On Model Typing

Jim Steel, Jean-Marc Jézéquel

Softw Syst Model
DOI 10.1007/s10270-006-0036-6

REGULAR PAPER

On model typing

Jim Steel · Jean-Marc Jézéquel

Received: 31 January 2006 / Revised: 23 May 2006 / Accepted: 29 May 2006

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. We suggest extensions to existing type system formalisms to support these concepts and their manipulation. Using a simple example we show how this extended approach permits more flexible reuse, while preserving type safety.

Keywords MDA · MOF · Metamodelling · Type systems · Typing · Model transformation

Communicated by Dr. Lionel Briand.

J. Steel (✉) · J.-M. Jézéquel
Irisa (INRIA & University of Rennes),
Campus Universitaire de Beaulieu,
35042 Rennes Cedex, France
e-mail: Jim.Steel@irisa.fr

J.-M. Jézéquel
e-mail: Jezequel@irisa.fr

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.

Given this heritage, 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 [13], pattern-based [17], and graph-transformation [18] approaches. Concurrently, a number of efforts are being undertaken to develop or extend programming languages to better deal with models as data structures [15].

The vast majority of these efforts have chosen to use type systems developed for use within object-oriented development. However, as discussed in [11] and mentioned in [19], the use of such type systems in a model-oriented context renders programs somewhat brittle with respect to changes in the metamodel, often failing in response to changes that ought not to affect their operation.

Most important, 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 Sect. 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 Sect. 3 we present a definition of model types with a rule for model type substitutability, based on intuitive concepts from model-driven engineering and building upon research from object-oriented type systems. In Sect. 4 we show how a language and type system supporting these concepts might be built as an extension of existing formalisms. Section 5 discusses several further issues related to model typing, including reflection and type inference. Section 6 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 (applied 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 vary (as either supertypes or subtypes) in the opposite direction to those 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 Fig. 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.

