Softw Syst Model (2013) 12:105–119
DOI 10.1007/s10270-011-0191-2

SPECIAL SECTION PAPER

Typing artifacts in megamodeling

Andrés Vignaga · Frédéric Jouault · María Cecilia Bastarrica · Hugo Brunelière

Received: 30 November 2009 / Revised: 14 November 2010 / Accepted: 16 January 2011 / Published online: 16 February 2011

Abstract Model management is essential for coping with the complexity introduced by the increasing number and varied nature of artifacts involved in model-driven engineering-based projects. Global model management (GMM) addresses this issue by enabling the representation of artifacts, particularly transformation composition and execution, within a model called a megamodel. Type information about artifacts can be used for preventing type errors during execution. Built on our previous work, in this paper we present the core elements of a type system for GMM that improves its original typing approach and enables both typechecking and type inference on artifacts within a megamodel. This type system is able to deal with non-trivial situations such as the use of higher order transformations. We also present a prototypical implementation of such a type system.

Keywords Model transformation · Type system · Megamodeling

Communicated by Jeff Gray and Richard Paige.

A. Vignaga (✉) · M. C. Bastarrica
MaTE, Department of Computer Science,
Universidad de Chile, Santiago, Chile
e-mail: avignaga@dcc.uchile.cl

M. C. Bastarrica
e-mail: cecilia@dcc.uchile.cl

F. Jouault · H. Brunelière
AtlanMod, INRIA Rennes Center, Bretagne Atlantique,
Ecole des Mines de Nantes, Nantes, France
e-mail: frederic.jouault@inria.fr

H. Brunelière
e-mail: hugo.bruneliere@inria.fr

1 Introduction

In the field of software development, the increasing use of model-driven engineering (MDE) in recent years has lead to more and more complex situations. Indeed, MDE mainly suggests basing the software development and maintenance processes on chains of model transformations. A single transformation is often quite easy to handle, but as soon as industrial use cases are tackled, we are faced with large sets of MDE artifacts (e.g., models, metamodels, transformations) from which a solution has to be assembled. Thus, in order to be able to use them, but keeping the complexity of MDE under control, we need to count on 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) [5].

As the managed modeling resources may be of different nature, some support for efficiently organizing them is required. In order to cope with this heterogeneity, a GMM solution has to rely on an architecture which allows precisely typing all the involved entities and corresponding relationships. This should prevent 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.

Currently, the GMM approach 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. Moreover, 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). Information based on this typing approach suffices for most common cases. However, this scheme notably fails when transformations explicitly depend on these semantic links like in the two following cases: (a) when a metamodel is used as input to a transformation (i.e., a type used as a value), and (b) when a transformation is used as input to another transformation (i.e., a function used as a value). Under these circumstances, it may not be possible to automatically infer a complete type for some elements and errors may be inadvertently introduced. For this reason, a more complex typing approach is required.

In [27], we introduced **cGMM**, a predicative dependently typed calculus addressing a static type system [7] dedicated to the core constructs of GMM and its main extensions. We also extended **cGMM** in [25] and [26] for addressing the rest of the GMM extensions. In this paper we discuss an upgraded version of the core **cGMM** with major improvements for dealing with the identified limitations of the original GMM typing approach. Expressing GMM elements as terms of our calculus enables to statically typecheck these elements in a mechanical fashion. The calculus was implemented as a stand-alone prototype which is accessed through a command line for testing and validation purposes. Such a prototype is intended to be evolved and integrated to AM3 [1], a tool realizing GMM.

This paper is organized as follows: Section 2 describes the GMM approach to model management, characterizes the limitations of its original typing approach, and introduces an example illustrating them. Section 3 details our formal system by providing the syntax of terms and types, type judgments, as well as the set of type rules that form the type system. A proof of type soundness is also provided. Section 4 revisits the example in order to demonstrate the application of the type system for solving it. Section 5 discusses its prototypical implementation and its integration to the AM3 tool. Section 6 discusses related work. Section 7 concludes and states future work.

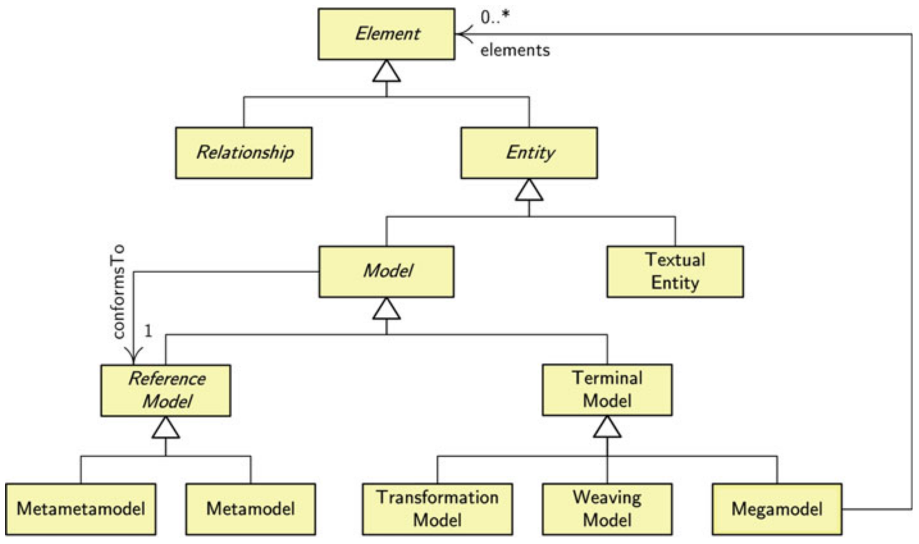
2 Global model management

In this section we summarize the basic concepts of GMM that enable an understanding of the general context. We also discuss how typing is currently addressed and its limitations. For illustrating these issues we discuss an example, which will be revisited after our solution is presented.

2.1 Global model management conceptual framework

The global model management approach is based on several basic concepts (see Fig. 1) which can be used for representing any concrete case within MDE-based projects. Most of these concepts, corresponding to a generic conceptual MDE framework, have already been presented in [11]. In addition, the concept of a *megamodel* is introduced as a building block for *modeling in the large* [5]. The principle is the following: for each real-world complex system or process, there can be a *megamodel* [3] representing the different artifacts involved (e.g., models), along with their relationships by specifying associated metadata. The type of an artifact, relationships among artifacts, the identifier of a given artifact and its location, etc., are examples of such registered metadata.

Figure 1 shows some selected basic constructs of GMM and their relations. A megamodel is a collection of manage-



context Metametamodel inv: self.conformsTo = self
context Metamodel inv: self.conformsTo.oclIsKindOf(Metametamodel)
context TerminalModel inv: self.conformsTo.oclIsKindOf(Metamodel)

Fig. 1 GMM conceptual framework

able elements, and Element is their root class. An element may be either an Entity (i.e., an artifact) or a Relationship among entities. MDE approaches generally introduce the following three different kinds of models, which occur in the conceptual framework of GMM:

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

Several kinds of *terminal models* may be considered, for example, *weaving models*, and *transformation models*. A *megamodel* is also a specific kind of terminal model. As it is a terminal model, a megamodel conforms to a specific metamodel: the *metamodel of megamodels* [16]. If represented as models, available tools, services and service parameters may also be managed by a megamodel. 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.

In addition, the current GMM framework proposes different kinds of relationships between models (not shown in Fig. 1). The *model transformation* relationship allows specifying source and target reference models of a given transformation model, and can thus be regarded as its signature. From an execution point of view, the *transformation record* relationship offers a way of representing the metadata needed for any potential execution of a given transformation. This allows specifying its actual input and output models.

In summary, a megamodel can be viewed as a metadata repository where precise representations of models and links between them are stored and made available to users for varied purposes. In particular, the framework should be able to represent type information for adequately typing each element in a megamodel, and provide precise directions on how to use that information.

