N° d'ordre: 3523

THÈSE

présentée

devant l'Université de Rennes 1

pour obtenir

le grade de : DOCTEUR DE L'UNIVERSITÉ DE RENNES 1
Mention INFORMATIQUE

par

James Richard Heron STEEL

Équipe d'accueil : Triskell - IRISA
École Doctorale : Matisse
Composante universitaire : IFSIC

Titre de la thèse :

Typage Des Modèles

à soutenir le 23 avril 2007 devant la commission d'examen

M. :	Thomas	JENSEN	Président
MM. :	Krzysztof	CZARNECKI	Rapporteurs
	Birger	MØLLER-PEDERSEN	
	Bernhard	RUMPE	
MM. :	Robert	FRANCE	Examinateurs
	Jean-Marc	JÉZÉQUEL	

Table des matières

Chapitre 1

Background and Motivation

Model-Driven Engineering (MDE) is an approach to the specification, construction, validation, and maintenance of software systems in which the central concept is that of models. MDE has grown from a number of origins, most notably the object-oriented analysis and design languages and methodologies [Boo94, RBP$^+$91, Jac91] and Computer-Aided Software Engineering (CASE) endeavours in the 80s and 90s to automate software engineering [OFMP$^+$95, BGMT88].

The models which are at the core of MDE are each built with respect to a metamodel, which defines the structure of a language as the concepts and associations which it offers the modeller. Model-driven systems are then assembled from components that store, transform, manipulate and otherwise treat models according to certain metamodels. These components can be built using either generative or generic techniques, or built specifically for a given metamodel.

The integration of two or more of these components therefore depends on a shared understanding of the metamodels being used. Equally, the ability of a given component to function correctly with models whose metamodels are variations on that against which the component was written, depends on having some technique for determining whether the variant metamodel is in some way compatible with the originally specified metamodel. These problems, of ensuring the safe reuse of model-driven components in light of metamodel evolution, and of allowing the safe composition of these elements into systems, must be addressed if MDE is to be a sustainable approach to system development and maintenance.

This general problem space, of ensuring the safe functioning of programs in light of structural variation, has been the subject of considerable study in the field of type systems. Object-oriented systems, in particular, have extensive foundations in formalisms that specify, for example, under what conditions objects of a certain type may be used in a situation where objects of another type are anticipated.

In addition to statically ensuring the absence of certain errors in programs or assembly of systems, types and type systems also provide formal abstractions for structuring systems that are useful throughout the software engineering process. For example, Fleurey, Steel and Baudry show in [FSB04] that a clear and accurate expression of the type of models used by a transformation is useful for guiding the automated generation of test cases when testing it.

In this thesis I show how ideas from type systems may be applied to the problem of model-driven engineering to define a more formal notion of models, model types, and model substitutability. Combined, these elements may be used to provide a sound basis for reuse and integration in model-driven systems.

In this chapter, I give a survey of model-driven engineering, of existing approaches to supporting reuse and integration in model-driven systems, and of work in type systems that addresses related issues. Chapter 2 then shows how ideas from type systems may be adapted and extended to provide a basis for typing models in order to allow the definition of reusable model-driven components. This is illustrated in Chapter 3, which describes the application of these model typing principles to support their use in the Kermeta model-driven engineering language. Chapter 4 then discusses the application of model typing, and its implementation in Kermeta, to the expression problem. Finally, Chapter 5 offers conclusions and perspectives.

1.1 Model-Driven Engineering

The defining characteristic of MDE [BFJ$^+$03] is the use of models for representing the important artefacts in a system, be they requirements, high-level designs, user data structures, views, interoperability interfaces, test cases, or implementation-level artefacts such as pieces of source code. Each of the models in an MDE system is constructed from a language specified using a metamodel, which captures the concepts and relationships of the language in a structured and regular form. Relative to these metamodels, the models can then be stored, manipulated, and transformed to other models, and to implementation artefacts, in order to form coherent systems to solve software problems.

The benefits of building systems in this way stem from two qualities ; the diversity of languages used for modelling pieces of the system, and the uniformity of the meta-language used for the specification of these diverse languages.

The use of metamodels permits each element of the system to be described using a language tailored to the domain concepts of the aspect it describes. This allows for a separation of concerns in which domain experts may express themselves using the terminology that is familiar and appropriate to their expertise. In this way, domain experts may be more directly involved with the specification, construction and maintenance of systems, rather than having to have their knowledge filtered through engineers. In addition, describing the system using more natural concepts leads to more intuitive and understandable system designs.

On the other hand, each of the languages used to build models for the elements of the system is described using a uniform formalism for metamodels. This uniformity permits the extensive use of both *generic* and *generative* engineering [CE00] techniques, which obviates the need to reimplement language features that are common across languages, such as serialization, versioning, transformation and, to a certain extent, editing. Instead, these can be implemented once for all models (generic), or their generation can be either partially or entirely automated (generative).

The Model-Driven Architecture (MDA) trademark [Obj03] as marketed by the Object Management Group (OMG), presents a model-driven approach to system development beginning using high-level, platform-independent models (PIMs), which are incrementally transformed or refined into lower-level, platform-specific models (PSMs), which are in turn reified into implementation artefacts such as program code. This automation of the construction of systems from high-level models allows the relevant software engineering expertise to be captured as reusable model transformations, and applied more reliably and efficiently.

However, in addition to the refinement patterns associated with MDA, MDE systems may work in the other direction, building or recovering high-level models from existing implementation artefacts, as is common in reverse engineering [RD99]. Equally, systems may make use of models at run-time [BBF06], where external interactions are related back to changes in models within the system.

Much of the work done in relation to model-driven engineering has focused on the development phase of the software lifecycle. However, there have also been a number of works on other phases, such as requirements gathering and specification of systems using models [NBKS06, Kle06], and model-based approaches to testing [Rum03, Fle06, SL04]. Model-driven engineering is largely orthogonal to the choice of software process, supporting agile techniques [Rum06] as well as more traditional processes [Kru03].

1.1.1 Modelling Architecture

The most important concepts in MDE are those of models and of metamodels. These terms may be understood with reference to a layered architecture, as shown in Figure 1.1.

The top layer of this architecture is the meta-language. In the context of the OMG's MDA, and in most approaches to MDE, the meta-language is defined by the Meta-Object Facility, or MOF, which was first published by Crawley et al in [CDI+97] and by Baker and Le Goff in [BG97]. The meta-language is the

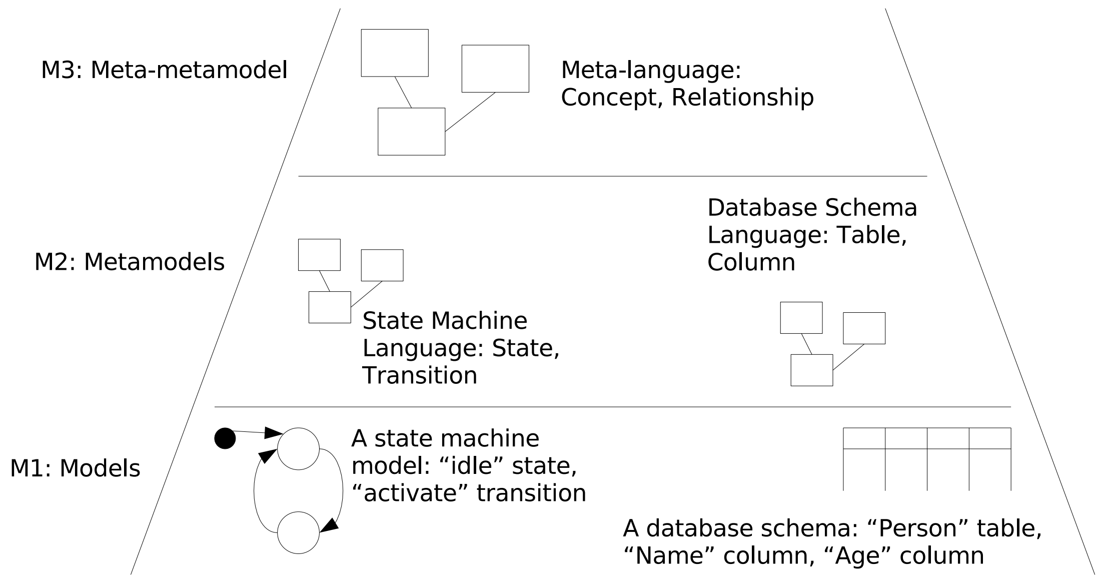

M3: Meta-metamodel
Meta-language: Concept, Relationship
M2: Metamodels
Database Schema Language: Table, Column
State Machine Language: State, Transition
M1: Models
A state machine model: “idle” state, “activate” transition
A database schema: “Person” table, “Name” column, “Age” column

FIG. 1.1 – Layered Architecture : MOF, Metamodels, and Models

language in which the structures of subsequent modelling languages may be defined as metamodels. The meta-language is defined by the meta-metamodel, a canonical set of classes and associations between them, and is the fixed point of the modelling hierarchy, in that it is defined in terms of itself (known as meta-circularity). In addition to defining the way in which metamodels are built, the meta-language details the basic relationship between model-level objects and the metamodel-level concepts that classify them.

Modelling languages, which appear in the M2 level of Figure 1.1, may be specified by instantiating the classes and associations of the meta-language to give a metamodel which defines the concepts and associations of the language. A language for designing state machines, for example, will define concepts such as named states and transitions, which are linked together by relationships ; a transition might have relationships to its source and target states. Of course, this language will add constraints that are specific to its semantics. For example, the state machine language might specify that states have unique names.

Models then reside at the M1 level. For example, in the case of the state machine language, a model will be the definition of a given state machine, for example a machine describing the behaviour of some electronic device. The basic nature of these models as instances of the metamodel that defines them is given by the meta-language, which also provides them with basic facilities such as introspection.

There are a number or relationships that hold between these layers. The view used as the basis for the separation into layers is that of instantiation or classification, in that a metamodel is conceptually an instance of the meta-metamodel, and a model an instance of a metamodel. Alternatively, there is a subtyping view, which works in the other direction. The meta-metamodel is a metamodel, albeit one that has been made canonical. All metamodels are also models. One important relationship not shown in the figure is that of representation. A model is generally a representation of some system or concept, and the metamodel is a representation of the modelling language, specifically of its structure. Representation currently plays little or no formal role in the construction of model-driven systems.

One of the popular starting points for explaining MDE, and particular MDA, is the so-called four-layer architecture, which was initially introduced to explain the relationship of a number of OMG modelling specifications, including MOF and the Unified Modelling Language (UML). This architecture incorporates an additional layer below M1[1] which represents program data. In fact, this extra layer is really a consequence of the fact that the language given at M2 was typically UML. UML is part of a class of languages, other examples being object-oriented programming languages, which themselves include a notion of instantiation, implying the existence of a level below M1. For example, the language for designing UML class diagrams resides at M2. These may be used to construct a class diagram for a workflow language, and this workflow language may be used to build a workflow, the model of which would reside at an M0 level. However, the semantic of being able to instantiate M1 models is not common to all modelling languages, and thus such an M0 level is not implied by model-driven engineering in general. In fact, the mechanisms of the MOF mean it is possible to construct hierarchies of 2, 3, 4 or more layers ; we use three here to illustrate the most common usage.

An important overall consideration of this hierarchy is that it is an aid to understanding, and serves little formal or tangible role in the development of model-driven systems. Model-driven tools typically do not include the notion of meta-levels, although there are exceptions, such as the MetaCASE toolset[TR03]. There are a number of works, such as [FEBF06], [AK03] and [Küh06] which discuss the more philosophical and didactic aspects of modelling and model-driven engineering, including representation relationships, and the nature and number of meta-levels.

1.1.2 The Meta-Object Facility

Since the concepts of model and metamodel are defined relative to the meta-language, the meta-language of a modelling hierarchy is its most important part.

[1]This four-layer architecture is the historic basis for the numbering of the meta-levels. A more conceptually coherent approach would be to consider the meta-language as M0, since this is the fixed point.

By far the most common meta-language used is the Meta-Object Facility, as standardised by the Object Management Group (OMG).

The early MOF specifications, including the ISO-standardised version [Int05], were expressed with a mapping from the concepts and associations used to define metamodels to CORBA interfaces which permitted their use as constructors and classifiers for object representations of models. As a result, the original MOF metamodels made use of CORBA data types for building metamodels. In subsequent revisions, however, the dependence of MOF upon CORBA has been removed.

The basic concepts provided by the MOF for the definition of metamodels are classes and associations. Classes are used to define the concepts of the metamodel, and may include both operations and typed attributes. Classes may also be defined as inheriting from one or more other classes, in which case they inherit the operations and attributes of the superclasses, as in classical object-oriented inheritance.

Associations provide a link between two classes, allowing instances of their objects to be related by links. This is similar to defining attributes on each of the classes and ensuring that they are synchronised. Associations and classes may be organised into packages. Associations may also express composition semantics, whereby objects at a one end of an association are composed by an object at the other end, with the implication of coincident object lifecycle.

Operations, attributes, parameters on operations, and the ends of associations, may be restricted in how objects may participate in the respective role, by using multiplicity constraints. These include upper and lower cardinality bounds, and flags for indicating whether objects participating in the role are unique, or ordered. So, for example, an attribute which has a cardinality range 0..7 and is specified as not unique but ordered, would mean that an object would store a sequence holding between 0 and 7 elements.

As a result of the original link to CORBA, the original MOF specifications relied on CORBA for the data types available as the building blocks for metamodels. Since then, the list of data types has been reduced, and now only String, Boolean, Integer and UnlimitedNatural remain. It is also possible to define enumeration types, as collections of named literals. For example, a *Colour* enumeration might consist of the literals *Red*, *Green* and *Blue*.

In addition to providing constructs such as classes and associations for the definition of metamodels, MOF also describes the relationship between the classes of a metamodel (M2 in the Figure 1.1) and the interconnected objects in a model constructed from that metamodel (layer M1). For the most part, this is simply the definition of Object, and its reflection interfaces for accessing its metaclass, and the corresponding ability of a type to verify that an Object is one of its instances.

1.1.3 Essential MOF (EMOF)

The most recent MOF specification [Obj06a], version 2.0, defines two variants of the MOF metamodel. The Essential MOF (EMOF) presents a relatively small number of concepts corresponding roughly to those found in object-oriented programming languages, in order to support simple mapping to programming language APIs and XML formats. Complete MOF (CMOF) is roughly a superset of EMOF that includes a number of more sophisticated constructions, such as first-class associations with subset redefinition, richer object reflection, and relationships for the redefinition of packages. In this thesis, we will deal with EMOF, in the hope that techniques for managing reuse which work with EMOF may be extended to work with CMOF.

The metamodel of EMOF is shown in Figure 1.2. The most significant simplification with respect to other versions of MOF is that there is no class for defining associations. Instead, the *Property* class has been generalised to encompass both attributes and participation by a class in an association. An association is defined as a pair of properties which reference one another using the *opposite* metaproperty. This has the meaning that the insertion of an object $o1$ into a property on another object $o2$ (in an imperative programming language, for example, this might be $o2.p := o1$) will imply the simultaneous insertion of $o2$ into the opposing property on $o1$ ($o1.q := o2$).

We may instantiate the classes of the EMOF metamodel to create a metamodel for describing a modelling language. Figure 1.3 shows a very simple EMOF metamodel in the form of a class diagram, where each class is an instance of $EMOF :: Class$, each attribute or reference to another class is an instance of $EMOF :: Property$, etc. Since each element of the diagram is an instance of an EMOF class, one can imagine an equivalent representation of the metamodel using an instance-diagram notation.

In the same way that the EMOF metamodel classes may be instantiated to produce the state machine metamodel, the state machine metamodel classes may then be instantiated to produce a specific state machine model. Figure 1.4 shows such a model for a state machine with three states, one of which is an initial state, and three transitions between them.

This state machine may alternatively be shown using the specific syntax of state machines. This diagram, shown in Figure 1.5, presents the same model as Figure 1.4, but using a different concrete syntax.

1.1.4 Models and Metamodels

The EMOF metamodel includes most concepts necessary for defining models and metamodels. However, two elements that are conspicuously lacking are those

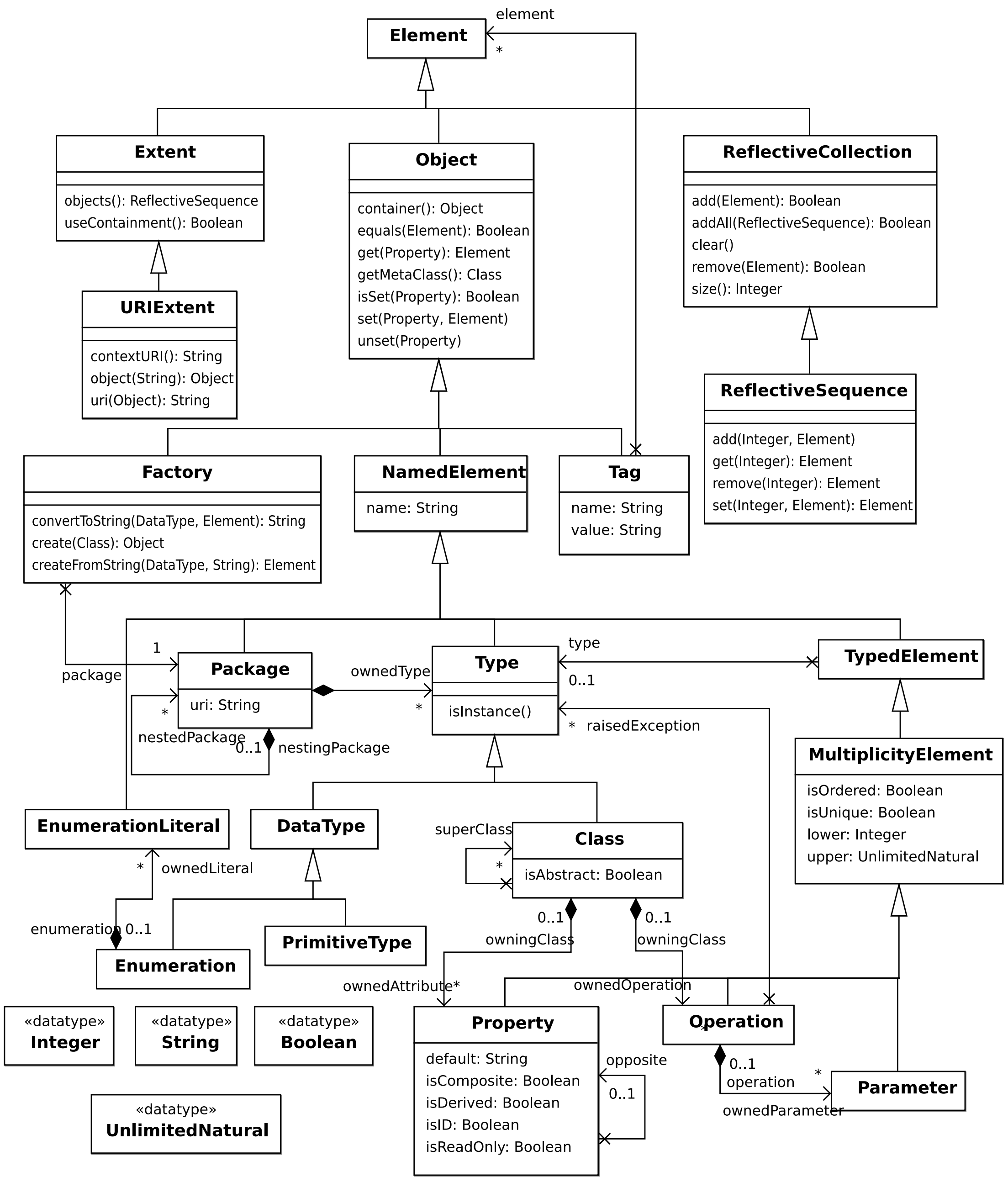


FIG. 1.2 – The Essential MOF metamodel

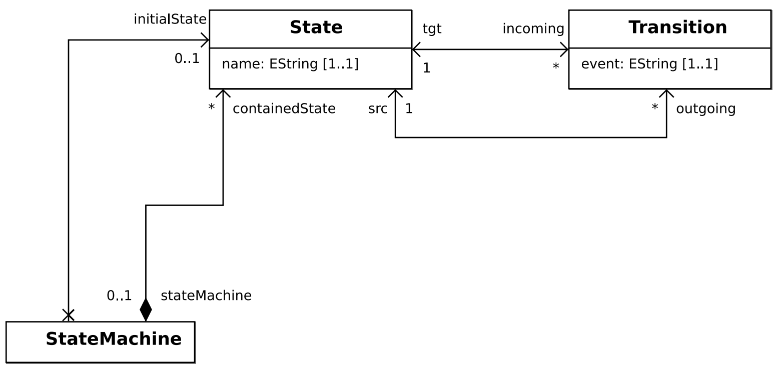


FIG. 1.3 – A State Machine Metamodel

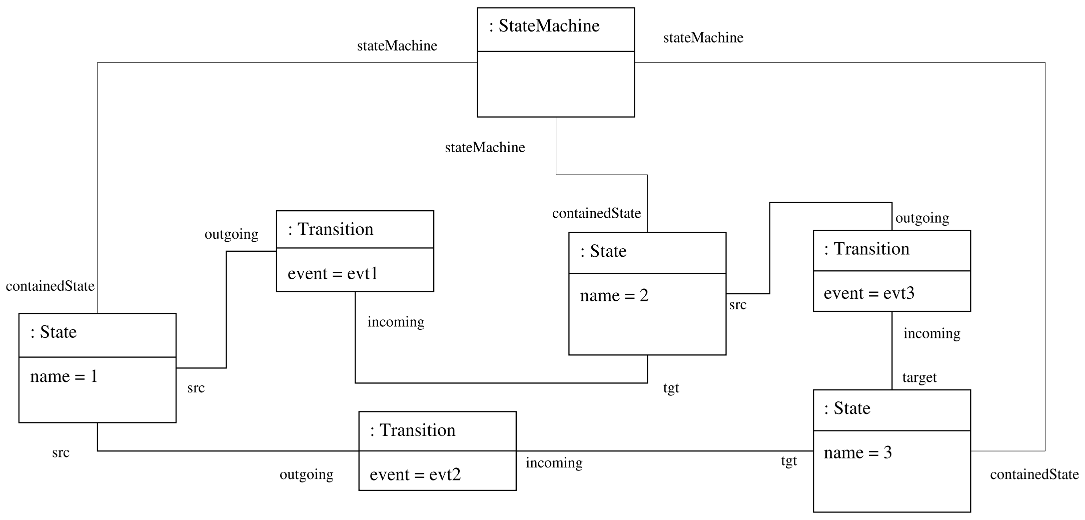


FIG. 1.4 – A State Machine Model

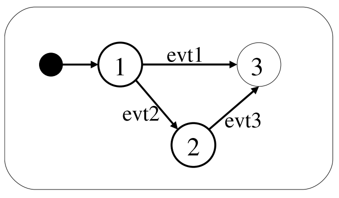


FIG. 1.5 – A State Machine Diagram

of "model" and "metamodel" !

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 introduces the notion of an extent, and makes 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 as a type, since we can no longer be guaranteed that all objects within an extent will possess a type contained by a single package.

From the point of view of the structure of a model, rather than the type of its objects, there are two general approaches to defining a concept of a model.

The first approach, 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 [Obj04a].

The alternative and more general approach, and the one evident as Extent in MOF 2.0, is to define a model as just a set of objects. In some ways, this is a similar approach to that of graph transformation systems (see Section 1.2.2.3 for a summary of graph transformations), in that a model is simply a graph of objects, without regard for the types of the objects and links that are contained.

Having chosen one of these approaches, we may use the description given in Section 1.1.1 to derive a meaning for models. A metamodel is simply a model whose elements are all instances of MOF classes.

So, for a tree- or containment-based approach to defining models, we may take a MOF Class high up in the containment hierarchy, typically *Package*, and allow instances of that class to be definable as metamodels.

Alternatively, in the second, graph-based approach, we may simply say that any model, i.e. set of objects, whose members are all instances of MOF classes, is a metamodel.

1.2 Elements of Model-Driven Engineering

There are three significant classes of elements from which model-driven systems are composed; those that store or mediate access to models, those that transform or manipulate models, and those that convert models into another form.

In this section we will describe a number of significant languages or technologies that are used for the definition and/or implementation of each of these classes of elements. We are particularly interested in generic or generative techniques [CE00], that provide a language designer with the ability to build components either automatically or based on a declarative or high-level description of the necessary function.

This preference is one reason that we do not discuss the construction of model editors, since these are frequently not built from declarative descriptions, but implemented for each metamodel using general-purpose programming languages that manipulate models through programmatic APIs, or through the import or export of serialised forms.

Specific components like this, that are developed for use with a particular metamodel, are no less important to the construction of MDE systems. During the typical lifetime of a language specified using a metamodel, it is quite common to begin with an entirely generic toolset, and to gradually add specific components as the metamodel stabilises. However, they are difficult to enumerate in a general discussion about model-driven system development.

1.2.1 Model stores

As explained in Section 1.1.2, early versions of the MOF specifications were written as mappings to programmatic interfaces. Because of this, the form of models and the manner in which they are stored were often conflated. This has been largely resolved in the MOF 2.0 specifications, which deal separately with concepts being modelled and their concrete representation forms as technology mappings.

There are two main ways in which models are stored and subsequently accessed. The first is through a mapping of metamodels and models to a programmatic interface, such that each may be accessed using existing general-purpose programming language interfaces. The second is through mapping metamodel and models to XML technologies, so that they may be accessed or exchanged as XML documents.