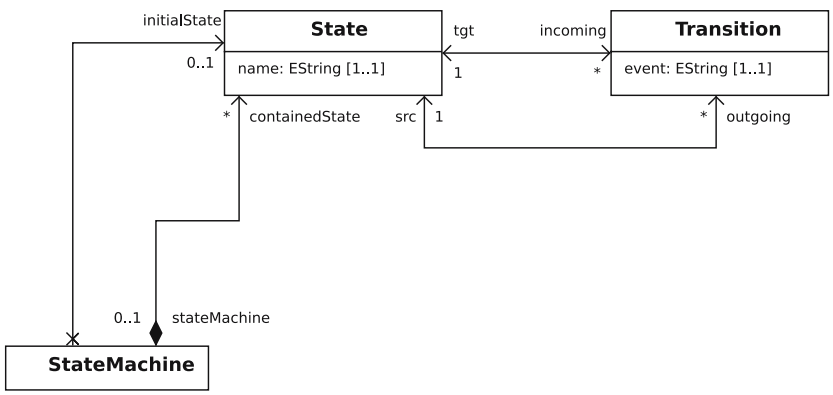

Fig. 1 Simple state machine metamodel

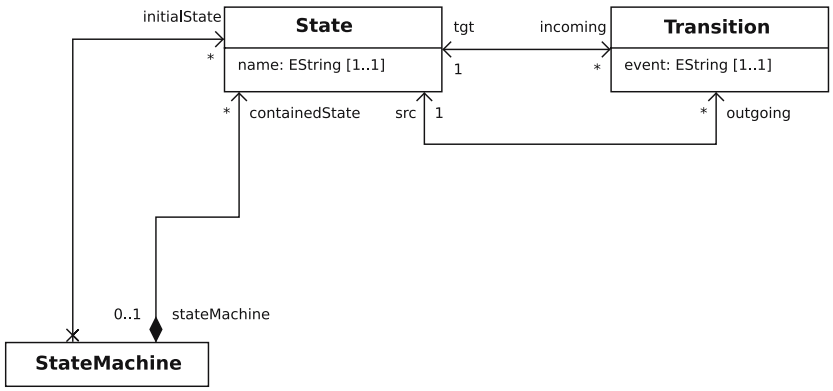

Fig. 2 State machine metamodel with multiple start states

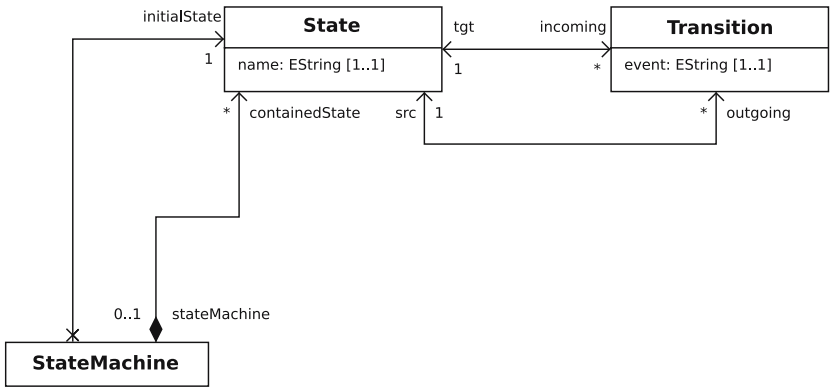

Fig. 3 State machine metamodel with mandatory start states

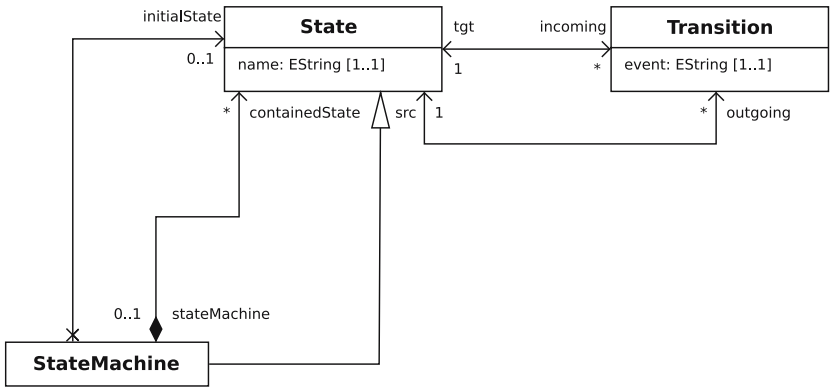

Fig. 4 Composite state machine metamodel

The 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 Sect. 5).

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

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

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

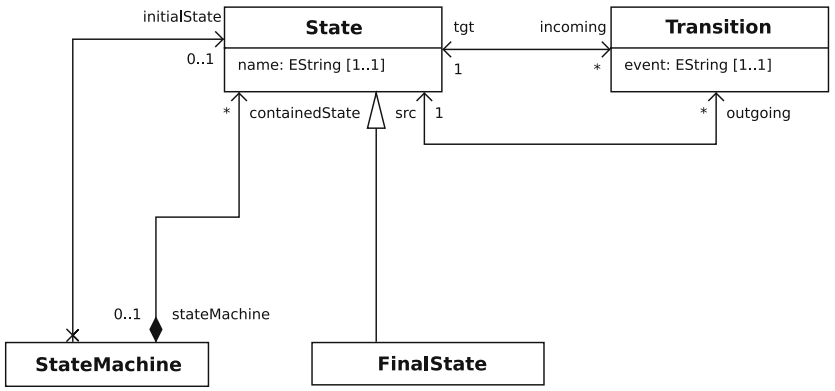

Fig. 5 With final states

[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 relationships were defined as binary associations, which in turn contained association ends, which specified characteristics such as the upper and lower bounds, uniqueness and orderedness of the association in a given direction. Navigability was 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, meaning that the model then consists of an instance of that class and all objects contained by (or perhaps reachable from) this root instance. 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 [16]. 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, one intuitive 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 an architectural 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 Sect. 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 and type checking

The previous section loosely describes 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 [14], suggests that a type is not the same thing as a class, the terminology used by MOF is somewhat misleading. Since MOF is a signature language, i.e., unable to specify behaviour, a MOF class is in fact more analogous to an object 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 Sect. 2.1, the model type required for our transformation is in essence the metamodel shown in Fig. 1. In fact, the only significant difference between model types and metamodels is the structuring provided by packages and relationships between packages.

Having established the structures with which we will type models, the question remains: 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.

3.2 Object-type conformance

As mentioned earlier, type systems for object-based languages are reasonably well-understood, and are increasingly being implemented in the most popular object-oriented programming languages. Typically, the relation used for conformance of one object type to another is subtyping.

Subtyping, as described briefly in Sect. 2, requires that the operations defined on two object types show covariance of their return types and contravariance of their parameter types. If we consider each MOF property to be a pair of accessor/mutator methods this means that subtyping for MOF classes requires invariance of property types.

Unfortunately, one of the strong motivating cases for a polymorphic notion of model types is to allow transformations to keep working as metamodels evolve over time. One of the most common evolutions seen in metamodels is the addition of a property to a class. In this case, any reference to such a class will vary its type. More formally, the addition of the attribute will likely cause a covariant property type redefinition somewhere in the metamodel.