2.2 Limitations of the current typing approach

As mentioned earlier, the current solution to typing in GMM follows a simple approach: in principle, all entities are models. Each model conforms to a concrete reference model, which is its type [16]. Such a has-type relation (denoted by '$:_{c2}$') is therefore defined as follows:

$$conformsTo(m, M) \Leftrightarrow m :_{c2} M,$$

(for any model m and reference model M)

However, GMM involves other elements different from entities: relationships. Some elements have dual representations; for example, a transformation may be regarded as a model (i.e., transformation model) but also as a relationship (i.e., model transformation) [4]. When regarded as a model, the type of a transformation is the metamodel it conforms to. For ATL (AtlanMod Transformation Language) transformations [10,12], this type is plainly *ATL*, which does not carry information about source and target types. When a transformation is regarded as a relationship, such a relationship is actually unidirectional and thus the transformation is understood as a function on models. Metadata associated with a transformation refers to the type of source and target models. However, such models are typed as models irrespective of whether they are transformations or not. In conclusion, typing in GMM is not actually based on the $:_{c2}$ typing relation only. It is also (implicitly) based on metadata as well. For first-order transformations, such a typing approach suffices. In what follows, however, we show that in some specific cases sensitive type information is lost.

Consider a higher order transformation (HOT) T which produces another transformation t. In the current typing approach, T is considered as a function, but t is considered as a value. As a consequence of this situation they are typed differently. Metadata of T refers to the types of its source and target elements. In particular, the type of the target element is the type of t, which is the metamodel t conforms to (e.g., *ATL*). The type of t as a function does not fit into this scheme and thus t is only partially typed. We do know that it is a transformation, but we do not know the types of its source and target elements.

This typing approach presents an interesting benefit though. Some form of genericity is introduced: a HOT taking an ATL transformation as source accepts any model conforming to *ATL* (i.e., *any* ATL transformation), regardless of the number and type of its source and target elements. This capability enables HOTs and is something we would like to preserve.

Another situation where a $:_{c2}$-based approach may lead to a loss of type information is when transformations operate on metamodels, or more precisely, on reference models. In fact, the type of any reference model received or generated by such a transformation is a metametamodel (e.g., *KM3* [11]). Then, from the type of the transformation, it is possible to know that a reference model is involved, but not which one. If such a reference model occurs in the target type, then it is not possible to correctly type that transformation with the current approach.

When the two situations described so far happen simultaneously, even harder problems arise. The *KM32ATLCopier* transformation [2] is a simple transformation which introduces a complex typing problem and suffices for illustrating both issues. In what follows, we discuss its typing according to the current approach and its limitations. Later on, we show that properly typing this transformation will require advanced type theory concepts. *KM32ATLCopier* is a HOT which receives a reference model M and produces an identity transformation (called a copier), which is specifically applicable to models conforming to M. The type of the resulting copier transformation clearly depends on M. The type of *KM32ATLCopier* as an entity is just the metamodel that entity conforms to

$$\mathit{KM32ATLCopier} : ATL \qquad (\mathrm{T}_e)$$

In turn, the type of *KM32ATLCopier* as a relationship, which is even richer in type information, may be extracted from the header of its ATL definition:

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

Using the usual notation for function types, this type can be expressed as

$$\mathit{KM32ATLCopier} : KM3 \rightarrow ATL \qquad (\mathrm{T}_r)$$

This type expresses that *KM32ATLCopier*: (i) is a transformation, (ii) accepts a KM3 metamodel as an argument, and (iii) it produces an ATL transformation. However, although this information is correct, the type is insufficient. First, M is not present as the argument of the transformation. Second, all we know about the result from its type is that it is a transformation. Third, it is not possible to specify that we know that both of result's source and target models are of type M.

By introducing other kinds of types, such as function types and parametric types, we will be able to deal with these issues. In Sect. 4 this example will be revisited and a type for *KM32ATLCopier* carrying richer information will be discussed.

3 A type system for GMM

Our solution is based on a calculus called **cGMM**. By defining a mapping between GMM constructs and **cGMM** typed terms, we are able to express elements within a megamodel as terms, and to statically typecheck them in a mechanical fashion.

A type system is intended for preventing type errors during the execution of a program. In GMM, the application of transformations may cause type errors. For us, type errors can be (i) the attempted execution of a non-transformation and (ii) the use of a transformation on arguments for which it was not defined. Note that an application a, of the form $(t\ x)$, relies on other elements contained in a megamodel $\mathfrak{M}$ (i.e., transformation t and model x). In our approach, we use a judgment $\Gamma \vdash \mathcal{A}$, where environment Γ represents a megamodel $\mathfrak{M}$, and assertion $\mathcal{A}$ is a type assertion of the form p:T. Term p represents application a and the assertion assigns type T to term p. If the judgment is valid, that is, it can be proved that p has type T in the context of Γ, then application a will not produce a type error when executed in $\mathfrak{M}$. Otherwise, the application will cause a type error and thus it should not be executed.

The **cGMM** calculus is a predicative dependently typed calculus, similar to the underlying language of Coq [23], the Predicative Calculus of (Co)Inductive Constructions (pCIC) [18,28]. Dependent products enable (dependent) function types for typing transformations, as well as parametric types for coping with genericity. For example, the *type* of the transformation produced by *KM32ATLCopier* is dependent on the *value* of the input parameter to *KM32ATLCopier* (i.e., reference model M mentioned in the example of the previous section). Higher-order functions naturally represent higher-order transformations. In addition, an infinite hierarchy of universes supports the notion of Type being a type (i.e., Type:Type), and enables a proper representation of the three levels of models (M1, M2 and M3) mentioned in Sect. 2.1.

In order to formalize the type system, we need to present some elements of our calculus first. We start by presenting its syntax and the mapping of **cGMM** terms to GMM constructs. We then address the typing of terms.

3.1 Textual syntax

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.

3.1.1 Sorts

Types are seen as terms and as such they should be typed. The type of a type is called a *sort*. In principle, we use types for typing models so we introduce the sort Type which intends to be the type of such types. Since sorts can be manipulated as terms they also should be given a type. Typing Type with itself leads to undecidable type systems [6], as the type-checking process may diverge. Therefore, we need to introduce many infinite sorts by means of a hierarchy of sorts Type_i for any natural i. Thus, our set of sorts $\mathcal{S}$ is defined by $\mathcal{S} \equiv \{\mathsf{Type}_i \mid i \in \mathbb{N}\}$. This provides predicativity to the calculus, as quantifying on a type of one level yields a type in a level above in the hierarchy (circularity in type formation is forbidden). Sorts in $\mathcal{S}$ satisfy the following property: Type_i:Type_{i+1}. In this way, we understand Type_0 as the type of all metamodels (e.g., *Class* : Type_0), which turns Type_0 into a metametamodel.

As in Coq, when referring to sort Type_i the user will never mention the index i explicitly, which is managed by the system. Therefore, from a user perspective Type:Type is safely assumed. Consequently, without indices, Type is a metametamodel which conforms to itself, as required by the first invariant in Fig. 1.

GMM is expected to support multiple metametamodels at the same time, for example KM3, ECore, and so on. In principle, providing support for one single metametamodel is not an issue since *promotion* and *demotion* mechanisms may by applied. If ECore were the only metametamodel, the KM3 metametamodel may be represented as a metamodel conforming to ECore (i.e. $KM3 :_{c2} ECore$). Then, a KM3 metamodel M is represented as a terminal model conforming to the metamodel version of KM3 (i.e., $M :_{c2} KM3$). Whenever needed, M (which is actually a terminal model) may be transformed, by a proper transformation, to an actual ECore-conforming metamodel M' (i.e., $M' :_{c2} ECore$, thus making M' an actual metamodel). This mechanism is called promotion. Demotion is the inverse operation and it is analogously defined. In fact, this is how multiple metametamodels are currently handled in AM3 [1], a concrete realization of GMM. In what follows, we refer to Type as an arbitrary metametamodel.

3.1.2 Terms

Terms are built from constants declared in the environment, dependent products, abstractions, applications, cartesian products, tuples, and projections. Assuming that T and U are terms, **cGMM** terms are as follows:

Type	A sort, the type of all types
c	A constant declared in the environment
x	A variable
$\lambda x{:}T.U$	An abstraction
$\Pi x{:}T.U$	A dependent product
$(T\ U)$	An application
$U_1 \times U_2$	A cartesian product
$\langle T_1, T_2 \rangle$	A tuple
$\pi_i(T)$	A tuple projection ($i \in \{1,2\}$)

Type is a metametamodel, and as such it belongs to M3. Constants in the environment map to models, either at M1, M2, or M3. If a constant, other than Type, is typed by Type (more precisely, by Type_0), it represents a metamodel, which is an element of M2. If it is typed by a term typed by Type, then it denotes a terminal model, which is an element of M1. For example, declaration $Class{:}\mathsf{Type}$ means that constant *Class* is assumed as a metamodel, and declaration $c{:}Class$ means that constant c is assumed as a terminal model of type *Class*.

