Using Modeler Intent in Software Engineering

by

Richard Salay

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Your file Votre référence
ISBN: 978-0-494-72285-5
Our file Notre référence
ISBN: 978-0-494-72285-5

Using Modeler Intent in Software Engineering

Richard Salay
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2010

Abstract

Models are used widely within software engineering and have been studied from many perspectives. A perspective that has received little attention is the role of *modeler intent* in modeling. Knowing the intent of the modeler supports both model comprehension by providing the correct context for interpreting the model and model quality by clearly defining what information the model must contain. Furthermore, formal expressions of this intent allow automated support for this. Despite the value that the knowledge of modeler intent can provide, there are no adequate means in the current state of modeling practice for expressing this information. The focus of this thesis is to address this gap by providing mechanisms for expressing modeler intent both explicitly and formally.

We approach this problem by recognizing the existence of a *role level* in modeling where the role each model plays defines what information it should contain and how this is related to the information in other models. The specification of these roles is what we refer to as the expression of modeler intent. We then present a framework that incorporates four aspects of modeler intent at the role level: the *existential intent* for a model that arises in response to the need for a set of information by stakeholders, the *content criteria* that express what information the model is intended to contain, *model relationships* that express how models are intended to constrain one another and the *decomposition criteria* that express the intent behind how a model is decomposed into a collection of models. A key contribution of this thesis is the specification of the *macromodeling language* as a new modeling language designed for the role level that supports the expression of all four aspects of modeler intent. We evaluate these techniques by applying them to two real-world modeling examples.

Acknowledgments

I would like to thank my supervisors John Mylopoulos and Steve Easterbrook for their always sound advice and help throughout the twists and turns of this PhD. As well, special thanks goes to Marsha Chechik and Alex Borgida for their support. I would also like to acknowledge the ongoing support of my sister, parents and friends and thank them for lending an ear whenever I needed one. To Katie and Easton, thank you for making it so easy to be your stepdad. Most of all, I'd like to thank my wife, Gwen, without whom I would never have embarked on this journey and who has lived my PhD with me each day, through the good and the bad. You have my everlasting gratitude.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modeling has been a central activity in software engineering since its earliest days. Models have been used extensively to describe and prescribe systems and software and the subsets of the world that are affected by them. They can be analytical or analogue and act as a bridge between our mental models and their subject matter [J09]. Analytical models provide a way to reason about systems while analogue models allow systems to be simulated and observed. In either case, this allows developers to find defects early when the cost of fixing such defects is much lower [J00]. Models also provide a way to manage the complexity of software development by supporting levels of abstraction that discard irrelevant details and allow alternative ways of decomposing problem representations. As a result, they are ideal ways to communicate knowledge about a system in a way that supports comprehension by a wide variety of stakeholders. Finally, the fact that models of software are also "software" means that they have the special property that they can be systematically transformed into what they represent [AGM08]. This property is the key to the recent explosion of interest in model-driven software development [S06].

Modeling has been studied from many perspectives within software engineering research. This includes work related to metamodeling [e.g., MBJK90, KH06], formal model semantics [e.g., HR04, MLPS97], horizontal and vertical consistency (refinement) [e.g., HKR05, LMT09], model transformations [CH06] and model management [e.g., B03, SNEC07]. A perspective that has received little attention is the role that modeler intentions play in modeling and this is the focus of this thesis. In order to gain a preliminary understanding of the concept of modeler intent and its relationship to modeling, we consider some perspectives from philosophy and linguistics.

It is common to think of a model as a kind of linguistic artifact and modeler intentions play a role in both in its linguistic and artifactual aspects. As something linguistic, a model is equivalent to a set of statements in a language [S03, LSS94]. Accordingly, we can apply the semiotic triad of syntax, semantics and pragmatics to it. For natural language utterances, pragmatics deals with the intentions of the speaker and the effect that context has on the meaning of these utterances [SEPb]. At the level of whole texts, pragmatics addresses how a text is structured. For example, Rhetorical Structure Theory [RST] classifies the ways in which an author may intend different types of relationships between pieces of text in order to achieve different communicative effects such as elaboration or juxtaposition. Thus, we expect that the modeler’s intentions affect both how models should be semantically interpreted and how collections of models are structured.

As an artifact, a model is an object that is created by some author with the intent that it satisfies a particular purpose [Lad97]. The success of the author is measured by how well the model fits the author's intentions [SEPa]. Here we take the author to be the modeler and so the modeler intentions are used to define the required content of a model. Thus, being a well-formed linguistic entity is not by itself sufficient for being a model – the linguistic entity must also have a purpose that the modeler intends.

The central premise of this thesis is that making modeler intentions explicit and formal is useful to both modelers and to users of models. It helps modelers to clarify their intentions and ensure that the model fits them and it helps users to understand the content and interpret the model correctly. Expressing the intentions formally allows some of these benefits to be obtained in an automated way using modeling tools.

As an illustrative example, consider the class diagram *DTollPrice* shown in Figure 1.1. This is based on a transportation system example we use throughout this thesis (see Appendix E for additional information). This diagram is syntactically well-formed and according to the usual semantics of class diagrams we can interpret it as being equivalent to the following set of statements:

- Class *Vehicle* has subclasses *Car*, *SUV* and *Truck.*
- Class *TollTicket* has subclasses *SingleTripTicket* and *MonthlyTicket.*
- Class *Vehicle* has attributes *weight* and *numPassengers* of type *int.*
- Class *Truck* has an attribute *cargo* of type *Ctype.*

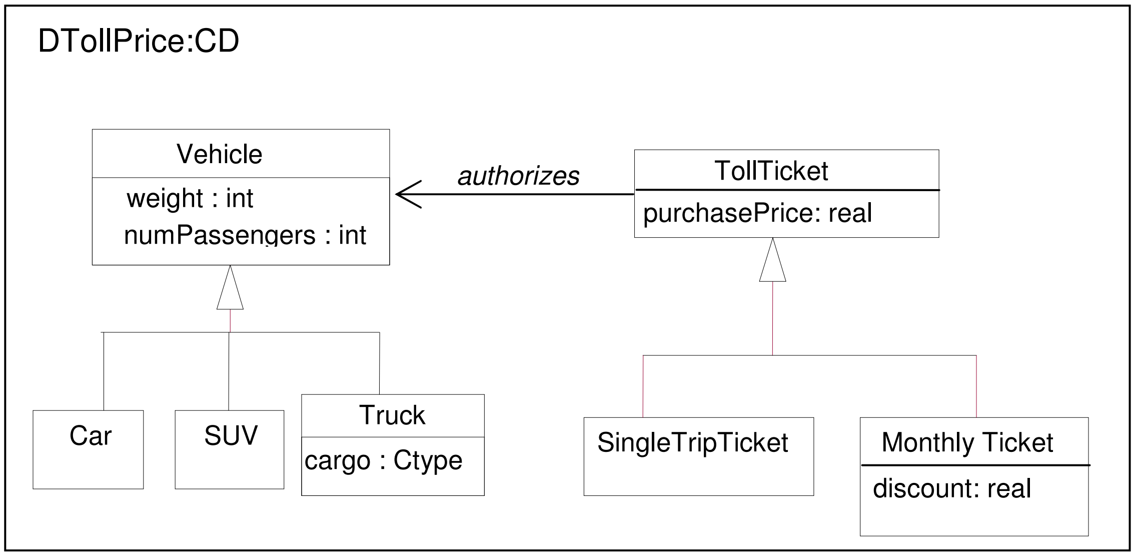


Figure 1.1.Transportation system diagram dealing with toll price.

- Class *TollTicket* has an *authorizes* association to class *Vehicle* and an attribute *purchasePrice* of type *real*.
- Class *MonthlyTicket* has an attribute *discount* of type *real.*

Now consider how this interpretation is affected as we disclose different aspects of the intent regarding this model with the following series of statements.

- (I1) *DTollPrice* is a (proper) submodel of the design model *TransportationSystemDesign*
    - We now know that this is not a complete model by itself but is part of a larger one and so we expect other submodels showing other parts of the design. Furthermore, we know that there may be other classes not shown here that may be related to these classes. We also know that this model is

described at the "design level" of detail (vs. implementation level, for example).

- (I1.1) The classes in *DTollPrice* are aggregated within the class *TransportationSystem* not shown in this diagram.
    - This elaborates (I1) and we can now infer the statement "classes *Vehicle* and *TollTicket* have aggregation relationships to class *TransportationSystem*"
- (I2) Diagram *DTollPrice* shows the parts of a transportation system dealing with toll payment
    - This gives us some sense of the purpose of the model and basis for assessing the relevance of the content to this purpose.
- (I2.1) All and only the attributes of vehicles that affect toll price are shown
    - This elaborates (I2) and we can now infer the statement that "*weight*, *numPassengers* and *cargo* are the only vehicle attributes that affect toll price"
- (I2.2) All the types of toll tickets are shown
    - This elaborates (I2) and we can now infer the statement that "*SingleTripTicket* and *MonthlyTicket* are the only types of *TollTicket*"

From a model user perspective, I1 helps us to understand the broader context of the model and its relationship to other model artifacts, I2 helps us to understand the purpose of the model and hence the rationale for its content, and I1.1, I2.1 and I2.2 allow us to infer additional statements that augment the standard semantics of the model. From a modeler perspective, asserting these statements helps to clarify what information to include or exclude from in *DTollPrice*. For example, I2.1 forces the modeler to think about whether there are any other attributes of vehicles that may affect toll price. As the transportation system design evolves, these statements provide guidance to other modelers that may modify *DTollPrice* to help them remain conformant to the original intentions – or perhaps, to challenge and change the intention if that is appropriate. Finally, if some of these statements can be formalized, then conformance checks and repairs can be performed in an automated way by modeling tools.

Expressing modeler intentions has a positive impact on model quality and this, in turn, can positively affect software quality [M05]. One of the earliest and most influential proposals for assessing model quality is based on the semiotic triad discussed above [LSS94]. Syntactic quality measures how well the model fits the modeling language and the goal is that all statements are syntactically correct. Semantic quality measures the relationship to the modeled domain and the goal is that all statements are correct (validity) and that all correct and relevant statements are included (completeness). Finally, pragmatic quality measures how the consumers interpret the model and the goal is accurate comprehension: i.e., that the consumer's interpretation is what was intended by the modeler. Based on this, the disclosure of modeler intentions supports improved model quality in two ways. First, it directly improves pragmatic quality by supporting comprehension. Second, it provides a way to assess the completeness aspect of semantic quality since we assume that the intended content of the model characterizes the set of statements that are relevant.

Despite the fact that expressing the intentions about models positively affects quality and is useful to stakeholders, modeler intentions are seldom made explicit and formal. In this thesis we take the position that this information should be given “first class” status and a multi-faceted approach is defined to support this stand.

1.1 Scope and structure of the thesis

In order to motivate the content and structure of the thesis, consider the framework in Figure 1.2 showing the different kinds of modeler intent that can arise within a modeling context. The entry point into the framework is through the need arising for a model due to the information requirements of some stakeholders such as software developers, testers, users, business decision makers, etc. For example, assume that the modeler is responding to the need of a group of software developers for the UML model *TransportationSystem* representing the detailed design of a transportation control system that the developers must build. This generates an initial *existential intent* on the part of the modeler that identified that the model *TransportationSystem* must be created. At this point, as the modeler considers the purpose of the model, they recognize that *TransportationSystem* is related to other models in well defined ways (arrow *R*). For example, *TransportationSystem* must *satisfy* the requirements model *TransportationSystemReq*, it must *refine* the architecture model *TransportationSystemArch*, etc. All of these are intended relationships that the model must conform to.

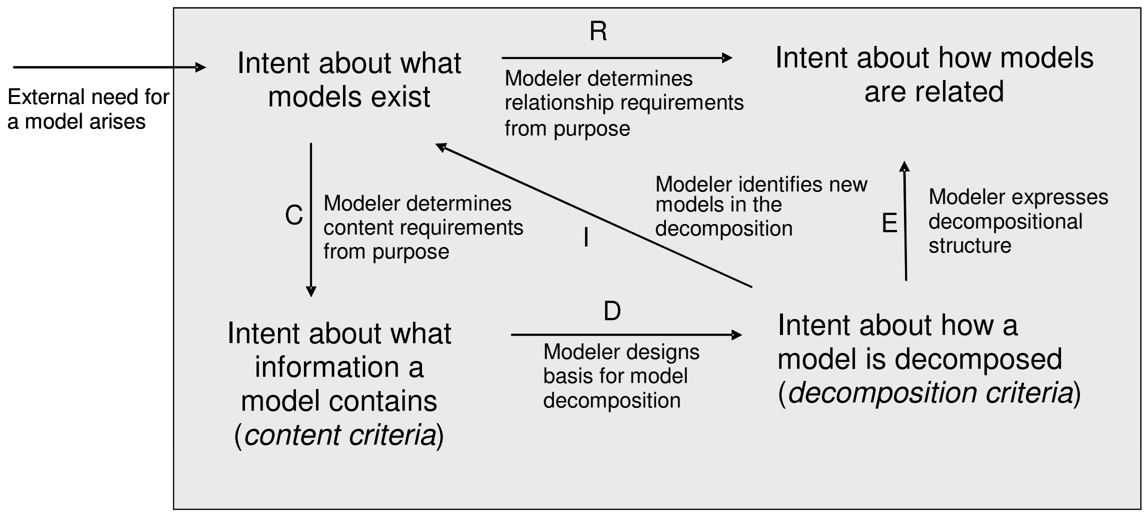

External need for a model arises
Intent about what models exist
R
Modeler determines relationship requirements from purpose
Intent about how models are related
C
Modeler determines content requirements from purpose
I
Modeler identifies new models in the decomposition
E
Modeler expresses decompositional structure
Intent about what information a model contains (content criteria)
D
Modeler designs basis for model decomposition
Intent about how a model is decomposed (decomposition criteria)

Figure 1.2. A framework for modeler intent.

