdash A : \text{Weaving} \)
3. \( \Gamma \vdash A : \text{Transformation} \), but
   
   (a) \( \Gamma \nmid A : \text{E2ETransformation} \), and
   
   (b) \( \Gamma \nmid A : \text{CompositeTransformation} \)

\textit{The set of model types } \( \mathcal{M} \text{ is defined as the set of types which satisfy the conditions above.} \)

**Remark** In case 1 of the definition above, term \( M \) could represent either a metametamodel, a metamodel, or a non functional terminal model which is not a weaving model. In case 2, \( M \) represents a weaving model. In case 3, \( M \) represents an ATL or TCS transformation model. In GMM no transformation model can be associated to either a \text{CompositeTransformation} or \text{ExternalToExternalTransformation} relationship. Therefore no composite transformation or variant of external-to-external transformation could be the source or the target of a model transformation.

**Definition 4.2.7 (admissible types for model transformations)** \textit{The set } \( \mathcal{M}_{\text{mt}} \text{ of admissible types for being the source or the target of a term typed by a type of type} \)
ModelTransformation is formed by the types in \( \mathcal{M} \) together with any cartesian product composed by types in \( \mathcal{M} \). In symbols:

\[
\mathcal{M}_{MT} \equiv \mathcal{M} \cup \{ M_1 \times M_2 \mid M_1 \in \mathcal{M} \land M_2 \in \mathcal{M} \}
\]

\[\Box\]

**Definition 4.2.8 (admissible types for composite transformations)** Let be \( \mathcal{M}^+ \) an extension of set \( \mathcal{M} \) with textual types. The set \( \mathcal{M}_{CT} \) of admissible types for being the source or the target of a term typed by a type of type CompositeTransformation is formed by the types in \( \mathcal{M}^+ \) together with any cartesian product composed of types in \( \mathcal{M}^+ \). In symbols:

\[
\mathcal{M}^+ \equiv \mathcal{M} \cup \{ A \mid \Gamma \vdash A : \text{Textual} \}
\]

\[
\mathcal{M}_{CT} \equiv \mathcal{M}^+ \cup \{ M_1 \times M_2 \mid M_1 \in \mathcal{M}^+ \land M_2 \in \mathcal{M}^+ \}
\]

\[\Box\]

Now we can proceed with the type rules for each member of the family of indexed product types. Starting with ATL transformations, we then address composite transformations and TCS projectors. We also include text-to-text transformations.

**Remark** A functional product \( \Pi_\alpha x : M_1, M_2 \), for any \( \alpha \), can only be dependent when \( x \) is a type (i.e., a metamodel or a metametamodel). Or equivalently, when \( M_1 : \text{Metamodel} \). Otherwise, it is non dependent. \[\Box\]

### 4.2.7.1 ATL Transformations

ATL transformations are atomic model transformations. They are defined outside the scope of a megamodel and are thus treated as black boxes. In practical terms, an ATL transformation is represented by a constant assumed in the environment. For this reason, we provide a formation rule for \( \Pi_{\text{ATL}} \) but not an introduction one. ATLTransformation is the type of all products indexed by \( \text{ATL} \).

\[\begin{align*}
(\Pi\text{-ATL Formation}) & \quad \Gamma \vdash A : \text{Type} \quad \Gamma, x : A \vdash B : \text{Type} \quad A, B \in \mathcal{M}_{MT} \\
\Gamma & \vdash \Pi_{\text{ATL}} x : A, B : \text{ATLTransformation}
\end{align*}\]

\[\begin{align*}
(\Pi\text{-ATL Elimination}) & \quad \Gamma \vdash M : \Pi_{\text{ATL}} x : A, B \quad \Gamma \vdash N : A \\
\Gamma & \vdash (M \ N) : B\{x/N\}
\end{align*}\]
4.2 Type Rules

(Sub II-ATL)

\[
\Gamma \vdash A_2 <: A_1 \quad \Gamma \vdash B_1 <: B_2 \\
\Gamma \vdash \Pi_{ATL}x:A_1.B_1 <: \Pi_{ATL}x:A_2.B_2
\]

Note that in rule (II-ATL Formation), both \(A\) and \(B\) must be types, but in particular they must be in \(\mathcal{M}_{MT}\). In rule (II-ATL Elimination), which is a type rule and not a reduction rule, when the product is non dependent, as \(x\) does not occur free in \(B\), the substitution \(B\{x/N\}\) simply yields type \(B\).

4.2.7.2 Composite Transformations

Composite transformations, unlike ATL transformations, are defined within a megamodel. Their definition is therefore subject to type checks. In addition to formation and elimination rules, an introduction rule enables the definition of this kind of transformations. Similarly to the ATL case, sort \texttt{CompositeTransformation} is the type of all functional products indexed by \(CT\). In (II-CT Formation), \(A\) and \(B\) are types, but belonging to \(\mathcal{M}_{CT}\) in this case.

\[(\Pi\text{-CT Formation})\]

\[
\Gamma \vdash A : \text{Type} \quad \Gamma, x:A \vdash B : \text{Type} \quad A, B \in \mathcal{M}_{CT} \\
\Gamma \vdash \Pi_{CT}x:A.B : \text{CompositeTransformation}
\]

\[(\Pi\text{-CT Introduction})\]

\[
\Gamma \vdash \Pi_{CT}x:A.B : \text{CompositeTransformation} \quad \Gamma, x:A \vdash M : B \\
\Gamma \vdash \lambda_{CT}x:A.M : \Pi_{CT}x:A.B
\]

\[(\Pi\text{-CT Elimination})\]

\[
\Gamma \vdash M : \Pi_{CT}x:A.B \quad \Gamma \vdash N : A \\
\Gamma \vdash (M \ N) : B\{x/N\}
\]

\[(\text{Sub II-CT})\]

\[
\Gamma \vdash A_2 <: A_1 \quad \Gamma \vdash B_1 <: B_2 \\
\Gamma \vdash \Pi_{CT}x:A_1.B_1 <: \Pi_{CT}x:A_2.B_2
\]

Remark In rule (II-CT Introduction), term \(M\) is the body of the composite transformation. This means that such a term expresses its definition. In particular, term \(M\) is assumed to be well-typed (see the right hand premise). This means that the body of a composite transformation is effectively typechecked. In addition, such a body may not be
any arbitrary term. Admissible expressions for the body of a composite transformation may only be an arbitrary composition of constants, projections and tuple constructions. A transformation argument may have bound occurrences in such expressions. However, it may not be applied even if it is a transformation. Admissible expressions \( \text{exp} \) are inductively defined by the following clause. If \( c \) and \( c' \) are constants declared in the environment, \( x \) is the transformation argument, \( i \in \{1, 2\} \), and \( E \) and \( E' \) are \( \text{exp} \)s, so are the following:

\[
(cE), (cx), (c', c), \pi_i(c), \pi_i(E), \langle c, c' \rangle, \langle c, E \rangle, \langle E, c \rangle, \langle x, E \rangle, \langle E, x \rangle, \langle E, E' \rangle
\]

Even though we are aware that currently in GMM a composite transformation may not be accepted as a source element or produced as a target element by a transformation (because it is a relationship and not an entity), we choose to include rule (Sub \( \Pi \)-\( \text{CT} \)). Such a rule will remain unused until composite transformation are considered as transformation models.

### 4.2.7.3 TCS Transformations

TCS transformations, like ATL transformations, are atomic projectors. Again, this means that they are externally defined with respect to a megamodel. As such, we do not provide any introduction rule. Instead, only one formation rule for extractions and one formation rule for injections are provided following our proposal in \([\text{Vig09c}]\). Only elements conforming to a reference model may be extracted or injected. This fact imposes a particular constraint between the source type and the target type, as shown in the formation rules. Furthermore, as the target type may never depend on the projector argument, product types belonging to sorts \( \text{TCSExtraction} \) and \( \text{TCSInjection} \) are never dependent. Finally, one single elimination rule suffices for both variants of \( \text{TCS} \)-indexed functional products.

\[
\begin{array}{c}
\frac{
\Gamma \vdash A : \text{ReferenceModel} \quad \Gamma \vdash \text{Text}(A) : \text{Textual}
}{
\Gamma \vdash A \text{ TCS } \rightarrow \text{Text}(A) : \text{TCSExtraction}
}
\end{array}
\]

\[
\begin{array}{c}
\frac{
\Gamma \vdash A : \text{ReferenceModel} \quad \Gamma \vdash \text{Text}(A) : \text{Textual}
}{
\Gamma \vdash \text{Text}(A) \text{ TCS } \rightarrow A : \text{TCSInjection}
}
\end{array}
\]
4.2 Type Rules

Unlike other function types already discussed, we do not provide special subtyping rules for products indexed by TCS. This is because the structure of such types inevitably leads to an invariant situation for both source and target types. This implies that a term of the form \( A \xrightarrow{TCS} B \) either of type TCSExtraction or TCSInjection would only be a subtype of itself, which is already implied by rule (Sub Introduction). In fact, if the term is of type TCSExtraction, then \( B \equiv \text{Text}(A) \). For \( A \xrightarrow{TCS} \text{Text}(A) \prec A' \xrightarrow{TCS} \text{Text}(A') \) to hold, as with other function types, \( A' \prec A \) and \( \text{Text}(A) \prec \text{Text}(A') \) must also hold. However, by rule (Sub Textual) we can only derive \( \text{Text}(A') \prec \text{Text}(A) \). If both subtyping judgments were to hold at the same time, the only possibility is that \( \text{Text}(A) = \text{Text}(A') \), by rule (Sub Antisym). In turn, this can only happen if \( A = A' \). The case when the product type is of type TCSInjection is similar.

### 4.2.7.4 Programs

Programs are essentially text-to-text transformations. These transformations are, again, atomic transformations intended to operate in specific technical spaces such as Gram-marware. Therefore they are a specific variant of external-to-external transformations. Currently, ExternalToExternalTransformation is an abstract class and is not further specialized in GMM. As a consequence, no transformation model or any other element may be an external-to-external transformation. Although this kind of transformations lacks support in GMM, we believe that the corresponding artifacts are not uncommon when TCS is involved, for example as parts of practical transformation chains. For this reason we decided to provide support for them in cGMM. Programs are typed by functional non dependent products indexed by EXE. Sort Program is the type of such types.

\[
\begin{align*}
\text{(II-Prog Formation)} & \quad \Gamma \vdash \text{Text}(A) : \text{Textual} \quad \Gamma \vdash \text{Text}(B) : \text{Textual} \\
& \quad \Gamma \vdash \text{Text}(A) \xrightarrow{EXE} \text{Text}(B) : \text{Program} \\
\end{align*}
\]

\[
\begin{align*}
\text{(II-Prog Elimination)} & \quad \Gamma \vdash M : A \xrightarrow{EXE} B \quad \Gamma \vdash N : A \\
& \quad \Gamma \vdash (M \; N) : B \\
\end{align*}
\]

\[
\begin{align*}
\text{(Sub II-Prog)} & \quad \Gamma \vdash A_2 \prec A_1 \quad \Gamma \vdash B_1 \prec B_2 \\
& \quad \Gamma \vdash A_1 \xrightarrow{EXE} B_1 \prec A_2 \xrightarrow{EXE} B_2
\end{align*}
\]
Like TCS transformations, programs need formation and elimination rules. However, unlike TCS transformations, program types do not constrain a strong relationship between the source and target types. As a result, types are contravariant in argument positions and covariant in result position, as usual. Types $A_i$ and $B_i$ in rule (Sub II-Prog) are actually textual types. However, there is no pair of contradictory requirements on the same pair of types, as was the case for TCS transformations.

4.2.8 AMW Models

AMW models are one concrete realization of weaving models. In Chapter 2 we argued that typing weaving models only by their weaving metamodel does not suffice for some situations, and in [VB09b] we proposed a new type for them.

We incorporate type information about the woven models to the type of a weaving model. To this end, we introduce a type constructor $AMWWeaving$ according to the formation rule below. Having on the one hand a weaving model $m_w$ conforming to $M_w$ and woven models of type $M_a$ and $M_b$, and having on the other hand another weaving model $m'_w$ also conforming to $M_w$ but woven models of type $M_c$ and $M_d$ instead, we now have $m_w : AMWWeaving(M_w,M_a,M_b)$, but we also have $m'_w : AMWWeaving(M_w,M_c,M_d)$. This type information reveals that both weaving models are similar, since they both conform to the $M_w$ weaving metamodel, but they are not of the same type. This situation corresponds to the case when woven types (i.e., $M_a$, $M_b$, $M_c$ and $M_d$) are metamodels, or in other words, when woven models are terminal models. When woven models are types themselves, they could be referenced in manipulations of either $m_w$ or $m'_w$, and they should be part of the corresponding weaving type as well. In Chapter 6 we address a concrete application scenario where woven types are metatamodels.

$$(AMWWeaving~Formation)$$

\[
\Gamma \vdash A : Metamodel
\]

\[
\Gamma, B_1 : ReferenceModel \vdash x_1 : B_1 \ldots \Gamma, B_n : ReferenceModel \vdash x_n : B_n \ n \geq 2
\]

\[
\Gamma \vdash AMWWeaving(A, x_1 :: B_1, \ldots, x_n :: B_n) : AMWWeaving
\]

$$(Sub~AMWWeaving)$$

\[
\Gamma \vdash A' < : A \quad \Gamma \vdash B'_1 < : B_1 \quad \ldots \quad \Gamma \vdash B'_n < : B_n \ n \geq 2
\]

\[
\Gamma \vdash AMWWeaving(A', B'_1, \ldots, B'_n) < : AMWWeaving(A, B_1, \ldots, B_n)
\]

Rule (AMWWeaving Formation) specifies how a valid weaving type is constructed. Type $A$ is the weaving metamodel (extension). In turn, $x_i$ and $B_i$ are the woven models and their types respectively. Such types can only be reference models. Just like a dependent product $\Pi x : A . B$ could degenerate to a function type $A \to B$ (variable $x$ is omitted) when variable $x$ does not occur in type $B$, variables $x_i$ are omitted from a
weaving type when they do not denote a type (i.e., when $B_i$ is a metamodel). Even though $x_i:B_i$ truly holds by hypotheses, in the weaving type we use `::' for relating $x_i$ and $B_i$. Using a `:' instead, could be mistaken with a local variable declaration, as in the product type above. Rather, we emphasize that $x_i$ is just another component of the type and it corresponds to the next element in the enumeration (i.e., $B_i$). Separating $x_i$ from $B_i$ just by a `,' could lead to confusion when some $x_i$ is omitted. The use of a different symbol should then avoid ambiguities in such cases. Finally, for the (Sub AMWWeaving) rule we followed the straightforward covariant approach.

### 4.2.9 Type Parametrization

Types occurring in terms are generally constants (i.e., sorts or constants declared in the environment). However, in some situations one may need to express terms in such a way that (some) types are actually parameters to be instantiated later, when actual types are supplied. In such cases, that term is parameterized with respect to a type variable. More concretely, the term $\Lambda x:A.M$ indicates that term $M$ is parameterized with respect to type variable $x$, which is of type $A$. Naturally, for the parameterized term to be of any use, variable $x$ should occur in $M$, otherwise $M$ would not be a parametric term. The type of a parametric term is a non indexed dependent product. Thus, when $M:B$, provided that $x:A$, we say that $\Lambda x:A.M : \Pi x:A.B$. This corresponds to an informal version of an introduction rule. Among other uses, type parameterization, also called polymorphism [Car97], is intended for supporting the form of genericity described in Section 2.5.2.1, as discussed below.

**Remark** Parametric types in contexts like ours are usually denoted as $\forall x:A.B$, as for example in [Car86a]. However, in contexts such as pCic [CPT09], where the calculus also embeds a logic system, $B$ can be a type but also a proposition. There, parameterization is denoted using the $\forall$ notation as well. The former scenario is called functional dependent product from $A$ to $B$, and the latter scenario is called universal quantification of variable $x$ of type $A$. Naturally, for the parameterized term to be of any use, variable $x$ should occur in $M$, otherwise $M$ would not be a parametric term. The type of a parametric term is a non indexed dependent product. Thus, when $M:B$, provided that $x:A$, we say that $\Lambda x:A.M : \Pi x:A.B$. This corresponds to an informal version of an introduction rule. Among other uses, type parameterization, also called polymorphism [Car97], is intended for supporting the form of genericity described in Section 2.5.2.1, as discussed below.

Parameterization is intended to be closed with respect to sorts. That is, the parameterization $B'$ of a type $B$, where $B$ is a type in sort $S$, is also a type in $S$. This is expressed in the formation rule (\Pi Formation) below. For example, having $B \equiv M_1 \xrightarrow{\text{ATL}} M_2$, we know that $S \equiv \text{ATLTransformation}$. Then, for some $A : \text{Type}$, having $B' \equiv \Pi x:A.B$, we conclude that $B' : \text{ATLTransformation}$. The intuition behind this is the following: if $B$ is the type of an ATL transformation, then $B'$ is still the (parametric) type of an ATL transformation.
Parameterization can be performed on any type, as \( A \) in rule (\( \Pi \) Formation) could be \textbf{Type} itself. This means that in a parameterization \( \Pi x : \text{Type}.B \) variable \( x \) could be instantiated with any type. However, in some situations it might be desirable to restrict the domain of \( x \). In such cases, \( A \) could be any appropriate subtype of \textbf{Type}, however excluding elements in \textbf{Metamodel}. If a term \( N \) is such that \( N : \text{Metamodel} \), then it can be derived that \( N : \text{Type} \) and thus \( N \) can play the role of \( A \). However, if \( x : N \) then \( x \) is a terminal model and not a type, which is not possible. Typically, when \( A \not\equiv \text{Type} \), it could be a term \( N \) such that \( N : \text{Metametamodel} \). Then, \( \Pi x : N \ldots \) could be interpreted as “for any reference model \( x \) conforming to \( N \ldots \).” In this case, \( x \) could be instantiated to any reference model conforming to metametamodel \( N \), including \( N \) itself.

\[
\begin{align*}
(\Pi \text{ Formation}) & \\
& \Gamma \vdash A : \text{Type} \quad \Gamma, x : A \vdash B : S \quad S \in \mathcal{S} \\
& \quad \Gamma \vdash \Pi x : A.B : S
\end{align*}
\]

\[
\begin{align*}
(\Pi \text{ Introduction}) & \\
& \Gamma \vdash \Pi x : A.B : \text{Type} \quad \Gamma, x : A \vdash M : B \\
& \quad \Gamma \vdash \Lambda x : A.M : \Pi x : A.B
\end{align*}
\]

\[
\begin{align*}
(\Pi \text{ Elimination}) & \\
& \Gamma \vdash M : \Pi x : A.B \quad \Gamma \vdash N : A \\
& \quad \Gamma \vdash (M \ N) : B\{x/N\}
\end{align*}
\]

\[
\begin{align*}
(\text{Sub} \ \Pi) & \\
& \Gamma \vdash A_2 <: A_1 \quad \Gamma \vdash B_1 <: B_2 \\
& \quad \Gamma \vdash \Pi x : A_1.B_1 <: \Pi x : A_2.B_2
\end{align*}
\]

In rule (\( \Pi \) Introduction), term \( M \) plays an important role. It determines type \( B \), and since the product type in the conclusion is formed by an application of rule (\( \Pi \) Formation), it also determines type \( S \) in that rule. Typically, term \( M \) has the form \( \lambda_{cT} y : A.N \), where parameter \( x \) may occur both in \( A \) and \( N \). This corresponds to a parameterized definition of a composite transformation, and thus \( S \equiv \text{CompositeTransformation} \). However, in some special cases, \( M \) could take the form of an application \( (M_1 \ M_2) \). In such a case, type \( B \) could be any type, making \( S \) any sort. In Chapter 6 some rather complex cases are discussed in detail, where type parameterization is applied to concrete problems.

Finally, rule (\( \Pi \) Elimination) corresponds to the notion of type instantiation, and rule (\( \text{Sub} \ \Pi \)) defines subtyping for parametric types as usual.

This concludes our formal presentation of the cGMM calculus.
4.3 Extending the Calculus

The version of cGMM presented in this chapter supports the current set of GMM extensions. As new extensions may be produced in the future, cGMM may need to be extended accordingly. Extending the calculus involves a number of activities, where their individual complexity depends on the new extension to be considered. In what follows we discuss the activities required for extending cGMM when extension GMM4X is incorporated to GMM. Here X stands for the domain that is supported by the new extension.

It is not possible to anticipate the intent of any future extension, however, based on previously addressed extensions, we detected some patterns. Extensions may involve types for non-functional elements, types for functional elements, or both. For example, GMM4AMW introduced the AMWWeaving type constructor (non-functional), GMM4ATL introduced the ΠATL dependent product type (functional), and GMM4TCS introduced both the Text type constructor (non-functional) and the ΠTCS non-dependent product type (functional).

