Model Typing for Improving Reuse in Model-Driven Engineering

Jim Steel and Jean-Marc Jézéquel

Irisa (INRIA & University of Rennes)
Campus Universitaire de Beaulieu
35042 Rennes CEDEX, France

Abstract. Where object-oriented languages deal with objects as described by classes, model-driven development uses models, as graphs of interconnected objects, described by metamodels. A number of new languages have been and continue to be developed for this model-based paradigm, both for model transformation and for general programming using models. Many of these use single-object approaches to typing, derived from solutions found in object-oriented systems, while others use metamodels as model types, but without a clear notion of polymorphism. Both of these approaches lead to brittle and overly restrictive reuse characteristics. In this paper we propose a simple extension to object-oriented typing to better cater for a model-oriented context, including a simple strategy for typing models as a collection of interconnected objects. Using a simple example we show how this extended approach permits more flexible reuse, while preserving type safety.

1 Introduction

From the perspective of the data structures involved, model-driven computing can be seen as a progression from object-oriented computing. Models are, in essence, composed of objects linked together using first-class bidirectional relationships, where the structure of the objects and the relationships between them are typically defined by a MOF, or MOF-like, metamodel. The presence of these relationships has the effect that model structures are much more tightly coupled than object structures.

For this reason, it is hardly surprising that the majority of approaches to developing languages for manipulating models have adopted formalisms based on those found in object-oriented programming languages.

The study of languages for manipulating these model structures is active. In 2001, the OMG issued an RFP soliciting languages for defining model transformations, as mappings between models. In response, many languages have been developed, using variously logic-based[10], pattern-based [13], and graph-transformation [14] approaches. Concurrently, a number of efforts are being undertaken to develop or extend programming languages to better deal with models as data structures [2,17].

The vast majority of these efforts have chosen to use type systems developed for use within object-oriented development. However, as discussed in [8] and mentioned in [16], the use of such type systems in a model-oriented context renders programs somewhat brittle with respect to changes in the metamodel.

Most importantly, however, is that these systems do not truly allow the user to specify their transformations or programs in terms of models and types of models, but rather in terms of objects within models. This is counter-intuitive to the user.

To resolve this, we discuss necessary extensions to object-oriented typing to deal with the relationships defined in MOF metamodels. Using this extended notion of object typing, we propose a definition of a model type, including a definition of substitutability of model types and a discussion of reflection and inference of model types.

In section 2, we provide a background on typing and models and the role of typing in model-driven engineering, including a motivating example. Following this, in section 3 we present a definition of model types with a rule for model type substitutability, including an illustration using the example. Section 4 discusses the implication of this definition for the related issues of model type reflection and model type inference. Section 5 discusses a number of related works from the domains of both MDE and type systems.

2 Background

Generally speaking, a type can be understood as a set of values on which a related set of operations can be performed successfully. Once types have been defined, it is possible to use them in operation specifications of the form: if some input of type X is given, then the output will have type Y. Type safety is the guarantee that no run-time error will result from the application of the operation to the wrong object or value. A type system is a set of rules for checking type safety (a process usually called type checking since it is often required that enough information about the typing assumptions has been given explicitly by the designer or programmer, so that type checking becomes mostly a large bookkeeping process).

Type checking is said to be static when it is performed without program execution (typically at compile-time or bind-time). It aims at ensuring once and for all that there is no possibility of interaction errors (of the kind addressed by the type system). Not all errors can be addressed by type systems, especially since one usually requires that type checking is easy; e.g., with static type checking it is difficult to rule out in advance all risks of division-by-zero errors.

Type systems allow checking substitutability when services are combined: by comparing the data types in a service interface, and the data types desired by its caller, one can predict whether an interaction error is possible (e.g. producing a run-time error such as "Method not understood"). Conformance is generally defined as the weakest (i.e., least restrictive) substitutability relation that guarantees type safety. Necessary conditions (applying recursively) are that a caller must not invoke any operation not supported by the service, and the service must not return any exception not handled by the caller. Conformance has a property called contravariance: the types of the input parameters of a service must conform in opposite to the types of its result parameters.

2.1 Example

We consider as a motivating example a simple model transformation that takes as input a state machine and produces a lookup table showing the correspondence between the current state, an arriving event, and the resultant state. The input metamodel for this transformation is presented in figure 1. The output metamodel, not shown, can be assumed to be a simple database language, but in any case we will focus on the conformance of the input type.