Before the modeler actually creates *TransportationSystem*, they must decide what information should be in it based on what information they think would satisfy the needs of the developers (arrow *C*). This gives rise to an intention about what the content of *TransportationSystem* should be and we refer to a characterization of this as *content criteria*. The modeler may then recognize that this information should not be created as a single monolithic model but must instead be decomposed into multiple related "partial" models (arrow *D*). Doing this involves both the identification of new models (arrow *I*) and the expression of the decompositional structure of the set of partial models (arrow *E*) that we call *decomposition criteria.*

In this thesis, modeler intent is expressed using a new type of model called a *macromodel*. A macromodel allows a set of models to be represented at a macroscopic level of detail (i.e., without looking at the particular content). The different kinds of modeler intent shown in the framework are expressible within a macromodel. This consists of:

- The existential intent concerning models and collections of models
- The relationships and the hierarchical grouping structure that must hold between models in order to satisfy their purpose
- The *content criteria* that express the information that a model is required to contain in order to satisfy its purpose.
- The *decomposition criteria* that expresses the basis on which a model is decomposed into smaller models.

In Chapter 2 we analyse the concept of modeler intent in order to define it more clearly, distinguish it from other related notions and define the scope of the thesis more precisely. This analysis is used in a review of related work in Chapter 3 to assess the adequacy of current approaches in modeling research for the problem of expressing modeler intent. Chapter 4 introduces our formalism for expressing relationships between models and the classification of these into relationship types. Chapter 5 elaborates the approach to expressing content criteria about models based on this. In Chapter 6, the macromodel language is described and its syntax and formal semantics are defined and in Chapter 7 the issue of decomposition criteria is explored. Chapter 8 describes two in-depth applications of the approach to actual modeling projects in order demonstrate and evaluate our methods. Then in Chapter 9, a prototype tool is described that implements a key part of the automation benefit of the approach. Finally, in Chapter 10 we draw some conclusions and point to future work.

Parts of this work have been published previously. The approach to relationship types in Chapter 4 and the core of Chapter 6 on macromodeling can be found in [SME08, SME09]. The development of content criteria for submodels that comprises most of Chapter 5 can be found in [SM09]. In that paper the earlier term “coverage criteria” was used for this instead of “content criteria” as we do here. Parts of Chapter 9 can be found in [SME09].

Chapter 2

Preliminaries

In this chapter our aim is to analyze the notion of modeler intent more closely in order to clarify key concepts and distinctions and provide the conceptual groundwork for the subsequent chapters. We will use the intent framework introduced in Chapter 1 and shown in Figure 1.2 as a structuring guide for the analysis.

2.1 The general setting

Modeler intent is a kind of information about models. As such it exists in the *development world* [MBJK90] that consists of artifacts such as models, activities such as modeling and actors such as modelers.

2.1.1 Actors

We assume that a modeler works in the context of a particular *modeling project* that consists of an evolving set of interrelated models. Note that the modeling project can be part of a larger effort such as a software development project, documentation project, etc. Furthermore, if a development methodology is being followed, we think of the modeling project as being part of an instance of this methodology. We begin by identifying some key stakeholder roles relative to a model in the context of a modeling project:

- *Modeler*
    - *Definer*: The model definer is a modeler who decides that the model must exist and defines what information it should contain.
    - *Producer*: The producer is a modeler who creates the content of the model in accordance with the requirements of the definer.
- *Consumer*: The consumer uses the model to satisfy their goals.

The modeler role is subdivided into the definer and producer roles to reflect the fact that a modeling project may involve many modelers and that the modeler who decides that a given model must exist may not be the same one that creates the model. For example, a senior designer on a software project may play the definer role for a certain model that a junior designer must create as producer. The junior designer may then play the definer role with respect to how they wish to subdivide the model into various submodels and then play the producer role in creating them. We assume that the modeler's intentions are manifested when they are playing the definer role and that any expression of modeler intent is created by a definer. As discussed in the introduction, both the producer and consumer may use these expressions of modeler intent to support their activities.

2.1.2 Modeler's intentions vs. stakeholder's intentions

We distinguish the concept of modeler's intentions from that of stakeholder's intentions discussed in early requirements engineering. The intentions of a stakeholder regarding a system "to be" consist of the goals that these stakeholders have about the system. These can be expressed using modeling languages such as i* [YM97]. In the context of modeling we might consider the corresponding stakeholders to be the potential consumers of a model. These consumers have goals about how they intend to use the models that will be produced and hence this defines the purpose of these models. A modeler interprets these goals and from them forms an idea of the information and structure a model should have in order to satisfy stakeholder goals. Thus, in this thesis, we assume that the consumer's intentions are their goals for using a model while the modeler's intentions are the result of translating these goals into a specification of what information a model should contain and how it should be structured.

Note that when there are multiple modelers, they may disagree about the consumer's intentions – i.e., there may be conflicting modeler intentions for a given model. One of the benefits of expressing modeler intent formally is that these kinds of conflicts can be detected since inconsistency checking can be formalized. Another benefit is that a formal expression of intent brings a level of precision that helps clarify the points of disagreement. This is useful even when there is only one modeler since it helps to clarify for themselves what they really intend.

2.1.3 Intent about the subject vs. intent about a model

Models are typically used to prescribe something or to describe something. We will call the thing that the model is about, the *subject* [J09], and we can take a model to be equivalent to some set of statements about the subject [S03]. In a descriptive modeling scenario, there is an existing subject that is being described and the content of the model can be true or false depending on whether or not it correctly describes the subject. In a prescriptive modeling scenario (e.g., software specification), the subject does not (yet) exist and we are concerned with the consistency of the content of the model – i.e., it must be the case that a subject that satisfies the model could exist.

If the modeler is creating a prescriptive model of the subject then they are using the model to express their intent about the subject. For example, a design model of a piece of software expresses how the modeler intends the software to work. However, the modeler intent about the subject is different than the modeler intent about a particular model of the subject. In the former case, the intent is concerned with the desired properties of the subject while in the latter case it is concerned with the desired properties of the model. For example, if the subject of the design model *RoadRail* is a transportation system, then the intent that the transportation system supports automated signals at all railway crossings is an intent about the subject and this could potentially be expressed within *RoadRail*. On the other hand, the intent that the model *RoadRail* shows how the road and rail components of the transportation system are related is an intent about the model, not the transportation system.

In general, we don't expect the intent about the model to bias the intent about the subject that is expressed within the model. The intent about the subject is concerned with what sentences about the subject are true or false while the intent about the model is concerned with what *kinds* of sentences the model should express and this should not say whether these sentences are true or false. Still, this kind of separation is not always possible. For example, simply asserting that there exists a model *RoadRail* with the intent described above already assumes that the transportation system *has* both road and rail components and this is clearly saying something about the transportation system itself. It turns out that the intent about models and the intent about the subjects of these is models are tangled in various ways and we will tease these out in later chapters of this thesis.

2.2 Expressing existential intent – model roles

2.2.1 Model roles vs. models

An important distinction that is not often made clear in research on modeling and is central to the concerns of this thesis is that of model roles vs. the models that play those roles. We define a *model role* within a modeling project as a reification of the need for a model that satisfies a particular purpose while a model is an artifact that can play (or *realize*[1]) one of these roles. Thus, a model role is not merely an attribute of a model – it has its own existence independently of the models that play it. The acknowledged existence of a model role within a project represents an *existential intent* on the part of the modeler (as definer): an expectation that a model playing this role should exist within the project.

In practice, when actors in a modeling project talk about models, they are usually talking about model roles rather than the actual artifacts that play those roles. For example, consider the following typical sentences:

1. "Where is the error handling model?"
2. "Who will create the structural model of the traffic light controller?"

[1] We will use the terms *play* and *realize* interchangeably. Thus a model is a realization of a role.

3. "Here is the latest version of the toll ticket purchase flowchart."

In the three sentences above, the phrases "error handling model", "structural model of the traffic light controller" and "the toll ticket purchase flowchart" all refer to model roles rather than models themselves. Only in sentence (3) is an actual model referred to as the referent of the word "Here". The lifetime of a model role is longer than that of the models playing the role and can exist even if no model plays the role. Furthermore, different models can play a given role at different times although only one can play the role at a time. For example, in sentence (2) it is clear that no structural model exists yet so the model role precedes the existence of a model playing the role. In sentence (3) the referent of "Here" is the model that is playing the role at the time that the sentence was uttered.

2.2.2 Model level vs. role level

In the current state of practice, most explicit modeling activity takes place at the *model level* rather than at the *role level*. That is, modelers spend most of their time expressing the content of particular models rather than their intentions about that content. The role level is a macroscopic level of abstraction on the model level since model roles identify models without being specific about their content.

Abstraction is a powerful mechanism for managing complexity by supporting top-down understanding but in order for a level of abstraction to be useful it must provide some form of summarization of the details that are omitted [M09]. We will assume that a summary is any property that abstracts from the content of the model. Thus, the modeler intent at the role level is one such summary and we argue that this summary in particular is a key one for supporting a stakeholder's comprehension of a collection of interrelated models.

To see this, consider the following different possible summaries of diagram *DTollPrice* in Figure 1.1:

1. *DTollPrice* contains seven classes
2. *DTollPrice* contains some details for classes *Vehicle* and *TollTicket*
3. *DTollPrice* contains all information related to toll ticket pricing
4. *DTollPrice* contains classes *Vehicle* and its three subclasses and class *TollTicket* and its two subclasses

These are all valid summaries of *DTollPrice* and give different amounts of information about it. However, of these, (3) is distinguished because it defines the role intended to be played by the model and expresses the modeler intent. In general, the choice of summary used should correspond to the task for which the abstraction will be used. For example, if the goal is to efficiently decide which model has more than nine elements, then a summary like (1) is most appropriate. What we suggest is that the summary that is most appropriate for supporting the task of *model comprehension* is the one that is drawn from the purpose of the model because this explains why it contains the information it contains. A key message of this thesis is that explicitly modeling the role level as part of normal modeling activity benefits stakeholders by providing this useful level of abstraction on the collection of models in a project.

2.2.3 Bottom-up vs. top-down modeling

The typical approach to modeling based on the model level is bottom-up: models are created incrementally by adding content to them and new models are introduced as needed. During bottom-up modeling, the act of expressing the intent about a model is useful because it forces a modeler to clarify what the purpose of the model is and provides a way to ensure that the content is consistent with this purpose.

The fact that a model role can precede the existence of a role player means that a definer can express existential intent as a way to direct modeling activities. Thus, in addition to bottom-up modeling, the existence of the role level creates an opportunity for *top-down modeling* by allowing the required models and their intent to be specified before content is created. While bottom-up modeling is more *organic*, top-down modeling is more *designed* since the "information architecture" of how content is distributed across multiple models can be prescribed. This also provides a mechanism for managing the modeling process by allowing different model roles to be assigned to different modelers.

2.2.4 Model roles vs. model types

The type of a model is defined by its modeling language and this is typically characterized by a metamodel, has a notation and has associated tools such as editors. A model role is a *use* of a modeling language in a particular context. Thus, model roles and model types are related but distinct concepts.

A given model type can be used for many roles and a given role could be modeled using potentially many types; however, the combinations are not arbitrary. For example, a

Statechart may be used to play the roles “behaviour of class *Car*”, “process for purchasing a toll ticket” or “behaviour of the traffic network” but it could not be used to play the role “organizational structure of toll operators” because Statecharts do not provide the appropriate concepts required for this modeling task. On the other hand, the role “process for purchasing a toll ticket” could be modeled using a Statechart, Flowchart, UML Activity Diagram or UML Sequence Diagram but not using a Class Diagram.

2.2.5 Role types vs. roles

We can also distinguish *role types* from roles. A role type characterizes a class of similar roles and can partially specify the modeler intent for its instances. For example, *DesignModel* is a role type that partially defines a purpose and relationships to other role types like *RequirementsModel* and *ArchitecturalModel* but it may not be specific about some details such as the model type and other relationships that should hold. Each instance of *DesignModel* is a particular role such as “the design of the transportation system” which has a similar but distinct purpose from the role “the design of the accounting system”.

2.3 Expressing intent about content – role constraints

2.3.1 Model purpose

As discussed in the introduction, a model is taken to be a linguistic artifact and so it is a collection of information that is constructed to satisfy a purpose. Model purpose is the key driver of modeler intent. Within a software development context, Lange [LC05] has classified some of the possible purposes a model may have (See Table 2.1). If the context is broadened to include entire organizations that produce software, then the possible

Table 2.1. Lange's enumeration of kinds of model purpose within software engineering.

Modification	The model and the system enable easy modification. Possible modifications are corrections, i.e. removal of errors, extensions of the system, or adaptive changes due to changed requirements.
Testing	The model is used to generate test cases.
Comprehension	The model and the system are easily comprehensible, i.e. the system elements and their functionality are modelled such that they can be understood correctly in a reasonable amount of time.
Communication	The model enables efficient communication about the system's elements, their behavior and the design decisions. Communication includes communication during the development phase with different stake holders and documentation for understanding the system in later phases such as maintenance.
Analysis	The purpose of the model is to explore and analyse the problem domain including its key concepts and making some early design decisions.
Prediction	The model is used to make predictions about quality attributes of the eventual implemented system.These predictions are used to make early and therefore cheaper improvements of the architecture and design.
Implementation	The model is used as basis for (manual) implementation of the source code of the system.
CodeGeneration	The model is used to automatically generate the source code of the system. Code generation can be complete or partial (only a skeleton of the source code is generated).

purposes of models can include such things as support for persuasion (sales & marketing), training (human resources) and decision making (marketing & management)[2].