1.2.1.1 Object stores

The MOF 1.x suite of specifications, including the ISO standard [Int05], was defined in terms of a mapping from MOF metamodels to CORBA IDL [Int99] interfaces. Models were then accessed according to the CORBA language mapping preferred by the programmer. Since Java was, at the time, the most popular programming language amongst MDE developers, Java/CORBA was the predominant form for accessing models. Because of this predominance, a more specific mapping was created, called the Java Metadata Interface, or JMI, as a Java Community Process specification [Sun02]. The most significant implementation of JMI is the open-source Metadata Repository (MDR) project[2].

With the advent of MOF 2.0, the form of models has been separated from their mapping to programmatic interfaces. For the time being, the most significant implementation of MOF 2.0 is the Eclipse Modeling Framework (EMF), developed as part of the Eclipse project supported by IBM [BSM$^+$03][3]. Although EMF is built upon a variant of EMOF called ECore, there is a simple mapping between the two, and import and export of EMOF metamodels is supported. EMF provides a mapping from metamodels to Java interfaces, as well as default implementations of these interfaces using an XMI backend.

A number of dynamic object-oriented languages are also the targets for tools providing object-based APIs for accessing models. In [DG06], the authors describe using Smalltalk for model-driven engineering tasks using a formalism that resembles a slimmed-down version of MOF. Similarly, the Coral repository, as described in [AP04], provides a Python API for accessing, modifying and storing models defined using their SMD metamodelling language. RubyMOF[4] is a Ruby tool for loading, storing and manipulating EMOF models, inspired by EMF, which has been used as a testbed for model transformation languages in [CMT06]. In fact, the use of dynamic languages often offers advantages for the implementation of model repositories, particularly for addressing implementation problems such as those related to metacircularity.

1.2.1.2 XML-Based Model Interchange (XMI)

An alternative to storing and accessing models via programmatic interfaces is to use an XML-based format.

The XML-Based Model Interchange (XMI) [Obj05a] specification presents a mapping for models and metamodels to XML documents and schemata/DTDs. Specifically, each metamodel may be mapped to an XML schema (or DTD), and models constructed from the metamodel will be mapped to XML documents that

[2]Metadata Repository Project Home Page : http ://mdr.netbeans.org
[3]Eclipse Modeling Framework : http ://www.eclipse.org/emf
[4]RubyMOF, http ://rmof.rubyforge.net

conform to that schema. Since metamodels are themselves constructed from the MOF metamodel, they may be represented either as a schema, or as an XML document corresponding to the schema generated from the MOF metamodel.

Of course, in the same way that a mapping to a programmatic API permits the use of general-purpose programming languages, the XMI mapping to XML documents permits the use of XML-enabled tools. For example, the EMF tool uses XMI documents as the default persistence mechanism underlying its programmatic API.

1.2.2 Model Transformation and Manipulation Languages

By exposing models and metamodel using mappings to programmatic APIs as described in Section 1.2.1.1, it obvious becomes possible to develop model-driven tools such as model transformers, editors or analysers using general-purpose programming languages.

However, another approach is to build specific languages for the construction of these tools. This second approach has a number of advantages. By using a model-driven approach to these problems, i.e. using a language formalised using a metamodel, an element such as a model transformation becomes itself accessible by the full range of model-driven tools. Also, by narrowing the problem space, it becomes possible to provide languages that are more specific and better adapted to solving the problem.

The most significant example of this has been the definition of model transformations, for which there have been a large number of languages proposed in recent years. In [CH06], Czarnecki and Helsen provide a survey of more than twenty transformation languages, based on a detailed survey of their characteristics and features. This survey shows a remarkably broad range of approaches, varying from relational and logic-based rule languages, to imperative languages resembling programming languages, to template languages and more.

As Tratt highlights in [Tra05], model transformation poses specific problems that warrant specific solutions, in particular for their reuse and integration within and between different tool environments.

In this section, we present a description of just a few model transformation languages[5], as an indication of the role of such languages in model-driven engineering. This is clearly not meant as a canonical list, nor should the selection be interpreted as meaningful in terms of the importance of the languages in current MDE research.

[5]As opposed to general transformation languages ; a discussion of model-to-text transformation languages is given in Section 1.2.3.2.

1.2.2.1 MOF 2.0 Query/Views/Transformations (QVT)

In 2002, the OMG issued a request for proposals for a language for the definition of queries, views, and transformations of MOF 2.0 models [Obj02]. A number of languages were proposed in response to this, focusing on transformations, and encompassing much of the range of approaches to model transformation.

The resultant specification [Obj05b], recommended for adoption in late 2005 and undergoing finalisation at the time of this thesis' writing, essentially presents three languages for the transformation of models. The first is a declarative relational language bearing strong resemblance to Tefkat [LS05] and to a lesser extent graph transformations [KÖ5] and ATL [BBDV03]. The second is an operational mappings language, which provides essentially an imperative manipulation language, as well as a good deal of syntactic convenience for the transformation of models. The third is a small declarative core language, which serves the additional role of providing the definitional basis for the relational language.

The notions of both model and model type, which are key to the contributions presented in this thesis, have in a recent revision been included in the QVT document, and resemble the ideas presented here. However, no basis is given for the substitutability of model types in any of the three QVT languages. From this point of view, the ideas in this thesis may be seen as providing a basis for, amongst other things, the reusability of model transformations as specified by QVT.

At the time of writing of this thesis, there are very few publicly available direct implementations of the QVT specification/s, although a number of implementations have been available for languages which have served as input to the specification process. It remains to be seen to what extent the specification process will serve as an aid to tool interoperability, and to what extent as a forum for the development of model transformation ideas and technology.

1.2.2.2 Tefkat

Tefkat [DGL$^+$03] is a declarative language and engine for model transformations originally developed by the Pegamento team at the Distributed Systems Technology Centre (DSTC), and now maintained as an open-source project[6]. The language, described under the name XMorph by Duddy, Gerber, Lawley, Raymond and Steel in [DGL$^+$03] (and in greater length in [DGL$^+$04]) emerged as the result of a series of experiences, described in [GLR$^+$02], with model transformation approaches based on XSLT, structured code, Prolog rules, and F-Logic rules. As shown by Lawley and Steel in [LS05], Tefkat presents a practical solution to model transformation problems using a rule- and pattern-based approach.

[6]Tefkat : The EMF Transformation Engine. http ://tefkat.sourceforge.net

Tefkat transformations are defined as sets of rules, each of which consist of source model constraints which must be satisfied in order for the rule to be applied, and target model constraints which must be satisfied during the transformation process. These constraints include checking the type of an object, the value/s given by a named property on an object, or the presence of a link between two objects.

Tefkat also makes extensive use of traceability models and/or relationships in order to control the construction and/or modification of target model elements as a function of the source model elements which cause the action. These traceability relationships may be used as a way to imply dependencies between transformation rules, and may also serve as a persistent record of a transformation execution.

Tefkat transformations are defined as working against a number of disjoint input and output extents. These serve only as extents for the matching (source extents) and creation/update (target extents) of model elements. They are not typed in terms of the types of objects which they may contain. Tefkat's configuration model, which provides for the modelling of transformation tasks, i.e. invocations of transformations against specific models which may be executed in series, also uses an untyped notion of models.

In [HLR06], the authors present an approach for live transformations in Tefkat, in which changes to source models and transformation definitions are automatically and incrementally propagated through transformations to changes in target models. This is based on persisting the internal state of the transformation execution and using this state to isolate changes based on metamodel elements. The approach is specific to logic-based transformation engines, as it is based on the modification of SLD trees, the internal representation of logic-based transformation executions.

1.2.2.3 Graph Transformation Languages

Graph transformation [BH02] is a field of study that incorporates a range of transformation languages and tools. The graphs in graph transformations vary, but are generally based on the simple mathematical representation of graphs as sets of nodes and edges. One relatively recent but important approach is the use of typed graphs, in which the structure of a graph may be governed or classified by a homomorphism to a type graph. Type graphs [CMR96] fill a similar role to metamodels in model-driven engineering, and in recent works have come more and more to resemble metamodels structurally.

Although there is variation between the different flavours, graph transformations consist of sets of rules for the transformation of a graph, and often (though not always) directives for the order of application of these rules.

Unlike other declarative model transformation languages such as Tefkat, graph transformation languages permit in-place modification of graphs. This represents a significant increase in expressive power, but requires a syntactic overhead in specifying transformations, such as, in the case of triple-graph grammars, a third graph to distinguish elements that may be deleted from those that may not.

Since source graphs may change during transformation, there are also significant implications for termination of graph transformations. This problem is typically resolved by providing control flow directives for ordering the application of rules within a graph transformation, such as constraints, automata, explicit rule stratification, or simple imperative control constructs.

With the rise of model-driven engineering and its emphasis on model transformation, the structural similarities with graph transformation has lead to extensive cross-pollination of ideas between the two fields. Languages such as Tefkat and Relational QVT bear a strong resemblance to graph transformations, and graph transformation languages have evolved to use structures more closely resembling models, in particular in the structures they use as types for graphs.

Taentzer et al [TEG^{+}05] show the application of four different graph transformation tools - AGG, AToM3, VIATRA2 and VMTS - to a fixed model transformation problem, and shows that their solutions resemble a hypothetical solution using relational QVT. Although the specific technologies for encoding metamodels (type graphs) and models (graphs) are different, the concepts are similar. This is even more evident in [KÖ5], in which the same problem is tackled using a triple graph grammar approach which uses MOF metamodels directly.

1.2.2.4 Kermeta

Kermeta [MFJ05, Fle06] is an executable metamodelling language and tool suite developed by the Triskell Team at the IRISA laboratory. Although there is little continuity of implementation between the two, Kermeta is the ideological successor of MTL [Pol05], as imperative model transformation languages where ' ' transformation' ' is interpreted as including any programmatic manipulation of models.

The language is structured as an extension of EMOF with the added capability of supporting the definition of the semantics of a metamodel. Most significantly, this means the ability to express the behaviour of the operations defined in metamodels, through the use of an action language based on common object-oriented programming principles.

The action language includes late-binding, familiar object-oriented control-flow constructs such as loop and conditional operators, as well as assignment into local variables and properties (both local and non-local). Property assignment also takes into consideration associations, modelled (as in EMF) using opposing properties, in that both ends of the association are affected by a single assignment. Property assignment also considers metamodel constraints such as unique containment and multiplicities.

Kermeta includes one major structural modification to EMOF ; the ability to define classes and operations with type parameters. This is done in such a way as to maintain compatibility with EMOF as much as possible. One motivation for this is for the provision of syntactic closures. Although each is useful in its turn, the combination of type parameters and syntactic closures allows for the definition of collection operators such as OCL's *collect*, *select*, and *reject*, which are particularly useful for meta-programming.

Since it provides sufficient expressive for arbitrary access and manipulation of models, Kermeta may also be used to express model transformations. This is shown in [MFV$^+$05], in which the authors attack set transformation problems using imperative object-oriented languages, including Kermeta. They show that in using these languages, a transformation author may apply many of the software engineering practices that have been proven to work in object-oriented development. In cases of model transformation involving reasonably clear mappings between concepts, this may involve a development overhead compared to declarative approaches, as the programmer must provide techniques for detecting patterns in source models and invoking rules manually. However, in more naturally algorithmic transformations, the imperative paradigm may be more familiar to programmers and less cumbersome than declarative approaches.

1.2.3 Printing and parsing

The third major class of components in model-driven systems includes those that manage the transition between models and other formalisms. For the most part, this means the transition from models to text or vice versa, since even diagrammatic, tabular or other non-textual representations are typically accessible through textual representation. For example, images may be represented using the XML-based format of scalable vector graphics, or graph diagramming tools such as GraphViz[7].

In this section we describe two techniques for managing this model-text transition.

1.2.3.1 Human-Usable Textual Notation

The Human-Usable Textual Notation (HUTN) as presented by Steel and Raymond in [SR01] provides a mapping from metamodels to concrete syntaxes that grew out of a desire for an alternative notation to XMI for representing models, that would be usable for reading and editing by humans. HUTN provides

[7]GraphViz : http ://www.graphviz.org/

a mapping from metamodels to concrete syntaxes that roughly resemble block-structured programming languages. For the most part, the mapping makes no allowances for specific metamodels, producing instead a generic form for any model, although it does provide for small customizations such as the denomination of identifying attributes and default values. The textual language resulting from the HUTN mapping may be implemented for both consumption, e.g. in the form of a parser, or for production, as a pretty-printer.

HUTN was adopted as an OMG specification [Obj04b] in 2004, but to some extent has been deprecated in that it was defined against MOF 1.4, and has not been modified to accommodate MOF 2.0, although the changes simplify, rather than complicate, the mapping. Furthermore, TokTok, the only open implementation of HUTN, has not been updated, and is no longer publicly available.

1.2.3.2 Tools for specific textual syntaxes

There are also a number of works that deal with generic techniques for the generation and loading of text from/to models in a manner specific to a given metamodel. This is clearly useful for languages with a predefined concrete syntax. Also, although it is considered by some to be bad practice, this helps for generating code directly from models without passing via model transformations. The majority of these model-text techniques are designed specifically for the production of text, although a few are designed for both production and parsing. This distinction is aligned to a certain extent to that between approaches based on templates, which are the preferred user paradigm for production of text, and approaches based on grammars, which offer surety that a document may be decidably parsed.

The OMG's Models-to-Text Transformation Language [Obj06b] presents a template-based language for the production of text from models, where elements from the model are shown as holes in the model to be filled by either queries (from QVT) or expressions navigating into the model.

A significant limitation of template-based approaches to text generation is that the models used to drive them may not be used for going in the other direction, i.e. parsing text files to create models. In Sintaks [MFF$^+$06], the authors restrict the operators available in their template language such that each construct has a specific semantic in the context of both textual synthesis and textual analysis. They also suggest a mapping of their syntax specification language to BNF grammars. Anti-Yacc [HRS02], proposed by Hearnden, Raymond and Steel, works in the other direction, by starting with a grammar-based approach and associating synthesis semantics to each element.

1.3 Existing approaches to reuse in MDE

The reusability of MDE components for the purposes of safely supporting evolution, composition, or integration of model-driven engineering components, has been the subject of considerably less study than the languages used for building them. There, are, however, a number of efforts in this area.

1.3.1 Composing Elements of Model-Driven Engineering

Two notable efforts in modelling the ways that models, transformations and other components may be represented within architectures of model-driven systems are Model-Bus and Megamodels.

Model-Bus [BGS04] takes a service bus approach to the integration of model-driven components. Components register services in terms of the models they accept as inputs and produce as outputs. These models may be typed by their physical encoding (such as those shown in Section 1.2.1), as well as a simple notion of model types as sets of metaclasses, but without any notion of model type substitutability or conformance[8].

In [BJV04], Bézivin et al present the notion of megamodels for representing MDE tools and components and the relationships between them. This includes the service-type aspects of model-aware tools, but also incorporates the relationships between models, metamodels, and even meta-metamodels, in order to support interaction with other technical spaces. However, for the most part this is done at a high-level, and without any notions of model types and model type substitutability or conformance.

The ideas in this thesis, of model types and model type matching, can be seen as very much complementary to both of these works, in providing a basis for allowing the connection of components with differing but conformant model types.

1.3.2 Modularization and Reuse in Graph Transformation

There are a number of approaches to modularization in graph transformation systems, many of which are summarised by Heckel, Engels, Ehrig and Taentzer in [HEET99]. These are typically based on the types of graphs as type graphs or schemata used by the transformations, and some among them support ideas of graph type substitutability. At the time, most approaches to modules in graph transformations were concerned more with inheritance-related aspects than substitutability relationships. In [EHC05], Engels, Heckel and Cherchago present a

[8]This notion represents a very early version of the work presented in this thesis, developed during the author's visit to the LIP6 Laboratory in July, 2004.

formalism for graph transformation module interconnection supporting substitutability based on type graph morphisms. Their approach goes further than the comparison of the type graphs of transformations, in that it also compares their transformation rules, suggesting a semantic, as well as syntactic, substitutability.

1.3.3 Alloy

The Alloy modelling system is a constraint-based modelling language that uses similar high-level constructs to the combination of UML class diagrams and OCL constraints. The significant difference, however, is that the structural modelling concepts are mapped to a constraint language, which makes for a smoother integration between the structural and constraint aspects of specification. In [EJT04], the authors present a type system for checking both the safety and relevance of constraints with respect to a structural definition. However, the type system they present does not provide any encapsulated notion of model or of model type.

1.3.4 Constructive type theory and MOF

In [Poe04] and [Poe06], Poernomo presents a mapping from the Meta-Object Facility to the constructs of constructive type theory, using Martin-Löf's predicative type theory. The principle motivation for this is to provide a formal basis for the construction of metamodels and models. However, the use of predicative types also offers the possibility of using this formalism to check the applicability and even correctness of transformations against certain metamodels, where transformations may be analogized as functions in the constructive type theory.

Constructive type theory is more expressive than that used as the basis for object-oriented type systems, and this extra expressive power can pose problems for the decidability of type checking. The few (functional) programming languages [Xi98, Aug99, MM04] that support the dependent types of constructive type theory are experimental and neither intended nor ready for wider consumption.

Although constructive type theory is intuitively an appealing avenue of research for providing a sound basis for model-driven engineering, it remains to be seen if it can be made practical enough for use in tools.

1.4 Type Systems

Type systems are amongst the most popular and successful lightweight formal methods used for verification in software engineering. In [Pie02], Pierce offers the following definition for type systems :

> A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.

Clearly, this definition is broad, and includes fields of study, such as the use of types for session [VVR03] or security [BdGdLJ02] concerns , which are of less interest in attempting to apply type systems ideas to general model-driven engineering. In the last twenty years, one of the most successful branches of type systems study, in terms of transition into industrial software-engineering practice, has been that of object-oriented type systems. Popular object-oriented programming platforms such as Java and .NET now include powerful static type systems supported by object calculi with formal definitions and proofs [IPW01, KP07].

Type systems are most commonly encoded as sets of logical deduction rules over the syntactic elements of the language to be checked. In a static type system, the type-checking process then verifies that a program is consistent according to these rules, using the redundancy included by the programmer in the form of syntactic elements such as type annotations.

As can be seen from their use in industrial programming languages, the foundations of object-oriented type systems are increasingly well understood. Nonetheless, there are a number of areas in which work continues to be done in order to support more expressive type systems capable of detecting errors in more complex systems. Mutually-recursive types represent one such area. Type systems supporting mutually recursive types allow object types to be defined and considered relative to one another, rather than in isolation, as is typically the case in simple object-oriented type systems.

In addition to providing assurance of the absence of type errors, an important characteristic of type systems is that they provide the programmer with mechanisms for abstraction, and secure these abstractions against operations that violate their integrity. In the case of functional programming languages, this is in the form of functions, while in object-oriented languages this is in the form of objects and object types. In applying type systems techniques to model-driven engineering, we wish to support models and model types as abstraction mechanisms.

Since models in model-driven engineering resemble and are indeed based on the objects used in object-oriented systems, it is logical to look to object-oriented type systems when seeking techniques for the elimination of errors within and between model-driven components. In this section we present a very brief review of object-oriented type systems, as well as a survey of approaches to mutually recursive types.

1.4.1 Object-Oriented Type Systems

This section presents a brief summary of object-orientation and object-oriented type systems, covering concepts such as objects, classes and object types, structural and nominal types, subtyping, and generic types. For more on the theory of object-orientation and its application to programming languages, there are a number of excellent books [AC96, Pie02, Bru02, Cas97] which cover this material in much greater detail.

The fundamental concept of object-oriented systems is the *object*. An object is generally an encapsulation of data, in the form of *instance variables*, and of behaviour, in the form of *methods*. The instance variables and methods of an object are characterised by types, which specify what values are acceptable for instance variables or method parameters, and what values might be expected in return from a method invocation.

Objects are often created, or *instantiated*, using *classes*. Classes define the form, i.e. the instance variables and method, of the objects they instantiate, and often provide the definition of the methods of the objects they instantiate. In this way, behaviour is shared between different objects instantiated from the same class. Classes may also be defined in terms of other classes, as *subclasses*, in which case their instantiated objects provide methods not only from the instantiating class, but also from those of which it is a subclass. This is also known as *inheritance*. The exact method which is executed when invoking a method on a class is determined using *dynamic dispatch*.

Objects may be classified, for example when constraining what values are acceptable in an instance variable of an object, using *object types*. Object types resemble classes, in that they specify what instance variables and methods must be present on an object. Indeed, in some languages, such as Java, classes often serve as object types. In different systems, object types may be structural, defined as the set of instance variables and methods they provide, or nominal, based on their name or identity. For example, Java uses nominal types, but the object calculi of Abadi and Cardelli [AC96] use structural types.

One of the more appealing features of object-oriented languages is their support for *subtyping*. One object-type may be considered to be a *subtype* of another, meaning that an object belonging to the subtype may be freely used in the place of an object of the supertype. From a set-theoretic viewpoint, the set of objects representing the subtype is a subset of the set of objects representing the supertype. There are many approaches to subtyping. One important distinction between approaches is between subtyping based on comparison of structural types and subtyping based on subclasses, such as in Java. The relationship between inheritance and subtyping varies between languages ; in Java, inheritance implies, or induces, subtyping, whereas this is not necessarily the case for other languages

[CHC90, LP91].

When formalising object-oriented languages using type systems, objects and object types are often encoded using record types[9].

Subtyping is typically based on the notions of *covariance* and *contravariance*. Specifically, in order for a type A to be considered a subtype of another type B, the return types of the corresponding methods must vary covariantly. That is, the return type of a method $A.f$ must be a subtype of the return type of $B.f$. By contrast, the types of method parameters must vary contravariantly, i.e. the type of a parameter p in a method $A.f(p)$ must be a supertype of the type of the parameter q in the corresponding method $B.f(q)$. In languages where instance variables are accessible, their types must be *invariant*, i.e. the types must not change in a subtype.

However, there are a number of cases where subtyping alone does not provide sufficient expressive power. For example, consider the simple case of a *Stack* type supporting two operations, $push(X)$ and $pop() : X$. In order to support stacks of different types - for example, stacks of integers or stacks of strings - subtyping would allow us to make the X type a supertype of both, e.g. *Object*.

```
class Stack {
  void push(X x) { ... }
  X pop() { ... }
}
```

However, this would allow a Stack to push Strings and then pop Integers, which is not what we need.

The solution to this, and an approach which is seeing increasing uptake in popular object-oriented programming languages, is to use *parameterized types*. A parameterized type is defined relative to a second type provided as a *type parameter*, which may be used in the place of object types within the definition of the type.

So in the case of our *Stack* type, we may parameterize it with a type parameter T, and use T instead of *Object* as the parameter type of $push()$ and the return type of $pop()$. This allows us to express that, although any type T is acceptable, the type must be the same between different occurrences.

```
class Stack<T> {
  void push(T x) { ... }
  T pop() { ... }
}
```

[9] As shown by Castagna in [Cas97], object types may equally be encoded using overloading.

The type provided as the parameter of a parameterized type may also be constrained to be somehow conformant to a *bounds* type. This conformance may be based on subtyping, or more complex rules such as F-bounded or match-bounded polymorphism.

1.4.2 From Objects to Models

The structures used in model-driven engineering, as defined by the MOF and described in Section 1.1.2, are very close to those used in object-oriented programming languages. For this reason, it is reasonable to expect that theories developed for type systems of object-oriented languages will apply or adapt well for models. The elements of MOF models which are less common in object-oriented systems are associations as first-class modelling elements, and the meta-level hierarchy, i.e. the use of metaclasses.

In [BW05], Bierman and Wren present a type system incorporating relationships as first-class concepts in an object-oriented language. The relationships they consider include many of the features used by MOF and UML, including the new (in the version 2.0 specifications) notions of association redefinition by subsetting, and multiplicities. Although they do not consider bidirectional relationships, these may be reasonably supported by dynamic checks.

In [THA05], Tobin-Hochstadt and Allen present a calculus for a language including metaclasses, based on the Featherweight GJ calculus [IPW01]. Their calculus is ostensibly inspired by the metaclass hierarchies of ObjVLisp, [Coi87], Smalltalk [GR83], and CLOS [KRB91], but is reasonably generic, and could conceivably be adapted for an MDE-like hierarchy like that presented in Section 1.1.1.

1.4.3 Virtual Types in Beta

An alternative to the use of type parameters for replacing the types used by a program is the use of *virtual types* [MMP89]. Virtual types were first proposed in the Beta language [KMMPN87].

Virtual types are named variables in a class which, unlike normal variables, are filled by a type. Like type parameters, they may be constrained by a type bounds, and used throughout the class in the place of object types. Unlike type parameters, virtual types are assigned from within a class, not by code that makes use of the class.

One important implication of including virtual types is that it breaks the idea that inheritance implies subtyping, since virtual types may be redefined in subclasses. In Beta, subclasses were assumed to generate subtypes, and thus type safety had to be ensured using dynamic checks; Beta's virtual types were not statically type-safe.

Since their introduction in Beta, virtual types have been formalised [IP02, EOC06], and have recently been incorporated into a number of programming languages, in particular as a basis for the parallel extension of mutually recursive types. We introduce three of these, gbeta's family polymorphism, LOOM's type groups, and Scala with its abstract type members, in Sections 1.4.4, 1.4.5, and 1.4.6. Other languages including virtual types or virtual classes include Tribe [CDNW07] and Concord [JDAO04].

1.4.4 gbeta and Family Polymorphism

gbeta [Ern99] is a generalisation of Beta proposed by Erik Ernst, and includes the notion of *family polymorphism* [Ern01], based on the virtual types found in Beta.

Family polymorphism is based on the idea that types are often not defined as standalone structures, but within the framework of a set of types that each refer to one another. These systems, or families, of types, may then be replaced, evolved, or refined as a group, permitting more sophisticated notions of redefinition and reuse.