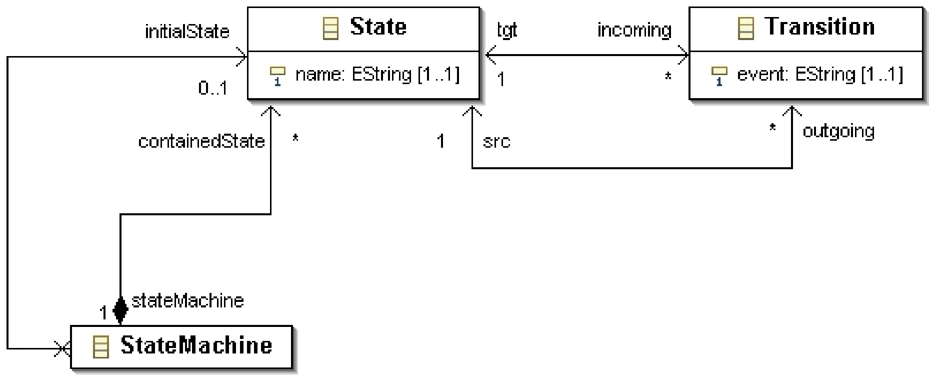

Fig. 1. Simple State Machine Metamodel

The choice of which language is used to implement the transformation, and even of which paradigm of language to use, is immaterial. Also immaterial is the choice as to whether the input and output types of the transformation are derived (inferred) or explicitly declared. (This choice is discussed further in section 4).

Having given this metamodel as the nominal input for the transformation, we consider that there are a number of variants of state machines whose instances might also be interesting as potential inputs to the transformation.

Initially, we might consider changing to the multiplicity of the "initial" reference from 0..1 to 0..*, for state machines with multiple start states (Figure 2), or from 0..1 to 1..1, mandating that each state machine have exactly one start state (Figure 3). Alternatively, we might apply the composite pattern by adding an inheritance of State by StateMachine, for composite state machines (Figure 4). Finally, we might consider the addition of a FinalState class as a new subclass of State (Figure 5).