For example, consider a comparison of the basic statechart metamodel in Fig. 1 with the composite statechart metamodel in Fig. 4. As a result of adding the inheritance link, the StateMachine class in the Composite metamodel has evolved to have two more properties: name of type String, and stateMachine, pointing towards a possible containing StateMachine. In isolation, the addition of these attributes might seem to preserve a subtype relationship between the two StateMachine classes. However, this would mean that the property Composite::State.stateMachine represents a covariant redefinition of Basic::State.stateMachine. This is a problem, since property types must be invariant in order to preserve subtyping. Furthermore, the interdependence between the three classes in the metamodels that results from having bidirectional references means that any addition of an attribute would break subtyping for every class in the metamodel.

Nonetheless, from the point of view of a program written to manipulate a Basic state machine, the addition of attributes should make no difference. The lack

of a subtyping (or subsumption) relationship between the classes only poses a problem from the point of view of an individual class. For instance, a composite State cannot be added to a Basic state machine, since an operation on the composite State may attempt to access the "name" or "containingState" property of the Basic state machine, resulting in a type error. However, provided that we specialise the classes in parallel, and ensure that instances of Basic classes and Composite classes do not mix, then there should be no problem.

As it turns out, there is another relationship discussed in type systems for comparison of object types: matching [6]. An object type T' matches another T (denoted $T' <\# T$), iff every method in T also occurs in T' with the same signature. The matching relation is weaker than subtyping; in particular it does not enjoy subsumption, i.e., objects of a matching type are not guaranteed to conform to the matched type. However, as Bruce shows in [6,7] and through the PolyTOIL and LOOM/LOOJ languages, using this relation between groups of types allows for a more flexible, but still statically type-safe, notion of re-use when dealing with the parallel specialization of inter-related object types. This comes with the caveat that matching classes are never used in the context of heterogenous collections. Notably, in the context of models, type-safety depends on models remaining homogenous with respect to a set of object types.

3.3 Changes for MOF object structures

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 fundamentally 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 match of C1, it must participate in a relationship A1′ with a class C2′ that is a match of C2, which fact depends on the original comparison of C1′ and C1.

One of the more significant differences with the object structure of MOF is the presence of multiplicities: upper and lower bounds, uniqueness and orderedness. In order for a MOF property to be considered conformant, not only must its type be a match, but also its multiplicity. For example:

- does a multi-valued property conform to a single-valued property?
- does an optional property conform to a mandatory property?
- does a set-valued property conform to a bag- or sequence-valued property?

In Sect. 4, we present a simplified language which does not consider orderedness or uniqueness, which issues we leave for later resolution, but does provide matching rules based on subsumption of multiplicity bounds.

3.4 Model-type conformance

Bruce further defines in [7] the $<\#$ relation between two type groups as a function of the object types which they contain. This is precisely what we need for determining whether a required model type may be satisfied by a provided model type. Specifically, Bruce states that:

> Type group $TG' <\# TG$ iff for each type MT in TG there is a corresponding type with the same name in TG' such that every method in $TG.MT$ also occurs in $TG'.MT$ with exactly the same signature as in $TG.MT$.

We may generalise this to model types by saying that:

> Model Type $M' <\# M$ iff for each object type C in M there is a corresponding object type with the same name in M' such that every property and operation in $M.C$ also occurs in $M'.C$ with exactly the same signature as in $M.C$.

4 Towards a type system for models

In this section we describe a formalism for reasoning about models and model types. To do this we propose a basic language for defining transformations as series of simple CRUD (Create, Read, Update, and Delete) operations on objects and models. The first section presents a grammar for this language, including a simplified version of the MOF structural concepts and a number of simplified operators for manipulating them. Following that, we describe a number of rules for type-checking a program written using the language. These rules and their explanation rely heavily on the work presented by Bruce and Vanderwaarts in [7].

4.1 Grammar of types and terms

Figure 6 shows the major structural concepts as defined by MOF. A class is defined with a name, a set

$$
\begin{array}{rl}
\mathsf{ClassDecl} ::= & \mathsf{class}\ c\ \mathsf{extends}\ \overline{c'} \\
& \{\ PropDecl^*\ OpDecl^*\ \} \\
\mathsf{PropDecl} ::= & ts\ p\ (\#\ p')? \\
\mathsf{OpDecl} ::= & ts\ o(\ \overline{ts_i\ x_i}) \\
t \in \mathsf{Type} ::= & \mathsf{boolean} \mid c \mid \mathsf{set}\langle c \rangle \\
ts \in \mathsf{TypeSpec} ::= & (\mathsf{optional})?\ t
\end{array}
$$

Fig. 6 Language grammar: MOF structural concepts

$$
\begin{array}{rl}
\mathsf{Trans} ::= & \mathsf{trans}\ \psi\ (\ m\ x\ )\ tb \\
\mathsf{ModelTypeDecl} ::= & \mathsf{modeltype}\ m\ \{\ \overline{c}\ \} \\
m \in \mathsf{ModelType} ::= & m \\
tb \in \mathsf{TransBody} ::= & \{\ \overline{s}\ \mathsf{return}\ e;\ \}
\end{array}
$$

Fig. 7 Language grammar: model types and transformations

of superclasses, and sets of property and operation definitions. Properties have names and types, and may be linked as opposites in order to approximate associations. Operations are named and have typed parameters and a return type. Multiplicities are not present as such, but to capture the important distinctions, we allow properties, parameters and operations to be typed as sets, and to be specified as optional or not.[1]

Transformations in this language take a single model as a parameter and manipulate it in-place. This represents a significant simplification of the approach taken by most transformation languages, notably in that there is no output model, and only one input model. This is done to avoid complications which come about from having multiple model types interacting within a single transformation which, while possible, is less easily understood.

