Using Macromodels to Manage Collections of Related Models

Rick Salay, John Mylopoulos, and Steve Easterbrook

Department of Computer Science, University of Toronto
Toronto, ON M5S 3G4, Canada
{rsalay,jm,sme}@cs.toronto.edu

Abstract. The creation and manipulation of multiple related models is common in software development, however there are few tools that help to manage such collections of models. We propose a framework in which different types of model relationships -- such as submodelOf and refinementOf -- can be formally defined and used with a new type of model, called a *macromodel*, to express the required relationships between models at a high-level of abstraction. Macromodels can be used to support the development, comprehension, consistency management and evolution of sets of related models. We illustrate the framework with a detailed example from the telecommunications industry and describe a prototype implementation.

Keywords: Modeling, Metamodeling, Macromodeling, Relationships, Mappings.

1 Motivation

In Software Engineering, it is common to model systems using multiple interrelated models of different types. A typical modeling paradigm provides a collection of domain specific modeling languages, and/or multi-view languages such as UML. The modeling languages are typically defined through a metamodel, however only very limited support is typically provided for expressing the relationships between the models.

Existing approaches to supporting model relationships tend to focus on how the contents of specific model instances are related (e.g. [7, 10]), rather than on the models themselves. Those approaches that do provide abstractions for expressing relationship between models typically concentrate on a limited set of relationship types required to support model transformation (e.g. [2, 4]) and/or traceability (e.g. [1]). In contrast, we argue that a rich, extensible set of relationship types for relating models is needed, to capture the intended meaning of the models and the various ways that models in a collection can constrain one another.

UML suffers from a similar limitation. The UML metamodel defines a number of diagram types as projections of a single UML model. However, developers rarely manipulate the underlying UML model directly - they work with the diagrams, and the intended purpose of each new diagram is only partially captured in the UML metamodel. For example, a developer might create an object diagram showing

P. van Eck, J. Gordijn, and R. Wieringa (Eds.): CAiSE 2009, LNCS 5565, pp. 141–155, 2009.

instantiations of a subset of the classes, chosen to be relevant to a particular use case scenario. The relationships of these objects to the classes in the model is captured in the metamodel, but the relationship between this particular object diagram and the set of behaviour models (e.g. sequence diagrams) that capture the scenario is left implicit.

The key point is that each model in a collection is created for a specific purpose, for example to capture important concepts, or to elaborate and constrain other models. Each model therefore has an *intended purpose* and a set of *intended relationships* with other models. Such relationships might include submodels, refinements, aspects, refactorings, transformations, instantiations, and so on. A precise definition of these intended relationships is needed to fully capture the intended meanings of the models.

We propose a framework for systematically extending a modeling paradigm to formally define model relationship types between model types and we introduce a new type of model, called a *macromodel*, for managing collections of models. A macromodel consists of elements denoting models and links denoting intended relationships between these models with their internal details abstracted away. The framework provides a number of benefits:

1. Understandability. When large models are decomposed into smaller ones, the representation of relationships is essential for understandability [9], thus, making the underlying relational structure of a set of models explicit helps comprehension.
2. Specifying constraints on models, even for models yet to be created. When constituent models change, they are still expected to maintain the relationships expressed in the macromodel, because the macromodel captures the intentions of the modelers on how the set of models should be structured.
3. Consistency checking. The macromodel can be used to assess an evolving set of models for conformance with modeler intentions even when mappings between models are incomplete.
4. Model evolution. A change to the macromodel can be taken as a specification for change in the collection of models it represents – either involving the removal of some models, changes in model relationships, or additions of new models. Thus, a macromodel can be used to guide model evolution.

In a short paper [14], we introduce the basic concepts of the framework. In this paper, we elaborate the details, provide a formal semantics for macromodels and describe a new implementation that integrates the framework with the model finder Kodkod [16] to support automated model management activities such as consistency checking and model synthesis.

The structure of this paper is as follows. Section 2 introduces the framework informally and then Section 3 provides a formal treatment. In Section 4, we describe a prototype tool implementation and Section 5 describes the application of the framework to a detailed example from the telecommunications domain. In Section 6 we discuss related work and finally in Section 7 we state some conclusions and discuss future work.

2 Framework Description

In the framework we assume that at the detailed level, the relationship between two (or more) models is expressed as a special kind of model that contains the mapping

relating the model elements. Furthermore, these relationships can be classified into types and that they can be formalized using metamodels. For example, Figure 1 shows an instance of the *objectsOf* relationship type that can hold between a sequence diagram and object diagram in UML. Each object symbol in the sequence diagram is mapped to an object symbol in the object diagram that represents the same object (via the identity relation *id*) and each message in the sequence diagram is mapped to the link in the object diagram over which the message is sent (via the relation *sentOver*). In addition, the mapping is constrained so that both *id* and *sentOver* are total functions and the mapping must be consistent in the sense that the endpoint objects of a message should be the same as the endpoint objects of the link to which it is mapped.

