Macro Support for Modeling in Software Engineering

Technical Report V.2.5

by
Rick Salay

1. Introduction

In Software Engineering, we typically model software using multiple "partial" models; thus, we are actually doing *multimodeling*. There are several reasons for this. Firstly, different views of the software are best captured using a modeling language most appropriate for that view. Secondly, the complexity of a model of a piece of software requires that it be decomposed into smaller parts and descriptions at different levels of abstraction. Thirdly, the widely accepted principle of "separation of concerns" requires that different models be created to address different purposes. Finally, different models can express the viewpoints of different stakeholders. Furthermore, the use of multiple models is typically supported by the development process - most contemporary development processes, such as RUP [13], mandate the use of multiple views and require some form of iteration in which a series of progressively more detailed models are created.

Much of the focus of modeling research is on what goes in within an individual model – i.e. what it takes to create or validate the content of a model so that it correctly reflects the modeled domain. We refer to this as the "micro" level of modeling. When we consider multimodeling, a "macro" level emerges as well where models and the relations between models become the objects of study. The central premise of this paper is that this macro level can provide opportunities for providing useful support to the software development process.

The emerging area of Model Management [2, 5, 3] also takes this macro view by defining a set of generic operators such as Merge and Diff to manipulate models and their "mappings." Here, the models of concern are schemas for data or process, such as database schemas or workflow definitions and the focus is on reducing the effort in the corresponding schema integration problem – i.e. how to create mapped and merged schemas to support interoperability. Our focus is slightly different. We are primarily interested in providing support at design time to the process of creating, understanding and manipulating the set of models of a software development process.

To some extent, the operators suggested within Model Management are applicable in this context as well. For example, model merging has been well studied in the context of Requirements Engineering [14]. However, while Model Management recognizes only one type of generic relationship between models called a "mapping," we find it necessary to define many types, just as there are many types of models. The need for this follows from the observation that different types of model relations arise in software engineering and that they correspond to different tasks: refinement relation types for elaborating models, refactoring relation types for model evolution, overlap relations for merging or decomposing models, functional types for transforming models and relations into other types of models and relations, etc. Furthermore, the introduction of relation types leads to a more unified approach to model management itself since model management operators are just particular kinds of relation types (functional ones) and hence they themselves can be expressed within the same scheme. Finally, as with model types, relation types can be defined using metamodels and hence support for this can be borrowed from existing work on metamodeling.

1.1. Motivating Scenarios

In order to motivate the above discussion, we present three example scenarios of providing useful support to modeling at the macro level. These will be further elaborated in Section 3 after introducing the required formalism in Section 2.

1.1.1. Scenario 1: Exposing Multimodel Structure

Consider a software project with a UML model consisting of large number of diagrams of various kinds. One could document the intent of each diagram but it is difficult to apprehend a large set of diagrams and how they are related to one another, especially since UML provides no way of indicating the relationships between these diagrams. If these relations were made explicit and were classified, this would help. If the relations could be automatically inferred from the underlying UML model, this would be even better. Figure 1.1 illustrates a fragment of a "macromodel" showing the diagrams and their relations for an example domain taken from a telecommunications standards document. Even without understanding the domain here it should be clear how the expression of the relations between models helps to explicate the underlying structure in this multimodel. We will return to this example in Section 3.1.

1.1.2. Scenario 2: Structuring the Development Process

Assume that the UML model for a project was only partially complete; however, as chief architect, we know what diagrams need to be created and how they should be related to each other and to existing diagrams. For example, we may assign the task of creating the state machine diagram for a given class in a known class diagram to developer Jack. It would be useful if we could assert the type of relationship that the state machine diagram has with the class diagram. This could be used to partially automate the check for consistency of the diagram that Jack produces and hence improve the efficiency of the development process and quality of the design. Establishing such relations can be used to structure the development process.

1.1.3. Scenario 3: Model Manipulation Operators

Consider the refinement of communication diagrams for a hypothetical library management system shown in Figure 1.2. Here Ref1 denotes the refinement relation and although it is not shown here, it actually has internal structure that contains the internal details of how the elements of M1 are mapped to the elements of M2. Furthermore, this refinement relation has a very specific structure - it is an instance of a type of relation in which the communication diagram on the left has single message and the one on the right shows a refinement for this message. Call this type of relation SimpleRefCommD.

Now consider the communication diagram refinement shown in Figure 1.3. Ref2 is also a refinement relation, although not an instance SimpleRefCommD. Call Ref2's type of refinement relation, RefCommD. Note that given M3 and Ref1 we have enough information to infer Ref2 and hence, M4. In particular, we get Ref2 by applying Ref1 to each message in M3

that it matches and then M4 is just the right hand communication diagram of the result. It might be a useful capability if we were able to define this procedure in general as an operator so that we could always "automatically refine" a communication diagram using a known simple refinement since this would provide a way to modularize and reuse refinements.

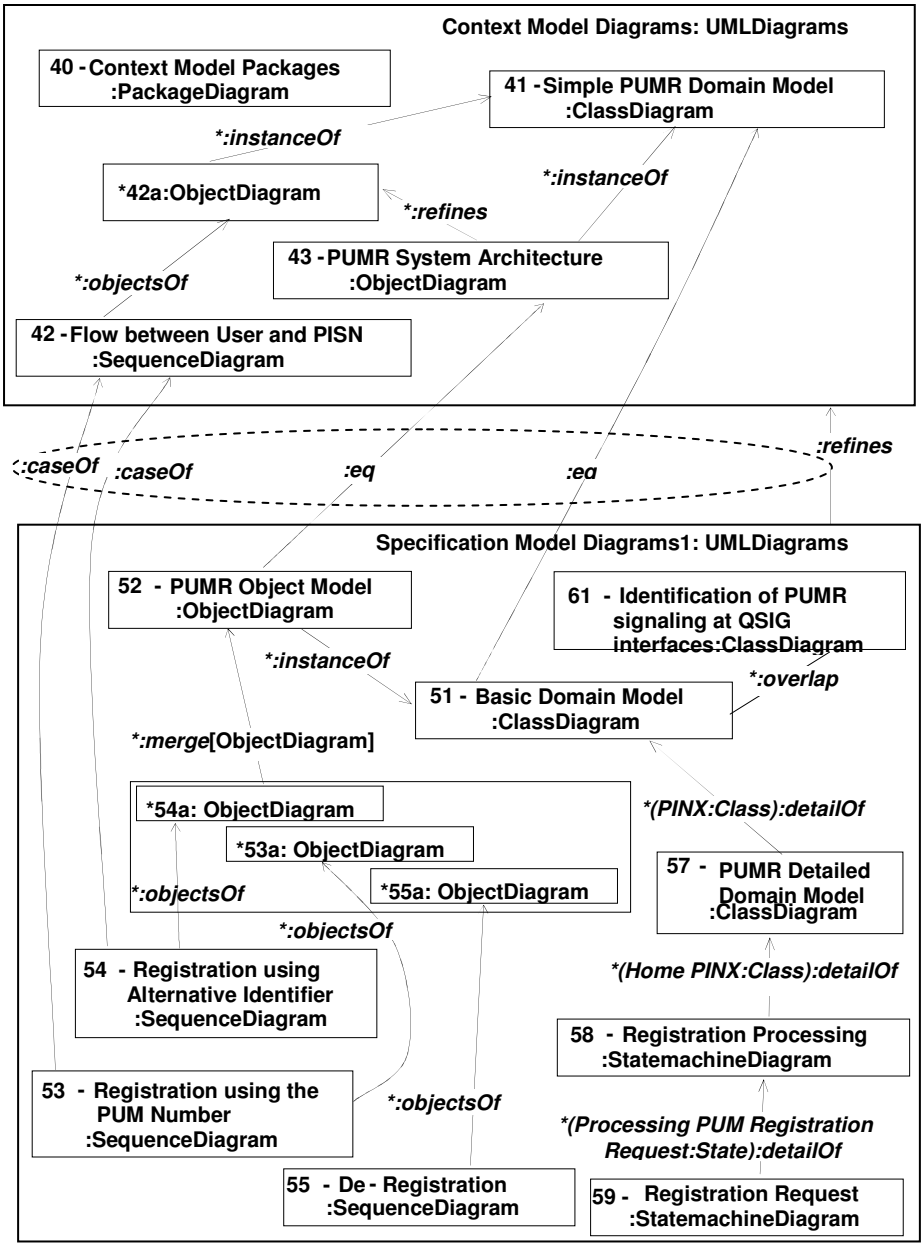


Figure 1.1: Telecommunications macromodel example

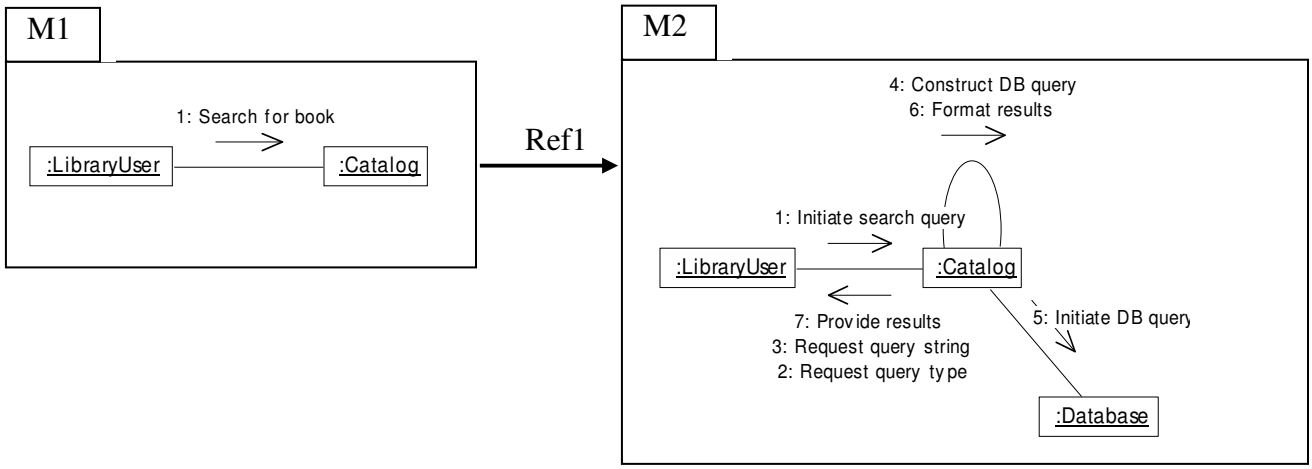


Figure 1.2: A sequence diagram refinement

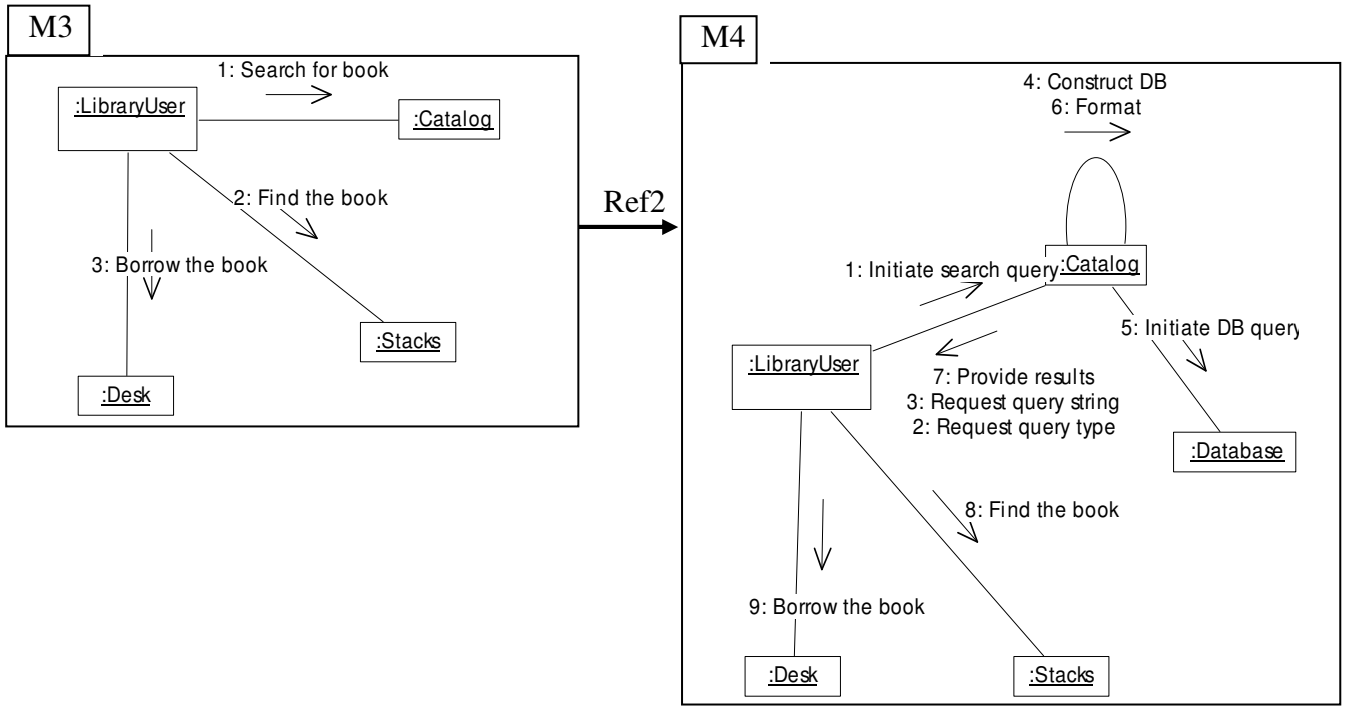


Figure 1.3: A derived refinement

2. Foundations

This section describes the foundational concepts, theory and formalism of our approach to multimodeling.

2.1. Assumptions about Models and Model Relations

Within software engineering, models are used to specify or describe something – usually a software system. If we call the thing that the model is about, the *subject*, then we can take a model to be equivalent to some set of statements about the subject [15]. Thus, a model is a structure (syntax) that has an interpretation (semantics).

A distinction is often made between the abstract and the concrete syntax of a model, where the latter consists of the actual shapes, lines, characters, etc. from which a model is constructed while the former abstracts away from these and instead just expresses a model in terms of the different types of symbols needed. For example, the concrete syntax of a UML class diagram includes boxes and lines while the abstract syntax talks about class symbols and association symbols. Although the concrete syntax is an important aspect of a model and its effectiveness as a language [8], in this paper, we will assume that models are characterized in terms of their abstract syntax only. Thus, the structure of a model will consist of a set of abstract symbols and abstract relations between them.

Following the assumption that model are like (sets of) statements and are hence, "linguistic" entities, we can identify three aspects of a relationship that may hold between models:

- Syntactic relation: two models are related syntactically iff the structure of one is constrained by the structure of the other. For example, models that share some symbols are syntactically related.
- Semantic relation: two models are related semantically if interpretations of one model constrain the possible interpretations of the other.
- Pragmatic relation: a pragmatic relation between models is one that occurs in the context of their use. For example, the creation of one model must precede another in a methodology, one model is a newer version of another model, one model has the same creator as another model, etc.

These aspects of relations can co-occur. For example, if model A is newer version of model B, then it is pragmatically related. In addition, since they are about the same domain, they are semantically related and finally, B may share much structure with A and hence they may be syntactically related. In this paper, we focus on syntactic and semantic relations – i.e. relations that have to do with the content of the model rather than the context of use.

2.2. Metamodels and Model Types

A metamodel is used to specify a class of models with similar characteristics – i.e. a *model type*. If MM is a metamodel, then we will denote the model type it specifies as [[MM]]. For example, we may have the metamodel StateDiagram where [[StateDiagram]] is the set of all state diagrams. Note that the same model type could be specified by many metamodels. For example, if we have the OMG metamodeling language MOF [9] and the Eclipse metamodeling language Ecore [6], we may have StateDiagram:MOF $\neq$ StateDiagram:Ecore but [[StateDiagram:MOF]] = [[StateDiagram:Ecore]] = [[StateDiagram]]. Furthermore, not every model type may be expressible in a given metamodeling language. For example, if Model is the model type consisting of all models, it is not hard to show that there is no MOF metamodel MM:MOF such that [[MM:MOF]] = Model[1].

If M is a particular state diagram, then it is an instance of the model type [[StateDiagram]] but we will extend this idea to its metamodel(s) and also say that M is an instance of StateDiagram. That is, a model will be considered both to be an instance of its model type and any metamodel that specifies the model type. Following usual conventions for typing we will write this as M:StateDiagram.

2.2.1. Metamodeling Language

Although models have both syntax and semantics, it is common for the metamodel to only characterize the syntactic aspect of the models in a given model type. This is the case with both MOF and Ecore. One reason for this may be that while the syntax of a modeling language is often easily formalizable, the semantics is typically not. For example, the UML specification expresses the abstract syntax as a metamodel but the semantics is expressed using english [17] - the formalization of UML semantics continues to be an area of intense research.

In this paper, we follow a similar tack; however, rather than using either MOF or Ecore, we have chosen to use a version of order-sorted first order logic with transitive closure (henceforth referred to as FO+) as the metamodeling formalism. There are a number of reasons for this. Firstly, it is widely known and has comparable expressive power to the other metamodeling approaches. Secondly, it has a textual representation and this is more convenient when discussing formal issues. Finally, general theoretical concepts, such as logical consequence, are well understood in this context.

Using FO+ we can define the abstract syntax of a model type as a pair $\langle \Sigma, \Phi \rangle$ where Σ is called the signature and defines the types of model elements and how they can be related while Φ is a set of axioms that defines the well-formedness constraints for models. Thus, a metamodel $\langle \Sigma, \Phi \rangle$ is a presentation of an FO+ theory and each finite model (in the model theoretic sense) of this theory will be considered to be a model that is an instance of the metamodel.

[1] Say MM:MOF defines n model element types. Since Model contains all models, we can always find one that has n+1 element types and hence MM:MOF can't be the metamodel.

For example, we define the abstract syntax of a simplified UML class diagram as follows.