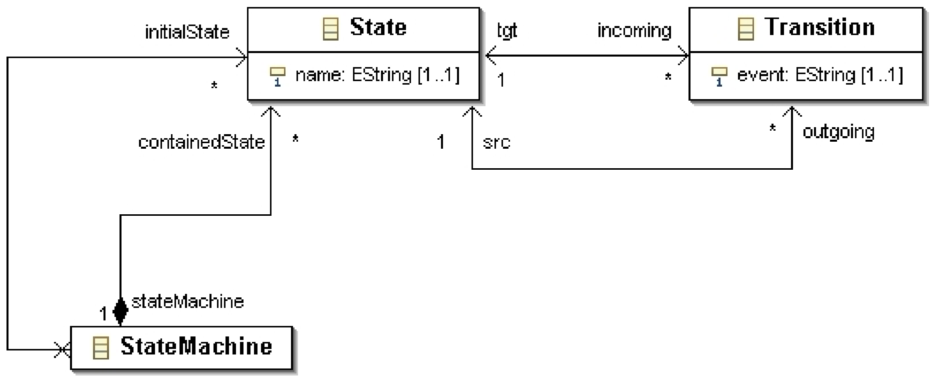

Fig. 2. State Machine Metamodel with multiple start states

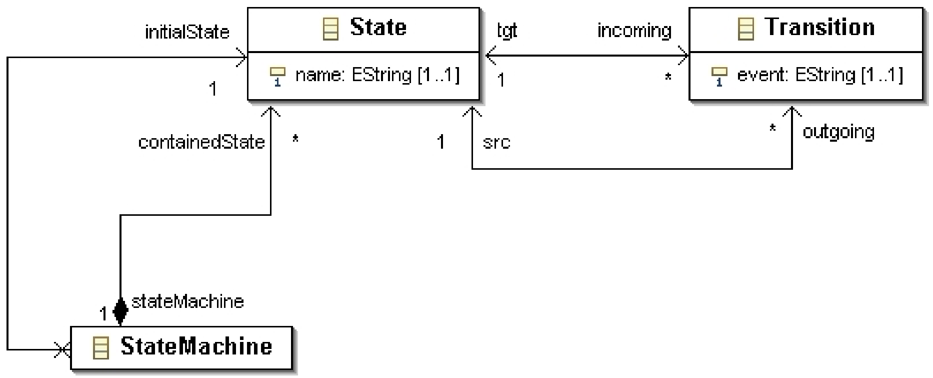

Fig. 3. State Machine Metamodel with mandatory start states

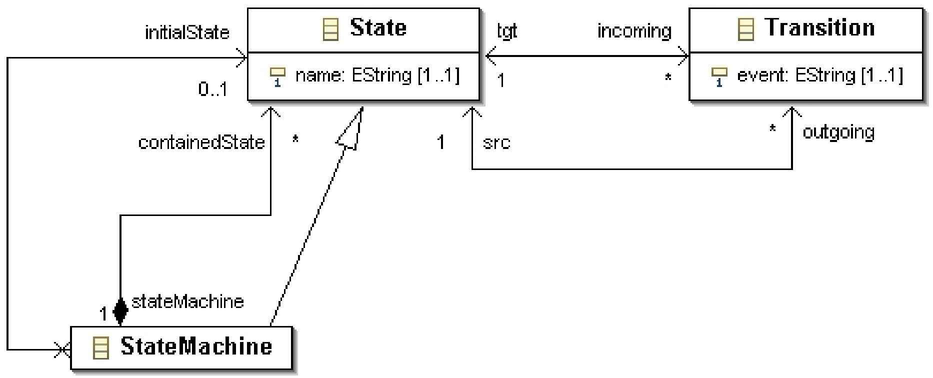

Fig. 4. Composite State Machine Metamodel

The question is, then, does the initial transformation written for models conforming to Figure 1 still work with models conforming to these variant metamodels?

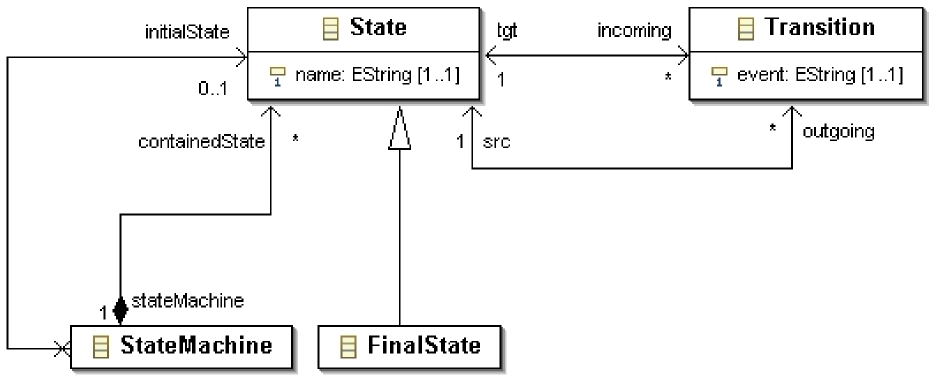

Fig. 5. With Final States

2.2 Objects, And Their Types

Although research is ongoing into the fine details, the basic notions of objects and the type systems that describe them are by now reasonably well-understood [1]. As mentioned briefly above, the main difference between the objects seen in classical object-oriented systems and the objects used within models is the presence of (potentially) bidirectional relationships.

In MOF 1.x, these were defined as binary associations, which in turn contained association ends, which defined characteristics such as the upper and lower bounds, uniqueness and orderedness of the association in a given direction. Navigabilities were specified by the addition of references.

In MOF 2.0, relationships are defined as a pair of references, each of which defines the details formerly kept by association ends. These references may link to another reference, thus forming a bidirectional relationship. This change entails a subtle change of expressivity but, in effect, yields the same type of relationships.

2.3 Models And Metamodels

The MOF specifications, unlike those of UML, have never included a formal definition of either a model or a metamodel. By convention, and intuitively, the latter has usually been used as a synonym for a MOF package. In many MOF 1.x implementations, a model was defined as a “package instance”, a term not defined in the specifications, but an intuitive concept that could contain objects instantiated from any class within a given MOF package. While intuitive, these definitions were somewhat limiting for situations where cross-"model" references were common.

MOF 2.0 has introduced the notion of an extent, and made explicit the fact that extents may contain objects instantiated from classes from different packages. This recognises the increasing abundance of models which reference other models; these are intuitively, and may now be considered as, single models. However, this leaves us without a firm idea of a metamodel, since we can no longer be guaranteed that all objects within an extent will possess a type contained by a single package.