The benefit of defining different relationship types such as *objectsOf* is that their instances express the relationships between models at two levels of abstraction. At the detail level, it shows how the elements of the models are related. At the aggregate, or *macro* level, it expresses a fact about how the models are related as a whole and conveys information about how a collection of models is structured. Thus, we can use these to express meaningful macromodels such as the one in Figure 2 that shows the relationships between some of the models and diagrams of a hypothetical library management system. Here we have added the relationship types *caseOf* that holds between a sequence diagram and its specializations, *actorsOf* that holds between a model and an organization chart and *diagramsOf* that holds between a collection of UML diagrams and the UML model they are diagrams of.

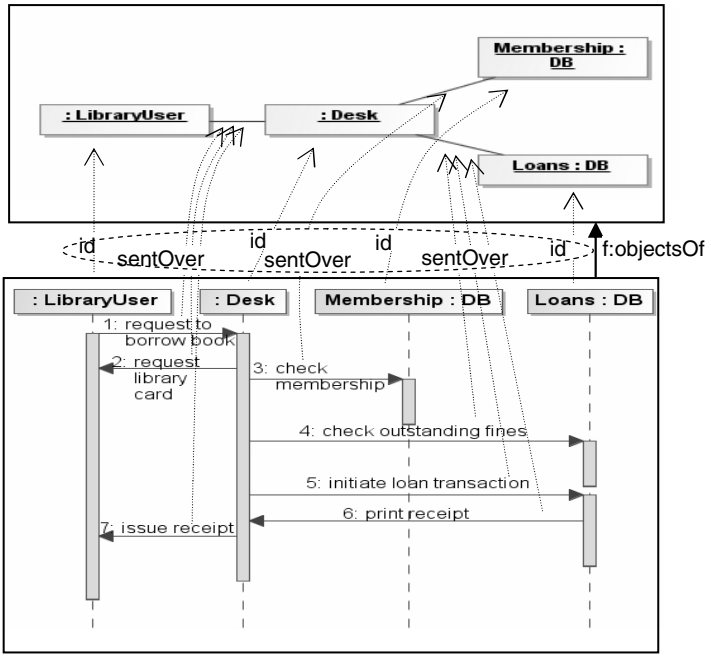

Fig. 1. A relationship between a sequence diagram and an object diagram

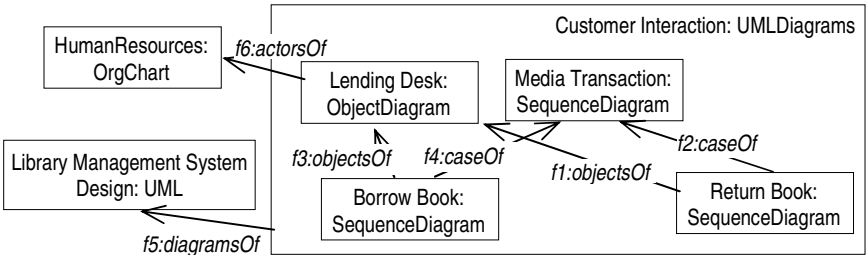

Fig. 2. Partial macromodel of a library system specification

A macromodel also has a metamodel and can contain well-formedness constraints on the valid configurations of models. For example, in Figure 2, a macromodel of type UMLDiagrams can only contain symbols denoting UML diagrams.

The symbols in a macromodel are realized by actual models and mappings. As the development of the library management system proceeds and as these artifacts evolve we expect the relationships expressed in the macromodel to be maintained. Thus, the macromodel provides a new level of specification in which the intentions of the modelers are expressed at the macroscopic level as models that must exist and relationships that must hold between models. Since these constraints are formalized using the metamodels of the relationship types involved, they can be leveraged by automated support for model management activities such as consistency checking and change propagation. We illustrate this in Section 5 with the prototype MCAST (Macromodel Creation and Solving Tool).

The application of the framework to a particular modeling paradigm involves the following steps:

1. The relationship types that are required for relating model types are defined using metamodels.
2. The metamodel for macromodels is defined in terms of the model and relationship types.
3. For a given development project within the paradigm, an initial macromodel is created expressing the required models and their relationships. As the project continues, both the macromodel and the constituent realizations of the models and relationships (i.e. mappings) continue to evolve. During the project, the macromodel is used as a way of supporting the comprehension of the collection of models in a project, specifying extensions to it and for supporting model management activities with tools such as MCAST.

We describe the formal basis for these steps below.

3 Formalization

The problem of how to express relationships between models has been studied in a number of different contexts including ontology integration [6], requirements engineering [10, 11] and model management [3]. Bernstein [3] defines the

relationship in terms of the semantics of the models: two models are related when the possible interpretations of one model constrain the possible interpretations of the other model. Thus, it is a binary relation over the sets of interpretations of the models.

At the syntactic level, this relationship can be expressed by embedding the models within a larger *relator* model that contains the mapping showing how the elements of the models are related. Figure 3 shows how the *objectsOf* relationship type between sequence and object diagrams shown in Figure 1 is defined using metamodels. Note that these are simplified versions of portions of the UML metamodel that correspond to the content of these diagrams. In order to formally relate these metamodels, we exploit the following similarity between metamodels and logical theories: a metamodel can be taken to consist of a pair $\langle \Sigma, \Phi \rangle$ where Σ defines the types of elements used in a model and is called its *signature* while Φ is a set of logical sentences over this vocabulary that define the well-formedness constraints for models. Thus, a metamodel is a logical theory and the set of finite models of this theory, in the model-theoretic sense, is also the set of models that the metamodel specifies. We designate this set $\mathrm{Mod}(\Sigma, \Phi)$.