1. Add new terms. The inductive definition 4.1.2 of terms may need to be extended.
   - Add new clauses. If the extension involves a new type constructor, then a new clause for it should be introduced.
   - Add new indexes for product types. If a new product type is introduced, then element X should be included in the set of possible values for α.
   - Add new sorts. Extension GMM4X is likely to require some new sorts, just like the already supported extensions did. In such a case, the following steps are required.
     - Create set $S_X$ with the new sorts, for extending the working set $S$.
     - Update the hierarchy in Figure 4.1 with the sorts in $S_X$. Such sorts may be placed anywhere in the hierarchy, except as a superclass of Type. Abstract sorts may be freely reused, but Type should be kept as the top element of the hierarchy.
     - Generate for the sorts in $S_X$ new type rules, by instantiating schema (*)& in Section 4.2.3.

2. Add new type rules. Even if new sorts are not required for supporting extension GMM4X, specific type rules will be. We identify at least two basic cases.
   - Non-functional types. These types are likely to be elements of a sort which is a direct subtype of Type, or any other sort which is not a subtype of Transformation. At least a formation type rule is required for defining the new type constructor. If possible, a subtype type rule should be also introduced.
4.4 Discussion

- **Functional types.** These types are likely to be elements of a sort which is a subtype of Transformation. A $\Pi_X$ product type is introduced, either dependent or non-dependent, along with formation and elimination type rules at least. If elements of $\Pi_X$ are atomic (i.e., defined outside the scope of a megamodel), as for GMM4ATL, then no introduction type rule is required. However, if those elements are defined within a megamodel, as for the GMM4CT extension, then an introduction type rule is mandatory. Finally, a subtyping type rule should be also introduced.

Rules in Section 4.2 may be used as a guide for performing this activity.

3. **Review the calculus’ properties.** In the next chapter, a number of important meta-theoretic properties of cGMM are addressed. A part of extending the calculus is introducing new type rules, however, another part of the extension is reviewing the impact of those rules on the calculus’ properties.

- **Soundness.** The proof of the soundness theorem may be affected by the new constructs. It should be checked and properly updated if required.

- **Type inference.** As new types are introduced, the type inference algorithm will probably involve new cases. Additionally, the proof of correctness of the algorithm should be updated accordingly.

This concludes our discussion on extending the cGMM calculus for encompassing future GMM extensions.

4.4 Discussion

The results presented in this chapter are an improvement of our previous version of cGMM [VJB09, VJB11], partially developed in conjunction with the AtlanMod team. In both versions, a single metametamodel was supported, and its circularity was addressed by defining an infinite sequence of universes $\text{Type}_i$, where $\text{Type}_i : \text{Type}_{i+1}$. This technique is applied for coping with impredicativity (i.e., circularity) of types, and thus keeping consistency, in calculi with an embedded logic system such as pC1C [CPT09] or ECC [Luo90]. In addition, based on pC1C, we redefined the notion of static typing environment with assumptions and definitions, which is a more natural means to declaring atomic and composite elements. As a consequence, we simplified both the syntax of terms and the type rules. In [VB09a], we realized that the (name of a) transformation language should be a part of the type of transformations artifacts expressed in that language. In [VB09b] and [Vig09c] we proposed specific types for addressing issues concerning weaving artifacts, and textual entities and projectors, respectively. Finally, our
4.4 Discussion

approach to subtyping was greatly influenced by the work of Cardelli [Car97], especially for dependent types [Car86b, Car88].

GMM is about managing models and other MDE-related resources which are defined elsewhere. So far the only exception to this is that composite transformations are in fact defined within GMM. Typing becomes a critical issue when execution is considered, and can be studied both at intra-resource and inter-resource levels. In the former case, typing deals with elements within a resource, and the focus is on their internal properties. For example, a type system for a transformation language could ensure that produced models will satisfy some properties [CH06], such as good behavior. In the latter case, elements to be typed are the resources themselves. Typing in GMM mainly takes this second form. However, well typing of composite transformations (intra-resource level) is important to us as well. Similarly to GMM, [Fav04] presents a metamodel for describing MDE concepts and their relationships. Unlike GMM, only core concepts are considered and no tool support is reported. In particular, the typing of those concepts is not addressed or discussed, as we did for GMM.

Model typing is addressed in [SJ07] for investigating transformation reuse. A form of subtyping for model types (i.e., metamodels) enables a sort of subsumption on models. Under some circumstances the same transformation may be applied to models of different types. A basic transformation language was introduced for discussing those circumstances, and a type system was defined for it. In that language, transformations are in-place procedures rather than functions, thus they may not be composed. In addition, they are not treated as models and HOTs are not addressed. Although it is related to inter-resource issues due to the subtyping relation, that type system, compared to ours, mainly deals with internal concerns of transformation definitions.

Constructive Type Theory was used in [Poe06] for encoding the MOF layered metamodeling architecture. In particular, an infinite hierarchy of sorts was used for that purpose. However similar, the MOF hierarchy presents an extra level (i.e., the M0 level) compared to GMM’s. Additionally, the dual representation of elements at one level as types of that level and instances of types of the level above was represented, requiring reflection maps for establishing such a correspondence. In cGMM, for example, an element in M2 is at the same time an instance of an element in M3 and the type of an element in M1. Moreover, that theory considered only one metametamodel: MOF. Being focused on MOF, such a formalism is closed to the representation of MOF-based artifacts, which includes metamodels, models, and so on, but excludes other MDE-based artifacts. In particular, model transformations and their execution were not considered in that framework.

Type checking of compositions of transformations has been addressed in [Wil04] and with more detail in [VAB+07]. Both approaches use different notions of model typing, and like ours, they require the same type for connecting two adjacent subtransformations. However, none of them provides explicit rules to that end. Additionally, HOTs as well
as other cases discussed in this work are not handled.

GMM involves transformations written in different languages (e.g., ATL, TCS, Composite Transformations). However, this fact does not introduce any inter-language interoperability scenario. This is because GMM does not deal with values that are local to transformations, and transformations and typed values are in the same logical environment (i.e., the underlying megamodel). As a result, no conversion is required. Matthews and Findler [MF09] proposed a typed approach for dealing with foreign values in a multi-language context. In their lump embedding approach, a boundary between two languages is a cross-language cast that indicate a switch of languages. This is actually the role of projectors in a multi-technical space scenario; a textual entity needs to be converted before being handled by a model transformation. Although boundaries are reduction rules and projectors are user-defined functions, based on the types we introduced in Section 4.2.7.3, projectors and boundaries share a common purpose. Boundaries in the lump embedding only convert foreign values. The notion of natural embedding would become relevant to our work if foreign functions, as those we introduced in Section 4.2.7.4, are also considered as GMM elements.
Chapter 5

Properties of cGMM

In this chapter we study meta-theoretic properties of the calculus. The main properties are type soundness and decidability. Type soundness states that the evaluation of well-typed terms does not cause type errors (i.e., the notion of well-typing corresponds to a notion of good behavior). Since soundness refers to term evaluation, a semantics for GMM is required. We introduce a structural operational semantics for megamodels. Such a semantics defines a transition relation between terms which describes how individual steps of computation take place. Type soundness follows from proving a number of fundamental properties of such a transition system: confluence, strong normalization and type preservation, in addition to a characterization of ill-behaved terms and a proof of another property: ill-behaved terms lay outside the set of well-typed terms.

Decidability provides a direct basis for computer implementation of our calculus. We show that the problem of typechecking and type inference are decidable. Decidability follows from an algorithm that solves both problems. We prove that such an algorithm is correct, meaning that it is sound with respect to the type system (i.e., it infers only valid typings) and that it is complete (i.e., if a term has a valid type, then the algorithm will find it). Due to subtyping, terms may have more than one type. This has a direct impact on the type inference algorithm. For this reason, we start by studying the notion of principal types, and we introduce the notion of direct types. The latter are needed to overcome some particularities introduced by our calculus.

The remainder of this chapter is structured as follows. In Section 5.1 we address the notions of principal and direct types. In Section 5.2 we introduce a structural operational semantics for megamodels and prove a number of properties from which type soundness of cGMM is derived. In Section 5.3 we address decidability and provide a correct typechecking algorithm. Section 5.4 closes with a discussion of related work.
5.1 Principal and Direct Types

As we have type inclusions induced by the subtyping relation, type uniqueness for terms fails in cGMM. This means that a term may have more than one type. However, the notion of principal type is applicable to cGMM. The principal type of a term is the type from which all other types of the term can be derived [vBBF99]. Such a notion in cGMM may not be based on the subtyping relation only. This is because the type of a term may be derived by the application of rules which do not involve subtyping, particularly rules (Metametamodel Circ) and (Metamodel Formation). Instead, we introduce a cumulativity relation \( \preccurlyeq \), which is type cumulative by definition, for defining principal types. Finally, we introduce the notion of direct type of a term. It is a slight variation of the notion of principal type, which will be used in the definition of the type inference algorithm presented later.

Definition 5.1.1 (cumulativity relation) Let \( \preccurlyeq \subseteq T \times T \) be the relation over terms defined as follows. \( A \preccurlyeq B \) if and only if \( \Gamma, M : A \vdash M : B \). \( \square \)

Note that \( A <: B \) implies \( A \preccurlyeq B \) by application of rule (Sub Elimination). The contrary is not true as \( M : B \) might be derived from \( M : A \) by the application of other rules, such as (Metametamodel Circ) and (Metamodel Formation). Based on the cumulativity relation \( \preccurlyeq \) we now introduce the notion of principal type of a \( \Gamma \)-term.

Definition 5.1.2 (principal type) A is called a principal type of \( M \) under \( \Gamma \) if and only if

1. \( \Gamma \vdash M : A \), and
2. for any \( \Gamma \)-type \( A' \), \( \Gamma \vdash M : A' \) if and only if \( A \preccurlyeq A' \).

We use \( T_\Gamma(M) \) to denote the principal type of \( M \) under \( \Gamma \). \( \square \)

Since by definition both <: and \( \preccurlyeq \) enjoy the diamond property, the principal type of a \( \Gamma \)-term is unique. Next, we show that principal types do exist.

Theorem 5.1.3 (existence of principal type) Every \( \Gamma \)-term \( M \) has a principal type.

Proof Let \( A \) be the minimal type of \( M \) under \( \Gamma \) with respect to relation \( \preccurlyeq \). That is, if \( T \equiv \{ B \mid \Gamma \vdash M : B \} \), then \( A = \min_{\preccurlyeq}(T) \). \( T \) is a finite set since \( \preccurlyeq \) is well-funded, and thus type \( A \) exists. Then, \( \Gamma \vdash M : A \) holds, and \( A \preccurlyeq A' \) for any \( A' \in T \). This implies that \( A = T_\Gamma(M) \). \( \square \)
5.2 Soundness

For type inference we use the notion of principal type, however with a single exception. For metametamodels, their principal type is $\text{Metametamodel}$, but our algorithm requires such elements to be typed by themselves. To that end, we introduce the notion of direct type which can be intuitively understood as the same of that of principal type but including the exception mentioned before. Uniqueness of direct types follows from the uniqueness of principal types.

**Definition 5.1.4 (direct type)** A direct type of $M$ under $\Gamma$, denoted as $D_\Gamma(M)$, is defined as

$$D_\Gamma(M) \triangleq \begin{cases} M & \text{if } T_\Gamma(M) = \text{Metametamodel} \\ T_\Gamma(M) & \text{otherwise} \end{cases}$$

This concludes our preliminary definitions. In the next section we address type soundness. We start by introducing a semantics for megamodels, and then we prove a number of properties of the type system with respect to that semantics which lead to a proof of soundness.

### 5.2 Soundness

Our approach to type soundness follows the strategy proposed by Wright and Felleisen in [WF94]. Such an approach is based upon an operational formulation of the semantics of the language by rewriting. The basic idea is that each intermediate step of an evaluation of a term is itself a term, and the complete evaluation of a term is performed by a sequence of transitions to a new state: $M_1 \Rightarrow M_2 \Rightarrow \ldots$ Each intermediate state is a term, and therefore the type system itself may be used for deducing a type for them. A transition sequence may diverge, or may reach a final state where no further evaluation is possible. Such a state either represents the result of the evaluation or a type error, and is the meaning that a semantic function assigns to terms. In this context, proving type soundness reduces to proving that well-typed terms yield only well-typed results.

The concrete strategy we adapted for proving type soundness for $\text{cGMM}$ relies on a number of properties. First, every transition sequence, either diverges or reaches a unique final state. This means that the semantic function is in fact a partial function. Second, transition sequences starting from a well-typed term do not diverge. This still does not prevent well-typed terms to produce execution errors. The following property addresses this issue. Third, terms yielding an execution error are necessarily ill-typed. Weak soundness follows from these three properties, as they mean that well-typed terms do not go wrong. Fourth, transitions preserve types. This finally leads to the strong
soundness theorem based on the syntactic connection between answers of the semantic function and types:

\[ \text{if } \Gamma \vdash M : A \text{ then } \Gamma \vdash [M]_\Gamma : A. \]

Proofs of type soundness naturally rely on the language, but they also heavily rely on the formulation of the language semantics. The approach of Wright and Felleisen is motivated by the observation that in past experiences even a minor change in the language or in the formulation of the semantics could require a complete new approach to re-establish soundness [WF94]. With their syntactic approach, the structure of the proof always remains the same. This is very important to us, since cGMM is expected to encompass future extensions to GMM.

### 5.2.1 Semantics

In this section we introduce a structural operational semantics for megamodels. Its purpose is to describe how computation takes place. The meaning of terms is specified by a transition system. First, we assume the existence of a \( \Delta \) function that abstracts away the precise set of atomic transformations (i.e., assumed constants typed by a function type), which also satisfies a typability condition.

**Definition 5.2.1 (\( \Delta \)-typability)** Let \( C \) be the set of terms which are assumed constants \( M \) such as \( \Gamma \vdash M : \Pi \alpha x : A. B \), where \( \alpha \in \{\text{ATL}, \text{TCS}, \text{EXE}\} \). We assume the existence of a partial function that interprets the application of atomic transformations:

\[ \Delta : C \times T \rightarrow T \]

This function must satisfy the following typability condition for type soundness to hold:

\[
\Delta(M, N) \text{ is defined } \land \Gamma \vdash \Delta(M, N) : B\{x/N\}
\]

**Remark** The condition above implies that if terms \( M \) and \( N \) have the wrong types, then function \( \Delta \) is undefined. Moreover, it also implies that a given atomic transformation \( M \) is defined on all values of its source type (i.e., \( M \) is a total function). In practical contexts though, atomic transformations may actually behave as partial functions. In fact, for example if the implementation of an atomic transformation \( M \) or the definition
of \(N\) contain errors, then in some cases \(\Delta(M,N) = \bot\) (i.e., \(M\) does not produce a result at all). It is not possible for the type system to prevent such situations. Thus, by assuming \(\Delta\) restricted to a particular atomic transformation as a total function, we explicitly rely on the “well-implementedness” of atomic transformations.

\[\Box\]

The transition system is based on a transition relation that describes how the individual steps of computation take place. The definition of such a relation involves a new judgment and is given by a set of rules as follows.

**Definition 5.2.2 (transition relation \(\Rightarrow\))** The transition relation over terms is defined by the following rules. Such rules specify the relation independently of the well-typing of terms. In fact, rules apply to any environment \(\Gamma\), even to ill-formed ones.

\[
\begin{align*}
\text{(Trans } \delta) & \quad \frac{\text{}}{\Gamma \vdash (c := t : T) \in \Gamma} \\
\text{(Trans } \delta) & \quad \frac{\text{}}{\Gamma \vdash c \Rightarrow t} \\
\text{(Trans } \Delta) & \quad \frac{\text{(c : } \Pi_\alpha x:A.B) \in \Gamma \quad \alpha \in \{ATL, TCS, EXE\}}{\Gamma \vdash (c M) \Rightarrow \Delta(c, M)} \\
\text{(Trans } \beta-\text{CT}) & \quad \frac{\text{}}{\Gamma \vdash ((\lambda_{ct}x:A.M) N) \Rightarrow M\{x/N\}} \\
\text{(Trans } \beta) & \quad \frac{\text{}}{\Gamma \vdash ((\Delta x:A.M) N) \Rightarrow M\{x/N\}} \\
\text{(Trans } \sigma) & \quad \frac{i \in \{1, 2\}}{\Gamma \vdash \pi_i((M_1, M_2)) \Rightarrow M_i}
\end{align*}
\]

We additionally consider the following concepts:

1. Terms at the left side of the \(\Rightarrow\) symbol are called redexes. More specifically, \(\delta\)-redex for rule (Trans \(\delta\)), \(\beta\)-redex for rules (Trans \(\Delta\), (Trans \(\beta\)-CT) and (Trans \(\beta\))), and \(\sigma\)-redex for rule (Trans \(\sigma\)). The term at the right side of the \(\Rightarrow\) symbol is their corresponding contractum.
2. \( \Gamma \vdash M \Rightarrow M' \) means that an occurrence of a redex \( R \) in term \( M \) was replaced by its contractum leading to term \( M' \). More precisely, if \( M[R] \) is a term where redex \( R \) occurs and \( C \) is its contractum, then \( \Gamma \vdash M[R] \Rightarrow M[C] \).

3. A term is in normal form if and only if it does not contain any redex.

4. The transition \( \Gamma \vdash M \Rightarrow M' \) expresses the first step of execution of term \( M \) under \( \Gamma \). There are two possibilities depending on \( M' \):
   
   (a) If \( M' \) is in normal form, then no further transitions may take place, and thus the computation has terminated.

   (b) If \( M' \) is not in normal form, then the execution of \( M \) is not completed and the remaining computation is expressed by \( M' \).

If \( M \) is not in normal form (i.e., it contains redexes) and there is no \( M' \) such that \( \Gamma \vdash M \Rightarrow M' \), we say that \( M \) is stuck.

5. A term \( M_1 \) is strongly normalizable if and only if every transition sequence of the form \( M_1 \Rightarrow M_2 \Rightarrow M_3 \Rightarrow \ldots \) is finite. If \( M_1 \) is not strongly normalizable, then there exists at least one divergent transition sequence, and we say that \( M_1 \) diverges (denoted as \( M_1 \uparrow \)).

We use \( \Rightarrow^* \) to denote the reflexive and transitive closure of \( \Rightarrow \).

Remark Rule (Trans \( \delta \)) expresses the semantics of a definition, as it expands such a reference into its value. In turn, rules (Trans \( \Delta \)), (Trans \( \beta \)-CT) and (Trans \( \beta \)) respectively express the semantics of the application of an atomic transformation, the application of a composite transformation and a type instantiation. Finally, rule (Trans \( \sigma \)) expresses the semantics of a tuple elimination.

The meaning of terms can now be defined by means of the following semantic function.

**Definition 5.2.3 (semantic function)** The semantic function is a partial function defined as

\[
[M]_{\Gamma} \triangleq \begin{cases} 
V & \text{if } \Gamma \vdash M \Rightarrow^* V \\
\bot & \text{if } M \text{ yields a stuck term}
\end{cases}
\]

Remark: \( M \) may diverge.

Based on the transition system presented above, in the next section we address a proof of type soundness for cGMM.
5.2 Soundness

5.2.2 Type Soundness

For proving strong type soundness we have to prove a number of properties first. On
the one hand, we shall prove that a well-typed term yields a unique term in normal form
and of the right type. On the other hand we shall prove that ill-typed terms either yield
stuck terms or diverge. To that end, we shall prove the following concrete properties:

(Prop1): term $V$ (the value), if there exists for an untyped term $M$ (the argu-
ment), it is unique. This proves that the semantic function is in fact a function,
and that normal forms are unique.

(Prop2): well-typed terms do not diverge. Together with (Prop1), this proves
that well-typed terms that do not stuck have a unique normal form.

(Prop3): stuck terms are untypable. This proves that well-typed terms do not
stuck. In turn, together with (Prop1) and (Prop2), this proves weak type sound-
ness, i.e., a well-typed term yields a unique term $V$ in normal form, and an ill-typed
term either yields $\bot$ or diverges. In other words, only well-typed terms will not go
wrong.

(Prop4): transition relation $\Rightarrow^*$ preserves types. With all the above, this finally
proves strong type soundness, i.e., additionally to weak soundness, term $V$ is of
the right type.

(Prop1) is implied by the Church-Rosser theorem for the transition relation $\Rightarrow^*$.
(Prop2) is equivalent to proving that every well-typed term is strongly normalizable.
For proving (Prop3) we shall characterize stuck terms first. Finally, (Prop4) is plainly
Subject Reduction.

The most basic property of our transition system is the Church-Rosser theorem. However, before addressing it, we introduce an alternate transition system for parallel
transitions, which simplifies the proof. A parallel transition involves the contraction of
some (possibly all or none) of the redexes in a term, starting from within and proceeding
outwards. Such a system is defined as follows.