[2] Of course, one may come up with "unorthodox" purposes for a model, e.g., to impress my boss, to decorate my wall, etc. however we do not consider these types of purposes in this thesis.

Although the possible purposes of a model seem quite diverse, the only thing a model can actually provide is a *set of information*. Thus, we may reduce the question of the model's purpose to that of what information it should provide and by what means – i.e., what the requirements for the model are. With software artifacts we can have both functional and quality (or, non-functional) requirements. Similarly, with models we can state its *content* requirements (what information it should provide) and its quality requirements (how it provides the information).

The content requirements of a model define specifically what information belongs in the model and what does not. The quality requirements of a model specify more generic properties that the information must satisfy and these can be captured by model quality metrics such as complexity, balance, conciseness, precision, flexibility, etc. Model quality metrics have received significant research attention [e.g., LC05, M98]. In this thesis, we focus exclusively on content requirements. The modeler (as definer) interprets the content requirements as a *set of constraints* that the information in the model must satisfy – i.e., as a specification for the model content. Thus, we take this set of constraints to represent the modeler intent about the model's content.

2.3.2 Kinds of model constraints

At this point we need to consider more carefully the term "constraint" used in the above exposition. The concept of constraint is a very general one and different kinds of constraints have different functions within modeling. To help characterize these kinds, consider the taxonomy of constraints shown in Figure 2.1. We describe these as follows:

- *Type constraints* are due to the rules for correct usage of the modeling language used by a model independently of the particular purpose for which the model is used. These include semantic constraints that ensure that the content is semantically interpretable and consistent and syntactic constraints that ensure that the model is renderable using a particular notation.
- *Role constraints* are due to the intended purpose of the model and thus are usage context dependent. From a linguistic point of view, these carry information about pragmatics and determine how the same model could be interpreted differently in different usage contexts. This is further subdivided:
    - *Method-level constraints* are role constraints due to the development methodology being followed to create the content. These include constraints that require the existence of models playing particular roles, constraints on the sequence of modeling activities, model dependency constraints, etc.
    - *Project-level constraints* are role constraints due to the modeler's design decisions and interpretations of stakeholder needs within a particular project (i.e., in an occurrence of a development method). These include constraints that require the existence of models playing particular roles, constraints that define what content is relevant to and required by the purpose, and constraints that govern the decomposition of a model.

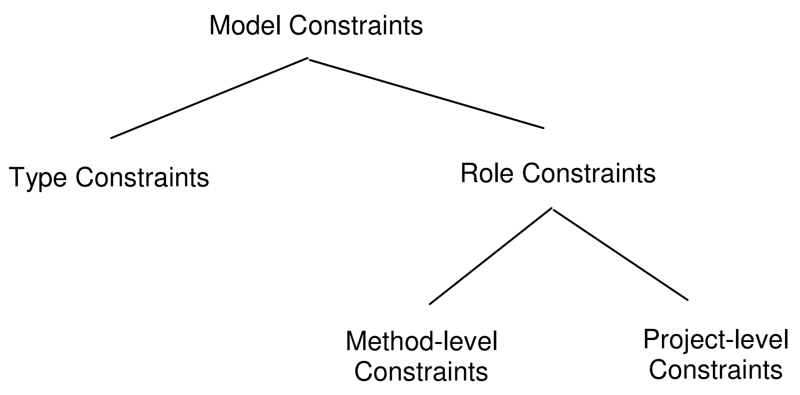


Figure 2.1. A taxonomy of constraints that apply to models.

For example, assume that we have some UML class diagram *DTollTicket* from the transportation system example and consider the different kinds of constraints on its content as shown in Table 2.2. Most of the research on model constraints within software engineering has focused on type constraints and associated notions like consistency [e.g., LMT09]. To a lesser extent, work on software methods has addressed method-level role constraints [e.g., NFK94, SPEM]. The focus of this thesis is on project-level role constraints.

Characteristics of project level role constraints

A key difference between project-level role constraints and the other two kinds is the level of generality of the constraint. Type constraints are common to all models of the same type (i.e., same language) and are typically defined as part of the metamodel for the model type. Method-level role constraints are common to all models playing the same role type in different occurrences of a method (i.e., different projects). If these are expressed, they are found as part of the method definition. In contrast to both of these,

Table 2.2. Examples of different kinds of model constraints.

Kind of constraint	Constraint on *DTollTicket*
Type	(C1) *DTollTicket* can't contain a class that is a subclass of itself.
Role (method-level)	(C2) *DTollTicket* cannot use multiple inheritance (C3) Every class of *DTollTicket* requires a corresponding Statemachine Diagram to show its behaviour.
Role (project-level)	(C4) *DTollTicket* only contains only the subclasses and details of class *TollTicket*.

project-level role constraints are specific to a particular model role within a particular project[3]. This means that we view these role constraints to be *part* of the modeling project rather than outside or above it. Thus, a violation of such a constraint could be addressed either by changing the model content or by changing the constraint itself to reflect a change in the intent about the content. For example, if constraint (C4) is violated by *DTollTicket* because it contains the class *Car* and this is not a subclass of *TollTicket*, then a valid response on the part of the definer is to recognize that the intent of *DTollTicket* has evolved and it now should also allow subclasses of *Vehicle* of which *Car* is one.

A software method typically identifies the abstract input and output roles for development activities (e.g., requirements gathering, design, etc.) as well as the constraints between these roles. Since these are defined at the method level scope that

[3] Note that since a metamodel is kind of model, we can also talk about the intent of the modeler who creates a metamodel. This is still a project level constraint but for a modeling language design project. In Chapter 8, we illustrate this by formalizing role constraints for some submodels of the UML metamodel.

spans multiple projects we could consider them to be expressing the intent of the *development organization* rather than the intent of modelers. As such, the method level can only be used to express a limited amount detail regarding these roles. For example, the Rational Unified Process (RUP) [K00] identifies the existence of an “elaboration phase architecture model” but in a particular project this would typically consist of multiple partial models (or diagrams) to address different concerns, different stakeholder views, different modeling languages, different decompositions, etc. Each of these specific models in the elaboration phase has distinct modeler intentions associated with them and these cannot be expressed at the method level.

2.3.3 Usage of role constraints

The role level constrains (via role constraints) the possible project configurations at the model level to those that conform to the modeler’s intentions. This can be utilized to support modeling in various ways including:

- Conformance checking: model defects can be identified by checking for conformance with the role constraints.
- Extension-to-conformance: when model roles exist for which there is no corresponding model, or when models are incomplete, the role constraints limit the possible content for this missing information. Extension to conformance is the process of inferring missing content by extending the incomplete models according to the constraints.
- Change propagation: when a change in a model causes role constraints with related models to be violated then a repair action may require the related models

to be changed. These changes may then instigate changes in other models, and so on. Thus, role constraints between models can drive change propagation.

When role constraints are formalized, these supports can be built into modeling tools and automated. A key objective of this thesis is to provide formal expressions of modeler intent in order to enable this.

2.4 Expressing intent about relationships

The purpose of a model may require that its content constrain the content of other models or be constrained by the content of other models. We refer to a constraint that is intended to hold between particular roles as an intended model relationship. Specifically, when such a constraint is intended to hold between certain roles then it means that the possible combinations of models that can play these roles are restricted because only certain combinations satisfy the constraint.

There are different kinds of relationships one might express between roles but in this thesis we limit ourselves to ones that express intent about the content of the models that play the roles. To illustrate, consider the following three sentences concerning model roles *R1* and *R2*:

1. *easierToPlayThan*(R1, R2)
2. *createdBefore*(R1, R2)
3. *refines*(R1, R2)

Sentence (1) asserts that role *R1* is easier to play than *R2*. Assume that we interpret this relationship as saying that any model that can play *R1* (i.e., satisfies its role constraints) can also play *R2* but not vice versa. Sentence (2) says that in the project, role *R1* must be filled (i.e., its existential intent must be satisfied) before *R2* is filled. Finally, sentence (3) says that at any given point in time in the project, the model that plays *R1* must refine the model that plays *R2*.

Of these three cases, (1) does not express an intended relationship between models since it either holds or doesn't hold regardless of the particular models that play *R1* and *R2*. Sentences (2) and (3) do express intent about models since they may or may not be satisfied by the models that play the roles; however, only sentence (3) expresses intent about the relationship between the *content* of the models that play the roles. Sentence (2) is not an intent about content but rather expresses a temporal constraint on the order in which roles get filled. Thus, of these, we only study case (3) in this thesis since our focus is on modeler intent about model content.

2.4.1 Mappings

In order to express a model relationship it is usually required that we assume the existence of a *mapping* between the elements of the models involved. A mapping, like a model, is an artifact that must occur in the project and we think of it as playing the role represented by the intended model relationship. For example, to say that the design model role *TransportationSystem* must refine *TransportationSystemArch* corresponds to the following intentions:

- There is an existential intent that a mapping (call it *SysArchMap*) exists in the project that maps the elements of the *TransportationSystem* to those of *TransportationSystemArch* that they refine, and,
- There is a constraint that *SysArchMap* must satisfy that ensures that this mapping is a valid refinement.

2.4.2 Relationship types

The fact that expressing modeler intent requires that role constraints be specified for each role means that this adds significant effort to the modeling activity. The use of role types that package sets of role constraints can help alleviate this. In a similar way, *relationship types* allow sets of constraints that are commonly expressed between roles to be packaged and reused. Furthermore, relationship types provide a level of abstraction on the constraints expressed at the role level. For example, *UMLrefines* is a relationship type that can hold between two UML models and it encapsulates a set of constraints that must hold between these models. Furthermore, it expresses the fact that one model *refines* the other and this carries a different meaning for the stakeholder than another relationship type between UML models that says that one model is a *submodel* of the other. Thus, relationship types are both an effort reduction mechanism and an abstraction mechanism and are practical necessities for expressing modeler intent.

2.5 Expressing intent about model decomposition

Until this point in our analysis, we have only considered modeler intent with respect to individual models or between particular models participating in a relationship. Another common situation is that the modeler decides that a model that is required (i.e., for which there is a model role) should be decomposed into a collection of interrelated models rather than being created as a single model. There are many reasons to do this. For example, in order to manage complexity, the model may be decomposed into smaller parts and into different levels of abstraction. A model may be decomposed because it is not renderable as-is and it must be split into diagrams that have well defined notations. This is the case with the UML – it has no single notation for the entire modeling language and so a UML model must always be decomposed into diagrams in order to be rendered. Another reason to decompose a model is to support some task – e.g., assigning the parts to different teams.

Thus, like a model, every collection representing a decomposition has a purpose. The constraints that characterize the resulting modeler intent about such a collection have three facets:

- The collection decomposes a model which must itself have a well defined purpose – i.e., it decomposes a model role.
- The collection must conform to the constraints imposed by the particular purpose of the decomposition of this model role.
- The model role must be decomposed according to a particular method, or *decomposition criterion*, that yields a collection of model roles as the constituents of the decomposition.

To understand decomposition criteria, first note that we have argued that no model exists without a purpose and so this must also be the case for the models that comprise the decomposition. It is the modeler's responsibility to define the purpose of each model within the decomposition and this gives rise to a model role and a corresponding existential intent for each constituent model.

A second consideration is that the purpose of the decomposition often underdetermines the decomposition since there may be many possible decompositions that can satisfy this purpose. Thus, the modeler's intentions regarding a decomposition are driven partly by the purpose of the decomposition and partly by their own design decisions on how to achieve this purpose. For example, assume that we have the model role *VehicleTypes* in the transportation system design. The purpose of this model is to show all the types of vehicles that are used in the transportation system. Assume that the modeler (as definer) decides that this model is too complex and must be decomposed. Consider the following two possible decompositions:

- D1 = {*LightVehicleTypes, MidrangeVehicleTypes, HeavyVehicleTypes*}
- D2 = {*PassengerVehicleTypes*, *CommercialVehicleTypes*, *ServiceVehicleTypes*}

D1 represents a decomposition of *VehicleTypes* on the basis of vehicle weight while *D2* is a decomposition on the basis of vehicle function. Both may satisfy the purpose of managing complexity and both define the purpose for each constituent model but each has a different basis for the decomposition. Thus the basis for the decomposition is also part of the modeler intent expressed as part of the decomposition criteria for the collection.

Earlier we made the distinction between model roles and models pointing out that model roles have an independent existence from models. We now extend this observation to collections and define collection roles. Specifically, a *collection role* is a collection of model roles with a collective purpose. Although we don't preclude the possibility that other kinds of collection roles may be possible, in this thesis, our focus is on collection roles that are decompositions of model roles.

Chapter 3

Related Work

We begin this review of related work by noting that while there is some work relating to various types of intention within software engineering, the specific issue of expressing modeler intent has not been a significant focus of academic research (as far as the author can determine). As a result, we first briefly review some non-model related work on intention and then for the remainder of the review we assess the ways in which different areas of existing modeling practice and research are inadequate as mechanisms for expressing modeler intent. To help guide the review, we refer to Table 3.1 which lists the

Table 3.1. Key requirements of an approach for expressing modeler intent.