Transformation models are typed by functional products. A functional product $\Pi x{:}T.U$ denotes a function mapping a value x of type T to a value of type $U(x)$. The type of the result depends on argument x, with U specifying the dependence [14]. The degenerate case where U is a constant function (i.e., U does not actually depend on x) corresponds to classic function types. For emphasizing their functional nature, in the dependent case a functional product is denoted by $x{:}T \to U$, while in the non-dependent case a functional product is denoted by $T \to U$. For example, declaration $Class2Relational{:}Class \to Relational$ means that constant *Class2Relational* is assumed as a transformation of type $Class \to Relational$. Dependent products are used for representing parametric types as well. For emphasizing their non-functional nature, parametric types are denoted as $\forall x{:}T.U$ (i.e., type U is parameterized by x of type T).

GMM manages two kinds of transformations. On the one hand, transformation models can be externally defined in a suitable transformation language, such as ATL. We call this kind of transformations *atomic* transformations, and they are seen as black-box operations on models where their internal definition is not accessible by the GMM environment. Currently, the only external transformation language supported by GMM is ATL through the GMM4ATL extension. In this work we assume that all atomic transformations are defined in ATL. On the other hand, transformations can be defined within a megamodel, using the language provided by the GMM4CT extension, as compositions of other existing transformations, regardless of their kind. We call them *composite* transformations and they are model transformations (i.e., relationships) and not transformation models (i.e., entities).

Atomic transformations are *assumed* in the environment. For example, declaration $c : T$ represents a transformation called c of type T, where T is a functional (either dependent or non-dependent) product. We say that c is assumed because its definition is not explicitly provided. In turn, composite transformations are *defined* in the environment. For example, declaration $c := t : T$ represents a transformation called c of type T and defined by term t. Term t is a λ-abstraction and its body can only be formed by compositions of applications, tuple constructions, and projections. Another use of definitions within an environment are applications. In this case term t is an application, and term T is the type of the resulting element.

Finally, transformations accept only one argument and produce only one result. A cartesian product, which is a non-dependent form of Σ-types [13], enables a functional product type with multiple sources and multiple targets. Tuples are pairs, and projections extract a component out of a tuple. For example, $\pi_1(\langle T_1, T_2 \rangle)$ yields T_1, and $\pi_2(\langle T_1, T_2 \rangle)$ yields T_2. Generalized cartesian products and tuples may be achieved by iterating our binary cartesian products and tuples. As a remark, a HOT is a transformation that operates and/or produces other transformations. Thus, a HOT is expressed as a function which either has a parameter typed by a function type, or returns an element typed by a function type. Free variables and substitution are defined as usual. Substituting a term T to free occurrences of a variable x in a term U is denoted as $U\{x/T\}$.

Example For illustrating some of the ideas introduced above, we define a composite transformation which is composed of two atomic transformations. We first assume both atomic transformations:

$$Class2Relational : Class \to Relational$$

$$Relational2SQL : Relational \to SQL$$

For such assumptions to be well formed, constants *Class*, *Relational*, and *SQL* should have been previously assumed in the environment. Then, as constants *Class2Relational* and *Relational2SQL* are in the environment, it is possible to define the composite transformation:

$$Class2SQL := \lambda x{:}Class.(Relational2SQL\ (Class2Relational\ x)) : Class{\to}SQL$$

The term playing the role of *t* in the definition of constant *Class2SQL* is a λ-abstraction *a*. The body of *a* is the proper composition of *Class2Relational* and *Relational2SQL*. If an element within a megamodel should correspond to *Class2SQL*, then its definition in the megamodel should correspond to *a*. In addition, assuming $c : Class$ in the environment (i.e., in the megamodel there is a terminal model *c* conforming to metamodel *Class*), it is possible to define the following application:

$$s := (Class2SQL\ c) : SQL$$

Here, the term playing the role of *t* in the definition of constant *s* is an application *p*. In the megamodel, there is a transformation record corresponding to *p* (i.e., the application of *Class2SQL* to *c*), and the terminal model which is the target element in that transformation record corresponds to *s*. Finally, the type of *Class2SQL* is the type of *a* (i.e., $Class \to SQL$), and the type of *s* is the type of *p* (i.e., *SQL*). Both types may be explicitly provided by the user. However, they can be inferred from terms *a* and *p*, respectively. This is what type inference is about. In assumptions, types must be provided explicitly. In definitions, type *T* is inferred from term *t*. Our prototype exhibits this same behavior.

3.2 Typing

A type system is a collection of type rules; however, they are always formulated with respect to a static typing environment for the program fragment being checked. A *static typing environment* records the type of free variables during the processing of program fragments. For example, the has-type relation *a*:*A* is expressed in the context of a static typing environment Γ that contains information about free variables of *a* and *A*.

3.2.1 Judgments

The description of a type system starts with the description of a collection of judgments of the form $\Gamma \vdash \mathcal{A}$ where Γ is a static typing environment, $\mathcal{A}$ is an assertion, and the free variables of $\mathcal{A}$ are declared in Γ. The static typing environment can be understood as a list of declarations (either assumptions or definitions) of distinct constants. A static typing environment then maps to the notion of megamodel. The empty environment is denoted by ∅. The form of $\mathcal{A}$ determines the different judgments to be used within a type system. For our system, we need the following judgments:

$\Gamma \vdash \diamond$	Γ is a well-formed environment
$\Gamma \vdash T : U$	*T* is a well-formed term of type *U* in Γ

A judgment can be regarded as *valid* or *invalid*. Validity formalizes the notion of well-typed programs and is based on type rules. Type rules are used to carry out step-by-step deductions, i.e., type derivations, which formally prove that judgments are valid.

3.2.2 Type rules

Figure 2 shows some selected rules of **cGMM**. They are similar to the type rules for pCIC in [23]. Note that we do not support logical propositions and their proofs (i.e., there is no such type Prop). This means that **cGMM** is not a logic system where propositions are proved by rule derivations. Rather, **cGMM** is only a type system where rules enable the derivation of typing judgments. Indices *i*, *j* and *k* are arbitrary natural numbers, and constructor Π forms either parametric types (∀) or function types (→) as discussed earlier.

Rule (Env ∅) is an axiom stating that an empty environment is a valid environment. This means that an empty megamodel is a valid megamodel.

Rules (Env Assum) and (Env Def) extend an environment with a declaration of a constant, provided that the constant is not already declared and its type is a valid type. This corresponds to adding a new element to a megamodel. In turn, rules (Assum) and (Def) enable the extraction from the environment of type information about declared constants. Rule (Ax) formalizes the property stated in 3.1.1 which holds for universes within $\mathcal{S}$.

Rule (Prod) constructs dependent products which correspond to transformation types and parameterized types. In turn rule (Abs) constructs abstractions which correspond to composite transformation definitions and type abstractions. Finally, rule (App) types applications. When the type of *t* is a type parameterization, the application is a type instantiation. Otherwise, it is a functional application. Note that type substitution in *T* occurs only when the type of *t* is dependent.

$$\text{(Env }\varnothing\text{)}\ \frac{}{\varnothing \vdash \diamond} \qquad \text{(Env Assum)}\ \frac{\Gamma \vdash T{:}s \quad s \in \mathcal{S} \quad c \notin \Gamma}{\Gamma, c{:}T \vdash \diamond} \qquad \text{(Env Def)}\ \frac{\Gamma \vdash T{:}s \quad s \in \mathcal{S} \quad c \notin \Gamma}{\Gamma, c{:=}t{:}T \vdash \diamond}$$

$$\text{(Assum)}\ \frac{\Gamma', c{:}T, \Gamma'' \vdash \diamond}{\Gamma', c{:}T, \Gamma'' \vdash c : T} \qquad \text{(Def)}\ \frac{\Gamma', c{:=}t{:}T, \Gamma'' \vdash \diamond}{\Gamma', c{:=}t{:}T, \Gamma'' \vdash c : T} \qquad \text{(Ax)}\ \frac{\Gamma \vdash \diamond \quad i < j}{\Gamma \vdash \mathsf{Type}_i : \mathsf{Type}_j}$$

$$\text{(Prod)}\ \frac{\Gamma \vdash T{:}\mathsf{Type}_i \quad i \le k \quad \Gamma, x{:}T \vdash U : \mathsf{Type}_j \quad j \le k}{\Gamma \vdash \Pi x{:}T.U : \mathsf{Type}_k} \qquad \text{(Abs)}\ \frac{\Gamma \vdash \Pi x{:}T.U : \mathsf{Type}_i \quad \Gamma, x{:}T \vdash t{:}U}{\Gamma \vdash \lambda x{:}T.t : \Pi x{:}T.U}$$

$$\text{(App)}\ \frac{\Gamma \vdash t : \Pi x{:}U.T \quad \Gamma \vdash u{:}U}{\Gamma \vdash (t\ u) : T\{x/u\}}$$

Fig. 2 Sample type rules of **cGMM**

In the next section, we revisit the example of Sect. 2 in detail and present a type derivation which involves the application of many of the rules discussed earlier.