This view, of mutually referential types that may be treated as groups, fits very neatly with the MDE concept of a metamodel as a set of concepts and the relationships between them. The notion that these might then be interchanged as groups offers promise for being able to reuse MDE components with models defined using a metamodel other than that for which the components were written.

Although gbeta's type system has not been proven to be sound, Igarashi presents a lightweight version of family polymorphism in [ISV05], including a sound type system.

1.4.5 Type Groups

An alternative approach to the definition of mutually referential types is that taken by Bruce and Vanderwaart in [BV99]. Their LOOM language proposes the concept of *type groups*, with similar aims to the family polymorphism of Ernst. In [BV99], the virtual types that are used in order to permit type groups are formalised using a form of type recursion.

Recursive types are those that are defined with reference to other types. The classic motivating problem for recursive types is that of *binary methods*. A binary method is a method defined on an object type for which the parameter is the object type itself. The classic case is an equality method. Suppose we have a class

X which defines a binary *equals* method. A class Y, which inherits from X, may wish to refine the equals method to accept only Y objects as arguments.

```
public class X {
  public Boolean equals (X other) { ... }
}

public class Y extends X {
  public Boolean equals (Y other) { ... }
}
```

However, parameter types of methods must be contravariant with the type which defines the method in order to support subtyping. So, in this example, Y is not a valid subtype of X.

One solution to this problem is to introduce the notion of a *self-type*, as supported by Eiffel [Mey92] as the *like Current* construct, or by LOOM as *ThisType*. Rather than using X as the type for the *other* parameter, we can use *ThisType*. By doing this, the signature of *equals*() need not be redefined in order that the parameter type be changed to Y; this is automatically the case when *ThisType* is reevaluated in the context of Y.

```
public class X {
  public Boolean equals (ThisType other) { ... }
}

public class Y extends X { }
```

This recursion (called simple recursion) can be seen as having an implicit variable that is resolved in the context of a given type. Like virtual types, the addition of recursive types break the implication of subtyping by inheritance, as described in [CHC90].

Type groups in LOOM are formalised by generalising simple recursion to include references to other types, based on their names. A reference to a type T may be interpreted, not as a direct reference to a type, but as a variable or role name, to be filled by a specific type within the context of the given class.

The redefinition of type groups, and most importantly the rebinding of the virtual types within the type groups, is done using the matching relation, $<\#$. This relation is more permissive than subtyping, in particular in that it does not enjoy subsumption. For example, the redefinition of the return type of a method between subtypes must be covariant with respect to subtyping, i.e. the redefined returned type must be a subtype of the original return type. In matching, however, the return type is required only to be a matching type.

Although LOOM uses type groups for the definition of mutually recursive types, type groups are not themselves types. That is, there are no terms in LOOM whose type is a type group.

Type groups from [BV99] are also the basis for the dependent types in Concord, which is presented in [JDAO04] along with a decidable type system including a sketched proof of soundness.

1.4.6 Scala

Scala [OZ05] is a modern object-oriented programming language that incorporates many elements from functional programming, and can be considered functional, inasmuch as all functions are first-class objects in the language. Scala offers an expressive type system, including advanced features such as explicit self-type references, parameterized types with variance annotations and both upper and lower bounds, modular mixin composition, and abstract type members.

Scala is based on the νObj calculus proposed by Odersky in [OCRZ03]. νObj is a nominal calculus that uses a limited form of dependent types which allows for the definition of virtual types as type-valued members of classes. As a partial implementation of νObj, Scala supports the use of what it calls *abstract type members*, which resemble and perform the role of virtual types, but with a sounder formal basis.

[OCRZ03] shows the νObj type system as undecidable. However, [CGLO06], presents a type system FS_{alg} representing the subset of νObj corresponding to the key type system features, including abstract type members, offered by Scala, and offer proofs for the decidability of subtyping and type assignment.

1.5 Summary and Motivation

Model-Driven Engineering is about systems that store, manipulate, and otherwise use models. The mechanisms with which these models are defined, transformed, and used for generating tools or performing actions, range widely from programming languages, to declarative rule languages, to template languages, to control-agnostic object or XML stores. What these model-driven pieces have in common is that the models they manipulate, although defined using different languages, share a common structure defined by the modelling meta-language, i.e. MOF.

Because of this, when seeking a way to reason about the ways in which these components may be safely reused and integrated, the logical place to look is at the types of the models that they use. However, although MOF provides adequate concepts for the objects and object types that are used to make up models, it says little about what models are, nor how we may reason about the types of models.

There are presently few works related to MDE-specific approaches to integration and reuse, and fewer stiller that attempt it on a tool-neutral basis. The approaches there are typically based either on the typing only of the objects within models, or on the use of metamodels without consideration for the conditions under which they may be substitutable.

To address this, we consider approaches to reuse and integration from the domain of object-oriented type systems. There is an increasing interest in the typing of systems involving mutually referential types, in particular through the formalization and use of virtual types. LOOM's type groups, and the family polymorphism of gbeta, take this further by reasoning about collections of interrelated types as groups or families, and address the conditions under which one group may be replaced by another.

Chapitre 2

The Principles of Model Typing

2.1 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 [GLR+02], pattern-based [QVT05], and graph-transformation [Sen03] approaches. Concurrently, a number of efforts are being undertaken to develop or extend programming languages to better deal with models as data structures [MFJ05].

The vast majority of these efforts have chosen to use type systems developed for use within object-oriented development. However, as discussed in [EJT04] and mentioned in [SL04], 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 Section 2.2, 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 2.3 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 Section 2.4 we show how a language and type system supporting these concepts might be built as an extension of existing formalisms.

2.2 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 certain behaviours are not possible for programs that are verified according to a set of rules called a type system.

The process of verifying these rules for a given program is called type checking. 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).

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.

2.2.1 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 2.1. The out-

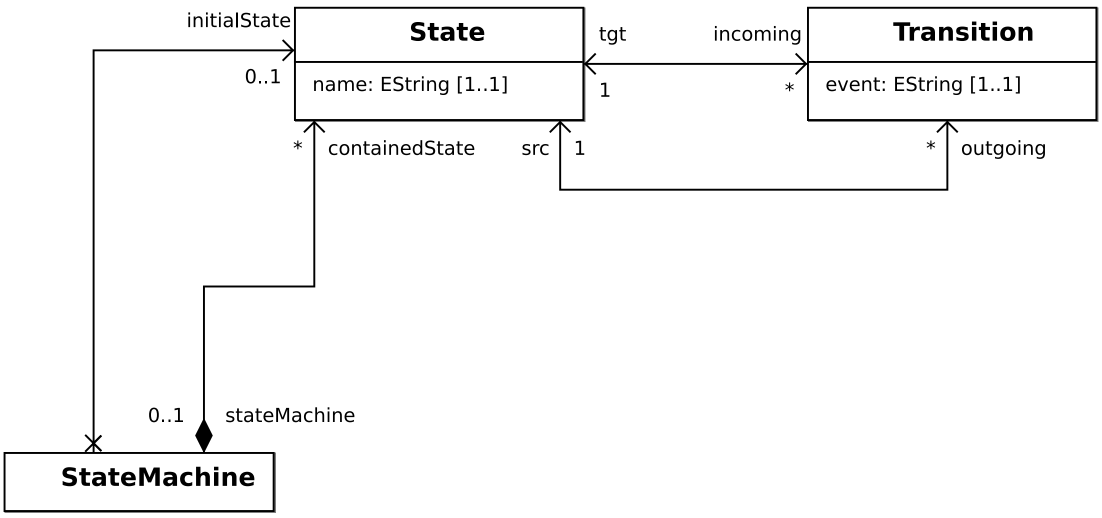


FIG. 2.1 – Simple State Machine Metamodel

put 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.

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. All that is important is knowing what models may arrive as a parameter, and particular what operations may be made on them. These may be assumed to be some variation of the CRUD (Create, Read, Update, Destroy) operators.

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 (Figure 2.2), or from 0..1 to 1..1, mandating that each state machine have exactly one start state (Figure 2.3). Alternatively, we might apply the composite pattern by adding an inheritance of State by StateMachine, for composite state machines (Figure 2.4). Finally, we might consider the addition of a FinalState class as a new subclass of State (Figure 2.5).

The question is, then, does the initial transformation written for models conforming to Figure 2.1 still work with models conforming to these variant metamodels ?

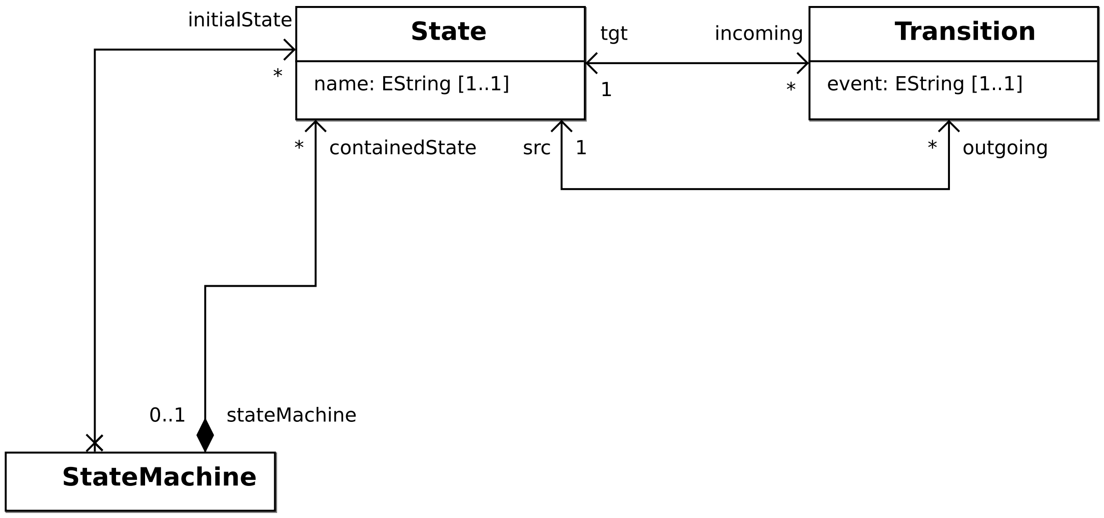


FIG. 2.2 – State Machine Metamodel with multiple start states

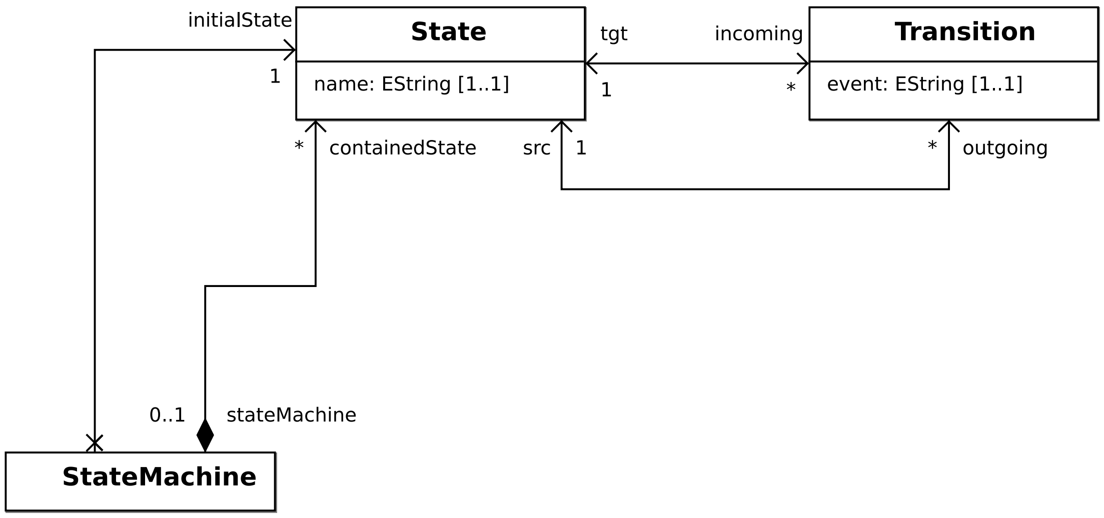


FIG. 2.3 – State Machine Metamodel with mandatory start states

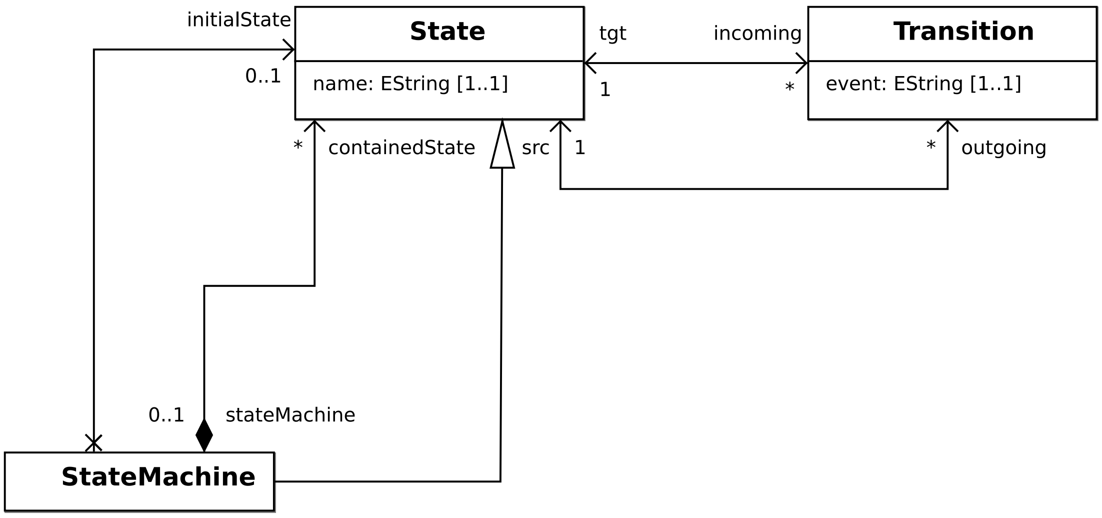


Fig. 2.4 – Composite State Machine Metamodel

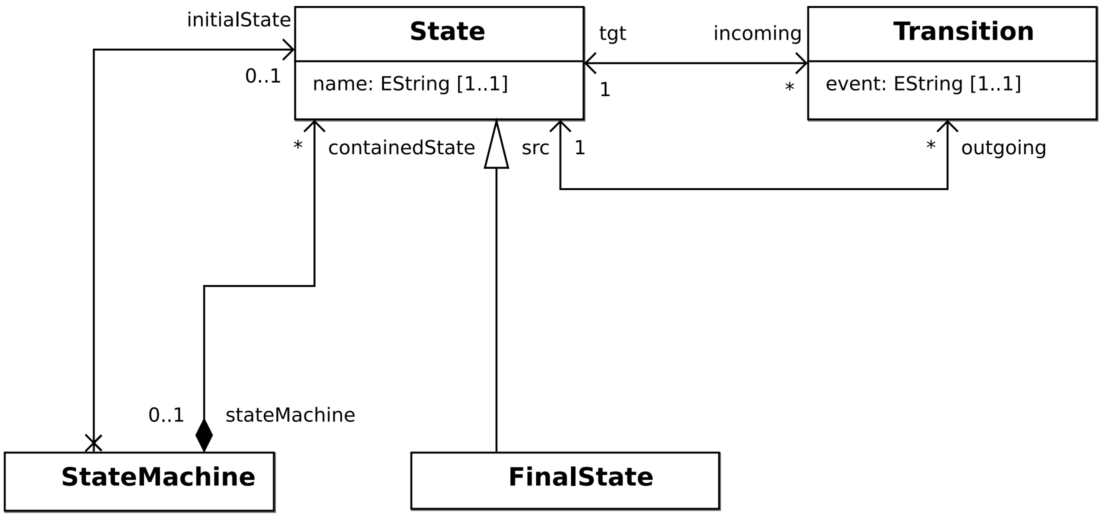


Fig. 2.5 – With Final States

2.2.2 Objects, And Their Types

As described in Section 1.4.1, the basic notions of objects and the type systems that describe them are by now reasonably well understood [AC96]. Also, as mentioned in Section 1.4.2, 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.

2.2.3 Models And Model Types

As discussed in Section 1.1.4, there are two approaches for delimiting what represents a model; the container approach, where one element represents the "root" of the model and the elements it contains or references are included as contents, and the graph approach, where the model is a collection of elements and the links between them, without a root.

This second approach is the one we will take, since it is more general, and does not suffer in the case of languages that do not have a natural root element.

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.

2.2.4 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 artefacts 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 2.2.1 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.

2.3 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.

2.3.1 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 [LP91], 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 Section 2.2.1, the model type required for our transformation is in essence the metamodel shown in Figure 2.1. 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.

2.3.2 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 earlier 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 Figure 2.1 with the composite statechart metamodel in Figure 2.4. 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.

The need for property type invariance is simply demonstrated. Subtyping requires that any code which is safe on a given type must be safe on its subtype. So a program written for a Basic *State* : *Exp* must work for a subtype. Even a single line written for *Basic* :: *State* such as

```
s.stateMachine := Basic::StateMachine.new
```

is not acceptable for $Composite :: State$. The type of $Composite :: State.stateMachine$ is $Composite :: StateMachine$, and a parameter of type $Basic :: StateMachine$ is not acceptable.

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. $Composite :: State$ is not a subtype of $Basic :: State$, as we have seen. $StateMachine$ has a property $containedState$ of type $State : Exp$, which may not vary, so $Composite :: StateMachine$ is not a subtype of $Basic :: StateMachine$. Similarly for $Transition$.

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 $State : Exp$ 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, we are interested not in replacing one type, but in replacing a set of types. 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 [BSvGF03]. 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 [BV99, BSvGF03] 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 heterogeneous collections. Notably, in the context of models, type-safety depends on models remaining homogenous with respect to a set of object types.

2.3.3 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, other than the runtime constraint for synchronisation.

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 Section 2.4, 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 whether elements are optional or mandatory, and single- or multi-valued. An approach for matching multiplicities in their entirety may be found in Section 3.2.4.1.

2.3.4 Model-Type Conformance

Bruce further defines in [BV99] 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 $M.C$.

2.4 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

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

FIG. 2.6 – Language Grammar : MOF structural concepts

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 Vanderwaart in [BV99].

2.4.1 Grammar of types and terms

Figure 2.6 shows the major structural concepts as defined by MOF. A class is defined with a name, a set 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 in their full detail, 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

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 might also be considered as valid types elsewhere in the language (variables, expressions, etc.), but in the interests of explanation we will limit their use to transformation parameter types.

The grammar elements for declaring model types and transformations are shown in Figure 2.7.

For the body of transformation, we provide a basic set of types and terms corresponding to a simple expression language (Figure 2.8). There are variables, assignments, invocations of operations and transformations, and conditional/ite-

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

FIG. 2.7 – Language Grammar : Model Types and Transformations

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

FIG. 2.8 – Language Grammar : Expressions and Statements

ration statements, etc. The only operator that might be considered unusual is the $\rtimes\!\!\!\ltimes$ 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[1]. However, few imperative 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.

[1]Indeed, many rule-based languages, such as Tefkat [DGL$^+$03], are built around some sort of filter functionality.

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

FIG. 2.9 – Signatures of class and transformation tables

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 Figure 2.9. 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.

2.4.2 Language semantics

The semantics of the language is largely the same as that presented in [BV99]. 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 rely 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 [BV99].

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 effectively parameterized 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, specifically transformation parameters, using a model type, thus permitting them to function as models (as described in Section 1.1.1). Semantically, these terms then function much as collections, whose elements may be thought of as being typed as the union of the types declared in the model type.

2.4.3 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.

The object type matching rule, and in turn the matching rules for properties, operations and parameters, modified to account for multiplicities, are shown in Figure 2.10. 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 (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 2.11 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 Section 2.3.4.

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

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

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

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

Fig. 2.10 – Selected type-checking rules for model types : Object-type matching

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

$$\frac{\begin{array}{c} \mathcal{T}(\psi) = (x, \_, m_1, \_) \\ C, E \vdash e : m_2 \\ C \vdash m_2 <\!\# \ \ m_1 \end{array}}{C, E \vdash \psi(e)} \qquad (\text{TS}\textsc{TransInv})$$

$$\frac{\begin{array}{c} C, E \vdash x : m \\ C, E \vdash e : c \\ C, E \vdash c \in m \end{array}}{C, E \vdash x \mathrel{+}= \ e} \qquad (\text{TS}\textsc{ModelAdd})$$

$$\frac{\begin{array}{c} C, E \vdash x : m \\ C, E \vdash c \in m \end{array}}{C, E \vdash x \bowtie c : set < c >} \qquad (\text{TS}\textsc{ModelFilter})$$

FIG. 2.11 – Selected type-checking rules for model types : model types

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, for example to ensure that a model type includes the transitive closure of object types referred to as types of properties, operations or parameters.

↱ matches →	Simple	Multiple-Start	Mandatory-Start	Composite	With-Final-States
Simple (Figure 2.1)	✓	NO	NO	NO	NO
Multiple-Start (Figure 2.2)	NO	✓	NO	NO	NO
Mandatory-Start (Figure 2.3)	✓	NO	✓	NO	NO
Composite (Figure 2.4)	✓	NO	NO	✓	NO
With-Final-States (Figure 2.5)	✓	NO	NO	NO	✓

TAB. 2.1 – Model Type Conformance Relation for State Machine Variants

2.4.4 Application to the examples

If we apply the ModelTypeMatch rule to the example metamodels provided in Section 2.2.1, we are able obtain the model type matching relation shown in Table 2.1.

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 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.

2.5 Conclusion

This chapter has presented a basic approach to formalising models and their types, and to providing a mechanism by which models of one type may be used where models of another type were expected.

Models have formalised simply as graphs of objects. These are then typed by sets of object types, including the types of the links between the objects, in the form of MOF Classes. A notion of model type substitutability has been obtained based on Bruce's notion of Type Groups and type group matching, which has then been simply adapted and extended to model types.

This has been demonstrated as a simple language to demonstrate the ideas of model typing, including a number of type rules which may be added to a simple object-oriented language to support polymorphic model transformations.

The principles that have been presented here are applied to a real-scale model transformation and manipulation language, Kermeta, in the next chapter.

Chapitre 3

Implementing Model Types in Kermeta

The language we present in Chapter 2 is a simple language with the sole purpose of demonstrating the idea of model types and its potential advantages in Model-Driven Engineering. However, it is insufficient for real use in the development of model-driven systems. To this end, model types have been implemented in the model-driven programming language, Kermeta.

This chapter describes the implementation of model types in Kermeta. In section 3.1, Kermeta is presented in detail, including its type system. Section 3.2 lists and discusses in detail a number of requirements that result from the application of the principles from Chapter 2 to a larger-scale model-driven engineering platform. Section 3.3 then describes concretely the changes made to Kermeta to instrument it for models and model types. Finally, Section 3.4 illustrates the use of model types by applying it to a workbench for the manipulation of state machines.

3.1 The Kermeta Language and Environment

As introduced in Section 1.2.2.4, Kermeta [MFJ05, Fle06] is an open-source metamodelling language developed by the Triskell team at IRISA. It has been designed as an extension to EMOF 2.0, a recent version of the MOF specification (as introduced in Section 1.1.3), to be the core of a meta-modelling platform. Kermeta extends EMOF with an action language that allows the specification of semantics and behaviour of metamodels, using an imperative, object-oriented action language which is used to provide an implementation of operations defined in metamodels.

The Kermeta action language has been specifically designed to manipulate models. It includes both object-oriented features and model-specific features. Kermeta includes familiar object-oriented static typing, with multiple inheritance and behaviour redefinition/selection with a late-binding semantics, as well as type parameters, and syntactic closures à la Smalltalk or Eiffel. To make Kermeta suitable for model processing, more specific concepts such as opposite properties (i.e. associations) and handling of object containment have been included. This offers a syntactic advantage over using an existing language such as Java in combination with libraries such as EMF[BSM$^+$03] or JMI[Sun02].

3.1.1 Kermeta Language Features

The Kermeta type system is heavily influenced by that of Eiffel, and to a lesser extent that of Java. Also, conformance with the EMOF metamodel is considered very important, and this consideration influences the structure of the type system.

The most significant structural change with respect to EMOF is the addition of type parameters for classes and operations. For classes, this is modelled by introducing a separation between the definition of a class and its use as a classifier. For example, the definition of $Collection < X >$ is no longer a $Class$, but instead a $ClassDefinition$, whereas $Collection < String >$ is a $Class$. In this way, a class consists of a reference to a class definition, and a set of bindings of the class definition's type parameters to the types that fill them. In order to maintain consistency with the EMOF metamodel, the attributes previously available on $Class$, which now reside primarily on $ClassDefinition$, such as inheritance and ownership of properties and operations, are made available on $Class$ as derived attributes.

In addition to being defined using the action language, operations may be declared as abstract. This makes them similar in functionality to the signature definitions of methods on interfaces in Java. Additionally, Kermeta introduces the concept of semantically abstract classes. This means that a class may be declared as not abstract and yet define or inherit one or more abstract operations, yet still type-check. In this case, the class is declared semantically abstract, and may not be instantiated.

The action language of Kermeta is not significantly different from what one expects in a modern object-oriented programming language. Unlike Java, there is no return statement - the programmer must assign into an implicitly declared $result$ variable. This, as well as the absence of *continue* and $break$ statements in loops, helps to regularize control flow.

Assignments are modified such that assigning to a reference which participates in an association will also affect the other end, i.e. it is the association as a whole which is updated. This also takes composition into account. For example, assigning an object into a contained role will automatically remove it from any other contained roles in which it is participating.

