Verification of Megamodel Manipulations Involving Weaving Models

Andrés Vignaga and María Cecilia Bastarrica

MaTE, Department of Computer Science, Universidad de Chile
{avignaga,cecilia}@dcc.uchile.cl

Abstract. Global Model Management (GMM) provides a framework for managing large sets of interrelated heterogeneous and complex MDE artifacts. Megamodels are a special kind of model introduced by GMM for containing MDE artifacts. Managing artifacts then involves the manipulation of the megamodel that contains them. Such manipulations can be regarded as programs on megamodels. A precise static typing approach enables the prevention of type errors during the execution of such programs and contributes to their verification. Weaving models express relationships between models. Their role in MDE is becoming more important because of the increasing number of powerful applications they enable. In this paper we show that a metamodel-based typing approach for weaving models may lead to situations were further execution is unsafe, and how a new type defined for them solves the problem.

1 Introduction

Model-Driven Engineering (MDE) mainly suggests basing the software development and maintenance processes on models and chains of model transformations. A few MDE artifacts (e.g., models, metamodels, transformations) can be easily managed, but when industrial use cases are tackled, large sets of interrelated heterogeneous and complex artifacts become unmanageable. Global Model Management (GMM) [3] is a solution for coping with more complex cases by modeling in the large. There, a *megamodel* [2] is a model storing references to models and relationships between them. As a megamodel may refer to any kind of MDE artifacts, it may be logically seen as an environment where elements, as in a programming language, can be either declared or defined. Declared elements are existing models (implicitly initialized variables) and externally defined transformations (operations). Other elements may be defined within a megamodel, and they are internally defined composite transformations (operations) and models resulting from the application of operations (explicitly initialized variables). A series of such constructs can be understood as a program whose execution manipulates the contents of a megamodel.

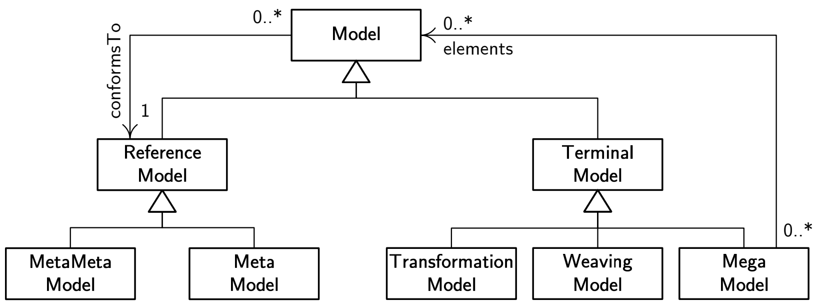
Executability, in this context, introduces the notion of an execution error within a megamodel. One form of execution error is a type error. A definition of type error depends on a specific language, but always includes the application of a function on arguments for which it was not defined, and the attempted application of a non-function [18]. Therefore, typing is critical in GMM. A type system addresses a set of type errors $\mathcal{E}_{\mathcal{T}}$ and may be used for determining well typing. A well typed program, with respect to a consistent type system, then exhibits good behavior (i.e., does not cause type errors from $\mathcal{E}_{\mathcal{T}}$ upon execution, and thus program behavior is not unpredictable). In a concrete project, a megamodel would refer to all involved artifacts, and the manipulation of such a megamodel would encompass the progress of project-specific processes. For this reason, a megamodel-centric tool, such as AM3 [1], should prevent the occurrence of execution errors in general, and type errors in particular, when its underlying megamodel is manipulated. We addressed the problem of static typing in GMM in [16]. Static checking is a form of program verification [4]. Since megamodels are themselves models, and megamodel manipulations are regarded as programs, our typing approach contributes to verification in MDE.

GMM's original typing approach was informally based on a form of "type by metamodel" relation. In [16] we showed that in some non-trivial cases such an approach enabled the loss of sensitive type information preventing any form of further typechecks. Our new typing approach introduced other types and replaced the original has-type relation for some specific kinds of models, most notably, (higher-order) transformation models. This approach prevents the loss of type information and provides a formal means to reason about the typing of core GMM elements within a megamodel. However, we realized that when the type of a model is related to other types the original has-type relation enables checks that result weaker than what is desirable, suggesting that our approach needed further improvements. In [15] we successfully experimented with this idea by introducing a new type for ATL libraries. A related case is the more complex case of weaving models, for example enacted by AMW [5] models. A weaving model captures fine-grained relationships between elements of distinct models in the form of links. It conforms to a metamodel that specifies the semantics of such links. In turn, macromodels are a proposal for model management similar to megamodels. [12] stresses the importance of defining relationships among models within a macromodel. Such relationships may be represented in a megamodel by means of weaving models, and thus the same argument applies to their importance within GMM. In this paper we extend our typing approach for GMM by defining a specific type for weaving models which enables stronger checks in the context of megamodel manipulations.