CD =

sorts class, association, attribute, operation

func startClass: association → class

endClass: association → class

attrClass: attribute → class

opClass: operation → class

pred subClassOf: classSymb × classSymb

axioms

'a class cannot be a subclass of itself'

$\forall c$: ¬TC(subClassOf(c, c))

Here, the combination of the **sorts**, **func** and **pred** sections describe the signature Σ and the **axioms** section describes Φ. The sorts define the element types that can occur in a diagram while the functions and relations show how they are connected in a diagram. To express the constraint that no class can be a subclass of itself, we use the transitive closure operator TC on the subClassOf relation. We need the ability to compute the transitive closure of a relation in order to define acyclicity constraints such as this within a model since these cannot be expressed in first order logic.

For simple cases we can also show a metamodel signature diagrammatically as in Figure 2.1. The nodes represent sorts, edges like "⟶" denote directed binary relations and edges like "⟶" denote functions with one argument. However, note that the complete metamodel includes the axioms.

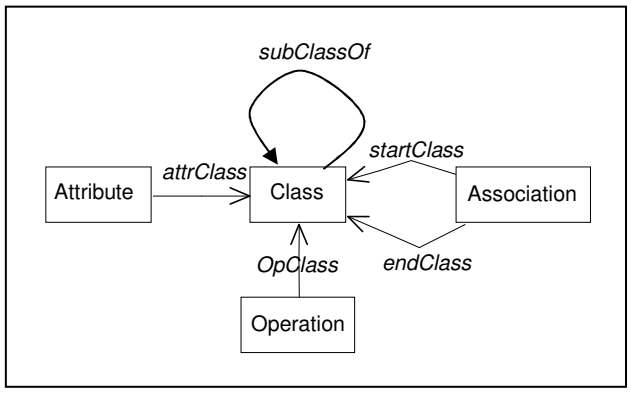


Figure 2.1: Metamodel of CD

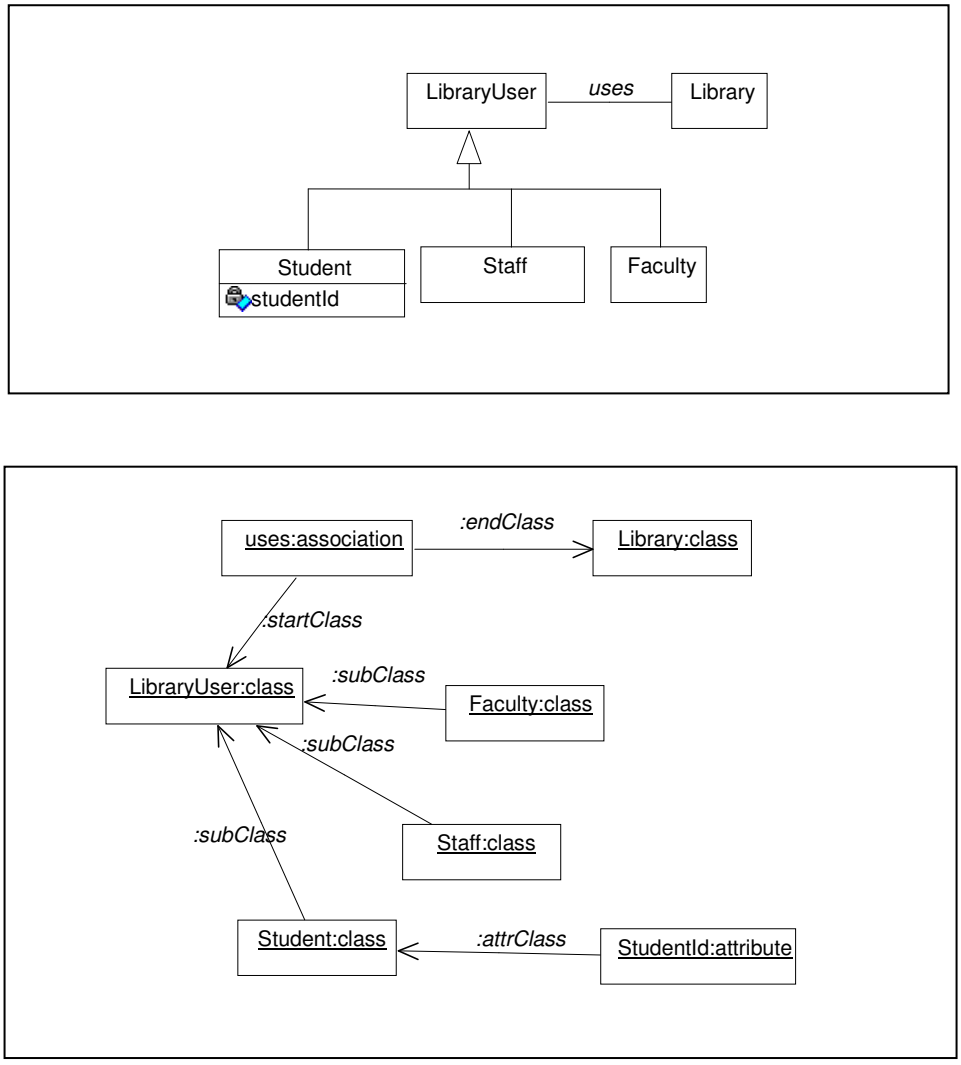


Figure 2.2: A class diagram and its abstract syntax as an instance of CD

Figure 2.2 shows a class diagram based on the library management system example both using its concrete syntax and its abstract syntax as an instance of CD (shown as a UML object diagram).

2.2.2. Formalization

We briefly review the formalization of FO+[2]. A signature Σ is a tuple $\langle S, F, P, \leq, \alpha_F, \alpha_P \rangle$ which is defined as:

- A set of sort symbols $S = \{S_1, S_2, \ldots S_k\}$ and $\leq \subseteq S \times S$ gives the order relation over sorts.

[2] For a more detailed description of order sorted first order logic please see [ref].

- A set of function symbols $F = \{f_1, f_2, \ldots f_m\}$ and an arity function $\alpha_F: F \rightarrow S^*$ giving, for each function, the result sort followed by the list of argument sorts.
- A set of predicate symbols $P = \{p_1, p_2, \ldots p_n\}$ and an arity function $\alpha_P: P \rightarrow S^*$ giving, for each predicate, the list of argument sorts.

Here, S* denotes a (possibly empty) list of sorts. A function with no arguments is a constant of the type given by the result sort. We define the following "helper" definitions. Let $\#f = |\alpha_F(f)| - 1$ - i.e. it is the number of arguments of function f. Also, $\alpha_F(f)[i+1]$ is the ith argument sort of f and $\alpha_F(f)[1]$ is the result sort. Similarly, we define $\#p = |\alpha_P(p)|$ and $\alpha_P(p)[i]$ for predicates.

Finally, we define Sen(Σ) to be the usual set of first order sentences (plus TC) over the signature Σ. Thus, for a metamodel $\langle \Sigma, \Phi \rangle$, we have that $\Phi \subseteq Sen(\Sigma)$.

In order to define the semantics of a metamodel[3] we first recall the standard notion of a first order interpretation. An interpretation τ of Σ is an assignment that maps

- Each sort symbol S to a set $[[S]]_\tau$, such that if $S_i \leq S_j$ then $[[S_i]]_\tau \subseteq [[S_j]]_\tau$
- Each predicate symbol p with $\alpha_P(p) = \langle S_1, S_2, \ldots S_n \rangle$ to a relation $[[p]]_\tau \subseteq [[S_1]]_\tau \times [[S_2]]_\tau \times \ldots \times [[S_n]]_\tau$
- Each function symbol f with $\alpha_f(f) = \langle S, S_1, S_2, \ldots S_n \rangle$ to a function $[[f]]_\tau: [[S_1]]_\tau \times [[S_2]]_\tau \times \ldots \times [[S_n]]_\tau \rightarrow [[S]]_\tau$

Let Mod(Σ) denote the set of all interpretations of Σ and define the relation $\models \subseteq Sen(\Sigma) \times Mod(\Sigma)$ to be the standard satisfaction relation for order sorted first order logic with transitive closure. Thus, for sentence $\varphi \in Sen(\Sigma)$, $\tau \models \varphi$ means that τ satisfies φ (i.e. φ holds true in τ). For a metamodel $\langle \Sigma, \Phi \rangle$, define $Mod(\Sigma, \Phi)$ be the set of interpretations that satisfy all the sentences in Φ. Since we are only interested in models that have a finite number of symbols, we can define the specified model type $[[\langle \Sigma, \Phi \rangle]] \subseteq Mod(\Sigma, \Phi)$ to be the subset of these interpretations that have finite sorts.

We will use an algebraic specification style of expressing metamodels as shown in above in Section 2.2.1; however, rather than formalize this notation at the outset, we will just use it and discuss any new constructs as they arise. We do this because some of the constructs are closely tied to the theoretical framework we are building and only become understandable in the context of the associated exposition.

2.2.3. Metamodel Morphisms and Reducts

In the next section, our goal will be to define model relation types in terms of metamodels. In order to do this we require a way to show how metamodels can be related and how this corresponds to how the model types they specify are related. To do this, we draw on a basic

[3] Not to be confused with the semantics of the models described by the metamodel.

concept from theory of institutions within algebraic specification [7]. The idea here is that since both the sentences $\mathrm{Sen}(\Sigma)$ and interpretations $\mathrm{Mod}(\Sigma)$ are completely determined by the signature Σ, if we map one signature into another, this should induce corresponding mappings between their sentences and interpretations as well. Furthermore, when these mappings preserve the satisfaction relation then it effectively shows how to map one logical theory into another in a sound way. Since we are using logical theories to express metamodels and their interpretations to express models, this means that it shows how to map metamodels and the model types they specify.

We first define the signature mapping as a *signature morphism*. Given signatures Σ_1 and Σ_2, a signature morphism $f:\Sigma_1 \rightarrow \Sigma_2$ is a mapping from the sorts, functions and predicates of one signature to those of the other such that the subsort relation and arity is preserved. Formally, f consists of injective[4] functions $\langle f_S:S_1 \rightarrow S_2, f_F:F_1 \rightarrow F_2, f_P:P_1 \rightarrow P_2 \rangle$ such that the following conditions hold:

1. $\forall S_i, S_j \in S_1.\ S_i \leq_1 S_j \Rightarrow f_S(S_i) \leq_2 f_S(S_j)$
2. $\forall f \in F_1.\ \#f = \#f_F(f) \wedge \forall i \in \{1, \ldots, \#f + 1\}.\ \alpha_{F2}(f_F(f))[i] = f_S(\alpha_{F1}(f)[i])$
3. $\forall p \in P_1.\ \#p = \#f_P(p) \wedge \forall i \in \{1, \ldots, \#p\}.\ \alpha_{P2}(f_P(p))[i] = f_S(\alpha_{P1}(p)[i])$

Intuitively, a signature morphism $f:\Sigma_1 \rightarrow \Sigma_2$ is a mapping that shows how to "rename" elements of Σ_1 into (a subset of) elements of Σ_2. We can use this to define a natural translation function $\mathcal{S}f:\mathrm{Sen}(\Sigma_1) \rightarrow \mathrm{Sen}(\Sigma_2)$ for sentences that replace the sort, function and predicate symbols in each sentence by their image under f.

Now, given metamodels $\langle \Sigma_1, \Phi_1 \rangle$ and $\langle \Sigma_2, \Phi_2 \rangle$, a *metamodel morphism* $f: \langle \Sigma_1, \Phi_1 \rangle \rightarrow \langle \Sigma_2, \Phi_2 \rangle$ consists of a signature morphism $f_\Sigma:\Sigma_1 \rightarrow \Sigma_2$ such that $\Phi_2 \models \mathcal{S}f_\Sigma(\Phi_1)$. Intuitively, this means that the axioms of $\langle \Sigma_1, \Phi_1 \rangle$ follow logically from the axioms of $\langle \Sigma_2, \Phi_2 \rangle$ after the signature renaming by f_Σ.

In the same way that f_Σ induces the sentence translation function $\mathcal{S}f_\Sigma:\mathrm{Sen}(\Sigma_1) \rightarrow \mathrm{Sen}(\Sigma_2)$ it also induces the *reduct* function $\mathcal{M}f_\Sigma:\mathrm{Mod}(\Sigma_2, \Phi_2) \rightarrow \mathrm{Mod}(\Sigma_1, \Phi_1)$ that uses f_Σ to transform each model of $\langle \Sigma_2, \Phi_2 \rangle$ into a corresponding model of $\langle \Sigma_1, \Phi_1 \rangle$. For each $\tau 2 \in \mathrm{Mod}(\Sigma_2, \Phi_2)$, we define $\tau 1 = \mathcal{M}f_\Sigma(\tau 2)$ as follows:

1. $\forall S \in S_1, [[S]]_{\tau 1} = [[f_\Sigma(S)]]_{\tau 2}$
2. $\forall f \in F_1, [[f]]_{\tau 1} = [[f_\Sigma(f)]]_{\tau 2}$

[4] Relaxing this restriction yields a more general type of signature morphism but for purposes the this paper, injective functions are what we need and this restriction simplifies some of the technical details.

3. $\forall p \in P_1, [[p]]_{\tau 1} = [[f_\Sigma(p)]]_{\tau 2}$

Thus, $\tau 1$ is formed from $\tau 2$ by discarding any sorts, functions and predicates not found in the image of f_Σ. This is why $\mathcal{M}f_\Sigma$ could be considered to be a projection function that extracts a $\langle \Sigma_1, \Phi_1 \rangle$ model embedded within each $\langle \Sigma_2, \Phi_2 \rangle$ model. For notational simplicity, we will henceforth drop the signature morphism indicator and just write $\mathcal{S}f$ and $\mathcal{M}f$ for metamodel morphism f.

In order for reducts to work as we expect them to, we must also show that the satisfaction condition for institutions holds:

$$\forall M \in Mod(\Sigma_2, \Phi_2), S \in Sen(\Sigma_1, \Phi_1).\ \mathcal{M}f(M) \models_1 S \Leftrightarrow M \models_2 \mathcal{S}f(S)$$

We do not prove this here but direct the reader to [1] in which proofs are provided for a wide range of similar logics.

Figure 2.3 illustrates the notions of metamodel morphisms, sentence translations and reducts.

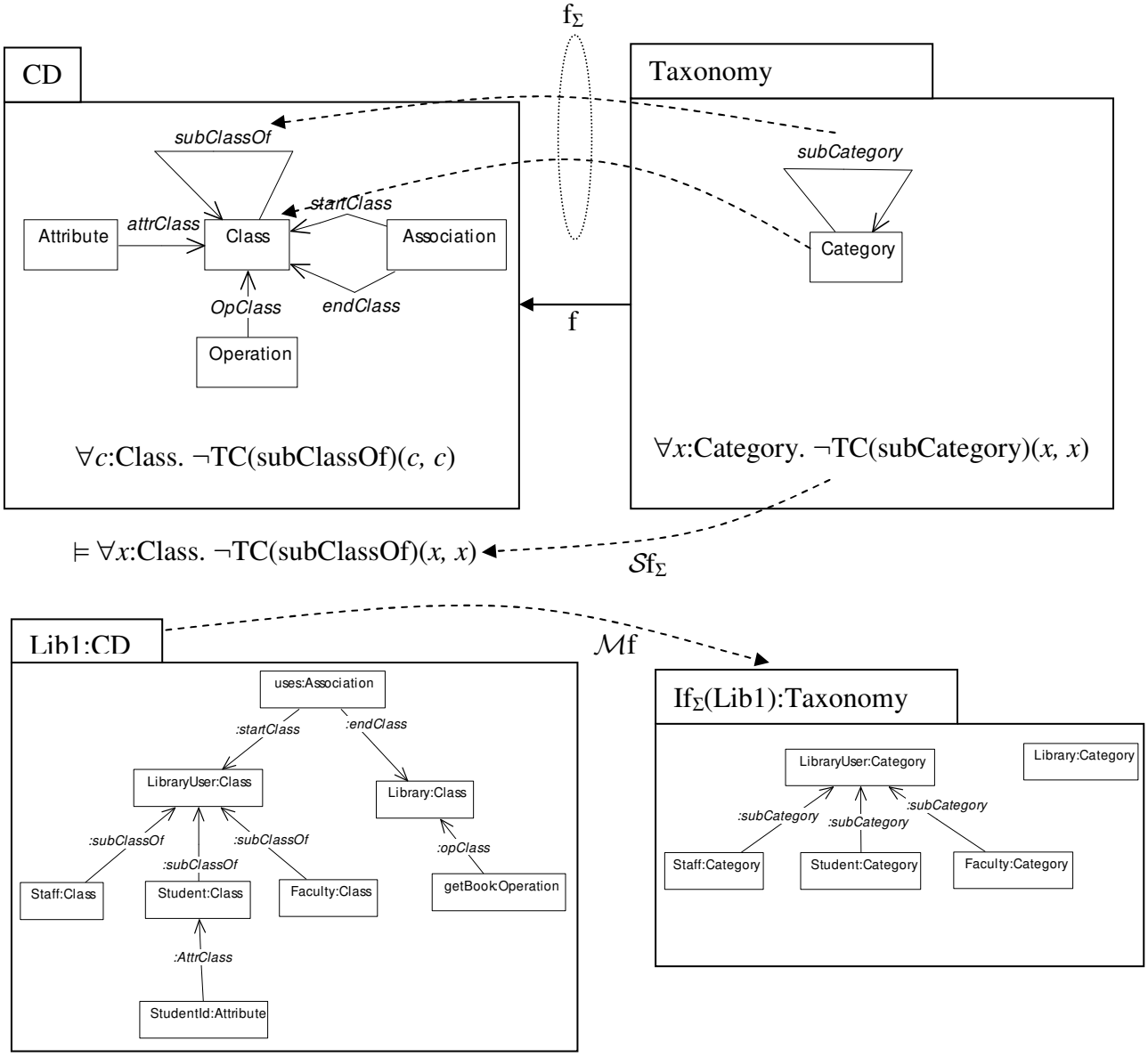


Figure 2.3: Example of a metamodel morphism

Here, we relate the metamodels Taxonomy = $\langle \Sigma_{\text{Taxonomy}}, \Phi_{\text{Taxonomy}} \rangle$ and CD = $\langle \Sigma_{\text{CD}}, \Phi_{\text{CD}} \rangle$. The metamodel morphism f:Taxonomy → CD has f_Σ which maps the sort Category to the sort Class and the predicate subCategory to the predicate subClassOf. With the resulting sentence translation function we see that $\mathcal{S}f$("$\forall x$:Category. ¬TC(subCategory)(x, x)") = "$\forall x$:Class. ¬TC(subClassOf)(x, x)" which clearly follows logically from Φ_{CD} = {$\forall c$:Class. ¬TC(subClassOf)(c, c)}. The bottom portion of the figure shows how an example of a particular model Lib1:CD is transformed via the reduct $\mathcal{M}f$ into an instance of Taxonomy.