3.2.3 Meta-theoretic properties of the calculus

The purpose of a type system is to prevent programs from causing type errors during their execution. A type system is *sound* when only well-typed programs execute without type errors [7]. This property of a type system is demonstrated by means of a soundness theorem. A proof of soundness rests upon the semantics of the underlying language, and other properties such as subject reduction, confluence, and strong normalization. In what follows, we discuss both components of type soundness.

Semantics. The semantics of **cGMM** describes how computation takes place, where the meaning of terms is specified by a transition system. Such a transition system is based on a transition relation ($\triangleright$) that describes how individual steps of computation are performed. In turn, the transition relation is defined by reduction rules. In **cGMM** three different reductions are supported: β, δ, and σ. The first reduction refers to the functional application of composite transformations in the classic way: substitutes the actual parameter to all occurrences of the formal parameter in the body of the function. The reduction rule is as follows [23]:

$$((\lambda x{:}T.U)\ u) \triangleright_\beta U\{x/u\}$$

In turn, the second reduction expands a defined constant into its definition. The reduction rule is as follows [23]:

$$x \triangleright_\delta t \quad (\text{if } x := t : T \text{ is in the environment})$$

Finally, the third reduction refers to tuple projection. The reduction rule is as follows [13]:

$$\pi_i(\langle T_1, T_2\rangle) \triangleright_\sigma T_i \quad (i \in \{1, 2\})$$

Note that β-reduction is not intended for applications where the applied function is an atomic transformation because its definition is not provided. For such applications we assume a Δ function [30] that abstracts away the precise set of atomic transformations. This introduces an additional reduction rule:

$$(t\ u) \triangleright_\Delta \Delta(t, u) \quad (\text{if } t : \Pi x{:}T.U \text{ is in the environment})$$

In the above rule, t is an atomic transformation and the value $\Delta(t, u)$, of type $U\{x/u\}$, is what an actual execution environment would produce. Additionally, for such a value to be defined (i.e., not being $\perp$), argument u must be of type T [30]. The introduction of this last rule implies that the results of computations are always expressed in terms of applications of the Δ function. For example, using the declarations of the example of Sect. 3.1.2 we have the following reductions of constant s, where the last term cannot be further reduced:

$$\begin{array}{l} s \\ \quad \triangleright_\delta \\ (Class2SQL\ c) \\ \quad \triangleright_\delta \\ (\lambda x{:}Class.(Relational2SQL\ (Class2Relational\ x))\ c) \\ \quad \triangleright_\beta \\ (Relational2SQL\ (Class2Relational\ c)) \\ \quad \triangleright_\Delta \\ (Relational2SQL\ \Delta(Class2Relational, c)) \\ \quad \triangleright_\Delta \\ \Delta(Relational2SQL, \Delta(Class2Relational, c)) \end{array}$$

Properties. We now review the properties that enable a proof of type soundness. Subject reduction states that reductions preserve types (i.e., if T of type U reduces, then it does so to a value also of type U). Confluence (i.e., the Church–Rosser theorem) states that different reductions starting from a given term will eventually meet in the same term. In some systems, type conversion is additionally required. Type convertibility $T = U$ is achieved when terms T and U reduce to the same normal form. This enables a rule which says that two convertible well-formed types have the same inhabitants. In this way, terms of a type before a reduction are also typed by the type resulting from the reduction. Note that all these properties are not sufficient for type soundness because they do not rule out the case in which T has a type but it does not reduce (i.e., term T diverges). Systems where reductions for all typable terms do terminate are called strongly normalizing.

Our calculus is a subset of pCIC, which enjoys all the properties discussed above [23]. Furthermore, it exhibits a number of restrictions derived from how GMM is defined:

(a) In a definition $c := t : T$, term t may only contain applications of functions which were declared in the environment before c. This excludes constant c itself and, as a result, **cGMM** does not support any form of recursion (fixpoint operators are not supported).
(b) In an application $(T\ U)$, term T can only be a constant which was declared in the environment before such an application is ever processed. This means that T does not need to be reduced for inferring the type of the application.
(c) Type equivalence is not structural but rather by-name. In an assumption $c : T$, term T cannot be or include an application. For example, provided that transformation T' returns a metamodel, assumption $c : (T'\ U)$ is not acceptable. Alternatively, $t := (T'\ U)$ should be defined first, and only then $c : t$ can be assumed. This means that, in the original assumption, term T does not need to be reduced and type equivalence is directly determined by name equality.

Following the approach introduced in [30], a proof of type soundness can be structured as follows: On the one hand, we shall prove that well-typed terms yield 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 (i.e., non-normalized terms which cannot be further reduced) or diverge. To that end we proceed as follows, where $\triangleright^*$ denotes the reflexive and transitive closure of $\triangleright$:

1. If term M reduces to a value V, then such a value is unique. This property is implied by the Church–Rosser theorem for $\triangleright^*$, and proves that normal forms are unique.
2. Well-typed terms do not diverge. This is equivalent to proving strong normalization, and together with the above property, it proves that well-typed terms that do not stuck have a unique normal form.
3. Stuck terms are untypable. This proves that well-typed terms do not stuck. Together with the aforementioned properties it proves that a well-typed term yields a unique term in normal form, and that an ill-typed term either stucks or diverges. This is weak type soundness [30] and means that well-typed terms will not go wrong.
4. Transition relation $\triangleright^*$ preserves types. This is subject reduction, and with all the above, this proves strong type soundness (i.e, additionally to weak soundness, the reached term is of the right type).

In what follows we sketch a proof for each of the properties discussed earlier.

Theorem 3.1 (Church–Rosser theorem) *If $M_1 = M_2$, then there exists M such that $M_1 \triangleright^* M$ and $M_2 \triangleright^* M$.*

Proof Sketch A proof of the Church–Rosser theorem is analogous to the proof in [13] for the extended calculus of constructions (ECC), which is an ancestor of pCIC. The details omitted from such a proof can be found in [19]. □

As a corollary of the Church–Rosser theorem, we have that the normal form of a term is unique, if it exists. Next we show that well-typed terms do have a normal form.

Theorem 3.2 (strong normalization) *If $\Gamma \vdash M : A$ then M is strongly normalizable.*

Proof Sketch A proof of strong normalization is simplified by the restrictions discussed earlier. We show that it is not possible to find an infinite sequence of reductions starting from a well-typed term. Since the set of transition rules is finite, an infinite transition sequence necessarily involves an infinite application of at least one of the rules. As discussed before, an environment is a finite sequence of declarations. Infinite applications of δ require an infinite environment, or the reintroduction of an already unfolded constant. Based on (a), this latter scenario is not possible. In turn, β may only be applied when the definition of a composite transformation was already unfolded. Then the amount of applications of β in every transition sequence is less or equal to the amount of applications of δ. Based on (b), applied functions are constants. In addition, further applications of Δ involve applications to be nested within the argument. As a result, infinite applications of Δ either require an infinite environment or infinite terms. Since every term is finite, this is not possible. Finally, an application of σ yields a component of the original term. Then infinite applications of σ require infinite environments, infinite terms, or infinite applications of the other rules for enlarging the resulting term. By the above arguments, this latter scenario is not possible. □

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. As in [30], we approximate the notion of a stuck term with the notion of faulty terms. A faulty term is a term that contains a subterm of the form $(t\ u)$, where t is an atomic transformation and $\Delta(t, u)$ is undefined. Note that all terms that yield a stuck term are faulty, although the opposite is not necessarily true.

