Typing Textual Entities and M2T/T2M Transformations in a Model Management Environment

Andrés Vignaga
MaTE, Computer Science Department, Universidad de Chile
Santiago, Chile,
avignaga@dcc.uchile.cl

Abstract

Global Model Management (GMM) is a model-based approach for managing large sets of interrelated heterogeneous and complex MDE artifacts. Such artifacts are usually represented as models, however as many Domain Specific Languages have a textual concrete syntax, GMM also supports textual entities and model-to-text/text-to-model transformations which are projectors that bridge the MDE technical space and the Grammarware technical space. As the transformations supported by GMM are executable artifacts, typing is critical for preventing type errors during execution. We proposed the eGMM calculus which formalizes the notion of typing in GMM. In this work, we extend eGMM with new types and rules for supporting textual entities and projectors. With such an extension, those artifacts may participate in transformation compositions addressing larger transformation problems. We illustrate the new constructs in the context of an interoperability case study.

Keywords: Model-Driven Engineering; Global Model Management; Type System; Model Transformation; Textual Concrete Syntax

1. INTRODUCTION

Models have been increasingly used in the last decades by software developers for coping with the growing complexity of computer-based systems. Model-Driven Engineering (MDE) positioned models as first class constructs, which may be expressed in Domain Specific Languages (DSL) and mechanically manipulated by transformations. MDE-based projects then involve a variety of artifacts such as models, metamodels, transformations, and so on. While handling a small amount of such artifacts is easy, the large number of interrelated, heterogeneous and complex artifacts involved in industrial cases quickly becomes unmanageable. Global Model Management (GMM) [1] is a model-based approach for managing MDE artifacts. GMM is centered on the notion of a megamodel [2]. A megamodel is a special model which logically contains manageable artifacts and relationships among them.

Some artifacts involved in an MDE-based project may additionally have an alternate representation which is considered to be external with respect to the realm of models. Then it is possible to represent an artifact \( a \) as a model \( a_m \) and as a non-model \( a_e \). On the one hand, \( a_m \) can be manipulated via model transformations. In the other hand, the \( a_e \) representation of \( a \) could be required per se, and also can be more appropriately manipulated either manually by a developer or automatically by existing external tools. Bridging those two worlds enable the generation of \( a_m \) from \( a_e \) and vice versa. One common case of external representations is the case of textual languages. In particular, 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 edition. When the external representation is textual, conversion is known as model-to-text and text-to-model transformations. Textual Concrete Syntax (TCS) [3], [4] is a concrete technology for realizing such kinds of transformations.

As megamodels may contain executable artifacts, their application to other elements may lead to type errors during execution. Typical type errors are the application of a non-transformation, and the application of a transformation on arguments for which it was not defined. Typing elements within a megamodel is an effective means for preventing such kind of execution errors. GMM's original typing approach of artifacts was informally based on typing by metamodel. In [5] we showed that such an approach cannot appropriately handle non-trivial cases, and we proposed the eGMM calculus which formally addresses typing in GMM. The case of model-to-model transformations was further treated in [6]. GMM currently supports TCS models and textual representations of artifacts; however, their
typing according to the new approach was not addressed yet. In this work, we extend \textit{cGMM} for supporting textual entities, and both model-to-text (M2T) and text-to-model (T2M) transformations.

The rest of this paper is structured as follows. In Sect. 2 we provide background information on the technical spaces we will be dealing with and on the GMM approach, and we overview the \textit{cGMM} calculus. In Sect. 3 we address the typing of textual entities, and M2T and T2M transformations. In Sect. 4 we demonstrate our approach by typing artifacts involved in a case study taken from the bibliography. Section 5 discusses the implementation of the extension to \textit{cGMM} introduced in Sect. 3. Section V concludes.

2. BACKGROUND

In this section we provide an overview of the MDE and Grammarware technical spaces. We also overview the GMM approach, or Megamodeling, from a broad point of view, for taking into consideration both textual entities and bridges. Finally, we summarize the key concepts of the \textit{cGMM} calculus.

2.1 MDE and Grammarware Technical Spaces

Artifacts can be organized into technical spaces [7]. A technical space 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. There are actually different metalanguages for defining modelling languages. For example, ECore is one such metalanguage and determines the EMF technical space. In turn, the Grammarware [8] technical space enables the definition of language grammars. We consider EBNF as its metalanguage.

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 \textit{projectors}. A projector is an artifact that lives in a technical space \( R \); if it transfers artifacts from \( R \) to another technical space it is called an \textit{extractor}, and it is called an \textit{injector} if it transfers artifacts in the opposite direction.