Definition 5.2.4 (parallel transition relations) The parallel one-step transition re-
lation over terms is defined by the following rules.

\[
\begin{align*}
\text{(PTrans Sym)} & \\
\Gamma \vdash M \Rightarrow^1 M \\
\text{(Trans $\delta$)} & \\
(c := t : T) \in \Gamma & \quad \Gamma \vdash c \Rightarrow^1 t
\end{align*}
\]
5.2 Soundness

\[(\text{PTrans } \Delta)\]
\[
(\alpha : \Pi_{\alpha} \, x:A \cdot B) \in \Gamma \quad \Gamma \vdash M \Rightarrow^1 M' \quad \alpha \in \{\text{ATL, TCS, EXE}\}
\]
\[
\Gamma \vdash (\alpha \, M) \Rightarrow^1 \Delta(\alpha, M')
\]

\[(\text{PTrans } \beta \text{-CT})\]
\[
M \Rightarrow^1 M' \quad N \Rightarrow^1 N'
\]
\[
(\lambda_{\text{CT}} x : A.M) \Rightarrow^1 M'[x \mapsto N']
\]

\[(\text{PTrans } \beta)\]
\[
M \Rightarrow^1 M' \quad N \Rightarrow^1 N'
\]
\[
(\lambda x : A.M) \Rightarrow^1 M'[x \mapsto N']
\]

\[(\text{PTrans } \sigma)\]
\[
M_1 \Rightarrow^1 M'_1 \quad M_2 \Rightarrow^1 M'_2 \quad i \in \{1, 2\}
\]
\[
\pi_i(\langle M_1, M_2 \rangle) \Rightarrow^1 M'_i
\]

We also define the parallel n-step transition as \(M \Rightarrow^0 N\) if and only if \(M = N\); \(M \Rightarrow^\alpha N\) if and only if \(M \Rightarrow^\alpha M' \Rightarrow^0 N\) for some \(M'\).

\[\square\]

Remark Note that \(M \Rightarrow^* N\) if and only if \(M \Rightarrow^\alpha N\) for some \(n \in \mathbb{N}\).

\[\square\]

Lemma 5.2.5 If \(M \Rightarrow^1 M'\) and \(N \Rightarrow^1 N'\) then \(M[N] \Rightarrow^1 M'[N']\).

Proof Obvious from the definition of \(\Rightarrow^1\).

\[\square\]

Lemma 5.2.6 (diamond property of \(\Rightarrow^1\)) If \(M \Rightarrow^1 M_1\) and \(M \Rightarrow^1 M_2\), then \(M_1 \Rightarrow^1 M'\) and \(M_2 \Rightarrow^1 M'\) for some \(M'\).

Proof The proof is by induction on the structure of \(M\). By lemma 5.2.6 we reduce case analysis to the following two cases:

1. \(M \equiv M_1 \cdot M_2\). By induction hypothesis we have that if \(M_2 \Rightarrow^1 X\) and \(M_2 \Rightarrow^1 Y\), then \(X \Rightarrow^1 Z\) and \(Y \Rightarrow^1 Z\) for some \(Z\). There are several cases on the structure of \(M_1\):

   (a) \(M_1 : \Pi_{\alpha} x : A \cdot B\). By rule (PTrans \(\Delta\)) we have that \(M \equiv M_1 \cdot M_2 \Rightarrow^1 \Delta(M_1, X)\) and \(M \equiv M_1 \cdot M_2 \Rightarrow^1 \Delta(M_1, Y)\). Then by lemma 5.2.5 we have that \(\Delta(M_1, X) \Rightarrow^1 M'\) and \(\Delta(M_1, Y) \Rightarrow^1 M'\), for \(M' \equiv \Delta(M_1, Z)\).

   (b) \(M_1 : x : A \cdot B\). By rule (PTrans \(\beta\)) we have that \(M \Rightarrow^1 M'\) and \(M \Rightarrow^1 M''\) for some \(M'\) and \(M''\). Then by lemma 5.2.5 we have that \(M' \Rightarrow^1 M''\) and \(M'' \Rightarrow^1 M''\) for some \(M''\).
5.2 Soundness

(b) $M_1$ is defined by term $M_1' \equiv \lambda CT. x : A. N$. Similarly, by rules (PTrans $\delta$) and (PTrans $\beta$-CT) we have that $M \equiv M_1 M_2 \Rightarrow^1 N\{x/X\}$ and $M \equiv M_1 M_2 \Rightarrow^1 N\{x/Y\}$. Again, by lemma 5.2.5, we have that $N\{x/X\} \Rightarrow^1 M'$ and $N\{x/Y\} \Rightarrow^1 M'$, with $M' \equiv N\{x/Z\}$.

(c) $M_1$ is defined by term $M_1' \equiv \Lambda x : A. N$. This case is analogous to the previous one, but applying rule (Trans $\beta$).

2. $M \equiv \pi_i(\langle M_1, M_2 \rangle)$. We consider the case when $i = 1$ only, the other is analogous. By induction hypothesis we have that if $M_1 \Rightarrow X$ and $M_1 \Rightarrow Y$, then $X \Rightarrow Z$ and $Y \Rightarrow Z$ for some $Z$. By rule (PTrans $\sigma$) we have that $M \equiv \pi_1(\langle M_1, M_2 \rangle) \Rightarrow X$ and $M \equiv \pi_1(\langle M_1, M_2 \rangle) \Rightarrow Y$. Then we have $X \Rightarrow Z$ and $Y \Rightarrow Z$.

This completes the proof of the lemma. □

Lemma 5.2.7 (diamond property of $\Rightarrow^n$) If $M \Rightarrow^m M_1$ and $M \Rightarrow^n M_2$, then $M_1 \Rightarrow^m M'$ and $M_2 \Rightarrow^n M'$ for some $M'$.

Proof By $m \times n$ applications of lemma 5.2.6. □

Now we have all the lemmas required for proving the Church-Rosser theorem for transition relation $\Rightarrow^*$. 

Theorem 5.2.8 (Church-Rosser theorem) If $M_1 = M_2$, then there exists $M$ such that $M_1 \Rightarrow^* M$ and $M_2 \Rightarrow^* M$.

Proof We only have to prove that the transition relation $\Rightarrow^*$ has the diamond property, i.e., if $M \Rightarrow^* M_1$ and $M \Rightarrow^* M_2$, then $M_1 \Rightarrow^* M'$ and $M_2 \Rightarrow^* M'$ for some $M'$. From lemma 5.2.7, the diamond property for the transition relation $\Rightarrow^*$ holds, and hence the theorem. □

Corollary 5.2.9 (uniqueness of normal forms) The normal form of a term is unique, if it exists.

Now that we have established (Prop1), we proceed to prove (Prop2) by proving strong normalization for well-typed terms. To that end, we first introduce the following concepts.

Lemma 5.2.10 (free variables) If $\Gamma \vdash M : A$ then $FV(M) \cup FV(A) \subseteq FV(\Gamma)$. 

5.2 Soundness

**Proof** By induction on derivations. It suffices to see that in any possible rule that enables the conclusion of $\Gamma \vdash M : A$, the premises ensure that all free variables are in fact declared in $\Gamma$.

Lemma 5.2.10 enables the definition of the following measure, which will be used for proving strong normalization.

**Definition 5.2.11 (level of a term)** Suppose $\Gamma \vdash M : A$, then the level of term $M$ with respect to environment $\Gamma$ is defined as

$$L(M, \Gamma) \triangleq \begin{cases} 
0 & \text{if } M \in S \\
\mathit{p} & \text{if } M \in \Gamma_\mathit{c} \text{ at position } \mathit{p} \\
|\Gamma_\mathit{c}| + 1 & \text{if } M \notin \Gamma_\mathit{c} \text{ and } M \notin S
\end{cases}$$

The following lemma proves that any well-typed term is built only of constants at a lower level.

**Lemma 5.2.12** If $\Gamma \vdash M : A$, then for every term $M' \in \text{FV}(M)$ (i.e., $M'$ is a constant occurring in $M$), $L(M', \Gamma) < L(M, \Gamma)$.

**Proof** There are two possibilities:

1. $M \notin \Gamma$. By definition, $L(M, \Gamma) = |\Gamma_\mathit{c}| + 1$. By lemma 5.2.10, every $M' \in \text{FV}(M)$ satisfies $L(M', \Gamma) = \mathit{p} \leq |\Gamma_\mathit{c}| < |\Gamma_\mathit{c}| + 1$.

2. $M \in \Gamma$. By definition, $L(M, \Gamma) = \mathit{m} \leq |\Gamma_\mathit{c}|$. Let $\Gamma'$ be the minimum prefix of $\Gamma$ such that $M \notin \Gamma'$ and $\Gamma' \vdash M : A$ holds. By rules (Env Assumption) and (Env Definition) $\Gamma'$ exists, and necessarily satisfies $|\Gamma'_\mathit{c}| < \mathit{m}$. By the same argument as before, $L(M', \Gamma) = L(M', \Gamma') \leq |\Gamma'_\mathit{c}| < \mathit{m}$.

This completes the proof of the lemma.

**Remark** Lemma 5.2.12 implies that there is no form of recursion for well-typed terms in cGMM. This is because recursion, either direct or indirect, in the declaration of a constant $M$ in a well-formed environment $\Gamma$ requires $L(N, \Gamma) \geq L(M, \Gamma)$, for at least one constant $N \in \text{FV}(M)$, which is not possible.

**Theorem 5.2.13 (strong normalization)** If $\Gamma \vdash M : A$ then $M$ is strongly normalizable.
Proof Our proof consists in showing that there are no infinite transition sequences starting from a well-typed term. The set of transition rules for relation ⇒ is finite, therefore an infinite transition sequence necessarily involves an infinite application of at least one of the rules. We shall prove that none of the rules may be applied infinite times in a transition sequence. For proving non-divergence, we associate a measure taken from a well-funded relation to terms. Showing that each step of a transition sequence decreases such a measure suffices to prove non-divergence, as there are no infinite descending chains in a well-funded set. We proceed by case analysis on transition rules:

1. (Trans δ): Infinite applications of this rule either require an infinite set of defined constants, which is not possible as environments are finite, or the reintroduction of an already unfolded constant. By lemma 5.2.12, when c is unfolded into t, all constants occurring in t have a level which is strictly less than that of c. Then c may not be reintroduced in the transition sequence, since it would require c to occur within a subterm at a lower level. Again by lemma 5.2.12, this is not possible.

2. (Trans β-CT): This rule may only be applied when the definition of a composite transformation was already unfolded. That is, for (Trans β-CT) to be applied, a previous application (Trans δ) is mandatory as in (for simplicity, both applications are shown consecutively):

   \[
   \ldots \Rightarrow c \ M \xrightarrow{\delta} ((\lambda_{CT} x : A. N) \ M) \xrightarrow{\beta_{CT}} N\{x/M\} \Rightarrow \ldots
   \]

   The amount of applications of (Trans β-CT) in every transition sequence is less of equal than the amount of applications of (Trans δ). Since the latter amount is finite, so is the former.

3. (Trans β): This case is analogous to the previous one.

4. (Trans Δ): Similarly to case 1, infinite applications of this rule require either an infinite set of constants, or the same constant to be reintroduced. (Trans Δ) may be applied to a redex of the form (c M), where c is an atomic transformation. The corresponding contractum is Δ(c, M) and c could occur in M if it contains a subterm of the form (c M'). Infinite applications of (Trans Δ) on c require M' to contain a subterm (c M'') and so on. Since terms are finite, it is not possible.

5. (Trans σ): For this case we use a different measure. An application of (Trans σ) to πᵢ((M₁, M₂)) yields Mᵢ and thus decreases the size of the argument of πᵢ. Since terms are finite, for applying (Trans σ) infinite times, Mᵢ must yield a larger term Mᵢ'. The transition sequence from Mᵢ to Mᵢ' in turn must involve at least the application of a rule different from (Trans δ). Such kind of sequences must occur an infinite amount of times. If possible, it would imply that at least one of the rules discussed above is applied infinitely. By the arguments above it is not possible.
5.2 Soundness

All transition rules can only be applied a finite number of times in a transition sequence starting from a well-typed term, and as a consequence no sequence diverges. □

By strong normalization, no well-typed term diverges, and by Church-Rosser, if a term converges it does to a unique normal form. Intersecting these two conditions we have so far that well-typed terms either converge to a unique normal form or they get stuck. The next step is then proving that well-typed terms do not stuck. Since determining whether a well-typed term yields a stuck expression is not a decidable property, we approximate the set of terms that become stuck with a set of faulty terms.

**Definition 5.2.14 (faulty terms)** The faulty terms of cGMM are the terms containing a subterm of the form \((M_1 M_2)\), where \(M_1 : \Pi_\alpha x:A.B (\alpha \in \{ATL,TCS,EXE\})\) and \(\Delta(M_1, M_2)\) is undefined.

The notion of faulty terms is a conservative approximation to the notion of stuck terms. In fact, all terms that yield a stuck term are faulty, however the opposite is not true. If we consider an environment \(\Gamma\) such that:

\[
\Gamma \equiv \Gamma', f : A \overset{ATL}{\rightarrow} B, a : A, ct := \lambda_{CT} x:A'.(f a) : A' \overset{CT}{\rightarrow} B, g : A'' \overset{ATL}{\rightarrow} B'', c : C
\]

where \(\Gamma'\) contains the required declarations for making \(\Gamma\) well-formed, then by rule (Trans \(\delta\)) and (Trans \(\beta-CT\)) we can prove \(\Gamma \vdash (ct (g n)) \Rightarrow^* \Delta(f,a)\), thus term \((ct (g c))\) is not stuck. However it is faulty, as \(\Delta(g,c)\) is undefined (because terms \(g\) and \(c\) do not satisfy the \(\Delta\)-typability condition). Now we can prove the following lemma.

**Theorem 5.2.15 (faulty terms are untypable)** If \(M\) is faulty, then there are no environment \(\Gamma\) and \(\Gamma\)-type \(A\) such that \(\Gamma \vdash M : A\).

**Proof** It suffices to show that subterms of \(M\) that cause \(M\) to be faulty are untypable. We would proceed by case analysis on the form of the subterms, but in cGMM there is only one case.

Suppose that \((M_1 M_2)\), where \(\Gamma \vdash M_1 : \Pi_\alpha x:A.B (\alpha \in \{ATL,TCS,EXE\})\), is faulty and \(\Gamma \vdash (M_1 M_2) : B\{x/M_2\}\). By rules (\(\Pi\)-ATL Elimination), (\(\Pi\)-TCS Elimination) and (\(\Pi\)-Prog Elimination), we know that \(\Gamma \vdash M_2 : A\). Then by the \(\Delta\)-typability condition, \(\Delta(M_1, M_2)\) is defined, but this contradicts the assumption that \((M_1 M_2)\) is faulty. □

By theorem 5.2.15, no well-typed term is faulty. Therefore, now we may safely argue that well-typed terms do not go wrong (i.e., weak soundness). Our last step before reaching strong soundness is proving type preservation.

**Theorem 5.2.16 (subject reduction)** If both \(\Gamma \vdash M_1 : A\) and \(\Gamma \vdash M_1 \Rightarrow M_2\) then \(\Gamma \vdash M_2 : A\).
Proof The proof proceeds by case analysis according to transition relation $\Rightarrow$:

1. (Trans $\delta$): In this case $M_1 \equiv c, M_2 \equiv t$ and $(c := t : T) \in \Gamma$. By rule (Definition) we have $c : T$, and trivially, $t : T$.

2. (Trans $\Delta$): In this case $M_1 \equiv (M N), M_2 \equiv \Delta(M,N)$ and $\Gamma \vdash M : \Pi_{\alpha: A.B} (\alpha \in \{ATL, TCS, EXE\})$. If $\Gamma \vdash (M N) : B\{x/N\}$, by rules (II-ATL Elimination), (II-TCS Elimination) and (II-Prog Elimination), we know that $\Gamma \vdash N : A$. Then, by (\Delta-Typability), $\Gamma \vdash \Delta(M,N) : B\{x/N\}$.

3. (Trans $\beta$-CT): In this case $M_1 \equiv (\lambda_{CT}x:A.M) N$ and $M_2 \equiv M\{x/N\}$. From $\Gamma \vdash (\lambda_{CT}x:A.M) N : B\{x/N\}$, by rule (II-CT Elimination) we have $\Gamma \vdash N : A$ and $\Gamma \vdash \lambda_{CT}x:A.M : \Pi_{CT}x:A.B$. From the latter, by rule (II-CT Introduction), we have $\Gamma, x : A \vdash M : B$. Since $\Gamma \vdash N : A$, we finally have $\Gamma \vdash M\{x/N\} : B\{x/N\}$.

4. (Trans $\beta$): This case is analogous to the previous one.

5. (Trans $\sigma$): In this case $M_1 \equiv \pi_i(\langle N_1,N_2 \rangle)$ and $M_2 \equiv N_i (i \in \{1,2\})$. From $\Gamma \vdash \pi_1(\langle N_1,N_2 \rangle) : A_1$ and $\Gamma \vdash \pi_2(\langle N_1,N_2 \rangle) : A_2$, by rule (Product Elimination) we have $\Gamma \vdash \langle N_1,N_2 \rangle : A_1 \times A_2$. Finally, by rule (Product Introduction) we have $\Gamma \vdash N_1 : A_1$ and $\Gamma \vdash N_2 : A_2$.

This completes the proof of the theorem. \qed

Theorem 5.2.17 (strong soundness) If $\Gamma \vdash M : A$ then $\Gamma \vdash [M]_\Gamma : A$.

Proof By definition 5.2.3 there are three cases:

1. $M$ diverges. This case is not possible, since $M$ is well-typed, and by theorem 5.2.13, $M$ cannot diverge.

2. $[M]_\Gamma = \bot$. This case is not possible either, since $[M]_\Gamma = \bot$ implies $\Gamma \vdash M \Rightarrow^* M'$, where $M'$ is faulty. By theorem 5.2.16, $\Gamma \vdash M' : A$, which is a contradiction, since by theorem 5.2.15, faulty terms are untypable.

3. $[M]_\Gamma = V$. This is the only possible case, and $\Gamma \vdash M \Rightarrow^* V$. Theorem 5.2.16 finally implies $\Gamma \vdash V : A$.

This completes the proof of the theorem. \qed

With this result, we have shown that in cGMM the semantic function for a well-typed term returns a unique normalized term of the expected type. In turn, the semantic function for a term that causes a type error returns $\bot$. 
5.3 Decidability

In this section we study the decidability of cGMM. We present a correct algorithm for typechecking and type inference. In particular, the algorithm is sound as it infers only valid typings, and is complete as if a term has a valid typing then the algorithm will find it. The existence of such an algorithm, together with the decidability of the subtyping and cumulativity relations, establish the decidability of the calculus. This result is a property which exclusively applies to the type system, and is independent of the relation of the type system to the semantics addressed in the previous section.

5.3.1 Decidability of Subtype and Cumulativity Relations

Lemma 5.3.1 (decidability of $\prec$) It is decidable whether $\Gamma \vdash M \prec N$ for a given environment $\Gamma$ and arbitrary well-typed terms $M$ and $N$.

Proof First, since cGMM is based on by-name equivalence, term equality is decidable. Then, as $\Gamma$ is finite, it is always possible to check whether or not there exists a sequence of terms $T_1, \ldots, T_n$, such that $T_1 \equiv M$, $T_n \equiv N$, and $\Gamma \vdash T_i \prec T_{i+1}$ for every $i \in \{1, \ldots, n-1\}$. By successive applications of rule (Sub Trans), such a sequence of judgments constitutes a derivation of $\Gamma \vdash M \prec N$.

Corollary 5.3.2 (decidability of $\preceq$) It is decidable whether $\Gamma \vdash M \preceq N$ for a given environment $\Gamma$ and arbitrary well-typed terms $M$ and $N$.

Proof The decidability of the cumulativity relation $\preceq$ follows from that of the subtyping relation.

5.3.2 Decidability of Typechecking and Type Inference

Definition 5.3.3 (typechecking) Given an environment $\Gamma$ and terms $M$ and $A$, typechecking is to check whether term $A$ is a correct type and term $M$ is of type $A$ in $\Gamma$. This is, finding a derivation of judgement $\Gamma \vdash M : A$.

Definition 5.3.4 (type inference) Given an environment $\Gamma$ and a term $M$, type inference is to find a term $A$, such that judgement $\Gamma \vdash M : A$ is derivable.

In cGMM, typechecking and type inference are directly performed on constants to be declared within an environment only. Assumptions of the form $c : T$ are typechecked. However, since constant $c$ is assumed as being of type $T$, it is only necessary to check whether $T$ is a correct type. This reduces to perform type inference on $T$ and check
whether its type belongs to $S$. In turn, in definitions of the form $c := t : T$, term $T$ is not explicitly provided. Therefore, such a term $T$ is the result of performing type inference on term $t$. Since typechecking and type inference are mutually dependent, we simultaneously define them in the type inference algorithm shown next.

**Definition 5.3.5 (algorithm of type inference)** The algorithm of type inference $\text{Type}(\cdot : \cdot)$ when given an environment $\Gamma \equiv c_1:T_1, \ldots, c_n:T_n \mid A_1 <: B_1, \ldots, A_m <: B_m$ and a term $M$, it checks whether $M$ is well-typed in $\Gamma$, and if so, $\text{Type}(\Gamma; M) = D_{\Gamma}(M)$, the direct type of $M$ under $\Gamma$; otherwise, it returns $\bot$ (i.e., fail). In what follows, we assume that every considered environment is valid (definition 5.3.6). $\text{Type}(\cdot : \cdot)$ is defined by induction on the structure of $M$.