A big part of metaprogramming is the treatment of collections. To assist with this, Kermeta supports syntactic closures. This allows for the definition of functions such as *select*(), *collect*() and *reject*() on collections, where the parameter is a function. For example, the *select*() operation is parameterized by a function which is used as a filter on the collection's elements. The combination of syntactic closures and type parameters allows for the definition of a method like *collect*(), in which the return type for the method is defined as the return type of the function passed as a parameter.

Clearly, there are number of different kinds of type in Kermeta. From EMOF, there are Classes (modified as above), Enumerations, and Datatypes (which are treated in Kermeta as type aliases, much like in EMF). Additionally, there are type parameters, since these may be used to type properties, operations, parameters, etc. Function types are also present, in order to support syntactic closures, as seen in the function-typed operations available on collections.

3.2 Requirements for model types in Kermeta

This section presents a number of high-level considerations that emerge when implementing model types into a more realistically scaled model-driven development platform, such as Kermeta. This includes concerns such as the way in which model types are defined, ensuring homogeneity of models, providing a sufficiently permissive relation for model type matching, and providing an operator for creating a model type as an extension of another.

3.2.1 Basic Requirements

To recapitulate from Chapter 2, there are two basic motivations, and thus requirements, for building model typing into a language, in this case Kermeta.

Firstly, the programmer should be able to manipulate models as first-class terms in the language. This includes the creation of new models from model types, the safe addition and removal of objects to/from models, and the inspection of the objects contained by the model.

Secondly, the programmer should be able to write programs that work against a family of different model types. That is, it should be possible to write a program against a given model type and have it work correctly upon models of a type that matches the initial type.

3.2.2 Defining Model Types

One of the most significant differences between model types in Kermeta and the features offered by programming language techniques like type groups and family polymorphism (as described in Section 1.4) is that model types are constructed by reference to object types, not as containers of object types. There are three main reasons for this choice.

Firstly, building model types by reference allows the construction of model types from classes defined elsewhere. This means that model types may be defined without having to reorganize existing metamodels into model types, which is particularly important since many metamodels are defined as standards, within organisations such as the OMG. Building model types by reference is also important for building models, since by implication it allows the construction of models from objects whose types are defined elsewhere.

Along similar lines, building model types by reference to classes also allows a single class to be used in several different model types. Again, by implication, this allows a single object to be used in models of different types. If model types were defined as containers of object types, then each object would be usable only in the model type within which its class had been defined. In this way, including classes by reference allows for overlapping model types, which is useful for the construction of, for example, annotation models or traceability models.

Thirdly, separating the definition of model types from the definition of classes allows for a clearer separation of the roles of packages and model types. In particular, it allows for the definition of a merge operator only for packages, rather than for both packages and model types. This operator is discussed in Section 3.2.5.

As described in Section 3.1.1, Kermeta adds to EMOF a distinction between concrete types, and type definitions, which may include unbound type parameters. Therefore, another consideration for model types is whether they reference type definitions or types. This is essentially a question of whether or not to include generic classes with unbound type parameters. Since this decision is heavily dependent on the matching relation being used, we will address it after detailing the matching relation used in Kermeta, in Section 3.2.4.2.

3.2.3 Homogeneity of Models

One of the primary concerns when building model types into a programming language is of ensuring homogeneity of models. That is, ensuring that the types of all objects in a model are present in the model's model type.

Since we are using pointwise object-type matching instead of subtyping, a type A' within a model type MT' is not a subtype of the corresponding type A in MT, even if $MT' <\# \; MT$. As such, it is very important to ensure that at no time does a program attempt to add an object of type $MT' :: A'$ into a model of type MT.

The implication of this is that model types are not polymorphic. For example, an attempt to add an object of type $MT :: A$ to a variable of type MT is validated by the type-checker, but attempting this when the runtime model is of type MT' would result in a non-homogeneous model.

Because of this, reuse is achieved instead through the use of type parameters. Just as a class may declare type parameters to be satisfied by object types, we add the ability to declare model type parameters, to be satisfied by model types. The distinction between the two is made on the basis of the bounds constraint of the type parameter.

A model type parameter MTP bounded by model type MT, once declared, also implicitly introduces a set of virtual type parameters $MTP :: A$, $MTP :: B$, corresponding to the object types contained by MT. This model type parameter may be filled by any model type MT' such that $MT' <\# MT$, in which case each virtual type parameter $MTP :: A$ will be filled by whichever object type in MT' matches $MT :: A$ in the pointwise object-type match relation.

Once these virtual type parameters have been implicitly bound, they may not be re-bound. This, and the fact that the binding is done in response to a unique pointwise matching of model types, ensures that the virtual type parameters will be both consistent amongst themselves, and stable for the duration of the parameterized class. Both of these conditions help ensure that the models being treated remain homogeneous.

3.2.4 Model Type Matching in Kermeta

Model-type matching in Kermeta has been adapted slightly from the definition in Chapter 2 (which is in turn based on the matching in [BV99]). Most significantly, the requirement that matching classes have the same name has been removed. This has a number of important implications :

- It becomes possible for a model type to match another in a number of different "ways" since, without the uniqueness constraint of names to rely upon, the pointwise matching relation between object types is no longer sure to be 1-to-1. A non-ambiguous object type match is required for any operation upon a virtual type parameter (for example, instantiation), so these ambiguous model type matches must be detected and rejected by the type checker.[1]
- Matching is therefore no longer reflexive, since a model type may match itself ambiguously.
- The algorithm for establishing matching is much more costly, since it becomes a constraint satisfaction problem rather than a simple evaluation of 1-to-1 conformance or non-conformance. When checking if a model type consisting of m types matches another consisting of n types, there are $m \times n$

[1]A future extension will be to permit the programmer to disambiguate a model-type match by providing a number of object-type matches explicitly.

3.2.4.1 Matching Multiplicities

One of the significant simplifications in the language presented in Section 2.4.1 was to represent multiplicities as simply multi-valued or not, and optional or not. Since multiplicities in MOF are somewhat more sophisticated, the differences must be considered in order to match properties, operations and parameters in Kermeta.

Multiplicities in MOF consist of four elements ; upper and lower bounds expressed as naturals, and Boolean flags for orderedness and uniqueness. Unbounded collections may be expressed using an infinite upper bound (often represented by -1), while optionality is represented using a lower bounds of zero. Orderedness and uniqueness have meaning only on multiplicities whose upper bound is greater than one, and combining them allows for the specification of sets, bags, ordered sets and sequences.

The goal of multiplicity matching is that any operation written against an element having multiplicity M may be safely performed on an element of multiplicity M'. The relation is therefore highly dependent on the language being considered. For example, in declarative languages such as Tefkat [LS05], there is little difference between the operations available on single-valued elements and those available on multi-valued elements, since both return a single element. In Kermeta or EMF, by contrast, accessing single-valued elements will return single elements while accessing multi-valued elements will return collections.

In Kermeta, therefore, the first consideration we make when comparing multiplicities is to ensure that the distinction between single-valued and multi-valued elements is respected. Thus, an upper bound of 1 is a privileged value when we are considering matching.

In an environment such as Java/EMF, accessing a multi-valued collection will return the same type (EList) regardless of the orderedness or uniqueness of the collection. This is not true in Kermeta, which has a different type for each of the four possible collections. The subtype hierarchy of these collection types is shown in Figure 3.1.

This diagram shows that ordered collections are compatible with (and even defined as subclasses of) unordered collections, and thus orderedness may be added to (but not removed from) a multiplicity and preserve matching. By contrast, varying the uniqueness constraint, either by addition or removal, of a multiplicity does not preserve matching[2].

Varying the upper and lower bounds of a collection in ways other than between single- and multi-valued, is a more complicated consideration. Taking a

[2]Interestingly, the reason for this, that a given element may or may not be safely added multiple times to a collection, is not statically detectable, so for the purposes of a static type system, there is little advantage to enforcing it.

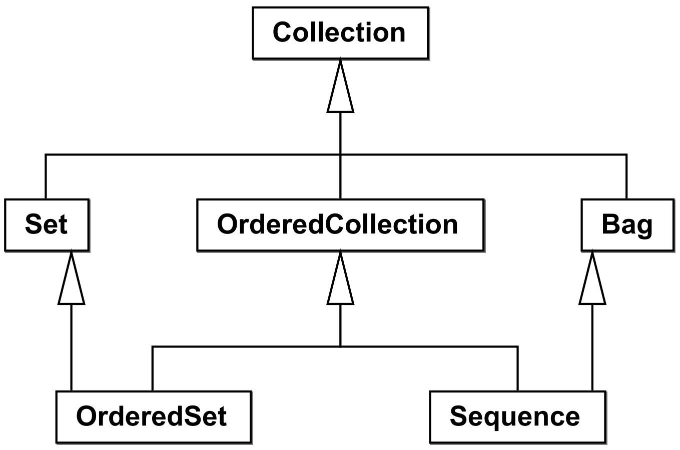


FIG. 3.1 – Subtyping hierarchy of collection types in Kermeta

set-theoretic view of types, it is tempting to say that any contraction of a multiplicity range, i.e. raising of the lower bound and/or lowering of the upper bound, preserves a subsumption relationship between the sets of models permissible in a model type, and thus should preserve matching. However, in the event that the programmer adds n elements into a property of multiplicity range $0..n$, such a contraction would result in a runtime overflow error on the property. Indeed, this would suggest that broadening the multiplicity range would be acceptable. However, attempting to copy the property's contents into a collection of size n might fail if its upper bound was raised. What one finds is that the only way of ensuring that no errors are introduced by varying a multiplicity is to make the bounds invariant. This, however, is inconvenient for the programmer.

The truth is that, even disregarding the variation of multiplicity ranges, errors related to underflow (not enough elements) or overflow (too many elements) are impossible to detect statically in an imperative language such as Kermeta. Control flow constructs such as loops and conditionals, for example, make it impossible to count additions into, or extractions from, a collection. Therefore, varying the multiplicity range makes no difference to our ability to statically detect errors.

However, it is not intuitive to allow the programmer to vary the ranges arbitrarily, since it would allow even non-overlapping ranges, such 0..2 and 50..100 to be compatible with one another. To this end, we have taken the decision in Kermeta to allow contraction of multiplicity ranges but not broadening, since it preserves the set-theoretic viewpoint for the user.

3.2.4.2 Model Types and Generic Classes

As mentioned in Section 3.2.2, there is a question of whether to reference type definitions, including generic classes with unbound type parameters, or concrete types only.

When a program/transformation is written, it works with a set of types. These types might be primitive types, enumerations, parameter-less classes (as opposed to unparameterized generic classes), or parameterized classes (i.e. generic classes with bound type parameters). To be thorough, the model type required by this program or transformation should provide a type for each of these types.

However, the presence of parameterized types has the potential to make automated matching more complicated. Notably, it becomes possible for a generic class to satisfy a given required type if only it can be parameterized by the right type. However, iterating over the set of possible type parameters that might make the generic into a matching type would be extremely inefficient and probably infinite.

One simplification of this problem is to match type definitions rather than types. That is, a generic required type may be satisfied by a provided type with the same number of type parameters having compatible bounds constraints. A non-generic required type may not be satisfied by a generic type, regardless of how it might be parameterized. This rules out a number of potentially valid matches, but greatly simplifies the matching algorithm.

At this point, Kermeta model types may not include classes with type parameters. However, in order that this simplified approach to matching generic classes might be supported at some point in the future, model types are defined using type definitions, and not types.

3.2.5 Merging

Through model-type parameters that may be substituted using matching, it is possible to write code that is reusable for multiple model types. However, another important aspect of reuse is for the definition of model types themselves.

To support this, rather than defining an extension mechanism for model types, a *merge* operator is provided for packages. This merge mechanism is based on the package merge provided in UML2, which is also present in CMOF, and is also conceptually similar to the modular mixins used in Scala [OZ05]. This mechanism, although conceptually similar, is distinct from class inheritance, and allows a package to be defined in terms of the elements that it adds to another, instead of having to redefine the shared features explicitly.

An example of merge may be seen in Figure 3.2. So when package $P2$ merges another, $P1$, for each class definition ($P1 :: A$ and $P1 :: B$ present in the extended package, a class definition ($P2 :: A$ and $P2 :: B$) will be available in the extending package. Where the extending package contains an explicit definition for a class, such as in the case of B, the resulting class definition $P2 :: B$ will offer features (properties and operations) as if it had inherited from the corresponding class

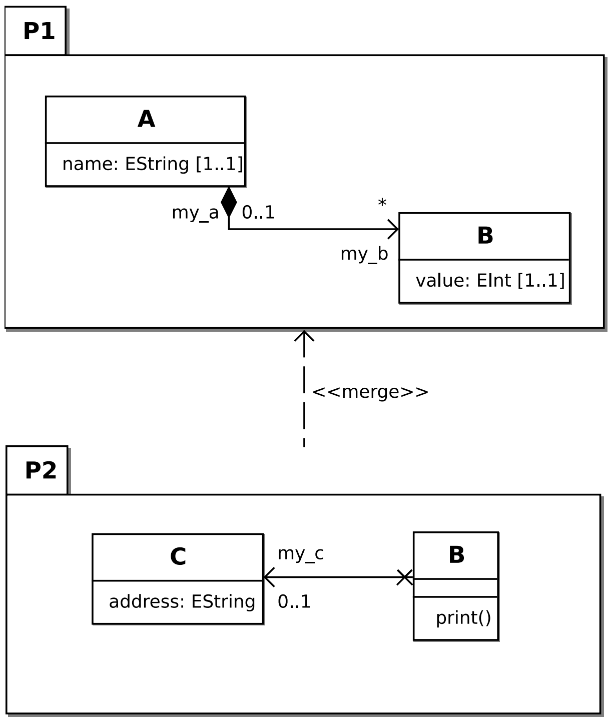


FIG. 3.2 – A simple example of package merging

definition $P1 :: B$. The types that are accessible on $P2$ are the same as those shown in the $P3$ package show in Figure 3.3.

Obviously, this inheritance of features depends upon the static detection of any conflicts between the merged features and those explicitly defined on the extending class definition, which is done using the same logic as for normal class inheritance. This acquisition of features, as well as the detection of conflicts, also extends to the merging of multiple model types, using the rules for conflict detection and resolution that are used elsewhere in Kermeta [Fle06]. The exception to this is for references to other types.

As a general rule, references to types are treated as virtual in all cases where the referred type is present in the package. This includes types of properties, inherited types, return and parameter types for operations, and also occurrences in expressions, such as types of variables and actual type parameters. In the case where a package is defined as an extension of another, these virtual types are automatically rebound to the appropriate types in the extending package. So, in the case of $P2 :: A$, the type of the property $P2 :: A.my\_b$ will be $P2 :: B$, and not $P1 :: B$.

This type rebinding requires a modification of the runtime treatment of type literals; for example, the instantiation of an object type in the model type must

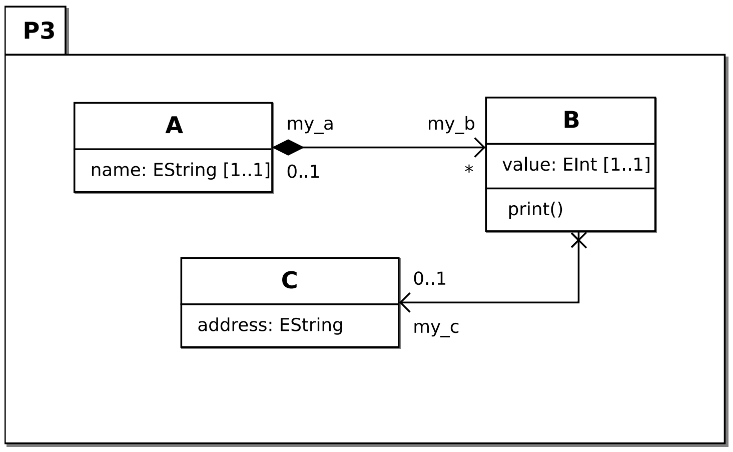


FIG. 3.3 – The effective result of package merging

be treated as an instantiation of the class to which the virtual type resolves.

Once it is possible to extend a set of classes using a merge of the package in which they are defined, the respective model types may be constructed simply by reference to the two sets. That is, in the case of package $P2$ which merges $P1$, two model types $MT2$ and $MT1$ may be defined as the sets of classes in each of $P2$ and $P1$.

An object type $P2 :: A$ that results from the merging of package $P1$ by another, $P2$, is not a subtype of the corresponding "merged" object type $P1 :: A$. For example, $P2 :: A$ cannot be a subtype of $P1 :: A$, since the type of the my_b property has changed, and subtyping requires property types to be invariant. However, class that are merged in this way do preserve a matching relationship $<\#$. This lack of subtyping is one of the key differences between merging and object type inheritance, which does guarantee subtyping.

However, although merging of object types results in matching, the same cannot be said for the model types corresponding to two merged packages. As mentioned earlier, a model type MT' matches another MT iff for each included object type in MT, there exists a unique matching object type in MT'. While merging does ensure that there will be a match for each object type, it can make no guarantees that there will be only one. That is, a "merged" model type could introduce ambiguous object-type matches.

3.3 Building Model Types into Kermeta

The addition of model types to the Kermeta platform is very much a cross-cutting concern. The process requires changes to almost every aspect of the platform, from the metamodel, type checker, standard library and runtime to user components such as integrated editors and import/export tools.

This section describes the approach by which support for model types and models have been implemented into Kermeta. There is no attempt to describe the entire development process. The focus is rather on the mechanisms by which the requirements outlined in Section 3.2 are handled.

Section 3.3.1 describes the construction of the Kermeta metamodel and the changes that were made to it to support model typing. The changes made to the Kermeta concrete syntax are described in Section 3.3.2. Section 3.3.4 describes the algorithm used for determining whether model types match one another, which is key for supporting reuse. Sections 3.3.5 and 3.3.6 present the changes to the Kermeta framework and interpreter, respectively. Finally, Section 3.3.7 discusses the provision for runtime access to type parameters, a consequent change made while adding support for models and model types.

3.3.1 Changes to the Kermeta Metamodel

In keeping with the paradigm of model-driven development, the Kermeta language is, of course, specified using a metamodel. To provide an improved separation of concerns for the language designers, this metamodel is built up by merging a number of “aspects”, each representing a structural or behavioural consideration of the language. In this section we present a number of these aspects and show how these are refactored, built upon and added to by an aspect supporting model typing. A more complete description of Kermeta and its construction may be found in [Fle06].

Since Kermeta is designed from the ground up as an extension of EMOF, the first aspect is just that; the part of the Kermeta language representing those elements taken from EMOF. However, this aspect, shown in Figure 3.4, is not exactly EMOF. In order to add support the definition of parametric classes, there is a separation between class definitions and types. Generic classes are defined as $ClassDefinitions$ and realized into concrete types as $Classes$ with the necessary $TypeVariableBindings$. Compatibility with the EMOF metamodel is maintained using derived properties on $Class$, which substitute for type parameters and provided the properties and operations from $ClassDefinition$.

In order to support the language features presented in Section 3.1.1, the Kermeta metamodel also includes an aspect representing the extensions made for its type system. These are shown in Figure 3.5. This includes support for function types, product types, the void type, as well as type parameters ($TypeVariable$), and a mechanism ($TypeVariableBinding$) for binding type parameters to types in order to construct concrete parameterized types. Originally, $TypeVariable$ was a concrete class. However, in order to allow for the definition of type parameters that bind to model types rather than object types, the class was rena-

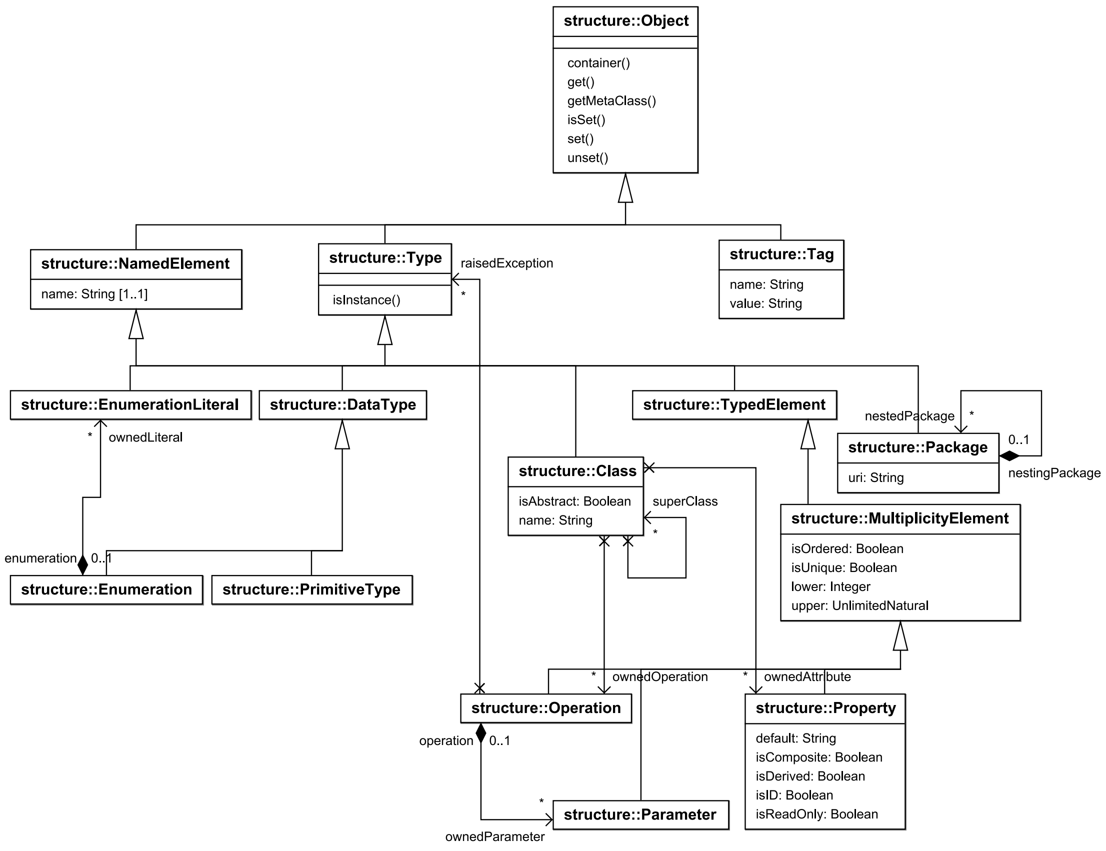


Fig. 3.4 – Kermeta metamodel : Base Structure

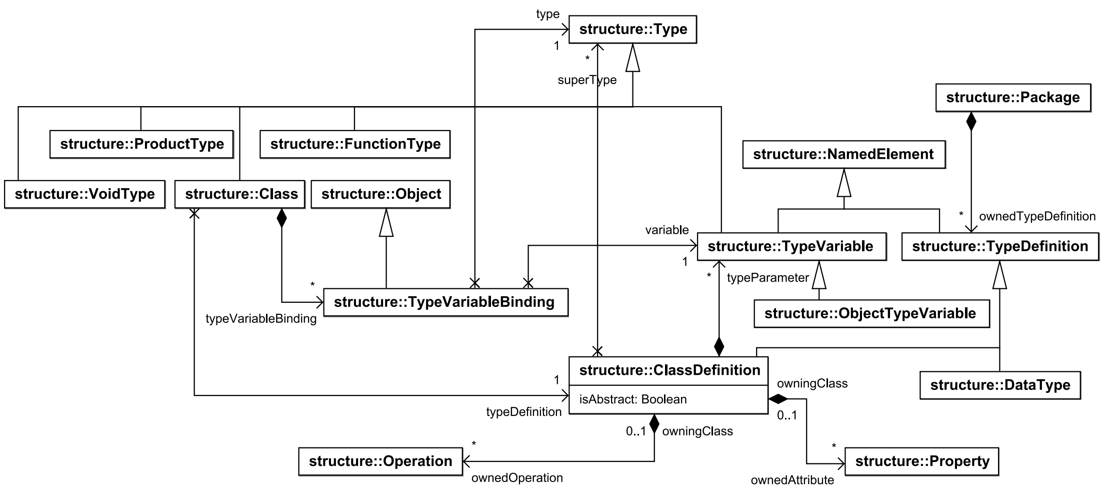


FIG. 3.5 – Kermeta Metamodel : Type System Structure

med as *ObjectTypeVariable*, and its features extracted to an abstract superclass *TypeVariable*.

There are a number of aspects provided for the specification of Kermeta's action language, the majority of which we will not present here, since they are not concerned by the addition of models or model types.

The majority of the changes made for model types are implemented as a new model typing aspect, show in Figure 3.6.

There are three main concerns provided for by this metamodel fragment : model type definition, models, and the necessary changes to type parameters to support model-typed type parameters and virtual type parameters.

Model types are defined using the *ModelType* class, which inherits from both *Type* and *TypeDefinition*. Model types include type definitions with the *includedTypeDefinition* reference, although they may not recursively include other model type definitions. Additionally, model types provide an *isModelTypeOf*() operation, which serves a role analogous to that of *isInstance*() defined on *Type*, in that it replies whether the model type is indeed the type[3] of the model passed as a parameter.

The *Model* class is defined as a subclass of *Object*. This is done to ensure compatibility with EMOF with respect to reflection. For example, the reflective *invoke*() method defined on *Object* take as parameters, and returns, *Objects*. Since we wish to be able to invoke model-type-enabled operations in the same ways as we do other operations, including reflectively, we must be able to pass models as parameters to, and accept models as results from, reflective invocation.

[3]Since model types are monomorphic, as explained in Section 3.2.3, a model has only one type.