The parameter type of a transformation is a model type, which is a collection of object types. Model types are also valid types elsewhere in the language (variables, expressions, etc.), but in the interests of explanation we will focus on their use as transformation parameter types.

The grammar elements for declaring model types and transformations are shown in Fig. 7.

For the body of transformation, we provide a basic set of types and terms corresponding to a simple expression language (Fig. 8). There are variables, assignments, invocations of operations and transformations, and conditional/iteration statements, etc. The only operator that might be considered unusual is the $\asymp$ operator, which filters a model by a given class to return a list of all objects of that type found within the model.

The filtering of a model to retrieve all instances of a type is an operation that is used frequently in model transformations.[2] However, few existing languages propose it as an operator, instead proposing the functionality through a library function (e.g., *allInstances*() or *all_of_kind*()). Having a clear concept of a model type allows the definition of the operator with a much more accurate type signature.

We do not provide an expression language here for the bodies of operations, but it might be assumed to be the same as that of transformations. We keep them separate only for reasons of explanation.

The signatures that result from a program in this language are shown in Fig. 9. A class definition is a tuple $(\{c_1, \ldots, c_k\}, \mathcal{P}, \mathcal{O})$, where $\{c_1, \ldots, c_k\}$ is a sequence of superclasses, $\mathcal{P}$ is a map of property names to types (with booleans for optionality and multi-valuedness) and opposite properties (in order to form associations), and $\mathcal{O}$ is a map of operation names to definitions. An operation definition is a tuple of the return type and a map of parameter names to types.

A transformation is a tuple $(x, \mathcal{L}, m, tb)$, where x is the parameter, $\mathcal{L}$ is a map from local variable names to their types, m is the parameter's model type, and tb is the transformation body.

4.2 Language semantics

The semantics of the language is largely the same as that presented in [7]. The notable exception is that, in that work, the authors present the semantics of their virtual types as a generalisation of the semantics of their *MyType* operator (i.e., the recursive type), which is absent from the language presented here. While this may seem a gross difference, the semantics and proof of virtual types presented relied upon *MyType* only as an explanatory aid, and its absence does not fundamentally affect the workings of virtual types. As a result, our language here might be seen as supporting mutually recursive types, but not singly recursive types.

Also absent in the language presented here is any discussion of the internal state of objects, i.e., of instance variables. (Instance variables are not to be confused with our properties, which are instead considered to function as pairs of accessor/mutator methods). Once again, for the purposes of defining the semantics, the matter of internal state may be considered as being treated in a similar manner to [7].

Thus, following the semantics of virtual types from LOOM, it can be considered that each of the types used within the body of a transformation in association with the model type is virtual. Thus, the transformation is

[1] This is an equivalent formalism to that used in the ECore modeling language, from the Eclipse Modeling Framework [8].

[2] Indeed, many rule-based languages, such as [9], are built around some sort of filter functionality.