Beyond these conventions, there are two general approaches to defining a concept of a model. The first, that taken by UML, is to designate some class as being a root node for the model, whereby the model thus consists of instances of that class and all objects contained by (or perhaps reachable from) that object. However, this does not work in the case of models which lack a single root element, as is common in cases such as models containing tags or models of, for example, collaborative processes[12]. The alternative and more general approach, the one evident as Extent in MOF 2.0, is to define a model as just a set of objects.

Taking this second definition, the obvious choice for the type of a model is the set of the object types of all the contained objects. The details of such a definition are given in the next section.

2.4 Typing in Model-Driven Engineering

The application of typing in model-driven engineering is seen at a number of levels.

At a fine-grained level, languages that manipulate and explore models need to be able to reason about the types of the objects and properties that they are regarding within the models. For this level of granularity, an object-based approach to typing is probably more natural and appropriate.

From a broader perspective, there is also a need to reason about the types of artifacts handled by the transformations, programs, repositories and other model-related services. It is at this level that an appropriate type system should allow us to reason about the construction of coherent systems from the services available to us. While it is possible to define the models handled by these services in terms of the types of the objects that they accept, we argue that this is not a natural approach, since these services intuitively accept models as input, and not objects.

Having established that services might accept and produce models, it follows that they should specify a type for these models. Furthermore, having established these type declarations, it is also useful to find a semantic for substitutability that allows the maximum possible flexibility and reuse, while still assuring that the services do not receive models whose elements they do not understand.

For example, the sample transformation described in section 2.1 can be said to accept state machines as input, and should accept as many of the noted variants as possible, provided that at no point the transformation attempts an action on the model that is not possible.

3 Model Types and Model Type Substitutability

In this section we provide a simple structure for the type of a model and discuss the conditions under which one model type may be substituted for another. This includes an analysis of the dependence of model typing upon object typing, and the extensions necessary for object typing to function correctly in this new context. We demonstrate the application of model types using the example presented earlier.

3.1 Model Types

The previous section defines a model type as the set of object types for all the objects contained in a model. However, this is a definition based on reflection, and the aim of model types is rather targeted at transformation or model-based programming languages, where reflection will not be the dominant manner of determining types. Therefore, we need to redefine our model type more basically.

So what structures do we have? Normal MOF reflection upon an object yields a MOF class. While literature on type systems, such as [11], suggests that a type is not the same thing as a class, MOF is something of a special case. Since MOF is a signature language, i.e. unable to specify behaviour, a MOF class is in fact more analogous to a type than to a class in type system terminology. We therefore content ourselves to define a model type as a set of MOF classes (and, of course, the references that they contain).

In the example presented in section 2.1, the model type required for our transformation is in essence the metamodel shown in figure 1. In fact, the only significant difference between model types and metamodels is the structuring provided by packages and relationships between packages.

3.2 Model-Type Checking

Under what conditions may one model type, i.e. set of object types, be considered conformant, or substitutable, for another? Quite simply, each object type in the required set must be "understood" by the candidate set. Clearly, this returns to a situation of object type conformance.

Object-Type Conformance The presence of relationships, in whichever form, defined between classes has little effect on the overall approach on the typing of objects. The structure of an object type remains the same. Indeed, if one considers a relationship as a mutually dependent pair of references, they do not differ fundamentallly from the properties seen commonly in object-oriented systems. There is, of course, a stronger prevalence of cyclic dependencies between the conformance of classes. For example, consider a class C1 in a relationship A1, consisting of two references R1 and R2, with another class C2. For a class C1' to be considered a subtype of C1, it must participate in a relationship A1' with a class C2' that is a subtype of C2, which fact depends on the original comparison of C1' and C1.

As has been already presented in [15], there are many possible approaches to the type conformance of object types in a model-based context, ranging from the currently-predominant approach based on subclass-based conformance, as in Java, to structural conformance. As is also presented in [15], these must be extended to ensure that the covariance (for operation return types and property types) and contravariance (for parameter types) rules take into account the structural differences such as multiplicities on associations (using subsumption and inverse subsumption respectively).