Institution theory [5] provides a general way to relate logical theories by mapping the signatures of the theories in such a way that the sentences are preserved. We take a similar approach for metamodels and define the notion of a *metamodel morphism* between two metamodels as follows: a metamodel morphism $f{:}\langle \Sigma_A, \Phi_A \rangle \rightarrow \langle \Sigma_B, \Phi_B \rangle$ is a homomorphism of the signatures $f_\Sigma{:}\Sigma_A \rightarrow \Sigma_B$ such that $\Phi_B \models f(\Phi_A)$ - where we have abused the notation and have used f as a function that translates sentences over Σ_A to sentences over Σ_B according the mapping $f_\Sigma{:}\Sigma_A \rightarrow \Sigma_B$. Thus, by establishing a metamodel morphism from metamodel A to metamodel B we both map the element types and set up a proof obligation that ensures that any B-model contains an A-model that can be "projected" out of it.

In the example of Figure 3, the metamodel morphisms p_{OD} and p_{SD} map in the obvious way into the relator metamodel of *objectsOf* and the fact that they are metamodel morphisms ensures that every well-formed instance of *objectsOf* contains a well-formed instance of *ObjectDiagram* and of *SequenceDiagram* that it relates. In particular, an instance of *objectsOf* constrains the *SequenceDiagram* to have a subset of the objects of the *ObjectDiagram*, and it contains a functional association *sentOver*:*Message* $\rightarrow$ *Link* that satisfies the consistency constraint that when it maps a message to a link then the endpoint objects of the message must be endpoint objects of the link. Thus,

$$\Phi_{objectsOf} = p_{OD}(\Phi_{\mathrm{ObjectDiagram}}) \cup p_{SD}(\Phi_{\mathrm{SequenceDiagram}}) \cup$$
$$\{\forall m{:}\mathrm{Message}.$$
$$(linkStart(sentOver(m)) = messageStart(m) \wedge$$
$$linkEnd(sentOver(m)) = messageEnd(m)) \vee$$
$$(linkEnd(sentOver(m)) = messageStart(m) \wedge$$
$$linkStart(sentOver(m)) = messageEnd(m))\}$$

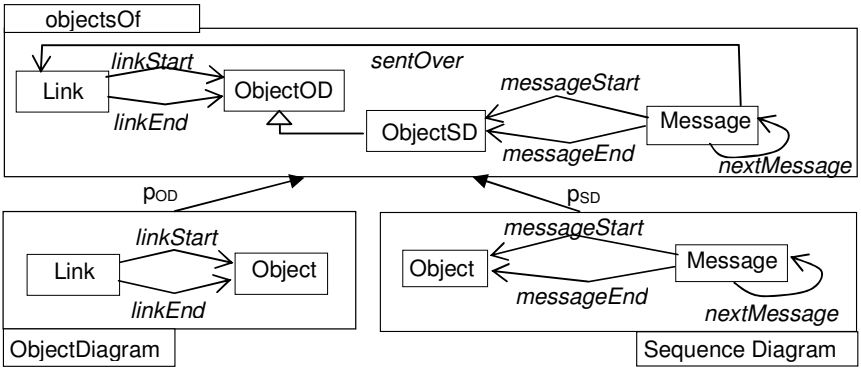

Fig. 3. Defining the *objectsOf* relationship type

The key benefit of using metamodel morphisms for relating metamodels is that the approach can be formulated in a way that is independent of the metamodeling language since each metamodeling language can define its own type of metamodel morphism. Furthermore, institution theory provides a formal means to relate different logics using *Institution morphisms* [5] and thus our approach extends similarly to multiple metamodeling formalisms; however, we do not pursue this direction further in this paper as it is secondary to our interests here.

In addition to "custom defined" relationship types such as *objectsOf* there are a variety of useful generic parameterized relationship types that can be automatically constructed from the metamodels for the associated model types. We discuss two of these briefly as they are used in Section 5.

Given any model type T, we can define the relationship type *eq*[T] where *eq*[T](M_1, M_2) holds iff there is an isomorphism between M_1 and M_2 and where we interpret corresponding elements as being semantically equal – i.e. they denote the same semantic entities. A second parameterized type is *merge*[T]. Given a collection of models K consisting of T-models, *merge*[T](K, M) holds iff M is the smallest T-model that contains all of the models in K as submodels. Note that a unique merge M may not always exist.