Requirement
Must be able to distinguish model role, model and model type
Must be able to attach role constraints to and between model roles and be able to assess these against sets of models playing the roles.
Role constraints must be formalizable in order to support automation.
Must be able to define relationship types that package sets of constraints.
Must be able to express collections of models and relationships at the role (macroscopic) level of abstraction.
Must be able to express hierarchical structured role collections.
When a collection represents a decomposition of model, must be able to express the intent about how a model is decomposed.

set of requirements that any method or formalism used to express modeler intent must satisfy. This is distilled from the analysis in Chapter 2. We order the review roughly from the most relevant to least relevant and group them under the following categories: model metadata, method engineering, aspect-oriented modeling, model management and traceability.

3.1 Non-model related work on intention

In [MMW02], Mens et. al. proposes the notion of *intentional source-code views* to help with software maintenance. The idea is to declaratively define views on source code that extract certain subsets of code related to a particular interest or concern. They are considered to be intentional because the view definition defines the intent of the view. The benefit is that it eases code comprehension and analysis and allows required constraints to be enforced within the code by defining relationships between the views. Apart from being about code rather than models, this differs from our focus in that these views are not a part of the primary artifact (in this case, the code) but represent additional supporting artifacts that aid with code maintenance. In contrast, our concern is with the intent regarding the primary artifact produced by the modeler. Nevertheless, this work has similarity to the approach we take to expressing content criteria in Chapter 5.

The work of Yu and Mylopoulos on using the i* language to model the intentional attributes of agents (e.g., goals, commitments, etc.) and their interrelationships [YM97] has significant applications to software engineering. In Section 2.1.2 we discussed this work and pointed out that it was more appropriate for describing the model consumer’s intentions rather than the modeler's intentions. We assume that the goal of the modeler is always to produce a model that satisfies the goals of consumer and so their intent is best expressed as a translation of consumer goals into role constraints on the model content. Thus, while i* provides a non-formal way to model the various and open-ended intentional states of the consumer, our focus is on creating formal expressions of constraints on the model that operationalize these.

The *intent specifications* of Leveson [L00] are an attempt to make specifications more effective for human use by structuring them according to principles from systems theory, cognitive psychology and human-computer interaction. For example, an intent specification for software contains five levels of increasing detail in the intent (i.e., why) dimension: system purpose, system principles, black box behaviour, design representation and code (physical representation). While this work is distant from our concerns in this thesis, the fact that it pays attention to the human factors aspects of specifications is relevant to modeler intent. In particular, the model consumer's effectiveness in comprehending a model or collection of models is affected by how models are presented and the expression of modeler intent. We have discussed some ways in which modeler intent affects comprehension in the introduction and Chapter 2 and this is elaborated further in subsequent chapters.

Intentional Software is the work of Simonyi et. al. [SCC06] and represents an approach to programming that raises the level of abstraction from programming languages to the language in which domain experts express their ideas. The result is that software more closely resembles what the domain expert intends rather than some encoding of it by programmers. This relates to the focus of this thesis in the sense that we assume that modeler intent is a translation from the consumer's intent into a formal language expressing role constraints. Applying the intentional software philosophy here would suggest that a model consumer ought to be able to express their model requirements directly using concepts from their domains of expertise. Although this is an interesting idea, it is outside the scope of the current thesis and we do not explore it further.

3.2 Model metadata

Within a software development project, models are typically treated as a type of document and are managed within a file system or source control system (e.g., Subversion [SubV]) along with their associated metadata. The role that a model plays and the associated modeler intent that defines this role appears to be a kind of model metadata. However, as we discussed in Chapter 2, a model role has an existence that is independent of and could precede the existence of models that play it. Thus, it does not quite fit the criteria of being metadata – i.e., we don't typically allow metadata values to exist without the object that they describe. Nevertheless, we consider how well existing types of metadata can be used to express modeler intent.

The National Information Standards Organization (NISO) identifies three main types of metadata for information resources [GR04]:

- *Descriptive metadata* describes a resource for purposes such as discovery and identification. It can include elements such as title, abstract, author, and keywords.

- *Structural metadata* indicates how compound objects are put together, for example, how pages are ordered to form chapters.
- *Administrative metadata* provides information to help manage a resource, such as when and how it was created, file type and other technical information, and who can access it.

Descriptive metadata is a key mechanism for expressing modeler intent. The Dublin Core standard [DC] is widely adopted as a reference model for descriptive metadata used in resource descriptions. Table 3.2 excerpts the subset of metadata elements that could be used to express aspects of modeler intent about content. The descriptive metadata that is actually associated with a model varies according to management system used. File and source control systems typically support only *Title* (i.e., the model name) and possibly, *Description*. Note that important information concerning the model role such as relationships to other models (*Relation, Source*) and information concerning *aboutness* (*Coverage*, *Subject*) is typically either missing or embedded in the model name.

The Dublin Core standard also contains a *Language* metadata element to identify the language of the resource. In the context of models, this is a reference to the model type of a model. As discussed in Chapter 2, the model type is related to model purpose in that it limits the possible purposes a model may have but does not specify purpose. Structural metadata is also an important aspect of modeler intent. Since we normally consider models to be monolithic instances of their metamodels, we assume that structural metadata applies only to collections of interrelated models. In this case, the expression of modeler's intent describes how the collection is structured.

The Dublin Core standard also contains a *Language* metadata element to identify the language of the resource. In the context of models, this is a reference to the model type of a model. As discussed in Chapter 2, the model type is related to model purpose in that it limits the possible purposes a model may have but does not specify purpose. Structural metadata is also an important aspect of modeler intent. Since we normally consider models to be monolithic instances of their metamodels, we assume that structural metadata applies only to collections of interrelated models. In this case, the expression of modeler's intent describes how the collection is structured.

The position taken in this thesis is that not only is the descriptive and structural metadata required for expressing modeler intent, but that this is inadequate unless these are

Table 3.2. The subset of Dublin Core metadata elements relevant to modeler intent.

Metadata element	Definition
Title	A name given to the resource.
Coverage	The spatial or temporal topic of the resource, the spatial applicability of the resource, or the jurisdiction under which the resource is relevant.
Description	An account of the resource.
Relation	A related resource.
Source	A related resource from which the described resource is derived.
Subject	The topic of the resource.

formalized. Unlike the content of a model, which is structured according to its metamodel and is subject to formal constraints and potentially, formal semantics, the content of the metadata are typically expressed in natural language and have neither structure nor formal semantics. The lack of structure prevents the automated enforcement of consistency in how modeler intent is expressed (e.g., varying naming standards). The lack of formality implies that there is no automated way to verify that the model content correctly corresponds to the modeler's intent about that content. This impacts model quality because stakeholders use this metadata to obtain information about model content at a high level of abstraction and if this is misunderstood or is incorrect then it negatively impacts the efficacy of model access activities (e.g., search) and model comprehension.

3.3 Method engineering

One of the earliest drivers of software engineering research was the desire to model and automate the software development process and the construction of the artifacts produced by it. As discussed in Chapter 2, software development methods identify specific roles played by models in a development project, thus it is relevant for our review. Telos [MBJK90] was an influential early approach for capturing information about the development process and defined a rich approach to metamodeling; however, Telos did not formalize the notion of collections of distinct interrelated models. By the mid 1990's a number of approaches had emerged that encompassed multiple model types and their interrelationships in the context of *method engineering* – the integration of different modeling languages with guidance in order to define new software development processes. A key contribution here out of the Requirements Engineering community is the ViewPoints framework [NKF94] which is the earliest approach we will review and the most closely aligned with goal of expressing modeler intent. More recently, the focus of these efforts have shifted toward the use of modeling frameworks that allow specialized development methods to be defined using Domain Specific Modeling Languages (DSML). Here the idea is to have a “metatool” in the sense that it allows a modeling environment for particular sets of DSML’s to be defined in terms of their metamodels. We briefly review four significant efforts in this direction: the Eclipse Graphical Modeling Framework (GMF) [GMF], the Generic Modeling Environment (GME) [BCR05], MetaEdit+ [MEdit] and Microsoft’s Software Factories [GS04]. Finally we consider the Software Process Engineering Metamodel (SPEM) [SPEM] - an initiative of the Object Management Group (OMG). Here the focus is on providing a high level definition of a method without defining the details of the artifacts involved.

3.3.1 ViewPoints

The ViewPoints framework [NKF94] was an influential early approach to relating heterogeneous collections of models in a metamodel language independent way. We examine this approach in greater detail since of all the approaches reviewed it most directly concerns itself with expressing modeler intent. The motivating idea is that developer’s viewpoints of systems are typically partial, heterogeneous and decentralized and hence there is a need to express the relationships between viewpoints in order to support integration. Within the framework, each *ViewPoint* has an owner and consists five parts:

- *Style* – representation scheme used to express the ViewPoint
- *Domain* – part of the world that the ViewPoint is representing
- *Specification* – the statements in the representation scheme describing the domain

- *Work Plan* – how and when the specification can be changed
- *Work Record* – describes the current state of the specification

Thus, a ViewPoint is an “envelope” that contains a model (*Specification*) and other metadata relevant to the development process. The semantics of a ViewPoint are limited to a statement of the *Domain* to which it applies. Although the ViewPoints framework does not use the terminology of metamodels, their elements are presented in the form given by *Style* defining the model element types and the *Work Plan* containing the well-formedness constraints. Nevertheless, no particular form of metamodel is mandated. The *Work Plan* also has other functions - it consists of the following:

- *Assembly Actions* – the actual primitive "edit" operations provided by a tool used to construct a specification in the given style.
- *In-ViewPoint Rules* – the well-formedness constraints.
- *Inter-ViewPoint Rules* – a set of rules that check the consistency between different related ViewPoints.

Thus, the relationships between ViewPoints are expressed as *Inter-ViewPoint Rules* and for a given source ViewPoint VP_S of a given type, this can take two forms:

$\{ps_S\} \Rightarrow \exists VP_D(t,d)\{ps_2\}$ (existence relationship)

$\forall VP_D(t,d).\ \{ps_S\ R\ VP_D{:}ps_D\}$ (agreement relationship)

The existence relationship says that if VP_S has a particular partial specification ps_S, there must exist a destination VP_D of template type t and domain d. The agreement relationship says that the relation *R* must hold between the specified partial specifications within VP_S and all the destination ViewPoints of template type t and domain d. These forms could be combined when appropriate.

A *ViewPoint template* is a way to characterize a *type* of ViewPoint and has only the *Style* and *Work Plan* slots. Thus, it extracts a metamodel along with its relationships to other metamodels. A development method can be defined as a collection of related ViewPoint templates. The *Work Plan* is used to automate the linkages between ViewPoints and thus provide the dynamics of a development method. *Inter-ViewPoint Rules* can be used in *Check Mode* to check whether the expressed relation holds and therefore check consistency. When the relation expressed is actually a function, it can also be used in *Transfer Mode* to enforce the relationship by generating the target ViewPoint (and/or some of its specification) from the source ViewPoint. Thus, in this mode, it acts as a ViewPoint transformation. For example, the creation of a class in a class diagram may cause the creation of a corresponding empty state machine diagram via an existence rule and then agreement rules may be used to partially populate it.

Another major thrust of the ViewPoints research was to determine how to deal with cases when *Inter-ViewPoint* rules fail – i.e., the case of ViewPoint inconsistency and what support could be provided to manage and resolve inconsistencies. Easterbrook et. al. explore this issue and conclude that support should take the form of offering the modeler a set of predefined resolution actions from which they can select and apply [EFKN92].

A ViewPoint is similar to model role in that it carries metadata about a model including constraints to other models. However, the ViewPoints framework uses this for managing decentralized development rather than for characterizing model purpose. Of all the approaches reviewed, the ViewPoints framework is the only one that provides a mechanism for expressing constraints at the project level rather than only at the method level. As a result it is capable of expressing role constraints and modelers in a distributed environment can use this to express how they intend their model to be related to other models. In addition, these can be expressed at the method level between ViewPoint templates to support reuse. Despite this, role constraints are not clearly distinguished from type constraints and thus there is no systematic approach to managing the differences between violations to these kinds of constraints.

A key weakness of the ViewPoint approach is that it only allows constraints to be expressed within ViewPoints and does not identify relationships as first class entities with their own type structure. In addition, it does not provide a mechanism for creating hierarchical collections of models or for the expression of the modeler intent concerning these. Finally, it does not take a sufficiently formal approach to the problem. Perhaps this is because, in order to support the goal of heterogeneity, no specific language is mandated for expressing the *In-ViewPoint* and *Inter-ViewPoint* rules; however, in more recent work [NKF03], the authors admit that it is very difficult do meaningful reasoning or analysis if a common language is not used for expressing relationships; thus, formal languages like first order logic are suggested.

3.3.2 Modeling frameworks

GMF, GME, MetaEdit+ and Software Factories all allow the use of metamodels for defining model types and provide a specific metamodeling language for this purpose. Constraints can be expressed using Object Constraint Language (OCL) [OCL] in the case of GMF and GME while MetaEdit+ and Software Factories provide their own proprietary constraint language. Although there is no concept of model role and constraints are expressed only as part of the metamodel, the orientation toward DSML's means that type constraints begin to act like role constraints. The idea here is to define model types that are highly specialized for particular roles and therefore the constraints can encode details that are specific to these roles.