Note that the transformation just removes all instances of all sorts, functions and predicates other than Class and subClassOf, and these are converted into instances of Catetgory related by subcategory.

2.3. Relation Models and Relation Model Types

Two models are related when the possible interpretations of one model constrain the possible interpretations of the other model. Thus, at the semantic level it is a binary relation over the sets of satisfying interpretations [2]. At the syntactic level, a relation between models can be expressed as larger model (relator model) containing the related models (component models). For example Figure 2.4 shows how class diagram M1:CD and communication diagram M2:CommD for the library management domain can be related by embedding them in relator model R1:RCC. Here, R1:RCC contains M1:CD, M2:CommD, a function mapping the objects in M2 to classes in M1 and a function mapping from the messages in M2 to operations in M1. We will call such a relator model, along with the designations of the embeddings of component models, a model relation. A central premise of this paper is that a fruitful thing to do is to classify model relations into model relation types. The approach we use for doing this is to define projection functions from the model type of the relator model to the model types of the component models that show how to "extract" the component models from each instance of the relator model. In Figure 2.4, the model type of the relator model is [[RCC]] and the two projection functions are π_{CD} and π_{CommD}.

2.3.1. Model Relation Types using Metamodels

In Figure 2.4, $\langle[[RCC]], \pi_{CD}, \pi_{CommD}\rangle$ defines a model relation type defined between model types CD and CommD, and we can assume that there is a metamodel RCC that characterizes the relator models of this relation type. Furthermore, the natural thing to consider is that π_{CD} and π_{CommD} are actually the reducts corresponding to two metamodel morphisms p_{CD}:CD→RCC and p_{CommD}:CommD→RCC that map these component metamodels into relator metamodel. Thus we can get π_{CD} and π_{CommD} "for free" since $\pi_{CD} = \mathcal{M}p_{CD}$ and $\pi_{CommD} = \mathcal{M}p_{CommD.}$ Figure 2.5 illustrates this.

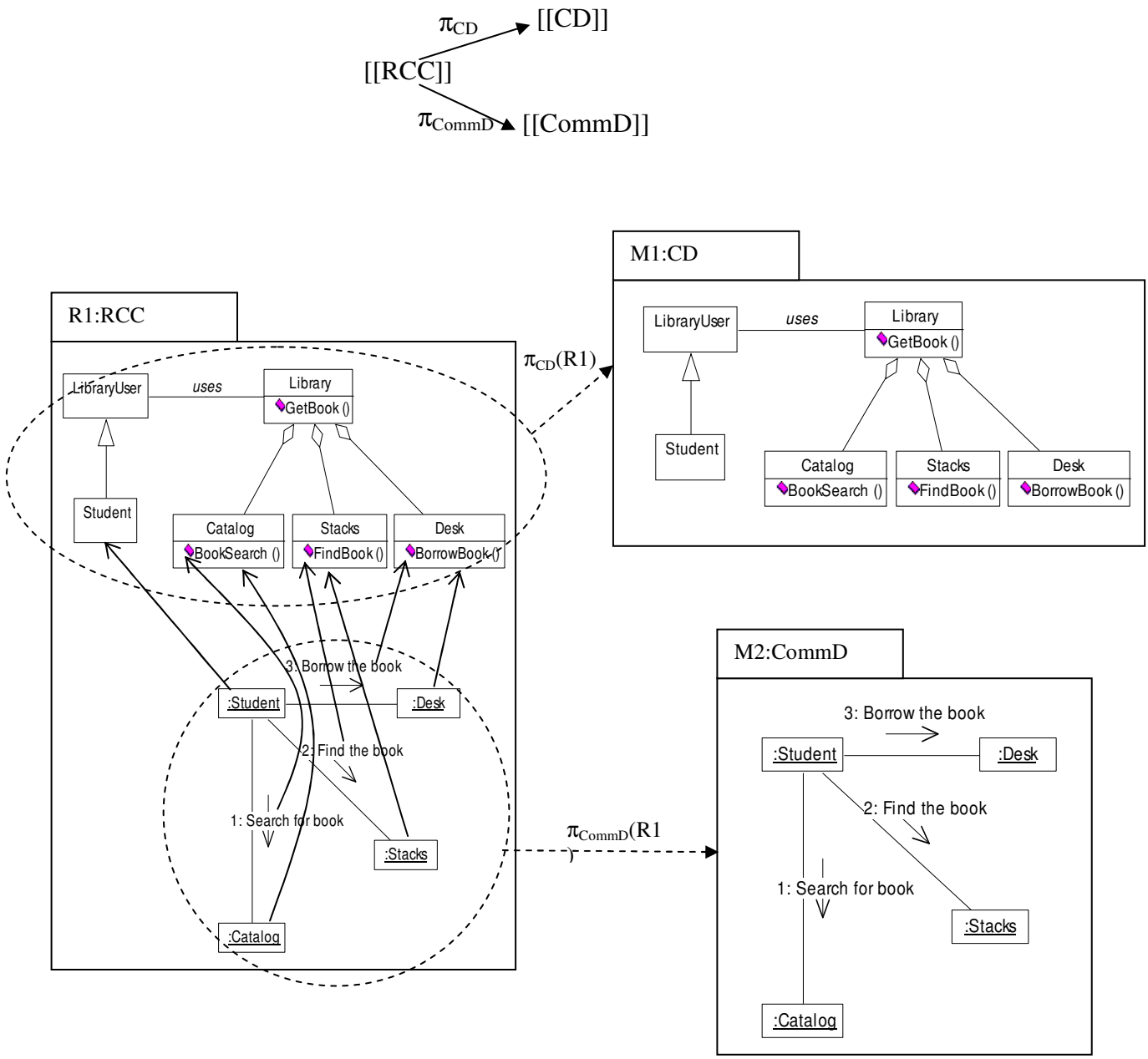


Figure 2.4: Relation between a communication diagram and a class diagram

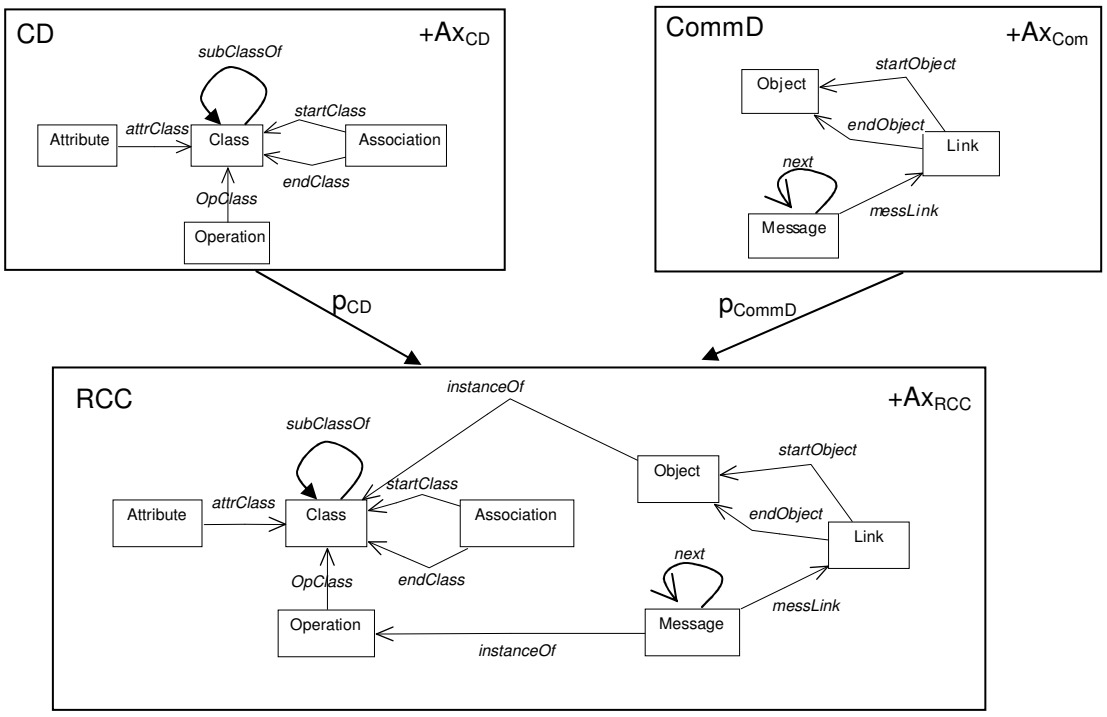


Figure 2.5: RCC in Terms of Metamodels

CD is defined as in Section 2.2.1. The details of CommD and RCC are as follows:

CommD =

sorts object, message, link

func first: message, last:message

startObject: link → object

endObject: link → object

messLink: message → link

next:message o→ message

axioms

'next'

'the message "first" has no predecessor'

$\forall x$:message. $\neg$(first = next(x))

'only the message "last" has no successor'

$\forall x$:message. $x \neq$ last $\Leftrightarrow \exists y$:message. y = next(x)

RCC = CD + CommD +

func instanceOf: object → class

instanceOf: message → operation

axioms

'instanceOf maps messLink to opClass'

$\forall x$:message.opClass(instanceOf(x)) =

instanceOf(endObject(messLink(x)))

In addition, we assume that p_{CD}:CD→RCC and $_{CommD}$:CommD→RCC are the obvious inclusion metamodel morphisms.

Note the following notational extensions used in these metamodels. The constants "first" and "last" are given as functions with no arguments and only a return type specified. The function "next" in CommD is a partial function since the last message has no next message. We designate a partial function using "o→". The metamodel RCC includes the metamodels for CD and CommD and this is expressed by the "RCC = CD + CommD +".

Each instance of RCC has the two "instanceOf" functions that map objects to classes and messages to operations. In addition, the axiom of RCC requires that if a message maps to an operation then the class of the receiver of the message should contain that operation.

Clearly, the reducts $\mathcal{M}p_{CD}$:[[RCC]]→[[CD]] and $\mathcal{M}p_{CD}$:[[RCC]]→[[CommD]] extract the CD and CommD out of each RCC instance. Thus ⟨[[RCC]], $\mathcal{M}p_{CD}$, $\mathcal{M}p_{CommD}$⟩ is the desired model relation type.

We can now generalize the procedure with RCC as follows. Given metamodels M_1, …, M_n, a relator metamodel R and metamodel morphisms f_1:M_1→R, …, f_1:M_1→R, we can define the model relation type ⟨[[R]], $\mathcal{M}f_1$, …, $\mathcal{M}f_n$⟩. Correspondingly, we will say that the metamodel for the relation type is ⟨R, f_1, f_2, …, f_n⟩ and call it a relation metamodel.

2.3.2. Soundness of a Relation Type

In the above example, the axiom in RCC eliminates inconsistent relations as instances. For example, say that it was possible to define a relator R2:RCC that was the same as R1:RCC in Figure 2.4 except that the message "Search for Book" maps to the operation "BorrowBook" (this violates the constraint). Under the usual, interpretations of CD and CommD, there could be no interpretation for M1:CD and M2:CommD that are related via R2:RCC since the receiver object of "Search for Book" does not contain the operation "BorrowBook." This means that the set of interpretations for R2:RCC is empty and hence it is an inconsistent relation.

Thus, the axioms of the relator metamodel can be used to define semantic consistency constraints that are syntactically expressible. In general, we are interested in relation types that are *sound* – i.e. where every relation of the type is consistent.

2.3.3. Derived Elements

An RCC relation as defined above can only partially express the way a communication diagram can be related to a class diagram. A richer relation could also show how the links of the communication diagram are instances of associations in the class diagram; however, here we run into a problem – what if the associations these links represent are not explicit, but rather derived? For example in Figure 2.4, the link between :Student and :Catalog is an instance of an implicit association "stuCat" between classes Student and Catalog that is derived from the "uses" association between LibraryUser and Library and the "part of" relation between Library and Catalog. In particular, stuCat is derived as follows:

- Derive stuUses(Student, Library) by specializing uses(LibraryUser, Library) to restrict its (left) domain to Student.

- Derive stuCat(Student, Catalog) by composing stuUses(Student, Library) with partOf(Library, Catalog)

If we had the appropriate algebraic operators available, we might express this as:

stuCat = composeAssoc(leftSubAssoc(Student, uses), partOf)

Note that a derivation such as this does not add any information to the model – stuCat was there all along, only implicitly. Thus, derivation is a way to extend a model, without adding information to it, in order to get to the appropriate form for establishing the relationship between elements.

We can support the expression of derived associations by defining a richer type of relation RCC1 as follows:

RCC1 = RCC +

sorts association1

subsorts association $\leq$ association1

func startClass1: association1 $\rightarrow$ class

endClass1: association1 $\rightarrow$ class

leftSubAssoc: class $\times$ association1 o$\rightarrow$ association1

composeAssoc: association1 $\times$ association1 o$\rightarrow$ association1

instanceOf: object $\rightarrow$ class

instanceOf: link $\rightarrow$ association

instanceOf: message $\rightarrow$ operation

axioms

‘association1 is derived from association’
association1 derivedFrom: association

‘startClass1 is an extension of startClass’
$\forall x$:association. startClass(x) = startClass1(x)
‘endClass1 is an extension of endClass’
$\forall x$:association. endClass(x) = endClass1(x)

‘instanceOf’
‘instanceOf maps startObject to startClass’
$\forall x$:link.instanceOf(startObject(x)) = startClass(instanceOf(x))
‘instanceOf maps endObject to endClass’
$\forall x$:link. instanceOf(endObject(x)) = endClass(instanceOf(x))
‘instanceOf maps messLink to opClass’
$\forall x$:message.opClass(instanceOf(x)) = instanceOf(messLink(x))

‘leftSubAssoc’
‘in leftSubAssoc(c, x) = y, c must be a subclass of the start class of x’
$\forall c$:class,x,y:association1.leftSubAssoc(c, x) = y
$\Rightarrow$ subClassOf(c, startClass1(x))
‘in leftSubAssoc(c, x) = y, c must be the start class of y’
$\forall c$:class,x:association1.startClass1(leftSubAssoc(c, x)) = c
‘in leftSubAssoc(c, x) = y, end class of x must be the end class of y’
$\forall c$:class,x:association1.endClass1(leftSubAssoc(c, x)) = endClass1(x)

‘composeAssoc’
‘in composeAssoc(x, y), the end class of x is the start class of y’
$\forall x$, y, z:association1.composeAssoc(x, y) =$z \Rightarrow$ endClass1(x) = startClass1(y)
‘in composeAssoc(x, y) = z, the start class of z is start class of x’
$\forall x$, y:association1. startClass1(composeAssoc(x, y)) = startClass1(x)
‘in composeAssoc(x, y) = z, the end class of z is end class of y’
$\forall x$, y:association1. endClass1(composeAssoc(x, y)) = endClass1(x)

‘the leftSubAssoc of a composeAssoc(x, y) is the same as first doing leftSubAssoc on x and then composing’
$\forall$ x, y:association1. leftSubAssoc(c, composeAssoc(x, y)) = composeAssoc(leftSubAssoc(c, x), y)

Here we introduce a new sort "association1" and use the "subsorts" directive to express the fact that it extends the sort "association" from CD (within the included RCC). This sort will be used to "glue" derived associations onto the existing CD using the derivation functions leftSubAssoc(..) and composeAssoc(..). Another notational extension is the "derivedFrom" construct (described below) used to enforce the constraint that every element of association1 must have been derived from an association in the CD – i.e. we do not want to allow instances of RCC1 in which there is an element of type association1 that was not derived because this would be adding "new information."

2.3.4. Formalizing "derivedFrom"

Derivation is an important technique for defining model relations. The intuition is that before we can relate the elements of different models, we may need to first change the form of the elements or move them to another level of abstraction *without adding more information* (but possibly taking some away).

The "DerivedFrom" construct used in the metamodel above is syntactic sugar used to express the constraint that in all models, the elements of one sort must all be derived from the elements of other sorts. Specifically, if metamodel M with sorts S, S_1, S_2, …, S_n contains the sentence 'S derivedFrom: S_1, S_2, .., S_n,' this means that for all models $M \in [[M]]$, and elements $s \in [[S]]_M$, s is expressed as a term consisting only of the elements of $[[S_1]]_M$, $[[S_2]]_M$, …,$[[S_n]]_M$ and the functions in M. Thus, every element of $[[S]]_M$ is computed from the elements of $[[S_1]]_M$, $[[S_2]]_M$, …,$[[S_n]]_M$ and hence is not net "new."

The use of derivedFrom can be converted into FO+ as follows:

1. Add a "universal" sort U and make every other sort a subsort of this.
2. Add the binary relation der(U, U).
3. For each function or partial function $h:T_1 \times \ldots \times T_m \rightarrow T$, add the axiom:

$$\forall x{:}T.\ (\exists t_1{:}T_1, \ldots\ t_m{:}T_m.\ h(t_1,\ldots,t_m) = x\ ) \Rightarrow (\mathrm{der}(x, t_1) \wedge \ldots \wedge \mathrm{der}(x, t_m))$$

4. Replace each occurrence of 'S derivedFrom: S_1, S_2, .., S_n' with the axiom:

$$\forall x{:}S, z{:}U.\ \mathrm{TC}(\mathrm{der}(x, z)) \Rightarrow (\exists y{:}S_1.y = z) \vee \ldots \vee (\exists y{:}S_n.y = z)$$

This says that every element of type S at the root of a derivation tree must have its leaves in elements of types S_1, …, S_n.

2.3.5. Structure of a Relation Metamodel

We have characterized a relation type in terms of a metamodel – and in particular, the metamodel of the relator; however, it is not clear that all FO+ metamodels should be *allowed* as relator metamodels. In some sense, we expect that a model relation should only contain enough information to express the way its component models are related and no more. Adding more information than this would seem to suggest that the information should be expressed in another

model rather than in a model relation. For example, there is nothing restricting us from adding another sort called "package" with associated functions to RCC as:

RCC2 = RCC +

sorts package

func packageOf: object → package

packageOf: link → package

packageOf: class → package

packageOf: association → package

However, this seems to be introducing information that is not directly relevant to the way a CD and CommD are related. It would seem more appropriate to create a separate type of model (e.g. package diagram) that expresses the packages and then have model relations between CD or CommD and this model.