Fig. 8 Language grammar: expressions and statements

$$
\begin{array}{rcll}
v \in \mathsf{Value} & ::= & \mathsf{true} \mid \mathsf{false} \mid \mathsf{null} \mid \mathsf{empty} & \\
l \in \mathsf{LValue} & ::= & x \mid & \textit{variable} \\
 & & e.p & \textit{property access} \\
e \in \mathsf{Expression} & ::= & v \mid & \textit{value} \\
 & & l \mid & \textit{l-value} \\
 & & e_1 == e_2 \mid & \textit{equality test} \\
 & & e \asymp c \mid & \textit{model filter} \\
 & & se & \\
se \in \mathsf{StatementExp} & ::= & \mathsf{new}\ c() \mid & \textit{instantiation} \\
 & & e.o(\overline{e'}) \mid & \textit{operation call} \\
 & & l \mathrel{+}= e \mid & \textit{set addition/association} \\
 & & l \mathrel{-}= e & \textit{set removal/dissociation} \\
s \in \mathsf{Statement} & ::= & ; \mid & \textit{skip} \\
 & & se; \mid & \textit{statement expression} \\
 & & l = e \mid & \textit{assignment} \\
 & & \psi(e) \mid & \textit{transformation invocation} \\
 & & \mathsf{if}\ (e)\ \{\ \overline{s_1}\ \}\ \mathsf{else}\ \{\ \overline{s_1}\ \}; \mid & \textit{conditional} \\
 & & \mathsf{for}\ (c\ x\ :\ e)\ \{\overline{s}\}; & \textit{set iteration}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C} \in \mathsf{ClassTable} & : & \mathsf{ClassName} \rightarrow \overline{\mathsf{ClassName}} \times \mathsf{PropMap} \times \mathsf{OpMap} \\
\mathcal{P} \in \mathsf{PropMap} & : & \mathsf{PropName} \rightarrow \mathsf{boolean} \times \mathsf{boolean} \times \mathsf{Type} \times \mathsf{PropName} \\
\mathcal{O} \in \mathsf{OpMap} & : & \mathsf{OpName} \rightarrow \mathsf{boolean} \times \mathsf{boolean} \times \mathsf{Type} \times \mathsf{ParamMap} \\
\mathcal{R} \in \mathsf{ParamMap} & : & \mathsf{ParamName} \rightarrow \mathsf{boolean} \times \mathsf{boolean} \times \mathsf{Type} \\
\mathcal{T} \in \mathsf{TransMap} & : & \mathsf{TransName} \rightarrow \mathsf{VarName} \times \mathsf{LocalMap} \times \mathsf{ModelType} \times \\
 & & \mathsf{TransBody} \\
\mathcal{L} \in \mathsf{LocalMap} & : & \mathsf{VarName} \rightarrow \mathsf{Type}
\end{array}
$$

Fig. 9 Signatures of class and transformation tables

effectively parameterised by the set of types used within its body in association with its model-typed parameter, i.e., by the set of types listed in the model type.

As in LOOM, the semantics of a transformation body at runtime involve an effective substitution of the model types with the matched types. That is, types within a transformation function as virtual types. For example, the invocation of an operation resolves not to the declared type, but to the actual class provided as a match to the declared type. Similarly for references to properties, and for the creation of new instances from classes.

The major structural addition in the language shown here is the ability to type terms using a model type, thus permitting appropriately-typed variables to function as models (as described in Sect. 2.3). Semantically, these variables then function as collections, whose elements are effectively typed as the union of the types declared in the model type.

4.3 Selected type-checking rules

In this section we present a number of interesting type checking rules that derive from the grammar and semantics of the language. These do not comprise a full type system; they are rather provided to illustrate the extensions that are implied for the extension of an existing type system in order to treat models and model types.

$$
\frac{\begin{array}{c}\mathcal{P}(p) = (n_{mand}, n_{mult}, t, \_) \\ \mathcal{P}(p') = (n'_{mand}, n'_{mult}, t', \_) \\ C, E \vdash t' <\!\!\# \ t \\ n'_{mand}\ or\ \neg\ n_{mand} \\ n'_{mult} = n_{mult}\end{array}}{C, E \vdash p' <\!\!\# \ p} \quad (\textsc{PropMatch})
$$

$$
\frac{\begin{array}{c}\mathcal{R}(r) = (n_{mand}, n_{mult}, t) \\ \mathcal{R}(r') = (n'_{mand}, n'_{mult}, t') \\ C, E \vdash t' <\!\!\# \ t \\ n'_{mand}\ or\ \neg\ n_{mand} \\ n'_{mult} = n_{mult}\end{array}}{C, E \vdash r' <\!\!\# \ r} \quad (\textsc{ParamMatch})
$$

$$
\frac{\begin{array}{c}\mathcal{O}(o) = (n_{mand}, n_{mult}, t, \{r_i\}_{i \le m}) \\ \mathcal{O}(o') = n'_{mand}, n'_{mult}, t', \{r'_i\}_{i \le m}) \\ C, E \vdash t' <\!\!\# \ t \\ n'_{mand}\ or\ \neg\ n_{mand} \\ n'_{mult} = n_{mult} \\ C, E \vdash r_i <\!\!\# \ r'_i \qquad for 1 \le i \le m\end{array}}{C, E \vdash o' <\!\!\# \ o} \quad (\textsc{OpMatch})
$$

$$
\frac{\begin{array}{c}\mathcal{C}(c) = (\_, \{p_i\}_{i \le m}, \{o_i\}_{i \le x}) \\ \mathcal{C}(c') = (\_, \{p'_i\}_{i \le m+n}, \{o'_i\}_{i \le x+y}) \\ C, E \vdash p'_i <\!\!\# \ p_i \qquad for 1 \le i \le m \\ C, E \vdash o'_i <\!\!\# \ o_i \qquad for 1 \le i \le m\end{array}}{C, E \vdash c' <\!\!\# \ c} \quad (\textsc{ObjTypeMatch})
$$