1. $M$ is a sort. Then, $\text{Type}(\Gamma; M) \triangleq \text{Type}$.

2. $M$ is a constant. Then, $\text{Type}(\Gamma; M) \triangleq \begin{cases} T_i & \text{if } M = c_i \land T_i \neq \text{Metametamodel} \\ M & \text{if } M = c_i \land T_i = \text{Metametamodel} \\ \bot & \text{if } M \notin \{c_1, \ldots, c_n\} \end{cases}$

3. $M$ is a variable introduced as $M : N$. Then, $\text{Type}(\Gamma; M) \triangleq N$.

4. $M \equiv \Pi_\alpha x : M_1.M_2$. There are several cases depending on $\alpha$:
   
   (a) $\alpha \equiv \text{ATL}$. Check whether $\text{Type}(\Gamma; M_1) = N$, where $\Gamma \vdash N : \text{Type}$ and $N \in \mathcal{M}_{\text{ATL}}$, and $\text{Type}(\Gamma;x : M_1; M_2) = N'$, where $\Gamma \vdash N' : \text{Type}$ and $N' \in \mathcal{M}_{\text{ATL}}$. If so, $\text{Type}(\Gamma; M) \triangleq \text{ATLTransformation}$; otherwise $\text{Type}(\Gamma; M) \triangleq \bot$.

   (b) $\alpha \equiv \text{CT}$. Check whether $\text{Type}(\Gamma; M_1) = N$, where $\Gamma \vdash N : \text{Type}$ and $N \in \mathcal{M}_{\text{CT}}$, and $\text{Type}(\Gamma;x : M_1; M_2) = N'$, where $\Gamma \vdash N' : \text{Type}$ and $N' \in \mathcal{M}_{\text{CT}}$. If so, $\text{Type}(\Gamma; M) \triangleq \text{CompositeTransformation}$; otherwise $\text{Type}(\Gamma; M) \triangleq \bot$.

   (c) $\alpha \equiv \text{TCS}$. There are two specific forms of $M$:
   
   i. $M \equiv M' \xrightarrow{\text{TCS}} \text{Text}(M')$. Check whether $\text{Type}(\Gamma; M') = N$, where $\Gamma \vdash N : \text{ReferenceModel}$. If so, $\text{Type}(\Gamma; M) \triangleq \text{TCSExtraction}$; otherwise $\text{Type}(\Gamma; M) \triangleq \bot$.

   ii. $M \equiv \text{Text}(M') \xrightarrow{\text{TCS}} M'$. Check whether $\text{Type}(\Gamma; M') = N$, where $\Gamma \vdash N : \text{ReferenceModel}$. If so, $\text{Type}(\Gamma; M) \triangleq \text{TCSInjection}$; otherwise $\text{Type}(\Gamma; M) \triangleq \bot$.

   (d) $\alpha \equiv \text{EXE}$. $M$ has the form $\text{Text}(M_1) \xrightarrow{\text{TCS}} \text{Text}(M_2)$. Check whether $\text{Type}(\Gamma; M_1) = N$, where $\Gamma \vdash N : \text{ReferenceModel}$, and $\text{Type}(\Gamma; M_2) = N'$, where $\Gamma \vdash N' : \text{ReferenceModel}$. If so, $\text{Type}(\Gamma; M) \triangleq \text{Program}$; otherwise $\text{Type}(\Gamma; M) \triangleq \bot$.

5. $M \equiv \Pi x : M_1.M_2$. Check whether $M_1 = \text{Type}$ or $\text{Type}(\Gamma; M_1) = N$, where $\Gamma \vdash N : \text{Metametamodel}$, and $\text{Type}(\Gamma;x : M_1; M_2) = N'$ with $N' \in S$. If so, $\text{Type}(\Gamma; M) \triangleq N'$; otherwise, $\text{Type}(\Gamma; M) \triangleq \bot$. 


5.3 Decidability

6. \( M \equiv \lambda x : M_1 . M_2 \). Check whether \( \text{Type}(\Gamma; M_1) = N \), where \( \Gamma \vdash N : \text{Type} \) and \( N \in \mathcal{M}_{CT} \), and \( \text{Type}(\Gamma, x : M_1; M_2) = B \), where \( B \neq \bot \). If so, \( \text{Type}(\Gamma; M) \triangleq \Pi_{x : M_1} B \); otherwise \( \text{Type}(\Gamma; M) \triangleq \bot \).

7. \( M \equiv \Lambda x : M_1 . M_2 \). Check whether \( M_1 = \text{Type} \) or \( \text{Type}(\Gamma; M_1) = N \), where \( \Gamma \vdash N : \text{Metametamodel} \), and \( \text{Type}(\Gamma, x : M_1; M_2) = B \), where \( B \neq \bot \). If so, \( \text{Type}(\Gamma; M) \triangleq \Pi_{x : M_1} B \); otherwise \( \text{Type}(\Gamma; M) \triangleq \bot \).

8. \( M \equiv M_1 \times M_2 \). Check whether \( \text{Type}(\Gamma; M_1) = N \), with either \( N \equiv \Pi_{\alpha : A.B} \) or \( N \equiv \Pi_{\alpha : A.B} \), for some \( A \) and \( B \), and \( \Gamma \vdash \text{Type}(\Gamma; M_2) \prec A \). If so, \( \text{Type}(\Gamma; M) \triangleq B \{x/M_2\} \); otherwise \( \text{Type}(\Gamma; M) \triangleq \bot \).

9. \( M \equiv M_1 \times M_2 \). Check whether \( \text{Type}(\Gamma; M_1) = N \), where \( \Gamma \vdash N : \text{Type} \), and \( \text{Type}(\Gamma; M_2) = N' \), where \( \Gamma \vdash N' : \text{Type} \). If so, \( \text{Type}(\Gamma; M) \triangleq \text{Product} \); otherwise \( \text{Type}(\Gamma; M) \triangleq \bot \).

10. \( M \equiv \langle M_1, M_2 \rangle \). Check whether \( \Gamma \vdash \text{Type}(\Gamma; M_1) = A \), where \( \Gamma \vdash A : \text{Type} \), and \( \Gamma \vdash \text{Type}(\Gamma; M_2) = B \), where \( \Gamma \vdash B : \text{Type} \). If so, \( \text{Type}(\Gamma; M) \triangleq A \times B \); otherwise \( \text{Type}(\Gamma; M) \triangleq \bot \).

11. \( M \equiv \pi_i (M') \). Check whether \( \text{Type}(\Gamma; M') = A \times B \), for some \( A \) and \( B \). If so, \( \text{Type}(\Gamma; M) \triangleq \begin{cases} A & \text{if } i = 1 \\ B & \text{if } i = 2 \end{cases} \); otherwise \( \text{Type}(\Gamma; M) \triangleq \bot \).

12. \( M \equiv \text{Text}(M') \). Check whether \( \text{Type}(\Gamma; M') = \text{ReferenceModel} \). If so, \( \text{Type}(\Gamma; M) \triangleq \text{Textual} \); otherwise \( \text{Type}(\Gamma; M) \triangleq \bot \).

13. \( M \equiv \text{AMWWWeaving}(W, T_1, \ldots, T_n) \). Check whether \( \text{Type}(\Gamma; W) = \text{Metamodel} \) and \( \text{Type}(\Gamma; T_i) = \text{ReferenceModel} \), for \( i \in \{1, \ldots, n\} \). If so, \( \text{Type}(\Gamma; M) \triangleq \text{AMWWWeaving} \); otherwise \( \text{Type}(\Gamma; M) \triangleq \bot \).

This completes the definition of the algorithm.

Although slightly more complicated by the use of the cumulativity relation and direct types instead of the subtyping relation and principal types, this algorithm is similar to that defined by Luo in [Luo90]. Our type inference algorithm was implemented as a part of the toolset discussed later in Chapter 7.

**Definition 5.3.6 (validity of an environment)** To see whether environment \( \Gamma \equiv c_1 : T_1, \ldots, c_n : T_n \mid A_1 \prec B_1, \ldots, A_m \prec B_m \) is valid, check:

1. \( \text{Type}(c_1 : T_1, \ldots, c_{n-1} : T_{n-1}; T_n) = S \in S \); and
2. \( \text{Type}(\Gamma; A_i) = \text{Type}(\Gamma; B_i) = \text{Metamodel} \) \((i \in \{1, \ldots, m\})\); and
5.3 Decidability

3. \( A_i \neq A_j \) (\( i \neq j \) with \( i,j \in \{1,\ldots,m\} \)).

If \( \Gamma \) is not valid, then \( \text{Type}(\Gamma; M) = \bot \) for any term \( M \).

Theorem 5.3.7 (correctness of type inference) The algorithm \( \text{Type}(\_, \_\_) \) is correct, i.e., given an environment \( \Gamma \) and a term \( M \),

\[
\text{Type}(\Gamma; M) = \begin{cases} \mathbb{D}_\Gamma & \text{if } M \text{ is well-typed} \\ \bot & \text{otherwise} \end{cases}
\]

Proof The proof is by the same induction used in definition 5.3.5. As in [Luo90], in this proof the only case worth mentioning about is when \( M \equiv M_1 M_2 \) (Luo handled \( \Sigma \)-types, while we only handle cartesian products, and thus the case when \( M \equiv \pi_i(M') \) is not addressed).

\( M \equiv M_1 M_2 \). If either \( \text{Type}(\Gamma; M_1) \) is not of the form \( \Pi_\alpha x:A_1.B_1 \) or \( \Pi x:A.B \), or it is but \( \text{Type}(\Gamma; M_2) \not\preceq A \), then by induction hypothesis \( M \) is not a \( \Gamma \)-term. Otherwise, we have \( \Gamma \vdash M : B\{x/M_2\} \) by one of the rules (\( \Pi-\alpha \) Elimination) or rule (\( \Pi \) Elimination). We have to show that \( B\{x/M_2\} \) is the direct type (more specifically, the principal type) of \( M \). Suppose \( \Gamma \vdash M : B' \) for some \( B' \prec B\{x/M_2\} \). We may assume that the last rule used to derive \( \Gamma \vdash M : B' \) was either one of the rules (\( \Pi-\alpha \) Elimination) or rule (\( \Pi \) Elimination):

\[
\Gamma \vdash M_1 : \Pi_\alpha x:A_1.B_1 \quad \Gamma \vdash M_2 : A_1 \\
\Gamma \vdash M : B_1\{x/M_2\}
\]

where \( B_1\{x/M_2\} \equiv B' \prec B\{x/M_2\} \). By induction hypothesis, \( \Pi_\alpha x:A.B = \text{Type}(\Gamma; M_1) \not\preceq \Pi_\alpha x:A_1.B_1 \). Therefore, \( B \preceq B_1 \), which in turn implies \( B\{x/M_2\} \preceq B_1\{x/M_2\} \), which is a contradiction. Then, \( B\{x/M_2\} = \mathbb{T}_{\Gamma}(M) \), and by definition 5.1.4, we finally have \( \mathbb{T}_{\Gamma}(M) = \mathbb{D}_\Gamma(M) \).

The decidability of the type inference algorithm and that of the cumulativity relation imply that the problem of typechecking is decidable.

Corollary 5.3.8 (decidability of typechecking) \( \text{cGMM} \) is decidable, i.e., it is decidable whether \( \Gamma \vdash M : A \) for arbitrary \( \Gamma \), \( M \) and \( A \).

Proof By definition of direct type, to see whether \( \Gamma \vdash M : A \), it suffices to check whether \( \Gamma \vdash \text{Type}(\Gamma; M) \not\preceq A \). By theorem 5.3.7 and corollary 5.3.2, this is decidable.
5.4 Discussion

Some of the results presented in this chapter were inspired by or adapted from other works. We took from the work of Luo [Luo90] the idea of defining principal types for dealing with multiple typing of terms. However, due to particularities of our calculus, we had to further define the notion of direct types as an extension of principal types for type inference purposes.

The structural operational semantics we introduced in Section 5.2.1, follows the basic notions presented by Nielson [NN07], but it obviously targets the specific megamodelling context. The use of a $\Delta$ function for abstracting away from atomic transformations was adapted from Wright and Felleisen [WF94]. In turn, $\delta$ and $\sigma$ reductions were taken from pCic [CPT09] and ECC [Luo90] respectively.

As already mentioned, the proof of soundness in Section 5.2.2 follows the approach proposed by Wright and Felleisen [WF94]. Their application of such an approach to Functional ML (a language based on Plotkin’s $\lambda_v$-calculus [Plo75]) involved a simpler semantics based on contexts and evaluation contexts, which satisfied Church-Rosser and resulted strongly normalizable. For this reason, we had to address these properties for the particular case of cGMM explicitly. For proving Church-Rosser we followed the high level approach of Luo [Luo90], where a proof of the diamond property of $\Rightarrow^1$ was only sketched. We adapted such a proof from Plotkin’s [Plo75]. Our proof of strong normalization was based on our intuition on the fact that a term is always reduced to an expression composed of simpler terms. Finally, the proof of subject reduction was adapted from the proof of Wright and Felleisen for Functional ML [WF94].

Our typechecking algorithm shares a similar structure to that of ECC [Luo90], and was influenced by insights from the work of Coquand [Coq96] and Cardelli [Car86b]. The proof of correctness of type inference was adapted from Luo’s [Luo90].
Chapter 6

Applications

Our type system for Global Model Management was completely formalized in Chapter 4 and its main properties were proved in Chapter 5. The presentation of cGMM included an abstract description of how different GMM artifacts and their manipulation within a megamodel are expressed as terms of the calculus within an environment. In this chapter, we address several model (or artifact) management situations for illustrating the concrete application of cGMM in general and its constructs in particular. Such management situations are in essence the registration to a megamodel of a number of artifacts involved in a specific working zone and the execution of some of them for producing new ones. Those working zones correspond to real-world case studies in MDE. Our primary selection criteria for case studies is not their complexity, neither in terms of the amount of different artifacts that are involved nor the internal technical difficulty of transformations. Rather, we focus on simpler cases which introduce complex typing situations. An extremely complex transformation may involve a trivial type, while, as we shall see, the application of advanced typing constructs could be required in some simple artifact manipulation contexts.

We start in Section 6.1 by revisiting the ATL copier transformation generation example already presented in Section 2.5.3. A type for the KM32ATLCopier HOT and DTT transformation and related artifacts are discussed. In Section 6.2, we discuss the paraphrasing of KM3 reference models and ATL transformations. This example involves a HOT as before but with type parameterization, and also composite transformations and a TCS extractor. A metamodel adaptation problem is addressed in Section 6.3, which focuses on AMW weaving models and involves cartesian products. Section 6.4 deals with the problem of performing automatic layout of elements within diagrams, which involves a composite transformation with cartesian products and tuples, TCS projectors, and a program. Finally, in Section 6.5 we address the promotion and demotion of models.
A megamodel is represented by an environment $\Gamma$, and elements within a megamodel correspond to constants declared (i.e., assumed or defined) in such an environment. Assumed constants have the form $c : T$, while defined constants have the form $c := t : T$. When a constant is assumed, the well-formedness of type $T$ is checked. This is typechecking. In turn, when a constant is defined, type $T$ is inferred from $t$. This is type inference.

Transformation $\text{KM32ATLCopier}$ [ATL09] is an atomic ATL transformation. As it is an atomic transformation, it is included in an environment as an assumption. In an environment containing assumption $\text{KM3} : \text{Metametamodel}$, the concrete assumption for $\text{KM32ATLCopier}$ is then:

$$\text{KM32ATLCopier} : M : \text{KM3} \rightarrow \text{ATL} \rightarrow M$$

The type assigned to $\text{KM32ATLCopier}$ is a function type which depends on value $M$. Its co-domain is another function type, however non-dependent, where both the domain and the co-domain are $M$ (here the dependency is apparent). The $\Pi$-based expression of that type would be $\Pi_{\text{ATL}} M : \text{KM3}. \Pi_{\text{ATL}x} : M.M$, which is much less intuitive if compared to the function type given above. Such a function type replaces both types $(T_e)$ and $(T_r)$ from Section 2.5.3.
Now we apply the \( \text{KM32ATLCopier} \) transformation to a concrete KM3-based meta-model (e.g., \( \text{SQL} \)). For this purpose we include in the environment the assumption \( \text{SQL} : \text{KM3} \). Artifacts involved in this application are illustrated in Figure 6.1. The application is expressed within a definition: the name of the result of the application is specified in \( c \)'s position, the application itself is specified in \( t \)'s position, and the type of the result occurs in \( T \)'s position. We name the resulting copier transformation \( \text{SQL-Copier} \), and the complete definition is then:

\[
\text{SQLCopier} := (\text{KM32ATLCopier SQL}) : \text{SQL}^{\text{ATL}} \rightarrow \text{SQL}
\]

In this chapter, we adopt the convention that when we display a type, or subexpression in general, in gray color, we mean that such subexpression is inferred and it does not need to be explicitly provided by the user. This is the case for the type of \( \text{SQLCopier} \). Such a type is what is returned by the type inference algorithm when it is fed with the \( (\text{KM32ATLCopier SQL}) \) functional application, and it replaces types \( (T_e') \) and \( (T_r') \) from Section 2.5.3. Note that such a result is different from \( \perp \); it not only means that it is the type of \( \text{SQLCopier} \), but it also means that the functional application is well-typed. Since our type inference algorithm was proven correct, it would not be necessary to typecheck the functional application against the inferred type. However, we show a derivation of the corresponding typing judgment to illustrate the operation of the type system and the type inference algorithm. In fact, such an algorithm builds the derivation tree from the root to the leaves. We first define an environment \( \Gamma \) as follows:

\[
\Gamma \equiv \text{KM3} : \text{Metametamodel}, \\
\text{SQL} : \text{KM3}, \\
\text{KM32ATLCopier} : M: \text{KM3}^{\text{ATL}} \rightarrow M^{\text{ATL}} \rightarrow M
\]

Then, the following typing judgment can be derived, which proves that the inferred type is a type for \( \text{SQLCopier} \):

\[
\Gamma \vdash (\text{KM32ATLCopier SQL}) : \text{SQL}^{\text{ATL}} \rightarrow \text{SQL}
\]

The core part of the derivation is shown in Figure 6.2. The proof of \( \Gamma \vdash \diamond \) is simple but lengthy. For simplicity we only provide a sketch of it.

**Proof Sketch** We define environment \( \Gamma_1 \equiv \text{KM3:Metametamodel}, \text{SQL:KM3} \). Then, by application of rule \( (\text{EnvAssumption}) \), and provided that \( \text{KM32ATLCopier} \notin \Gamma_1 \), we have to prove \( \Gamma_1 \vdash M: \text{KM3}^{\text{ATL}} \rightarrow M^{\text{ATL}} \rightarrow M : \text{Type} \). This judgment can be proved by applying rule \( (\text{Sub Elimination}) \) and derived rules such as \( (\text{Sub ModelTransformation-2}) \), when judgment \( \Gamma_1 \vdash M: \text{KM3}^{\text{ATL}} \rightarrow M^{\text{ATL}} \rightarrow M : \text{ATLTransformation} \) is proved. In turn, such a judgment can be proved by application of rule \( (\Pi\text{-ATL Formation}) \) when both
6.1 Copier Transformation Generation

Figure 6.2: Derivation of judgment $\Gamma \vdash (\text{KM32ATLCopier SQL}) : SQL \rightarrow SQL$

judgments $\Gamma_1 \vdash \text{KM3} : \text{ReferenceModel}$ and $\Gamma_1, M : \text{KM3} \vdash M \rightarrow M : \text{ATLTransformation}$ are proved. The former is proved by applying rule (Env Assumption) and derived rule (Sub Metametamodel). The latter, by a further application of rule (Π-ATL Formation).

□

We can now safely apply SQLCopier to a terminal model $s_1$ of type SQL, or any subtype of it. Note that if $s_1$ is of a different type, then the type derivation for $(\text{SQLCopier } s_1)$ we are about to discuss would not be possible to complete. This is equivalent to the type inference algorithm returning a ⊥ value when executed on $(\text{SQLCopier } s_1)$. For processing such an application, we define environment $\Gamma'$ as an augmentation of environment $\Gamma$ as follows:

$$\Gamma' \equiv \text{KM3} : \text{Metametamodel},$$
$$\quad SQL : \text{KM3},$$
$$\quad \text{KM32ATLCopier} : M : \text{KM3} \rightarrow M \rightarrow M,$$
$$\quad \text{SQLCopier} := (\text{KM32ATLCopier SQL}) : SQL \rightarrow SQL,$$
$$\quad s_1 : SQL$$

The application, again, occurs within a definition. We name the involved constant $s_2$. The expected type of such an application, and hence, of constant $s_2$, is SQL. The complete definition is then:

$$s_2 := (\text{SQLCopier } s_1) : SQL$$

Again, the type in gray is returned by the type inference algorithm, and typechecking is required for the application. Nevertheless, we show in Figure 6.3 the core part of a derivation of the following typing judgment:

$$\Gamma' \vdash (\text{SQLCopier } s_1) : SQL$$

As before, we do not prove the validity of environment $\Gamma'$. Note the application of
6.2 Paraphrasing Reference Models and Transformations

In this section, we discuss the typing of the artifacts involved in the case study of paraphrasing KM3 reference models and ATL transformations [Vig09b]. The main motivation for such a case study is that models are mostly rendered using graphical notations, however, graphical forms are usually complemented by a textual description in natural language. The main purpose of this description is enhancing understandability for human readers. Thus, such textual descriptions focus on documentation. Although documentation is an important aspect of a product, these textual descriptions are usually produced by humans. Writing documentation is a time consuming activity and its results may be of little value if it is not carried out properly. Maintaining documentation is also costly because the text and the model it describes must be synchronized. It is a common situation to find outdated descriptions or no descriptions at all.