The rest of this work is organized as follows. Section 2 describes the GMM approach, introduces megamodel manipulations, and discusses our current typing approach. Section 3 addresses the typing of weaving models. It describes the current limitations and introduces a new type for them. The implementation of this new type within our type system is addressed in Sect. 4. Section 5 concludes.

2 Megamodeling and Typing in GMM

In this section we summarize the key concepts of GMM, especially megamodels, and we discuss their manipulation and typing.



context MetaMetaModel **inv:** self.conformsTo = **self**
context MetaModel **inv:** **self**.conformsTo.oclIsKindOf(MetaMetaModel)
context TerminalModel **inv:** **self**.conformsTo.oclIsKindOf(MetaModel)

Fig. 1. Core and model management-specific GMM concepts

2.1 Global Model Management Approach

The purpose of GMM is to provide a framework where large sets of interrelated heterogeneous and complex MDE artifacts can be appropriately managed. Such artifacts include models, metamodels, transformations, any concrete variant of them, and even other domain specific artifacts. It provides a metamodel [8] which specifies all kinds of artifacts that are to be managed. Such a metamodel is incrementally organized into a core part, specifying general MDE concepts or artifacts, and a number of domain specific extensions which either introduce new concepts or refine existing ones. This extensibility enables GMM to be applied to different MDE approaches, which usually introduce the following three different kinds of models (their relationships are shown in Fig. 1):

- *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.

One novel element introduced by GMM is the notion of a megamodel. A megamodel is a variant of the notion of model, specific to the domain of model management. A megamodel is a container of other models and relationships among them. Transformation models and weaving models are also variants of the notion of terminal model. For example the GMM4ATL extension introduces the notion of ATLModel as a specialization of TransformationModel. In turn, the GMM4AMW extension introduces de notion of AMWModel as a specialization of WeavingModel.

One obvious application of megamodels is storing (references to) any model or artifact involved in a project. Provided that megamodel manipulation facilities are available, MDE artifacts exhibiting the characteristics mentioned at the beginning of this section can be properly managed. Naturally, these ideas are meant to be realized by a tool for enjoying such benefits in a practical context.

2.2 Megamodel Manipulation

For fulfilling its purpose, a megamodel needs to be properly manipulated. Basic actions on megamodels are CRUD operations with respect to the elements they contain. Elements within a megamodel may be either externally or internally defined. Externally defined elements (e.g., metamodels, terminal models, transformation models and so on) are treated as black boxes, and their creation within a megamodel is actually a *declaration.* Internally defined elements are compositions of other contained transformations and elements obtained from applying transformations to other contained elements. Their creation is actually a *definition.* All elements are implicitly or explicitly initialized, either by their external or internal definition.

Transformation models are executable elements and they are considered as operations. They can only be applied to some other elements already contained in a megamodel, and their results are automatically included in the megamodel. Manipulating a megamodel is then like programming, where the megamodel acts as an environment [9] which is updated with each new declaration or definition.

2.3 Typing in GMM

We regard the declaration and definition of elements within a megamodel as statements of a model-based programming language. The execution of a simple program composed of a sequence of such statements then manipulates the contents of a megamodel. We formalized such programming language by means of the cGMM calculus [16]. cGMM is a predicative dependently typed λ-calculus based on Constructive Type Theory, similar to a subset of the Predicative Calculus of (Co)Inductive Constructions (pCIC) [10, 17]. We map GMM constructs to cGMM terms which are based on standard concepts such as variables, dependent products and applications. We also define an environment where declarations and definitions (involving typed terms) are included. In this way, a program on a megamodel can be expressed as terms of cGMM. Then a type system for cGMM formally defines the notion of typing in GMM and provides a precise means for reasoning about types.