2.2 Global Model Management Approach

Every GMM provides an extensible metamodel which specifies all kinds of \textit{elements} that can be managed. A simplification of the basic part of such a metamodel is illustrated in Fig. 1. Managed elements are entities and relationships among them. The primary entities are models, and as such GMM focuses on the MDE technical space. Models are understood in three levels:

- \textit{terminal models} (M1) conform to \textit{metamodels} and are representations of real-world systems.
- \textit{metamodels} (M2) conform to \textit{metametamodels} and define domain-specific concepts.
- \textit{metametamodels} (M3) conform to themselves and provide generic concepts for metamodel specification.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure1.png}
\caption{Basic concepts of the Global Model Management conceptual framework}
\end{figure}
One core concept in GMM is a megamodel. A megamodel is a special kind of terminal model which logically contains manageable artifacts. Other variants of terminal models are transformation models and weaving models. In particular, transformation models can be executed on entities within a megamodel for producing other entities to be also included in the megamodel.

In GMM, not every entity is a model. In fact, GMM supports textual entities. Such entities are M1-level elements of the Grammarware technical space. For example, defining the Java metamodel in ECore enables the representation of a Java program as a terminal model in the EMF technical space. In turn, provided the Java grammar in EBNF, that same Java program may be represented as a source code in the Grammarware technical space. Finally, such source code can be also represented as a textual entity in a megamodel in the EMF technical space. A transformation from the Java model to the Java source code is a model-to-text transformation. If realized as a model in an MDE technical space, then it is an extractor. In turn, a transformation from the Java source code to the Java model is a text-to-model transformation. Again, if realized as a model, then it is an injector.

One interesting case is the case of TCS, which is a DSL for generating projectors such as those discussed above. A TCS artefact is a model (MDE technical space) from which both an injector and an extractor are generated. Such a model is specifically a TCSModel, which is a special kind of TransformationModel shown in Fig. 1. However, TCS has a textual concrete syntax, and therefore textual TCS specifications (Grammarware technical space) may be handled. For example, the target of some 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 some transformation. Finally, both projectors can be realized by TCS itself.

Managing textual entities and M2T and T2M transformations is a common situation in GMM.

### 2.3 Typing in GMM

We formalized a typing approach for GMM by means of the cGMM calculus [5]. cGMM is a predicative dependently typed λ-calculus based on Constructive Type Theory. It is similar to a subset of the Predicative Calculus of (Co)Inductive Constructions (pCIC) [9], [10]. We map GMM constructs to cGMM terms. Terms are based on standard concepts such as constants, abstractions, dependent products and applications. cGMM includes an environment where declarations and definitions (involving typed terms) are included. Such an environment represents the underlying megamodel, and its contained elements are represented by declarations and definitions. A type system for cGMM formally defines the notion of typing in GMM and provides a precise means for reasoning about types.

If a Type Theory (TT) technical space can be conceived, then we are bridging the MDE and TT technical spaces. A megamodel containing elements is a terminal model which is projected to an environment containing cGMM terms. Megamodeling and 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 where it is efficiently carried out.

The has-type relation in cGMM is based on the directed association from Model to ReferenceModel in Fig. 1. Model \( m \) is typed by reference model \( M \) (this is denoted as \( m \vdash \exists M \)) iff model \( m \) conforms to reference model \( M \). Note that as textual entities are not models; they cannot be typed this way. We use \( \lambda \) for typing reference models and terminal models which are not transformation models. Models in general are constants, and reference models in particular are constants which may occur at either side of a ‘:\ ’ operator. This implies that in cGMM we do not make a syntactical distinction between terms and types. We introduce an infinite hierarchy of universes for representing metametamodels. This enables a metametamodel to be typed by itself, for example, ECore: ECore. However, some elements are not typed according to \( \lambda \). This is the case of transformation models, which are typed by (possibly dependent) products. A non-dependent product represents classical function types, while dependent products represent either dependent function types (for handling transformations whose type depends on a metamodel) and universal quantifications (for handling higher-order transformations). Abstractions enable the definition of composite transformations, and applications represent transformation execution.

Finally, the type of some terminal models refer to other types, and new type constructors provide richer type information for those cases. We experimented with ATL libraries [6] and weaving models [11], and we concluded that specific types enable stronger checks. Following the same approach for textual entities, M2T and T2M transformations, we integrate such artifacts to cGMM for achieving similar benefits.