Theorem 3.3 (faulty terms are untypable) *If M is faulty, then there are no environment Γ and type A such that $\Gamma \vdash M : A$.*

Proof Sketch For showing that faulty terms are untypable it suffices to show that subterms of term M that cause M to be faulty are untypable. Let us assume that $(t\ u)$ is faulty, where t is an atomic transformation such that $t : \Pi x{:}T.U$, and that $(t\ u) : U\{x/u\}$ (i.e., $(t\ u)$ is typable). By rule (App) we know that $u : T$. Then, by the definition of Δ above, $\Delta(t, u)$ is defined, which contradicts the assumption that $(t\ u)$ is faulty. □

By Theorem 3.3, 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 strong soundness is proving type preservation.

Theorem 3.4 (subject reduction) *If both $\Gamma \vdash M_1 : A$ and $M_1 \rhd^* M_2$ then $\Gamma \vdash M_2 : A$.*

Proof Sketch A proof for subject reduction can be adapted from [30]. □

With this result, we have shown that in **cGMM** well-typed terms reduce to a unique normalized term of the expected type. In turn, a term that causes a type error fails to typecheck.

4 Example revisited

In this section we demonstrate the application of **cGMM** by revisiting the *KM32ATLCopier* example introduced in Sect. 2. A megamodel is represented by an environment Γ, 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. For the *KM32ATLCopier* transformation example we use *KM3* as a concrete metametamodel instead of Type as before.

Transformation *KM32ATLCopier* is an ATL transformation, and as such it is atomic. This means that such a transformation is to be assumed in the environment. The concrete assumption is then

$$\mathit{KM32ATLCopier} : M{:}KM3 \rightarrow M \rightarrow M$$

The type assigned to *KM32ATLCopier* is a function type which depends on value M. Its co-domain is another function type, where both the domain and the co-domain are M (here the dependency is apparent). The Π-based expression of that type would be $\Pi M{:}KM3.\Pi x{:}M.M$, which is much less intuitive. Compare this assumption with types (T_e) and (T_r) from Sect. 2.2.

Now we apply the *KM32ATLCopier* transformation to the *SQL* metamodel. This should produce a copier transformation, which we call *SQLCopier*. Such a definition is then

$$\mathit{SQLCopier} := (\mathit{KM32ATLCopier}\ SQL)$$

Note that the type of *SQLCopier* was intentionally omitted from the definition, as we want it to be inferred. A type inference algorithm should return type $SQL \rightarrow SQL$, meaning that such a type is the type of *SQLCopier*, but also meaning that the functional application defining it is well-typed. A correct type inference algorithm finds the right type for a term. In our case, it would not be necessary to typecheck the functional application against the inferred type. However, for illustrating the operation of the type system, we show a derivation of the corresponding typing judgment. In fact, the type inference algorithm builds the derivation tree from the root to the leaves. For this reason such a derivation is interesting. We first define an environment Γ as follows:

$$\begin{aligned}\Gamma \equiv\ & KM3 : KM3,\\ & SQL : KM3,\\ & \mathit{KM32ATLCopier} : M{:}KM3 \rightarrow M \rightarrow M\end{aligned}$$

Then, the following typing judgment can be derived, which proves that the inferred type is a type for *SQLCopier*:

$$\Gamma \vdash (\mathit{KM32ATLCopier}\ SQL) : SQL \rightarrow SQL$$

The derivation is shown in Fig. 3. For simplicity, we assumed that environment Γ is valid, since its proof is trivial. Now we can safely apply *SQLCopier*. To this end, we augment

$$\dfrac{\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash \mathit{KM32ATLCopier} : M{:}KM3 \rightarrow M \rightarrow M}\,(\text{Assum}) \qquad \dfrac{\Gamma \vdash \diamond}{\Gamma \vdash SQL{:}KM3}\,(\text{Assum})}{\Gamma \vdash (\mathit{KM32ATLCopier}\ SQL) : SQL \rightarrow SQL}\,(\text{App})$$

Fig. 3 Derivation of (*KM32ATLCopier SQL*): *SQL*→*SQL* in Γ

$$\dfrac{\dfrac{\Gamma' \vdash \diamond}{\Gamma' \vdash \mathit{SQLCopier} : \mathit{SQL} \to \mathit{SQL}}\,(\mathrm{Def}) \qquad \dfrac{\Gamma' \vdash \diamond}{\Gamma' \vdash \mathit{s1} : \mathit{SQL}}\,(\mathrm{Assum})}{\Gamma' \vdash (\mathit{SQLCopier}\ \mathit{s1}) : \mathit{SQL}}\,(\mathrm{App})$$

Fig. 4 Derivation of (*SQLCopier s1*): *SQL* in Γ'

environment Γ as follows:

$$\begin{aligned}\Gamma' \equiv\ & KM3 : KM3,\\ & SQL : KM3,\\ & \mathit{KM32ATLCopier} : M{:}KM3 \to M \to M,\\ & \mathit{SQLCopier} := (\mathit{KM32ATLCopier}\ \mathit{SQL}),\\ & s1 : SQL\end{aligned}$$

We now consider definition *s2* := (*SQLCopier s1*). The expected type of such an application, and hence, of constant *s2* is *SQL*. Again, this type is returned by the type inference algorithm, and no typechecking is required. Nonetheless, we show in Fig. 4 the derivation of the following typing judgment:

$$\Gamma' \vdash (\mathit{SQLCopier}\ s1) : \mathit{SQL}$$

As before, we do not prove the validity of environment Γ'. Note the application of rule (Def) for typing the *SQLCopier* constant. For typing *KM32ATLCopier*, in the previous derivation, we applied rule (Assum) instead. This is because *KM32ATLCopier* was assumed, while *SQLCopier* was defined.

The type of assumed constants is necessarily provided by the user. One may argue that, in cases such as *KM32ATLCopier*, the term that types an assumed constant may be too complex to define. GMM enables the use of dependently typed higher-order transformations, and that complexity is inevitably projected to types. Note that user-specified types for functions is a common practice in programming languages. We believe that suitable discovery mechanisms such as MoDisco [17] would, at least, assist users in the type specification process.

In **cGMM**, higher-order transformations involving dependent types, such as *KM32ATLCopier*, can be properly typed. Our calculus successfully deals with the identified limitations of the current typing approach, since the type of the result of an application of those transformations can now be inferred.

5 Implementation

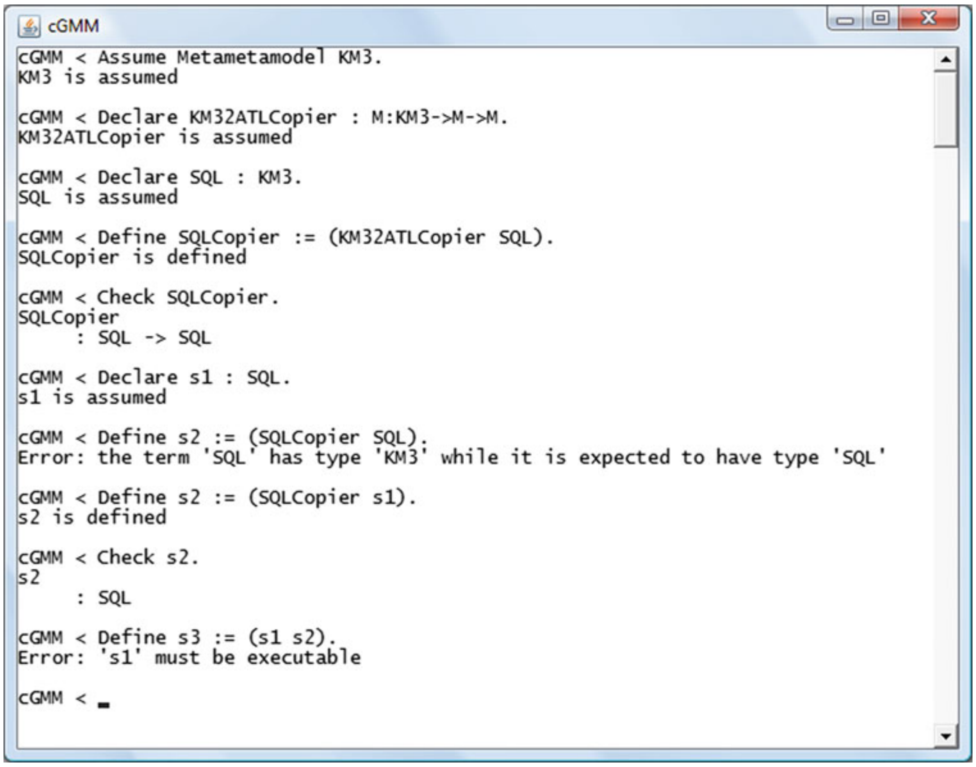
In this section we discuss our prototypical implementation of **cGMM** and its integration to a realization of GMM: the AM3 tool.