For example, rather than using a general purpose model type like a *Statechart* to describe the behaviour of digital watch KX25, a *DigitalWatchDiagram* could be used that provides a special notation for describing the different watch modes, the different display configurations, etc. A semantic constraint from the digital watch domain may specify that any change operation to an "hour" counter must be followed by a change operation to a "minute" counter. This type constraint may be equivalent to a role constraint used to express a part of the intent of using a *Statechart* for modeling digital watch behaviour. The drawback of this approach is that it leads to a proliferation of similar model types. Furthermore, it still only supports role constraints at the method level and not at the project level.

GME and MetaEdit+ have built-in support for a fixed set of model relationship types. For example, both of these have an "expand" type relationship that allows an element in one model to expand to other models that provide details of the element. Both GME and Software Factories allow the definition of transformations between different model types. In fact, Software Factories further classifies transformations as *Specialization*, *Elaboration*, *Derivation*, *Refactoring*, etc. and thus provides some direct support for expressing intent at the macroscopic level. However, none of these approaches provide general support for expressing non-transformation model relationships or their types. In addition, there is no special support for identifying hierarchical collections of related models apart from the implicit hierarchy formed by expanding elements to models within GME and MetaEdit+.

Of the four approaches, only Software Factories provides a mechanism for defining a model collection type that separates the model types from their relationship types. The Software Factory Schema consists of a set of model types and transformations that relate them. The other approaches place all model types together as part of one large metamodel in which model types can be related by sharing element types or by having references between their element types.

In general, a key weakness in all of these approaches is that they don't distinguish between models and model roles. As a result, within a particular project, these approaches only provide support for the model level and miss the macroscopic abstraction that a role level would provide. Limited macroscopic capability is available through collapse/expand of models into a single model "nodes" but this is a model navigation aid rather than a formal level of specification.

3.3.3 Standard Process Engineering Metamodel (SPEM)

The SPEM process modeling approach aims to characterize a software development process using a model that describes how development activities lead to the creation of different related artifacts (such as models). The metamodel includes the concept of a *work product* – the artifacts that act as inputs and outputs of tasks that comprise a development method. The work product as defined here is similar to what we call an role type – it represents an artifact that has a well defined purpose within the development method and is created and consumed by certain tasks. For example a “Create High-Level Design” task may consume the “Requirements” work product and produce the “High-Level Design” work product. The details of the structure and lifecycle of a work product can be expressed in a work product definition. Additionally, work product relationships can be expressed to show process dependencies between work products.

With the notion of work product, SPEM effectively captures the distinction between role type and model type; however, it is still limited to use at the method level rather than at the project level. Furthermore, although work product relationships can be used to express relationships between role types, these are limited to process sequencing constraints rather than constraints on the content of models playing roles.

3.4 Aspect oriented modeling

Aspect Oriented Modeling (AOM) has a close relationship to the notion of modeler intent. Here the idea is to provide a means for separately maintaining and developing *aspects* - subsets of the information in a model relating to particular *concerns* such as security or customization - and then allowing these to be woven together to produce the complete model when necessary. A concern is related to an aspect similarly to the way a purpose is related to a model role. In this sense, AOM seems to capture an essential part of what we mean by modeler intent. A wide variety of approaches for AOM have been proposed [SSR06] and thus we focus our review at the level of key concepts rather than specific approaches.

AOM has emerged out of Aspect Oriented Programming (AOP) and has taken similar approaches. The seminal work on aspect orientation comes from researchers at Xerox PARC who proposed the concept of AOP [KLMM97]; however, another highly influential strand of aspect research is known as multi-dimensional separation of concerns (MDSOC) [TOH99]. The first approach is called *asymmetric* and is based on the principle that a model can be factored into a base model and a number of aspects that contain fragments of concern-specific content called *advice*. An *aspect weaver* is program that is then responsible for merging the aspect content with the base content at the appropriate places called *join points* as specified by *pointcuts* in the aspect. The second approach is called *symmetric* and provides a general weaving mechanism that can intelligently merge a set of aspects to form a complete model. Of these two approaches, the advice/join point approach has emerged as the dominant one.

As discussed above, a concern is related to an aspect similarly to the way a purpose is related to a model role. For example, the aspect of a model that deals with the concern of *security* could represent a model role with the purpose of “showing the security related information.” However, although the spirit is similar, the objectives are quite different. The motivation of AOM is to provide techniques for separating content relating to different concerns in a manageable way in order to facilitate their subsequent recombination into a single model. Thus, AOM is fundamentally a model maintenance technique. In contrast, our goal is to articulate the intent of models in order to improve model comprehension, quality and to support automation.

These differences in objectives lead to important differences in the details. Not every model role corresponds to an aspect because an aspect always represents a subset of the information in a model but a model role need not represent a subset of a model. Relationships are supported between aspects (symmetric AOM) or between the aspect and the base (asymmetric AOM) but these are restricted to mappings that support the weaving process – the relationships do not carry semantic content.

Consider also that an aspect is neither really a model nor a representation of a model role – it is actually a set of model fragments annotated with the information required to weave it correctly with other aspects. In a sense, the role constraints (weaving information) are mixed in with the model content (model fragments). This is reasonable because AOM focuses on expressing aspects, not on expressing concerns. An aspect does not try to characterize what information should be in the aspect and what shouldn't. Thus, there is no strong incentive to clearly separate model roles from models and formally express the purpose of the model. In contrast, we are primarily interested in expressing concerns (i.e., purpose).

3.5 Model management

The field of *model management* deals with generic modeling frameworks that use an algebraic approach for raising the level of abstraction in modeling. These provide generic operators that can be used to manipulate models and their relationships (here called a “mapping”) in order to solve various modeling problems. The notion of model management was first proposed by Bernstein [B03] as an approach to address database metadata management problems such as schema integration and instance translation. Since then, it has been extended to apply to modeling problems in software engineering as well. For example, in [BCE06], the authors define a set of operators that are useful in software engineering. It is relevant to our concerns because of this focus on a macroscopic level of abstraction of models and their relationships. We briefly describe five model management frameworks and then assess them collectively in terms of their applicability for expressing modeler intent.

Sergey Melnik developed the Rondo system that realized Bernstein’s vision [Mel04]. In Rondo, models (i.e., schemas) are represented as directed labelled graphs that contain both the model and metamodel elements. A mapping between two models is called a *morphism* and is simply a binary relation between the nodes of the models. Rondo defines a small set of primitive operators such as *Domain*(*map*) that returns the domain model of a morphism, *Compose*(*map*, *map*) which composes morphisms, etc. These are then used to construct more complex operators using scripts.

In another approach, Diskin [D05] describes a complete formal framework for generic model management based on Category Theory. The category theoretic notion of a *sketch* is used as a “universal” syntax for models. A sketch is a directed labelled graph with some additional annotations representing constraints (called “graph predicates”). The kind of mapping used here does not just utilize the explicit information in a model but also the “implicit” information that could be derived from the model – thus, this can be used to help relate models without changing the net information content. Since these mappings are formally defined and form categories, they allow certain generic model manipulation operators (e.g., merge, diff, etc.) to be defined as standard category theoretic constructions. A useful category theoretic technique here is the “diagram” that allow collections of models and mappings to be expressed visually as a directed graph. Category theoretic approaches have also been applied elsewhere to define specific operators. For example, Sabetzadeh and Easterbrook define viewpoint merging operators [SE05].

The MoMENT framework of Boronat, Carsi and Ramos [BCR05] is a software model management infrastructure based on the algebraic specification. It is positioned as an approach to MDE and borrows both from Bernstein & Melnik’s work by implementing their model management operators and from MDA (Model Driven Architecture) [MDA] by implementing QVT [QVT05] and OCL. At the high-level, metamodels are expressed using Ecore and mappings are expressed as QVT mappings. At the low level, metamodels are translated to algebraic specifications expressed in the Maude language and models are translated to terms in the algebra specified by the metamodel. QVT mapping are also translated to algebraic specifications.

The Epsilon software model management framework of Kolovos, Paige and Pollack [KPP06] is integrated with the Eclipse platform and consists of an evolving set of task-specific languages with supporting tools to help implement model management operators.

The motivating principle is that each language captures the patterns required for efficiently implementing a particular type of model management operation. For example, the merging language EML is a rule-based language with a matching phase that establishes correspondences between models being merged and a merging phase for combining these elements into the merged model. Epsilon provides support for two types of mappings. "Links" (mappings) between models can be expressed by using an intermediate model whose metamodel imports the metamodels of the mapped models. Another way to define mappings specifically between metamodels is to use the Epsilon Transformation Language (ETL). A mapping consists of a set of rules for transforming instances of the source metamodels into instances of the target metamodels.

The focus of the Atlas Model Management Architecture (AMMA) [BJRV05] is to facilitate large-scale MDE by providing a set of tools intended to support "global model management." Metamodels are defined using a variant on the Ecore metametamodel. Mappings can be established between metamodels using the Atlas Transformation Language (ATL) to define model transformations. Mappings between models can be expressed using a *Weaving Model*. Each weaving model is an instance of a weaving metamodel that extends the core weaving metamodel provided by AMMA. A *MegaModel* is a special kind of model whose elements are references to resources used in an MDE environment such as models, metamodels, transformations, editors, etc. The intent is to use megamodels in a variety of ways from facilitating the integration of different MDE environments to providing an interface for invocation of tools (e.g., transformations) on models.

As is evident from the above, the focus of model management is on model manipulation rather than on model presentation. As such, it does not directly fit our scenario for expressing modeler intent. Among all the frameworks, AMMA is the one that comes closest to meeting our requirements for expressing modeler intent. All of the frameworks allow new types of transformations to be defined but only AMMA and Epsilon allow the definition of multiple non-transformation relationship types - the others rely on a single type of generic mapping to relate models. AMMA and Diskin's framework allow for representing models and their relationships at the macroscopic level using MegaModels and Diagrams, respectively, so in a sense they can express information at the role level rather than just at the model level; however, the role constraints expressible are limited to the available types of relationships. Diagrams can only be used for flat collections but MegaModels are extensible to arbitrarily structured collections since they have their own user-definable metamodel.

3.6 Traceability

The work on model traceability originally emerged within requirements engineering research out of the need to draw connections between software requirements and the artifacts that need to satisfy the requirements. Since that time the scope has broadened considerably. In an overview on model traceability [ANR06], the authors state "Traceability relationships help stakeholders understand the many associations and dependencies that exist among software artifacts created during a software development project." Standard relationship types have been proposed for relating the elements of artifacts. For example [RJ01] suggests types like *depends on*, *justifies*, *validates*, *describes*, etc.

Although both traceability information and expressions of modeler intent are useful for comprehending the relationships between artifacts, the motivation and results are quite different. Traceability relationships are expressed at the model level between the individual elements of related models. This allows stakeholders to trace elements in one model to related elements in other models. In contrast, modeler intent is expressed at the role level as constraints that the models that play related roles must satisfy. Traceability information shows “what is” while modeler intent shows “what must be.” Finally, modeler intent includes intended constraints on individual models (i.e., unary relationships) as well as the intended structure of collections of models. Thus, modeler intent has a wider scope than traceability relationships. These differences suggest that traceability information and expressions of modeler intent are complementary rather than competing approaches for supporting the comprehension of collections related models.

3.7 Summary

In this chapter we have reviewed the work in software engineering that could be considered to be related to expressing modeler intent. Our main finding is that there is no existing work that directly addresses this issue. Nevertheless, although it is not their primary objective, a number of approaches stand out as being partially a “fit” to the problem of expressing modeler intent. We summarize these as follows:

- Model metadata standards [GR04],[DC]. Modeler intent is often expressed informally within model metadata such as names and descriptions. This approach is inadequate for our purposes because it lacks formality and underlying principles that define the basis for correctness or completeness. Furthermore, there is an ontological difference between metadata and model roles: the existence of model metadata depends on the existence of a model but a model role can precede the existence of any model playing it.
- ViewPoints framework [NFK94]. ViewPoints are like model roles in that they “wrap” a model with additional information including constraints with other ViewPoints. Despite this similarity, ViewPoints are for managing decentralized development rather than for characterizing model purpose. Furthermore they have no notion of relationship types, hierarchy, decomposition or special handling of role constraints.
- Software Process Engineering Metamodel [SPEM]. SPEM allows model roles to be defined as part of a method definition along with process sequencing constraints on the roles. Unfortunately, it does not support role constraints on content and this is required for defining modeler intent. Furthermore, roles can only be defined at the method level and not the project level.
- Atlas Model Management Architecture [BJRV05]. This modeling infrastructure supports the definition of relationship types and has a special type of model called a MegaModel that can be used to express configurations of models with associated metadata. Although it does not have special support for concepts such as decomposition, hierarchy and role constraints, the generic metamodel-driven

> nature of it suggests that it could be customized to support this. Despite this, the focus of MegaModels is on integrating modeling environments and so it is descriptive, expressing “what it is” in contrast to modeler intent which says “what must be.”

We now turn to the description of our approach to expressing modeler intent. We begin with an exposition of how model and relationship types can be formally defined.

Chapter 4

Models, Relationships and Relationship Types

In Chapter 2 we distinguished the model type from the model role saying that the model type was primarily responsible for the syntactic and semantic aspects of a model while the model role concerned the pragmatic aspects. Furthermore, we argued that model relationships express role constraints between model roles and that classification of model relationships into relationship types provides a way to abstract and reuse role constraints. In this chapter we focus on model types, relationships and relationship types and formalize our use of these notions. The main contributions are as follows:

- We define an approach for defining relationship types using metamodels but we do so in a metamodeling language independent way.
- We characterize semantic and syntactic properties of relationship types that are of relevant to the expressions of modeler intent described in subsequent chapters.
- We propose a taxonomy of abstract relationship types and define some of these formally. In particular the *submodelOf* relationship type is elaborated in detail.