### 3. Typing Textual Entities and M2T/T2M Transformations

Textual entities are not models and therefore the \( \lambda \) relation is not applicable to them. In turn, typing a TCS model by \( \lambda \), as it is a transformation, produces the loss of type information about its source and target [5]. As an overall result, it would not be possible to know in general if a given textual entity could be safely passed as an argument to a given injector, and for a projector, it would not be possible to know what type of element it safely accepts or
produces. Such situations are sources of type errors that \texttt{cGMM} must prevent. In this section we start by defining a type constructor for defining types for textual entities, and based on them, we use dependent products for typing M2T and T2M transformations.

### 3.1 A Type for Textual Entities

We start by defining type \texttt{Textual}, which is the type of all textual types. In other words, an instance of \texttt{Textual} is a type. This is formalized by the following rule:

\[
\begin{align*}
\Gamma & \vdash \text{Textual} \\
\Gamma & \vdash X : \text{Textual} \\
\Gamma & \vdash X : \text{Type}
\end{align*}
\]

A textual entity is a textual representation of a terminal model which conforms to a given metamodel. Textual entities also conform to the grammar of their language. We can categorize textual entities according to the grammar they conform to. However, grammars are external to the MDE technical space. But as grammars correspond to metamodels, and metamodels live in fact in the MDE technical space, we then categorize textual entities according to the metamodel that types their corresponding model representation. To that end, we introduce a type constructor of the form \texttt{[Text A]} for a given metamodel \(A\), \texttt{[Text A]} is the type of all textual entities which are a textual representation of a terminal model typed by \(A\). Note that if we shift one level up, the situation is essentially the same, and we could have textual representations of metamodels as well. Furthermore, if we shift one level up again we still find the same situation. We can relax the constraint on \(A\) as expressed in the following rule:

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

Note that when \(A\) is a reference model, it can be trivially proved that \texttt{[Text A]} : \texttt{Type}. Now, having a transformation that accepts textual entities of type \texttt{[Text Class]}, it will not accept textual entities of type \texttt{[Text Relational]}, since both types are different.

As a final remark, model-to-model transformations cannot involve textual entities. Therefore the introduction of this type requires an additional constraint on the type formation rule of M2M transformations: neither the source nor the target types may involve a textual type.

### 3.2 Types for M2T and T2M Transformations

Model-to-text and text-to-model transformations are transformations, and therefore they should be typed analogously to model-to-model transformations. In \texttt{cGMM}, such transformations are typed by dependent products. A dependent product may represent a (dependent) function type, which is appropriate for recording the source and target types of the transformation. Additionally, any variant of transformation can be the source of a higher-order transformation (HOT). As HOTs need to know the language in which such a source was defined [6], we need to introduce further types corresponding to concrete projector definition languages. Since currently the only language for that matter that is supported in GMM is TCS, we introduce the following two rules:

\[
\begin{align*}
\Gamma & \vdash X : \text{TCSExtraction} \\
\Gamma & \vdash X : \text{M2TTransformation}
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash X : \text{TCSInjection} \\
\Gamma & \vdash X : \text{T2MTransformation}
\end{align*}
\]

\texttt{M2TTransformation} and \texttt{T2MTransformation} are the types of all model-to-text and text-to-model transformations respectively, regardless of the concrete realization technology. For making those two types effective types we need further rules:

\[
\begin{align*}
\Gamma & \vdash X : \text{M2TTransformation} \\
\Gamma & \vdash X : \text{M2ETransformation}
\end{align*}
\]
where \texttt{M2ETransformation} and \texttt{E2MTransformation} are the types of all model-to-external and external-to-model transformations respectively. Note that “text” is just one particular case of “external”. In this way, types for new foreign technical spaces can be flexibly added to our type system. We complete this chain of rules with rules involving types \texttt{Projector} (the type of all possible projectors), \texttt{ExternalTransformation} (the type of all transformations that are not model-to-model), \texttt{Transformation} (the type of all transformations), and so on until we reach \texttt{Type}.