The case study addresses the automatic generation of textual descriptions of models. Models are transformed to descriptions in natural language. A model transformation first produces a model of a simplified natural language which contains the full text of the description. A model extractor then generates the actual text from that intermediate model. The transformation that produces the intermediate textual model naturally relies on the language used for expressing the model to be described. This means that for each language a specific transformation for processing the corresponding models is needed. Specifically, the languages that are addressed are KM3 for reference models and ATL for transformation models.

The artifacts involved in this case study include the KM3-based SimpleLanguage rule (Definition) for typing the SQLCopier constant. For typing KM32ATLCopier, in the previous derivation, we applied rule (Assumption) instead. This is because KM32ATLCopier was assumed, while SQLCopier was defined.

In conclusion, in cGMM a HOT and DTT transformation such as KM32ATLCopier can be properly typed and safely applied. More importantly, a resulting transformation such as SQLCopier can also be typed and applied.

\[
\begin{align*}
\Gamma' &\vdash \diamond \\
\Gamma' &\vdash SQLCopier : SQL^{\text{ATL}} \rightarrow SQL \quad \text{(Definition)} \\
\Gamma' &\vdash s1 : SQL \quad \text{(Assumption)} \\
\Gamma' &\vdash (SQLCopier s1) : SQL \quad \text{(II-ATL Elimination)}
\end{align*}
\]

**Figure 6.3:** Derivation of judgment $\Gamma' \vdash (SQLCopier s1) : SQL$
6.2 Paraphrasing Reference Models and Transformations

Figure 6.4: Artifacts involved in the paraphrasing of the KM3 metametamodel

metamodel for expressing the intermediate textual models, ATL transformations \( KM32SL \) and \( ATL2SL \) which respectively produce intermediate textual models representation from KM3 reference models and ATL transformation models, and the TCS extractor \( SLExtactor \) which produces the final text. The generation of the final text from a reference model or from a transformation model involves the successive execution of \( KM32SL \) and \( SLExtactor \), and \( ATL2SL \) and \( SLExtactor \), respectively. Therefore, composite transformations \( ParaphraseKM3 \) and \( ParaphraseATL \) are introduced. Figure 6.4 illustrates the first set of artifacts. Composite transformation \( ParaphraseKM3 \) does not explicitly involve intermediate model \( tmp \), however it is represented for clarity reasons. Finally, \( Text(SimpleLanguage) \) is actually a type, not a metamodel. Still, we chose to include it in the M2 level.

In the rest of this chapter we do not deal with type derivations as in the previous section. Rather, we handle inferred types directly in definitions (we still use text in gray for readability). In definition 4.2.7 we introduced set \( M_{MT} \) for specifying rules such as (II-ATL FORMATION). For convenience, we introduce here a predefined sort \( MTType \). Then an element \( A \) in that sort satisfies both \( \Gamma \vdash A:Type \) for every environment \( \Gamma \), and \( A \in M_{MT} \). The presentation of the artifacts is based on specifying an environment containing all the corresponding terms and their types. Therefore, such a sequential presentation of terms may be regarded as a script. For clarity, we define two different environments, even though they contain repeated elements, for showing the artifacts of paraphrasing reference models and paraphrasing transformation models separately. Of
course, one single environment containing the union of all definitions may be handled. For the case of paraphrasing KM3 reference models, a concrete environment could be:

\[
\begin{align*}
\text{KM3} & : \text{Metametamodel}, \\
\text{SimpleLanguage} & : \text{KM3}, \\
\text{SLE} & : \text{SimpleLanguage}^{\text{TCS}} \xrightarrow{TCS} \text{Text} (\text{SimpleLanguage}), \\
\text{KM3SL} & : \text{KM3} \xrightarrow{\text{ATL}} \text{SimpleLanguage}, \\
\text{ParaphraseKM3} & := \lambda_{\text{CT}} x : \text{KM3} \cdot (\text{SLE} (\text{KM3SL} x)) \\
& : \text{KM3} \xrightarrow{\text{CT}} \text{Text} (\text{SimpleLanguage}), \\
\text{KM3Text} & := (\text{ParaphraseKM3} \text{ KM3}) : \text{Text} (\text{SimpleLanguage})
\end{align*}
\]

Note that even if the source type of \(\text{KM3SL}\) is \(\text{KM3}\), its target type does not depend on the value received as an argument. For this reason, \(\text{KM3SL}\) is not a DTT transformation. In turn, constant \(\text{KM3Text}\) represents a textual entity, since its type is \(\text{Text} (\text{SimpleLanguage})\). For the case of paraphrasing ATL transformation models, a concrete environment could be:

\[
\begin{align*}
\text{KM3} & : \text{Metametamodel}, \\
\text{SimpleLanguage} & : \text{KM3}, \\
\text{SLE} & : \text{SimpleLanguage}^{\text{TCS}} \xrightarrow{TCS} \text{Text} (\text{SimpleLanguage}), \\
\text{ALT} & : \Pi A : \text{MTType} . \Pi B : \text{MTType} . (A^{\text{ATL}} B) \xrightarrow{\text{ATL}} \text{SimpleLanguage}, \\
\text{ParaphraseATL} & := \Lambda X : \text{MTType} . \Lambda Y : \text{MTType} . \lambda_{\text{CT}} x : (X^{\text{ATL}} Y). (\text{SLE} (\text{ALT} x)) \\
& : \Pi X : \text{MTType} . \Pi Y : \text{MTType} . (X^{\text{ATL}} Y) \xrightarrow{\text{CT}} \text{Text} (\text{SimpleLanguage}), \\
\text{Families} & : \text{KM3}, \\
\text{Persons} & : \text{KM3}, \\
\text{FamiliesPPersons} & : \text{Families} \xrightarrow{\text{ATL}} \text{Persons}, \\
\text{F2PText} & := (\text{ParaphraseATL} \text{ Families Persons Families2Persons}) \\
& : \text{Text} (\text{NaturalLanguage})
\end{align*}
\]

This case deserves several remarks. First, transformation \(\text{ALT2SL}\) is a HOT, however, unlike \(\text{KM32ATLCopier}\) which produces a transformation, \(\text{ALT2SL}\) accepts an ATL transformation, and more specifically, any ATL transformation. For this reason, the source type of \(\text{ALT2SL}\) is \(A^{\text{ATL}} B\), where \(A\) and \(B\) are type parameters (note the use of the non-indexed form or \(\Pi\) for that purpose). Second, any application of \(\text{ALT2SL}\) (e.g., that occurring within the definition of \(\text{ParaphraseATL}\)) requires the instantiation of parameters \(A\) and \(B\). In the case of \(\text{ParaphraseATL}\), they are instantiated to parameters \(X\) and \(Y\), respectively. Such parameters are in turn instantiated to metamodels \(\text{Families}\) and \(\text{Persons}\) in the application of \(\text{ParaphraseATL}\) within the definition of \(\text{F2PText}\). This
Paraphrasing Reference Models and Transformations

is because the typechecker needs both metamodels for checking compatibility between the instantiation of \( X^{ATL} \rightarrow Y \) (i.e., the type of argument \( x \) of ParaphraseATL) and the type of transformation Families2Persons (i.e., the actual parameter in the application). Third, we colored in gray the type instantiations of \( A \) and \( B \) to \( X \) and \( Y \) (in ParaphraseATL), and of \( X \) and \( Y \) to Families and Persons (in F2PText). This means that the implementation of the typechecker can actually infer the values of the type parameters, and thus they do not need to be provided explicitly by the user. With this, the concrete definition of F2PText is F2PText := (ParaphraseATL Families2Persons), which is simpler.

As a final remark, we discuss a more complex situation within this case study. ATL2SL is an ATL transformation which produces a model with a textual description of any ATL transformation, including itself. An application of ATL2SL to itself requires careful examination, since its type is more complicated than that of transformation Families2Persons. In fact, even if performed by the typechecker, the instantiation of parameters \( A \) and \( B \) still needs to be carried out, in both occurrences of ATL2SL. Let us define tmp as the application of ATL2SL to itself:

\[
\text{tmp} := \lambda X:\text{MTType}.\lambda Y:\text{MTType}. (\text{ATL2SL} \ ?_1 \ ?_2 (\text{ATL2SL} \ ?_3 \ ?_4))
\]

\[
: \Pi X:\text{MTType}.\Pi Y:\text{MTType}.\text{SimpleLanguage}
\]

The first unexpected thing in this definition is the occurrence of parameters \( X \) and \( Y \) in binding the application and its type. Such parameters are meant to be used in place of placeholders \( ?_3 \) and \( ?_4 \). As already stated, parameters \( A \) and \( B \) need to be instantiated, in this case in the inner occurrence of ATL2SL, even if it occurs as a value. Recall that a transformation which is received as an argument may not be applied. However, in type theory such a constraint does not apply, and therefore parameters corresponding to the inner occurrence of ATL2SL must be instantiated. Moreover, the inferred type for tmp (i.e., SimpleLanguage) does not actually depend on parameters \( X \) and \( Y \). In summary, types for placeholders \( ?_3 \) and \( ?_4 \) are still required. The other two placeholders are simply determined: \( ?_1 \) is the source of the inner application, and \( ?_2 \) is its target. Now, the complete definition can be expressed as (compare it to the application of ATL2SL within ParaphraseATL above):

\[
\text{tmp} := \lambda X:\text{MTType}.\lambda Y:\text{MTType}. \\
(\text{ATL2SL} \ (X^{ATL} \rightarrow Y) \ \text{SimpleLanguage} \ (\text{ATL2SL} \ X \ Y))
\]

\[
: \Pi X:\text{MTType}.\Pi Y:\text{MTType}.\text{SimpleLanguage}
\]

Gray subterms are inferable. In fact, System Coq [CPT09] is indeed capable of handling this exact same situation. Then, the typechecker can be instructed for recognizing situations like this and further inserting inferable information. The definition of tmp would be more readable if written as:
$$tmp := (ATL2SL \; ATL2SL)$$

However, the inferred type for $tmp$ is still $\Pi X:MTType.\Pi Y:MTType.SimpleLanguage$, which is in fact theoretically correct, even though $SimpleLanguage$ was the intuitive result. Terminal model $tmp$ is expected to be used as a parameter for $SLExtractor$ for producing the textual description of $ATL2SL$. Such a definition is:

$$ATL2SLText := (SLExtractor \; (tmp \; ?_5 \; ?_6))$$

Since $SLExtractor$ expects as an argument a model conforming to $SimpleLanguage$, we still need to instantiate parameters $X$ and $Y$ in $tmp$’s type. We can parameterize this application with parameters $C$ and $D$, both of type $MTType$, but they both would appear in the type of $ATL2SLText$ just like $X$ and $Y$ appeared in the type of $tmp$ above. An alternative approach could be instantiating $?_5$ and $?_6$ to arbitrary types of $MTType$, thus avoiding parameters $C$ and $D$. This approach may seem odd in principle, however it works, since it is clear that we need types for $?_5$ and $?_6$, but it is also clear that, in cases like this, it does not matter what types they are as long as they are of type $MTType$. Note that this same situation applies to placeholders $?_3$ and $?_4$. Fixing them, values for $?_1$ and $?_2$ can be inferred, and $?_5$ and $?_6$ would be no longer needed. For the sake of illustration, let $M$ and $N$ be such arbitrary types, the complete environment for such an approach would be:

$$KM3 : Metametamodel,$$
$$SimpleLanguage : KM3,$$
$$SLExtractor : SimpleLanguage \xrightarrow{TCS} Text(SimpleLanguage),$$
$$ALT2SL : \Pi A:MTType.\Pi B:MTType.(A \xrightarrow{ATL} B) \xrightarrow{ATL} SimpleLanguage,$$
$$M : MTType,$$
$$N : MTType,$$
$$tmp := (ATL2SL \; (M \xrightarrow{ATL} N) \; SimpleLanguage \; (ATL2SL \; M \; N)) : SimpleLanguage,$$
$$ATL2SLText := (SLExtractor \; tmp) : Text(SimpleLanguage)$$

The use of arbitrary types is only applicable when the target type of the transformation, in this case $ATL2SL$, does not depend on the parameters. When the target type does depend on the parameters, they are obviously required. In the discussion section, at the end of this chapter, we discuss this latter case in more detail.

In conclusion, parametric types provide the desired form of genericity (i.e., a HOT transformation accepting as an argument any transformation of one concrete language). However, this feature, inherited from the type-by-reference model approach originally chosen for GMM, and which completely relies on the non-executability of transformation
arguments, has its pros and cons. On the one hand, it enables a family of interesting applications, such as the case study discussed in this section or the case of measuring ATL transformations [Vig09a]. On the other hand, it requires cumbersome type definitions or inference approaches.

6.3 Metamodel Adaptation

An interesting scenario for discussing the typing of weaving models is the model adaptation problem as presented in [GJCB09]. This problem was originally investigated by Sprinkle [SAL+02] using graph transformations, and the weaving-based solution was addressed by us in [VB09b]. When a metamodel $\text{MM}_a$ evolves into a metamodel $\text{MM}_b$, the concern is to adapt any terminal model $\text{M}_a$ conforming to $\text{MM}_a$ to the new metamodel
6.3 Metamodel Adaptation

version $\text{MM}_b$. The proposed solution is a three-step adaptation illustrated in Figure 6.5. First, a matching process realized by transformation $\text{Matching}$ computes the equivalences and changes between $\text{MM}_a$ and $\text{MM}_b$. Second, an adaptation transformation $\text{AdaptMM}_a\text{MM}_b$ is derived from those discovered equivalences and changes by transformation $\text{AdaptationGeneration}$. Finally, transformation $\text{AdaptMM}_a\text{MM}_b$ from terminal model $\text{Ma}$ produces terminal model $\text{Mb}$, which conforms to metamodel $\text{MM}_b$.

Equivalences and changes between metamodels are expressed by means of a weaving model $\text{matchMM}_a\text{MM}_b$ conforming to the $\text{Match}$ weaving metamodel extension. $\text{Match}$ introduces different variants of links for indicating those model elements that are present in both metamodels, and those which were added or deleted from $\text{MM}_b$ with respect to $\text{MM}_a$. In this case woven models are not terminal models; they are metamodels instead.

The main artifacts involved in this application scenario include the ECore-based $\text{Match}$ weaving metamodel extension, the two KM3-based versions $\text{PetriNetsV1}$ and $\text{PetriNetsV2}$ of the Petri Nets metamodel, the ATL (DTT) transformation $\text{Matching}$ which produces matching weaving models, the ATL (HOT) transformation $\text{AdaptGen}$ which is an adaptation transformation generator, and the ATL adaptation transformation $\text{AdaptPNV1PNV2}$. As before, we directly show a concrete environment containing the declarations representing the artifacts described above and all other artifacts required for a complete execution of the case study. Although for an environment $\Gamma$ declarations in $\Gamma_c$ are separated from declarations in $\Gamma_e$, in the environment below such declarations were intermixed for clarity:

\begin{verbatim}
ECORE : Metametamodel,
AMWCore : ECore,
Match : ECore,
Match <: : AMWCore,
KM3 : Metametamodel,
PetriNetsV1 : KM3,
PetriNetsV2 : KM3,
Matching : A:KM3×B:KM3 $\xrightarrow{\text{ATL}}$ AMWWaving(Match,A::KM3,B::KM3),
matchPNV1PNV2 := (Matching (PetriNetsV1,PetriNetsV2))
 : AMWWaving(Match,PetriNetsV1::KM3,PetriNetsV2::KM3),
AdaptGen : PIX:KM3.PIY:KM3.AMWWaving(Match,X::KM3,Y::KM3) $\xrightarrow{\text{ATL}}$ X$\xrightarrow{\text{ATL}}$ Y
AdaptPNV1PNV2 := (AdaptGen PetriNetsV1 PetriNetsV2 matchPNV1PNV2)
 : PetriNetsV1$\xrightarrow{\text{ATL}}$PetriNetsV2,
PNV1 : PetriNetsV1,
PNV2 := (AdaptPNV1PNV2 PNV1) : PetriNetsV2
\end{verbatim}

In the environment above, using transformations $\text{Matching}$ and $\text{AdaptGen}$, we pro-
duced transformation $\text{AdaptPNV1PNV2}$, which generates terminal models conforming to metamodel $\text{PetriNetsV2}$ from terminal models conforming to $\text{PetriNetsV1}$. In particular, terminal model $\text{pnV2}$ is an adaptation of terminal model $\text{pnV1}$. Note that $\text{Matching}$ is a DTT, but it is not a HOT. In turn, $\text{AdaptGen}$ is a parametric HOT.

Note that the occurrence of both $A$ and $B$ in the target type of $\text{Matching}$ are bound to their declaration in its source. Therefore, they are to be replaced when $\text{Matching}$ is applied and concrete values are provided for them. The target type of $\text{AdaptGen}$ is expressed in terms of the woven models (i.e., parameters $X$ and $Y$), and not in terms of the woven types. This is enabled by the fact that, in this case, woven models are metamodels (i.e., types). Such woven models (i.e., metamodels $\text{PetriNetsV1}$ and $\text{PetriNetsV2}$) were used in the definition of $\text{AdaptPNV1PNV2}$ for instantiating $X$ and $Y$ respectively.

The type of $\text{matchPNV1PNV2}$ not only provides more information about it, but it also provides a finer classification of weaving models. Formerly, with the original typing approach, its type was plainly $\text{Match}$. If additionally considering two versions of the Relational metamodel, namely $\text{RelationalV1}$ and $\text{RelationalV2}$, with our typing approach a possible intermediate weaving model $\text{matchRelV1RelV2}$ would be of type $\text{AMWWeaving} (\text{Match}, \text{RelationalV1}::\text{KM3}, \text{RelationalV2}::\text{KM3})$, which is a different type of $\text{matchPNV1PNV2}$'s.

### 6.4 Automatic Layout

In this section we demonstrate our approach on the case study presented in [DJ07]. The problem addressed in that case study refers to the modification of the layout of model elements within a class diagram. An element within a class diagram may contain coordinates information which is used for placing the element at a specific location when the diagram is rendered as a graphic. Regarding the diagram as a graph, an auto-layout algorithm for properly placing model elements for display can be implemented. Graphviz/Dot implements one such algorithm [GKN06]. However, the algorithm operates on a Dot textual representation of the graph to be processed. The concrete choices are: (a) implementing an auto-layout algorithm which directly operates on the class diagram, and (b) reusing the Graphviz/Dot algorithm. Assuming that alternative (b) is preferable, this poses the problem of feeding the algorithm with appropriate input [DSGJ09]. Dot grammar is expressed in EBNF and therefore the problem reduces to bridging the MDE and the Grammarware technical spaces. The solution consists of some model-to-model transformations and projectors.

Figure 6.6 illustrates all required artifacts. A Sample Class Diagram is transformed, via the Notation2Dot model-to-model ATL transformation, to a model representation of a graph conforming to Dot. Such a model representation is extracted, via the TCS DotExtractor extractor, to a textual entity representing the same graph. The auto-layout
program Dot.exe inserts appropriate layout coordinates to the graph, which is injected by the TCS DotInjector injector back to the MDE technical space. Both the model representation of the graph with layout coordinates and the original class diagram are merged by the Merge ATL transformation for producing the Final Class Diagram.