Some "heuristic" criteria for possible reasonable extensions to RCC might include the following:

- Functions that derive new elements from existing elements in the CD or CommD.
- Sorts that are completely derived from elements in the CD or CommD.
- Predicates that have at least one argument being a CD sort and at least one argument being a CommD sort.

Some extensions that would seem unreasonable include:

- Sorts whose elements cannot be derived from either CD or CommD (e.g. the case of package above).
- Predicates that have all their arguments in either CD or CommD.

We leave the deeper analysis and formalization of these criteria to future work.

2.3.6. Micro and Macro Semantics

Since model relations are classified by relation types and relation types have particular semantics, knowing the relation type carries useful information even if the details of the relation are not visible. We will refer to this information as the macro semantics of a relation while the micro semantics concerns the meaning of the internal details of the relation. The value of this macro semantics is the key theme of this paper. Consider the following examples of model relation:

- R:overlapStatechart(A:Statechart, B:Statechart)
  - macro semantics: A shares some states/transitions with B
  - micro semantics: R contains the mapping of common elements
- R:detailOf (A:Statechart,B:Statechart)

    - macro semantics: A shows the substates/transitions of a state in B
    - micro semantics: R identifies the state of B
- R:substateRefinement (A:Statechart, B:Statechart)
    - macro semantics: A has more fine-grained states than B but the behaviour expressed by B is also expressed by A
    - micro semantics: R maps the states of A to their superstate in B
- R:refactorX (A:Statechart, B:Statechart)
    - macro semantics: B is a type X structural modification of A but expresses the same behaviour
    - micro semantics: R maps the before/after modifications

In each of these cases, the macro semantics conveys some useful information about syntactic and semantic relationships[5] between statecharts A and B. When applied to the more general case of a multimodel with many different types of relationships, the resulting macromodel turns out to be a useful level of abstraction for understanding and developing the multimodel.

2.3.7. Classes of Model Relation Types

The relation types that commonly arise in software engineering can themselves be organized into different classes. In fact, the a key part of the macro semantics that come from a relation type is due to the class it is in. We list some of these classes of relation types in Table S1. Here, each class is parameterized by given model types and it is characterized by the way its relation types relate the models of the given types .

Relation Type Class	Macro semantics
Detail(M, M)	The second M-model is a refinement of an element in the first M-Model
Homomorphism(M, M)	There is a structure preserving mapping from the first M-model to the second M-model.
Submodel(M, M)	The first M-model is part of the second M-model
Aspect(M, M1)	The M-model consists of a set of fragments of the M1-model.
Transformation(M) → M1	Each M-model is transformed into an M1–model
Projection(M) → M1	The M1-model is constructed by discarding some sorts, functions and predicates of the M-model.
Translation(M) → M1	The M1-model is semantically equivalent to the M-model but syntactically different.
Abstracting Transformation(M) → M1	The transformation is many-one but the meaning of the M1-model follows from the meaning of the M-model.

[5] Note that the macro semantics is the meaning conveyed by the relation type and this can include information about both the syntax and the semantics of an instance of that type.

Aggregating abstraction(M, M)	Each element of the first M-model is refined into a set of elements in the second M-model.
Refactoring(M, M)	The second M-model shares semantic properties with the first M-model
DerivedExtension(M, M)	The first M-model is a submodel of the second M-Model but all additional elements of the second M-model are derived from elements in the first M-model.

2.3.8. Constructed Relation Types

The fact that these classes of relation types are parameterized by model type suggests that a generic approach could be taken, in some cases, to producing the relation type of a given class for a given model type – constructed relation types. We discuss three of these briefly here as they are used in the examples of Section 3. Given any model type T, we can define *submodelOf*[T] where for any T-models M_1:T and M_2:T, *submodelOf*[T](M_1, M_2) holds iff each element of M_1 is also an element of M_2 and each relation between elements that holds in M_1 also holds in M_2. For example, if we had sequence diagrams M_1:*SequenceDiagram* and M_2:*SequenceDiagram* with metamodel as in Figure 2, then when *submodelOf*[*SequenceDiagram*](M_1, M_2) holds this means that every *Object* of M_1 is an *Object* of M_2, every *Message* of M_1 is a *Message* of M_2 and if *nextMessage*(m1, m2) holds for messages in M_1 then this is also true in M_2. Appendix B elaborates the submodelOf relation type further.

Each submodel relation *submodelOf* [T] has useful property that it is a partial order over T-models. This also implies that given any multimodel K consisting of T-models, we can define *merge*[T](K) as the least upper bound of K – i.e. the least model of type T such that has every model in K is a submodel. Note that this may not always exist. For example in Figure 5, there are two ClassDiagrams with the same classes but they cannot be merged into a single class diagram because this would violate the well-formedness condition for ClassDiagrams that a class cannot be a direct or indirect subclass of itself. Thus, *merge*[T] is a partial function from T-multimodels to T-models.

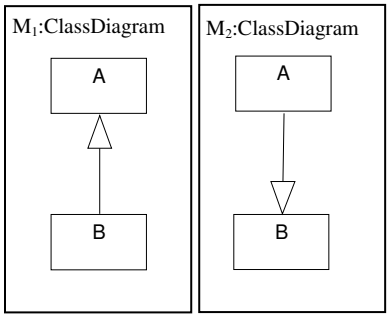


Figure 2.6: Un-mergeable class diagrams

Finally, we define *eq*[T] as follows: *eq*[T](M_1, M_2) holds iff there is an isomorphism between M_1 and M_2 where we interpret corresponding elements as being semantically equal – i.e. they denote the same semantic entities. This is a special case of the homomorphism relationship defined in Appendix B.

We leave the further investigation of constructed relation types to future work.

2.4. Multimodels and Macromodels

Now that we have an approach for defining relationship types we can characterize a *multimodel type* in terms of the model types and relationship types that it contains and the type of macromodel that can be used to define its structure. A macromodel is a hierarchical model whose elements are used to denote models and model relationships and can express integrity constraints on these. A multimodel is a macromodel and collection of models/relationships that conform to their types and in addition, conform to the constraints in their macromodel.

Figure 2.7 depicts a simple example of this. The upper part is a multimodel metamodel

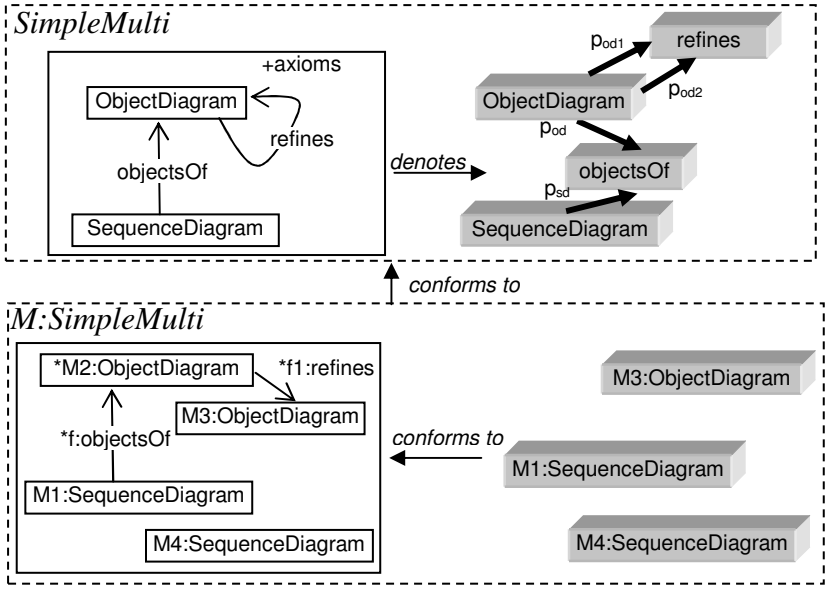


Figure 2.7: A multimodel metamodel and an instance of it

SimpleMulti containing a metamodel for instance macromodels (right side) and a set of metamodels for the model and relationship types (left side). The axioms of the macromodel limit the possible well-formed collections of models and relationships that can occur in a multimodel of this type. Note that the notation for macromodels expresses binary model relationship types as directed arrows between model types although they should be understood as consisting of a relator model and two metamodel morphisms. The simple illustration in Figure 2.7 does not depict hierarchy but Figure 1.1 shows a more complex one that we used in the example in Section 3.

The lower part of Figure 2.7 depicts a particular multimodel *M*:*SimpleMulti* that conforms to the metamodel. This consists of a macromodel and the set of models and relationships 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 multimodel and they are just placeholders. A model or relationship may be unrealized for two main reasons. Firstly, in the context of a development project they may represent models and relationships that are required to be created. Despite the fact that they do not (yet) exist, they can still participate in defining integrity constraints. Secondly, the model or relationship may be derived from others and hence it is "transient." The macromodel in Figure 2.7 expresses 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 FO+ sentences we have:

SequenceDiagram(M4),
SequenceDiagram(M1),
ObjectDiagram(M3),

$\exists f, f1, m2.\ \text{objectsOf}(f) \wedge p_{sd}(f) = \text{M1} \wedge p_{od}(f) = m2$

$\wedge\ \text{refines}(f1) \wedge\ p_{od1}(f1) = m2 \wedge p_{od2}(f1) = \text{M3}$

$\wedge\ \text{ObjectDiagram}(m2)$

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.

2.4.1. Macromodel Syntax

Figure 2.8 shows the abstract syntax of the macromodel language. The following well-formedness constraints hold over macromodels:

(R1) There is a unique root *Macromodel*

$\forall m, m1$:Macromodel.

$(\neg\exists m2$:Macromodel.contains($m2$, m)) $\wedge$ ($\neg\exists m3$:Macromodel.contains($m3$, $m1$)) $\Rightarrow m = m1$

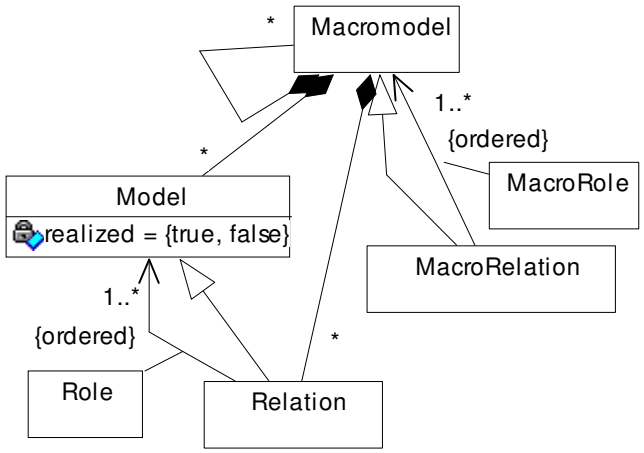


Figure 2.8 – Metamodel for macromodels

(R2) If any of its arguments are unrealized then a *Relation* is unrealized

$\forall r$:Relation$\exists ri$:Role.

ri.Relation = $r \wedge ri$.Model.realized = **false** $\Rightarrow$ r.realized = **false**.

The notation of macromodels is summarized as follows:

- A *Model* element is represented as a box containing the model name. 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 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. 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 and with optional *Macrorole* labels. It optionally can show its contents as a dashed oval linked to the main the arrow or diamond symbol.

Note that there are no means to directly refer to the internal structure of a *Model* or *Relation* element. Macromodels are intended to stay at the level of abstraction of models and model relations and so the internal structure of any simple model is considered opaque.

2.4.2. Macromodel Semantics

In this section we define the formal semantics of macromodels. Our aim is to formally define the set of multimodels that are instances of a macromodel. A macromodel is interpreted by mapping it into a semantic domain.

Assume that we have a macromodel K which has a metamodel T. To define the formal semantics of macromodels we will proceed by first translating T to an FO+ signature Σ_T reflecting the different model and relationship types in it. We then construct a particular "universal" interpretation J_T of this that consists of all possible models and relationships using these types. Then, K is translated to a FO+ theory $\langle \Sigma_T \cup \Sigma_K, \Phi_K \rangle$ where Σ_K consists of a set of constants corresponding to the realized models and relationships in K and Φ_K is a set of axioms as illustrated in the example above. We characterize a multimodel of K as any interpretation J_M that extends J_T to $\Sigma_T \cup \Sigma_K$. That is, each assignment of the constants in Σ_K to elements of the appropriate types in J_T is a multimodel.

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 $\mathrm{Mod}(\Sigma_X)$
- To each predicate symbol X: S_X assign the unary relation defined by the set $\mathrm{Mod}(\Sigma_X, \Phi_X)$
- To each function symbol p:$S_Y \rightarrow S_X$ assign the function $\mathcal{M}$p: $\mathrm{Mod}(\Sigma_Y) \rightarrow \mathrm{Mod}(\Sigma_X)$ induced by the signature morphism p_Σ:$\Sigma_X \rightarrow \Sigma_Y$

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.

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.

Based on these algorithms, the full translation of the example in Figure 2.6 is as follows:

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($m2$)

3. Example Scenarios Revisited

In this section, we reconsider the example scenarios described in the “Motivating Problems” section of the introduction and apply the formalism discussed above to addressing these.

3.1. Scenario 1: Exposing Multimodel Structure

The discussion in this scenario in the introduction suggested that a macromodel of a multimodel is a potentially useful view in software development. In order to do a preliminary assessment of the validity of this hypothesis, we have taken an existing set of UML diagrams that comprise a UML design model, a hypothetical multimodel metamodel for UML and constructed the corresponding multimodel and macromodel based on this schema. The informally observed

result is that the macromodel provides significant support in helping to understand the content of the UML model and the way it is decomposed into diagrams.

The UML diagrams are taken from [10] which is a document of the European Telecommunications Standards Institute (ETSI) and describes a methodology for standards development based on UML. As an example to illustrate the methodology, the document details the development of a standard for a 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]

The example consists of three models: a context model (4 diagrams), a requirements model (6 diagrams) and a specification model (32 diagrams).

Figure 3.1 shows the multimodel metamodel for "UMLMulti" that was used. The relation types are described informally as follows:

Relation Type	Macro Semantics
detailOf(ClassDiagram, ClassDiagram)	detailOf(A:ClassDiagram, B:ClassDiagram) holds iff the following conditions are true: o A and B share exactly one class. Call this class *X*. o Every association in B is either an aggregate of *X*, a composite of *X* a generalization or a realization.
overlap(ClassDiagram, ClassDiagram)	overlap(A:ClassDiagram, B:ClassDiagram) holds iff A and B share at least one class.
detailOf(Statechart, ClassDiagram)	detailOf(A: Statechart, B:ClassDiagram) holds iff A specifies a state machine for some class in B.
detailOf(StateDiagram, StateDiagram)	detailOf(A: Statechart, B: Statechart) holds iff A specifies a substatechart for some state in B.
instanceOf(ObjectDiagram, ClassDiagram)	instanceOf(A:ObjectDiagram, B:ClassDiagram) holds iff every object and link in A is an instance of a class and association, respectively, in B.
objectsOf(SequenceDiagram, ObjectDiagram)	objectsOf(A:SequenceDiagram, B:ObjectDiagram) holds iff the following conditions are true: o Every object in A is found as an object in B o If a message is passed from object *X* to object *Y* in A then there is a link in B between *X* and *Y*

refines(UMLDiagrams, UMLDiagrams)	refines(A, B) holds iff models in A have refinement relations to models in B. These may be any of the following: o refines(ObjectDiagram, ObjectDiagram) o eq(Model, Model) o caseOf(SequenceDiagram, SequenceDiagram)
diagramsOf(UMLDiagrams, UML)	diagramsOf(A, B) holds iff each model M in A has a diagramOf(M, B) relation and each relation f in A as a SubmodelOf(f, B) relation.
refines(ObjectDiagram, ObjectDiagram)	refines(A, B) holds iff o Every object in B is refined into one or more objects in A and every object in A is in exactly one such a refinement. o Every link in B is refined into one or more links in A and every link in A is in exactly one such a refinement. o The link refinements must be consistent with the object refinements of their endpoints.
caseOf(SequenceDiagram, SequenceDiagram)	caseOf(A, B) holds iff the following conditions are true: o Every object in B is refined into one or more objects in A and every object in A is in exactly one such a refinement. o Every message in B is specialized and then refined into one or more messages in A and every message in A is in exactly one such refinement. o The message refinements must be consistent with the object refinements of their endpoints.

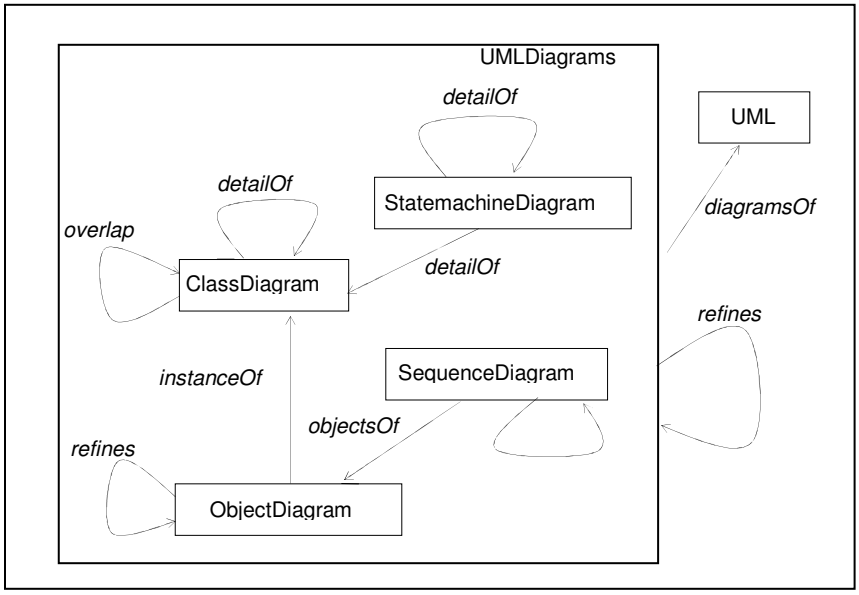


Figure 3.1: Multimodel metamodel for UMLMulti

Appendix A shows an example relation for each relation type listed. Figures 3.2 and 3.3 show two portions of the resulting macromodel.