Now we can introduce type formation rules for TCS extractions and injections. For an extraction, the reference model involved in the target must be the same as that in the source:

\begin{align*}
\Gamma \vdash A : \texttt{ReferenceModel} & \quad \Gamma \vdash [\texttt{Text A}] : \texttt{Textual} \\
\hline
\Gamma \vdash A \rightarrow [\texttt{Text A}] : \texttt{TCExtraction}
\end{align*}

In turn, for an injection, the constraint is exactly the inverse:

\begin{align*}
\Gamma \vdash A : \texttt{ReferenceModel} & \quad \Gamma \vdash [\texttt{Text A}] : \texttt{Textual} \\
\hline
\Gamma \vdash [\texttt{Text A}] \rightarrow A : \texttt{TCSIjection}
\end{align*}

One of the motivations for defining bridges between technical spaces \(TS_1\) and \(TS_2\) is the reuse in \(TS_1\) of a facility already existing in \(TS_2\) which is too costly to define in \(TS_1\) [4]. Although GMM does not currently support transformations which completely lie in a foreign technical space, such transformations do occur in practice and we believe it would be useful if \texttt{cGMM} could be able to support them. In this case, we consider “programs” which are closed to the Grammarware technical space. Such transformations then could occur in the definition of a composite (model-to-model) transformation. As before, we introduce types and their corresponding rules. \texttt{Program} is the type of all text-to-text transformations, and \texttt{E2ETransformation} is the type of all external-to-external transformations, which is a subtype of type \texttt{Transformation} mentioned before. Then the type formation rule for \texttt{Program} is:

\begin{align*}
\Gamma \vdash [\texttt{Text A}] : \texttt{Textual} & \quad \Gamma \vdash [\texttt{Text B}] : \texttt{Textual} \\
\hline
\Gamma \vdash [\texttt{Text A}] \rightarrow [\texttt{Text B}] : \texttt{Program}
\end{align*}

In the next section we discuss a case study involving all the constructs presented so far.

\section*{4. Example}

In this section we demonstrate our approach on the case study presented in [12]. 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 are 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 [13]. 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. 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 2 illustrates all required artifacts. A \texttt{Sample Class Diagram} is transformed, via the \texttt{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 \texttt{DotExtractor} extractor, to a textual entity representing the same graph. The auto-layout program \texttt{Dot.exe} inserts appropriate layout coordinates to the graph, which is injected by the TCS \texttt{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 \texttt{Merge} ATL transformation for producing the \texttt{Final Class Diagram}.
Now we discuss each **eGMM** term representing an artifact of this case study. Terms usually depend on other terms already in the environment. For example declaring a terminal model which is typed by a metamodel that is not in the environment will cause an error. Therefore we proceed in the proper order. We start by declaring the metamodel:

\[ ECore : ECore \]

Then we declare the metamodels typed by the previously declared metamodel:

\[ Notation : ECore \]
\[ Dot : ECore \]

Now we proceed with declaring all required transformations:

\[ Notation2Dot : Notation \rightarrow Dot \]
\[ DotExtractor : Dot \rightarrow [Text Dot] \]
\[ DotExe : [Text Dot] \rightarrow [Text Dot] \]
\[ DotInjector : [Text Dot] \rightarrow Dot \]
\[ Merge : Notation \times Dot \rightarrow Dot \]

*Notation2Dot* is an ATL transformation, *DotExtractor* is a TCS extractor, *DotExe* is a program, *DotInjector* is a TCS injector, and *Merge* is another ATL transformation. From the source and target types of a transformation, it is straightforward to recognize which kind of transformation it is. Note that since *Dot* is a metamodel (and thus a reference model), the types of *DotExtractor*, *DotExe* and *DotInjector*, by virtue of the rules discussed in the previous section, are valid types. The sample class diagram is declared as follows:

\[ sampleCD : Notation \]

Once all pre-existing artifacts are declared, we can apply the transformations in order to obtain all intermediate artifacts. Applications are defined as follows:

\[ sampleGrM := (Notation2Dot \ sampleCD) \]
\[ sampleGrT := (DotExtractor \ sampleGrM) \]
\[ sampleGrTwC := (DotExe \ sampleGrT) \]
\[ sampleGrMwC := (DotInjector \ sampleGrTwC) \]
\[ finalCD := (Merge<sampleCD,sampleGrMw>) \]
Type rules for applications require the applied element to have a functional type. Furthermore, they require arguments to be of the proper type. In this way, definitions that do not follow the rules are considered as invalid since they would cause an execution error. Note that while declared elements are explicitly typed, types for defined elements are derived. For example, when the definition of \texttt{sampleGrMwC} is typechecked, the type \texttt{Dot} is inferred and assigned to it.