Typing in cGMM is based on the original has-type relation considered in GMM. Such relation, denoted as $:_{c2}$, is based on the directed association from Model to ReferenceModel in Fig. 1; $m :_{c2} M$ (model m is typed by reference model M) iff model m conforms to reference model M. We use $:_{c2}$ for typing reference models and terminal models which are not transformation models. Models in general are variables, and reference models in particular are variables which may occur at either side of a ':' operator (for this reason we do not make a syntactical distinction between terms and types). However, since this type operation accepts only reference models at its right side, typing by metamodel is not always appropriate, and some elements are not typed according to $:_{c2}$. This is the case of transformation models, which are typed by (possibly dependent) products. A non-dependent product represents classical function types, where dependent products represent either dependent function types (for handling transformations whose type depends on a metamodel) and universal quantifications (for handling higher-order transformations). We realized that the type of other terminal models refer to other types as well, and we investigated the applicability of new type constructors for those cases. We experimented with the simple case of ATL libraries [15] and we concluded that a new type enables stronger checks. In the next section we analogously proceed with the more complex case of weaving models, whose types refer to other types as well.

3 Typing Weaving Models

3.1 Model Weaving

Model weaving enables the definition of user-defined relationships between models. More specifically, a relationship is realized by a set of links which connect model elements contained in the models involved in the model weaving. Furthermore, such set of links is the contents of a model, a weaving model. For example, assuming that we want to define a model weaving between two models m_a and m_b, we define a weaving model m_w whose contents may be $\{<a_1, b_1>, <a_2, b_1>, \ldots\}$, where $<a_i, b_j>$ denotes that a_i and b_j are linked, and a_i and b_j are model elements such that $a_i \in m_a$ and $b_j \in m_b$. Note that model weavings may be n-ary in general. A model weaving is essentially a mapping, and as such, there is a number of useful applications [12]. In particular we could mention transformation specification and tracing. Tracing models are a particular usage of weaving models. In that case, a link between two elements means that one of them was created (during the execution of a transformation) from the other. As another and more concrete example, a model weaving could be defined between the models that represent the implementation and the deployment views of the same system. Then, a component may be linked to a node, and the meaning of that link is that the component will be deployed on the node. The conclusion is that links may have different meanings, which directly depend on the purpose of the model weaving in which they participate. Being a terminal model, a weaving model needs to conform to a (weaving) metamodel. Such a metamodel, in turn, needs to define the semantics of links. As a consequence, a separate weaving metamodel is required for each kind of relationship one wishes to define.

AMW [5] is one possible realization of the notion of model weaving. It proposes a concrete approach to deal with the problem of multiple weaving metamodels. Such metamodels exhibit among them more commonalities than differences. Therefore, AMW defines a *core weaving metamodel* that factorizes those commonalities, from which several specific extensions may be produced. In particular, an abstract notion of link is defined, which is expected to be specialized as needed by the weaving metamodel extensions.

3.2 Example

An interesting case study of model weaving applicability is the model adaptation problem presented in [6]. When a metamodel M_a evolves into a metamodel

M_b, the concern is to adapt any terminal model m_a conforming to M_a to the new metamodel version M_b. The proposed solution is a three-step adaptation. First, a matching process computes the equivalences and changes between M_a and M_b. Second, an adaptation transformation is derived from those discovered equivalences and changes. Finally, such transformation from m_a produces m_b, which conforms to M_b.

Equivalences and changes between metamodels are expressed by means of a weaving model conforming to the *Match* weaving metamodel extension. *Match* introduces different variants of links for indicating those model elements that are present in both metamodels, and those which were added or deleted from M_b with respect to M_a. Note that in this case woven models are not terminal models; they are metamodels instead.

cGMM terms for the matching process (first step) and its application to concrete metamodels reveal how GMM currently handles the typing of weaving models. Assuming we are working within the EMF technical space (i.e., the metametamodel is *ECore*), the matching process is performed by an ATL transformation which can be declared as:

$Matching : ECore \times ECore \rightarrow Match$

This transformation accepts two metamodels, both of them conforming to *ECore*, and produces a weaving model conforming to *Match*. Then considering two different versions of Petri Nets metamodels *PetriNetV1* and *PetriNetV2*, it is possible to define a weaving model based on an application of *Matching* as follows:

$match := (Matching\ (PetriNetV1, PetriNetV2))$

A query on the environment for the type of this generated weaving model returns $match : Match$. Note that the type is plainly the *Match* metamodel. This is because weaving models are currently typed using $:_{c2}$. Next we discuss the consequences of this result.