3.1 Macromodels

Now that we have an approach for defining relationship types we can characterize a *macromodel type* in terms of the model types and relationship types it contains. A macromodel is a hierarchical model whose elements are used to denote models that must exist and the relationships that must hold between them. Like model and relationship types, a macromodel type is defined using a metamodel. Figure 4 depicts a simple example of this. The upper part shows a macromodel metamodel *SimpleMulti* (left side) and the set of metamodels for the model and relationship types it can depict (right side). The axioms of the macromodel metamodel limit the possible well-formed collections of models and relationships that the macromodel can represent. The notation for macromodels expresses binary model relationship types as directed arrows between model types however they should be understood as

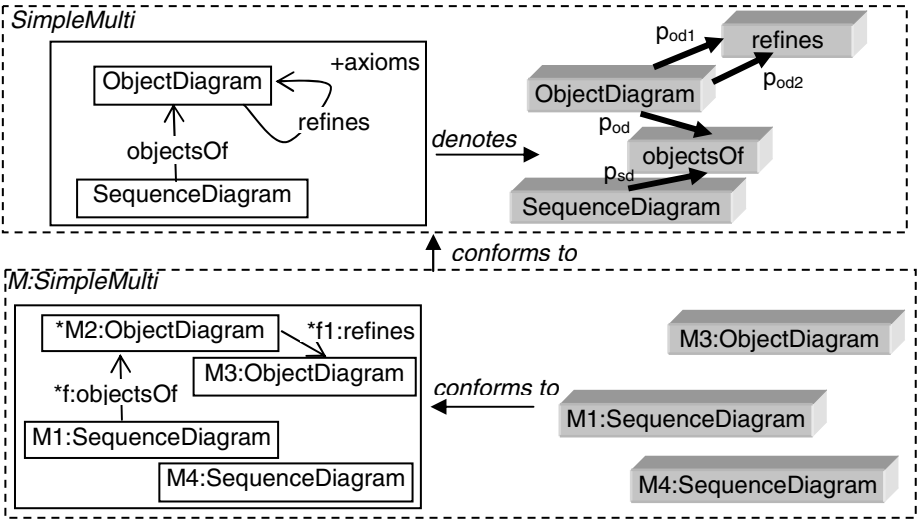

Fig. 4. A macromodel metamodel and an instance of it

consisting of a relator model and two metamodel morphisms. The simple illustration in Figure 4 does not depict hierarchy but Figure 9 shows a more complex one that does that we used in the example of Section 5.

The lower part of Figure 4 depicts a particular macromodel *M*:*SimpleMulti* and a collection of models that conform to it. The asterisk preceding *f*:*objectsOf*, *M2*:*ObjectDiagram* and *f1*:*refines* indicate that they are "unrealized" models and relationships. This implies that there is no corresponding instance for these in the collection of models specified by the macromodel and they are just placeholders used to express more complex constraints. The macromodel in Figure 4 expresses the fact that the collection should contain models M1, M3 and M4 and these must satisfy the constraint that M4 is a sequence diagram and the object diagram corresponding to sequence diagram M1 is a refinement of object diagram M3. Translated into first order logic we get the set of sentences shown in Figure 5.

sorts $S_{objectsOf}$, $S_{refines}$, $S_{ObjectDiagram}$, $S_{SequenceDiagram}$
pred objectsOf: $S_{objectsOf}$, refines: $S_{refines}$, ObjectDiagram: $S_{ObjectDiagram}$,
SequenceDiagram: $S_{SequenceDiagram}$
func p_{od}: $S_{objectsOf} \rightarrow S_{ObjectDiagram}$, p_{sd}: $S_{objectsOf} \rightarrow S_{SequenceDiagram}$,
p_{od1}: $S_{refines} \rightarrow S_{ObjectDiagram}$, p_{od2}: $S_{refines} \rightarrow S_{ObjectDiagram}$,
M1, M4: $S_{SequenceDiagram}$, M3: $S_{ObjectDiagram}$
axioms
SequenceDiagram(M4),
SequenceDiagram(M1),
ObjectDiagram(M3),
$\exists f$: $S_{objectsOf}$, $f1$: $S_{refines}$, $m2$: $S_{ObjectDiagram}$. objectsOf(f) $\wedge$ $p_{sd}(f)$ = M1 $\wedge$ $p_{od}(f)$ = $m2$
$\wedge$ refines($f1$) $\wedge$ $p_{od1}(f1)$ = $m2$ $\wedge$ $p_{od2}(f1)$ = M3 $\wedge$ ObjectDiagram($m2$)

Fig. 5. Example translation

Here we are using projection functions with the same name as their corresponding metamodel morphisms to associate relator models with the models they relate. Each connected set of unrealized elements in the macromodel is translated to an existential sentence with the unrealized models and relationships as existentially quantified variables. We will use this translation approach for defining the semantics of macromodels in general, below.

3.2 Macromodel Syntax and Semantics

We now define the syntax and semantics of macromodels formally. Figure 6 shows the abstract syntax and well-formedness rules of the macromodel language[1]. The notation of macromodels is summarized as follows:

- A *Model* element is represented as a box containing the model name and type separated by a colon. When the name is preceded with an asterisk then it is has the *realized* attribute set to **false**.
- A *Relation* element is represented with an arrow, if binary, or as a diamond with *n* legs, if *n*-ary. It is annotated with its name, type and with optional *Role* labels. When the name is preceded with an asterisk then it is has the *realized* attribute set to **false**.
- A sub-*Macromodel* element is represented as a box containing its name and type. It optionally can show the sub-macromodel as the contents of the box.
- A *Macrorelation* element is represented with an arrow, if binary, or as a diamond with *n* legs, if *n*-ary. It is annotated with its name, type and with optional *Macrorole* labels. It can optionally show its contents as a dashed oval linked to the main the arrow or diamond symbol.