However, to use these approaches when typing a model as a collection of objects, there are additional requirements that must be met which do not appear when evaluating object type conformance in isolation. Specifically, there are certain rules pertaining to the preservation of the identity of classes when assessing conformance of relationships. These rules are most evident in resultant axioms such as:

1. A reflexive relationship may not be satisfied by a non-reflexive relationship.
2. Similarly, a non-reflexive relationship may not be satisfied by a reflexive relationship.

Having added this constraint, the choice remains open as to which algorithm one uses when assessing object type conformance. So as to avoid confusing our examples by using metamodel extension techniques, we will proceed using the structural conformance proposed in [15], with appropriate extensions for the above constraint.

Model-Type Conformance Using our object type conformance rule, the question of whether a required model type may be satisfied by a provided model type is determined by checking whether for each required object type, there exists a conformant object type in the provided type.

It is very important to note here the reason for avoiding a reflective definition of model type. In particular, in requiring that each object type exist in the provided model type, we do not mandate that there exist an instance of each object type. Put more simply, there is a difference between the absence of an object of a given type in a model and the absence of the object's type in the model's model type. The model type of a state machine without transitions would still contain the Transition class.

This difference poses no problems if we remain within the domain of transformations or programs with manifest typing, but may become problematic for dynamically determining the type of a model using reflection, or for a language that determines its input or output model types using inference. These problems are discussed in section 4.

3.3 Details and Demonstration

The approach for testing model type conformance discussed above may be summarised by the following steps:

– For each class (object type) in the required model type, find all conformant object types in the provided model type. Using structural conformance from [15], this means that
  - Each operation on the required class must be satisfied by a conformant operation according to covariance on return type, contravariance on parameter types and appropriate subsumptions of multiplicities
  - Each property on the required class must be satisfied by a conformant property according to covariance on type, subsumption on multiplicity, etc

– Eliminate conformant provided types which violate the identity rules described above

– Ensure that for each class in the required model type, there remains at least one conformant type in the provided model type.

More formally, conformance of a provided model type MT_p to a required model type MT_r may be defined as follows.

First, we establish the object type conformance relation $image(MT_r.types \rightarrow MT_p.types)$, such that:

$\forall C_p : Class \in MT_p,\ \forall C_r : Class \in MT_r,$
$C_p \in image(C_r) \Longleftrightarrow$
  $\forall O_r : Operation\ \in C_r.allOperations(),$
  $\exists O_p : Operation\ \in C_p.allOperations() \mid$
    $O_p.name = O_r.name,$ and
    $O_p.type\ \leq O_r.type$ (return type covariance), and
    $O_p.multiplicity \subseteq O_r.multipicity$ (return type multiplicity subsumption), and
    $\forall Pa_r : Parameter \in O_r.ownedParameter,$
    $\exists Pa_p : Parameter \in O_r.ownedParameter\ \mid$
      $Pa_r.type \leq Pa_p.type$ (parameter type contravariance), and
      $Pa_r.multiplicity \subseteq Pa_p.multipicity$ (parameter multiplicity subsumption (inverse))
  $\forall Pr_r : Property \in C_r.allProperties(),$
  $\exists Pr_p : Property \in C_p.allProperties()\ \mid$
    $Pr_p.type \leq Pr_r.type$ (property type covariance), and
    $Pr_p.multiplicity \subseteq Pr_r.multipicity$ (property multiplicity subsumption), and
    $Pr_r.isReadOnly = false \Rightarrow Pr_p.isReadOnly = false$

Having established this relation $image(MT_r.types \rightarrow MT_p.types)$, check the identity constraints described above, by ensuring that:

$\forall C_r \in MT_r,\ image(C_r) \neq \emptyset$ (each required class has a conformant)
$\forall C_r \in MT_r, \forall P_r \in C_r.allProperties(),$
  $P_r.type = C_r \Rightarrow image(P_r.type) = image(C_r),$ (reflexive properties are satisfied by reflexive properties) and

$P_r.type \neq C_r \Rightarrow image(P_r.type) \cap image(C_r) = \emptyset$ (non-reflexive properties are satisfied by non-reflexive properties)

Applying these steps to the example metamodels provided in section 2.1, we obtain the model type conformance relation shown in Table 1.