4.1 Notational conventions

We define the notational conventions that will be used in this chapter and for the remainder of this thesis. Names of model role constants will be given in mixed case beginning with an upper case letter. For example, *DTollPrice*, *TransportationSystem*, *M*, M_1, *M1, etc*. all represent model roles. Names of model constants will be as model roles but underlined. This allows us to indicate a model as well as the role it plays. Thus, the model <u>*DTollPrice*</u> plays the role *DTollPrice*. When we wish to show multiple models that can play the same role (but at different times), we will use superscripts as in $\underline{DTollPrice}^1$, $\underline{DTollPrice}^2$, …. The names of variables representing model roles or models will follow the same conventions as for constants except that they begin with a lower case letter. Thus, *base, m*, m_1, *m1* all represent variables that take model roles as values. Correspondingly, <u>*base*</u>, <u>*m*</u>, $\underline{m_1}$, <u>*m1*</u> are model variables.

Names of model types are strings of upper case letters such as *CD*, *SD*, *UML*, etc. In order to avoid confusion with model roles, these will be indicated explicitly to be types when it is not clear from the context of usage. Models can be typed and this is indicated using the prefix ": *T*" where *T* is a type name. Examples: <u>*DTollPrice*</u>:*CD*, $\underline{M_1}$:*UML*, etc. When the typing convention is used with a model role it indicates the type of model that should play the role. Thus, *DTollPrice*:*CD* is a model role constrained to be played by a model of type *CD*.

4.2 Models

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 [G98], in this thesis, 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.

4.2.1 Metamodels and model types

A metamodel is commonly used to define 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 defines as [[*MM*]]. For example, we may have the metamodel *SD* where [[*SD*]] is the set of all sequence diagrams. Note that the same model type could be defined by alternative metamodels. For example, if we use the OMG metamodeling language MOF [MOF06] and the Eclipse metamodeling language Ecore [EMF], we may have $SD{:}MOF \neq SD{:}Ecore$ but [[*SD*:*MOF*]] = [[*SD*:*Ecore*]] = [[*SD*]]. 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*[4]. If a model type is definable using a metamodel we will say it is *concrete*, otherwise it is *abstract*.

If <u>*M*</u> is a particular sequence diagram, then it is an instance of the model type [[*SD*]] but we will extend this idea to its metamodel(s) and also say that <u>*M*</u> is an instance of *SD*. That is, a model is

[4] 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 and be finite.

considered both to be an instance of its model type and of any metamodel that specifies the model type. Following usual conventions for typing we will write this as $\underline{M}:SD$.

4.2.2 Metamodeling language

In this thesis, rather than using a practice-oriented metamodeling language such as MOF or Ecore, we have chosen to use order-sorted first order logic with transitive closure[5] (henceforth referred to as FO+) as the metamodeling formalism. There are a number of reasons for this. First, variants on first order logic are widely known and have comparable expressive power to the other metamodeling approaches. Second, 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 $\langle \Sigma, \Psi \rangle$ where Σ is called the *signature* and defines the types of model elements and how they can be related, while Ψ is a set of constraints[6] that define the well-formedness rules (i.e., type constraints) for models. Thus, a metamodel $\langle \Sigma, \Psi \rangle$ 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. Note that not all FO+ theories have finite models but since a metamodel is only useful if it admits finite models we will consider it to be inconsistent if it has none. Thus, the metamodel consisting of a single sort S and a function $f{:}S \rightarrow S$ with the constraint that S is non-

[5] For a more detailed description of order sorted logic please see [GM92].

[6] Although we are talking about logical theories we will use the term "constraints" rather than "axiom" because it is more in keeping with the concerns of the thesis.

empty and *f* is injective but not surjective, is a consistent FO+ theory but an inconsistent metamodel because it has infinite models but no finite ones.

For example, we can define the abstract syntax of (simplified) UML class diagrams in as follows[7].

```
CD =
   sorts  class, assoc, attr, operation
   func   startClass: assoc → class
           endClass: assoc → class
           attrClass: attr → class
           opClass: operation → class
   pred   subClassOf: class × class
   constraints
          // a class cannot be a subclass of itself
          ∀c:class · ¬TC(subClassOf(c, c))
```

The signature Σ_{CD} consists of the triple $\langle sorts_{CD}, func_{CD}, pred_{CD}\rangle$ where $sorts_{CD}$ is the set of element types that can occur in a model while $func_{CD}$ is the set of functions and $pred_{CD}$ is the set of predicates used to relate the elements. We will say $\Sigma_{T1} \subseteq \Sigma_T$ to mean that $sorts_{T1} \subseteq sorts_T$ and $pred_{T1} \subseteq pred_T$. The **constraints** section describes Ψ_{CD}. Note that the quantifier $\exists!$ means "there exists one and only one" and the operator *TC* takes a predicate and produces its transitive closure.

[7] We are using the style of algebraic specification here.

For simple cases we can also show a metamodel signature diagrammatically as in Figure 4.1. The nodes represent sorts, edges like “ ——►” denote directed binary relations and edges like “⟶” denote functions with one argument.

Figure 4.2 shows a class diagram based on the transportation system example both using its concrete syntax and its abstract syntax as an instance of *CD* (shown as a UML object diagram).

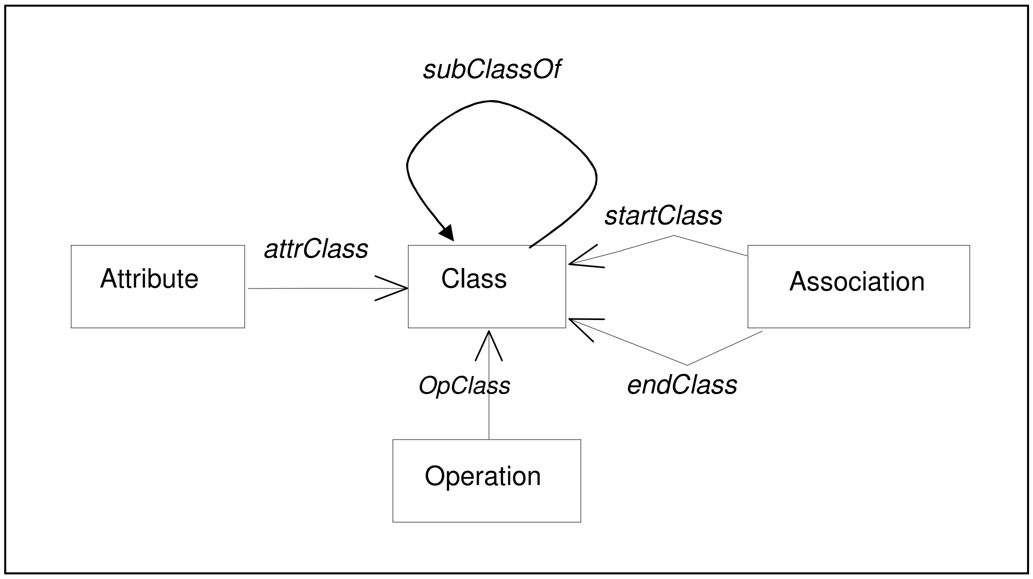
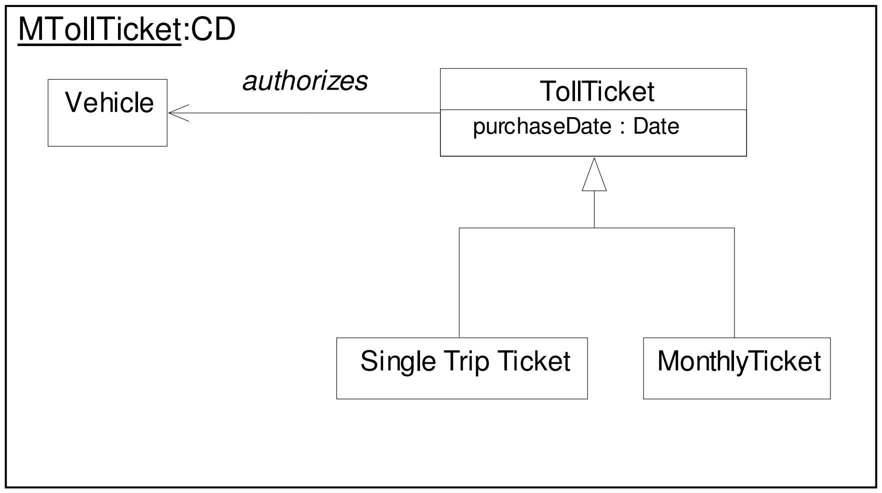


Figure 4.1. A graphical expression of the signature for *CD*.



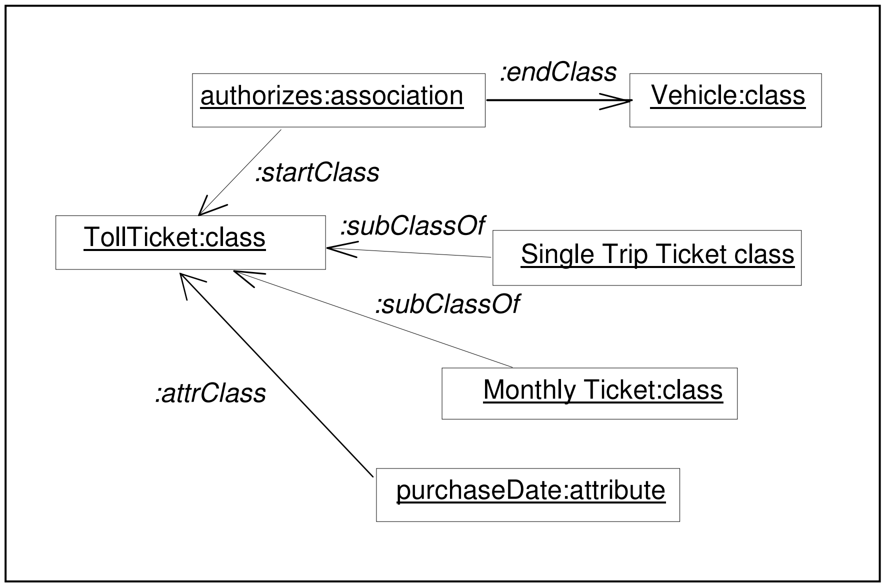


Figure 4.2. A class diagram and its abstract syntax as an instance of *CD*.

4.2.3 Formalization

We briefly review the formalization of FO+ (see [GM92] for more details). 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 $\leq \subseteq S \times S$ giving the order relation over sorts.
- 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 $Sen(\Sigma)$ to be the usual set of first order sentences (plus TC) over the signature Σ. Thus, for a metamodel $\langle\Sigma, \Psi\rangle$, we have that $\Psi \subseteq Sen(\Sigma)$.

In order to define the semantics of a metamodel[8] 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 $[[S_i]]_\tau \subseteq [[S_j]]_\tau$

[8] This should not to be confused with the semantics of the models described by the metamodel.

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

4.2.4 Constructed model types

Some abstract model types can be concretized in a generic way by defining a type constructor – we call these *constructed* model types. A simple but useful constructed model type is *Set*[*T*] that produces model types consisting of a single sort *T* and that have sets of *T* elements as models. This constructor concretizes the abstract model type *Set* consisting of sets of elements of the same type. The definition is:

```
Set[T] =
   sort T
```

For example *Set*[*Class*] is the model type consisting of sets of *Class* elements, *Set*[*Usecase*] is the model type consisting of sets of *Usecase* elements, etc. A useful variant on *Set*[*T*] is *One*[*T*] that defines model types having singleton *T* sets as models. The definition is:

```
One[T] =
  sort T
  constraints
 // there is exactly one element
 $\forall x_1, x_2{:}\mathrm{T} \cdot x_1 = x_2$
 $\exists x{:}\mathrm{T}$
```

We will see in subsequent chapters that *One*[*T*] is useful because it provides a way to lift an element type to the model level and hence allows it to participate in model relationships.

4.2.5 Metamodel morphisms and reducts

In Section 4.3, our goal will be to define model relationship types in terms of metamodels. In order to do this we require a way to show how metamodels can be related and what this means for the model types they specify. For this, we draw on a basic concept from theory of Institutions within algebraic specification [GB92]. The idea here is that since both the sentences *Sen*(Σ) and interpretations *Mod*(Σ) are completely determined by the signature Σ, if we map one signature into another, this will 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 h:$\Sigma_1 \rightarrow \Sigma_2$ 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, h consists of functions $\langle h_S:S_1 \rightarrow S_2,\ h_F:F_1 \rightarrow F_2,\ h_P:P_1 \rightarrow P_2\rangle$ such that the following conditions hold:

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

Intuitively, a signature morphism h:$\Sigma_1 \rightarrow \Sigma_2$ 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 $Sen(h)$:$Sen(\Sigma_1) \rightarrow Sen(\Sigma_2)$ for sentences that replace the sort, function and predicate symbols in each sentence by their image under h.

Given metamodels $\langle \Sigma_1, \Psi_1\rangle$ and $\langle \Sigma_2, \Psi_2\rangle$, a *metamodel morphism* h: $\langle \Sigma_1, \Psi_1\rangle \rightarrow \langle \Sigma_2, \Psi_2\rangle$ consists of a signature morphism h_Σ:$\Sigma_1 \rightarrow \Sigma_2$ such that $\Psi_2 \vDash Sen(h_\Sigma)(\Psi_1)$. Intuitively, this means that the axioms of $\langle \Sigma_1, \Psi_1\rangle$ follow logically from the axioms of $\langle \Sigma_2, \Psi_2\rangle$ after the signature renaming defined by h_Σ.