Fig. 10 Selected type-checking rules for model types: object-type matching

The object type matching rule, and in turn the matching rules for properties, operations and parameters, modified to account for multiplicities, are shown in Fig. 10. There are two considerations here. First, collections are treated differently in the language than singletons, since they are subject to set addition and removal operators. As a result, multi-valued properties

$$\frac{C, E \vdash \{c_i\} \quad for\ 1 \le i \le m \qquad C, E \vdash \{c'_j\} \quad for\ 1 \le j \le m+n \qquad C, E \vdash c'_i <\# \ c_i \quad for\ 1 \le i \le m}{C, E \vdash \{c_i\}_{i \le m} <\# \ \{c'_j\}_{j \le m+n}} \ (\textsc{ModelTypeMatch})$$

$$\frac{\mathcal{T}(\psi) = (x, \_, m_1, \_) \qquad C, E \vdash e : m_2 \qquad C \vdash m_2 <\# \ m_1}{C, E \vdash \psi(e)} \ (\textsc{TSTransInv})$$

$$\frac{C, E \vdash x : m \qquad C, E \vdash e : c \qquad C, E \vdash c \in m}{C, E \vdash x \mathrel{+}= e} \ (\textsc{TSModelAdd})$$

$$\frac{C, E \vdash x : m \qquad C, E \vdash c \in m}{C, E \vdash x >< c : set < c >} \ (\textsc{TSModelFilter})$$

Fig. 11 Selected type-checking rules for model types: model types

(or operations, or parameters) cannot conform to single-valued, nor vice-versa. The *mandatory* property, somewhat similar to MOF's lower bounds, obeys subsumption, which in this simplified case is reduced to a *nor* operator.

These rules represent only a small change from those commonly seen in type system definitions in order to support polymorphism (more specifically, match-bounded polymorphism). The notable change is the treatment of multiplicities.

Figure 11 shows a number of rules that have been added in order to treat models and their types.

Matching between two model types is determined by ModelTypeMatch, provided that both are valid model types, and that there exists a pairwise matching of the object types (following the description given in Sect. 3.4.

There are three rules shown for operators dealing with model-typed variables, i.e., models.

TSTransInv checks that the expression used as a parameter to a transformation invocation is model-typed, and that this model type is a match to the declared parameter type.

TSModelAdd permits an element to be added to a model using the += operator, provided that the model's type is a valid extant model type containing the object type of the element to be added.

TSModelFilter ensures that an object type *c* used as a filter on a model *x* is indeed present in the model type *m* of the variable, and that the return type of such an operation is a collection of the filtering object type, i.e., *set* $< c >$.

There are a number of other rules implied by the presence of model types and model-typed expressions in the language, which are not presented here in the interests of brevity. These include basic rules for model type matching, including reflexivity and transitivity, and conformance of all model types to the *Top* model type, {*Object*}. There are also a number of well-formedness rules to ensure, for example, that a model type includes the transitive closure of object types referred to as types of properties, operations or parameters.

4.4 Application to the examples

If we apply the ModelTypeMatch rule to the example metamodels provided in Sect. 2.1, we are able obtain the model type matching relation shown in Table 1.

The relation shows clearly that all of the variants barring those with multiple start states are acceptable for transformations written against a Basic state machine metamodel. We can see that the addition of new classes (FinalState), the tightening of multiplicity constraints (Mandatory), and the addition of new attributes (indirectly with Composite State Charts, via the added inheritance relationship) have not broken model-type matching. However, multiple start states clearly pose a problem should a transformation attempt to navigate the *initialState* property to obtain a single State object.

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. (A simple state chart might be mistaken for a composite state chart that does not use composition.)

In the other sense, basic state charts do not match any of the variants, nor do any of the variants match each

Table 1 Model type conformance relation for state machine variants

↱ matches →	Simple	Multiple-start	Mandatory-start	Composite	With-final-states
Simple (Fig. 1)	Yes	No	No	No	No
Multiple-start (Fig. 2)	No	Yes	No	No	No
Mandatory-start (Fig. 3)	Yes	No	Yes	No	No
Composite (Fig. 4)	Yes	No	No	Yes	No
With-final-states (Fig. 5)	Yes	No	No	No	Yes

other. The effect of insisting on name equivalence when matching object types may be seen in the non-conformance of basic state charts to those with final states; applying a name-independent structural conformance, these model types would be equivalent, and thus would match.

5 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.

5.1 Model topologies

The approach presented above for typing objects within the context of a metamodel is based loosely on a structural, rather than a nominative approach to subtyping. We do, however, require that matching object types in a model type preserve the same name, unlike the structural conformance used, for example, in ML or Ruby.