$\Rsh$ conforms to $\rightarrow$	Simple	Mult-St	Mand-St	Comp	Final
Simple (Figure 1)	YES	YES	NO	NO	YES
Multiple-Start (Figure 2)	NO	YES	NO	NO	NO
Mandatory-Start (Figure 3)	YES	YES	YES	NO	YES
Composite (Figure 4)	YES	YES	NO	YES	YES
With-Final-States (Figure 5)	YES	YES	NO	NO	YES

Table 1. Model Type Conformance Relation for State Machine Variants

In this relation we can see that the addition of new classes (FinalState) and the broadening of multiplicity constraints have not broken the subtyping relationship, but that tightening of multiplicities has. It is notable also that composite state charts are found to be subtypes of simple state charts, although the reverse might have been more intuitive. The effect of using structural conformance is seen by the conformance of simple state charts to those with final states.

4 Further Considerations

Having considered the general idea of types for models and presented an approach for verifying the conformance of model types, we now proceed to discuss two related issues, those of model type reflection and model type inference.

4.1 Model Type Reflection

Reflection is one of the key features of model-driven engineering. The ability to ask an object about what features it provides allows for the creation of generic tools that work regardless of the metamodel from which the object was instantiated. Many services such as XML and textual serialization and deserialization, model repositories, and code generators, already make extensive use of object reflection.

Having added an idea of a model type, it is clearly necessary to consider the problem of model type reflection. That is, if a user provides a model to a service, it should be possible to determine the type of the model by looking at the types of the objects that it contains.

As discussed, the main difficulty with model reflection is the difference between the presence of an object of a given type within a model and the presence of the object type in the model's type. This problem makes it impossible to simply determine (using object reflection) the object type of each member of the model and return that as the type of the model.

Intuitively, the problem requires finding all classes that may be associated with the objects already present in the model. As a general problem, this requires a form of existential quantification, which is something not available in current MDE tools. In lieu of this, an alternative is to use bounded existential quantification, such as searching for all referring object types within a given set of packages, e.g. those already containing object types obtained from object reflection. This is, however, a partial solution that requires further consideration.

4.2 Model Type Inference

A closely related issue to model type reflection is that of model type inference.

In the example transformation presented earlier, we deliberately did not discuss how the input model type was determined, in order to remain independent of the choice of language used for implementing the transformation language. There are two alternatives for determining this type. In manifest typing, as is commonly seen in languages such as Java and C#, for example, types are defined by the user. By contrast, in languages such as ML, types are inferred from the code written by the user.

One can imagine that a similar approach could be used by a model transformation language. A transformation or program whose definition constructs models from a limited set of classes might be able to determine its output model type from the statements creating the objects. Obviously, this has a lot in common with the reflection problem discussed above, and one would imagine that, having determined the classes used in the definition, similar techniques might be used to determine more accurately the complete model type.

While model typing and model type reflection are problems that can be considered largely independent of the choice of model transformation or programming language, model type inference is clearly not. Inference on transformations defined using a rule/pattern-based language such as XMorph[7] will require a different solution to inference on programs defined using a more imperative language such as MTL[17].

5 Related Work

The problem of organising models, transformations, programs and other development artifacts to form coherent model-driven systems is a field just beginning to attract attention. In [5], the authors discuss a model bus, for describing model services and mediating access to them including automation of coercion of models to ensure compatibility. In [3], the idea is presented of a megamodel, a system or registry of models and the relations that exist between them, most significantly those of conformance and representation.

The study of type systems for object systems is a well-researched field. More recently, a number of works have begun to extend this field towards type systems for more tightly coupled systems of objects.

In [4], the authors present an extension to Java to provide for first-class relationships between classes, including a formal definition for the resultant type system. Their proposal includes a notion of relationship subtyping based on set membership, which bears a resemblance to the idea of association subsetting presented in the UML 2.0 Infrastructure.

In [8], the authors present a system for checking the type compatibility of constraints on object models expressed in Alloy, a language similar in purpose to OCL. They propose an algorithm using bounding types and base types to determine whether an expression has meaning with respect to a given object model. Since this approach is based on the UML class diagram metamodel, which bears significant structural similarity to that of MOF, this algorithm would apply straight-forwardly to MOF metamodels.

Both of these works discuss, albeit from different perspectives, the problem of checking the types of objects defined in the context of an object model that is more tightly coupled than those traditionally used in object systems. However, in each case, they consider only the typing of objects, and not of models as a whole. By contrast, the extension presented here attempts, as much as possible, to make orthogonal the question of typing the objects within a system, and to address rather the problem of typing the model as a set of objects. Indeed, our approach depends upon the existence of an object-typing algorithm that is able to handle the presence of first-class relationships, and thus these works can be seen as complementary.