Assume that we have a macromodel K which has metamodel T. To define the formal semantics of macromodels we proceed by first translating T to a first order signature Σ_T reflecting the different model and relationship types in it and then translating K to the theory $\langle \Sigma_T \cup \Sigma_K, \Phi_K \rangle$ where Σ_K consists of a set of constants

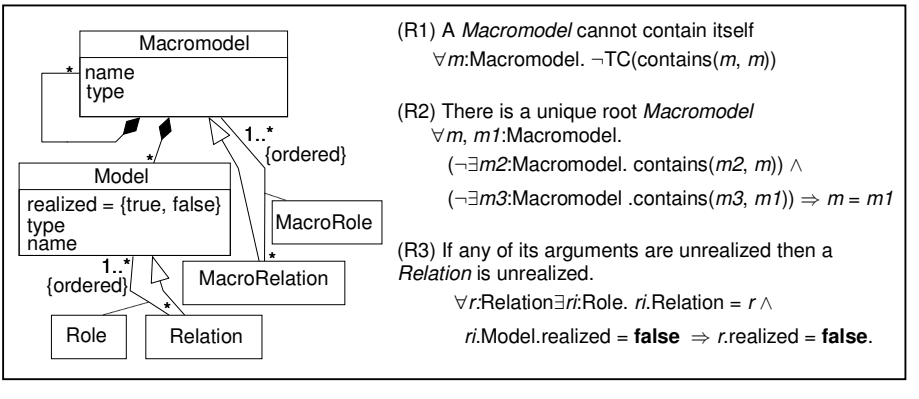

Fig. 6. Abstract syntax and well-formedness constraints of macromodels

[1] TC(*pred*(*x*, *y*)) denotes the transitive closure of *pred*(*x*, *y*).

The translation algorithm for T is as follows:
Σ_T is initially empty, then,
- For each metamodel $X = \langle \Sigma_X, \Phi_X \rangle$ denoted by a *Model* or *Relation* element, add a sort symbol S_X and a unary predicate symbol $X{:}S_X$ to Σ_T
- For each metamodel morphism $p{:}\langle \Sigma_X, \Phi_X \rangle \rightarrow \langle \Sigma_Y, \Phi_Y \rangle$ denoted by a *Role* element, add a function symbol $p{:}S_Y \rightarrow S_X$ to Σ_T

The interpretation J_T is constructed as follows:
- To each sort symbol S_X assign the set $\text{Mod}(\Sigma_X, \varnothing)$
- To each predicate symbol $X{:}S_X$ assign the unary relation defined by the set $\text{Mod}(\Sigma_X, \Phi_X)$
- To each function symbol $p{:}S_Y \rightarrow S_X$ assign the function $p{:}\ \text{Mod}(\Sigma_Y, \varnothing) \rightarrow \text{Mod}(\Sigma_X, \varnothing)$ induced by the signature morphism $p_\Sigma{:}\Sigma_X \rightarrow \Sigma_Y$

The translation algorithm for K is as follows:
Σ_K and Φ_K are initially empty, then,
- For each realized *Model* or *Relation* element M of type X add the constant $M{:}S_X$ to Σ_K and the axiom '$X(M)$' to Φ_K.
- For each *Role* element of type p from realized relation R to a realized model M, add the axiom '$p(R) = M$' to Φ_K.
- For each connected set $S=\{M_1, \ldots, M_n\}$ of unrealized *Model* and *Relation* elements, add the axiom '$\exists m_1, \ldots, m_n.\ \phi_S$' to Φ_K where ϕ_S is a conjuction constructed as follows:
  ϕ_S is initially empty, then,
  - For each element M_i of type X add the conjunct '$X(m_i)$' to ϕ_S.
  - For each *Role* element of type p from relation M_i to realized model M, add the conjunct '$p(m_i) = M$' to ϕ_S.
  - For each *Role* element of type p from relation M_i to model M_j, add the conjunct '$p(m_i) = m_j$' to ϕ_S.
  - For each *Role* element of type p from realized relation R to model M_i, add the conjunct '$p(R) = m_i$' to ϕ_S.

Fig. 7. Semantic interpretation algorithms

corresponding to the realized models and relationships in K and Φ_K is a set of axioms. Figure 5 is the result of performing the translation to the example in Figure 4. We then construct a "universal" interpretation J_T of Σ_T that consists of all possible models and relationships using these types. Any collection M of models and relationships conforms to K iff it is an assignment of the constants in Σ_K to elements of the appropriate types in J_T such that Φ_K are satisfied.

Figure 7 shows the algorithms involved. Note that in the translation algorithm for K, the connected sets of unrealized *Model* and *Relation* elements are obtained by treating the macromodel as a graph and forming the maximally connected subgraphs consisting of unrealized elements.

4 Prototype Implementation: MCAST

The prototype implementation MCAST is built in Java on the Eclipse-based Model Management Tool Framework (MMTF) described in [12] and leverages the Eclipse Modeling Framework (EMF) and related components. Figure 8 shows the architecture

of MCAST. MMTF already provides a simplified version of a macromodel called a Model Interconnection Diagram (MID) used as an interface for invoking the model manipulation operators and editors that can be plugged into the framework. MCAST extends this to a full macromodel editor and provides the Solver module that utilizes the Kodkod [16] model finding engine to solve model management problems expressed using annotated macromodels. We now revisit the three steps for utilizing the framework as described in Section 2 and describe how these steps are implemented using MCAST.