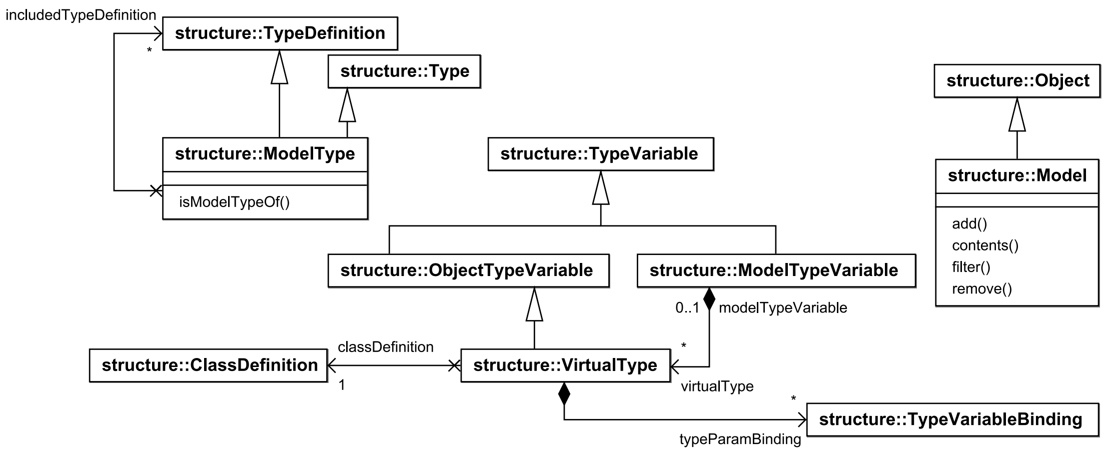


FIG. 3.6 – Kermeta Metamodel : Model Typing Structure

```
class Model inherits Object {
  operation add(o : Object) is do ... end
  operation remove(o : Object) is do ... end
  operation contents() : Collection<Object> is do ... end
  operation filter( t : Type ) : Collection<Object> is do ...
     end
}
```

Listing 3.1 – The Model Class

Therefore, *Model* inherits *Object*.

Model defines the simple operations that one might expect of a container of objects : the addition (*add*()) and removal (*remove*()) of objects, and the accessing either of all objects (*contents*()) or all that conform to a given type (*filter*()). To clarify the diagram above, the signature of *Model* in Kermeta is show in Listing 3.1.

The *add*(), *remove*(), and *filter*() methods are treated specially by the type-checker. Invocations of the former two are accepted only when the expression passed as a parameter is of a type which is "understood by" (is present in) the model type. Calls to *filter*() may use as arguments only types (specifically *Classes* or *VirtualTypes*) belonging to the model's type (which is either a *ModelType* or a *ModelTypeVariable*), and has its return type optimised by the type checker to be a collection of the type passed as a parameter. These special treatments are necessary since Kermeta's type system (like those of most imperative object-oriented languages) does not support dependent types.

In order to support code reuse through the use of model-type parameters as detailed in Section 3.2.3, *ModelTypeVariable* has been defined as a sub-

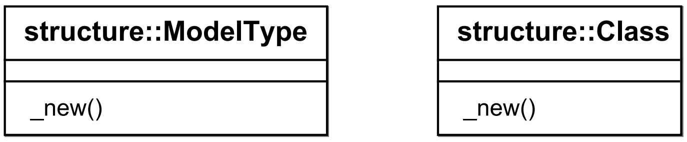


FIG. 3.7 – Kermeta Metamodel : Instantiation

class of *TypeVariable*. Each *ModelTypeVariable* introduces a number of virtual type parameters, which are modelled by the *VirtualType* class. Specifically, a *ModelTypeVariable* will contain (via the *virtualType* property) one *VirtualType* for each *ClassDefinition* included in the *ModelType* given by the type parameter's *supertype* constraint. Virtual types resemble classes, in that each consists of a reference to a class definition and a set of type variable bindings.

The final fragment of the Kermeta metamodel affected by the implementation of model types that of instantiation, which has been modified for instantiation of models from model types. Instantiation of objects from classes is represented by a *new*() operation on *Class*, which resides in a separate metamodel fragment. To this fragment, we add a similar *new*() operation on *ModelType*, which yields a *Model*. The resultant fragment is shown in Figure 3.7.

3.3.2 Changes to the Kermeta Concrete Syntax

This section presents the changes made to Kermeta's concrete syntax for model types. These changes fall into three categories ; changes made for defining model types, changes for models, and changes for model-typed and virtual type parameters. For the most part, these changes do not involve changes to Kermeta's grammar, but to the loading process during the transition from the abstract syntax tree to the Kermeta model.

Syntactically, a model-type definition is introduced using the *modeltype* keyword and a name. This is followed by a comma-separated list of type definitions (as fully-qualified names) enclosed within braces . The EBNF rule for this is as follows.

```
modelTypeDefn ::= "modeltype" ID
        LCURLY qualifiedID (COMMA qualifiedID)* RCURLY;
```

In pure object-oriented languages like Smalltalk, objects have little or no specific syntax ; everything is an object. Even in multi-paradigm object-oriented languages such as Java[4], objects are simply any term whose type is a class ; once again, there is little syntactic presence.

[4]Multi-paradigm in that it includes objects as well as value-types such as integers.

Since we propose to integrate both object- and model-oriented paradigms in Kermeta, we take an analogous approach. A model has little or no syntax ; any term in the language whose type is a model type (or a model-type parameter) is a model.[5] Operations on models for addition, removal and inspection of objects appear using normal operation calls rather than with specific infix syntax. Instantiation of model types, like instantiation of classes, also appears as a normal operation call, rather than with a special syntax as in Java. In this way, there is no change to Kermeta's concrete syntax for models.

Model-typed type parameters do not vary syntactically from object-typed type parameters. They appear within comma-separated lists within angled brackets in class or operation declarations, as a name followed by a bounds constraint. The differentiation between the two types of type parameter is made at load-time based on the type of the bounds constraint. If the constraint is absent, or is a class, then the type parameter is an $ObjectTypeVariable$. If the bounds constraint is a model type, then it will be a model-typed parameter. Thus, unbounded model-type parameters must be declared using the *Top* model type, $Model$. Subsequent references to the type parameter, as with other type parameters, are simply by name. These changes do not affect Kermeta's grammar, since they are treated by the loader, during the conversion from an abstract syntax tree to a model.

As described in Section 3.2.3, a model-type parameter will implicitly introduce a number of virtual type parameters for each class definition in the model type. Since these are implicitly introduced, there is obviously no syntax for their introduction. When they are subsequently used, these types are used relative to the scope of the model-type parameter, separated by the :: symbol. So, if a model-type parameter V is bounded by a model type M containing a class C, then the corresponding virtual type may be referenced as $V :: C$. Like the changes for type parameters, Kermeta's grammar is not affected. The changes are made in the interpretation of qualified identifiers in the loader.

3.3.3 Changes to the Kermeta Type Checker

The changes made to the Kermeta type-checker in order to support checking programs involving models and model types are many and are scattered throughout the module's code. However, there are four main elements treated :

– validity of model type definitions

[5]Object-oriented languages are designed for writing programs that treat objects, and many (typically imperative object-oriented languages like Java or Eiffel) have little or no syntax for declaring objects. In the same way, Kermeta with model types, as a model-oriented language, is designed for writing programs that manipulate models, and has little or no syntax for declaring models.

- extra checks and optimisations for methods whose parameter and return types are incompletely specified by the Kermeta metamodel
- type conformance related to model types, model-type parameters and virtual types
- evaluation of model-type matching for model-type parameters

It is necessary to check that the definition of a model type is coherent. In fact, the only requirement at this point is that the type names be unique, in order that virtual type names will be deterministically resolvable. One possible future check might be to ensure a degree of local coherence. At present it is possible for a model type to include a class C that makes reference to another class D, and for the model to also include another, unrelated class, also named D. Although this does not effect matching (the link to the originally referred is kept and treated according to subtyping), it may be confusing for the programmer.

There are a number of occasions in Kermeta where the type system is unable to adequately capture the type constraints present on an operation. The $new()$ operation, for example, is specified as returning *Object*, but in fact will return an instance of the class upon which it is invoked. Since Kermeta's type system does not support dependent types, this must be enforced by an optimisation of the return type in the type-checker.

There are four methods for which checks or optimisations of this kind are introduced because of model types. The $new()$ operation on *ModelType* returns a $Model$, but is optimised to return a model of the correct type, in the same way as $new()$ on $Class$. The arguments of the $add()$ and $remove()$ operations on $Model$ are typed as $Object$, but in fact must be one of the object types present in the model type of the model. Similarly, the $Type$ passed to the $filter()$ operation must, if it is known statically (i.e. is a literal), be one of the types present in the model type of the model : a $Class$ if the model is typed by a $ModelType$, or a $VirtualType$ if the model is typed by a $ModelTypeVariable$. Also, the return type of $filter()$, specified in the metamodel as $Collection < Object >$, is optimised in the type-checker to be a collection of the type passed as a parameter (once again, if the filter type is known statically).

Since there are new terms (models) and types (model types, model-type parameters, and virtual types) in the language, consideration must also be given to type conformance and type substitutability. In fact, these are very simple.

Models only exist when expressed as terms whose types are model types or model-type parameters, and thus do not need to be checked further. Virtual types may be used to type objects in the same way that object-type parameters are.

As explained in Section 3.2.3, model types are monomorphic, so type substitutability is simple ; a model type is not substitutable for any other type. The same applies for a model-type parameter ; a model-type parameter may not be substituted for any other type, including the model type which serves as its bounds constraint. Virtual types also have no conformance relation with their bounds constraint (the class from which they take their name; the bounds is not explicit). However, when two virtual types in the same model-type parameter have bounds which are related by subtyping, the virtual types will also be related by subtyping. That is, if the bounds for a model-type parameter V is a model type M including classes C and D such that C is a subtype of D, then the virtual type $V :: C$ will be a subtype of the virtual type $V :: D$.

3.3.4 An Algorithm for Model Type Matching

The decision described in 3.2.4 to not consider the names of types when matching model types, makes the algorithm for determining matching significantly more complex.

The dependencies involved when evaluating model type matching are heavily cyclical. For example, a class C' matches another C if for each property $C.p$ of type D, there is a matching property $C'.p$ of type D', such that D' matches D. Thus, $C' <\# C$ requires $D' <\# D$. However, if $C.p$ is part of an association between C and D, then D will have a property $D.q$ of type C, so $D' <\# D$ will equally require $C' <\# C$. This is a symmetric dependency, and since there are no restrictions on cyclic properties, cycles involving three and more types are also possible.

The implication of these cyclic dependencies is that, in the general case, an object type cannot definitively be locally proven to match another. However, there are a number of ways that a provided object type C' may be locally proven to fail to match another C. These are formalised in Listing 3.2 as OCL constraints for provisional local type matching.

A class provisionally matches another if it has matching properties and operations for each property and operation on the required class. Also, abstract classes may not match non-abstract classes, since a program may attempt to instantiate it.

Properties, operations and parameters may be checked to see if their multiplicity matches another. Singleton and multi-valued multiplicity elements do not match one another

The check for matching properties requires that they have the same name and composition semantics, and covariantly matching multiplicities. A readOnly property may not match a non-readOnly property. A property without an opposite may not match a property with an opposite, and properties with opposites must have opposites with the same name.

The check for matching operations requires that the two operations have the same name and arity, and covariantly matching multiplicities. The respective parameters of the operations must have covariantly matching multiplicities.

```
context MultiplicityElement def:
  multiplicityMatchFor(m : MultiplicityElement) :
     Boolean =
    (m.upper = 1) implies (self.upper = 1)
    and self.upper <= m.upper
    and self.lower >= m.lower
    and m.isOrdered implies self.isOrdered
    and self.isUnique = m.isUnique

context Property def:
  localMatchFor(p : Property) : Boolean =
    self.name = p.name
    and self.multiplicityMatchFor(p)
    and not c.isReadOnly implies not self.isReadOnly
    and self.isComposite = p.isComposite
    and not p.opposite->isOclUndefined implies
        not self.opposite->isOclUndefined
        and (self.opposite.name = p.opposite.name)

context Operation def:
  localMatchFor(o : Operation) : Boolean =
    self.name = p.name
    and self.multiplicityMatchFor(o)
    and self.ownedParameter->size() = o.ownedParameter
       ->size()
    and Set{1..o.ownedParameter->size()}->forAll( i |
        o.ownedParameter->at(i).multiplicityMatchFor(
           self.ownedParameter->at(i)) )

context ClassDefinition def:
  localMatchFor(c : Class) : Boolean =
    not c.isAbstract implies not self.isAbstract
    and c.ownedAttribute->forAll( p |
         self.ownedAttribute->exists( p2 | p2.
            localMatchFor(p) ) )
    and c.ownedOperation->forAll( o |
         self.ownedOperation->exists( o2 | o2.
            localMatchFor(o) ) )
```

Listing 3.2 – OCL Constraints for provisional local type matching

The other conditions are those that cannot be assessed locally. The types of properties and the return types of operations must match covariantly (i.e. the provided type must match the required type). For parameters, contravariant type matching is required. That is, either the provided parameter type, or one of its subtypes in the model type, must match the required parameter type.

When name matching is required, each type in the required model type has only one type in the provided model type which potentially matches, since (qualified) type names in MOF are unique. Thus, cyclic dependencies are not problematic, since one failing object type match will break the model type match. An algorithm might thus simply ensure that there exists a like-named provided type for each required type, and that the provided type does not violate any local conditions.

By contrast, ignoring type names means that a required object type may be matched by many provided object types. The algorithm must ensure that there is one *and only one* provided object type that matches the required type. Thus, unlike the name-dependent matching, there arise cases where evaluating only local conditions will result in ambiguous matches, where non-local conditions might resolve these ambiguities.

To avoid this, the algorithm for model type matching is based not on finding object types that match, but instead eliminating object types that do not match. In evaluating a potential object type match, the algorithm may either reject the match outright, or introduce dependencies by which it may be eliminated by evaluation of subsequent matches. The structures used are shown in Figure 3.8.

A binding is a pair of types (Classes); one required ($from$) and one provided (*to*), which represents a potential object-type match. A Dependency is a relationship whereby one *conclusion* binding requires another *premise* (SimpleDependency) or a set of other *premises* (DisjunctiveDependency). The matcher itself maintains lists of candidate bindings and of dependencies that have not been disproven.

To begin the evaluation of a model type match, an initial set of candidate object-type bindings is constructed, consisting of all possible matches, i.e. the cross-product of the object types in the provided and required model types. After construction, the list of candidates may not be added to, only reduced. The pseudo-code for building the list of initial candidates is show in Figure 3.9.

Following this, each candidate match is evaluated in turn. If a candidate depends on another binding, and this other binding is still possible (i.e. is still in the candidate set), a new dependency is created and added to the collection. In the case of contravariant matches for parameter types, a new DisjunctiveDependency is created whose premises are each of the possible bindings (i.e. one binding for the provided parameter type and for each of its subtypes). This evaluation process is shown in pseudo-code in Figure 3.10.

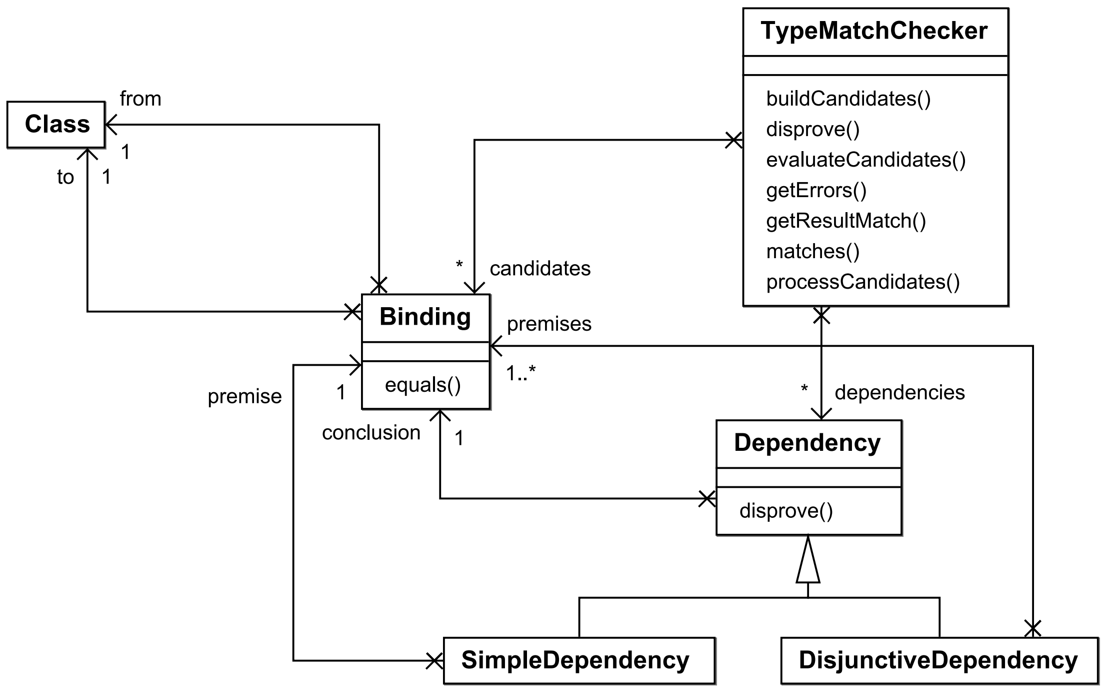


FIG. 3.8 – Structure of model type matcher

```
operation TypeMatchChecker.buildCandidates()
  for each Class $C\_r$ $\in$ requiredTypes
    for each Class $C\_p$ $\in$ providedTypes
      candidates.add ( new Binding($C_{\mathrm{r}}$, $C_{\mathrm{p}}$) )
```

FIG. 3.9 – Constructing a list of candidate bindings

The evaluation of a binding may fail if it is immediately impossible for a dependency to be satisfied, e.g. if the premise is not present in the list of candidates. The binding may also fail if the binding violates one of the local constraints from above. In either of these cases, a disproof process is carried out. In this disproof process, the binding is removed from the list of candidates, and all dependencies are inspected to determine whether their premise included the failed binding. If so, and the dependency is broken, the disproof process is recursively applied for the conclusion of the dependency. The disproof mechanism is shown in pseudocode in Figure 3.11.

The final processing step is to cull ambiguous matches which are related by inheritance. If a required object type is matched by multiple provided types, but there is a single matching type which is a supertype of all the others, then the subtypes are removed. This is shown in Figure 3.12.

```
operation TypeMatchChecker.processCandidates()
  for each Binding $cand$ $\in$ candidates
    boolean $hasFailed$ = true
    if not $cand$.to.localMatchFor($cand$.from)
      $hasFailed$ := true

    for each Property $p$ $\in$ $cand$.from.getAllProperties()
      Property $matchProp$ := $cand$.to.getPropertyByName($p$.name)
      Binding $newBinding$ := new Binding ($p$.type, $matchProp$.type)
      if candidates contains $newBinding$
         dependencies.add (new Dependency($newBinding$ $\rightarrow$ $cand$) )
      else
         $hasFailed$ := true

    for each Operation $o$ $\in$ $cand$.from.getAllOperations()
      Operation $matchOp$ := $cand$.to.getOperationByName($o$.name)
      Binding $newBinding$ := new Binding ($o$.type, $matchOp$.type)
      if $newBinding$ $\in$ candidates
         dependencies.add (new Dependency($newBinding$ $\rightarrow$ $cand$) )
      else
         $hasFailed$ := true
      for each Parameter $param_i$ $\in$ $o$.ownedParameter
        Parameter $matchParam_i$ := $matchOp$.ownedParameter
        Set⟨ Class ⟩ matchParamSubtypes :=
            getAllSubtypes($matchParam_i$.type)
        DisjunctiveDependency $newDependency$ :=
            new DisjunctiveDependency ( $\rightarrow$ $cand$ )
        for each Class $matchParamSubtype$ $\in$ $matchParamSubtypes$
          Binding $newBinding$ :=
              new Binding ($param_i$.type $\rightarrow$ $matchParamSubtype$)
          if $newBinding$ $\in$ candidates
            $newDependency$.premises.add ($newBinding$)
        if $newDependency$.premises.isEmpty()
          $hasFailed$ := true
    if $hasFailed$
      disprove($cand$)
```

FIG. 3.10 – Evaluating candidate matches

```
operation TypeMatchChecker.disprove($bind$ : Binding)
  for each Dependency $dep$ $\in$ dependencies
    if disprove($bind$, $dep$)
      if $dep$.conclusion $\in$ candidates
        eliminate($dep$.conclusion)
      dependencies.remove($dep$)
  candidates.remove($bind$)

operation SimpleDependency.disprove($bind$ : Binding) : Boolean
  result := premise.equals($bind$)

operation DisjunctiveDependency.disprove($bind$ : Binding) : Boolean
  if $bind$ $\in$ premises
    premises.remove($bind$)
  result := premises.isEmpty()
```

FIG. 3.11 – Recursively disproving candidate matches

```
operation TypeMatchChecker.evaluateCandidates()
  for each Class $required$ $\in$ requiredTypes
    if $\exists$ Class $provided1$, $provided2$ |
        Binding ($required$ $\rightarrow$ $provided1$) $\in$ candidates
        and Binding ($required$ $\rightarrow$ $provided2$) $\in$ candidates
        and $provided1$ $<:$ $provided2$
      candidates.remove( Binding ($required$ $\rightarrow$ $provided1$)
```

FIG. 3.12 – Removing matches related by inheritance

If, after this processing is completed, there remains exactly one binding for each required object type, then the model type match is successful. The relation given by the remaining bindings is stored on the model of the Kermeta program, in the *Class.virtualTypeBinding* property, to avoid recalculation.

The algorithm is based on assessing all possible type matches based on a cross product of the provided and required object types. Since each of these processed exactly once, the matching algorithm is $O(n^2)$ complex, which is reasonably costly. However, since model types even for large metamodels rarely exceed a hundred types, the complexity is manageable.

3.3.5 Changes to the Kermeta Standard Library

A model type in Kermeta is a collection of type definitions : enumerations, type aliases, and most importantly class definitions. Model types are used monomorphically for the classification and also creation of models, and as bounds for model type parameters using model type matching. Model types also offer the following operations :

new() : Model Creates a new model. Since Kermeta does not support dependent types, in the case where the model type is a literal, the type of the new model is optimized by the type-checker to be the model type in question.

isModelTypeOf(m : Model) : Boolean Returns true if the provided model is a model of this model type.

A model in Kermeta is any term whose type is a model type. Conceptually, a model is a bag of objects, supporting the following operations :

add(o : Object) Adds the object into the model. This method is statically checked to ensure the type of the parameter is present in the model type of the model.

remove(o : Object) Remove the object from the model, if it is present.

contents() : Collection<Object> Returns a collection containing the objects in the model.

filter(t : Type) : Collection<Object> Returns all elements of the model that are instances of the provided type. The type checker requires that the parameter be in the form of a literal (so that the type is known statically) and that the type is present in the model type of the model. Since Kermeta's type system does not support dependent types, the return type is optimized by the type-checker to be a collection of the type passed as a parameter .

3.3.6 Changes to the Kermeta Runtime

There are a very few changes in the Kermeta runtime as a result of model types.

The runtime representation of a model type is simply a link to their included types. Models at runtime maintain a link to their contained objects, as well as a link to the model type from which they were created.

As a result of the introduction of package merge, type references in Kermeta code may be rebound in a class that acquires the code via a merge. To enable this, the runtime representation of a class includes bindings for any types that are rebound in this way.

3.3.7 Runtime access to Type Parameters

Another significant change made while implementing model types in Kermeta was to provide the programmer with runtime access to type parameters. This change is not directly required for model typing, but is useful in that model types encourage heavier use of type parameters.

When writing a model transformation, it is frequently necessary to create new objects and add them into a model. When writing against a specific model type, this is not problematic, since one may simply instantiate the class required.

However, in the case of a transformation which has been written to be generic to a set of model types, it is not so simple. There is no class available to instantiate ; the programmer has access only to a virtual type, in other words a type parameter, and type parameters do not provide for instantiation. Before this change, type parameters were used only as types, and not as terms, in the language.

To alleviate this, an incidental change made to Kermeta during the implementation of model typing was to provide the programmer with runtime access to type parameters.

When the programmer uses a normal variable, he/she has access to all of the features that are statically known to be available on the object. Initially, however, type parameters had meaning only as a type-checking convenience. For example, although all types provide an $isInstance()$ operation, the programmer did not have access to the $isInstance()$ operation on a type parameter. This is despite the fact that the parameter is sure to have been filled some type which provides the method.

If one considers only $ObjectTypeVariables$, then it is certain that generic code at runtime will be executed in the context of a class whose type parameters (or $VirtualTypes$) are bound to some instance of $Class$. Therefore, it seems reasonable to provide access to the operations available on $Class$. The same applies for $ModelTypeVariables$ and the operations available on $ModelTypes$.

One problem with this is that it makes certain static checks impossible. Specifically, instantiation involves a static check that the class being instantiated is not abstract (nor semantically abstract). This is not possible in the case of a type parameter, since the exact class to be instantiated, and thus whether or not it is abstract, is not statically known.

The alternative to providing runtime access to type parameters is to insist that the model type provide factory methods for each type which the programmer wishes to instantiate. This is an imposition to the programmer, particularly when the cost, the loss of static certainty that abstract classes will not be instantiated, may be limited to those cases where type parameters are instantiated. By contrast, the potential benefit also includes access by type parameters to the full range of reflective interfaces, which is potentially a powerful tool for metaprogramming.

The strategy taken, therefore, was to allow access on type parameters to those operators that might be statically known (based on the bounds constraint) to be available on the type to which they would bind at runtime. The check for abstract class instantiation remains in those cases where the target is statically known (type literals), and is implemented as a runtime check for those cases (type parameters) where it is not.