Now we discuss the environment and each cGMM term representing an artifact of this case study:

\[
\begin{align*}
\text{ECore} & : \text{Metametamodel}, \\
\text{Notation} & : \text{ECore}, \\
\text{Dot} & : \text{ECore}, \\
\text{Notation2Dot} & : \text{Notation}^{\text{ATL}} \rightarrow \text{Dot}, \\
\text{DotExt} & : \text{Dot} \xrightarrow{\text{TCS}} \text{Text}(\text{Dot}), \\
\text{DotProg} & : \text{Text}(\text{Dot})^{\text{EXE}} \rightarrow \text{Text}(\text{Dot}), \\
\text{DotInj} & : \text{Text}(\text{Dot}) \xrightarrow{\text{TCS}} \text{Dot}, \\
\text{Merge} & : \text{Notation} \times \text{Dot}^{\text{ATL}} \rightarrow \text{Notation}, \\
\text{sampleCD} & : \text{Notation}, \\
\text{sampleGrM} & := (\text{Notation2Dot sampleCD}) : \text{Dot}, \\
\text{sampleGrT} & := (\text{DotExt sampleGrM}) : \text{Text}(\text{Dot}), \\
\text{sampleGrTwC} & := (\text{DotProg sampleGrT}) : \text{Text}(\text{Dot}), \\
\text{sampleGrMwC} & := (\text{DotInj sampleGrTwC}) : \text{Dot}, \\
\text{finalCD} & := (\text{Merge \{sampleCD,sampleGrMwC\}}) : \text{Notation}
\end{align*}
\]
In the environment above, it can be clearly appreciated that in addition to the initial sampleCD and the final finalCD terminal models, a large amount of intermediate artifacts were generated. In fact, applying the complete process to another class diagram will produce a complete set of other intermediate artifacts. In this way, the megamodel could become unnecessarily crowded. Since in cGMM we can even type programs such as DotProg, we are able to define a term representing a composite transformation which models the complete process. Clearly, such a definition makes no explicit use of intermediate artifacts as the approach discussed above. To that end, we define the composite transformation AutoLayout, which replaces all intermediate artifacts in the environment above, and an alternate way of defining finalCD:

\[
AutoLayout := \lambda_{ct} x: \text{Notation.} \\
(Merge \langle x, (\text{DotInj} (\text{DotProg} (\text{DotExt} (\text{Notation2Dot} x))))\rangle): \text{Notation,} \\
sampleCD : \text{Notation,} \\
finalCD := (AutoLayout \ sampleCD) : \text{Notation}
\]

The type of the result of AutoLayout, and in this case of finalCD, is Notation, as expected.

### 6.5 Promotion and Demotion

When discussing technical spaces in Section 2.2, we mentioned that the MDE technical space includes different metametamodels, each of which induces a concrete model-based technical space. In this way, the MDE technical space can be actually regarded as a family of such technical spaces.

MDE-based tools physically handle models through software components called model handlers. A model handler provides primitives for creating new models, loading and saving models, adding and deleting model elements, and reading and writing model element properties. A model handler is specialized in one single metametamodel, that is, it supports such primitives for a specific model format. For example, the ATL virtual machine includes model handlers for EMF, UML2 and NetBeans/MDR. This means that it is capable of directly handling models expressed in ECore, MOF and MDR formats. In turn, the AM3 tool currently includes only one single model handler for EMF, the am3.modelhandler.emf plug-in [Mod09b]. However, GMM is intended to manage artifacts corresponding to MDE technical spaces other than EMF as well. In fact, a Kernel Modeling Framework (KMF) [JBB09] is being developed. KMF provides common interfaces for model manipulation, which may be realized by a number of different implementations. Currently, a default implementation based on the KM3 metameta-
Figure 6.7: Promotion and demotion of a KM3 metamodel in EMF

model is provided, but other implementations based on other metametamodels can be supported. This way, the KMF modeling framework is regarded as metametamodel agnostic. However, for the time being, only the EMF technical space is natively supported by AM3.

Providing support for non-native MDE technical spaces (i.e., bridging MDE technical spaces) can be achieved by some promotion and demotion mechanisms [BBC+05]. They are illustrated in Figure 6.7. If ECore is the only metametamodel, the KM3 metametamodel may be represented as a metamodel conforming to ECore (i.e., KM3 : ECore). Then, a KM3 metamodel MM is represented as a terminal model conforming to the metamodel version of KM3 (i.e., MM : KM3). Whenever needed, MM (which is actually a terminal model in M1) may be transformed by a proper transformation (in this case KM32EMF [ATL09]), to an actual ECore-conforming metamodel MM’ (i.e., MM’ : ECore, thus making MM’ an actual metamodel in M2). This mechanism is called promotion. Then, a terminal model M which is supposed to conform to MM is defined to actually conform to MM’. Demotion is the inverse operation and it is analogously defined, based, in this case, on transformation EMF2KM3 [ATL09].

We believe it is interesting taking a look to such mechanisms from the precise typing perspective of cGMM. The following environment represents the abstract promotion
situation described above:

\[
\begin{align*}
ECore &: \text{Metametamodel}, \\
KM3 &: \text{ECore}, \\
KM32EMF &: \text{KM3} \xrightarrow{\text{ATL}} \text{ECore}, \\
EMF2KM3 &: \text{ECore} \xrightarrow{\text{ATL}} \text{KM3}, \\
MM &: \text{KM3}, \\
MM' &: (\text{KM32EMF} \ MM) : \text{ECore}, \\
M &: MM'
\end{align*}
\]

Even though it is not actually used above, note that \text{EMF2KM3} is not a DTT transformation, since its target type does not depend on the input value (which is not even specified). Finally, using the type rules introduced in Chapter 4, we could easily prove, under the above environment, that \( MM':\text{Metametamodel} \).

6.6 Discussion

In this final section we discuss a number of issues we have identified while addressing different case studies. Concerning the example studied in Section 6.1, we realized that in GMM a copier for ATL transformation models can be generated by an application of the \text{KM32ATLCopier} transformation to the ATL metamodel. However, in \text{cGMM} the transformation may only produce copiers of models directly typed by KM3-based metamodels, and thus ATL transformations are excluded. This is because the expected type of an ATL copier is \( \Pi A:\text{MTType}\. \Pi B:\text{MTType}. (A\xrightarrow{\text{ATL}} B) \xrightarrow{\text{ATL}} A\xrightarrow{\text{ATL}} B \), which may not be produced by an application of \text{KM32ATLCopier}. It is important to make a distinction between parameterized types and dependent types. In the former case, a type parameter is instantiated using information provided in the invocation. In the latter, arguments are bound to actual parameters (which carry usable type information as well). An instantiation does not involve artifacts at runtime, while a binding does. The type of the copier produced by \text{KM32ATLCopier} would be \( \text{ATL} \xrightarrow{\text{ATL}} \text{ATL} \). Even though the definition of such a copier is appropriate, the type system will not validate its application on a transformation. The case of copiers for transformations needs to be separately addressed. As stated by Pierce, enabling language features without a precise typechecking in mind may rise tricky situations \cite{Pie02}.

In Section 6.2, we addressed issues concerning the inference of type information for type instantiation. Inference is discussed by Cardelli \cite{Car86b} and implemented in System CoQ \cite{CPT09} as a feature called implicit arguments. As we already discussed, in \text{cGMM} information cannot be inferred in cases like the one we addressed. However, parameters not occurring in the resulting type enabled different approaches. Instead,
when parameters do occur in the resulting type, eliminating unused parameters by using arbitrary types is not possible, and the typechecker should signal the problem. We provide next an example of such a case:

\[
S : \Pi A:MTType.\Pi B:MTType.(A^{\text{ATL}} B) \rightarrow \text{ATL} A,
\]

\[
T := \Lambda X:MTType.\Lambda Y:MTType.(S (X^{\text{ATL}} Y) X (S X Y))
\]

\[
: \Pi X:MTType.\Pi Y:MTType.X^{\text{ATL}} Y
\]

In this case, parameters \(X\) and \(Y\) occur in the inferred type, in particular, such a type is expressed in terms of them. For this reason, following the described approach is incorrect (the inferred type would be expressed in terms of those arbitrary types). Information on the outer application of \(S\) can be inferred if the information on the inner application is explicitly provided.

The type of assumed constants is necessarily provided by the user. One may argue that, in cases such as \(\text{KM32ATLCopier}\), the term that types an assumed constant may be too complex to define for modelers lacking formal background. GMM enables the use of dependently typed higher-order transformations, and that complexity is inevitably projected to their types. Such a complexity is inherent to GMM and is not introduced by our calculus. Note that manifest typing is a common practice in many typed programming languages such as Java or C#. A balance between explicitly and implicitly generated type annotations should be pursued [Pie02]. We believe that suitable discovery mechanisms such as MoDisco [BJM10] or AM3’s Megamodelling Discovery feature [Mod09b] would, at least, assist users in the type specification process.
Chapter 7

Typing MDE Artifacts

In the previous chapters we defined the \( \text{cGMM} \) calculus and proved a number of its properties, and we applied such definitions to different real-world case studies which posed situations that challenged the current version of GMM. That concludes the presentation of our contribution from a theoretical perspective. However, as presented earlier in this thesis, the GMM approach has a deep practical motivation. In fact, the AM3 Project [AM309] is dedicated to bringing the GMM ideas to life by means of the AM3 tool. In turn, our work was motivated by precisely the limitations such a tool exhibits in some real-world applications. So far we were able to overcome such limitations, but at the level of Formal Methods. The practical contribution of our work is achieved when we shift to Software Engineering and realize our ideas as a working tool. Moreover, our ultimate practical contribution is concluded when such an asset is integrated with the tool that originated our work.

This chapter is focused on the application of our work as perceived by an MDE-based software developer who manages his/her artifacts following the GMM approach in real-world situations using the AM3 tool. In that context, we discuss how the \( \text{cGMM} \) calculus can be used for actually typing MDE artifacts within a GMM tool, and characterize the concrete functionalities that such a tool should expect from a realization of our type system. The implementation of a type system is a hard task, even if it is based on formal specifications. A correct type system should validate well-formed programs and enable their execution. In addition, a useful type system must be correct, but also it must be able to detect ill-formed programs and prevent their execution, ideally, providing precise reasons to that. Even though the proof of its properties provide confidence on the definitions that constitute a type system, its implementation and practical application is what shows if it fulfills its ultimate goal. Field experimentation provides valuable feedback, especially when a type system is defined and implemented from scratch. The notion of pluggable type systems introduced by Bracha [Bra04] enabled a realization approach that
yielded the TypeSystem4GMM extension plug-in for AM3, our ultimate implementation goal, and a self-contained version of the type system which provided rich feedback during construction for validation purposes. The latter version does not address computations as AM3 does, and focuses specifically on typing only. It provides a user interface and allows a human user to request typechecking on artifacts as AM3 would do. Both tools share a common component, which is the actual responsible of providing the type-related functionalities. On this basis, most of the maintenance effort falls on that component, while tool-specific modifications can be realized at a reasonable low cost. Furthermore, this loosely coupled integration between the type system and AM3 enables separation of concerns. In this way we are able to experiment on the type system without losing focus on it, while encompassing AM3’s evolution, still relying on the help and benefits of the interactive version.

The remainder of this chapter is structured as follows. In Section 7.1, we describe our approach to typing GMM artifacts in a practical context and specify the functionalities that a realization of our type system provides to GMM-based environments or tools. Section 7.2 deals with the realization of our type system. We address the architecture of the toolset that was generated, composed of two separate versions of the type system. We close in Section 7.3 with a discussion on some issues concerning the design and implementation of our tools.

7.1 Our Approach to Typing

The goal of a type system is to prevent type errors to occur during the execution of a program. Typing can be informally addressed by agreeing upon some notion of type for program elements and instructing the compiler or runtime environment to detect and signal some undesired conditions on the types of such elements. Such an approach relies on both the appropriateness of the definitions of types and the conditions to catch. Since such definitions are informally and usually poorly documented, detecting all possible undesired conditions using those elements is a hard task to accomplish. In Chapter 2 we showed that an informal typing approach in some specific situations leads to programs where, from some point of their execution on, their behavior becomes unpredictable. In turn, a formal type system is defined by precise definitions which enable the detailed proof of properties which give confidence on those definitions. With the definition of the cGMM calculus in Chapter 4 and the proof of some of its properties in Chapter 5, we follow the latter approach for typing MDE artifacts.

Our approach to typing MDE artifacts using cGMM is based on mirroring elements within a megamodel. For every element there is a term that represents it and carries its type, and for the megamodel containing those elements there is an environment that contains the homologous terms. Elements range from entities (e.g., models) to relationships
7.1 Our Approach to Typing

A megamodel in AM3 containing elements and their mapping to an environment and its corresponding terms

(e.g., transformation records representing transformation applications). The containing megamodel is a living artifact. Its collection of elements evolves over time as elements are inserted or removed, and transformations (which produce new elements) are executed. Terms embody a typed version of elements, and a proper use of type information guards the evolution of a megamodel. If a typed homologue $t$ of an element $e$ can be built in the context of the environment $\Gamma$, which corresponds to megamodel $\mathfrak{M}$, then it means that $e$ will not cause a type error and it may be safely included in $\mathfrak{M}$; otherwise, the integrity and further behavior of $\mathfrak{M}$ is compromised.

A megamodel corresponds to an environment and elements correspond to terms. The latter correspondence follows the mapping introduced along Chapter 4, and illustrated in Chapter 6, between GMM and cGMM constructs. Figure 7.1 illustrates an example of such a mapping. On the left side, AM3’s Megamodel Navigator (one of the panes in the user interface illustrated in Figure 2.8) shows the entities (only) involved in the Class-to-Relational model transformation [ATL09]. Other elements, more specifically relationships, are not shown. On the right side, the corresponding cGMM environment contains all the required terms. Entities in the megamodel navigator display their names and the specific variant of entity they are (e.g., Terminal Model, ATL Module, etc.), while their types, according to $:c_2$, are not shown. The types of entities are shown as annotations of the corresponding terms. Some elements have dual representations in AM3, but such representations ultimately refer to the same conceptual element. In particular, the Class2Relational ATL module is associated to a relationship, whose information exactly
7.1 Our Approach to Typing

maps to the type of the Class2Relational constant. Note that this is a particular case, as discussed in Chapter 2, since Class2Relational in neither a HOT nor a DTT. If the transformation was in either category then the mapping would have required additional information. In turn, the body of the definition of constant aRel (i.e., the application of Class2Relational to aClass) maps to a transformation record, which is a relationship, that has terminal model aRel as its target model. Note that any constant name occurring in the declaration of another constant, that is, at the right side of a ‘:’ or a ‘:=’, has been necessarily declared in the environment before that constant. The only exception to this is constant Metametamodel which is predefined in cGMM. In this way, we associate GMM elements and their manipulation, via terms, to cGMM types. The typechecking algorithm, built on the type rules presented in Chapter 4, then completes the job.

The diagram in Figure 7.2 shows the use cases for typing MDE artifacts with a realization of cGMM, which we refer to as TypeSystem. A generic consumer of such functionalities is a GMM Environment. Concrete consumers are a GMM Tool, such as AM3, and an Interactive FrontEnd, which enables a user directly operating the type system. The Declare Constant use case enables feeding the environment with constants,
and is thus the main functionality. Since constants can be either assumed or defined, there exist two varieties of this use case: Assume Constant and Define Constant use cases are specific forms of declaring constants. The Assume Constant use case consists in inserting a term of the form \( c : T \) into an environment \( \Gamma \). It involves typechecking subterm \( T \), that is, determining whether it is a valid term, and then determining whether such a term is actually a type. In the example of Figure 7.1, the assumption of constant \( \text{Class2Relational} \) required that the provided type was a valid term, which in turn required both \( \text{Class} \) and \( \text{Relational} \) to be valid terms, and particularly types. By already being in the environment, the type system assumes that necessary typechecks were already performed and thus their validity is not checked again. However, the type system does check that such terms are types, as required by the (II-ATL Formation) rule. Since they are typed by \( \text{KM3} \) and it is a metamodel, they both satisfy the condition. After this operation, the type system signals either a type error or success. In the latter case, constant \( c \) is inserted into \( \Gamma \) and its type is assumed to be \( T \).

The Define Constant use case consists in inserting a term of the form \( c := t : T \) into an environment \( \Gamma \). Note that the information provided by the actor in this case is just \( c := t \). This use case involves typechecking subterm \( t \) (here \( t \) is not expected to be a type), and inferring subterm \( T \), that is, the type of \( t \). In the example of Figure 7.1, the definition of constant \( aRel \), which is the application of \( \text{Class2Relational} \) to \( aClass \), involves two steps. The first step is making sure that \( \text{Class2Relational} \) is a transformation and that \( aClass \) is of the appropriate type, as required by rule (II-ATL Elimination). Here, the ATL-based variant of that rule is used because \( \text{Class2Relational} \) is an ATL transformation. The second step is inferring a type for the application, and hence for constant \( aRel \). By means of the (II-ATL Elimination) rule, the result of the type inference is \( \text{Relational} \). After this operation, the type system signals a type error or returns the inferred type. In the latter case, constant \( c \) is inserted into \( \Gamma \) and its type is assumed to be \( T \). Since the type system does not address computations, subterm \( t \) becomes irrelevant for further uses of \( c \) and it served for inferring type \( T \) only. TypeSystem deals only with typing, but in a GMM tool such as AM3, \( t \) is also used for computing the actual value of \( c \).

The Check Constant use case returns the type of a constant in the environment. When a constant of the form \( c : T \) or \( c := t : T \) is included in an environment \( \Gamma \), checking constant \( c \) always returns type \( T \). This functionality may be regarded as redundant since type \( T \) was explicitly handled by the actor when constant \( c \) was declared. On the one hand, if constant \( c \) was assumed, the actor indeed provided type \( T \) when the Assume Constant was enacted. On the other hand, if constant \( c \) was defined, the type system returned type \( T \) to the actor as a result of the corresponding enactment of the Define Constant use case. In spite of that, we believe that checking the type of a constant is a useful functionality by itself, especially in an interactive context.

The Reset Constant use case removes a constant \( c \) from the environment \( \Gamma \). This includes removing any constant which may depend, either directly or indirectly, on con-
7.2 Tool Support

In the previous section we showed how a cGMM environment and terms can be used for typing GMM artifacts. Furthermore, we described the functionalities that a realization of our type system should provide to a GMM-based user environment. In Chapter 6 such functionalities were applied to several case studies, demonstrating the theoretical value of cGMM. However in those applications, type system’s machinery was manually driven. For example, type derivations in Figures 6.2 and 6.3 were carried out and checked by hand. This use of cGMM by a developer resembles the work of a mathematician when writing a proof on paper. Manual typechecking and type inference have a number of drawbacks. On the one hand those are long and tedious processes which are error-prone. On the other hand, in the case of formal type systems, the ability of mechanizing type-related tasks is not exploited. The practical value of cGMM is achieved when the type system is implemented, and moreover, when such an implementation is integrated with a GMM tool.

In this section we address the realization of our type system. Such a process yielded two separate tools. On the one hand, we developed a self-contained application which realizes the type system. Such an application provides a front end that enables developers to interact with the type system through a command line user interface. On the other hand, we developed a version of the type system that integrates with AM3. Both tools share a common core component which may be also reused in other contexts. We start by discussing the architecture of our toolset from a logical point of view, and then we address its main components. The shared core component is reviewed first, and the components that build each tool are separately treated next.

7.2.1 Architecture

Type systems are usually implemented within compilers or interpreters [Car97]. In fact, this is the case of most of the widely used programming languages such as the Java language. However, this does not necessarily mean that type system’s code needs to be scattered across the code of the compiler or interpreter. Bracha introduced the notion of pluggable type systems [Bra04]. Such type systems are naturally conceived as independent components, as if they were aspects of programming languages. Although the main intent of this approach is to enable different type systems to be plugged into a language for
targeting different program requirements, another of its features appealed us. Pluggable type systems can evolve independently of the underlying language. As a consequence, researchers can use such a framework to experiment with novel type systems \cite{ANMM06}. This scenario fits well our context.

Our ultimate goal is to integrate our type system with AM3, but we would prefer a separate implementation. This approach is indeed aligned to AM3’s architecture. By being an Eclipse component, AM3 is based on a set of Eclipse plug-ins, each of them having a specific role \cite[Mod09b, p. 8]{Mod09b}. Our type system is then accordingly implemented as an AM3 plug-in. As an intermediate step towards this goal, and for practical purposes, we have developed a self-contained version of the type system. The intent behind this approach is providing an environment where direct interaction of a human user with the type system is possible. This scheme enabled us to focus on the type system itself, and the resulting tool proved to be a valuable asset during the testing and debugging processes.

The interactive tool and the AM3 plug-in share a common component which embodies the core realization of the type system. The diagram in Figure 7.3 shows the top-level structure of the logical view of our toolset. The \texttt{TypeSystem} component is the core component, and exhibits the behavior specified in Figure 7.2. In turn, the \texttt{FrontEnd} component corresponds to the Interactive \texttt{FrontEnd} actor. Finally, the AM3 component corresponds to the GMM Tool actor. The TSPlugin component acts as a sort of \textit{Adapter} between the \texttt{ITypeSystem} and IGMM\texttt{TypeSystem} interfaces. In fact, functionalities exposed by the latter are a subset of those exposed by the former, as a GMM tool requires
fewer functionalities from the type system than an interactive front end. In the remainder of this section we review and discuss the most relevant issues regarding the TypeSystem, FrontEnd and TSPlugin components and their respective manifestations.

7.2.2 The Core of Our Type System

The TypeSystem component is the heart of our toolset. It realizes all the functionalities accessed by the AM3 (via TSPlugin) and FrontEnd components. The ITypeSystem interface realized by the TypeSystem component is the single entry point to the type system which follows a sort of Command design pattern. Input is provided to the component as strings which are processed by an AntLR-based parser in successive calls to the execute operation (using Command’s terminology). Such strings constitute commands that manipulate the underlying environment. Commands involve terms, and for expressing them we developed a textual language similar to Gallina [CPT09], the specification language of Coq. The syntax of such a language closely follows the syntax used in Chapter 6 for expressing terms. In turn, for expressing commands we developed a textual command language similar to The Vernacular [CPT09], the command language of Gallina. It includes a set of commands that enable the functionalities described before in Section 7.1. As an example, the following command sequence yields the environment Γ illustrated in the right part of Figure 7.1:

Assume KM3 : Metametamodel.
Assume Class : KM3.
Assume Relational : KM3.
Assume Class2Relational : Class->Relational.
Assume aClass : Class.
Define aRel := (Class2Relational aClass).