Step 1: Defining relationship types. In the formal treatment of section 3, relationship types are expressed using a relator metamodel plus metamodel morphisms. In the implementation we exploit the fact that EMF metamodels (i.e. Ecore) can directly reference other metamodels and thus rather than replicate the endpoint model types within the relator metamodel they are referenced as external metamodels. Axioms are expressed using a textual representation of Kodkod's relational logic language. Metamodels for model types are also expressed in this way.

Step 2: Defining a macromodel metamodel. MCAST allows metamodels for macromodels to be defined as Ecore metamodels that extend the base metamodel shown in Figure 6. Each model and relationship type is given as subclass of classes Model and Relation, respectively. In order to implement the mapping in the top part of Figure 4, these are annotated with references to the Ecore metamodels they denote.

Step 3. Managing the evolution of model collections. The Solver takes as input, a macromodel with a subset of the model and relationship elements annotated with references to existing models and relationships (i.e. relator models). It then transforms this into a Kodkod model finding problem and uses it to find solutions that assign the remainder of the elements to new models and relationships in such a way that the constraints expressed by the macromodel are satisfied. This can be used in two ways:

- Simple Conformance Mode: If the input consists of all of the realized elements assigned to existing models and relationships then a solution exists to the Kodkod problem iff this is a conformant collection and hence the Solver can be used for conformance checking.
- Extensional Conformance Mode: If simple conformance mode yields the result that the collection is non-conformant, some of the assigned models and relationships can be marked as "incomplete" and the Solver will allow these to be extended in order to find a conformant solution.

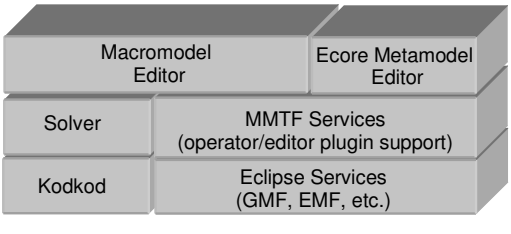

Fig. 8. MCAST Architecture

Table 1. Example Solver scenarios

Case	Input	Output
1	As in Figure 1, all marked *complete*.	Conformant
2	As in case 1 but *sentOver* relations removed from mapping *f:objectsOf*.	Non-conformant violates constraint that *sentOver* is a function.
3	As in case 2 but *f:objectsOf* marked *incomplete*	*f:objectsOf* can be uniquely extended to conformance
4	As in case 3 but the link from *:Desk* to *Loans:DB* removed.	*f:objectsOf* cannot be extended due to violation of the endpoint preservation axiom.

In extensional conformance mode, when new models and relationships are constructed as part of finding a conformant solution they are guaranteed to be consistent with the existing models/relationships but of course, this does not mean they are necessarily correct because there may be many possible consistent extensions. When the solution is unique, however, then it must be correct and hence this provides a way to do model synthesis. On the other hand, if a solution cannot be found, this indicates that there is no way to consistently extend the incomplete models/relationships and so this provides a way to do consistency checking with incomplete information.

Note that since Kodkod finds solutions by exhaustively searching a finite bounded set of possible solutions, the above results are valid only within the given bounds. Fortunately, there are common cases in which it is possible to compute upper bounds for model extension that are optimal in the sense that if a conformant extension cannot be found within the bounds then one does not exist. MCAST allows a metamodel to specify such bounds computations using special annotations within the metamodel.

We illustrate both usage modes using a macromodel consisting of the models and *objectsOf* mapping in Figure 1. Table 1 shows four cases in which we applied Solver. In case 1 we passed the models and mapping of Figure 1 (all marked as *complete*) and Solver determined that they satisfied the constraints and hence were conformant. In case 2 we removed all *sentOver* relation instances from the mapping and Solver found the models to be non-conformant because the constraint that *sentOver* is a total function from *Message* to *Link* was violated. Case 3 is the same except that the mapping was marked as *incomplete* and hence we use extensional conformance mode. In this case, we used an upper bound based on the fact that every *objectsOf* relationship is bounded by the models it relates and these would not be extended (i.e. they are marked *complete*). Solver responded by generating an extension of the mapping that filled in the missing *sentOver* links. In this case, Solver identified it as the unique extension that satisfied the constraints, thus it must be the correct one and so it had automatically synthesized the missing part of the mapping. In case 4, we modified the object diagram to remove the link from *:Desk* to *Loans:DB*. Now Solver could not find a conformant extension of the mapping and we can conclude that the models (and partial mapping) are inconsistent with the constraint that the *objectsOf*

relationship holds between the models. Cases 3 and 4 showed that it is possible to work usefully with incomplete (or even non-existent) mappings. This is significant because the creation of mappings is often given little attention in the modeling process and is considered to be overhead.