5.1 Implementation of cGMM

We developed the **cGMM** calculus as a Java stand-alone application called MK_1. It provides an environment which can be updated with assumptions and definitions. Such terms are representations of actual GMM elements within a megamodel, and the type system reasons about their types as required. MK_1 provides an *ITypeSystem* API which is used for feeding the environment with declarations, and for querying the type of terms within the environment. For expressing terms, we developed a simple textual language which is similar to *Gallina*, the specification language of Coq [23]. Calls to the API are translated to a textual command language similar to *The Vernacular*, the command language of *Gallina*. An ANTLR-based parser then builds **cGMM** terms from those commands. Type errors are handled by means of custom *TypeException* exceptions.

Figure 5 shows the commands involved in the *KM32ATLCopier* example discussed in the previous section. A console application that directly accesses the parser captures commands entered by a user and prints the results back. The `Assume` and `Declare` commands are used for assumptions, the `Define` command for definitions, and `Check` for retrieving type information. Assumed elements need to be explicitly typed; however, the system checks that the provided types are well formed. In turn, defined elements are checked for well-formedness, and their types are completely inferred by the type system. Such is the case of composite transformation `SQLCopier` and terminal model `s2`. Additionally, in the script of Fig. 5 we intentionally introduced two common error situations. First, we applied `SQLCopier` to an argument of the wrong type. The error message indicates the received type and the expected type. Second, we tried to apply `s1`, which is not a transformation, to `s2`. The error message indicates that the applied term is not executable.

Our implementation of the type system does not strictly follow the definition discussed in Sect. 3. The difference lies in how the Type:Type is realized. While our theoretic definition of **cGMM** includes an infinite hierarchy of sorts which emulates the Type:Type rule for achieving type soundness, our implementation drops the infinite hierarchy and includes that rule directly. This issue enables the possibility of divergence, for some cases, within our prototype. However, Cardelli stated that, in general purpose calculi with the Type:Type rule, examples leading to divergence in the typechecking process are extremely hard to reproduce [6]. In addition, a type system with the Type:Type rule is conceptually simpler than another with an infinite hierarchy of sorts. Furthermore, its implementation becomes simpler as complex mechanisms such as algebraic universes [23] for supporting the infinite hierarchy are not required. Our experiments with MK_1 on transformations from the ATL Transformation Zoo [2] and other practical transformations [25,26] were satisfactory. The



```
cGMM < Assume Metametamodel KM3.
KM3 is assumed

cGMM < Declare KM32ATLCopier : M:KM3->M->M.
KM32ATLCopier is assumed

cGMM < Declare SQL : KM3.
SQL is assumed

cGMM < Define SQLCopier := (KM32ATLCopier SQL).
SQLCopier is defined

cGMM < Check SQLCopier.
SQLCopier
     : SQL -> SQL

cGMM < Declare s1 : SQL.
s1 is assumed

cGMM < Define s2 := (SQLCopier SQL).
Error: the term 'SQL' has type 'KM3' while it is expected to have type 'SQL'

cGMM < Define s2 := (SQLCopier s1).
s2 is defined

cGMM < Check s2.
s2
     : SQL

cGMM < Define s3 := (s1 s2).
Error: 's1' must be executable

cGMM <
```

Fig. 5 The *KM32ATLCopier* example in the implementation of **cGMM**

right types were found for well-typed terms, and also ill-typed terms correctly threw exceptions during their construction. The question of whether a domain specific type system like the one we implemented, where the Type:Type rule is directly included, is safe from divergence is still open.

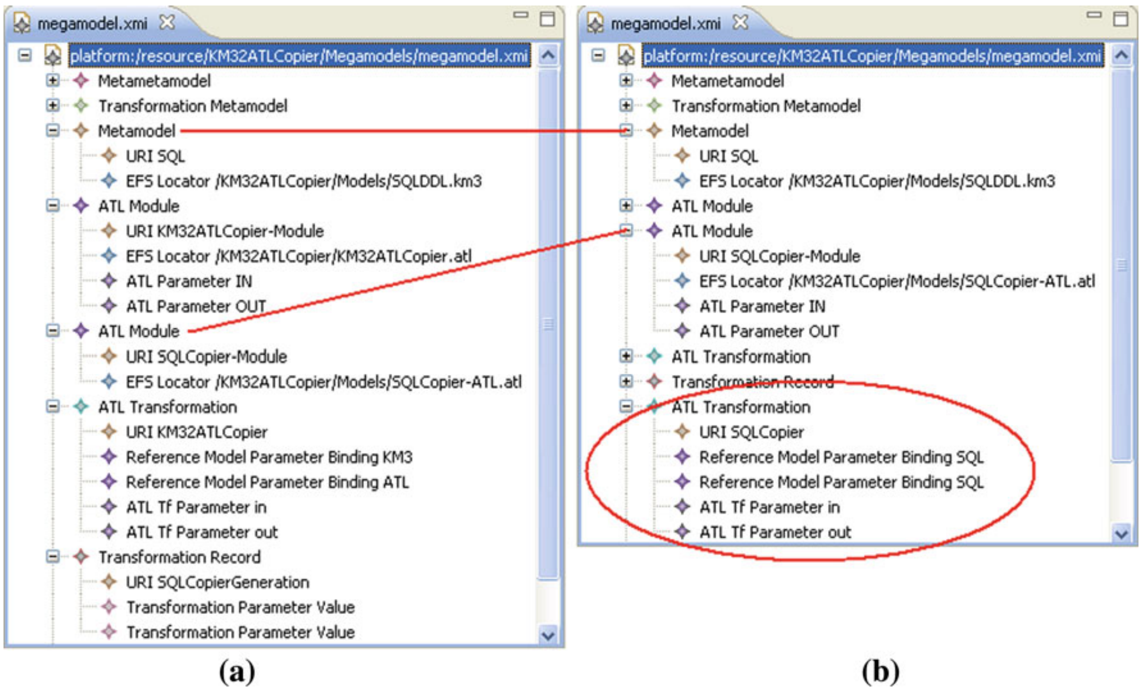
Our type system was prototyped as a separate application which can be easily tested and evolved. However, our ultimate goal is to integrate such an implementation with the prototypical realization of GMM, provided by the Eclipse-GMT AM3 project [1]. Next, we present some general information about its overall architecture and main features. Then, more details on the integration of the type system with the AM3 GMM prototype are given, still taking the same *KM32ATLCopier* transformation as a test example.

5.2 The eclipse-GMT AM3 global model management solution

The current version of the Eclipse.org *AM3* solution implements the conceptual framework described in Sect. 2, and thus it can be used as the GMM tool in the context of the integration. 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 [1]. The generic and extensible AM3 global model management solution, built on top of the Eclipse environment, provides not only the capabilities to explicitly specify the metadata associated with a given modeled system or MDE process, but also a standard *Megamodel Navigator* as well as generic and extensible editors for instantiating and editing the megamodel in a user-friendly way. In addition, it offers several extension points allowing the definition of domain-specific extensions of the tool (i.e., extending both the metamodel of megamodels and the related UI components). Thus, AM3 is composed of two distinct sets of Eclipse plug-ins:

- The *core* plug-ins provide the basic metamodel of megamodels, the core runtime environment, the main APIs and associated generic navigator, and editors.
- The *extension* plug-ins provide extensions of the metamodel of megamodels, related specific APIs and corresponding extensions of the UI (for instance specific editor pages, contextual actions, etc).