3.4 Example : The State Machine Workbench

This section presents a workbench for the treatment of state machines, in order to demonstrate model types in Kermeta by showing their use in a small example project. The majority of the code derives from a sample from the Kermeta project, and includes code presented in [MFV$^+$05]. It has been modified to use model types, in order that it should work for the different variants of state machines presented in Section 2.2.1.

The workbench consists of model type definitions, as well as operators for

- The import and export of state machine models to and from files (models are stored using the XMI format)
- The export of models to the *dot* format.
- The simulation of the state machines based on an event stream provided by the user through the command-line.
- Determinization of state machine models to remove non-deterministic transitions.
- Minimization of state machine models to produce machines with the minimal number of states such that the machine behaves in the same way.

In Section 3.4.1 we show the definition of the different state machine variants as packages and model types. Section 3.4.2 then shows a simple serializer for state machines that is reusable for the variants according to model type matching. A number of more useful operators may be added to the state machine workbench in a similar way, and these are detailed in Annex A.

A more in-depth discussion of using model types in Kermeta for the construction and extension of languages and operators upon them is presented in Chapter 4.

3.4.1 Model Types for State Machine variants

The following figures show the definitions and model types for the five different state machine variants described in Section 2.2.1.

Figures 3.3, 3.4, and 3.5 show the definitions and model types for basic state

```
package basic_fsm {

  class FSM {
    attribute alphabet : set String[0..*]
    attribute ownedState : set State[0..*]#owningFSM
    attribute ownedTransition : set Transition[0..*]
    reference initialState : State
    reference finalState : set State[0..*]
  }

  class State {
    attribute name : String
    reference owningFSM : FSM#ownedState
    reference incomingTransition : Transition[0..*]#target
    reference outgoingTransition : Transition[0..*]#source
  }

  class Transition {
    attribute input : String
    attribute output : String
    reference source : State#outgoingTransition
    reference target : State#incomingTransition
  }
}

modeltype basic_fsm_type { basic_fsm::FSM, basic_fsm::State,
   basic_fsm::Transition }

abstract class FSMException {}
class NonDeterminism inherits FSMException {}
class NoTransition inherits FSMException {}
class NoInitialStateException inherits FSMException{}
```

Listing 3.3 – Model Type : Basic State Machines

machines, state machines with mandatory start states, and state machines multiple start states, respectively.

The definitions for composite state machines and state machines with final states are defined using merges of basic state machines. These, and the associated model types, are shown in Figures 3.6 and 3.7 respectively.

3.4.2 State Machine Operators

Having defined the model types corresponding to the different state machine variants, operators for state machine models may now be defined as classes parameterized by a model type parameter.

```
package mandstart_fsm {

  class FSM {
    attribute alphabet : set String[0..*]
    attribute ownedState : oset State[0..*]#owningFSM
    attribute ownedTransition : set Transition[0..*]
    reference initialState : State[1]
    reference finalState : set State[0..*]
  }

  class State {
    attribute name : String
    reference owningFSM : FSM#ownedState
    reference incomingTransition : Transition[0..*]#target
    reference outgoingTransition : Transition[0..*]#source
  }

  class Transition {
    attribute input : String
    attribute output : String
    reference source : State#outgoingTransition
    reference target : State#incomingTransition
  }
}

modeltype mandstart_fsm_type {mandstart_fsm::FSM,
   mandstart_fsm::State, mandstart_fsm::Transition}
```

Listing 3.4 – Model Type : Mandatory Start State

```
package multstart_fsm {

  class FSM {
    attribute alphabet : set String[0..*]
    attribute ownedState : set State[0..*]#owningFSM
    attribute ownedTransition : set Transition[0..*]
    reference initialState : set State[0..*]
    reference finalState : set State[0..*]
  }

  class State {
    attribute name : String
    reference owningFSM : FSM#ownedState
    reference incomingTransition : Transition[0..*]#target
    reference outgoingTransition : Transition[0..*]#source
  }

  class Transition {
    attribute input : String
    attribute output : String
    reference source : State#outgoingTransition
    reference target : State#incomingTransition
  }
}

modeltype multstart_fsm_type { multstart_fsm::FSM,
   multstart_fsm::State, multstart_fsm::Transition }
```

Listing 3.5 – Model Type : Multiple Start States

```
package composite_fsm merges basic_fsm {
  class FSM inherits State { }
}

modeltype composite_fsm_type { composite_fsm::FSM,
   composite_fsm::State, composite_fsm::Transition }
```

Listing 3.6 – Model Type : Composite State Machines

```
package finalstates_fsm merges basic_fsm {
  class FinalState inherits State { }
}

modeltype finalstates_fsm_type { finalstates_fsm::FSM,
   finalstates_fsm::State, finalstates_fsm::Transition,
   finalstates_fsm::FinalState }
```

Listing 3.7 – Model Type : Final States

```
package fsm;

require "../kermeta/basic_fsm_type.kmt"
using kermeta::standard
using kermeta::utils
using kermeta::exceptions

class Serializer<MT : basic_fsm_type> {
  operation printFSM(fsm : MT::FSM) is do
    fsm.ownedState.each
    { s |
      stdio.writeln("State : " + s.name)
      s.outgoingTransition.each { t |
        var outputText : String
        if( t.output != void and t.output != "" )
          then outputText := t.output
          else outputText := "NC"
        end
        stdio.writeln("  Transition : " + t.source.name + "-("
            + t.input + "/" + outputText + ")->" + t.target.
            name)
      }
    }
  end
}
```

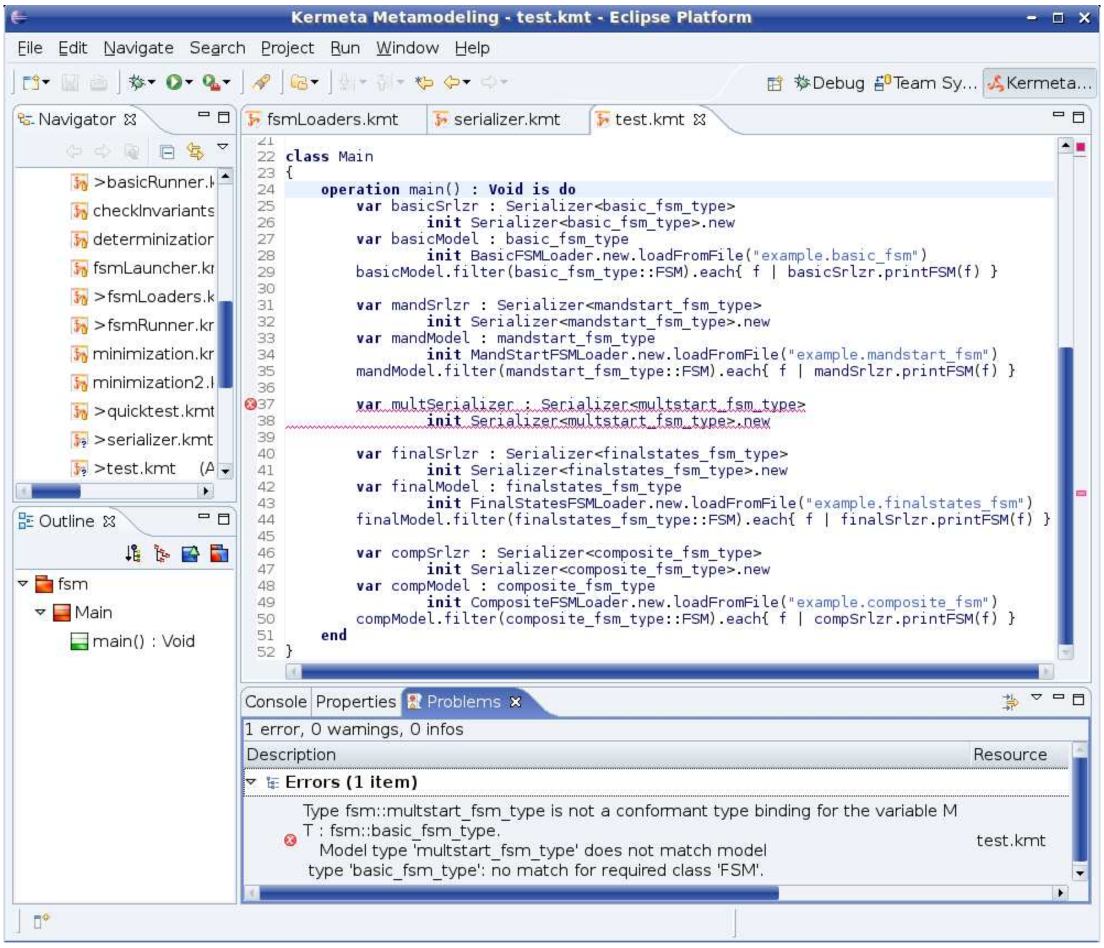
Listing 3.8 – State Machine Serialization

To illustrate this, Figure 3.8 shows a simple serializer for state machine models.

This class may then be reused to serialize either basic, mandatory-start, final-state, or composite state machines, as shown in the screen capture in Figure 3.13. As explained in Chapter 2, state machines with multiple start states are not a model-type match for basic state machines, and this is detected by the type-checker, as is shown by the error in the figure.

If we consider an example state machine such as the one in Figure 3.14, encoded using each of the four different metamodels, each of the four different uses of the serializer will work to produce the output shown in Listing 3.9. Obviously, final states are not shown, since their display would require an extension to the serializer. Composite states will be shown as separate state machines, but not as nested composites.

A number of other, more complex operators may be found in Annex A.



```
class Main
{
    operation main() : Void is do
        var basicSrlzr : Serializer<basic_fsm_type>
                init Serializer<basic_fsm_type>.new
        var basicModel : basic_fsm_type
                init BasicFSMLoader.new.loadFromFile("example.basic_fsm")
        basicModel.filter(basic_fsm_type::FSM).each{ f | basicSrlzr.printFSM(f) }

        var mandSrlzr : Serializer<mandstart_fsm_type>
                init Serializer<mandstart_fsm_type>.new
        var mandModel : mandstart_fsm_type
                init MandStartFSMLoader.new.loadFromFile("example.mandstart_fsm")
        mandModel.filter(mandstart_fsm_type::FSM).each{ f | mandSrlzr.printFSM(f) }

        var multSerializer : Serializer<multstart_fsm_type>
                init Serializer<multstart_fsm_type>.new

        var finalSrlzr : Serializer<finalstates_fsm_type>
                init Serializer<finalstates_fsm_type>.new
        var finalModel : finalstates_fsm_type
                init FinalStatesFSMLoader.new.loadFromFile("example.finalstates_fsm")
        finalModel.filter(finalstates_fsm_type::FSM).each{ f | finalSrlzr.printFSM(f) }

        var compSrlzr : Serializer<composite_fsm_type>
                init Serializer<composite_fsm_type>.new
        var compModel : composite_fsm_type
                init CompositeFSMLoader.new.loadFromFile("example.composite_fsm")
        compModel.filter(composite_fsm_type::FSM).each{ f | compSrlzr.printFSM(f) }
    end
}
```

```
1 error, 0 warnings, 0 infos
Description                                                              Resource
Errors (1 item)
    Type fsm::multstart_fsm_type is not a conformant type binding for the variable M
    T : fsm::basic_fsm_type.
      Model type 'multstart_fsm_type' does not match model                test.kmt
    type 'basic_fsm_type': no match for required class 'FSM'.
```

Fig. 3.13 – Multiple-start state machines do not match basic state machines

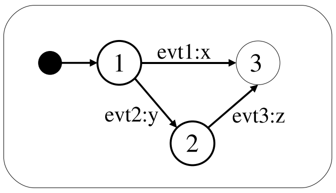


FIG. 3.14 – A sample state machine for serialization

```
State : 1
  Transition -(evt1/x)->3
  Transition -(evt2/y)->2
State : 2
  Transition -(evt3/z)->3
State : 3
```

Listing 3.9 – Output of state machine serialization

3.5 Conclusion

The process of building support for model types into Kermeta involves an expansion of the principles of model typing as provided in Chapter 2. This includes securing the model abstraction mechanism using parametric polymorphism rather than subtyping, providing a more sophisticated relation for model type matching, and supporting better code reuse when defining metamodels.

The addition of model types to Kermeta provides the language with first-class notions of models and model types, and allows the developer to build MDE tools for manipulating models that are robust to changes and variations in their metamodels. This is shown in the state machine workbench case study, and has been confirmed by early user feedback.

Chapitre 4

Model Types and The Expression Problem

4.1 Introduction

Managing simultaneous, unanticipated evolutions of a software system is one of the key issues of software engineering. It has echoes in many different software engineering communities, including Software Configuration Management, Software Product Lines, Design Patterns, Aspect-Oriented Software Development, Model-Driven Engineering and Advanced Programming Languages. In this context, the *expression problem* plays a similar role to *drosophila* [Mor10] for geneticists : a realistic yet simple example of a central problem of the domain. Originally named as such by Phil Wadler [Wad98], the *expression problem* boils down to the parallel extension of both a hierarchy of data types (e.g. ; an Exp class that is going to be subclassed with Lit, Plus, etc., see Figure 4.1) and a set of operations for processing them (e.g. ; eval, show, etc.). Odersky and Zenger [ZO05] list several constraints of this problem :

- Extensibility in both dimensions (data type and operations)
- Strong static type safety
- No modification or duplication of existing code
- Separate compilation
- Independent extensibility

This problem has been used as a benchmark to evaluate a wide variety of research ideas, ranging from design patterns and product-line engineering to aspect orientation and new language design [LHBC05]. In this chapter, we look at it from the point of view of Model-Driven Engineering, and more specifically, Model Typing, as an illustration of the interest of model types in addressing reuse in MDE. The main perspective offered in this chapter is that MDE allows us to reason about the expression problem in a *model-based* way, whereas most of the previous approaches were *class-based*. This is less a revolution than a slight shift of point of view. Model types provide the user with an environment capable of solutions to the expression problem similar to those proposed elsewhere, but using a lightweight taxonomy that more smoothly blends into existing practices in MDE.

In Section 4.2, we reformulate the Expression Problem with Model Types as a problem of operations on languages, using two different approaches, characterised as data-centric and operator-centric [Tor04]. These solutions are presented using the Kermeta language with its support for model types, as described in Chapter 3. Section 4.3 provides an assessment of our two solutions with respect to Odersky's and Zenger's criteria [ZO05] as well as to a few others, including transparency to unplanned extensions. Section 4.4 discusses related works and how they compare to the solutions in Kermeta.

4.2 Model Types And The Expression Problem

As related briefly above, the expression problem is a presentation of the difficulty associated with defining simultaneous extensions to a software system defined as a language and a set of operators on that language. Although it has existed for longer, the expression problem was named as such in an email by Philip Wadler to the java-genericity mailing list [Wad98]. Since then, a number of papers have appeared with solutions, which differ in method. Indeed, the problem is not one to be solved canonically, but a sandbox for demonstrating the features of programming languages and systems. The criteria by which solutions to the expression may be judged are discussed in Section 4.3.

The expression problem as it is usually presented, and as we shall treat it, involves a simple arithmetic expression language. Structurally, the language consists of expressions, which initially may be only numeric literals, but which in a later extension may include binary additions. There are also two operators given : an evaluator for expressions, and a pretty-printer. These operators may be defined against any of the structural languages. For example, Figure 4.1 shows an extension hierarchy with the base language without operators (*Base*) extended structurally for the provision of addition expressions (*BasePlus*), for the addition of evaluator operators for each of these (*BaseEval* and *BasePlusEval*, and a pretty-printer for the extension (*BasePlusShow*).

From an MDE perspective, the problem may be thought of as the definition of the metamodels for two modelling languages, *Base* and *BasePlus*, followed by the definition of transformations (or programs) for evaluation and pretty-printing. Although MDE tends to distinguish more strongly between data extension and operator extension, the problem is the same.

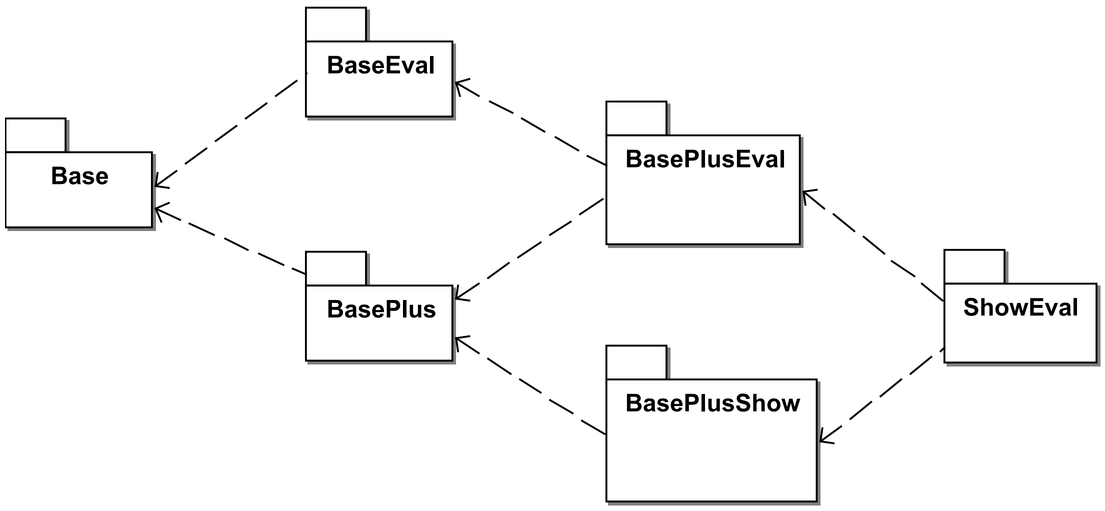


FIG. 4.1 – A hierarchy of language extensions

This section presents two approaches to the expression problem using model types. According to the classification of approaches presented in [Tor04], the two examples represent respectively data-centric and operator-centric approaches to the problem.

4.2.1 A Data-Centric Approach

Listing 4.1 shows the structures for a data-centric solution, as written in Kermeta. Specifically, *Base* represents a structural metamodel for an expression language with numeric literals. *BasePlus* is an extension of this metamodel to add the concept of an addition expression.

```
package Base {
  abstract class Exp { }
  class Num inherits Exp {
    attribute val : Integer
  }
}

modeltype BaseT { Base::Exp, Base::Num }

package BasePlus merges Base {
  class Plus inherits Exp {
    reference left, right : Exp
  }
}

modeltype BasePlusT
```

```
  { BasePlus::Exp, BasePlus::Num, BasePlus::Plus}
```

Listing 4.1 – Data-centric approach : structures

Clearly, these metamodels are very simple. In order to add support for evaluation and pretty-print operators, we again extend the metamodels to add them. Listing 4.2 shows three such extensions ; *BaseEval* and *BasePlusEval* are incrementally defined evaluators for *Base* and *BasePlus* respectively, and *BasePlusShow* is a pretty-printer for *BasePlus*. These extensions differ slightly from that of *Base* by *BasePlus*, in that new features are ”added” to classes.

```
package BaseEval merges Base {
  class Exp {
    operation eval() : Integer is abstract
  }
  class Num {
    operation eval() : Integer is do
      result := val
    end
  }
}

modeltype BaseEvalT { BaseEval::Exp, BaseEval::Num }

package BasePlusEval merges BasePlus, BaseEval {
  class Plus {
    operation eval() : Integer is do
      result := left.eval + right.eval
    end
  }
}

modeltype BasePlusEvalT
  { BasePlusEval::Exp, BasePlusEval::Num, BasePlusEval::Plus }

package BasePlusShow merges BasePlus {
  class Exp {
    operation show() : String is abstract
  }
  class Num {
    operation show() : String is do
      result := val.toString
    end
  }
  class Plus {
    operation show() : String is do
      result := left.show + " + " + right.show
    end
  }
}
```

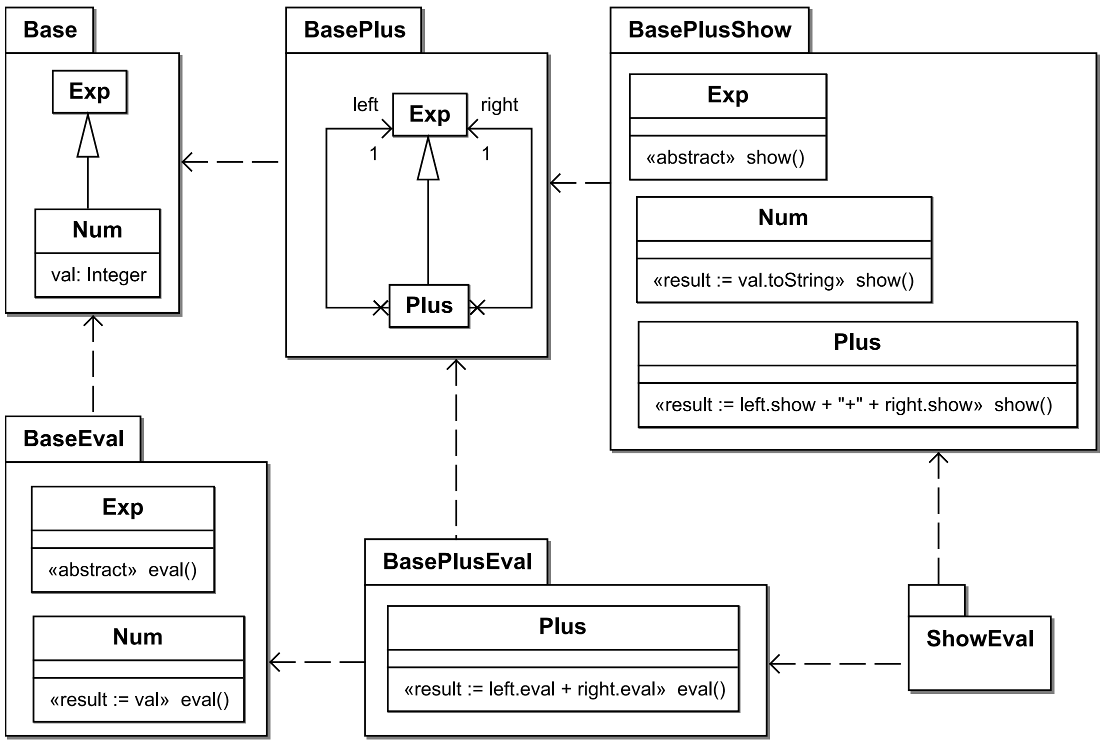


FIG. 4.2 – Diagram of data-centric solution

```

modeltype BasePlusShowT
  { BasePlusShow::Exp, BasePlusShow::Num, BasePlusShow::Plus }

package ShowEval merges BasePlusEval, BasePlusShow { }

modeltype ShowEvalT
  { ShowEval::Exp, ShowEval::Num, ShowEval::Plus }
```

Listing 4.2 – Data-centric approach : operators

Interestingly, models and model types figure only a little in this solution, which is based principally on merging the packages of successive variants. As can be seen with the *ShowEval* extension, the extensions written are also combinable in a diamond pattern, in order to provided ”independent extensibility”, as discussed in [ZO05].

For an alternative notation, Figure 4.2 shows the data-centric solution graphically using a UML class diagram notation[1].

[1]The model types associated with each package are not shown graphically.

4.2.2 An Operator-Centric Approach

The previous solution is not very typical of MDE. Specifically, MDE has a tendency to not mix the definitions of structures (metamodels) and operators. One reason for this is its origins in distributed systems[2]. If we were designing a solution for use in a web services or other RPC environment, the solution would not work, since a model coming from outside, although structurally conformant to our extensions, would not include the definitions of the operators we need.

To address this, we may take what Torgersen [Tor04] calls an operator-centric approach. In Listing 4.3, we show the Base and BasePlus metamodels again, but with an additional Visitor [GHJV95] class (generalized for return type), as well as corresponding generic *accept* operations on each of the concrete subclasses of *Exp*.

```
package Base {
  abstract class Exp {
    operation accept<R>(v : Visitor<R>) : R is abstract
  }
  class Num inherits Exp {
    attribute val : Integer
    method accept<R>(v : Visitor<R>) : R is do
      result := v.visitNum(self)
    end
  }
  abstract class Visitor<R> {
    operation visitNum(n : Num) : R is abstract
  }
}

modeltype BaseT { Base::Exp, Base::Num, Base::Visitor }

package BasePlus merges Base {
  class Plus inherits Exp {
    reference left, right : Exp
    method accept<R>(v : Visitor<R>) : R is do
      result := v.visitPlus(self)
    end
  }
  abstract class Visitor<R> {
    operation visitPlus(n : Plus) : R is abstract
  }
}

modeltype BasePlusT
```

[2]The MOF 1.x specifications were written as mappings of modelling language concepts to CORBA interfaces.

```
  { BasePlus::Exp, BasePlus::Num, BasePlus::Visitor }
```

Listing 4.3 – Operator-centric approach : structures

At this point, the second solution differs from the first only in that we have added a visitor pattern. The visitor class is extended in the *BasePlus* package to add a visit method for plus. As in the data-centric solution, this extension is achieved using package merge, not class inheritance, meaning that there is no subtype relationship between the two visitor classes.