5 A Detailed Example

As a more detailed illustration of the framework we applied it[2] to a design project taken from a standards document for the European Telecommunications Standards Institute (ETSI) [8]. The example consists of three UML models: a context model (4 diagrams), a requirements model (6 diagrams) and a specification model (32 diagrams) and details the development of the Private User Mobility dynamic Registration service (PUMR) – a simple standard for integrating telecommunications networks in order to support mobile communications. More specifically, it describes the interactions between Private Integrated Network eXchanges (PINX) within a Private Integrated Services Network (PISN). The following is a description from the document:

> "Private User Mobility Registration (PUMR) is a supplementary service that enables a Private User Mobility (PUM) user to register at, or de-register from, any wired or wireless terminal within the PISN. The ability to register enables the PUM user to maintain the provided services (including the ability to make and receive calls) at different access points." [pg. 43]

Figure 9 shows the macromodel metamodel *UMLMulti* that we constructed and Figure 10 shows part of the macromodel that we used it to create it. Note that our macromodels include models as well as "diagrams" of these models. We treat a diagram as a special type of model that identifies a submodel of the model for which it is a diagram. This allows both the diagram structure within a UML model and the relational structure across UML models to be expressed within a macromodel.

Due to lack of space we do not show the definitions of the relationship types of *UMLMulti.* Please see [13] for further details.

The diagram in Figure 10 shows two sub-macromodels representing the diagrams of the context model and a subset of the diagrams of the specification model.

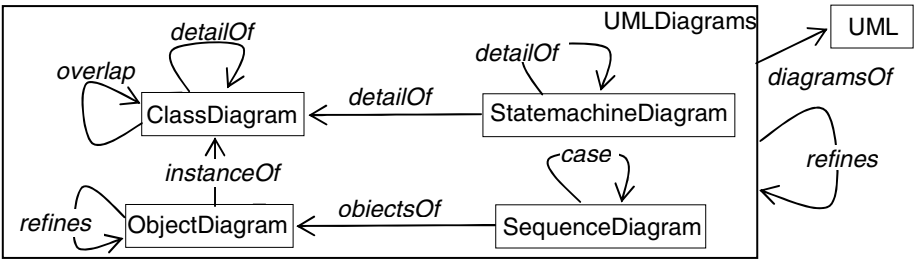

Fig. 9. UMLMulti

[2] Note that this example application was performed by hand – as future work we intend to implement it using MCAST.

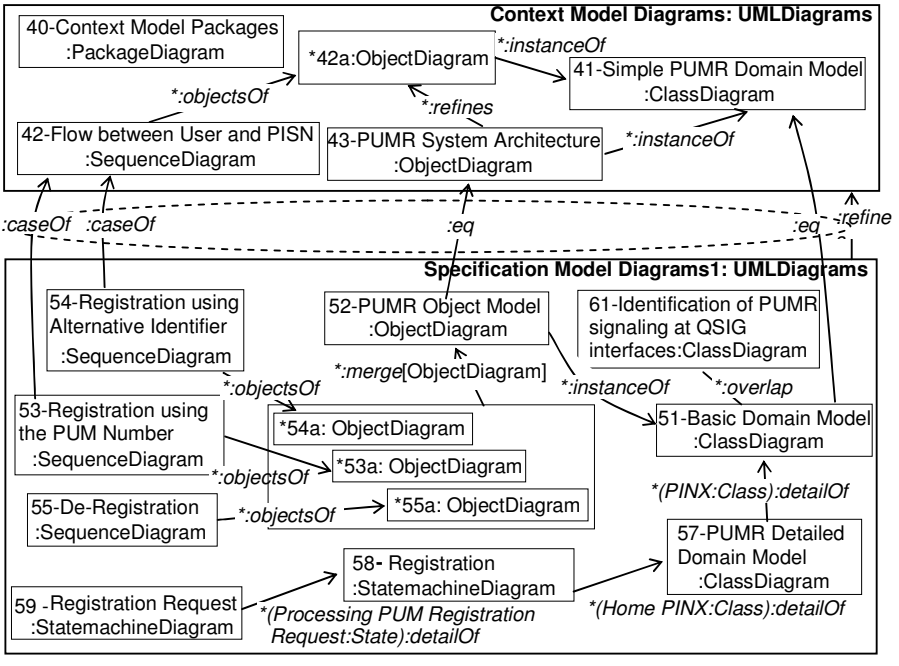

Fig. 10. PUMR Macromodel

Relationships are shown both among the diagrams within each UML model and also between the diagrams across the models. In the latter case, these are aggregated within the *refines* relationship that holds between the two collections of diagrams.

We found two interesting cases where we needed to express complex relationships using unrealized models. The relationship between sequence diagram 42 and class diagram 41 is expressed using the unrealized object diagram *42a and this is also used to show that object diagram 43 is a refinement of the objects in diagram 42. Another example is the one between the three sequence diagrams 53, 54, 55 and the object diagram 52. The macromodel shows that 52 is the smallest superset (i.e. the merge) of the object diagrams corresponding to each of these sequence diagrams.

Even without understanding the details of the PUMR domain, it should be clear how the expression of the relationships helps to expose the underlying structure in this collection of models and diagrams. In the process of constructing the macromodel, we observed that it significantly helped us to understand how the collection of diagrams contributed toward creating an overall model of the PUMR domain. Unfortunately, since this example involved an existing completed collection, we were not able to assess the hypothesis that the macromodel can be used throughout the development lifecycle to assess conformance and guide development. In order to do this, we are planning to do a more in depth case study that uses our framework from project inception through to completion.