With AM3, users can build their customized megamodeling solutions by extending either the core plug-ins or other already existing extension plug-ins. Indeed, a set of generic MDE extensions have already been developed: GMM4GlobalModelManagement which implements the GMM conceptual framework, GMM4ATL for dealing with model transformations in ATL, GMM4CompositeTransformations for supporting composite transformations, etc.



(a)

```
megamodel.xmi
platform:/resource/KM32ATLCopier/Megamodels/megamodel.xmi
  Metametamodel
  Transformation Metamodel
  Metamodel
    URI SQL
    EFS Locator /KM32ATLCopier/Models/SQLDDL.km3
  ATL Module
    URI KM32ATLCopier-Module
    EFS Locator /KM32ATLCopier/KM32ATLCopier.atl
    ATL Parameter IN
    ATL Parameter OUT
  ATL Module
    URI SQLCopier-Module
    EFS Locator /KM32ATLCopier/Models/SQLCopier-ATL.atl
  ATL Transformation
    URI KM32ATLCopier
    Reference Model Parameter Binding KM3
    Reference Model Parameter Binding ATL
    ATL Tf Parameter in
    ATL Tf Parameter out
  Transformation Record
    URI SQLCopierGeneration
    Transformation Parameter Value
    Transformation Parameter Value
```

(b)

```
megamodel.xmi
platform:/resource/KM32ATLCopier/Megamodels/megamodel.xmi
  Metametamodel
  Transformation Metamodel
  Metamodel
    URI SQL
    EFS Locator /KM32ATLCopier/Models/SQLDDL.km3
  ATL Module
  ATL Module
    URI SQLCopier-Module
    EFS Locator /KM32ATLCopier/Models/SQLCopier-ATL.atl
    ATL Parameter IN
    ATL Parameter OUT
  ATL Transformation
  Transformation Record
  ATL Transformation
    URI SQLCopier
    Reference Model Parameter Binding SQL
    Reference Model Parameter Binding SQL
    ATL Tf Parameter in
    ATL Tf Parameter out
```

Fig. 6 Megamodel Samples for the *KM32ATLCopier* transformation (respectively before/after type derivation)

5.3 Integrating the type system with the AM3 solution

In order to fully exploit the benefits of the presented type system in a concrete environment, it would be better to have it fully integrated with the current implementation of the AM3 tool. To that end, we introduce a new extension plug-in for GMM named TypeSystem4GMM. Such a plug-in wraps the implementation of the type system. When a type-related event occurs within the AM3 tool, it notifies the TypeSystem4GMM extension by an appropriate message of its provided interface *ITypeSystem* mentioned in Sect. 5.1. After the plug-in extension has processed the message, it returns the result in a suitable data structure, or throws a *TypeException* exception.

For the described mechanism to work, some modifications are required at specific locations of some of the plug-ins of the AM3 tool:

1. Extend the current GMM4GlobalModelManagement extension so that all the information needed by the type system for a successful evaluation can be represented within the underlying megamodel;
2. Modify the transformation executors in the GMM4ATL and GMM4CompositeTransformations 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 complete type information;
3. Update the corresponding editors. In the current implementation, terms such as *Class2Relational* from Sect. 3.1, which is a first-order atomic transformation, can be created. For the *KM32ATLCopier* case which is a HOT, specifying *KM3* as the source metamodel must introduce a variable *M*. Then, specifying *ATL* as the target metamodel should allow the user to express that *M* will be both the source and the target of the resulting transformation.

As an illustration, let us consider a megamodel registering the KM32ATLCopier model transformation (along with the KM32ATLCopier-Module transformation model[1]) and the SQL metamodel. After KM32ATLCopier is applied to SQL, according to the current AM3 implementation the SQLCopierGeneration transformation record is created. Its target model is therefore the SQLCopier-Module transformation model. The state of the megamodel is shown in Fig. 6a. Note that the source and target models of SQLCopier-Module were not created because the current typing approach does not provide enough information for doing so. Additionally, the corresponding model transformation (i.e., SQLCopier) was not created for the same reason. The megamodel resulting from deriving the type of the result of such an execution is shown in Fig. 6b. According to the type derivations discussed before, it is possible to know that the result of the execu-

[1] We adhere to the following naming convention. A transformation has a dual representation (i.e., it is represented as two separate elements): a transformation model (entity) and a model transformation (relationship). Since elements must have unique names, a transformation named X induces a relationship named X and an entity named X-Module.

tion has type $SQL \rightarrow SQL$, and thus the information for properly completing SQLCopier-Module and creating the SQLCopier relationship is available.

To summarize, TypeSystem4GMM is integrated with AM3 in such a way that it enables the inference of the previously lacking type information. As a result, the underlying megamodel can now be automatically updated with such a computed information.

6 Related work

The version of **cGMM** discussed in this paper is a major improvement of the version we introduced in [27]. 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. Based on these improvements, we also reimplemented our original prototype of the type system, for producing MK_1. A concrete architecture and a precise scheme for its integration with AM3 was also produced. In this work we also sketched a proof of type soundness. Finally, [25] is an extension to **cGMM** for typing textual entities, and model-to-text and text-to-model transformations. In turn, [26] extends **cGMM** for typing weaving models.

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 can in fact be 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 [8], 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, [9] 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 [22] 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, HOTs are not addressed. Our type system does not deal with intra-resource issues and focuses on inter-resource ones.

Constructive Type Theory was used in [20] 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 with 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. Since MOF was the only metametamodel, no additional hierarchies of sorts are required as in our case. Such a formalism focuses on MOF, and therefore only applies to MOF-based artifacts. This 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.

Typechecking of compositions of transformations has been addressed in [29] and in more detail in [24]. 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.

7 Conclusions and further work

GMM includes the notion of transformation execution and typing in that context is required for preventing type errors during that execution. We improved the current typing approach by proposing a type system that formally indicates how to reason about types in GMM. We showed how non-trivial situations, such as the use of HOTs, combined with dependent types, can now be handled.

The current version of **cGMM** enables one single metametamodel. GMM is designed for supporting many metametamodels concurrently. Nevertheless, the AM3 tool currently supports one single metametamodel, and emulates a multi-metametamodel environment by means of the promotion and demotion mechanisms. Such mechanisms rely on first-order atomic transformations and can be easily represented in **cGMM**. However, it would be interesting to extend **cGMM** for natively supporting a multi-metametamodel approach.

We prototyped **cGMM** with good results. Our prototype is intended to be fully integrated with the AM3 tool. Such a prototype does not implement a hierarchy of sorts; instead, it is based on the Type:Type rule. We have not yet encountered divergence in our domain specific calculus. The lack of recursion and the fact that definitions are ultimately compositions of assumed functions within a finite environment are a probable cause of it. The Type:Type rule would simplify both the calculus and its implementation, and particularly, a native support for various metametamodels would benefit from this. Type soundness of the implemented version of our calculus is one of our main directions of further research.

Our type system ensures good behavior but relying on the good behavior of atomic transformations. A stronger level of type safety would be achieved by integrating our type system with the type system of a transformation language. Both type systems need to be aligned and the result of the integration should still be sound, and this issue is delicate [30]. ATL would be an appropriate case for investigating this issue. In turn, our type system would benefit from including subtyping, not only for model types as in [22], but also for function types as well. This would enable substitutability for both models and transformations. Finally, composite transformations are currently defined by the user. Type information is key for supporting this manual process. But additionally, when building a composite transformation from a given set of source types to another set of target types, it may be possible to infer (parts of) well-typed chains of compositions. We plan another integration of MK_1 with Wires* [21]. That tool provides a graphical executable language for the orchestration of complex ATL transformations chains, but does not typecheck the defined compositions.

Acknowledgments We would like to specially thank Alvaro Tasistro for his valuable and numerous contributions to this work. The present work has been jointly supported by the IST-FP6 MODELPLEX European project [15] and the Conicyt-Chile/INRIA HOT project. The work of Andrés Vignaga has been partially funded by Conicyt-Chile.