One of the issues that might arise should one adopt a name-independent structural conformance approach is that model types that match with respect to individual object-type comparisons nonetheless do not resemble one another. For example, one can imagine a single bloated object type providing a match for all object types required in a metamodel, by including the union of properties and operations from all of the required types.

To resolve this, a matching of model types might additional enforce certain rules pertaining to the preservation of the identity of classes when assessing conformance of relationships. These rules might include:

1. A reflexive relationship may not be matched by a non-reflexive relationship.
2. A non-reflexive relationship may not be matched by a reflexive relationship.

One can imagine that a more general solution might lead to some sort of topological analysis of the relationships in a metamodel. In practice, however, the number of properties attached to object types makes relevant applications for such an analysis rare.

5.2 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.

The problem that arises when determining the model type of a model is that there is a need to consider object types that are not instantiated by the model itself. For example, consider a basic state machine that contains a single state but no transitions (uninteresting though such a model might be). By simply taking the object types of all objects in the model, the Transition type would not be included in the model type, making the model ineligible for a transformation written to manipulate basic state machines. There is a difference between the absence of an instance in the model and the absence of the type in the model type.

One solution might be to compute a transitive closure of all types referenced as types of properties, operations or parameters, which would certainly suffice for finding the Transition class for a transitionless state chart. However, there will arise other cases where the navigabilities of the references makes types unreachable by navigation.

As a general problem, this requires a form of existential quantification, which is something not available in current MDE tools. In lieu of this, one alternative would be to use bounded existential quantification, such as searching for all referring types within a given set of packages, e.g., those already containing object types obtained from object reflection.

5.3 Model type inference

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

In the example transformation language presented in Sect. 4, model types for transformation parameters were declared explicitly. There are, however, two alternatives for determining the types of, for example, operation parameters. 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. Similarly, a parameter type might be determined by examining which properties or operations were accessed, or what types were used as filters.

Obviously, this approach, of building a model type based on its usage, 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 likely not. Inference on transformations defined using a rule/pattern-based language such as XMorph [10] will require a different solution to inference on programs defined using a more imperative language such as MTL [15].

6 Related work

The formalism presented above is based heavily on the work done by Bruce et al on type groups, in particular with respect to the type-safe specialization of interrelated types. However, although Bruce introduces the notion of type groups in order to type objects whose types are inter-related, he does not allow terms in his language to be typed by type groups. That is, his language does not introduce the concept of a collection of interrelated objects, or a model.

Although in this paper we have based our approach on type groups, there exist other approaches [5] to the problem of parallel specialization of inter-related object types, including "family polymorphism" [12]. Although family polymorphism could potentially serve as an alternative basis for model typing, its formalism is less developed than that of type groups. Also, like type groups, family polymorphism does not consider the problem of typing graphs or collections of inter-related objects as terms in the language.

In [11], the authors present a system for checking the type compatibility of constraints on object models expressed in Alloy, a language similar in purpose to the combination of OCL and MOF. 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. However, the approach they present does not go so far as to encapsulate models nor their types, and thus does not present any notion of polymorphism, since it relies on an interpretation of the constraints with any new object model.

In [3], 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. Although we have not chosen to model MOF relationships in the same way in this paper, the formalisms they define might present an alternative to the reductionist view we have taken here.

One aspect of MOF that has not been greatly discussed in this paper is the meta-class hierarchy. In [20], the authors present a type system for a metaclass system, as an extension of a Java-like language. An interesting future effort might be to evaluate the possibility of integrating the typing considerations they discuss with MOF's meta-level functionality, as provided by MOF's reflection module.

Having formalised a concept of model and model type, there are a number of domains to which the ideas may be applied. Obviously, having defined a transformation language here, the most obvious choice is to incorporate the ideas into an existing language such as Kermeta [15] or Tefkat [9].

From an architectural point of view, 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 [4], 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 [2], 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. Such approaches would benefit from a type system to govern which models may be associated with which others.

7 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 system for checking the conformance of model types, independently of any given transformation language. A prototype implementation, based on and extending the formalisms presented here, has been integrated into the Kermeta model-oriented programming language/environment, implemented on the Eclipse/EMF platform, presented in this paper. This will help to validate the approach for larger-scale model-driven systems, and to inform the application of model-typing principles to other contexts, such as Model-Bus tool interoperability or Q/V/T transformation languages.