6 Related Work

Existing work on dealing with multiple models has been done in a number of different areas. The ViewPoints framework [10] was an influential early approach to multiview modeling. Our approach differs from this work in being more formal and declarative rather than procedural. Furthermore we treat relationships as first class entities and provide support for typing of relationships.

More recently, configurable modeling environments have emerged such as the Generic Modeling Environment (GME) [7]. None of these approaches provide general support for expressing model relationships or their types; hence, they have limited support for defining and expressing interrelated collections of models. Furthermore, the focus of these approaches is on the detail level (i.e. the content of particular models) rather than at the macroscopic level.

Process modeling approaches like the Software Process Engineering Metamodel (SPEM) [15] bear some similarity to our notion of a macromodel metamodel; however, our main focus is in the use of a macromodel at the instance level to allow fine-grained control over the ways in which particular models are related rather than the activities that consume and produce them. However, we believe that macromodels could complement process models by providing a means for specifying pre and post conditions on process activities.

The emerging field of Model Management [3] has close ties to our work but our focus is different in that we are interested in supporting the modeling process whereas the motivation behind model management is primarily model integration.

The term "megamodel" as representing models and their relationships at the macroscopic level emerged first in the work of Favre [4] and also later as part of the Atlas Model Management Architecture (AMMA) [2]. Macromodels bear similarity to these two kinds of megamodels, but the intent and use is quite different – to express the modeler's intentions in a development process.

Finally, the work on model traceability also deals with defining relationships between models and their elements [1]; however, this work does not have a clear approach to defining the semantics of these relationships. Thus, our framework can provide a way to advance the work in this area.

7 Conclusions and Future Work

By its very nature, the process of software development is an activity involving many interrelated models. Much of the research and tools for modeling is focused on supporting work with individual models at the detail level. Working with collections of models creates unique challenges that are best addressed at a macroscopic level of models and their inter-relationships.

In this paper we have described a formal framework that extends a modeling paradigm with a rich set of model relationship types and uses macromodels to manage model collections at a high level of abstraction. A macromodel expresses the relationships that are intended to hold between models within a collection. We have focused on two main ways that macromodels can support modeling. Firstly, they are tools for helping the comprehension of the collection by revealing its intended

underlying structure. Secondly, macromodels can be used to help maintain the model relationships as a collection evolves. In this capacity they are used to guide the development process by ensuring that modelers intentions are satisfied.

Finally, we described the prototype implementation MCAST that integrates the Kodkod model finding engine [16] as a way to support model management activities using macromodels. As part of future work, we are exploring other ways to use a macromodel to manipulate collections of models.

References

1. Aizenbud-Reshef, N., Nolan, B.T., Rubin, J., Shaham-Gafni, Y.: Model Traceability. IBM Systems Journal 45(3), 515–526 (2006)
2. ATLAS MegaModel Management website, http://www.eclipse.org/gmt/am3/
3. Bernstein, P.: Applying Model Management to Classical Meta Data Problems. In: Proc. Conf. on Innovative Database Research, pp. 209–220 (2003)
4. Favre, J.M.: Modelling and Etymology. Transformation Techniques in Software Engineering (2005)
5. Goguen, J.A., Burstall, R.M.: Institutions: Abstract Model Theory for Specification and Programming. J. ACM 39(1), 95–146 (1992)
6. Kalfoglou, Y., Schorlemmer, M.: Ontology mapping: the state of the art. The Knowledge Engineering Review 18(1), 1–31 (2003)
7. Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason IV, C., Nordstrom, G., Sprinkle, J., Volgyesi, P.: The Generic Modeling Environment. In: Workshop on Intelligent Signal Processing (2001)
8. Methods for Testing and Specification (MTS); Methodological approach to the use of object-orientation in the standards making process. ETSI EG 201 872 V1.2.1 (2001-2008), http://portal.etsi.org/mbs/Referenced%20Documents/eg_201_72.pdf
9. Moody, D.: Dealing with 'Map Shock': A Systematic Approach for Managing Complexity in Requirements Modelling. In: Proceedings of REFSQ 2006, Luxembourg (2006)
10. Nuseibeh, B., Kramer, J., Finkelstein, A.: A Framework for Expressing the Relationships Between Multiple Views in Requirements Specifications. IEEE TSE 20(10), 760–773 (1994)
11. Sabetzadeh, M., Easterbrook, S.: An Algebraic Framework for Merging Incomplete and Inconsistent Views. In: 13th IEEE RE Conference, Paris, France (2005)
12. Salay, R., Chechik, M., Easterbrook, S., Diskin, Z., McCormick, P., Nejati, S., Sabetzadeh, M., Viriyakattiyaporn, P.: An Eclipse-Based Tool Framework for Software Model Management. In: ETX 2007 at OOPSLA 2007 (2007)
13. Salay, R.: Macro Support for Modeling in Software Engineering. Technical Report, University of Toronto, http://www.cs.toronto.edu/~rsalay/tr/macrosupport.pdf
14. Salay, R., Mylopoulos, J., Easterbrook, S.: Managing Models through Macromodeling. In: Proc. ASE 2008, pp. 447–450 (2008)
15. Software Process Engineering Metamodel V1.1. Object Management Group, http://www.omg.org/technology/documents/formal/spem.htm
16. Torlak, E., Jackson, D.K.: A Relational Model Finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424. Springer, Heidelberg (2007)