This is already less concise and intuitive than the data-centric solution. However, it allows the definition of the *Eval* and *Show* operators outside of the language classes themselves, as in Listing 4.4.

```
class BaseEval<MT : BaseT> inherits MT::Visitor<Integer> {
  operation apply(e : MT::Exp) : Integer is do
    result := e.accept(this)
  done
  method visitNum(n : MT::Num) : Integer is do
    result := n.val
  end
}

class BasePlusEval<MT : BasePlusT> inherits Eval<MT>, MT::
    Visitor<Integer> {
  method visitPlus(p : MT::Plus) : Integer is do
    result := p.left.accept(self) + p.right.accept(self)
  end
}

class BasePlusShow<MT : BasePlusT> inherits MT::Visitor<String
    > {
  operation apply(e : MT::Exp) : String is do
    result := e.accept(this)
  done
  method visitNum(n : MT::Num) : String is do
    result := n.val.toString
  end
  method visitPlus(p : MT::Plus) : String is do
    result := p.left.accept(self) + " + " + p.right.accept(
        self)
  end
}
```

Listing 4.4 – Operator-centric approach : operators

The most interesting part of this solution is that the operator classes are defined as subclasses not of the $Base :: Visitor$ and $BasePlus :: Visitor$ classes, but of the $MT :: Visitor$ *virtual type parameters.*

In general, it can be dangerous to inherit from a type parameter, since the full details of the inherited type are not known when defining the class. Consider the example in Figure 4.5.

```
class C<T : D> inherits T {
  attribute foo : String
}
class D { ... }
class E inherits D {
  attribute foo : Integer
}
```

Listing 4.5 – Inheriting from a type parameter is potentially problematic

On its own, there is no problem with the class C. However, a concretization of the class as a type $C\langle E\rangle$ will obviously result in a conflict between the foo attributes in E and C. So, in addition to checking that our prospective actual type parameter E satisfies the bounds constraint (D), it needs to be checked for any conflicts arising through playing the role of superclass. The nature of this conflict detection is not essentially different from that conducted when detecting conflicts between multiple superclasses. The main difference is that it is performed not on the generic class definition, but on its use.

Using this pattern actually achieves nothing from the point of view of $BaseEval$. Since $BasePlus$ is a model-type match for $Base$, and since there are no conflicts between features in $BasePlus :: Visitor$ and $BaseEval$, it would be possible to create a type $BaseEval\langle BasePlus\rangle$. However, $BaseEval\langle BasePlus\rangle$ effectively inherits from $BasePlus :: Visitor$ but provides no implementation of the abstract method $visitPlus()$.

To recall from Section 3.1.1, Kermeta includes a notion called *semantically abstract* classes. A semantically abstract class is one that, although not declared as abstract, does not implement some necessary abstract operations. These classes are accepted by the type checker, but an attempt to instantiate them is not. Therefore, a type $BaseEval\langle BasePlus\rangle$ would pass the type checker, but we would be unable to instantiate it since, by not implementing $visitPlus()$, it is semantically abstract. In this way, a $BaseEval$ processor is not able to treat $BasePlus$ models.

When we come to define $BasePlusEval$, the use of model type parameters becomes useful. A direct inheritance of the visitor classes would have resulted in a mix of $Base$ and $BasePlus$ object types in the resultant $BasePlusEval$ signature. Being parameterized by a model type allows these types to be re-bound, however, uniformly to $BasePlus$ types.

To use the terminology of [Tor04], this second solution offers code-level reuse, since there is no non-trivial code duplicated between $BaseEval$ and $BasePlusEval$. It does not allow object-level reuse, though. That is, a $Base$ model cannot be evaluated or printed by the operators written against $BasePlus$.

Also, unlike the earlier data-centric solution, this operator-centric solution does not support recombination of the *PlusEval* and *ShowPlus* operators, because a class inheriting from the two classes would result in a conflict between the visit methods, which differ on return type.

4.3 Evaluation

An interesting aspect of the expression problem is that, increasingly, it is not a problem that is solved or not solved. Rather, what is important is the nature of the solution, in terms of its characteristics with respect to certain criteria. In this section we evaluate the two solutions, both individually and collectively as a reflection of the fit of model types to the problem.

4.3.1 Basic Criteria

Several basic criteria for assessing solutions to the expression problem have been established in [Wad98] (and well clarified in both [ZO05] and [Tor04]). These are

- Extensibility in both dimensions. It should be possible to add new data variants and new processors, and to do so any number of times.
- Strong static type safety. There should be no risk of unhandled combinations of data and operations.
- No modification or duplication. An extension should not require the modification of code in the extended system. Furthermore, the duplication of non-trivial code should be avoided.
- Separate compilation. The addition of an extension should not require the re-type-checking or recompilation of existing code.

Clearly, both the data-centric and operator-centric solutions above allow for the addition of both data and operator extensions, using package merges. This is done without modification of existing code, and without need for recompilation. Although model types do not support extension, their definitions are single lines of code, and thus not overly burdensome. Although we do not have a full formal type system for Kermeta, we believe that both of our solutions are statically type-safe.

Although we do not demonstrate further extensions, there is no restriction either for order of application or for number of extensions with the data-centric approach, provided that, for example, data extensions provide implementations for abstract methods. The operator-centric approach may also be extended in a linear fashion, subject to the conditions described in the next section.

4.3.2 Reuse and extensibility

In [ZO05], the authors add the criterion that the solutions be *independently extensible*. This means that independent extensions of a system should be recombinable to form a single extending form supporting both extensions.

As shown with the "diamond" merge, the recombination of extensions in our data-centric approach is not only possible, but very simple. By contrast, the operator-centric approach is found wanting in this criterion. Although data variant extensions may still be recombined (since they use the same extension technique as the data-centric approach), this is not true for the addition of new operators, since it is based on the visitor pattern. (The various visit methods of *BasePlusEval* and *BasePlusShow* would conflict in an evaluator inheriting from both).

As mentioned, one drawback of our data-centric approach is that it requires models to have been created using the types with extensions already added. This is part of what [Tor04] calls *object-level extensibility*. In general, this refers to the ability of a given extension to treat models constructed from previous or less-featured versions of the language.

As we discuss, our data-centric approach does not provide object-level extensibility. The operator-centric approach is more interesting. Clearly, *BasePlusEval* has been designed to support the evaluation of *BasePlus* models that do not have knowledge of evaluation. However, as we described, *BasePlusEval* is not capable of evaluating *Base* expressions, since $Base :: Visitor$ would not be an acceptable superclass (*Base* does not satisfy the bounds constraint *BasePlus* on the model type parameter *MT*). So, this solution provides object-level extensibility with respect to operator addition, but not with respect to data extension.

Another criterion of reuse is the necessity for the developer to anticipate the extension of the original system, and prepare the design of the system appropriately. The operator-centric solution we present requires the application of a visitor pattern in the initial system in order to support extension (specifically, in order to support the addition of new operators). Had the user not anticipated extension of the original system, it is not certain that they would have used such a pattern. A small amount of planning is required.

By contrast, the data-centric approach requires little or no anticipation by the developer. The initial design of the metamodels would not differ had the developer not expected the system to be extended. From this perspective, the solution is robust to evolution in that it does not depend on the user preparing the system for evolution.

4.3.3 User considerations

A third and important axis of evaluation is that of the user. From the point of view of the user, does the solution represent a simple and understandable representation of the problem they are confronting ? Does the solution require them to learn new concepts of the language in which they are solving it ?

Obviously, such judgements are highly subjective. There might often be a trade-off, for example, between solutions that are succinct but use advanced language features, and those that are more verbose and complex but use simpler constructs. The choice as to which is more appropriate will vary greatly depending on the developer using it.

The solutions we present here represent simple encapsulations of the problem. The data-centric solution, in particular, is a reasonably natural representation of the systems being modelled. The new language concepts that we introduce to permit this are essentially those of model and of model type. It is our hope that this paradigm is intuitive to developers who are already familiar with software modelling, and in particular those familiar with the ideas of model-driven engineering.

4.4 Related Work

The expression problem has been the subject of a number of recent works. The solutions that these propose vary with respect to both the criteria to which they aspire and the nature of the solutions they propose.

Wadler's initial posing of the problem in [Wad98] included a solution built using the visitor pattern in combination with classes parameterized by themselves. However, Wadler's solution contained a subtle type error and was later retracted.

This error is, to a certain extent, resolved by Torgersen in [Tor04], who proposes four solutions based on fairly elaborate use of parameterized types. The fourth, and most elaborate, of these, proposes a framework for a parameterized solution that offers advantages for both data-centric and operator-centric perspectives. Although they are complex, Torgersen's latter solutions also enjoy object-level reuse, meaning that outdated models may still be evaluated by extended versions of the system.

In [Bru03], Bruce presents a survey of approaches to the expression problem, using functional languages, the interpreter pattern, the visitor pattern, and a solution based on LOOM's type groups. The type groups solution is based on the work in [BV99], from which many of the ideas behind model types were drawn in Chapter 2, but nonetheless the solution differs greatly from those shown here in Kermeta. As pointed out in [ISV05], LOOM's inheritance between corresponding types in different type groups is considered to invoke subtyping, which results in the necessity of using exact types.

Igarashi et al, in [ISV05], present a formalization of lightweight family polymorphism, originally proposed but not formalised by Ernst in [Ern01], which shares the aims of type groups in wanting to handle the parallel specialization of mutually recursive types. Igarashi uses a deep extension relationship similar to our model type merging, but requires that types be explicitly notated as virtual (using ”relative path types”), whereas we treat this as the default usage for any types also present in the model type. Also, the class structure is used as the containing structure for type members, which allows for recursive structures. By contrast, we use the distinct model type construct, which rules out recursive containment, but allows us to use two distinct extension mechanisms : package merging, which does not result in subtyping, and class inheritance, which does (to an extent).

Zenger & Odersky propose a number of solutions in [ZO05] using the Scala language. Their first, data-centric, solution, based on deep mixin composition, bears a strong resemblance to our data-centric solution. The major difference is that their abstract types must be initialized by the programmer, whereas with model types this is done implicitly by the model type. They also provide an operator-centric solution which uses self-type references to avoid some of the type parameterization seen in Torgersen's solutions. On the whole, Scala's expressive type system makes for a wide range of powerful solutions, albeit that these depend on the programmer coming to grips with some fairly advanced typing concepts.

In [LHBC05], the authors approach the expression problem from the perspective of product families, treating the addition of data variants and operators as features. They compare five solutions based on different, even disparate, technologies, from aspect-oriented approaches such as AspectJ, to programming languages such as Scala, to more product-family-specific techniques such as the AHEAD system. They find none of these solutions wholly satisfying from a product-family perspective, and discuss the general problem from the point of view of a feature composition algebra.

One of the significant differences between the application of model typing here and that presented as examples in Chapters 2 and 3 is the use of package merge, as described in Section 3.2.5 to improve the reuse of class definitions.

There are a number of similar techniques for the simultaneous extension of interrelated object type definitions. In particular, of the above works, Bruce [BV99, Bru03], Igarashi [ISV05], and Odersky [ZO05] all define operators for this purpose. Of these, package merging most closely resembles Igarashi's family extension.

From the domain of MDE, the most recent version of UML [Obj05c] (and by inclusion that of CMOF) also introduces the notion of a package merge. This relationship, which is used extensively to construct the UML and MOF metamodels themselves, serves a similar goal to model type merge, and is indeed the origin of the name. Package merge has changed considerably during the UML 2 standardisation process, and its definition, though greatly improved, remains informal and incomplete. Like most of UML and MOF, the specification of the package merge relationship does not discuss its implications for type checking. We believe that the package merge will induce matching between model types (subject to the conditions in Section 3.2.5). Whether this is the case for UML package merge is to be investigated.

4.5 Conclusion

Since the expression problem is all about the definition of languages and operators upon them, it is of central of interest to MDE. Up to now however, relatively little consideration has been given in MDE to the definition of these languages and operators with consideration for extensibility or reuse based on rigorous theories.

In this chapter we have reformulated the expression problem in a MDE context, and we have proposed the notion of *model type* (i.e. using a group of classes as a type) to deal with independent evolutions. We have shown that model types present to the user an environment built on theoretical backgrounds and capable of solutions similar to those proposed elsewhere, but using a lightweight taxonomy familiar to existing MDE users. Initial feedback from users suggests that it effectively addresses reuse issues among meta-models and their operational semantics in an intuitive way.

Chapitre 5

Conclusion and Perspectives

Model-driven engineering is a developing field. Research into the theories, techniques, and tools for the various parts that make up a model driven system - models, transformations, injectors/extractors - is active, and is seeing uptake in industrial contexts. However, as MDE progresses, it will face the challenges that characterize software engineering such as managing scalability, reliability and of particular interest to this thesis, reuse and evolution.

In order to address these, MDE as a field must develop mechanisms for managing the reuse and integration of its component pieces in a manageable and sound way that ensures that systems do not become brittle or degrade.

To address this, this thesis presents a notion of model typing, which applies and adapts ideas from the domain of type systems to the specific structures and problems of MDE. Model types allow models to be represented as first-class, typed entities. Furthermore, it allows the artefacts that manipulate models to be defined in a way that is robust to variations and evolutions of the metamodels against which they were originally written.

5.1 Summary and Conclusion

Chapter 1 presents a summary of the state of the art in model-driven engineering, and a survey of existing approaches to managing the reuse, evolution and integration of its pieces. Also, a number of works in type systems research which treat related problems are reviewed for their potential applicability to the realm of model-driven systems.

In Chapter 2, the specific needs of a model-typing system are outlined using a simple example based on state machines. The ideas of type groups and type matching are then adapted for use with models and model transformations, yielding a simple model transformation language including a number of type-checking rules. It is shown that the application of this language allows a simple state machine transformation defined using such a language permits the use of a set of variant state machine metamodels.

The principles of model typing established in Chapter 2 are then illustrated in a wider setting in Chapter 3, which describes the addition of support for model typing to the Kermeta language. This includes a consideration of the specific requirements of model types in the context of Kermeta as an imperative language for modelling and model manipulation. After a description of the design of the implementation, it is demonstrated in the form of a workbench for the manipulation of state machines based on the motivating example.

The expression problem is associated with the simultaneous extension of an expression language for the addition of new operators and new language concepts. Chapter 4 argues that this problem is endemic to model-driven engineering, and argues that model types, and specifically their implementation in Kermeta, offer an approach to managing the problem based on intuitive MDE concepts which nonetheless offers expressivity comparable to existing programming-language approaches.

Two of the great contributions of object-oriented languages were to present a more natural paradigm for describing systems, and to include in that paradigm a system for allowing code written against one object type to work polymorphically against others. Model-driven engineering extends this by shifting the focus from thinking about the objects in a system, to thinking about the systems formed by structuring these objects. Model types pursue the second axis, by allowing us to reason about and refine the types of these systems as a whole rather than reasoning about the types individually. In so doing, they permit a more flexible notion of reuse than would otherwise have been possible.

Concretely, model types offer two significant benefits for the construction of model-driven systems. Having a formalised the concept of models and model types using well-founded ideas from type systems, we can reason about the model-driven artefacts such as model transformations that make up an MDE system, and the models they manipulate, as first-class concepts in a modelling language. Secondly, by providing a relationship for model substitutability, these artefacts may be defined in a way that is robust to variation and evolution of metamodels.

5.2 Perspectives

The ideas contributed in this thesis represent a first step towards formalising the foundations of model-driven engineering in the form of type systems. There are a number of future avenues of research which are evoked.

5.2.1 A full and richer type system

Chapter 2 shows how existing type systems might be modified to support the definition and matching of model types. However, a full formal type system including proofs for soundness and completeness, as well as for decidability of relationships such as model-type matching, has not yet been defined. The beginnings of a type system for Kermeta are provided in [Fle06], and would serve as a logical starting point for such a definition. as would the type systems defined for systems supporting virtual types, such as νObj [OCRZ03], the basis of Scala.

A full definition of the type system would also permit exploration of the application of other type systems ideas to model types. For example, variance annotations on type parameters, such as those found in Scala, permit more flexible parametric polymorphism by restricting the use of type parameters to either co- or contravariant roles. Annotations for model-type parameters might restrict the use of typed models for checking or construction only, and permit more flexible reuse.

The implementation of model types in Kermeta is based on manifest type declaration. An alternative might be to infer the model types used by a model transformation based on the code or rules which make up a transformation. Such type inference has been successfully applied in many functional programming languages.

5.2.2 Constraints

One element of metamodel definition that has not been considered in this thesis is that of constraints. In addition to defining the concepts and relationships of a language, metamodels frequently also include constraints that apply on models written using the language. These constraints, which are not expressible using structural concepts, may be defined using a constraint language such as OCL [Obj06c]. A logical step would be to include such constraints in model types, and to consider them when comparing model types.

In general, the problem of comparing the constraints which exist on two model types to see whether models of one type are guaranteed to satisfy the constraints of another is a problem of constraint subsumption. Depending on the constraint language, such comparison may very well be undecidable. One solution might be to relegate the checking of these constraints to a runtime type-check, in the same manner that some multiplicity checks are presently only assessed at runtime (see Section 3.2.4.1).

One basis for the evaluation of constraints on different model types might be the work of Edwards, Jackson and Torlak in [EJT04], in which they describe a type system for determining the re-applicability of constraints to different object models based on safety and relevance deductions.

5.2.3 Application of model typing to languages other than Kermeta

A type system is, to a large extent, dependent on the operators defined by the programming language. It would be interesting to see the extent to which the principles of model typing as outlined in Chapter 2 could be applied to languages based on very different paradigms than Kermeta. This might include rule-based languages such as Tefkat [LS05] or relational QVT [Obj05b], or grammar-based languages for specifying syntax, such as Sintaks.

The ideas of model typing might equally be applied at a higher level of abstraction for type-safe assembly of models, transformations and other tools to form model-driven tool chains. This includes shared access using mechanisms such as the Model-Bus [BGS04], as well as "megamodels" for representing the relationships between models, metamodels and tools in the style of [BJV04].

5.2.4 Model Types as an Abstraction Mechanism

One of the characteristics of types systems is that in addition to providing a static verification technique, they formalise mechanisms for abstraction. Here, models and model types have been formalised as abstraction techniques for composing model-driven systems. Whether these constructs are appropriate for encapsulating the reuse boundaries when building systems is a question best answered by observation and/or empirical measurement of their application to MDE problems.

One interesting example of this might be the use of model types as a structuring concept for defining modular operational semantics for languages specified using metamodels, as presented by Mosses in [Mos04]. Common metamodel semantics could be encoded as operational semantics modules and the model types upon which they depend syntactically. These could then be applied for families with structural and semantic commonalities.

5.2.5 A Type System for Models, Models for Type Systems

As discussed in a roundtable discussion at the EW-MDA2 workshop [Ste04], types have a number of roles to play in model-driven engineering. This thesis presents a technique for reasoning about the types of models. Specific languages which are defined using model-driven means frequently have specific type systems that apply for their models, which go far beyond the constraints enforced by their metamodel and OCL constraints.

The MDE approach to such a problem would be to define a metamodel in which the language designer might describe their type system, probably based on the deductive rule system in which type systems are currently formalised. Such a metamodel might be accompanied by tools for fully or partially automating the generation of a type-checker. Equally, a language's type system model, in combination with a model of its operational semantics, might be mapped to proof-checking tools which permit the construction of proofs for soundness, completeness, and decidability of relevant aspects of the language's definition.

Annexe A

State Machine Operators

There are three operators defined for state machines : simulation, determinization and minimization. Of these, only the first two are shown here, in the interests of brevity.

Each of these operators are applicable to basic state machines, mandatory start-state machines, composite state machines, and machines with final states. State machines with multiple start states are not supported, since as described in Section 2.4.4, their model type is not a match for basic state machines.

However, the operators do not consider the specificities of these variants. For example, the simulator will not descend into composite states if provided with a composite state machine. A presentation of the problem of extending operators in parallel with extending structures is presented in Chapter 4.

A.1 Simulation

Figure A.1 shows a simple simulator for state machines. The simulator accepts signals from the user on the command line, and fires the appropriate transitions, printing the strings specified by the activated transitions. The simulator terminates if it is provided with an input for which there is either no transition or multiple possible transitions (non-determinism), or if instructed by the user.

```
package fsm;

require "../kermeta/basic_fsm_type.kmt"
using kermeta::standard
using kermeta::utils
using kermeta::exceptions

class Runner<MT : basic_fsm_type> {

  reference currentState : Hashtable<MT::FSM, MT::State>
```

```

operation run(machine : MT::FSM) : Void raises FSMException
    is
do
  currentState := Hashtable<MT::FSM, MT::State>.new
  // reset if there is no current state
  if currentState.getValue(machine) == void
  then
    self.currentState.put(machine, machine.initialState)
  end
  from var str : String init "init"
  until str == "quit"
  loop
    stdio.writeln("Current state : " + currentState.getValue
        (machine).name)
    str := stdio.read("give me a letter : ")
    if str == "quit" then
      stdio.writeln("")
      stdio.writeln("quitting ...")
    else
      if str == "print" then
        stdio.writeln("")
        Serializer<MT>.new.printFSM(machine)
      else
        stdio.writeln(str)
        stdio.writeln("stepping...")
        do
          var textRes : String
          textRes := step(currentState.getValue(machine),
              str)
          if( textRes == void or textRes == "" )
          then
            textRes := "NC"
          end

          stdio.writeln("string produced : " + textRes)

          rescue (err : ConstraintViolatedPre)
                  stdio.writeln(err.toString)
                  stdio.writeln(err.message)
                  str := "quit"
                rescue (err : ConstraintViolatedPost)
                  stdio.writeln(err.toString)
                  stdio.writeln(err.message)
                  str := "quit"

          rescue(err : NonDeterminism)
            stdio.writeln(err.toString)
            str := "quit"
            rescue(err : NoTransition)
              stdio.writeln(err.toString)
              str := "quit"
                end
        end
      end
    end
  end

  // Go to the next state
  operation step(s : MT::State, c : String) : String raises
     FSMException is

  // Declaration of the pre-condition
  pre notVoidInput is
    c != void and c != ""

  do
    // Get the valid transitions
    var validTransitions : Collection<MT::Transition>
    validTransitions := s.outgoingTransition.select { t | t.
       input.equals(c) }
    stdio.writeln("Found " + validTransitions.size.toString +
       " valid transitions for " + s.name)
    // Check if there is one and only one valid transition
    if validTransitions.empty then raise NoTransition.new end
    if validTransitions.size > 1 then raise NonDeterminism.new
        end

    // Fire the transition
    result := fire(validTransitions.one)
  end

  // Declaration of the post-condition
  post notVoidOutput is
    result != void and result != ""

  // Create a new state from self state
  operation fire(transition : MT::Transition) : String is
  do
    // update FSM current state
    currentState.put(transition.source.owningFSM, transition.
       target)
    result := transition.output
  end
}
```

Listing A.1 – State Machine Simulation

A.2 Determinization

Figure A.2 shows the code for determinizing state machines. This code has been modified from that presented in [MFV+05]. The only change has been to extract the *combinations* property into a variable local to the algorithm, so that models defined without it may be normalized, and to replace direct type references with virtual types, in order that the code be reusable for state machine variants.

```
package fsm;

require kermeta
require "../kermeta/basic_fsm_type.kmt"

using fsm
using kermeta::standard
using kermeta::utils
using kermeta::persistence

class Determinization<MT : basic_fsm_type>
{
  reference processed_states : Set<MT::State>
  reference repository : EMFRepository

    attribute combinations : Hashtable<MT::State, Set<MT::State>>

    operation determinize(input : MT::FSM, output : MT::FSM, output_state : MT::State) is do

      if not processed_states.contains(output_state) then
        processed_states.add(output_state)
        // For each letter of the alphabet
        input.alphabet.each { nextl |
          // There exists a state x of q' (where q' is a P(Q))
          // and state-y from Q so that : x --l-->  y belongs to input.transitionSet
          var newq : MT::State init MT::State.new
        combinations.put( newq,
            input.ownedTransition.select { e | e.input.equals(nextl) }.
            select { a |
              output_state.name == a.source.name
              or
              combinations.getValue(output_state).detect { i |
                i.name==a.source.name } != void }.collect { b |
                  b.target }.asSet)


          newq.name := join(combinations.getValue(newq).collect{ a | a.name })

          // Add the state to the output automaton if we found one
          if (newq.name.size > 0) then
          // Add the new state
          if (output.ownedState.detect { e | newq.name == e.name } == void) then
            output.ownedState.add(newq)
          else
            newq := output.ownedState.detect { e | e.name == newq.name }
          end
          // Add the new transition
          var newt : MT::Transition
          newt.source := output_state
          newt.target := newq
          newt.input := nextl
          output.ownedTransition.add(newt)
          self.determinize(input, output, newq)
          end
      } // End of Loop
      end
    end

    operation join( str_seq : Collection<String>) : String is do
      result := ""
      from var it : Iterator<String> init str_seq.iterator
      until it.isOff
      loop
        result.append(it.next)
      end
    end

}
```

Listing A.2 – State Machine Determinisation

A.3 Input and Output

Typed persistence of models to and from XMI files is a significant motivator for model types. The provision of a general facility for loading resources from file to produce typed models is planned as a future extension for Kermeta. In the interim, loading and saving models must be performed with respect to the specific model type. In some cases, such as those of basic state machines and mandatory-start state machines, a degree of code sharing is possible.

Figure A.3 shows a class that, once parameterized, will turn an untyped resource, into a typed model. The loading of the textual form into an untyped *Resource* object is handled by Kermeta's existing I/O features, which are in turn built on (and thus closely resemble) those of EMF.

```
class GenericLoader<MT : basic_fsm_type> {
  operation resourceToModel(resource : Resource) : MT is
  do
        var newModel : MT init MT.new
        resource.instances.select {o | MT::FSM.isInstance(o)
           }.each { f |
          var fsm : MT::FSM
          fsm ?= f
          newModel.add(fsm)
          fsm.ownedState.each{s | newModel.add(s) }
          fsm.ownedTransition.each{t | newModel.add(t) }
        }
        resource.instances.select{o | MT::State.isInstance(o)
           }.each { s |
          var state : MT::State
          state ?= s
          newModel.add(state)
        }
        resource.instances.select{o | MT::Transition.
           isInstance(o) }.each { t |
          var transition : MT::Transition
          transition ?= t
          newModel.add(transition)
        }
    result := newModel
  end
}
```

Listing A.3 – State Machine Loader

Glossaire

$<\#$ Represents the matching relation.