There is also a body of work within the type system community on the grouping of object types. In [9] and [6], the authors propose respectively "family polymorphism" and "type groups". However, although the type structures are similar to those used in our approach, the problem under study in these works is one of object typing, particularly of binary and reflective functions, which, due to the simple type systems currently predominant in model-driven development, does not pertain to our domain. Furthermore, their approach is, once again, designed to aid in the typing of single objects (albeit objects whose types are dependent on other types in a group), and not for typing models as sets of objects. In light of this, these works might also be seen as complementary, if MOF models in fact required the sophisticated checks that they provide.

6 Conclusion

The lack of proper mechanisms for typing operations on models such as model transformations leads to brittle and overly restrictive reuse characteristics. In this paper we have proposed a simple extension to object-oriented typing to better cater for a model-oriented context, including a simple strategy for typing models as a collection of interconnected objects. Using a simple example we have shown how this extended approach permits more flexible reuse of model transformations accross various meta-models, while preserving type safety. We have proposed a simple algorithm for checking the conformance of model types, independently of any given transformation language. A prototype implementation of this algorithm is being implemented on the Eclipse/EMF platform, with the goal of testing its usefulness in several contexts, such as Model-Bus tool interoperability or Q/V/T transformations.

References

1. Martín Abadi and Luca Cardelli. *A Theory of Objects.* Springer, 1996.
2. Mariano Belaunde and Mikael Peltier. From edoc components to ccm components: A precise mapping specification. In *FASE*, pages 143–158, 2002.
3. Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. On the need for megamodels. In *OOPSLA and GPCE Workshop on Best Practices for Model Driven Software Development.*
4. Gavin Bierman and Alisdair Wren. First-class relationships in an object-oriented language. In *Foundations of Object-Oriented Languages (FOOL 2005).*
5. Xavier Blanc, Marie-Pierre Gervais, and Prawee Sriplakich. Model bus : Towards the interoperability of modelling tools. In *Model Driven Architecture: Foundations and Applications (MDAFA 2004).*
6. Kim B. Bruce. Some challenging typing issues in object-oriented languages. *Electr. Notes Theor. Comput. Sci.*, 82(7), 2003.
7. Keith Duddy, Anna Gerber, Michael J. Lawley, Kerry Raymond, and Jim Steel. Declarative transformation for object-oriented models. In P. van Bommel, editor, *Transformation of Knowledge, Information, and Data: Theory and Applications.* Idea Group Publishing, 2004.
8. Jonathan Edwards, Daniel Jackson, and Emina Torlak. A type system for object models. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 189–199. ACM Press, 2004.
9. Erik Ernst. Family polymorphism. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 303–326. Springer-Verlag, 2001.
10. Anna Gerber, Michael J. Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. Transformation: The missing link of MDA. In *Proc. 1st International Conference on Graph Transformation, ICGT'02*, volume 2505 of *Lecture Notes in Computer Science*, pages 90–105. Springer Verlag, 2002.
11. W. LaLonde and John Pugh. Subclassing $\neq$ subtyping $\neq$ is-a. *Journal of Object-Oriented Programming*, 3(5):57–62, January 1991.
12. Object Management Group. Enterprise collaboration architecture (ECA). OMG Document no. formal/2004-02-01, 2004.
13. QVT-Merge Group. Revised submission for MOF 2.0 Query/Views/Transformations RFP. OMG document number ad/2005-03-02, March 2005.
14. Shane Sendall. Combining generative and graph transformation techniques for model transformation: An effective alliance? In *Proceedings of 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, 2003.
15. Jim Steel and Jean-Marc Jézéquel. Typing relationships in MDA. In D.H. Akehurst, editor, *Second European Workshop on Model-Driven Architecture (EWMDA-2)*, 2004.
16. Jim Steel and Michael Lawley. Model-based test driven development of the tefkat model-transformation engine. In *15th International Symposium on Software Reliability Engineering (ISSRE 2004)*, pages 151–160, 2004.

17. Didier Vojtisek and Jean-Marc Jézéquel. MTL and umlaut NG - engine and framework for model transformation. *ERCIM news*, 2004.