3.3 A Type for Weaving Models

A terminal model (except an ATL library or any variant of transformation model), is typed using $:_{c2}$ by the metamodel it conforms to. A weaving model conforms to a given weaving metamodel. This is true both in the general case where a stand-alone weaving metamodel is defined, or in the particular case of AMW where the weaving metamodel is actually an extension of AMW's core weaving metamodel. In either case the metamodel a weaving model conforms to is a weaving metamodel, and as a result our discussion applies to AMW, but also for the general case.

If weaving model m_w conforms to weaving metamodel M_w, then we can say that $m_w :_{c2} M_w$. However, m_w links elements within some woven models. For simplicity let us assume that these models are just m_a and m_b, and that they are typed as $m_a{:}M_a$ and $m_b{:}M_b$. We can further assume another weaving model m'_w which conforms to M_w as well, but linking elements within $m_c{:}M_c$ and $m_d{:}M_d$. At this point, by applying the default typing approach, we would have that $m_w{:}M_w$ and $m'_w{:}M_w$. In other words, both weaving models would appear to have the same type. Although they are similar, both weaving models are not exactly of the same nature. But most importantly to us, sensitive type information is not captured.

As an example, let us resume the the case of model adaptation from the previous section. The second step consists in the generation of the adaptation transformation that performs the third step. This second step is carried out by a higher-order transformation (HOT), which we call *AdaptationGeneration*, accepting a matching model and producing the expected adaptation transformation. The problem is that we do not have sufficient information for properly declaring *AdaptationGeneration*. Such a declaration must have the following form (parentheses at the target are not necessary but included for clarity reasons):

$$\mathit{AdaptationGeneration} : \mathit{Match} \rightarrow (\mathbf{?} \rightarrow \mathbf{?})$$

We are unable to express the type of the generated transformation, even if we know that its source and target are the metamodels woven by the matching model. A direct consequence of this is that a generated adaptation transformation remains untyped. This prevents any type check on further applications, and as such, using it will not be safe.

For overcoming this problem we incorporate type information about the woven models to the type of a weaving model. We do this by introducing a *Weaving* type constructor of the form: $[\![\mathit{Weaving}\ A\ B]\!]$, where A is a weaving metamodel (extension) and B is a cartesian product of reference models. The type system only requires a type rule which enforces these constraints. In this way now we would have $m_w{:}[\![\mathit{Weaving}\ M_w\ M_a\ M_b]\!]$ and $m'_w{:}[\![\mathit{Weaving}\ M_w\ M_c\ M_d]\!]$ (in this syntax we omit the product operator). This type information reveals that both weaving models are similar, since they both conform to the M_w weaving metamodel, but they are not of the same type. Furthermore, with this extended type information we can now properly type all steps of the adaptation case study. First we need to declare again the transformation of the first step, now using a dependent product as a dependent function type:

$$\mathit{Matching} : A{:}\mathit{ECore} \times B{:}\mathit{ECore} \rightarrow [\![\mathit{Weaving\ Match}\ A\ B]\!]$$

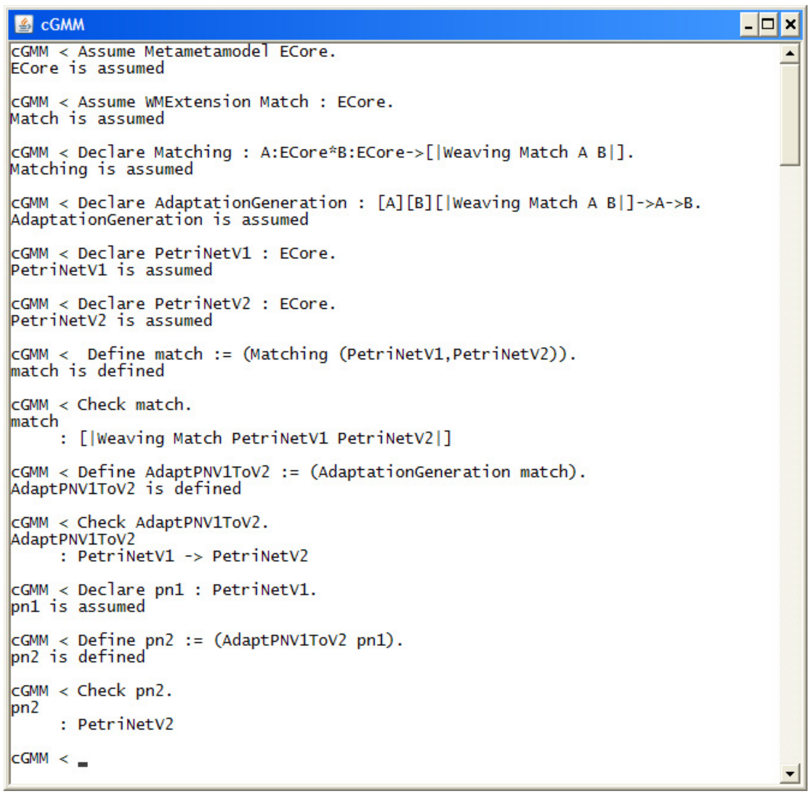
Now we can declare the transformation of the second step, in this case using a dependent product for universal quantifications:

$$\mathit{AdaptationGeneration} : \forall A{,}B{:}\,\mathit{Type}.[\![\mathit{Weaving\ Match}\ A\ B]\!] \rightarrow (\mathrm{A} \rightarrow \mathrm{B})$$

Then we define *match* as before. Note that a query for the type of *match* now returns $\mathit{match} : [\![\mathit{Weaving\ Match\ PetriNetV1\ PetriNetV2}]\!]$. Finally we can produce the adaptation transformation through the following definition:

$$\mathit{AdaptPNV1ToV2} := (\mathit{AdaptationGeneration\ PetriNetV1\ PetriNetV2\ match})$$

Metamodels *PetriNetV1* and *PetriNetV2* are used in the application above for instantiating quantified variables A and B respectively. A query on the environment for the type of the generated adaptation transformation then returns $\mathit{AdaptPNV1ToV2} : \mathit{PetriNetV1} \rightarrow \mathit{PetriNetV2}$ as expected.



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

cGMM < Assume WMExtension Match : ECore.
Match is assumed

cGMM < Declare Matching : A:ECore*B:ECore->[|Weaving Match A B|].
Matching is assumed

cGMM < Declare AdaptationGeneration : [A][B][|Weaving Match A B|]->A->B.
AdaptationGeneration is assumed

cGMM < Declare PetriNetV1 : ECore.
PetriNetV1 is assumed

cGMM < Declare PetriNetV2 : ECore.
PetriNetV2 is assumed

cGMM <  Define match := (Matching (PetriNetV1,PetriNetV2)).
match is defined

cGMM < Check match.
match
     : [|Weaving Match PetriNetV1 PetriNetV2|]

cGMM < Define AdaptPNV1ToV2 := (AdaptationGeneration match).
AdaptPNV1ToV2 is defined