Beta Object-oriented programming language defined in the 70s and 80s as a successor to the Simula languages. Was the first language to feature virtual types.

EMF Eclipse Modeling Framework.

EMOF Essential MOF. Part of MOF 2.0.

Expression Problem Symptomatic problem in programming and software engineering whereby a system is extended simultaneously by adding new operators and new terms to a language implementation.

Family Polymorphism Idea of polymorphism based on groups of interrelated types that evolve as a group rather than individually, proposed by Ernst and first demonstrated in gbeta.

HUTN Human-Usable Textual Notation.

JMI Java Metadata Interface.

Kermeta Imperative metamodelling / modelling / model-oriented programming language.

LOOM Object-oriented programming language based on matching rather than subtyping. Includes the notion of type groups.

Matching The matching relation, as used by Cardelli and Bruce, is a weaker form of type conformance than subtyping, in that it does not enjoy subsumption, but is useful in cases such as parametric polymorphism, as a more intuitive relationship than F-Bounds.

MDA Model-Driven Architecture. Trademark of OMG.

MDD Model-Driven Development.

MDE Model-Driven Engineering.

MDSD Model-Driven Software Engineering.

Metamodel Structure for defining the concepts and relationships of a modelling language using a meta-metalanguage, e.g. MOF.

Model A graph of objects and links used to represent some or all of a system. Defined in terms of a metamodel.

Model type The type of a model, represented by the set of classes of which objects in the model are instances.

MOF Meta-Object Facility. A meta-metalanguage for the definition of modelling languages. Standardised by the OMG.

MTL Model Transformation Language. Imperative language intended for model transformation as part of UMLaut framework. A precursor to Kermeta.

Multiplicity In MOF, a multiplicity consists of lower and upper bounds for an element's cardinality, as well as flags for orderedness and uniqueness.

OMG Object Management Group.

Operation In MOF, an operation is a named function with a return type and named parameters, each of which are characterised by multiplicities.

Property MOF mechanism for data abstraction on objects. Also used in related pairs for modelling associations in EMOF.

Quokka A small scrub-wallaby found on Rottnest Island, Western Australia.

QVT Query / View / Transformation. OMG specification for defining model transformations.

Recursion See recursion.

Tefkat Declarative model transformation language and engine. The language was formerly known as XMorph and the engine formerly known as Tarzan.

Virtual type A type defined within or relative to another type. First featured in Beta.

XMI XML-Based Model Interchange.

Bibliographie

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects.* Springer, 1996.

[AK03] Colin Atkinson and Thomas Kühne. Model-driven development : A metamodeling foundation. *IEEE Software*, 20(5) :36–41, 2003.

[AP04] Marcus Alanen and Ivan Porres. The coral modelling framework. In Kai Koskimies, Ludwik Kuzniarz, Johan Lilius, and Ivan Porres, editors, *Proceedings of the 2nd Nordic Workshop on the Unified Modeling Language NWUML'2004*, number 35 in General Publications. Turku Centre for Computer Science, Jul 2004.

[Aug99] Lennart Augustsson. Cayenne - a language with dependent types. In *Advanced Functional Programming*, volume 1608 of *LNCS*, pages 240–267. Springer, 1999.

[BBDV03] Jean Bézivin, Erwan Breton, Grégoire Dupé, and Patrick Valduriez. The ATL Transformation-based Model Management Framework. Technical Report TR03-08, University of Nantes, September 2003.

[BBF06] Nelly Bencomo, Gordon Blair, and Robert France. Summary of the Workshop Models@run.time at MoDELS 2006. *Satellite Events at the MoDELS 2006 Conference*, pages 226–230, October 2006. To appear.

[BdGdLJ02] Frédéric Besson, Thomas de Grenier de Latour, and Thomas Jensen. Secure calling contexts for stack inspection. *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 76–87, 2002.

[BFJ$^+$03] Jean Bézivin, Nicolas Farcet, Jean-Marc Jézéquel, Benoît Langlois, and Damien Pollet. Reflective Model Driven Engineering. *Proceedings of the 6th International Conference on the Unified Modeling Language, Modeling Languages and Applications (UML 2003)*, 2863 :175–189, 2003.

[BG97] Nigel Baker and Jean-Marie Le Goff. Meta Object Facilities and their Role in Distributed Information Management Systems. In

Proceedings of the EPS ICALEPCS97 conference, Beijing, China, November 1997.

[BGMT88] Gerard Boudier, Ferdinando Gallo, Regis Minot, and Ian Thomas. An overview of PCTE and PCTE+. In *SDE 3 : Proceedings of the 3rd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 248–257, New York, NY, USA, 1988. ACM Press.

[BGS04] Xavier Blanc, Marie-Pierre Gervais, and Prawee Sriplakich. Model bus : Towards the interoperability of modelling tools. In *European Workshop on Model Driven Architecture : Foundations and Applications (MDAFA'2004)*, volume 3599 of *LNCS*, pages 17–32. Springer, 2004.

[BH02] Luciano Baresi and Reiko Heckel. Tutorial introduction to graph transformation : A software engineering perspective. In *Proc. 1st International Conference on Graph Transformation, ICGT'02*, volume 2505 of *LNCS*, pages 402–429. Springer, 2002.

[BJV04] 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*, Vancouver, Canada, 2004.

[Boo94] Grady Booch. *Object-oriented analysis and design with applications (2nd ed.).* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[Bru02] Kim B. Bruce. *Foundations of Object-Oriented Languages : Types and Semantics.* MIT Press, Cambridge, MA, 2002.

[Bru03] Kim B. Bruce. Some challenging typing issues in object-oriented languages. *Electronic Notes in Theoretical Computer Science*, 82(7), 2003.

[BSM$^+$03] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework.* Eclipse Series. Addison-Wesley Professional, 2003.

[BSvGF03] Kim B. Bruce, Angela Schuett, Robert van Gent, and Adrian Fiech. Polytoil : A type-safe polymorphic object-oriented language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(2) :225–290, 2003.

[BV99] Kim B. Bruce and Joseph Vanderwaart. Semantics-driven language design : Statically type-safe virtual types in object-oriented languages. *Electronic Notes in Theoretical Computer Science*, 20, 1999.

[BW05] Gavin M. Bierman and Alisdair Wren. First-class relationships in an object-oriented language. In *19th European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *LNCS*, pages 262–286. Springer, 2005.

[Cas97] Giuseppe Castagna. *Object-Oriented Programming : A Unified Foundation*. Progress in Theoretical Computer Science. Birkhäuser, Boston, 1997.

[CDI$^+$97] Stephen Crawley, Scott Davis, Jadwiga Indulska, Simon McBride, and Kerry Raymond. Meta-meta is better-better ! In *Proceedings, Working Conference on Distributed Applications and Information Systems (DAIS 1997)*. Chapman & Hall, 1997.

[CDNW07] Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. Tribe : A simple virtual class calculus. In *Sixth International Conference on Aspect-Oriented Software Development (AOSD 2007)*. ACM Press, March 2007.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming : Methods, Tools and Applications*. Addison-Wesley, 2000.

[CGLO06] Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A core calculus for scala type checking. In *Proceedings, 31st International Symposium on Mathematical Foundations of Computer Science (MFCS 2006)*, volume 4162 of *LNCS*, pages 1–23. Springer, 2006.

[CH06] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3) :621–646, 2006.

[CHC90] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proceedings, 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 125–135. ACM Press, 1990.

[CMR96] Andrea Corradini, Ugo Montanari, and Francesca Rossi. Graph Processes. *Fundamenta Informaticae*, 26(3/4) :241–265, 1996.

[CMT06] Jesús Sánchez Cuadrado, Jesús García Molina, and Marcos Menárguez Tortosa. RubyTL : A Practical, Extensible Transformation Language. In *Proceedings, 2nd European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2006)*, volume 4066 of *LNCS*, pages 158–172. Springer, 2006.

[Coi87] Pierre Cointe. Metaclasses are first class : The ObjVlisp Model. In *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 156–162. ACM Press, 1987.

[DG06] Stéphane Ducasse and Tudor Gîrba. Using smalltalk as a reflective executable meta-language. In *Proceedings, 9th International Conference on Model-Driven Engineering Languages and Systems (MoDELS 2006)*, volume 4199 of *LNCS*, pages 604–618. Springer, 2006.

[DGL$^+$03] Keith Duddy, Anna Gerber, Michael J. Lawley, Kerry Raymond, and Jim Steel. Model transformation : A declarative, reusable patterns approach. In *Proceedings, 7th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2003*, pages 174–195. IEEE Computer Society, 2003.

[DGL$^+$04] 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.

[EHC05] Gregor Engels, Reiko Heckel, and Alexey Cherchago. Flexible interconnection of graph transformation modules. In *Formal Methods in Software and Systems Modeling*, volume 3393 of *LNCS*, pages 38–63. Springer, 2005.

[EJT04] Jonathan Edwards, Daniel Jackson, and Emina Torlak. 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*, pages 189–199. ACM Press, 2004.

[EOC06] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006)*, pages 270–282. ACM, 2006.

[Ern99] Erik Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[Ern01] Erik Ernst. Family polymorphism. In *ECOOP '01 : Proceedings of the 15th European Conference on Object-Oriented Programming*, volume 2072 of *LNCS*, pages 303–326. Springer, 2001.

[FEBF06] Jean-Marie Favre, Jacky Estublier, and Mireille Blay-Fornarino, editors. *L'ingénierie dirigée par les modèles : au-delà du MDA*. Lavoisier, 2006.

[Fle06] Franck Fleurey. *Langage et méthode pour une ingénierie des modèles fiable.* PhD thesis, Université de Rennes 1, October 2006.

[FSB04] Franck Fleurey, Jim Steel, and Benoit Baudry. MDE and validation : Testing model transformations. In *Proc. of the SIVOES-Modeva workshop, SIVOES (Specification Implementat ion and Validation Of Embedded Systems)-MoDeVa (Model Design and Validation)*, Rennes, November 2004.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns : elements of reusable object-oriented software.* Addison-Wesley Longman, 1995.

[GLR$^+$02] 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.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80 : the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1983.

[HEET99] Reiko Heckel, Gregor Engels, Hartmut Ehrig, and Gabriele Taentzer. Classification and comparison of module concepts for graph transformation systems. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Gregor Rozenberg, editors, *Handbook of graph grammars and computing by graph transformation : vol. 2 : applications, languages, and tools*, pages 669–689. World Scientific Publishing, Singapore, 1999.

[HLR06] David Hearnden, Michael Lawley, and Kerry Raymond. Incremental model transformation for the evolution of model-driven systems. In *Proceedings, 9th International Conference on Model-Driven Engineering Languages and Systems (MoDELS 2006)*, volume 4199 of *LNCS*, pages 321–335. Springer, 2006.

[HRS02] David Hearnden, Kerry Raymond, and Jim Steel. Anti-yacc : Mof-to-text. In *Proceedings, 6th International Enterprise Distributed Object Computing Conference (EDOC 2002)*, pages 200–211. IEEE Computer Society, 2002.

[Int99] International Organization for Standardization. *ISO/IEC 14750 :1999 : Information technology — Open Distributed Processing — Interface Definition Language.* International Organization for Standardization, Geneva, Switzerland, 1999.

[Int05] International Organization for Standardization. *ISO/IEC 19502 :2005 — Information Technology — Meta Object Facility (MOF).* International Organization for Standardization, Geneva, Switzerland, 2005.

[IP02] Atsushi Igarashi and Benjamin C. Pierce. Foundations for virtual types. *Information and Computation*, 175(1) :34–49, 2002.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java : A Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3) :396–450, 2001.

[ISV05] Atsushi Igarashi, Chieri Saito, and Mirko Viroli. Lightweight family polymorphism. In *Proceedings, The Third Asian Symposium on Programming Languages and Systems (APLAS)*, volume 3780 of *LNCS*, pages 161–177. Springer, 2005.

[Jac91] Ivar Jacobson. *Object-oriented software engineering.* ACM Press, 1991.

[JDAO04] Paul Jolly, Sophia Drossopoulou, Christopher Anderson, and Klaus Ostermann. Simple dependent types : Concord. In *ECOOP Workshop on Formal Techniques for Java Programs (FTfJP 2004)*, Oslo, Norway, June 2004.

[KÖ5] Alexander Königs. Model transformation by graph transformation : A comparative study. In *MoDELS 2005 Workshop on Model Transformation in Practice (MTiP)*, Montego Bay, Jamaica, October 2005.

[Kle06] Jacques Klein. *Aspects Comportementaux et Tissage.* PhD thesis, Université de Rennes 1, 2006.

[KMMPN87] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. The beta programming language. In *Research Directions in Object-Oriented Programming*, pages 7–48. MIT Press, 1987.

[KP07] Andrew Kennedy and Benjamin C. Pierce. On decidability of nominal subtyping with variance. In *2007 International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD'07)*, Nice, France, January 2007. To appear.

[KRB91] Gregor Kiczales, Jim Des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[Kru03] P. Kruchten. *The Rational Unified Process : An Introduction.* Addison-Wesley Professional, 2003.

[Küh06] Thomas Kühne. Matters of (meta-) modeling. *Journal of Software and Systems Modeling (SoSyM)*, 5(4) :395–401, December 2006.

[LHBC05] Roberto E. Lopez-Herrejon, Don Batory, and William Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP 2005)*, volume 3586 of *LNCS*, pages 169–194. Springer, 2005.

[LP91] Wilf LaLonde and John Pugh. Subclassing $\neq$ subtyping $\neq$ is-a. *Journal of Object-Oriented Programming*, 3(5) :57–62, January 1991.

[LS05] Michael Lawley and Jim Steel. Practical declarative model transformation with tefkat. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *LNCS*, pages 139–150. Springer, 2005.

[Mey92] Bertrand Meyer. *Eiffel : the language.* Prentice-Hall, 1992.

[MFF^{+}06] Pierre-Alain Muller, Franck Fleurey, Frédéric Fondement, Michel Hassenforder, Rémi Schneckenburger, Sébastien Gérard, and Jean-Marc Jézéquel. Model-driven analysis and synthesis of concrete syntax. In *Proceedings, 9th International Conference on Model-Driven Engineering Languages and Systems (MoDELS 2006)*, volume 4199 of *LNCS*, pages 98–110. Springer, 2006.

[MFJ05] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In *Proceedings, 8th International Conference on Model-Driven Engineering Languages and Systems (MoDELS 2005)*, volume 3713 of *LNCS*, pages 264–278. Springer, 2005.

[MFV^{+}05] Pierre-Alain Muller, Franck Fleurey, Didier Vojtisek, Zoë Drey, Damien Pollet, Frédéric Fondement, Philippe Studer, and Jean-Marc Jézéquel. On executable meta-languages applied to model transformations. In *MoDELS 2005 Workshop on Model Transformation in Practice (MTiP)*, Montego Bay, Jamaica, October 2005.

[MM04] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1) :69–111, 2004.

[MMP89] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes : A powerful mechanism in object-oriented programming. In *Proceedings, Conference on Object-Oriented Programming : Systems, Languages, and Applications (OOPSLA 1989)*, volume 24 of *SIGPLAN Notices*, pages 397–406. ACM, 1989.

[Mor10] Thomas Hunt Morgan. Sex limited inheritance in drosophila. *Science*, 32 :120–122, 1910.

[Mos04] Peter D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60 :195–228, 2004.

[NBKS06] Clémentine Nebut, Benoît Baudry, Souha Kamoun, and Waqas Ahmed Saeed. Multi-language support for model-driven requirement analysis and test generation. In *ECMDA workshop on Integration of Model Driven Development and Model Driven Testing.*, Bilbao, Spain, July 2006.

[Obj02] Object Management Group (OMG). Request for Proposal : MOF 2.0 Query/Views/Transformations RFP. OMG Document number ad/2002-04-10, April 2002.

[Obj03] Object Management Group (OMG). MDA Guide Version 1.0.1. OMG Document no. omg/2003-06-01, June 2003.

[Obj04a] Object Management Group (OMG). Enterprise Collaboration Architecture (ECA). OMG Document no. formal/2004-02-01, 2004.

[Obj04b] Object Management Group (OMG). Human usable textual notation (HUTN) specification, version 1.0. OMG Document no. formal/04-08-01, August 2004.

[Obj05a] Object Management Group (OMG). MOF 2.0/XMI Mapping Specification, v2.1. OMG Document number formal/05-09-01, September 2005.

[Obj05b] Object Management Group (OMG). MOF QVT Final Adopted Specification. OMG Document number ptc/2005-11-01, November 2005.

[Obj05c] Object Management Group (OMG). Unified Modeling Language (UML) : Infrastructure, version 2.0. OMG Document formal/05-07-05, July 2005.

[Obj06a] Object Management Group (OMG). Meta-Object Facility (MOF) Core Specification, Version 2.0. OMG Document no. formal/2006-01-01, January 2006.

[Obj06b] Object Management Group (OMG). MOF Models to Text Transformation Language Final Adopted Specification. OMG Document no. ptc/2006-11-01, November 2006.

[Obj06c] Object Management Group (OMG). Object Constraint Language (OCL) Specification, Version 2.0. OMG Document no. formal/2006-05-01, January 2006.

[OCRZ03] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proceedings, 17th European Conference on Object-Oriented Program-*

ming (ECOOP 2003), volume 2743 of *LNCS*, pages 201–224. Springer, 2003.

[OFMP$^+$95] Anders Olsen, Ove Færgemand, Birger Møller-Pedersen, Rick Reed, and J.R.W. Smith. *Systems Engineering with SDL-92.* North Holland, 1995.

[OZ05] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005)*, pages 41–57. ACM, 2005.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[Poe04] Iman Poernomo. A type theoretic framework for formal metamodelling. In *Architecting Systems with Trustworthy Components*, volume 3938 of *LNCS*, pages 262–298. Springer, 2004.

[Poe06] Iman Poernomo. The meta-object facility typed. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC)*, pages 1845–1849. ACM, 2006.

[Pol05] Damien Pollet. *Une architecture pour les transformations de modèles et la restructuration de modèles UML.* PhD thesis, Université de Rennes 1, June 2005.

[QVT05] QVT-Merge Group. Revised submission for MOF 2.0 Query/-Views/Transformations RFP. OMG document number ad/2005-03-02, March 2005.

[RBP$^+$91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-oriented modeling and design.* Prentice-Hall, 1991.

[RD99] Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *Proceedings, International Conference on Software Maintenance (ICSM'99)*, pages 13–22. IEEE Computer Society, 1999.

[Rum03] Bernhard Rumpe. Model-based testing of object-oriented systems. In *Proceedings, Formal Methods for Components and Objects (FMCO'02)*, volume 2582 of *LNCS*. Springer, 2003.

[Rum06] Bernhard Rumpe. Agile test-based modeling. In *Proceedings, International Conference on Software Engineering Research and Practice (SERP 2006)*, pages 10–15. CSREA Press, 2006.

[Sen03] 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, Anaheim, USA, 2003.

[SL04] Jim Steel and Michael Lawley. Model-based test driven development of the tefkat model-transformation engine. In *Proceedings, 15th International Symposium on Software Reliability Engineering (ISSRE 2004)*, pages 151–160. IEEE Computer Society, 2004.

[SR01] Jim Steel and Kerry Raymond. Generating human-usable textual notations for information models. In *Proceedings of the 5th International Conference on Enterprise Distributed Object Computing (EDOC 2001)*, pages 250–261. IEEE Computer Society, 2001.

[Ste04] Jim Steel. Types in MDA. In *Second European Workshop on Model-Driven Architecture (EW-MDA2)*, Canterbury, UK, September 2004.

[Sun02] Sun Microsystems. The Java Metadata Interface (JMI). Java Community Process Specification JSR-000040, SUN Microsystems, June 2002.

[TEG+05] Gabriele Taentzer, Karsten Ehrig, Esther Guerra, Juan de Lara, Laszlo Lengyel, Tihamer Levendovszky, Ulrike Prange, Daniel Varro, and Szilvia Varro-Gyapay. Model transformation by graph transformation : A comparative study. In *MoDELS 2005 Workshop on Model Transformation in Practice (MTiP)*, Montego Bay, Jamaica, October 2005.

[THA05] Sam Tobin-Hochstadt and Eric Allen. A core calculus of metaclasses. In *Fundamentals of Object-Oriented Languages (FOOL) 2005*, Long Beach, USA, January 2005.

[Tor04] Mads Torgersen. The expression problem revisited : Four new solutions using generics. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 123–143. Springer-Verlag, July 2004.

[TR03] Juha-Pekka Tolvanen and Matti Rossi. MetaEdit+ : defining and using domain-specific modeling languages and code generators. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 92–93. ACM Press, 2003.

[Tra05] Laurence Tratt. Model transformations and tool integration. *Software and Systems Modeling (SoSyM)*, 4(2) :112–122, 2005.

[VVR03] Antonio Vallecillo, Vasco T. Vasconcelos, and António Ravara. Typing the Behavior of Objects and Components using Session Types. *Fundamenta Informaticae*, 20 :1–18, 2003.

[Wad98] Philip Wadler. The expression problem. Email to the Java Genericity mailing list, December 1998.

[Xi98] Hongwei Xi. *Dependent Types in Practical Programming.* PhD thesis, Carnegie-Mellon University, September 1998.

[ZO05] Matthias Zenger and Martin Odersky. Independently extensible solutions to the expression problem. In *Proceedings of the 12th International Workshop on Foundations of Object-Oriented Languages (FOOL 12).* ACM, January 2005.

Table des figures

Listings

Publications

International Journals

1. Jim Steel and Jean-Marc Jézéquel. **On Model Typing**. *Journal of Software and Systems Modeling (SoSyM)*. Springer. To appear.

Book Chapters

1. Keith Duddy, Anna Gerber, Michael J. Lawley, Kerry Raymond and Jim Steel. **Declarative Transformation of Object-Oriented Models**. In P. van Bommel (ed), *Transformation of Knowledge, Information and Data : Theory and Applications*. IDEA Group Publishing, 2004.

International Conferences

1. Jim Steel and Kerry Raymond. **Generating Human-Usable Textual Notations for Information Models**. In *Proceedings, 5th International Conference on Enterprise Distributed Object Computing (EDOC 2001)*. Seattle, USA, 2001.
2. Anna Gerber, Michael J. Lawley, Kerry Raymond, Jim Steel and Andrew Wood. **Model Transformation : The Missing Link of MDA**. In *Proceedings, 1st International Conference on Graph Transformation (ICGT 2002)*. Barcelona, Spain, 2002.
3. David Hearnden, Kerry Raymond and Jim Steel. **Anti-Yacc : MOF-to-Text**. In *Proceedings, 6th International Conference on Enterprise Distributed Object Computing (EDOC 2002)*. Lausanne, Switzerland, 2003.
4. David Hearnden, Kerry Raymond and Jim Steel. **MQL : A Powerful Extension to OCL for MOF Queries**. In *Proceedings, 7th International Conference on Enterprise Distributed Object Computing (EDOC 2003)*. Brisbane, Australia, 2003.
5. Keith Duddy, Anna Gerber, Michael J. Lawley, Kerry Raymond and Jim Steel. **Model Transformation : A declarative, reusable patterns ap-**

proach In *Proceedings, 7th International Conference on Enterprise Distributed Object Computing (EDOC 2003)*. Brisbane, Australia, 2003.

6. Jim Steel and Michael Lawley. **Model-Based Test Driven Development of the Tefkat Model-Transformation Engine**. In *Proceedings, 15th International Symposium on Software Reliability Engineering (ISSRE 2004)*. St. Malo, France, 2004.
7. Jim Steel and Jean-Marc Jézéquel. **Model Typing for Improving Reuse in Model-Driven Engineering**. Montego Bay, Jamaica, 2005.
8. Erwan Brottier, Franck Fleurey, Jim Steel, Benoît Baudry and Yves Le Traon. **Metamodel-based Test Generation for Model Transformations : an Algorithm and a Tool**. In *Proceedings, 17th International Symposium on Software Reliability Engineering (ISSRE 2006)*. Raleigh, USA, 2006.

International Workshops

1. Jim Steel. **A Textual Notation Generator for MOF Models**. **UML in the .com Enterprise Workshop**. Palm Springs, USA, 2000.
2. Keith Duddy, Anna Gerber, Michael J. Lawley, Kerry Raymond and Jim Steel. **Modelware for Middleware**. In *Proceedings, Middleware Workshop on Model-driven Approaches to Middleware Applications Development (MAMAD 2003)*. Rio de Janeiro, Brazil, 2003.
3. Jim Steel and Jean-Marc Jézéquel. **Typing Relatinoships in MDA**. *2nd European Workshop on Model-Driven Architecture (EWMDA-2)*. Canterbury, UK, 2004.
4. Jim Steel. **Roundtable Report : Types in MDA**. *2nd European Workshop on Model-Driven Architecture (EWMDA-2)*. Canterbury, UK, 2004.
5. Franck Fleurey, Jim Steel and Benoît Baudry. **Validation in Model-Driven Engineering : Testing Model Transformations**. In *Proceedings, 1st International Workshop on Model Design and Validation, at ISSRE 2004*. Rennes, France, 2004.
6. Michael Lawley and Jim Steel. **Practical Declarative Model Transformation with Tefkat**. *MoDELS 2005 Workshop on Model Transformation in Practice (MTiP'05)*. Montego Bay, Jamaica, 2005.

Other Publications

1. Object Management Group (OMG). **Human-Usable Textual Notation (HUTN) Specification, Version 1.0**. *OMG Document number formal/2004-08-01*, 2004. (as primary author, editor, and FTF chair)