Commands Assume and Define are used for declaring constants. Note that the arrow symbol '->' occurring in the assumption of Class2Relational is not annotated with the name of the transformation language used for defining such a transformation. Since composite transformations may not be assumed, the type of injectors involves textual types (which do not occur in this declaration), and there is no transformation language currently supported by GMM other that ATL, then we conclude that the transformation language of Class2Relational must be ATL and no confusion can occur. This temporary simplification was made for clarity reasons as it greatly improves readability. In a future situation, when support for other transformation languages is required, such a simplification should be removed and suitable mechanisms for annotating arrow symbols in the textual syntax must be developed. The type of assumed constants must be explicitly provided as part of the Assume command. In turn, the type of the term that
7.2 Tool Support

defines a constant, such as $a_{Rel}$, does not need to be provided as part of the Define command since it is inferred by the type system. Such a type is returned as a result of the processing of the definition, or it can be explicitly requested to the type system by issuing a Check command (e.g., Check $a_{Rel}$).

From a logical perspective, the TypeSystem component is realized by a set of classifiers which are organized into the design packages illustrated in Figure 7.4. The core package introduces the Term abstract class and its variants according to Definition 4.1.2. Such variants are either concrete or abstract. For example, a projection, defined by clause number 9, is represented by the Projection concrete class. In turn, an application, defined by clause number 6, is represented by the Application abstract class. Since there are different rules for applications of transformations expressed in different languages, which have different premises (e.g., ($\Pi$-ATL Elimination), ($\Pi$-CT Elimination), etc.), there are different concrete variants of Application. These variants are ApplicationATL, ApplicationCT, and so on, and they are defined in packages atl, ct, and so on, respectively. Note that the ($\Pi$-ATL Elimination) rule specifies how an application of an ATL trans-

Figure 7.4: Classifiers realizing the behavior of the TypeSystem component
formation can be constructed. The premises of such a rule are mapped to preconditions of the constructor of the ApplicationATL class. In other words, an application object may not be constructed if the conditions that satisfy the premises of the corresponding rule are not met. As terms involve subterms, the Term class and its variants follow the Composite design pattern.

Packages amw, tcs, ct and so on, at the bottom of Figure 7.4, include concrete classes that realize terms associated to different languages or technologies. For example, package tcs includes classes for realizing the TCS-related terms involved in the rules specified in Section 4.2.7.3. The environment package defines the Environment class. Such a class is at the same time a container ($\Gamma_c$) of Constant objects and a container ($\Gamma_{<,}$) of SubtypeDeclaration objects (see Definition 4.2.1). Constant is a variant of Term, and is further specialized in Sort and Declaration. Sort objects represent sorts in $S$ (see Definition 4.1.1), and Declaration is specialized in Assumption and Definition. The exception package introduces the TypeException exception which is thrown when a type error occurs. Finally, the parser package introduces the AntLR generated classes cGMMLexer and cGMMParser. The latter class realizes the ITypeSystem interface and thus receives the commands sent from outer components.

From an implementation perspective, the TypeSystem component is manifested by the cGMM.jar artifact. Such an artifact packages together all the Java (bytecode) files that constitute the implementation of such a component, as well as the antlr-3.3.jar artifact. This latter artifact is the manifestation of the AntLR component shown in Figure 7.3, that is, the runtime of AntLR. The cGMM.jar artifact is central to both of the tools discussed next.

### 7.2.3 An Interactive Type System

As an intermediate step towards the development of a type system for AM3, we developed a stand-alone version of such a type system. This interactive version is made of components FrontEnd and TypeSystem specified in Figure 7.3. Using this tool, a human user may simulate the interaction of the type system with a GMM tool by properly issuing commands through a user interface, but without the burden of managing actual GMM elements. This approach enabled us focusing on the critical functionalities for validating the realization and manifestation of the TypeSystem component. Given the component connections shown in Figure 7.3, this validation applies at the same time to the version of the type system which is to be integrated with AM3.

From a logical perspective, the FrontEnd component is rather simple. It defines an object that loops reading text from the standard input and forwarding it as a command to the TypeSystem component, until a Quit command is entered. This scheme has an additional benefit beyond simplicity: no specific changes are required for encompassing TypeSystem’s evolution. This means that when changes to such a component occur, even
7.2 Tool Support

at the grammar level, the current manifestation of the FrontEnd component will still fit its purpose.

From an implementation perspective, the FrontEnd component is manifested by the frontend.jar artifact. As illustrated in Figure 7.5, such an artifact is independent from cGMM.jar, the manifestation of the TypeSystem component, and constitutes a basic presentation layer for our interactive tool.

Figure 7.6 shows a screenshot of the console-based user interface of the interactive type system. It depicts the commands involved in an actual work session for the KM32ATLCopier case study addressed in Section 6.1. Commands are issued here by a human user through the console. If executing such a case study on a concrete GMM tool, such a tool would send the exact same commands to the type system. In the case of an Assume X command, the message X is assumed is interpreted as a successful type-check. In turn, in the case of a Define X := Y command, the tool would receive back the result of a Check X, that is, the inferred type of Y, as a sign of success. Additionally, we intentionally introduced two common error scenarios for illustrating a very important aspect of a type system: error handling. In the first scenario, we applied SQLCopier to an argument of the wrong type. The error message indicates that the received type and the expected type do not match. In the second scenario, we tried to apply s1, which is not a transformation, to s2. The error message indicates that the applied term is not executable.

For convenience, we also included a Load command which reads and batch processes a script file with .cgmm extension. The contents of such scripts are sequences of commands such as that in Section 7.2.2 for the Class2Relational example. Although our interactive tool is a type system and not a complete GMM environment that computes actual values for elements, this functionality has a program execution flavor. Our command language may be regarded as a form of programming language, and the notion of “megamodel programming” may lead to interesting and powerful applications.
7.2 Tool Support

7.2.4 Integration with AM3

Our type system was specifically defined for working with GMM. In order to fully exploit its benefits, our ultimate goal is to integrate its implementation with a tool that realizes the megamodeling approach, particularly AM3. Such a tool is based on a set of Eclipse plug-ins. As discussed in Section 2.3.4, plug-ins are classified as core, extensions or utilities. To fulfill our goal we introduce a new extension plug-in called TypeSystem4GMM, which fits into the plug-in architecture of AM3 as illustrated in Figure 7.7.

From a logical perspective, this version of the type system is made of components TSPlugIn and TypeSystem specified in Figure 7.3. When a type-related event occurs within the AM3 tool, it notifies our extension plug-in by means of an appropriate message from the IGMMTyplSystem interface realized by the TSPlugIn component. Such a component wraps the TypeSystem component and is a form of Adapter for the ITypeSys-
7.2 Tool Support

Figure 7.7: Plug-in architecture of AM3 including the extension plug-in

The tem interface. After the type system has processed the message, it returns the result in a suitable data structure, or throws a type error exception. In either case, AM3 has to deal with such a result in a proper way.

From an implementation perspective, components AM3 and TSPlugIn are not manifested by packed artifacts like TypeSystem and FrontEnd. Rather, their manifestations are groupings of fine-grained artifacts such as Java source and byte code files, configuration files, ECore files, and so on. Figure 7.8 shows the artifacts involved in the manifestation of our plug-in. Groups of fine-grained artifacts are represented by packages, while packed artifacts are represented by the usual class-based notation. Furthermore, white-colored artifacts are provided outside of the scope of this work by third parties. The globalmodelmanagement artifact corresponds to the GlobalModelManagement extension plug-in [Mod09b]. It is then a part of a larger aggregation of artifacts that as a whole manifests the AM3 component. This artifact includes our typesystem package that contains the definition of information to be used by AM3 (i.e., data structures and exceptions), and the TypeSystemProvider class provided by the AM3 project team. Such
7.2 Tool Support

Figure 7.8: Implementation view of the plug-in version of the cGMM tool

a class is a Factory that provides a Singleton object which is a Façade to the type system. The globalmodelmanagement artifact also includes the extensioninterfaces package, where the IGMMTTypeSystem interface is actually defined. In turn, the typesystem4gmm artifact manifests the TSPlugIn component, and corresponds to our TypeSystem4GMM extension plug-in. As indicated in Figure 7.7, TypeSystem4GMM is an extension of the GlobalModelManagement plug-in. Among others, the typesystem4gmm artifact contains the TypeSystemImpl class which realizes the IGMMTTypeSystem interface and constitutes the façade. It also contains classes for adapting the information that flows between the AM3 tool and the TypeSystem component.

The process of successfully connecting our type system to AM3 is twofold. First, the type system (i.e., cGMM.jar) needs to be turned into an AM3 extension plug-in. This is achieved by means of typesystem4gmm. This part of the process is completed. Second, a number of modifications are required at specific locations of some of the plug-ins that...
7.2 Tool Support

make the AM3 tool. This corresponds to the tool using the plug-in. Some of such modifications were already discussed above and are completed, others will have to be performed by the AM3 project team as a part of their future agenda. As a summary, such modifications include:

1. Extend the current GlobalModelManagement extension so that all the information needed by the type system for a successful evaluation can be represented within the underlying megamodel. This was already completed in collaboration with the AM3 project team. The IGMMTypeSystem interface and the contents of the typesystem package of Figure 7.8 were agreed and realized.

2. Modify the transformation executors in the GMM4ATL, GMM4TCS and GMM4CT extensions, so that the required information is provided to the TypeSystem4GMM extension. The result of its evaluation is then retrieved by the AM3 tool, in order to automatically fill the megamodel with the appropriate type information.

3. Update the corresponding editors within every extension in order to enable the new form of type information within user interfaces. In the current implementation, terms such as Class2Relational from Section 7.1, which is a first-order transformation, can already be created. However, for the KM32ATLCopier case from Section 7.2.3, which is both a HOT and a DTT at the same time, specifying KM3 as the source metamodel (i.e., DTT case) must introduce a variable, namely \( M \). Then, specifying ATL as the target metamodel (i.e., HOT case) should allow the user to express that \( M \) will be both the source and the target type of the resulting transformation.

As an illustration of the effect of our type system on AM3 when the integration is complete, we discuss the KM32ATLCopier case study in terms of what could be accomplished on the current version of AM3. We start with a minimal megamodel \( \mathcal{M} \) for that case study. Figure 7.9 shows in (a) and (b) two different resulting scenarios \( \mathcal{M}_1 \) and \( \mathcal{M}_2 \) respectively. Megamodel \( \mathcal{M}_2 \) contains the same elements as megamodel \( \mathcal{M}_1 \) with the addition of element \( b_8 \) within a red frame. The original megamodel \( \mathcal{M} \) registers the following elements:

- The KM3 metametamodel (elements \( a_1 \) and \( b_1 \)).
- The ATL metamodel (elements \( a_2 \) and \( b_2 \)).
- The SQL metamodel (elements \( a_3 \) and \( b_3 \)).
- The KM32ATLCopier model transformation (elements \( a_6 \) and \( b_6 \)), which is a relationship, and its dual KM32ATLCopier-Module transformation model (elements \( a_4 \) and \( b_4 \)), which is an entity.
7.2 Tool Support

Figure 7.9: Megamodels for the KM32ATLCopier transformation case study

After KM32ATLCopier is applied to SQL, according to the current AM3 implementation, the SQLCopierGeneration transformation record is created (elements a7 and b7). Its resulting model is the SQLCopier-Module transformation model (elements a5 and b5). Note however the difference between elements a5 and b5. As already discussed in Section 2.5.3, the current AM3 implementation is not capable of deriving a type for the result of the application of KM32ATLCopier to SQL, which, according to our type system, it should be SQL → SQL. As a result, AM3 does not know the types of the source and target parameters of SQLCopier-Module. In a5, but not in b5, such parameters could not be created. Additionally, its dual model transformation, that is, SQLCopier (element b8), was not created in M1 for the same reason. Megamodel M1 is the result of the execution of KM32ATLCopier on M in the current version of AM3. In turn, megamodel M2 results from the manual creation of the SQLCopier model transformation and SQLCopier-Module’s parameters on M1. After executing this example on megamodel M we would have expected megamodel M2 as a result, and not M1 which is incomplete. With the information provided by our type system, AM3 could automatically produce megamodel M2 in one step from M, that is, without manual intervention.
7.3 Discussion

Our type system, both in terms of its theoretical definition and its implementation, underwent a series of versions and revisions before taking its actual form. In fact, our first experiments on typing MDE artifacts were conducted using the Predicative Calculus of (Co)Inductive Constructions (pCic), which is the underlying formal language of Coq [CPT09] version V8.0 and up. In our beginnings, we believed that pCic could play the role of our current cGMM and used Coq in a similar way we ended up using our interactive tool presented in Section 7.2.3. But we soon discovered that pCic was too powerful (i.e., included unapplicable features) and too general (i.e., enabled undesired cases from a GMM perspective) for being directly applied to our problem, and also lacked some basic specific features (e.g., native support for multiple metametamodels) which were mandatory for our context. As soon as we realized that a similar calculus would fit our purposes, we started defining cGMM and implementing our toolset. Those first experiments with Coq deeply inspired us, and the commonalities between pCic and cGMM, and their associated tools, are not accidental.

There are many different styles for implementing a type system. The current version of AM3’s type system is an example of an implementation scattered across the interpreter. Dorn and Leavens [DL07] introduce a domain-specific language for writing type system implementations, which has been embedded in both Haskell and Scheme. In [Jon99], Jones presents Haskell implementation of a typechecker for Haskell itself. Such a tool was not intended for practical use but rather for specification purposes. We implemented a pluggable type system. However, we do not claim we found, even as a side effect, an approach to realize the ideas of [Bra04]. Such a proposal involves other concerns than developing a separate component that can be plugged. We could rather say that we developed a particular realization of a pluggable type system. A possible realization of that approach in full extent is [ANMM06], where custom type rules extend the type system of Java.

Classes contained in packages environment, core, amw, tcs, and so on in Figure 7.4 could be understood as metaclasses in a metamodel describing types for GMM (i.e., a metamodel for our textual language for terms). This approach suggests the possibility of a model-based realization of our type system. This, in fact, is aligned with the way other AM3 plug-ins are actually realized. The state of the type system using such an alternative realization would be made, from a runtime point of view, of model elements instead of plain Java objects. As a consequence, state changes, which involve typechecking and type inference, could be achieved by means of (composite) model transformations. Based on the feasibility suggested by our experiments, we believe that it would be indeed a novel approach to implementing a type system. Unlike our current approach which goes beyond the borders of the MDE technical space, it would keep the solution to the problem of both managing and typing artifacts completely within such a technical space. Furthermore, if
the interactive version is still to be useful, TCS injectors and extractors could be defined for using a textual representation. This would require the definition of a metamodel for our command language. However, as discussed before, the command language heavily relies on the term language. Therefore, such a metamodel could be easily built extending that for terms.

Finally, as discussed in Section 7.1, unlike AM3, our tools do not deal with computations of values (here by values we actually mean artifacts). They deal with their typing only. Recall from Section 2.4.2, a calculus or formal system is uninterpreted, and a semantics is required for computing values by means of reductions. A calculus with semantics becomes a formal language. The semantics we introduced in Section 5.2.1 was not used in the implementation of our tools. Rather, it served us for proving the soundness theorem, which is one of the most important properties of cGMM. Typechecking and type inference are based on type rules only. Implementing our reduction rules would have turned our tools into interpreters of the command language. However, it would not be a full interpretation. Note that normal forms are always expressed in terms of applications of the Δ function to abstract (i.e., assumed) constants. Building an interpreter of megamodel manipulations goes beyond the scope of this work. We already have AM3 for those purposes.
Chapter 8

Conclusions and Further Work

In this thesis, a formal type system for Global Model Management and the integration with the AM3 tool of its implementation were introduced. This chapter presents our conclusions and perspectives of future directions. The main results of our work are summarized in Section 8.1. In Section 8.2, we discuss our contributions and draw some conclusions. Section 8.3 closes with an outline of future work.

8.1 Summary

We started with an overview and a discussion of Global Model Management and type systems, and a detailed account of their connection. To the best of our knowledge, that is one of the deepest discussions on GMM since its introduction. Although GMM explicitly considers the typing of artifacts, we found that such a specification is both informal and incomplete. In fact, we identified a number of limitations in the current typing approach that enable some ill-formed configurations to go unnoticed until they cause a type error. Particularly, one such limitation is manifested when dealing with a special kind of model transformation, which we named dependently typed transformations. The problem addressed in our thesis was that, under some specific conditions, GMM tools may behave unpredictably. Our proposed solution was based on a new typing approach that captures such conditions and prevents further tool execution. To that end we conceived a working zone where types, typechecking and type inference can be properly addressed, and we called it the Type Theory technical space.

A deep understanding of GMM constructs is required for characterizing the identified limitations of the typing approach and for elaborating the type definitions that address such limitations. The available specification of GMM constructs consists in a high-level description of the main metaclasses included in the core metamodel and some of its extensions (in particular, the GMM4AMW extension was left unspecified). We
therefore elaborated a detailed discussion on GMM constructs, focusing on type-related aspects. Such a discussion is based on the GMM specification and complemented with inspections of AM3 source code and observations of its runtime behavior. Particularly, the GMM4CT extension defines a language for composite transformations. The extent of valid transformation compositions was critical to us, and thus our discussion specifically dealt with the expressiveness of such a language.

On the basis of the previous discussion we defined a calculus where terms within an environment represent artifacts within a megamodel. We set a restricted form of model subtyping and organized the variants of types for artifact types into a hierarchy of sorts. Type rules then specify the types of artifacts, and formally determine valid expressions. Circularity introduced by metametamodels is generally solved by an infinite hierarchy of sorts, which poses serious implementation challenges [HP91]. On the grounds of GMM specifics earlier detected, our solution to circularity was based on the Type-Type rule. We identified the transformation language of a model transformation as a vital component of its type. Since GMM is expected to be extended with new transformation languages or technologies, we provided a detailed procedure for accordingly extending the calculus.

The proof of specific properties of the calculus provides confidence on the appropriateness of the definitions made so far. Type uniqueness fails in our calculus because of the subtyping relation. The standard notion of principal type of a term deals with this issue. However, the type of a term may be derived by the application of rules which do not involve subtyping. We therefore introduce a variation of principal type, which we call direct type, that ensures type uniqueness. Type soundness states that the evaluation of a well-typed term does not cause type errors. That is one of the most important properties of a type system. We defined an operational semantics for GMM, and on top of such a semantics we proved a number of theorems from which we established type soundness. Decidability is another critical property of a type system; an algorithm should be able to determine if a given term is really typed by a given type. We provided such an algorithm and proved its correctness. Given type soundness and decidability, we can rely on the appropriateness of the type system and its ability to be implemented as a tool.

On the basis of a sound definition, the type system underwent a number of case studies. Such case studies were selected amongst reported real-world applications that would have manifested the identified limitations of GMM if carried out in the context of a GMM tool. Regardless of their complexity as model transformation scenarios, such case studies embodied challenging typing situations. We successfully managed applications that involved a HOT and DTT transformation, a composite transformation that includes a DTT and an extractor, a composite transformation that includes model weaving, a composite transformation that includes injectors and extractors, and the promotion and demotion mechanisms.

As a part of the practical contribution of our work, we have realized substantial parts of the type system which correctly handled all the case studies discussed above. As a
result we developed a toolset consisting of a stand alone version and a pluggable version of the type system. The former version is an interactive tool where interaction of the type system with a GMM tool can be simulated by a human user operating that tool. This approach focuses on early validation of the core component and enables type system evolution. The latter version builds on such a core component and, by constituting a plug-in for the AM3 tool, it is the ultimate contribution of our work. Once the AM3 project team completes the modifications in the AM3 side for properly accessing the plug-in, the benefits of our type system will be available to developers within the megamodel user community.

8.2 Contributions and Conclusions

Based on the results summarized in the previous section, in this section we discuss our contributions to GMM in specific and to MDE in general. In addition, we also address a number of conclusions we have drawn at the completion of our work.

8.2.1 Contributions

GMM includes the notion of transformation execution, and in such a context, typing is required for preventing type errors during that execution. The main contribution of our work is the improvement of the current typing approach of GMM by proposing a type system that formally indicates how to reason about types. To the best of our knowledge, our type system is the first one applied to megamodeling. Our proof of type soundness contributes to providing confidence on the appropriateness of the type system’s internal mechanisms, in the sense that they function as expected. Now, HOTs and DTTs can be properly typed and thus they can be safely handled within a megamodel. Furthermore, stronger checks can be performed on TCS projectors and AMW weaving models. The diversity of the case studies we addressed contributes to providing confidence on the fact that we have successfully overcome the identified limitations. In addition, establishing decidability for our calculus contributes to ensuring that it is implementable. In that line, our reference implementation is ready to be integrated with AM3, and it contributes to GMM in general from a practical perspective. Furthermore, our work sets a formal baseline on which to evolve the typing of future extensions to GMM.