cGMM < Check AdaptPNV1ToV2.
AdaptPNV1ToV2
     : PetriNetV1 -> PetriNetV2

cGMM < Declare pn1 : PetriNetV1.
pn1 is assumed

cGMM < Define pn2 := (AdaptPNV1ToV2 pn1).
pn2 is defined

cGMM < Check pn2.
pn2
     : PetriNetV2

cGMM < _
```

Fig. 2. The complete adaptation problem in the implementation of cGMM

4 Implementation Remarks

We developed the type system for cGMM as a separate component. It provides an environment which can be updated with new declarations and definitions. It supports cGMM terms that correspond to all elements defined in the core GMM metamodel and its main 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 *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 terms, we developed a simple textual language which is similar to *Gallina*, the specification language of Coq [14]. 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 2 shows the commands involved in the case study of the previous section, being processed by a console application which directly accesses the parser. The `Assume` and `Declare` commands are used for declarations, the `Define` command for definitions, and `Check` for retrieving type information. Note that declared elements are explicitly typed, while the type of defined elements is completely inferred by the type system. In addition, the arguments for the application of *AdaptationGeneration* (within the definition of *AdaptPNV1ToV2*) to be used for properly instantiating quantified variables A and B were not used; they were inferred by the type system as well.

The impact on the implementation of the type system of adding the new type for weaving models was moderate. Naturally we needed to define WeavingType as a new variant of Type. Such type refers to a product of types restricted to reference models, and to a new variant of Metamodel, the WeavingMetamodel. Type substitution rules and algorithms for unification (inference of implicit arguments) and for type matching were also required.

The AM3 tool [1] 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 cGMM is being integrated with AM3 as another extension plugin behind the *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.

5 Conclusions and Further Work

Weaving models enable a number of powerful applications, and their typing is of particular interest in our context. In this paper we showed that a simple metamodel-based typing approach for weaving models may provoke the loss of type information which unavoidably makes any further execution of specific transformations unsafe. A new type constructor specifically defined for typing weaving models solved the problem.

Based on this experience, we can now proceed with other GMM elements which present similar cases. In particular, textual entities, which are not models, are currently untyped in GMM. A proper type constructor for textual entities would enable an improved typing for model-to-text and text-to-model transformations supporting concrete realizations, such as TCS [13], which are common MDE artifacts. In turn, our type system does not support subtyping yet. Even though our approach can handle the cases when a weaving metamodel is either self-contained or it is an extension of another, subtyping would enable the explicit representation of AMW's core weaving metamodel.

One of our main directions of future work is to fully integrate our implementation of cGMM with AM3. The modular architecture defined for this integration enables the reuse of the type system component. In that context, we are planning another integration with Wires* [11]. That tool provides a graphical executable language for the orchestration of complex ATL transformations chains, but does not currently typecheck the defined compositions.

References

1. AM3 Project. Internet: `http://www.eclipse.org/gmt/am3/`, 2009.
2. M. Barbero, F. Jouault, and J. Bézivin. Model Driven Management of Complex Systems: Implementing the Macroscope's Vision. In *15th ECBS'08*, pages 277–286. IEEE, 2008.
3. J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. Modeling in the Large and Modeling in the Small. In A. R. Uwe Aßmann, Mehmet Aksit, editor, *In proceedings of the European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004*, LNCS 3599. Springer Verlag, 2005.
4. R. Cartwright and M. Felleisen. Program Verification Through Soft Typing. *ACM Computing Surveys*, 28(2):349–351, 1996.
5. M. Didonet Del Fabro, J. Bézivin, F. Jouault, E. Breton, and G. Gueltas. AMW: A Generic Model Weaver. In S. Gérard, J.-M. Favre, P.-A. Muller, and X. Blanc, editors, *Proceedings of the 1ères Journées sur l'Ingénierie Dirigée par les Modèles*, pages 105–114. CEA LIST, June 2005.
6. K. Garcés, F. Jouault, P. Cointe, and J. Bézivin. Managing Model Adaptation by Precise Detection of Metamodel Changes. In R. F. Paige, A. Hartman, and A. Rensink, editors, *ECMDA-FA*, volume 5562 of *Lecture Notes in Computer Science*, pages 34–49. Springer, 2009.
7. F. Jouault, editor. *1st International Workshop on Model Transformation with ATL, MtATL 2009, Nantes, France, July 8-9, 2009, Proceedings*, CEUR Workshop Proceedings. CEUR-WS.org, To appear.
8. ModelPlex Project. Deliverable D2.1.b: "Model Management Supporting Tool". Internet: `https://www.modelplex.org//index.php?option=com_remository&Itemid=0&func=startdown&id=183`, March 2009.
9. H. R. Nielson and F. Nielson. *Semantics with Applications: An Appetizer.* Springer, March 2007.
10. C. Paulin-Mohring. *Le Système Coq.* Thèse d'habilitation, ENS Lyon, 1997.
11. J. E. Rivera, D. Ruiz-González, F. López-Romero, and A. Vallecillo. Orchestrating ATL Model Transformations. In Jouault [7].
12. R. Salay, J. Mylopoulos, and S. M. Easterbrook. Using Macromodels to Manage Collections of Related Models. In P. van Eck, J. Gordijn, and R. Wieringa, editors, *CAiSE*, volume 5565 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 2009.
13. TCS Project. Internet: `http://www.eclipse.org/gmt/tcs/`, 2009.
14. The Coq Proof Assistant Reference Manual. Internet: `http://coq.inria.fr/doc-eng.html`. Version 8.2, 2009.
15. A. Vignaga and M. C. Bastarrica. Typing ATL Models in Global Model Management. In Jouault [7].
16. A. Vignaga, F. Jouault, M. C. Bastarrica, and H. Brunelière. Typing in Model Management. In R. F. Paige, editor, *Second International Conference on Model Transformation. Theory and Practice of Model Transformations*, volume 5563 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 2009.
17. B. Werner. *Une Théorie des Constructions Inductives.* Thèse de doctorat, Université Paris 7, 1994.
18. A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Inf. Comput.*, 115(1):38–94, 1994.