3.1.1. Observations

Models vs. Diagrams

An interesting issue we encountered in the context of UML is the question of how to treat diagrams vs. models. In the modeling community a model is typically distinguished as the artifact of "core value" while the diagrams of the model are secondary as views of it. Despite this it is clear that in actual development projects, the majority of human interaction with the model is through diagrams and the choices made about how this information is distributed across the various diagrams is itself an important part of the information carried by the model. In order to address this, we decided to 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.

The diagram in Figure 3.2 illustrates this. It shows part of the PUMR macromodel containing two sub-macromodels representing the diagrams of the context model and a subset of the diagrams of the specification model. Relationships are shown both among the diagrams within each UML model and also between the diagrams across the models. Note that the relationships between diagrams are marked as unrealized because they are derivable from the underlying UML model and hence are not required to be maintained separately.

Complexity Management

It is clear that these macromodels provide significantly more structural information than the list of diagrams alone could provide. In particular, from the perspective of our personal experience, as we tried to understand the PUMR example, we found that we had a great difficulty in "grasping" the model as whole. Although the model was decomposed into many diagrams (or perhaps, because the model was decomposed into many diagrams) it was still difficult to understand the big picture. In fact, we did not have a good understanding *until* we created the macromodel and the underlying structure of the decomposition emerged.

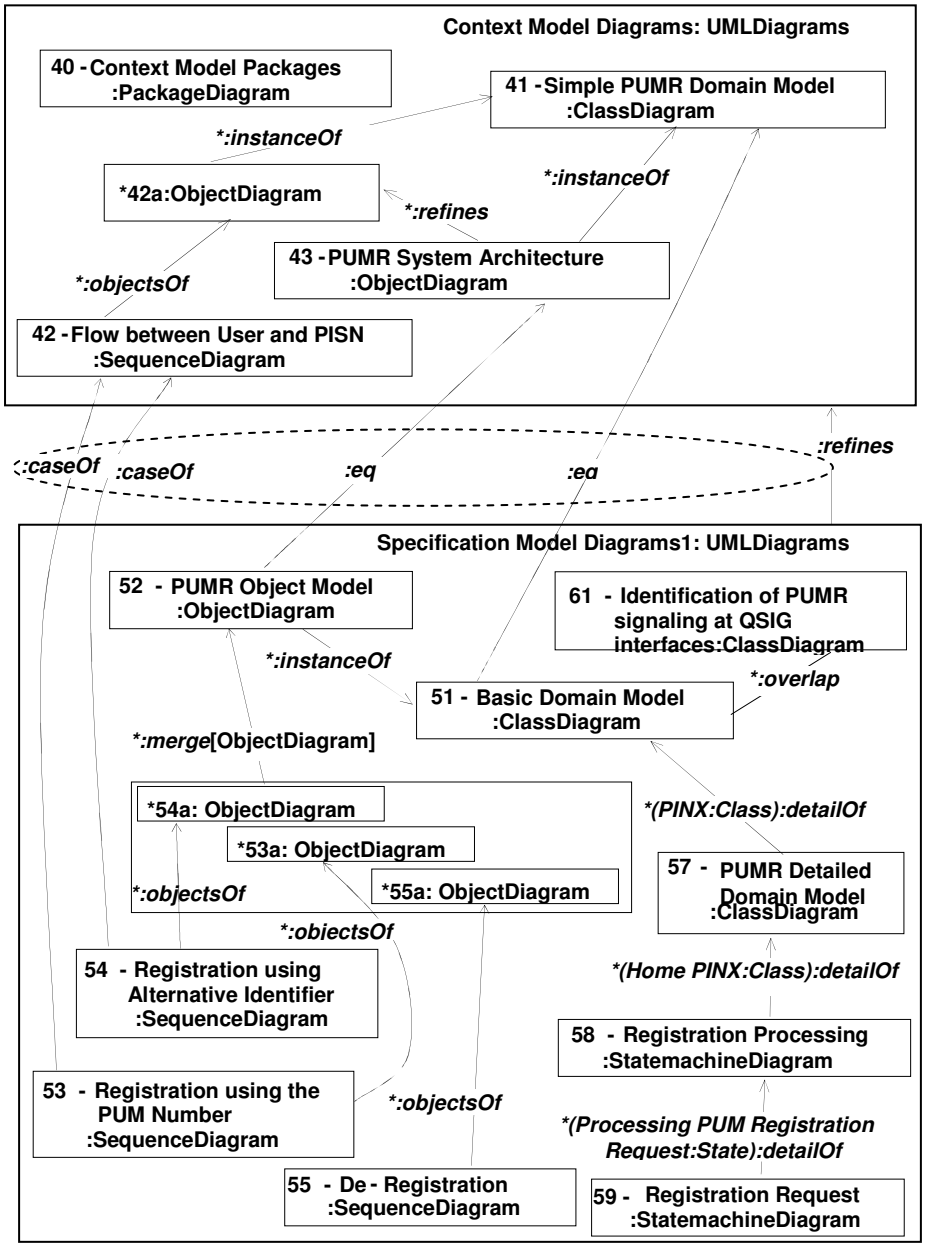


Figure 3.2: PUMR Macromodel 1

Thus, it would seem, that although the decomposition of a model into multiple models (i.e. a multimodel) provides a way to manage the complexity of a large model, it can create its own secondary complexity problem when the number of models itself becomes large. Managing this secondary complexity problem requires more than just a mechanism for grouping sets of models, such as the UML "package" construct. This is because, an important aspect of using multimodels to manage the primary complexity problem is way that multiple models decompose information this decompositional structure is only "visible" if the relations between the models are made explicit. Thus, macromodels play a key complexity management function in multimodeling.

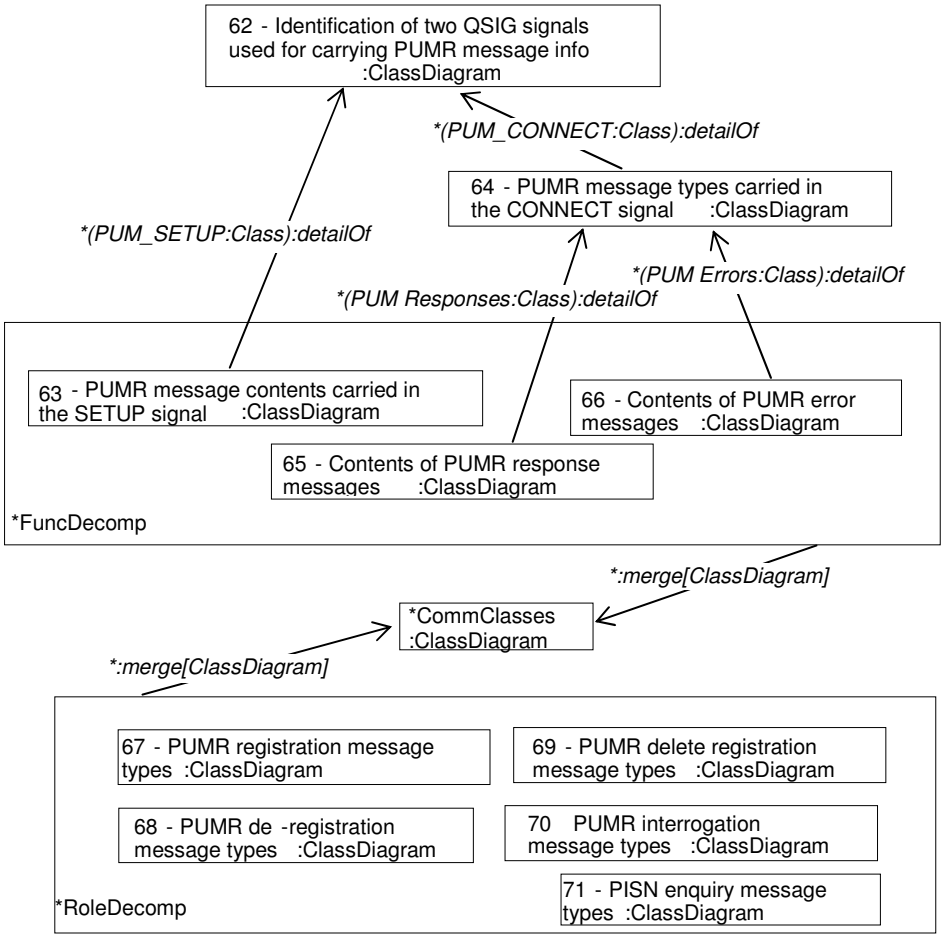


Figure 3.3: PUMR Macromodel 2

Relation Name Construction

Note that in the PUMR macromodels, for instances of the three "detailOf" relation types, we have used the relation name to indicate the name of the element that is being detailed in order to provide more information at the macromodel level. While this is only a convention and hence not formally enforced, it is interesting to consider the possibility that the naming scheme for a relation (or model) could be defined formally so that an informative name could be automatically constructed (or partially constructed as a prefix). For example, in the case of the detailOf relation types, information about the element that is being detailed can be extracted automatically using a query in terms of the relation metamodel that can be executed on any instance. Such an approach would be particularly valuable when multiple instances of relation type could exist, as with "detailOf." We do not explore this idea further here and leave it for future work.

Expressing Complex Relationships

We found several interesting situations where we needed to express complex relationships using unrealized models. In Figure 3.2, 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.

The lower part of the macromodel in Figure 3.3 shows a case where two collections of diagrams are related to one another in a way that suggests an "alternate decomposition" of the same set of classes. The set of class diagrams FuncDecomp = {63, 65, 66} and RoleDecomp ={67, 68, 69,70 ,71}each decomposes the same set of "communication message" classes in alternate ways. FuncDecomp groups classes according to whether they are "setup," "response" or "error" messages while RoleDecomp groups classes according to whether they are "registration", "de-registration", "delete registration", "interrogation" or "enquiry" messages. Note that both of these sub-macromodels are unrealized – i.e. they are introduced only for the purposes of expressing the alternate decomposition. The collections are related by the fact that they merge to the same class diagram containing the communication message classes.

3.2. Scenario 2: Structuring the Development Process

In scenario 1, a macromodel is used to express a known set of models and relations between them. Another possibility is to use a macromodel as template in which some of the models and

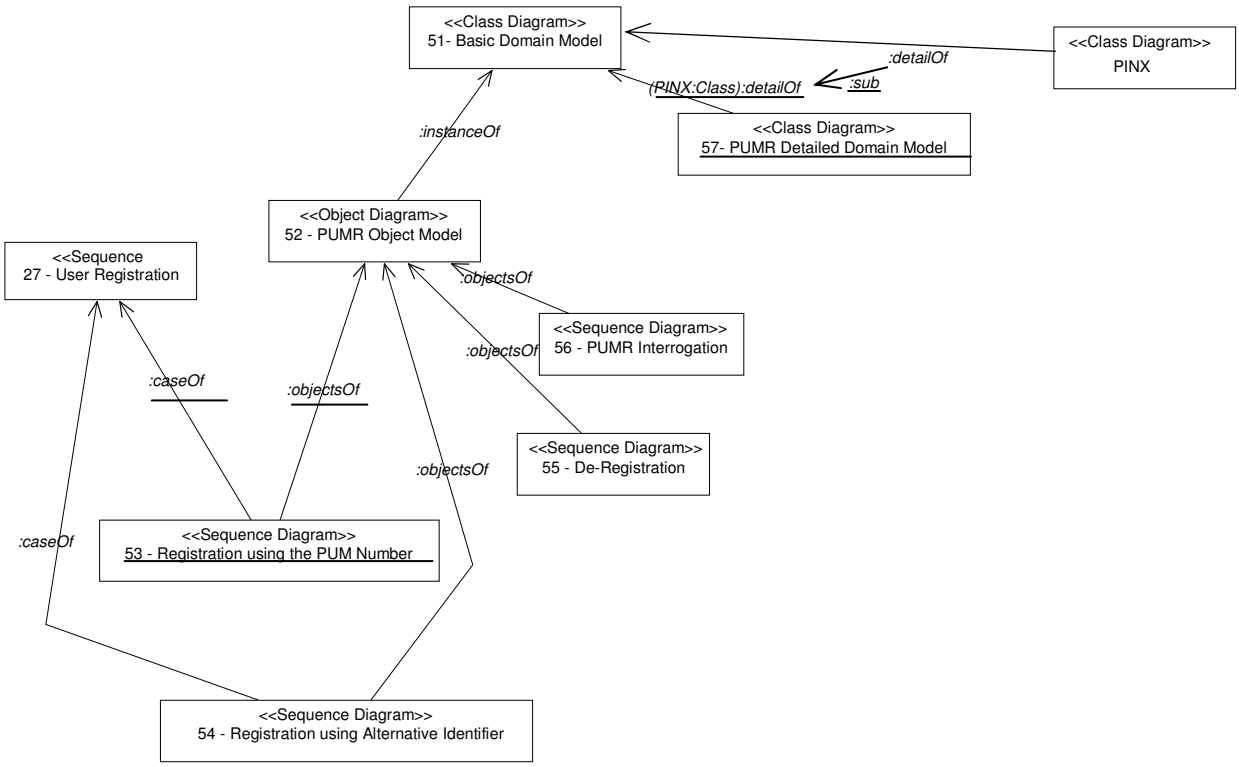


Figure 3.5: PUMR Macromodel Template

relations are just placeholders and may be unknown at some stage in the development process. Thus, we are using a macromodel to assert what models must be developed and how they must relate to existing models and other unknown models. For example, the macromodel in Figure 3.3 expresses an earlier development stage to the one in Figure 3.1, in which diagrams 56 and 57 are as yet not created. The underbars on the names indicate that they are placeholders for models and relations of particular types rather than actual models and relations. Note that when a model is represented by a placeholder then so must all relations connected to it since these cannot be known if any of the models they relate are not known. Thus, in this case, diagram 53 must be a sequence diagram which refines sequence diagram 27 and has its objects within object diagram 52. Note that this macromodel template not only requires that diagram 53 be created but also the details of the caseOf and objectOf relations and these must, of course, all satisfy the axioms of their respective metamodels.

In the case of diagram 57, we desire that it be a detail of the particular class PINX within the basic domain model (diagram 51). We cannot use the fact that the placeholder is named "(PINX:Class)" to enforce this because the naming here is an informal convention. One possibility is to partially construct diagram 57 by including the single class PINX within it; however, this "hides" part of the constraint expressed by template within a particular model. Another, more explicit, possibility, is to construct a separate model containing this information and assert that it must be a submodel (i.e. via :sub) to the placeholder. Thus, in the figure, we construct a new class diagram with a single class PINX within it and a detailOf relation that maps this class to the corresponding class PINX within diagram 51[6]. The :sub relation is asserted to hold between this detailOf relation and the placeholder (PINX:Class):detailOf – i.e. the former must be a sub-relation of the latter.

3.3. Scenario 3: Model Manipulation Operators

In this section we work through the details of the AutoRefine transformation operator proposed in the introduction. We will first define the refinement relation type RefCommD(CommD, CommD) and SimpleRefCommD(CommD, CommD) corresponding to the types of the refinement relations Ref2 and Ref1 in Figures 2.3 and 2.2, respectively. Next we define the transformation relation type AutoRefine(CommD, SimpleRefCommD, CommD).

In order to understand the refinement RefCommD first observe the basic case illustrated in Figure 3.5. In M1, upon invocation of the call mess 1[7], a series of calls mess 2 → mess *n*-1 are made on various objects before mess 1 returns and mess *n* is called. M2 shows an abstracted version of M1 in which we have rolled up or "absorbed" mess 2 to mess *n*-1 into a new message mess A that hides all the detail. Thus RefCommD is an application of one or more such message absorptions. In addition, we can decompose an absorption into a series of pairwise absorptions. This allows us to derive:

[6] One might think that because they have the same name, this mapping should be "automatic," but in general we don't want to make this assumption – i.e. we might have the same class in different models but having different names. If we want to make the assumption that naming is a global identifier then this can be built into the axioms of the relation type.

[7] For this example we will assume that messages are synchronous calls on an object.

mess A = absorb(mess 1, absorb(mess 2, … (absorb(mess n-2, mess n-1))…))

We can now use this to define the metamodel for RefCommD as follows:

RefCommD = CommD.1 + CommD.2 +

sorts messageDer

func absorb:messageDer × messageDer o→ messageDer

messLinkDer: messageDer → link.2

nextDer:messageDer → messageDer

id:message.1 → messageDer

id:link.1 → link.2

id:object.1 → object.2

axioms

'messageDer is derived from message.2'

messageDer derivedFrom: message.2

'messageDer is an extension of message.2'

messageDer ≤ message.2

$\forall x$: message.2. messLinkDer(x) = messLink.2(x)

$\forall x$: message.2. nextDer(x) = next.2(x)

'absorb'

'the receiver of the first message must be the sender of the second message

$\forall x, y, z$:messageDer. z = absorb(x, y) ⇒ startObject(messLinkDer(y)) = endObject(messLink(x))

'the messages must be contiguous'

$\forall x, y, z$:messageDer. z = absorb(x, y) ⇒ y = nextDer(x)

'the result message has the same sender and receiver as the first message'

$\forall x, y$:messageDer. startObject(messLinkDer(absorb(x, y))) = startObject(messLinkDer(x))

$\forall x, y$:messageDer. endObject(messLinkDer(absorb(x, y))) = endObject(messLinkDer(x))

'the successor of the result message is the successor of the second message'

$\forall x, y$:messageDer.nextDer(absorb(x, y)) = nextDer(y)

'the sender of the successor of the result must be the receiver of the result'

$\forall x, y$:messageDer. startObject(messLinkDer(nextDer(absorb(x, y)))) = endObject(messLinkDer(absorb(x, y))

'id'

$\forall x$:message.1. id(messLink.1(x)) = messLinkDer(id(x))

$\forall x$:message.1. id(next.1(x)) = nextDer(id(x))

$\forall x$:link.1. id(startObject(x)) = startObject(id(x))
$\forall x$:link.1. id(endObject(x)) = endObject(id(x))

This metamodel is similar to the example of RCC1 described in section 2.3.3. There are two component models of type CommD and an intermediate sort messageDer that is derived from the messages of the refined communication diagram CommD.2 using the absorb(..) derivation function. This allows new, more abstract, messages to of type messageDer to be added to CommD.2 until there is a 1-1 correspondence with messages in CommD.1 and then this mapping can be established using the id relation. Figure 3.6 shows the refinement for the example in Figure 1.3 partially illustrated on top of the concrete syntax. Both communication diagrams are seen here as part of the refinement Ref2:RefCommD. In addition, messages 1 to 7 in M4:CommD.2 are absorbed into a new message – for readability, only the final derived message is shown (with dashed lines) but the others are present as well. The successor of this message using nextDer(..) is the message "Get the book." Finally, all the elements of M3:CommD.1 are mapped to their corresponding elements via id(..) - for readability, only the mappings of messages are shown but objects and links are mapped via id(..) as well.