In the same way that h_Σ induces the sentence translation function $Sen(h_\Sigma)$:$Sen(\Sigma_1) \rightarrow Sen(\Sigma_2)$ it also induces the *reduct* function $Mod(h_\Sigma)$:$Mod(\Sigma_2, \Psi_2) \rightarrow Mod(\Sigma_1, \Psi_1)$ that uses h_Σ to transform each model of $\langle \Sigma_2, \Psi_2 \rangle$ into a corresponding model of $\langle \Sigma_1, \Psi_1 \rangle$. For each $\tau 2 \in Mod(\Sigma_2, \Psi_2)$, we define $\tau 1 = \mathrm{Mod}(h_\Sigma)(\tau 2)$ as follows:

1. $\forall S \in S_1, [[S]]_{\tau 1} = [[h_\Sigma(S)]]_{\tau 2}$
2. $\forall f \in \mathrm{F}_1, [[f]]_{\tau 1} = [[h_\Sigma(f)]]_{\tau 2}$
3. $\forall p \in \mathrm{P}_1, [[p]]_{\tau 1} = [[h_\Sigma(p)]]_{\tau 2}$

Thus, $\tau 1$ is formed from $\tau 2$ by discarding any sorts, functions and predicates not found in the image of h_Σ and duplicating those that multiple elements of Σ_1 map to. This is why $Mod(h_\Sigma)$ could be considered to be a projection function that extracts a $\langle \Sigma_1, \Psi_1 \rangle$ model embedded within each $\langle \Sigma_2, \Psi_2 \rangle$ model. For notational simplicity, we will henceforth drop the signature morphism indicator and just write *Sen*(h) and *Mod*(h) for metamodel morphism h.

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

$$\forall M \in Mod(\Sigma_2, \Psi_2), \varphi \in Sen(\Sigma_1, \Psi_1).\ Mod(h)(M) \models_1 \varphi \Leftrightarrow M \models_2 Sen(h)(\varphi)$$

We do not prove this here but direct the reader to [AKK99] in which proofs are provided for a wide range of similar logics.

Figure 4.3 illustrates the notions of metamodel morphisms, sentence translations and reducts. Here, we relate the metamodels *Taxonomy* = $\langle \Sigma_{\text{Taxonomy}}, \Psi_{\text{Taxonomy}} \rangle$ and *CD* = $\langle \Sigma_{\text{CD}}, \Psi_{\text{CD}} \rangle$. The metamodel morphism h:*Taxonomy* $\rightarrow$ *CD* has h_Σ which maps the sort *Category* to the sort *Class*

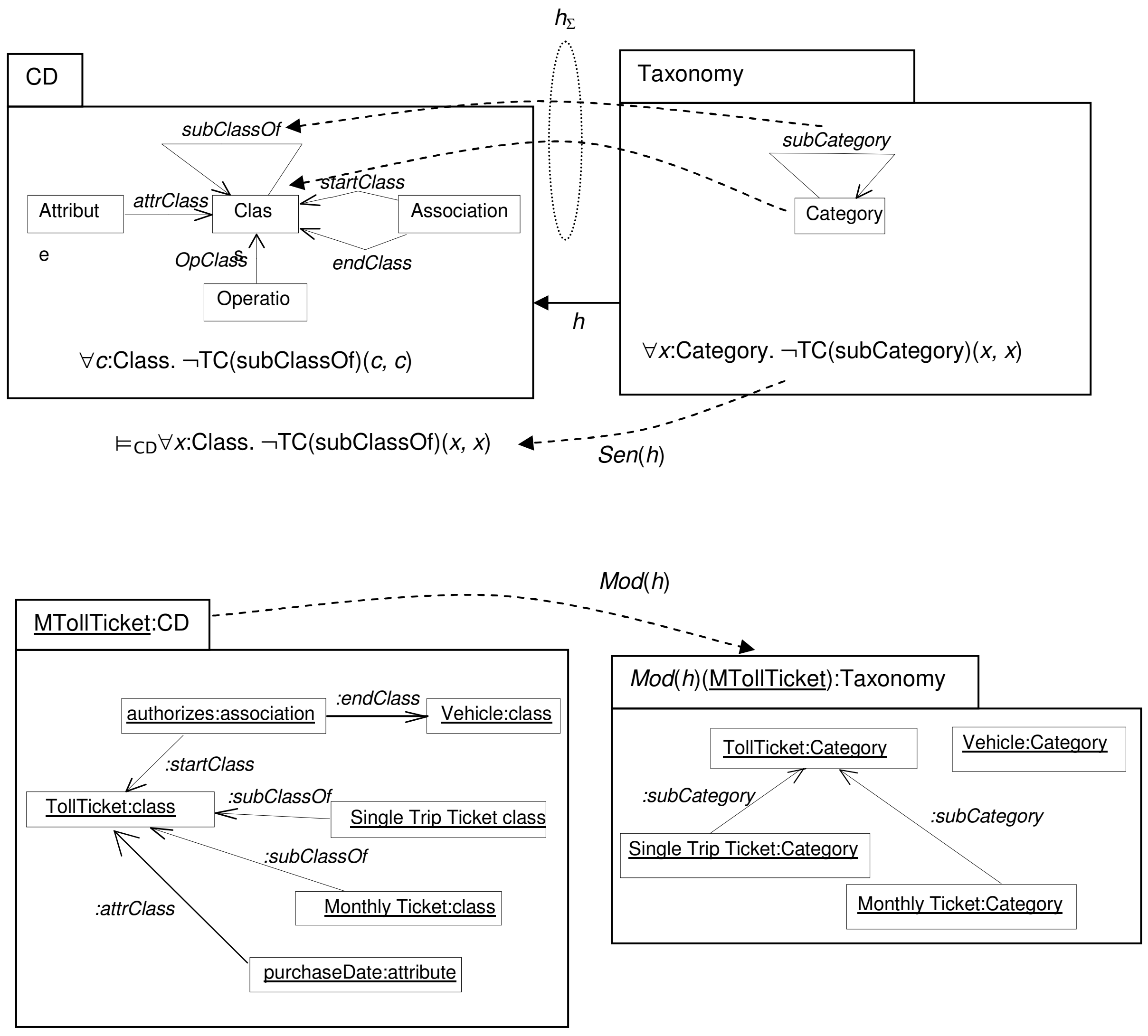


Figure 4.3. An example of a metamodel morphism.

and the predicate *subCategory* to the predicate *subClassOf*. With the resulting sentence translation function we see that *Sen*(f)("$\forall x$:Category. ¬TC(subCategory)(x, x)") = "$\forall x$:Class. ¬TC(subClassOf)(x, x)" which clearly follows logically from Ψ_{CD} = {$\forall c$:Class. ¬TC(subClassOf)(c, c)}. The bottom portion of the figure shows how an example of a particular model *Toll*:*CD* is transformed via the reduct *Mod*(h) 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 *Category* related by *subCategory*.

4.2.6 Model semantics

Although models have both syntax and semantics, it is common for the metamodel to only characterize the syntactic aspect of the models for a given model type. This is the case with both the MOF and Ecore metamodeling languages. 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 [UML2] expresses the abstract syntax as a metamodel but the semantics is expressed using English - the formalization of UML semantics continues to be an area of intense research [e.g., LMT09].

We make a similar assumption with the use of FO+ as a metamodeling formalism and take it to define only the syntax of a model type. However, we shall find it relevant to discuss model semantics and thus we need a framework for doing so. In general, defining the semantics of a modeling language requires the identification of the semantic domain and the definition of a semantic mapping from the syntactic structures to their meanings in the semantic domain

[HR04]. Since we have argued that a model is really just a graphical way of expressing a set of sentences about its subject, we will take a Tarskian approach to do this. In the same way that we can talk about the sets of interpretations and satisfying interpretations for sentences of a formal language, we assume that we have these for a model as well. In particular, for a model type *T* we define the following:

- $Intr_T$ is the function that maps each *T* model to its set of interpretations in the semantic domain.
- $Inst_T$ is the function that maps each *T* model to its set of satisfying interpretations (instances).

We will take these semantic mapping functions to capture the semantics of *T*. As noted above, the definitions of $Intr_T$ and $Inst_T$ are not typically given as part of the metamodel *T*. Note that for each *T* model $\underline{M}$, $Inst_T(\underline{M}) \subseteq Intr_T(\underline{M})$ since the satisfying interpretations are also interpretations. For example, when $T = CD$ and we consider the class diagram $\underline{MTollTicket}$ shown in Figure 4.2, each element of $Intr_{CD}(\underline{MTollTicket})$ is an assignment of sets to the class symbols *Vehicle*, *TollTicket*, etc. and an assignment of a relation to the association symbol *authorizes*. Such an assignment is also in $Inst_{CD}(\underline{MTollTicket})$ iff the sets assigned to *SingleTripTicket* and *MonthlyTicket* are subsets of the set assigned to *TollTicket*.

4.3 Model relationships

Two models are semantically related when the possible interpretations of one model constrain the possible interpretations of the other model. More formally, For two models $\underline{M1}$:*T1* and $\underline{M2}$:*T2* we can define the compatibility relation as follows.

Definition 4.1. ***Compatibility relation between two models.*** For models $\underline{M1}:T1$ and $\underline{M2}:T2$ we can define the compatibility relation $Comp_{\underline{M1},\underline{M2}} \subseteq Intr_{T1}(\underline{M1}) \times Intr_{T2}(\underline{M2})$ to be the pairs of interpretations of $\underline{M1}$ and $\underline{M2}$ that can co-occur as part of the same (larger) possible world.

Similar approaches to relating models semantically are taken elsewhere [G93, MB02]; however, we characterize some interesting properties of model relationships in terms of this relation and these are given in Table 4.1. The logical consequence property will be significant when we discuss partiality relationships in Section 4.3.2.

Table 4.1. Some formal properties expressed in terms of the compatibility relation.

Property	Occurrence condition	Description
$\underline{M1}$ is unrelated to $\underline{M2}$	$\mathrm{Comp}_{\underline{M1},\underline{M2}} = Intr_{T1}(\underline{M1}) \times Intr_{T2}(\underline{M2})$	$\underline{M1}$ and $\underline{M2}$ are unrelated since they do not constrain one another.
$\underline{M1} \vDash \underline{M2}$	$\forall x_1 \in Inst_{T1}(\underline{M1}), x_2 \in Intr_{T2}(\underline{M2}) \cdot \mathrm{Comp}_{\underline{M1},\underline{M2}}(x_1, x_2) \Rightarrow x_2 \in Inst_{T2}(\underline{M2})$	$\underline{M1}$ is a logical consequence of $\underline{M2}$ because every satisfying interpretation of $\underline{M1}$ only co-occurs with satisfying interpretations of $\underline{M2}$.
$\underline{M1}$ is inconsistent with $\underline{M2}$	$\neg\exists x_1 \in Inst_{T1}(\underline{M1}), x_2 \in Inst_{T2}(\underline{M2}) \cdot \mathrm{Comp}_{\underline{M1},\underline{M2}}(x_1, x_2)$	$\underline{M1}$ is inconsistent with $\underline{M2}$ because there are no co-occurring interpretations that satisfy both models.

At the syntactic level, a compatibility relation between models can be expressed[9] as a larger model we call the *relator* model containing the related models, or *endpoint* models. The interpretations of the relator model represent the larger world in which the interpretations of the endpoint models must co-occur. We will call a relator model, along with the designations of the embeddings of endpoint models, a *model relationship*. We can characterize the different ways that models can be related by classifying relator models into types. Thus, we can also define *model relationship types*.

For example, Figure 4.4 shows an instance of the *objectsOf* relationship type that holds between sequence diagram *BuyTollTicket*:*SD* and object diagram *Toll*:*OD*. The relationship between the models is expressed by embedding them in relator model *R1*:*objectsOf* along with a *mapping* between them that represents additional information used to connect the elements of the models. Here the mapping maps each object symbol in the sequence diagram to the object symbol in the object diagram that represents the same object (via the identity relation *id*) and maps each message in the sequence diagram to the link in the object diagram over which the message is sent (via the relation *sentOver*). The content of the relator model is constrained so that both *id* and *sentOver* are total functions and the mapping must be consistent in the sense that the endpoint objects of a message should be the same as the endpoint objects of the link to which it is mapped.