References

1. AM3 Project. Internet: http://www.eclipse.org/gmt/am3/ (2009)
2. ATL Transformations Zoo. http://www.eclipse.org/m2m/atl/atlTransformations/, (2009)
3. Barbero, M., Jouault, F., Bézivin, J.: Model driven management of complex systems: implementing the macroscopes vision. In: 15th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS 2008), 31 March–4 April 2008, Belfast, Northern Ireland, pp. 277–286. IEEE Computer Society (2008)
4. Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., Lindow, A.: Model transformations? Transformation models! In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1–6, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4199, pp. 440–453. Springer, New York (2006)
5. Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: Modeling in the large and modeling in the small. In: Aßmann, U., Aksit, M., Rensink, A. (eds.) Model Driven Architecture, European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Twente, The Netherlands, June 26–27, 2003 and Linköping, Sweden, June 10–11, 2004, Revised Selected Papers. Lecture Notes in Computer Science, vol. 3599, pp. 33–46. Springer, New York (2004)
6. Cardelli, L.: Typechecking dependent types and subtypes. In: Boscarol, M., Aiello, L.C., Levi, G. (eds.) Foundations of Logic and Functional Programming, Workshop, Trento, Italy, December 15–19, 1986, Proceedings. Lecture Notes in Computer Science, vol. 306, pp. 45–57. Springer, New York (1986)
7. Cardelli, L: Type Systems. In: Tucker, A.B. (ed.) The Computer Science and Engineering Handbook, pp. 2208–2236. CRC Press, Boca Raton (1997)
8. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. **45**(3), 621–646 (2006)
9. Favre, J.-M.: Towards a basic theory to model model driven engineering. In: 3rd Workshop in Software Model Engineering, Lisbon, Portugal (2004)
10. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. Sci. Comput. Program. **72**(1-2), 31–39 (2008)
11. Jouault, F., Bézivin, J.: KM3: a DSL for metamodel specification. In: Gorrieri, R., Wehrheim, H. (eds.) Formal Methods for Open Object-Based Distributed Systems, 8th IFIP WG 6.1 International Conference, FMOODS 2006, Bologna, Italy, June 14–16, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4037, pp. 171–185. Springer, New York (2006)
12. Jouault, F., Kurtev, I. : Transforming models with ATL. In: Bruel, J.-M. (ed.) MoDELS Satellite Events. Lecture Notes in Computer Science, vol. 3844, pp. 128–138. Springer, New York (2005)
13. Luo, Z.: An extended calculus of constructions. Ph.D. Thesis, University of Edinburgh (1990)
14. MacQueen, D.B.: Using dependent types to express modular structure. In: Thirteenth Annual ACM Symposium on Principles of Programming Languages, pp. 277–286. St. Petersburg Beach, Florida, January 1986
15. MODELPLEX IST-FP6 European Project: https://www.modelplex-ist.org/ (2009)
16. ModelPlex Project: Deliverable D2.13.b: Model Management Supporting Tool. https://www.modelplex.org//index.php?option=com_remository&Itemid=%0&func=startdown&id=183. March 2009
17. MoDisco: http://www.eclipse.org/MoDisco/ (2010)
18. Paulin-Mohring, C.: Le Système Coq. Thèse d'habilitation. ENS Lyon (1997)
19. Plotkin, G.D.: Call-by-name, call-by-value and the λ-calculus. Theor. Comput. Sci. **1**(2), 125–159 (1975)
20. Poernomo, I.: A type theoretic framework for formal metamodelling. In: Reussner R.H., Stafford J.A., Szyperski C.A. (eds.) Architecting Systems with Trustworthy Components, International Seminar, Dagstuhl Castle, Germany, December 12–17, 2004. Revised Selected Papers,. Lecture Notes in Computer Science, vol. 3938, pp. 262–298. Springer, New York (2006)
21. Rivera, J.E., Ruiz-González, D., López-Romero, F., Vallecillo, A.: Orchestrating ATL model transformations. In: Jouault, F. (ed.) 1st International Workshop on Model Transformation with ATL, MtATL 2009, pp. 34–46, Nantes, France, Proceedings, July 8–9, 2009
22. Steel, J., Jézéquel, J.-M.: On model typing. Softw. Syst. Model. **6**(4), 401–413 (2007)
23. The Coq Proof Assistant Reference Manual: Version 8.2. http://coq.inria.fr/doc-eng.html (2009)
24. Vanhooff, B., Ayed, D., Baelen, S.V., Joosen W., Berbers, Y.: UniTI: a unified transformation infrastructure. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F.(eds.) Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30–October 5, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4735, pp. 31–45. Springer, New York (2007)
25. Vignaga, A.: Typing textual entities and M2T/T2M transformations in a model management environment. In: Arenas, M., Bustos, B.

(eds.) 2009 International Conference of the Chilean Computer Science Society, pp. 115–122. IEEE Computer Society (2009)
26. Vignaga, A., Bastarrica, M.C.: Verification of megamodel manipulations involving weaving models. Technical Report TR/DCC-2009-9. Universidad de Chile, Computer Science Department, October 2009
27. Vignaga, A., Jouault, F., Bastarrica, M.C., Brunelière, H.: Typing in model management. In: Paige, R.F. (ed.) Theory and Practice of Model Transformations. Proceedings of Second International Conference, ICMT 2009, Zurich, Switzerland, June 29–30, 2009. Lecture Notes in Computer Science, vol. 5563, pp. 197–212. Springer, New York (2009)
28. Werner, B.: Une Théorie des constructions inductives. Thèse de doctorat, Université Paris 7 (1994)
29. Willink, E.D.: OMELET: exploiting meta-models as type systems. In: Akehurst, D.H. (ed.) 2nd European Workshop on MDA, pp. 160–164. University of Kent, Canterbury (2004)
30. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Inf. Comput. **115**(1), 38–94 (1994)

Author Biographies

Andrés Vignaga is a Ph.D. student at the MaTE team at the University of Chile. He received from Universidad de la República (Uruguay) an engineering degree in 2000 and an MSc degree in 2004. He has been working on applications of formal methods to object technology since 2000. His thesis dealing with the formalization of the semantics of state modification primitives of object systems won a CLEI/UNESCO award. His work on formal methods applied to model-driven engineering started in 2006. He completed two internships at the AtlanMod team at the Ecole des Mines of Nantes, while developing his Ph.D. thesis for the AM3 megamodeling tool. He currently manages the software development department of a software factory in Chile, where model-driven development is applied to industrial projects.

Frédéric Jouault is a researcher in the AtlanMod team. He received his Ph.D. in September 2006 from the University of Nantes. He did a postdoc at the University of Alabama at Birmingham in 2007. His research interests involve model engineering, model transformation, and their application to Domain-Specific Languages (DSLs) and model-based legacy reverse engineering. Frédéric created AtlanMod Transformation Language (ATL), a DSL for model-to-model transformation. He is now leading the development of ATL (language and toolkit) on Eclipse.org. He is in charge of the Eclipse modeling M2M project as well as a member of the modeling PMC. Frédéric is also involved in the MDE Diploma of Ecole des Mines de Nantes, of which he is deputy lead, and in which he notably teaches about ATL.

María Cecilia Bastarrica is an Assistant Professor at the Computer Science Department, at the Universidad de Chile. She coordinates the MaTE group (Model and Transformation Engineering) since 2007. She received her Ph.D. in Computer Science and Engineering from the University of Connecticut in 2000, a Master of Science from the Pontificia Universidad Católica de Chile in 1994, and a Bachelor in Engineering from the Catholic University of Uruguay in 1991. Her main research topics are software engineering, software architecture, model-driven engineering, and software product lines. Lately, her work has focused on applying using MDE techniques for modeling software processes.

Hugo Brunelière is an R&D engineer working in the field of Model Driven Engineering (MDE) for the AtlanMod team, with focus on tool interoperability (based on model transformation), reverse engineering, and global model management. He received a Master's Degree in Computer Science in 2006. Since then, he has been working as the responsible for the INRIA coordination on the MODELPLEX IST European project which ended in February 2010. He is active in the Eclipse community, both as an official project leader/committer (MoDisco, AM3, EMF Facet) and regular user of several other projects (EMF, ATL, etc). In addition, he has published over ten papers in various conferences and workshops around MDE, and he does teaching in the newly opened MDE Diploma at Ecole des Mines de Nantes.