In addition to the main contributions discussed above, our work contributes to GMM, and thus to MDE, in many other ways:

- Our work constitutes a concrete practical application of formal methods to Model-Driven Engineering. We introduced the notion of a technical space where types are formally addressed and shown a possible means to set a bridge between it and the
MDE technical space. This could enable a number of useful applications even in very different contexts.

- Actions on a megamodel in the current implementation of AM3 are performed by human users through a graphical user interface, mainly on a point-and-click basis. Our command language presented in Chapter 7 could be evolved into a practical megamodel-based programming language. In that scenario, the semantics we presented in Chapter 5 could be used as a foundation for formalizing such a language.

- Text-to-Text transformations (at least atomic ones) fall outside the scope of the MDE technical space and are not currently supported by GMM. However, they can be components of composite transformations within a megamodel (see Section 6.4 for an example). On the contrary, textual entities are GMM constructs representing non-MDE artifacts. Looking ahead for possible extensions, our type system already supports that kind of transformations and their composition to GMM-native transformations.

- GMM and AM3 are expected (and designed) to evolve with the addition of new metamodel extensions. In fact, our type system is one of such extensions. Our calculus would be of little practical help if it was not resilient to changes, or more precisely, to extensions. Together with its formal definition, we supplied a detailed procedure for guiding a possible extension process.

### 8.2.2 Conclusions

The main conclusion of this thesis is that a formal type system for GMM could be effectively defined and implemented. In more detail, the general conclusions of our work are:

- A sound type system that supports the current constructs of GMM could be formalized. Its application to a number real-world case studies validates its theoretical appropriateness. Additionally, GMM is intended to be evolved in multiple directions. The proposed extension mechanisms enable the type system to encompass GMM evolution in some of those directions.

- Some GMM scenarios involve constructs whose typing requires the application of advanced concepts, such as higher-order, polymorphism and dependent types, that could be odd to modelers. As such scenarios are not uncommon, GMM users are required to handle such complexity.

- The presented type system is decidable. This means that it can be implemented and thus it is of practical interest.
8.2 Contributions and Conclusions

• Finally, our prototypical implementation is ready for being integrated with a real-world tool such as AM3, for delivering its ultimate contribution to the practice of GMM. Our work then resulted in a sound theoretical definition that can be realized and integrated with an actual tool.

In addition to the conclusions presented above, we next discuss some other specific conclusions we have drawn and the main lessons we learned during the elaboration of our work:

• In what concerns to typing, MDE is still an immature discipline as fundamental definitions were not incorporated or agreed yet. Type systems are a common practice in programming languages, but that is not the case in modeling and transformations. For example, the ATL language completely lacks a type system. Its environment includes a model transformation which inspects the transformation under construction and produces a report with problems in it, but such an approach does not scale up. In general, a developer is forced to execute his or her transformation to find out that its definition contains errors that could have been detected by a typechecker.

• We found that transformations that accept metamodels as an input value, and when such a value occurs in the type of the result, require advanced constructs for being typed, no matter whether they are hard or straightforward to implement. Just as higher-order transformations are a special class of transformations and are the subject of specific research, we strongly believe that dependently typed transformations deserve a similar treatment. Our approach, which is based on standard dependent types, may introduce controls that could be weaker than required by practical applications. Therefore, the boundaries of how DTTs are expected to be used in practice should be more deeply investigated.

• Our type system ensures good behavior but relies on the good behavior of atomic (i.e., externally defined) transformations. The $\Delta$ function introduced in Chapter 5 abstracts away from cGMM the semantics of that kind of transformation. A type system for a transformation language could ensure that produced models will satisfy some properties [CH06], such as good behavior. Therefore, a stronger level of type safety would be achieved by integrating our type system with the type system of a transformation language. This issue is specially delicate [WF94], as both type systems need to be aligned and the result of the integration should still be sound. We strongly believe that such an integration is in general unfeasible, as alignment of disparate languages would be required. However, the ATL transformation language would be an appropriate candidate for investigating this issue.
8.3 Further Work

- Higher-order transformations can be regarded as the equivalent to higher-order functions. This is accurate in the sense that HOTs consume or produce transformations, or both. However, higher-order functions typically apply their functional arguments (see [Wik11] for a simple example). In turn, GMM exhibits a key difference. ATL transformations and composite transformations can only treat their transformation arguments as models (i.e., values). As a result, a HOT must be aware of the metamodel such an argument conforms to. If a HOT \( h \) that expects an argument \( a \) of type \( A \rightarrow B \) defined in one language, but actually receives a transformation of the same type \( A \rightarrow B \) that was defined in another language, then \( h \) will fail to read the contents of \( a \) and thus will cause an execution error. Annotating transformation types raised the amount of type rules in Chapter 4, but in exchange it allows us to avoid this kind of situations.

- The formal definition of our type system exhibits a number of benefits. First, key properties could be formulated precisely, and their corresponding proofs could be elaborated rigourously. Second, formality simplifies error detection. Third, the definition itself is an unambiguous specification for implementers (like ourselves), especially if a model-based realization such as that discussed in Section 7.3 is to be developed.

- Implementing a type system is not straightforward, especially when its definition is not yet stabilized and keeps evolving due to the feedback provided by its partial implementation. Because of the loosely-coupled architecture we chose, the interactive version of our tool was of great help during both design and validation. First, embedding a type system to an already developed tool could require a major re-engineering. Second, we believe that a premature integration with AM3 would have made such a validation process much longer and harder.

The contributions of this thesis enable a number of perspectives of future work. Some of them were already addressed in this section while discussing our contributions. In the next section we outline some important issues that can be further investigated.

8.3 Further Work

Our type system is meant to be integrated with the AM3 tool. As discussed in Chapter 7, we provided an implementation of its core component. Furthermore, such an implementation was wrapped as an AM3 extension plug-in. In that context, the main task that is to be completed for achieving the integration is the implementation of the modifications at the AM3 side which were discussed in Section 7.2.4. Such a task will be carried out by the AM3 project team as a part of their agenda. The full integration will result in
a major improvement of the AM3 tool, as it will include a dedicated type system for incorporating all the benefits discussed so far.

Other perspectives of future work can be divided in different topics. First, additional enhancements can be realized on AM3 once the type system is integrated to it. Second, issues can be examined within the cGMM calculus. Finally, other concerns apply to GMM in general. In what follows, we elaborate on each of these topics.

8.3.1 AM3 Tool

Our type system employs a combination of manifest and implicit typing. The type of a defined constant is inferred by the type system. In turn, a type annotation for each assumed constant has to be given explicitly. In cases where the constant is a transformation, specifying its type could be a burden or even complex for modelers lacking a formal background. A tool could be incorporated to AM3 for automating, or at least assisting, such a specification process. That tool would accomplish that goal by inspecting the definition of transformations, or other definitions, for extracting type information. Discovery mechanisms, such as MoDisco [BCJM10], could be applied. In particular, its discovery phase would generate model representations, if required, from implementation files. But since transformations are already represented as models, the understanding phase would perform a specific analysis for obtaining the desired information.

Composite transformations are currently designed and manually defined by end users. Type information about candidate transformations for forming the composition is key for supporting such a manual process. The type system requires that the type of a subtransformation matches the type of the input of the next subtransformation in the chain. Based on the types of the transformations recorded in the underlying megamodel, when building a composite transformation of type $A \rightarrow B$, AM3 could automatically infer (parts of) well-typed chains of compositions. Well-typing ensures that such proposed compositions are well behaved, however it does not ensure that they fit the purpose. The practical utility of inferred composition still needs to be validated based on the semantics of the individual subtransformations.

8.3.2 cGMM Calculus

Model transformations do not necessarily use all the constructs introduced by its source and target metamodels. Therefore, they may use and produce a subset of the model elements defined in the involved metamodels. For example, a transformation $t$ of type $A \rightarrow B$ could be defined for using only some of the elements defined in $A$, and producing some of the elements defined in $B$. Let $A_t$ and $B_t$ be the projections (i.e., subsets) of metamodels $A$ and $B$ according to $t$. Now let us assume a transformation $t'$ of type $B \rightarrow C$, where $B_{t'}$ and $C_{t'}$ are analogously defined. According to our typing approach,
for composing $t$ and $t'$, the fact that the target type of $t$ and the source type of $t'$ is $B$ suffices. However, [CJ09] claims that $B_t$ and $B_{t'}$ should at least overlap for the composition to be meaningful. In fact, if $B_t \cap B_{t'} = \emptyset$, then the composition will yield empty results, as the rules in $t'$ will not match any of the target elements of $t$. If this is the case and we still compose such transformations, we will get a well-typed, but useless composition. Meaningless compositions could be statically prevented if we typed transformations as $t : A_t \rightarrow B_t$ and $t' : B_{t'} \rightarrow C_{t'}$, provided that we have a means for appropriately comparing $B_t$ and $B_{t'}$. Note that $t$ is not actually a partial function. If a source model $a$ contains only model elements corresponding to constructs not defined in $A_t$, then the application $(t a)$ would still be correct. Subsetting the domain or codomain of a function is not common, at least in mathematics, but this application is worth further investigation in cGMM. Note, though, that $A_t$ is not necessarily a complete metamodel. As a result, metamodel subsets could not be used in general as the source or target types of transformations.

Composite transformations, as defined by GMM, are rather limited in their expressive power. In turn, Wires* [RRLV09] is a tool that provides a graphical executable language for the orchestration of complex ATL transformation chains including loops and conditionals. However, defined compositions are not typechecked. Our calculus could be extended for supporting such a language. On the one hand, type rules for the additional constructs are not complicated to define. On the other hand, as already discussed, changes to the type system should be associated to a reexamination of the proofs of its meta-theoretic properties. In this particular case, the semantics defined in Chapter 5 would need to be extended with the addition of new reduction rules. As a result, most of the proofs associated to soundness will have to be adjusted accordingly. An extended cGMM could support more powerful composition mechanisms in AM3, but it could be integrated with other tools as well.

The instantiation of a type parameterization introduced in Section 4.2.9, according to rule (II ELIMINATION), requires explicitly specifying the type which is being used for the instantiation (type $N$ in the mentioned rule). In most common cases, such types can be inferred, as discussed in Section 6.2. That mechanism is known in System Coq as implicit arguments, and simplifies the formulation of terms involving type instantiations. In the example of Section 6.2, the definition of $F2PText$ is meant to be expressed in a practical context as:

$$F2PText := \text{(ParaphraseATL Families2Persons)}$$

instead of explicitly providing types Families and Persons, which can be inferred since they are the source and target metamodels of the Families2Persons transformation, as in:

$$F2PText := \text{(ParaphraseATL Families Persons Families2Persons)}$$
8.3 Further Work

Using Coq’s syntax and its implicit arguments, each of them denoted by underscores, our abbreviated form would be written as:

\[ F2PText := (ParaphraseATL \_\_ Families2Persons) \]

In Section 6.6 we identified a case where instantiating types cannot be inferred (neither by Coq). For producing such a case we require an uncommon situation: an application to itself of a HOT whose result type depends on either the source or the target types of the input transformation. Recall that in cGMM a HOT which occurs as an actual parameter will not be executed, and thus it does not strictly need to have its type parameters instantiated. In CoQ this requirement does apply though, and it is the reason for the inference to fail. Finding a mechanism to overcome this issue for cGMM would prevent users from explicitly instantiating types and fully enable inference.

Transformation languages currently supported by GMM enable unidirectional transformations only. However, languages such as QVT enable a form of bidirectional transformations. Such a construct does not have a direct counterpart in cGMM. The introduction of two functions, one for each possible direction, or even an overloaded operation, could be a starting point towards supporting transformation bidirectionality.

Finally, other forms of type errors could be explored. On that basis, cGMM could be extended and still undetected situations could be further prevented.

8.3.3 GMM Approach

GMM is about managing artifacts and their relationships. Such an approach could evolve in a number of directions, which include reviewing currently supported concepts, and investigating new kinds of artifacts and relationships, between either managed or new artifacts. We also believe that a formal typing approach would also influence these future discussions. As an example of the latter dimension, module superimposition [WSD10] is a mechanism applicable to rule-based transformation languages (e.g., ATL) which is not currently supported by GMM. It allows for extending and overriding transformation rules defined in a module. In principle, to a typing extent, it is necessary for an ATL module to have the same (function) type as another module for superimposing it. It is also necessary for both modules to be ATL modules. That check can be performed from information contained in their type. As another example, metamodel merge is a relationship also not currently supported by GMM which is intensively used in the definition of the UML metamodel [OMG10]. Similarly to metamodel extension, such a relationship would not have a deep impact on the type system. However, it would be necessary to investigate if terminal models conforming to a merged metamodel exhibit some specific properties, just like subsumption applies when extensions are used.
As an example of the first dimension, subtyping when applied to metamodels deserves further discussion. AM3 supports both metamodel and metamodel extension. Tracing back such a feature to the GMM metamodel, we find that “a reference model may extend many reference models”. The problem is that AM3 does not enforce the obvious constraints: a metamodel may extend a metamodel, and even a metamodel may extend a metamodel. This fact, again, suggests that the topic is, at least, immature. A form of metamodel extension may be observed in MOF [OMG06] for the Essential MOF (EMOF) and the Complete MOF (CMOF) versions. The meaning of a sub-metamodel should be addressed from many viewpoints. Metamodels are types, but their instances are types as well. Additionally, a metamodel must conform to itself. However, it is not evident if it must conform to its super-metamodel as well. Let us assume a metamodel that introduces the typical object-oriented constructs of class and association. It could be extended with a metamodel that introduces and uses a generalization relationship. Then, the sub-metamodel conforms to itself as it can be expressed using the constructs introduced by it. However, it would not be possible to express it using the constructs of its super-metamodel only. Further discussing these kinds of issues would benefit the GMM approach as it could support a wider range of applications.
Bibliography


[Car88] Luca Cardelli. Structural Subtyping and the Notion of Power Type. In Conference Record of the Fifteenth Annual ACM Symposium on Principles of Pro-


## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Three-layered organization of basic MDE notions</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Model transformation</td>
<td>11</td>
</tr>
<tr>
<td>2.3</td>
<td>Model weaving</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>Bridging technical spaces</td>
<td>14</td>
</tr>
<tr>
<td>2.5</td>
<td>Modular structure of the metamodel of megamodels</td>
<td>17</td>
</tr>
<tr>
<td>2.6</td>
<td>Selected constructs from the GMM metamodel</td>
<td>19</td>
</tr>
<tr>
<td>2.7</td>
<td>MDE artifacts and their representation in a megamodel</td>
<td>20</td>
</tr>
<tr>
<td>2.8</td>
<td>Megamodel Navigator and Editor Page in AM3</td>
<td>22</td>
</tr>
<tr>
<td>2.9</td>
<td>Selection of an actual parameter in AM3</td>
<td>29</td>
</tr>
<tr>
<td>2.10</td>
<td>The KM32ATLCopier transformation applied to SQL produces SQLCopier</td>
<td>33</td>
</tr>
<tr>
<td>2.11</td>
<td>Bridging the MDE and Type Theory technical spaces</td>
<td>36</td>
</tr>
<tr>
<td>3.1</td>
<td>Variants of GMM relationships</td>
<td>39</td>
</tr>
<tr>
<td>3.2</td>
<td>Dual representations and transformation parameters</td>
<td>40</td>
</tr>
<tr>
<td>3.3</td>
<td>Partial metamodel of GMM4ATL extension</td>
<td>41</td>
</tr>
<tr>
<td>3.4</td>
<td>Partial metamodel of GMM4TCS extension</td>
<td>43</td>
</tr>
<tr>
<td>3.5</td>
<td>Partial metamodel of GMM4CT extension</td>
<td>44</td>
</tr>
<tr>
<td>3.6</td>
<td>Partial metamodel of GMM4AMW extension</td>
<td>46</td>
</tr>
<tr>
<td>4.1</td>
<td>Hierarchy of cGMM sorts</td>
<td>56</td>
</tr>
<tr>
<td>6.1</td>
<td>Artifacts involved in the generation of SQLCopier transformation</td>
<td>92</td>
</tr>
<tr>
<td>6.2</td>
<td>Derivation of judgment $\Gamma \vdash (KM32ATLCopier ; SQL) : SQL^{\text{ATL}}$</td>
<td>94</td>
</tr>
<tr>
<td>6.3</td>
<td>Derivation of judgment $\Gamma' \vdash (SQLCopier ; s1) : SQL$</td>
<td>95</td>
</tr>
<tr>
<td>6.4</td>
<td>Artifacts involved in the paraphrasing of the KM3 metamodel</td>
<td>96</td>
</tr>
<tr>
<td>6.5</td>
<td>Artifacts involved in the model adaptation problem</td>
<td>100</td>
</tr>
<tr>
<td>6.6</td>
<td>Artifacts involved in the automatic layout problem</td>
<td>103</td>
</tr>
<tr>
<td>6.7</td>
<td>Promotion and demotion of a KM3 metamodel in EMF</td>
<td>105</td>
</tr>
<tr>
<td>7.1</td>
<td>Mapping AM3 elements to cGMM terms</td>
<td>110</td>
</tr>
<tr>
<td>7.2</td>
<td>Use case diagram showing the basic functionality of the type system</td>
<td>111</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

7.3 Logical view of the type system’s architecture . . . . . . . . . . . . . . . . . . . . 114
7.4 Classifiers realizing the behavior of the TypeSystem component . . . . . . . . 116
7.5 Implementation view of the interactive version of the cGMM tool . . . . . . . 118
7.6 KM32ATLCopier case study in the interactive version of the cGMM tool . . . . 119
7.7 Plug-in architecture of AM3 including the extension plug-in . . . . . . . . . . 120
7.8 Implementation view of the plug-in version of the cGMM tool . . . . . . . . . 121
7.9 Megamodels for the KM32ATLCopier transformation case study . . . . . . . 123
Index

<,  53, 101
Δ-typability, 76
Γ, see environment
Λ, 67, 97
Π, 59, 97
Πα, 61
ΠATL, 62, 92, 97
ΠCT, 63, 97, 104
⇒,  77
⇒1,  79
⇒*,  78
⇒n,  80
λCT, 63, 97, 104
() ( ),  57, 104
DΓ, 75
TΓ, 74
L, 82
S, 55
ATL, see ΠATL
CT, see ΠCT
EXE, 65, 103
TCS, 64, 97, 103
πi, 57
≤,  74
×,  57, 101, 103
:c2,  27
::,  67, 101
AMWWeaving, 66, 101
Text, 59, 64, 65, 97
[M], 26, 76, 78
AM3Core, 17
GMM4AMW, 17, 45
GMM4ASM, 17
GMM4ATL, 17, 40
GMM4CT, 17, 43
GMM4TCS, 17, 42
GlobalModelManagement, 17
TypeSystem4GMM, 119
Type, 54
c2,  10
AM3, 21, 108
plug-ins, 21, 119
AMW, 12
ATL, 10
AtlanMod
  MegaModel Management, see AM3
  Model Weaver, see AMW
  Transformation Language, see ATL
calculus, 24
eGMM, 48
c2,  10
Γ-term, 52
Γ-type, 52
environment, 50
assumption, 51
definition, 51
subtyping assertion, 51
validity, 88
extending, 69
free variables, 50
judgements, 51
realization, 113
soundness, 79
INDEX

substitution, 50
terms, 49
use cases, 111
well-typing, 52
conformance relation, 9
derivation, 51, 93
DTT, 10, 30, 32, 91, 101

formal system, see calculus
Global Model Management, see GMM
GMM, 1, 6, 15
metamodel, 16
AMW extension, see GMM4AMW
ATL extension, see GMM4ATL
composite transformations extension, see GMM4CT
core, see AM3Core
global model management extension, see GlobalModelManagement
TCS extension, see GMM4TCS
tool, see AM3
typing, 27

HOT, 10, 30, 32, 91, 101

ill-typed, 24

MDE, 1, 7
megamodel, 1, 16, 18
Megamodelling, 15
model, 7
metametamodel, 9
metamodel, 8, 9
terminal model, 9
model transformation, 9
dependently typed, see DTT
higher-order, see HOT
transformation model duality, 20
model weaving, 11, 31, 101

weaving model duality, 20
Model-Driven Engineering, see MDE
normal form, 50
pCic, 3, 37, 49, 90
Predicative Calculus of Inductive Constructions, see pCic

redex
\( \beta \), 77
\( \delta \), 77
\( \sigma \), 77

sort, 49
subsumption, 53

TCS, 13, 96, 102
technical space, 13
bridging, 13
projector, 14
extractor, 14
injector, 14
term
faulty, 84
in normal form, 78
stuck, 78

Textual Concrete Syntax, see TCS
textual entity, 13, 18, 31
type, 23
derror, 23
type inference, 25, 86
algorithm, 87
type system, 24
decidability, 26
pluggable, 108
soundness, 26
typechecker, 24
typechecking, 24, 86

well-typed, 24
This report has been written with a trial version of WinEdt5 (v5.4), TeXLive4a for Win32, on Microsoft® Windows Vista.

Version 1.0 compiled 18th November 2011 at 2:01.

All rights reserved.