Finally, since we can even type programs such as \texttt{DotExe}, 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 we discussed above. We define transformation \texttt{AutoLayout} as:

\begin{verbatim}
AutoLayout (in: Notation) :=
  (Merge <in,
   (DotInjector
    (DotExe
      (DotExtractor
        (Notation2Dot in)
      )
    )
  )
)
\end{verbatim}

This transformation accepts an argument \texttt{in} of type \texttt{Notation}, and its body is defined as the composition of the transformations declared before. The body is typechecked using the same rules for applications already discussed. Using type rules, it can be proved that the type of \texttt{AutoLayout} is simply \texttt{Notation→Notation}. Then an alternate way of defining \texttt{finalCD} would be:

\texttt{finalCD := (AutoLayout sampleCD)}

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

5. IMPLICATION
We developed the \texttt{cGMM} calculus as a stand-alone component. It provides an environment which can be updated with new declarations and definitions. It supports \texttt{cGMM} terms that correspond to all elements defined in the core GMM metamodel and all its current extensions. Such terms are type-centric representations of actual GMM elements and the type system reasons about their types as required. The component provides an \texttt{ITypeSystem} API which is used for feeding the environment with new declarations and definitions, and for querying the type of elements within the environment. For expressing terms, we developed a simple textual language which is similar to \texttt{Gallina}, the specification language of Coq [14]. Calls to the API are translated to a textual command language similar to \texttt{The Vernacular}, the command language of \texttt{Gallina}. An ANTLR-based parser then builds \texttt{cGMM} terms from those commands. Type errors are handled by means of custom \texttt{TypeException} exceptions.

Figure 3 shows the commands involved in the case study of the previous section, being processed by a console application which directly accesses the parser. The \texttt{Assume} and \texttt{Declare} commands are used for declarations, the \texttt{Define} command for definitions, and \texttt{Check} for retrieving type information. Declared elements are explicitly typed, however, the system checks that the provided types are well-formed. In turn, terms in definitions are checked for well-formedness, and their types are completely inferred by the type system. Such is the case of composite transformation \texttt{AutoLayout} and terminal model \texttt{finalCD}. We checked their type after their definition. Additionally, in the script of Fig. 3 we intentionally introduced two common error situations. First, we applied \texttt{AutoLayout} to an argument of the wrong type. The error message indicates the received type and the expected type. Second, we applied \texttt{sampleCD} which is not a transformation. The error message indicates that applied element must be executable.

The AM3 tool [15] is a set of Eclipse plugins that realize the GMM approach. Plugin architecture mimics the structure of GMM's metamodel: a core plugin and extension plugins for specific domains. AM3 provides a Megamodeling perspective which in turn provides a generic megamodel navigator and editors. Our implementation of \texttt{cGMM} is being integrated with AM3 as another extension plugin behind the \texttt{ITypeSystem} interface. When a type-related event occurs, AM3 issues an appropriate command to the type system and further exchange type information as required by the case. Such a loosely coupled integration simplifies testing and type system evolution, and enables the substitution of the type system as well as its reuse in other contexts.
Models are not the only way in which artifacts are represented in development processes. From an MDE technical space perspective, external representations of artifacts are in some cases not only needed but also required. Bridging the MDE technical space and other foreign technical spaces enables transference of artifacts among spaces and reuse of facilities available across them. One important technical space is Grammarware, where grammars are defined. As very often DSLs have textual concrete syntaxes, textual representations of artifacts are a common situation even in MDE-based projects. GMM supports both textual entities and even a concrete technology for realizing model-to-text and text-to-model transformations. Typing in GMM is critical for preventing a form of execution errors: type errors. In this paper we extended our typing approach introduced in [5] for typing textual entities and M2T/T2M transformations in a megamodeling context. To that end, we introduced new types, a type constructor for generating types for textual entities and a particular usage of dependent products for projectors. This was formalized in cGMM by means of a set of new type rules, which were incorporated in the implementation of cGMM. As an interesting extension, we even typed text-to-text programs, which are not currently supported in GMM, for enabling the definition of composite transformations that may represent end-to-end transformation problems.

Our typing approach may be understood as a bridge between the MDE and TT technical spaces. Terms in our cGMM calculus within an environment are external counterparts of elements within a megamodel. The results of typechecking terms in the external environment are projected back to the megamodel. We plan to further investigate this view of our work. In turn, scripts such as that in the previous section could lead to a textual language for expressing programs which manipulate megamodels.

One of our main directions of future work is to complete the integration of our implementation of cGMM with AM3. The modular architecture defined for this integration enables the reuse of the type system component in other contexts. We are currently planning the integration of cGMM with Wires* [16]. That tool provides a graphical executable language for the orchestration of complex ATL transformation chains. Although the composite transformations supported by cGMM are conceptually simpler, unlike cGMM, Wires* does not currently typecheck the defined compositions.
References