References

1. Abadi, M., Cardelli, L.: A Theory of Objects. Springer, Berlin (1996)
2. Bézivin, J., Jouault, F., Valduriez, P.: On the need for megamodels. In: OOPSLA and GPCE Workshop on Best Practices for Model Driven Software Development (2004)
3. Bierman, G., Wren, A.: First-class relationships in an object-oriented language. In: Foundations of Object-Oriented Languages (FOOL 2005) (2005)
4. Blanc, X.: Gervais, M.P., Sriplakich, P.: Model bus: Towards the interoperability of modelling tools. In: Model Driven Architecture: Foundations and Applications (MDAFA 2004) (2004)
5. Bruce, K.B.: Some challenging typing issues in object-oriented languages. Electr. Notes Theor. Comput. Sci. **82**(7), 1–29 (2003)
6. Bruce, K.B., Schuett, A., van Gent, R., Fiech, A.: Polytoil: a type-safe polymorphic object-oriented language. ACM Trans. Program. Lang. Syst. **25**(2), 225–290 (2003)
7. Bruce, K.B., Vanderwaart, J.: Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. Electr. Notes Theor. Comput. Sci. **20**, 50–75 (1999)
8. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework Eclipse Series. Addison-Wesley, Reading (2003)
9. Duddy, K., Gerber, A., Lawley, M.J., Raymond, K., Steel, J.: Model transformation: A declarative, reusable patterns approach. In: Proceedings of 7th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2003, pp. 174–195, Brisbane, Australia (2003)
10. Duddy, K., Gerber, A., Lawley, M.J., Raymond, K., Steel, J.: Declarative transformation for object-oriented models. In: van Bommel, P. (ed.) Transformation of Knowledge, Information, and Data: Theory and Applications. Idea Group Publishing, Hershey, PA (2004)
11. Edwards, J., Jackson, D., Torlak, E.: A type system for object models. In: SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT 12th International Symposium on Foundations of Software Engineering pp. 189–199. ACM Press, New York (2004)
12. Ernst, E.: Family polymorphism. In: ECOOP '01: Proceedings of the 15th European Conference on Object-oriented Programming. pp. 303–326. Springer, Heidelberg (2001)
13. Gerber, A., Lawley, M.J., Raymond, K., Steel, J., Wood, A.: Transformation: the missing link of MDA. In: Proceedings of 1st International Conference on Graph Transformation, ICGT'02, vol. 2505 of Lecture Notes in Computer Science, pp. 90–105. Springer, Heidelberg (2002)
14. LaLonde, W., Pugh, J.: Subclassing $\neq$ subtyping $\neq$ is-a. J. Object-Oriented Program. **3**(5), 57–62 (1991)
15. Muller, P.-A., Fleurey, F., Jézéquel, J.-M.: Weaving executability into object-oriented meta-languages. In: Briand, L.C., Williams. C. (eds.) MoDELS, vol. 3713 of Lecture Notes in Computer Science pp. 264–278. Springer Heidelberg (2005)
16. Object Management Group: Enterprise collaboration architecture (ECA). OMG Document no. formal/2004-02-01 (2004)
17. QVT-Merge Group: Revised submission for MOF 2.0 Query/Views/Transformations RFP. OMG document number ad/2005-03-02 (2005)
18. Sendall, S.: 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)
19. Steel, J., Lawley, M.: Model-based test driven development of the tefkat model-transformation engine. In: 15th International Symposium on Software Reliability Engineering (ISSRE 2004) pp. 151–160 (2004)
20. Tobin-Hochstadt, S., Allen, E.: A core calculus of meta-classes. In: Fundamentals of Object-Oriented Languages (FOOL) (2005)

Author's biography

Jim Steel is a PhD student at the INRIA/Irisa laboratory at the university of Rennes, and has been working in the field of model-driven engineering and metamodelling since 1997. He graduated with a B.InfTech(Hons 1) from the University of Queensland in 1999, with his honours thesis dealing with the generation of human-usable textual notations for MOF meta-models, which work later served as the basis of the OMG standard. After graduating, he worked for four years as a research scientist with the Pegamento project at the Distributed Systems Technology Centre (DSTC), in Brisbane, Australia. During this time he participated in numerous standards for model-driven engineering, including HUTN, the UML Profile for EDOC, and MOF2.0 QVT. He also participated in the development of prototypes including the TokTok notation generator and the Tefkat model transformation engine.

Prof. Jean-Marc Jezequel received an engineering degree in Telecommunications from the ENSTB in 1986, and a Ph.D. degree in Computer Science from the University of Rennes, France, in 1989. He first worked in the Telecom industry (at Transpac) before joining the CNRS (Centre National de la Recherche Scientifique) in 1991. Since October 2000, he is a Professor at the University of Rennes, leading an INRIA research team called Triskell. His interests include model-driven software engineering based on object-oriented technologies for telecommunications and distributed systems. He is the author of the books "Object-Oriented Software Engineering with Eiffel" and "Design Patterns and Contracts" (Addison-Wesley 1996 and 1999), and of more than 90 publications in international journals and conferences. He is a member of the steering committee of the MODELS/UML conference series. He also served on the editorial boards of IEEE Transactions on Software Engineering and on the Journal on Software and System Modeling: SoSyM and the Journal of Object Technology: JOT.