SimpleRefCommD is just a version of RefCommD that is constrained so that the first (unrefined) communication diagram contains a single message. This is defined as follows:

SimpleRefCommD = RefCommD +
axioms
$\forall x,y$: message.1. $x = y$

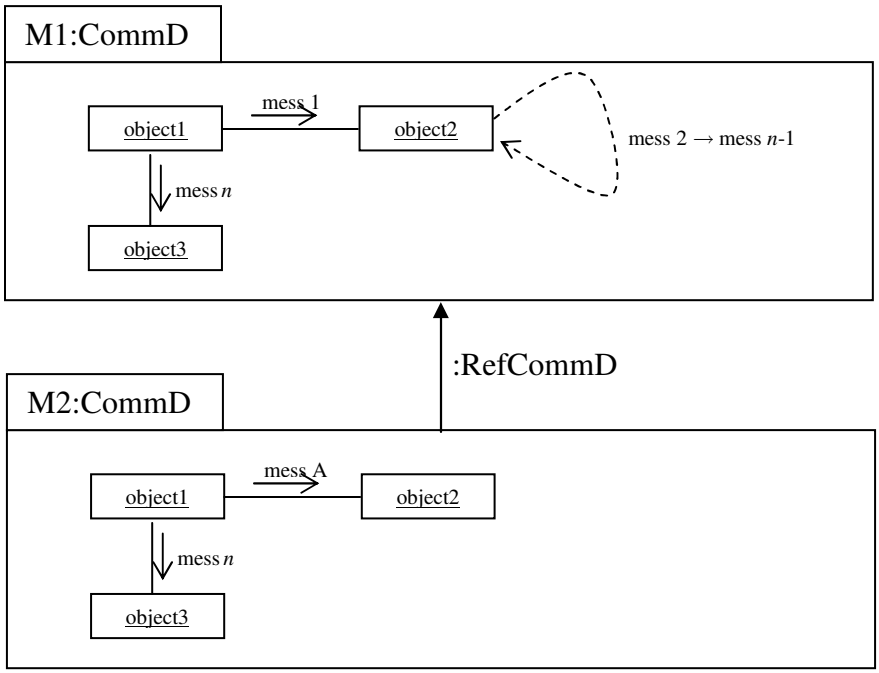


Figure 3.5: Absorbing messages

3.3.1. AutoRefine

With RefCommD and SimpleRefCommD defined we turn to defining AutoRefine(CommD, SimpleRefCommD, CommD).

In order to compute AutoRefine(X:CommD, A:SimpleRefCommD, Z:CommD) we essentially need to construct a B:RefCommD such that the following conditions hold:

1. X:CommD = B.CommD.1
2. Z:CommD = B.CommD.2
3. Each message in B.CommD.1 that matches A.CommD.1 refines to A.CommD.2 in B.CommD.2
4. Each message in B.CommD.1 that does not match A.CommD.1 refines to itself in B.CommD.2 – i.e. it is the identity refinement.

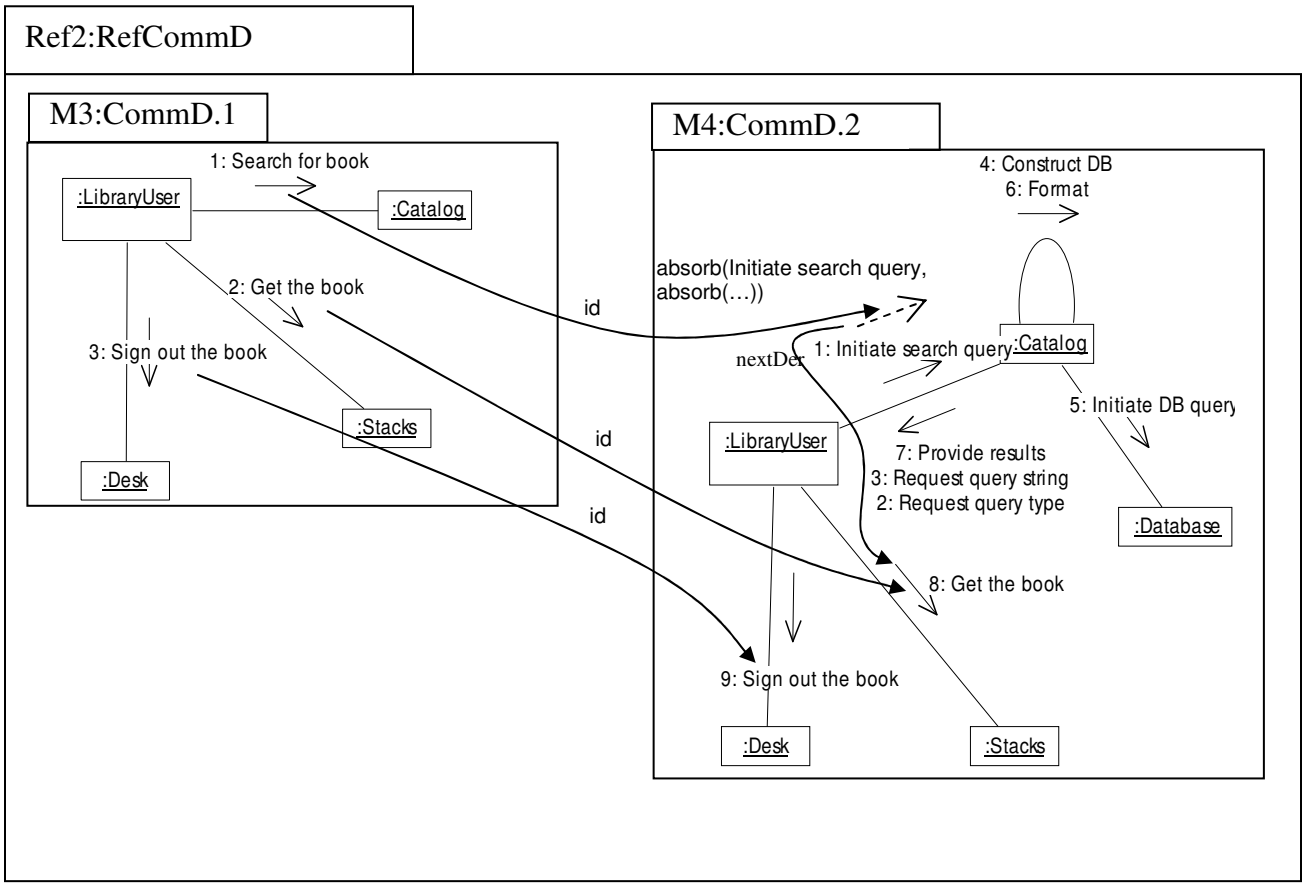


Figure 3.6: A derived refinement

By now it should be clear that encoding this as an FO++ metamodel is just an exercise in logic programming. However, it is interesting to consider whether it would be possible to define AutoRefine by staying at the same level of abstraction as the four steps expressed above without delving into the detailed specification required in an FO++ metamodel. Although, achieving this is beyond the scope of this paper, we sketch how it could be accomplished. First we define the multimodel metamodel shown in Figure 3.7. Metamodels CommD, RefCommD and SimpleRefCommD are defined as above. IdRefCommD is the trivial "identity refinement" version of SimpleRefCommD and is defined by adding constraints to SimpleRefCommD. SubRef is the submodel relation for RefCommD. Given these, we could write a definition of AutoRefine in a hypothetical logical language over the multimodel metamodel as follows:

AutoRefine(x:CommD, a:SimpleRefCommD, z:CommD) :=

Min(b:RefCommD, SubRef).
$x = \mathrm{R1}(b) \wedge$
$z = \mathrm{R2}(b) \wedge$
$\forall y$:RefCommD.:SubRef(y, a) $\wedge$:SimpleRef(y)
$\Rightarrow y = \mathrm{RS}(a) \vee$:IdRef(a)

This says that z:CommD is the refined side of the smallest RefCommD refinement of x:CommD such that every simple sub refinement of it is either a:SimpleRefCommD or the identity refinement. The use of minimization is necessary here in order to select one from an infinite set of candidates for b:RefCommD that satisfies the criterion. Minimization is relative to the submodel relation SubRef. The details of a logical language for models such as this are left to future work.

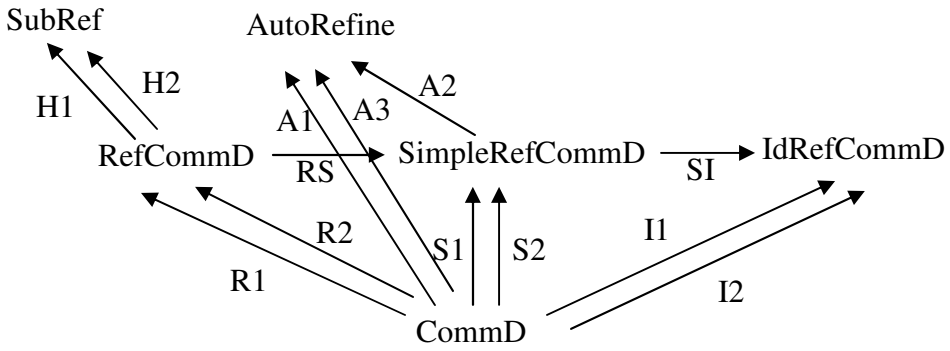


Figure 3.7: Multimodel metamodel for AutoRefine

4. Related Work

OMG has defined a standard called QVT (Query/Views/Transformations) [12] for defining model transformations in terms of their MOF metamodels using a relational approach[8]. It is intended as a key specification required for the Model Driven Architecture (MDA) initiative [11]. The QVT declarative language called the "Relations Language," allows a transformation between different model types to be defined as a set of relation specifications that must hold between the elements of the models of those types. A relation specification consists of a collection of structural patterns that bind to subsets of elements in each model and a set of constraint expressions that these bound elements must satisfy. These can form a hierarchical structure by referencing other relations from within the constraint expressions. A transformation can either be used either passively to check a set of models to determine whether the relations hold, or actively to enforce the relations that do not hold by adding and/or deleting structure to/from the target model. Model transformation is implemented by applying relations in enforcement mode with an empty target. In addition, when a transformation is applied, to a set of models, the result is a set of relation binding instances. This provides a trace of the relation application.

There are some strong similarities of QVT to what we are proposing. A QVT transformation definition plays the same role as what we call a relation metamodel. The relation specifications within a transformation correspond to the functions, predicates and axioms defined within a relation metamodel. The trace instance of a transformation application is what we call a model relation (i.e. an instance of a relation type). However, there are some important differences from our approach as well. While QVT is focussed on transformations, we consider this to be just one class of relation type – i.e. a relation type in which one argument can be completely determined from the other arguments. Other classes are required as well in order to express the relations between models more generally since not all models stand in a transformation relation to each other. A more important difference is that we are taking a uniform and algebraically grounded approach to defining relation types in terms of model types by using metamodel morphisms. This allows us to define higher order relation types such as the AutoRefine transformation described in scenario 3. It is not clear how one would correspondingly extend QVT to define transformations of transformations. Additionally, since our work is not intrinsically contingent on the metamodeling language FO+, we could apply it using MOF/OCL and hence get the benefits of our approach with MOF.

The emerging field of Model Management [2] has close ties to our work. This has developed from the problems of dealing with multiple models in meta data management such as schema integration. A key part of the approach here is to express the relationship between two models by defining a "mapping" between them and then models and mappings can be treated as basic units that can be manipulated at the "macro" level by using a set of generic model management operators. For example, performing the operation map12 = Match(M_1, M_2) can find a mapping between models M_1 and M_2 while another operation M_3 = Diff(M_1, map12) will construct a

[8] It also supports procedurally defined transformations but this is not relevant to our purposes here.

new model consisting of the parts of M_1 not referenced by map12. This basic idea has been elaborated in different ways. For example, Diskin [5] defines the semantics of model management operators in terms of specific category theoretic constructions. In contrast, the group [3] leaves the operator implementations open but defines the algebraic properties that must hold between them.

In general, we view model management operators as a particular subclass of a transformation relation type; however, we acknowledge that generic operators may not always be sufficient. Some types of transformations, such as AutoRefine, are specialized to particular types of models/relations and hence cannot be expressed in a completely generic way. On the other hand, some types of relations, such as the submodel merge operator can be expressed generically. Although, we don't elaborate the formalism for defining such generic operators in this paper, there is a natural extension of our work to do so and we discuss this in section 5.

The work on algebraic specification has several connections to our work. Firstly, we consider a metamodel to be an algebraic specification and each of its models, in the model theoretic sense, is what we consider a model (instance of the metamodel) in the modeling sense. This is related to, but different from, the MOMENT model management framework [4] where model types are defined using algebraic specification as abstract data types that define the algebraic operations required to construct models from their components. Secondly, the ideas of institution theory [7] are used in showing how a metamodel morphism induces sentence translation and model projection (i.e. reduct functions); however, the version of this that we need does not require category theory and hence takes a simplified form. Finally, work has also been done on the notion of a "structured specification"[16] consisting of specifications connected by links that show their relations and this is reminiscent of our notion of a multimodel; however, our purposes are quite different from this work. In algebraic specification, a structured specification a decomposition of a larger specification into smaller ones and the focus is modularity and reuse. In our case, the individual models in a multimodel are primary and we are interested in understanding how they relate. Thus, we admit relations such as refinement and refactoring which are not considered as structuring relations in a structured specification.

5. Future Work

In this section we recap the topics that have been identified as candidates for future work. In Section 2.3.7 we listed a set of classes of relation types. Each of these classes can be characterized in terms of syntactic and semantic properties that all relation types in the class share. Because they are so pervasive in software engineering, it seems worthwhile to both enumerate such a catalog and to formally define their characterizing properties. In addition, there is a close link between these and the generic operators proposed in Model Management. Both have a degree of independence from the particulars of any specific model type – or rather, they can parameterized by the metamodel for a model type to produce a version of a relation type "customized" for the model type. The examples of the submodel and homomorphism relations in Appendix B illustrate this. A natural extension of our approach that would provide a way of dealing with these generic situations is to consider defining *metamodel* relation types whose instances are relations between metamodels - these would need to be defined at the metametamodel level. For example, this would allow us to define an operator that takes an arbitrary metamodel for some model type M and transform it into a metamodel for the submodel relation for models of type M, a merge operator for the models of type M, etc. Thus, generic model management operators may be able to be defined as metamodel relation types.

Another area that requires further investigation is the criteria for what should be allowable in a relator metamodel. It was discussed in Section 2.3.5 that not all metamodels seem to be appropriate for specifying a relation type. Although we listed some "heuristics" to address this, we suspect that this can be given a more formal foundation.

In the example macromodel of Section 3.1.1. we made two important observations that suggests the need for further investigation. First, we noted that in some cases there is a natural and informative way to construct relation names from the relation details. Second, when alternate decompositions exist in the multimodel then we need a way of moving to a hierarchical multimodel (and macromodel) and to be able to define relations between entire multimodels. It was suggested that defining the concept of a "multimodel morphism" and then defining relations based on these may be a way to approach this.

Finally, in Section 3.1.3 a hypothetical logical language over a multimodel metamodel was introduced in order to provide a higher level of abstraction for defining relation types. This takes the idea of moving from the micro to the macro level and applies it to support the definition of the relation types themselves.

6. Conclusions

In this paper we have suggested and explored three scenarios in which macro support for multimodeling in software engineering would be useful. First, it was proposed that the use of macromodels to reveal the structure of a multimodel would help with the model comprehension problem and we actually found this to be the case with the PUMR example from the telecommunications standards document. Second, the use of a macromodel template provides a way to structure the development process by allowing a lead designer to specify the models that must be created and also the types of relations that must hold between these and models that already exist. Finally, the fact that relations can hold between other relations means that operators can be defined as transformation relation types that manipulate relations. The AutoRefine operator is an example of this.

In order to support these scenarios, a general theoretical framework was defined for specifying model types and relation types in terms of metamodels and metamodel morphisms. These were then used to define the notions of a multimodel metamodel, multimodel and macromodel. Although these constructs provide a basic foundation for a formal theory of modeling, we plan to extend them further in a number of ways as suggested in Section 5. The ultimate goal of this work is to define a formal framework for modeling that can serve as a basis for tools that facilitate the multimodeling in software engineering.

7. References

[1] Astesiano, E., Kreowski, H. -J., and Krieg-Bruckner, B. *Algebraic Foundations of Systems Specification*. 1st. Springer-Verlag New York Inc., 1999.

[2] Bernstein, P. A. "Applying Model Management to Classical Meta Data Problems," In *Proc. CIDR*, 2003

[3] Brunet, G., Chechik, M., Easterbrook, S., Nejati, S., Niu, N., and Sabetzadeh, M. 2006. "A manifesto for model merging." In *Proceedings of the 2006 international Workshop on Global integrated Model Management* (Shanghai, China, May 22 - 22, 2006). GaMMa '06. ACM Press, New York, NY, 5-12.

[4] Boronat, A., Carsí, J. A., Ramos, I. "An Algebraic Baseline for Automatic Transformations in MDA," *Electr. Notes Theor. Comput. Sci*. 127(3): 31-47 (2005)

[5] Diskin, Z. "Mathematics of Generic Specifications for Model Management I," In *Encyclopedia of Database Technologies and Applications* 2005: 351-358

[6] *Eclipse Modeling Framework*: http://www.eclipse.org/modeling/emf/docs/

[7] Goguen, J.A. and Burstall, R.M. "Institutions: Abstract Model Theory for Specification and Programming." *J. ACM* 39(1): 95-146, 1992.

[8] Gurr, C. "On the isomorphism, or lack of it, of representations." In *Visual Language Theory*, K. Marriott and B. Meyer, Eds.: Springer Verlag, 1998, pp. 293-306.

[9] *Meta-Object Facility (MOF™) – Core specification V2.0:* http://www.omg.org/technology/documents/formal/MOF_Core.htm

[10] *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 (2001-08): http://portal.etsi.org/mbs/Referenced%20Documents/eg_201_872.pdf

[11] *Model Driven Architecture (MDA)* : http://www.omg.org/mda/

[12] *MOF™ Query / Views / Transformations (QVT) – Final Spec*: http://www.omg.org/cgi-bin/doc?ptc/2005-11-01

[13] *Rational Unified Process*: http://www-128.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf

[14] Sabetzadeh, M. and Easterbrook, S. “An Algebraic Framework for Merging Incomplete and Inconsistent Views.” *13th IEEE International Requirements Engineering Conference*, Paris, France, 2005.

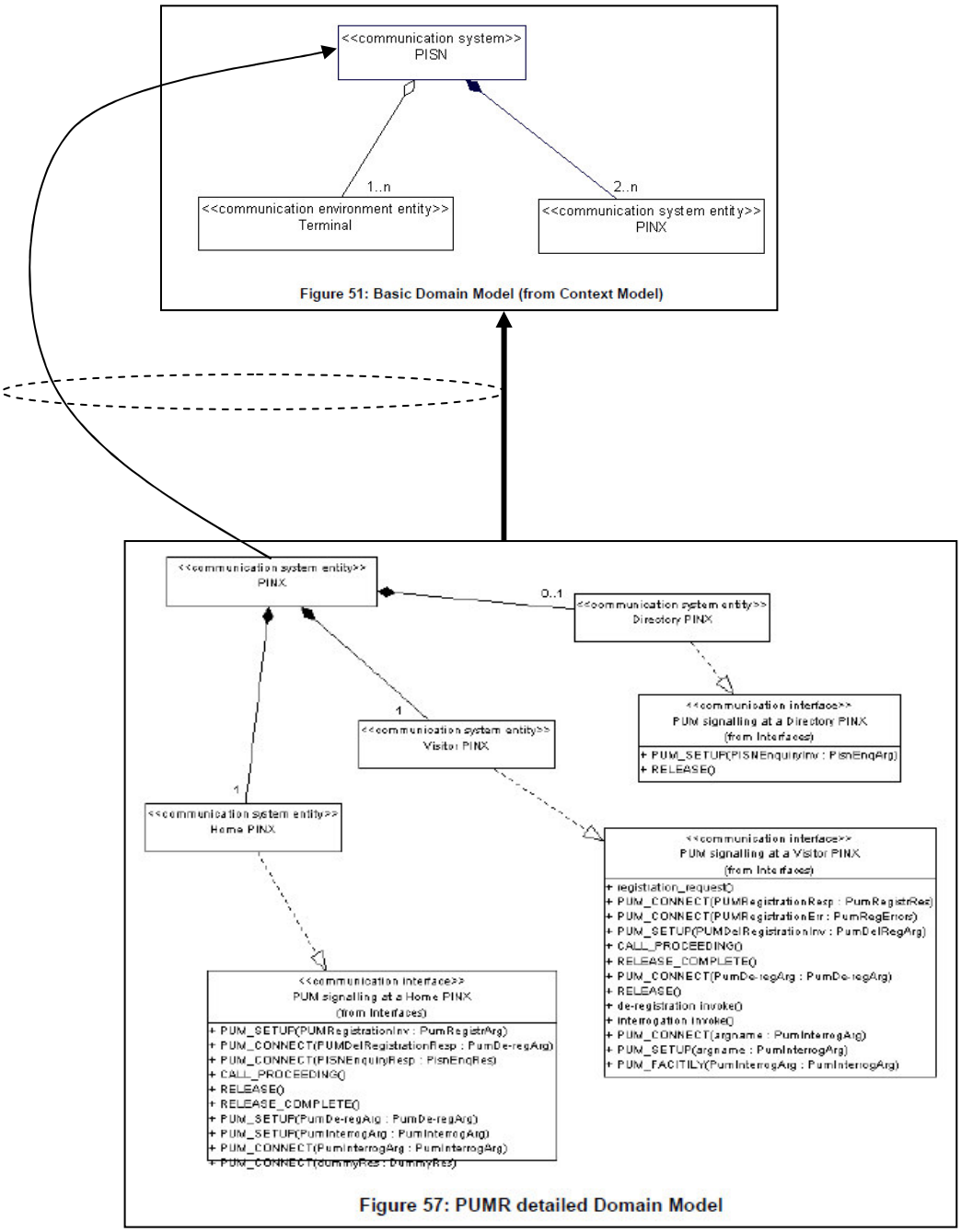
[15] Seidewitz, E. "What Models Mean," *IEEE Software*, vol. 20, no. 5, pp. 26-32, Sept/Oct, 2003.

[16] Tarlecki, A. “Abstract specification theory: an overview,” In M. Broy and M. Pizka, editors, *Models, Algebras, and Logics of Engineering Software*, volume 191 of NATO Science Series | Computer and System Sciences, pages 43-79. IOS Press, 2003.

[17] *UML 2.0 Metamodel*: http://www.omg.org/cgi-bin/apps/doc?ptc/04-10-05.zip

8. Appendix A – Examples of PUMR Diagrams

8.1. detailOf(ClassDiagram, ClassDiagram)



8.2. overlap(ClassDiagram, ClassDiagram)

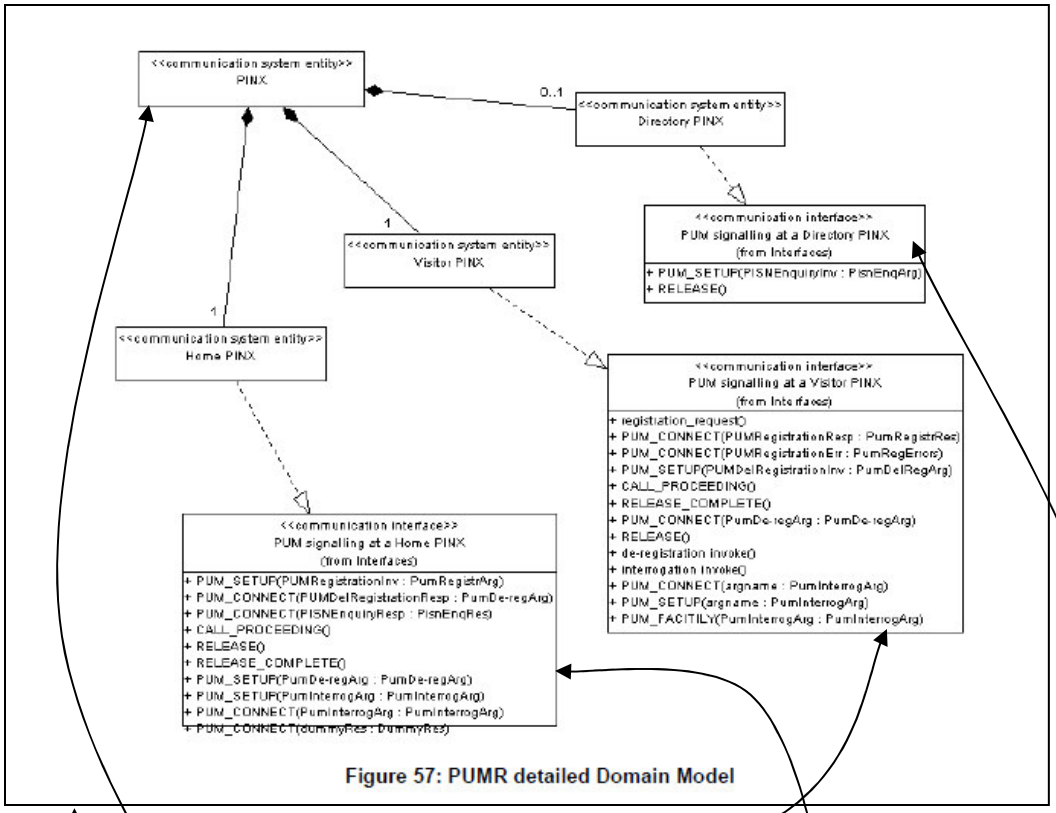


Figure 57: PUMR detailed Domain Model

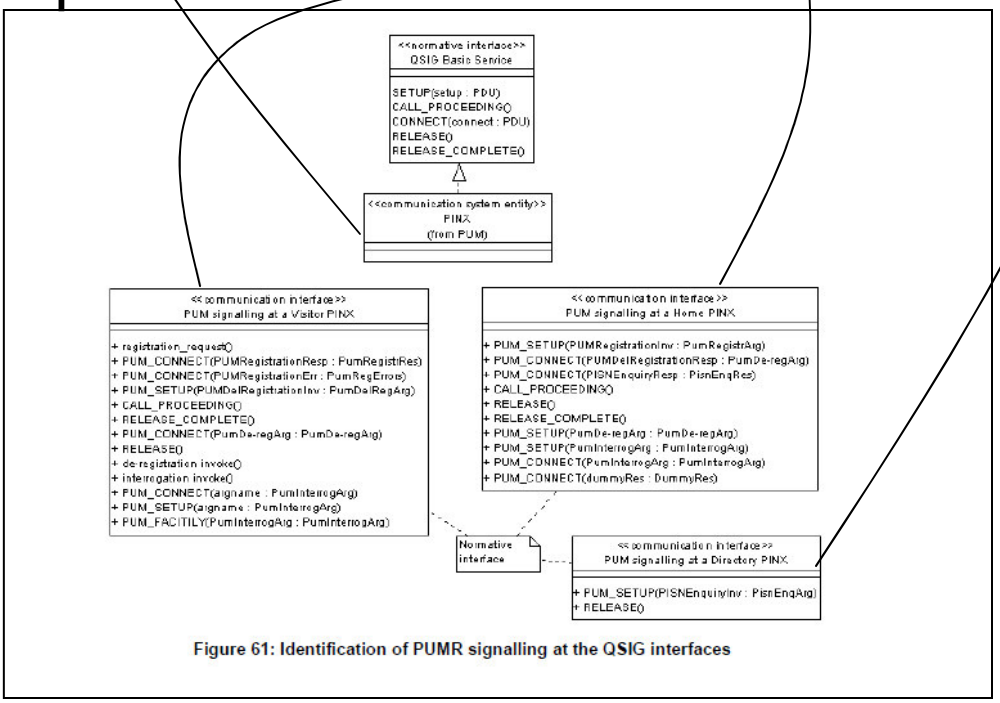


Figure 61: Identification of PUMR signalling at the QSIG interfaces

8.3. detailOf(StateDiagram, ClassDiagram)

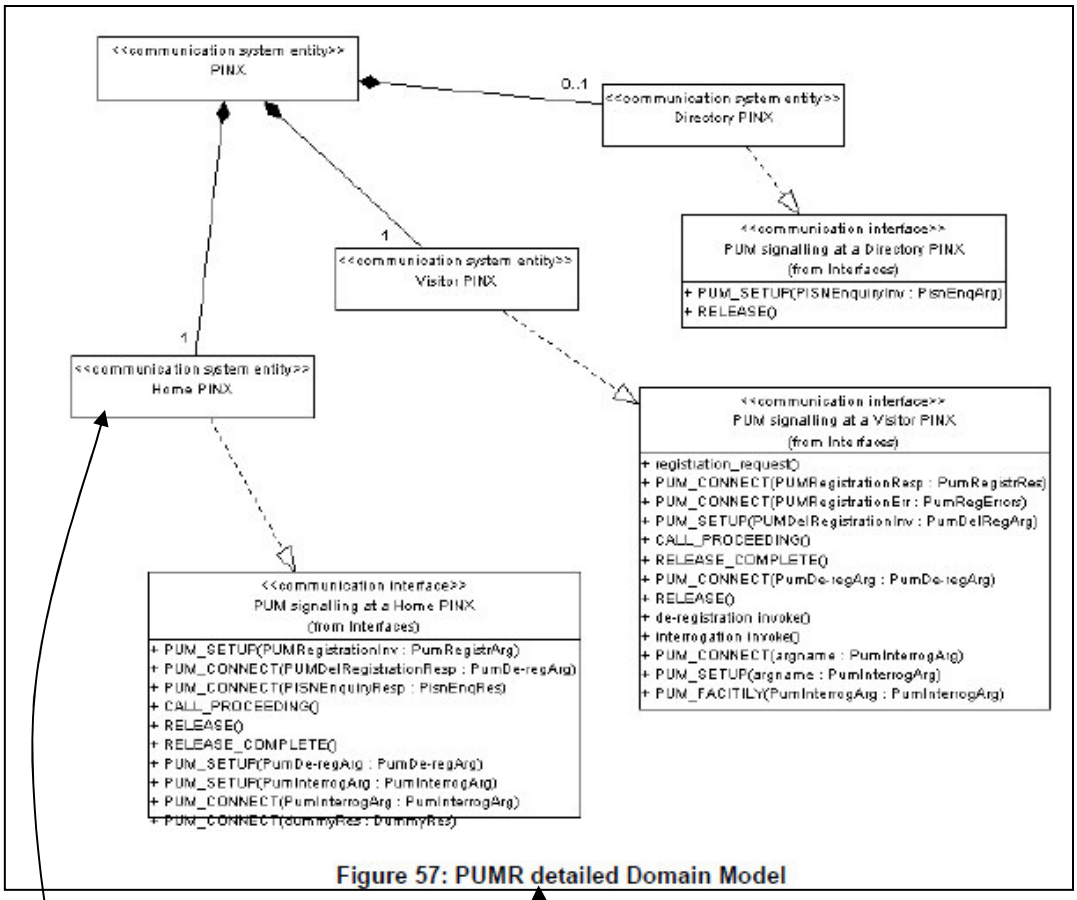


Figure 57: PUMR detailed Domain Model

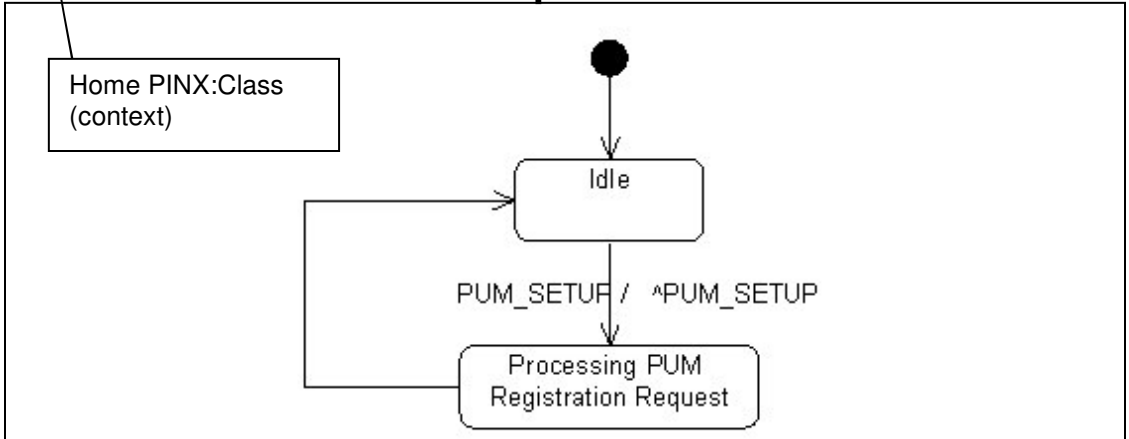


Figure 58: Statechart diagram showing the registration processing at the Home PINX

8.4. detailOf(StateDiagram, StateDiagram)

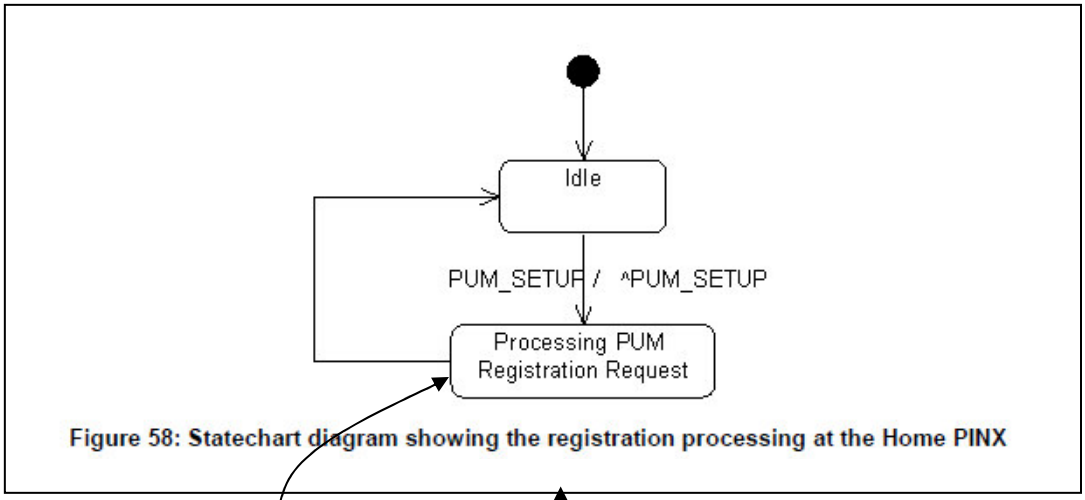


Figure 58: Statechart diagram showing the registration processing at the Home PINX

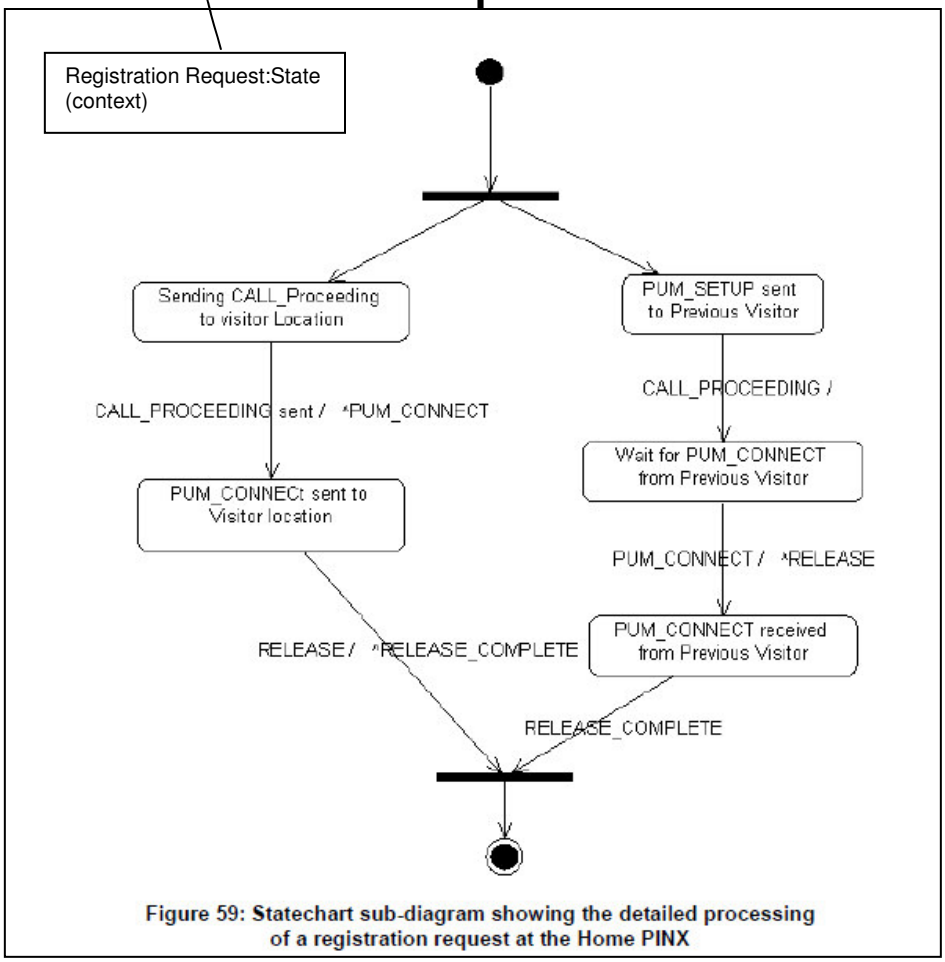


Figure 59: Statechart sub-diagram showing the detailed processing of a registration request at the Home PINX

8.5. instanceOf(ObjectDiagram, ClassDiagram)

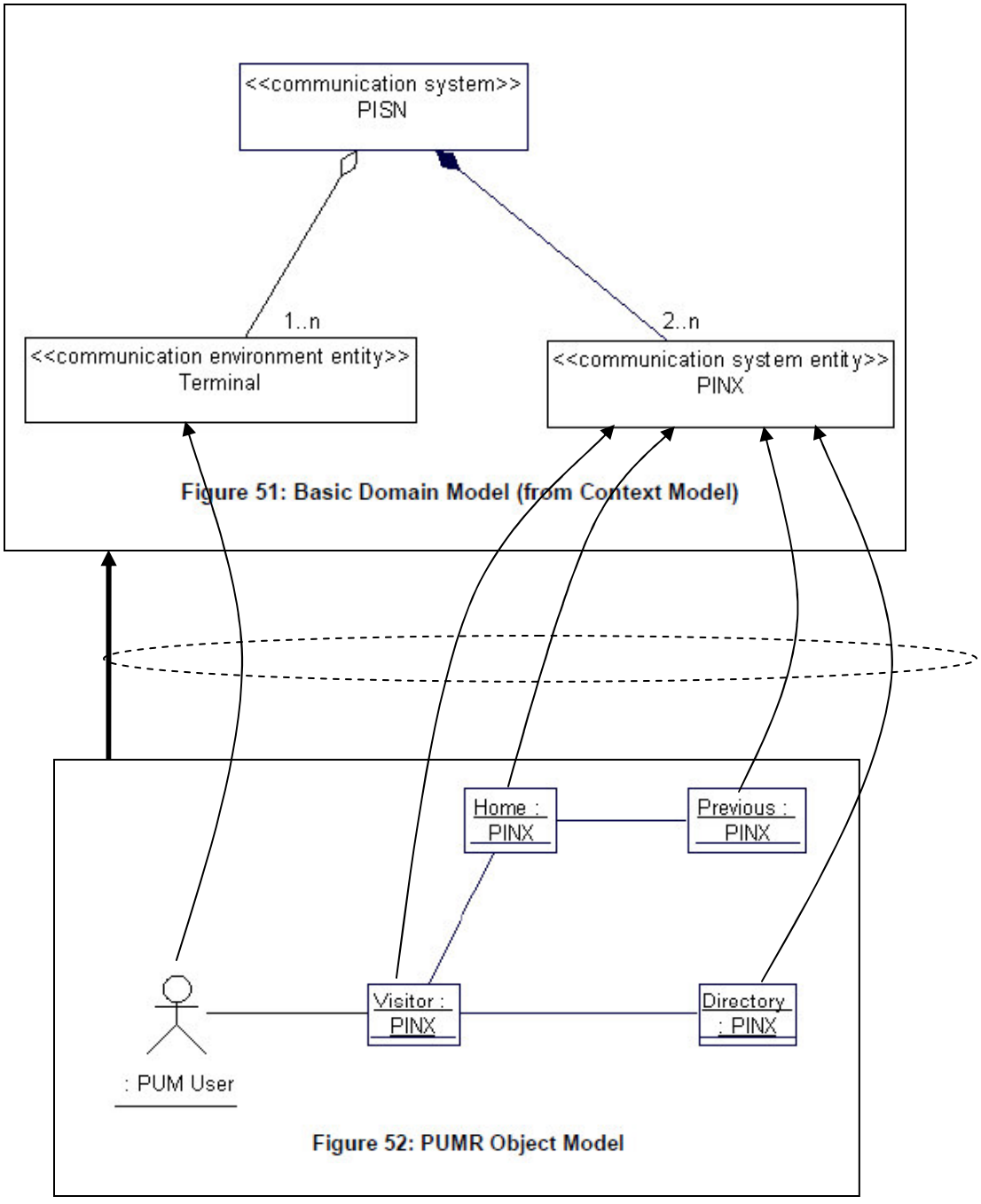


Figure 51: Basic Domain Model (from Context Model)

Figure 52: PUMR Object Model

8.6. objectsOf(SequenceDiagram, ObjectDiagram)

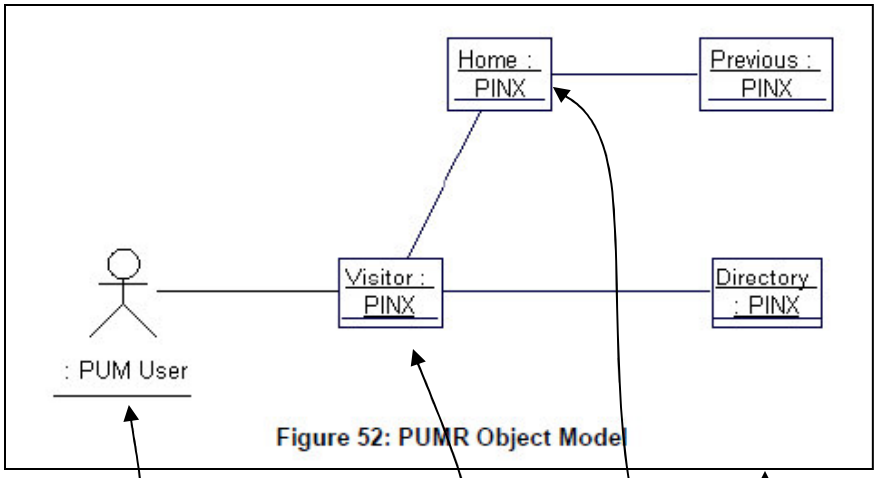


Figure 52: PUMR Object Model

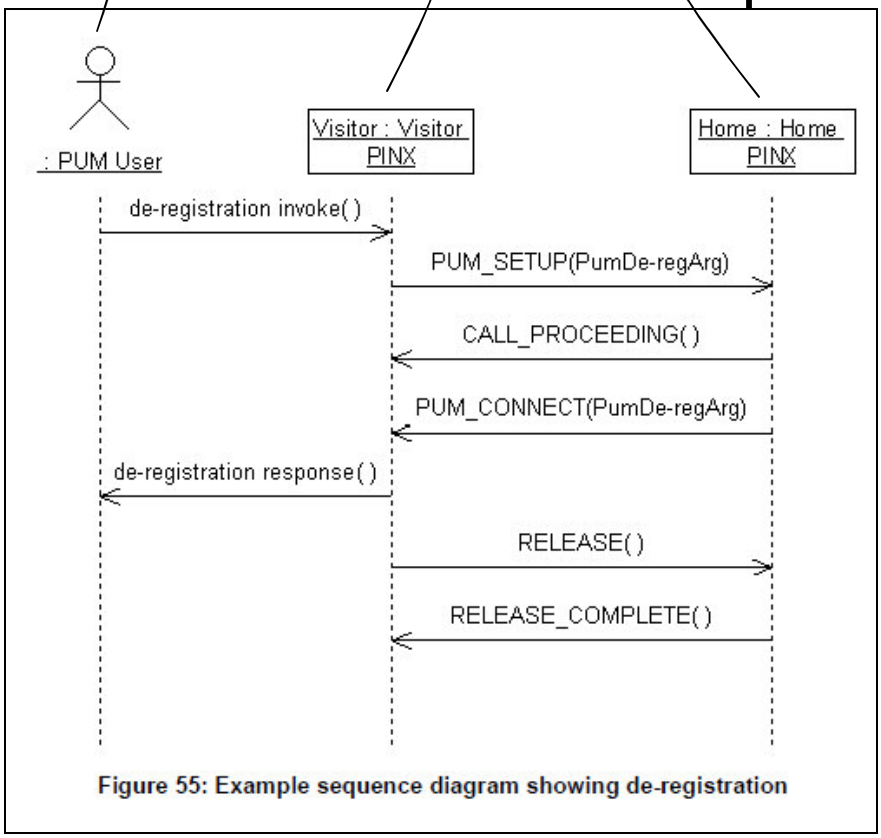


Figure 55: Example sequence diagram showing de-registration

8.7. caseOf(SequenceDiagram, SequenceDiagram)

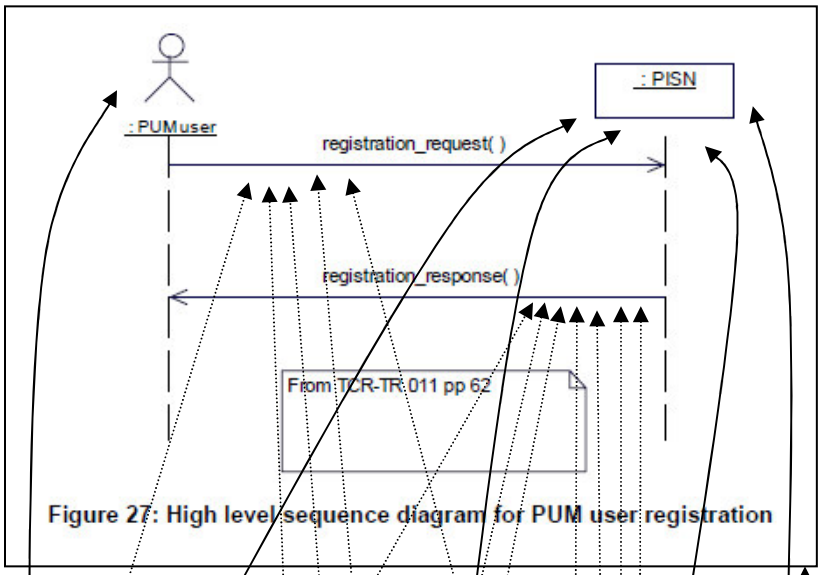


Figure 27: High level sequence diagram for PUM user registration

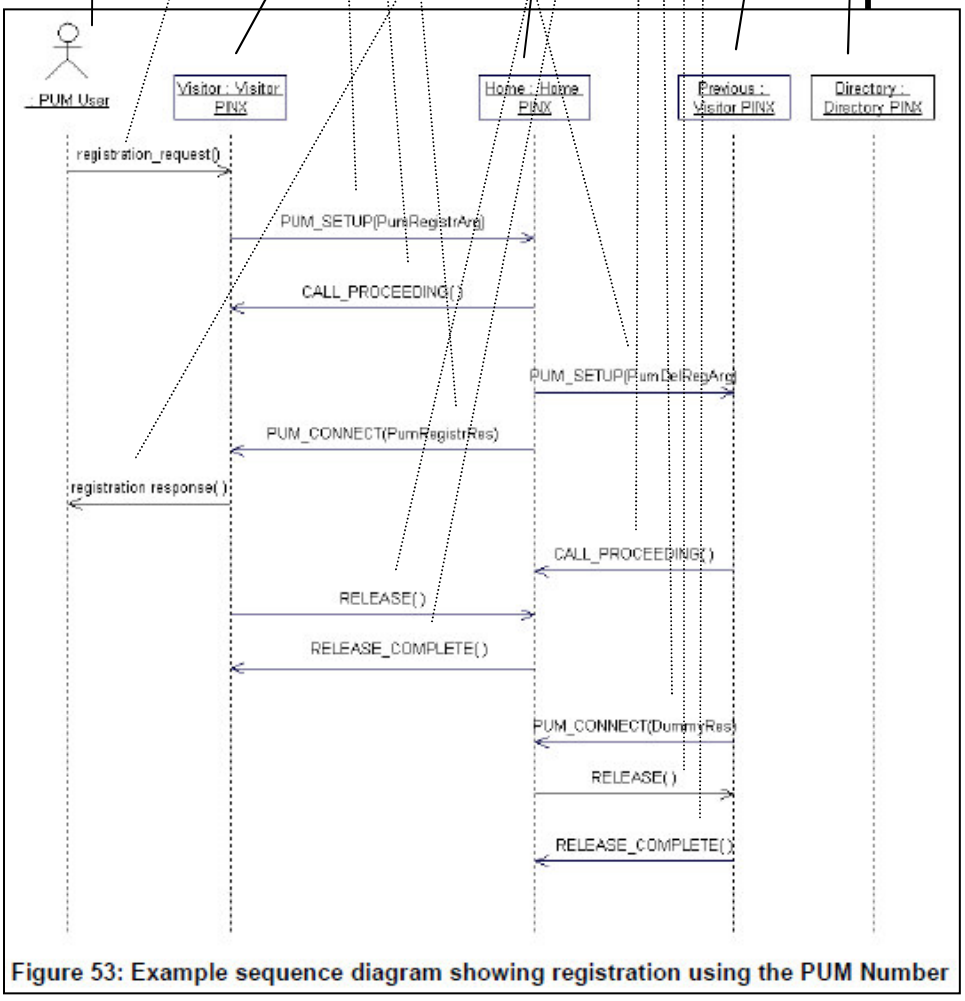


Figure 53: Example sequence diagram showing registration using the PUM Number

9. Appendix B – Submodel and Homomorphism

One of the most basic types of relations between models is that of the submodel relation. Intuitively, given a metamodel M, A:M is a submodel of B:M iff the elements of A are a subset of the elements of B and any function or relation that holds between elements in A also holds in B. The metamodel for submodel over CD (as defined above) can be expressed as:

sub(CD,CD) = CD.1 + CD.2 +

subsorts class.1 $\leq$ class.2

association.1 $\leq$ association.2

attribute.1 $\leq$ attribute.2

operation.1 $\leq$ operation.2

axioms

'CD.1 is a subset of CD.2'

$\forall x$:association.1. startClass.1(x) = startClass.2(x)
$\forall x$:association.1. endClass.1(x) = endClass.2(x)
$\forall x$:attribute.1. attrClass.1(x) = attrClass.2(x)
$\forall x$:operation.1. opClass.1(x) = opClass.2(x)

π_{sub1}:CD $\rightarrow$ sub(CD, CD), π_{sub2}:CD $\rightarrow$ sub(CD, CD) are metamodel monomorphisms into CD.1 and CD.2, respectively.

With the submodel relation, the two class diagrams actually *share the same symbols* in their abstract syntax. That is, the same class, association, attribute and operation symbols appear in submodel as in the whole model. Thus, the submodel could be considered to simply be a delineation of a portion of the whole model.

A more typical type of "submodel" relationship in software engineering is where the models are syntactically distinct, but semantically overlap. This is based on the standard algebraic notion of homomorphism. Homomorphism is an important type of syntactic relationship between models that can be easily defined over any type of model given our metamodeling language. Given a metamodel M, and two models A:M, B:M, a homomorphism f:A:M$\rightarrow$B:M is a function from the elements of A to the elements of B that preserves the element sorts and the way they are related via the functions and relations of M.

For example, the metamodel for homomorphisms over CD can be expressed as:

hmorph(CD, CD) = CD.1 + CD.2 +

func map:class.1 $\rightarrow$ class.2

map:association.1 $\rightarrow$ association.2

map:attribute.1 → attribute.2

map:operation.1→ operation.2

axioms

'mapping must preserve structure of CD.1'

$\forall x$:association.1, y:association.2. map(x) = y ⇒
map (startClass.1(x)) = startClass.2(x)

$\forall x$:association.1, y:association.2. map(x) = y ⇒
map (endClass.1(x)) = endClass.2(x)

$\forall x$:attribute.1, y: attribute.2. map(x) = y ⇒
map (attrClass.1(x)) = attrClass.2(x)

$\forall x$:operation.1, y: operation.2. map(x) = y ⇒
map (opClass.1(x)) = opClass.2(x)

$\pi_{hmorph1}$:CD → hmorph(CD, CD), $\pi_{hmorph2}$:CD → hmorph(CD, CD) are metamodel monomorphisms into CD.1 and CD.2, respectively.

In order to make this a proper model relation, an interpretation for instances of hmorph(CD, CD) is required. Let us assume that the portions due to CD.1 and CD.2 are interpreted as CD and the "map" functions are as semantic identity – that is, map(s1, s2) ⇒ s1 denotes the same thing as s2. Under this interpretation, :hmorph(A:CD,B:CD) preserves syntactic structure since the image of A under the mapping is a submodel of B and A has the same semantic content as this submodel. This means that homomorphism with the semantic identity interpretation captures the notion of the submodel relationship between syntactically distinct models.

The submodel and homomorphism relation types have the characteristic that they can be defined in a generic way for any model type given its metamodel.