The formal approach we use for defining relationship types is to define projection functions from the model type of the relator model to the model types of the endpoint models that show

[9] Of course, we are only talking about compatibility relations that are expressible syntactically.

how to “extract” the endpoint models from each instance of the relator model. In Figure 4.4, the model type of the relator model is [[*objectsOf*]] and the two projection functions are π_{OD} and π_{SD}.

$$\begin{tikzcd}
 & {[[OD]]} \\
{[[objectsOf]]} \arrow[ur, "\pi_{CD}"] \arrow[dr, "\pi_{SD}"'] & \\
 & {[[SD]]}
\end{tikzcd}$$

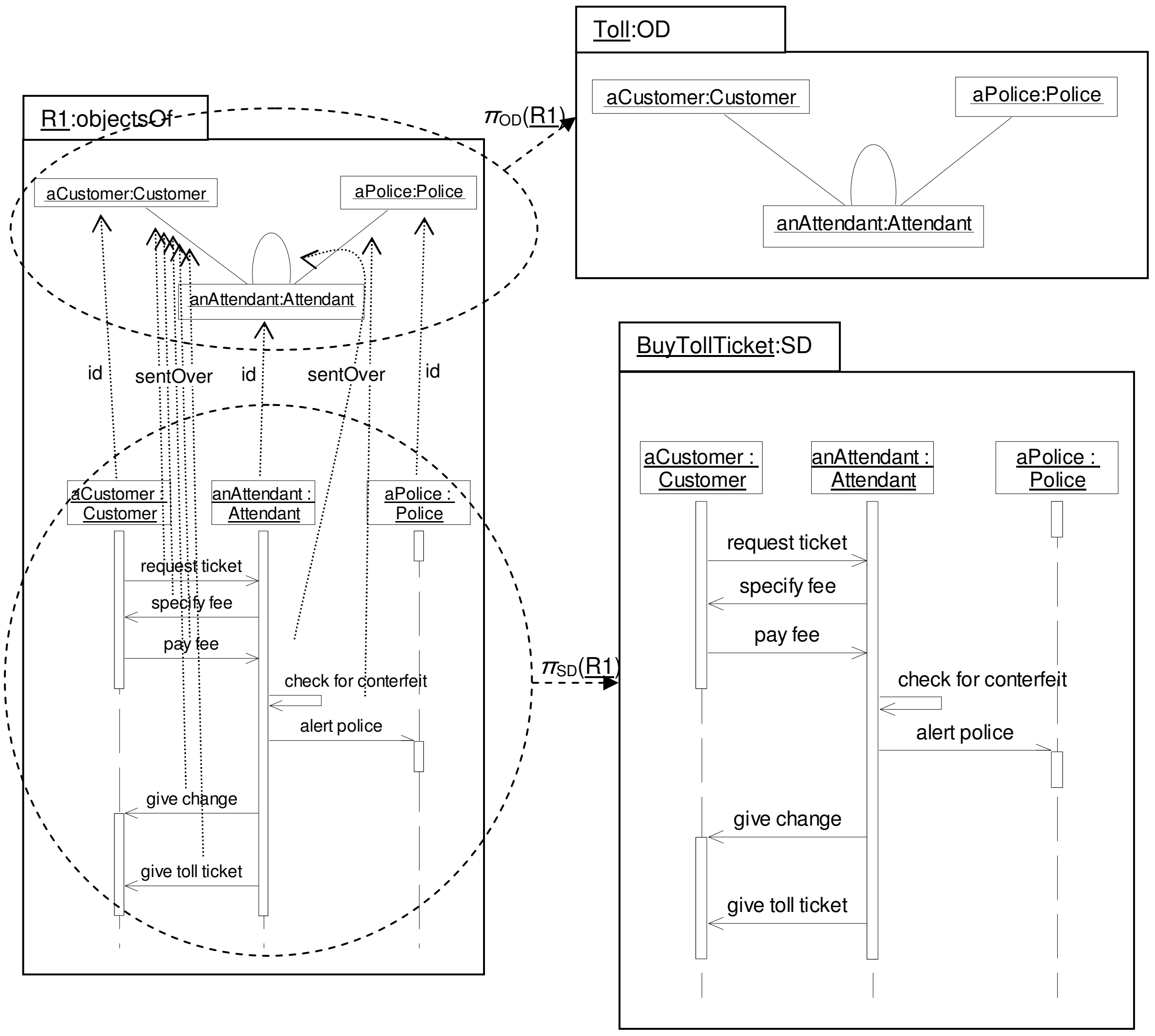


Figure 4.4. A relationship between a sequence diagram and an object diagram.

4.3.1 Defining model relationship types using metamodels

In Figure 4.4, the triple $\langle[[objectsOf]], \pi_{OD}, \pi_{SD}\rangle$ defines a model relationship type defined between model types *OD* and *SD*, and we can assume that there is a metamodel *objectsOf* that characterizes the relator models of this relationship type. Furthermore, the natural thing to consider is that p_{OD} and p_{SD} are the reducts corresponding to two metamodel morphisms $p_{OD}:OD\rightarrow objectsOf$ and $p_{SD}:SD\rightarrow objectsOf$ that map these endpoint metamodels into relator metamodel. Thus, we can get π_{OD} and π_{SD} "for free" since $\pi_{OD} = Mod(p_{OD})$ and $\pi_{SD} = Mod(p_{SD})$. Figure 4.5 illustrates the metamodels and metamodel morphisms involved.

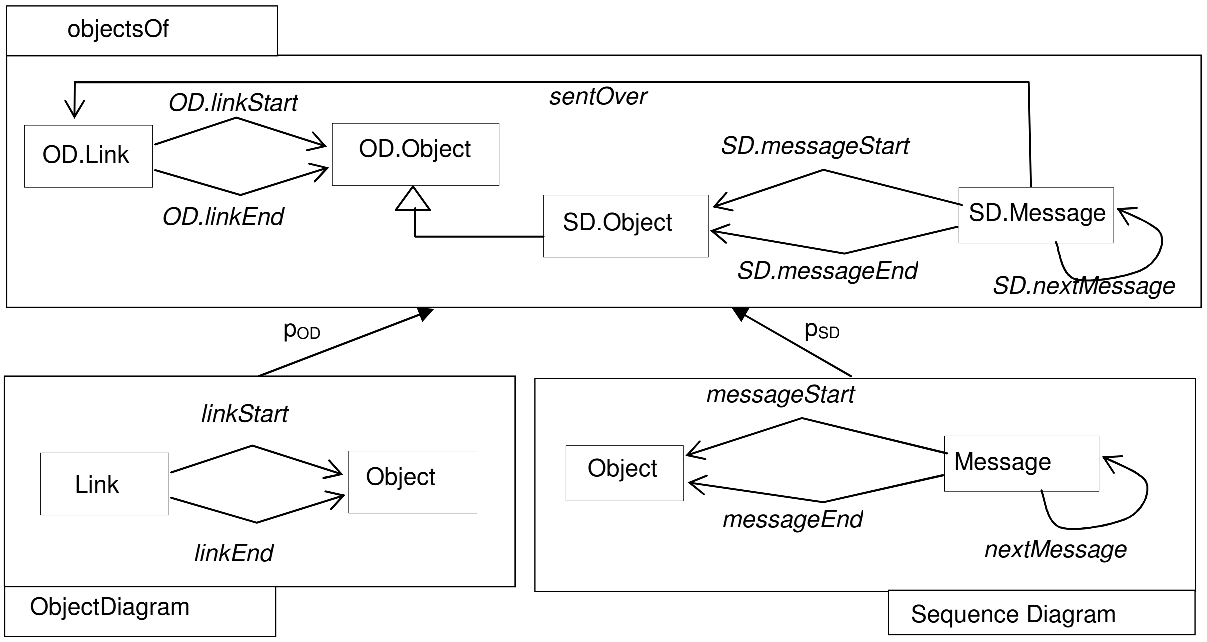


Figure 4.5. Signature part of the *objectsOf* relationship type.

The full definitions of the metamodels[10] *OD*, *SD* and *objectsOf* are as follows:

```
OD =
        sorts   Object, Link
        func    linkStart: Link → Object
                linkEnd: Link → Object


SD =
        sorts   Object, Message
        func    first : Message, last : Message
                messageStart: Message → Object
                messageEnd: Message → Object
                nextMessage : Message o→ Message

        constraints
                // the message “first” has no predecessor
                        ∀x:message. ¬(first = next(x))
                // only the message “last” has no successor
                        ∀x:message. x ≠ last ⇔ ∃y:message. y = next(x)


objectsOf =   theOD.OD + theSD.SD +
          subsort  theSD.Object ≤ theOD.Object
          func   sentOver : theSD.Message → theOD.Link
          constraints
                // sentOver preserves endpoint incidence
                        ∀x: theSD.Message. messageStart(x) = linkStart(sentOver(x))
                        ∀x: theSD.Message. messageEnd(x) = linkEnd(sentOver(x))
```

[10] These are simplified versions of the corresponding UML diagram types for the purposes of illustration.

In addition, we assume that p_{CD}:*OD*$\rightarrow$*objectsOf* and p_{SD}:*SD*$\rightarrow$*objectsOf* are the obvious inclusion metamodel morphisms. The following notational extensions used in these metamodels. Functions with no arguments and only a return type represent constants – e.g., *first* and *last* in *SD*. We designate a partial function using "o→" notation. For example, the function *next* in *SD* is a partial function since the last message has no next message. The subsort designator allows subsetting of sorts to be specified. The fact that the metamodel *objectsOf* imports the metamodels for *OD* and *SD* is expressed by the expression "objectsOf = theOD.OD + theSD.SD + ". Note that the imported signature elements are qualified with a prefix to ensure that there are no naming conflicts. Thus, *theOD.Object* is a distinct sort from *theSD.Object*.

Each instance of *objectsOf* has the *sentOver* functions that map messages to links. In addition, the constraint on *sentOver* in *objectsOf* requires that the endpoint objects of a message should be the same as the endpoint objects of the link to which it is mapped. The constraint that the objects of the sequence diagram should be a subset of objects in the object diagram is expressed using the subsort designation[11]. Clearly, the reducts *Mod*(p_{OD}):[[*objectsOf*]]$\rightarrow$[[*OD*]] and *Mod*(p_{SD}):[[*objectsOf*]]$\rightarrow$[[*SD*]] extract the *OD* and *SD* endpoint models out of each *objectsOf* instance. Thus $\langle$[[*objectsOf*]], *Mod*(p_{OD}), *Mod*(p_{SD})$\rangle$ is the desired model relationship type.

We can now generalize the procedure with *objectsOf* as follows. Given metamodels $T_1, \ldots, T_n$, a relator metamodel R and metamodel morphisms $h_1{:}T_1\rightarrow R, \ldots, h_n{:}T_n\rightarrow R$ we can define the

[11] In Figure 4.4 this was expressed using the identity relation *id* which is defined over all sorts.

model relationship type $\langle[[R]], Mod(h_1), \ldots, Mod(h_n)\rangle$. Correspondingly, we will say that the metamodel for the relationship type is $\langle R, h_1, h_2, \ldots, h_n\rangle$ and call it a *relationship metamodel*.

A key benefit of using metamodel morphisms for defining model relationship types is that the approach can be formulated in a way that is independent of the metamodeling language since each metamodeling language can define its own type of metamodel morphism. Furthermore, Institution Theory provides a formal means to relate different logics using *Institution morphisms* [GB92] and thus our approach can be extended similarly to heterogeneous metamodeling formalisms. We do not pursue this direction further in this thesis as it is secondary to our interests here.

4.3.2 General characteristics of relationship types

In this section we identify some properties of a relationship type that will be useful in subsequent chapters.

Mapped relationship types

A relationship type may or may not have mapping content. The relationship type defined by metamodel $\langle R, h_1, h_2, \ldots, h_n\rangle$, has mapping content just in case there is exists an element of R that is not found in the image of any metamodel morphism h_i. In this case, that element would not be projected into any of the endpoint models and so it represents additional information not found in any of these models. If a relationship type has mapping content we call it a *mapped* relationship type. For example, *objectsOf* is a mapped relationship type because the *sentOver* association is not found in either the endpoint *OD* or the *SD*. In contrast, the *submodelOf* relationship type defined in Section 4.3 has no content beyond its endpoint models and so is not mapped.

Pure relationship types

In a relationship type, the relator model may or may not be uniquely determined (up to isomorphism) by its endpoint models. Consider relationship type defined by $\langle R, h_1, h_2, \ldots, h_n\rangle$, and a tuple of models $\langle \underline{M}_1, \ldots, \underline{M}_n\rangle$ of types $T_1, \ldots, T_n$, respectively. Define $R(\underline{M}_1, \ldots, \underline{M}_n) = \{\underline{r}: R \mid h_1(\underline{r}) = \underline{M}_1 \wedge \ldots \wedge h_n(r) = \underline{M}_n\}$. That is, $R(\underline{M}_1, \ldots, \underline{M}_n)$ is the set of possible conformant relator models of type R for $\langle \underline{M}_1, \ldots, \underline{M}_n\rangle$. Then there are three possible cases:

C1. $|R(\underline{M}_1, \ldots, \underline{M}_n)| = 0$

C2. $|R(\underline{M}_1, \ldots, \underline{M}_n)| > 0$ and all $r \in R(\underline{M}_1, \ldots, \underline{M}_n)$ are isomorphic

C3. $|R(\underline{M}_1, \ldots, \underline{M}_n)| > 0$ and some $r \in R(\underline{M}_1, \ldots, \underline{M}_n)$ are non-isomorphic

In case (C1) no relator model exists for $\langle \underline{M}_1, \ldots, \underline{M}_n\rangle$ and so this relationship does not hold. In case (C2), a unique relator model exists, up to isomorphism. Thus, if R is a mapped relationship type then the mapping is derived from $\langle \underline{M}_1, \ldots, \underline{M}_n\rangle$. When R is such that for all tuples $\langle \underline{M}_1, \ldots, \underline{M}_n\rangle$, only (C1) or (C2) occurs, then we say that R is a *pure* relationship type.

The *objectsOf* relationship type is not a pure relationship type. Note that in Figure 4.4, case (C2) holds since there is only one conformant way to define the *sentOver* association; however, if we change *Toll* by adding another link between *aCustomer* and *anAttendant* then case (C3) would hold. The definition of pure relationship type requires that (C3) cannot hold for any tuple of endpoint models. The relationship type *submodelOf* is pure. It is easy to see that any relationship type that is not mapped must be pure since the only way to get more than one relator model in $R(\underline{M}_1, \ldots, \underline{M}_n)$ is to have mapping content. Pure relationship types are significant because it is possible to test conformance of the endpoint models to the relationship without specifying any mapping information between the models (i.e., since the mapping information is derived).

Transformations

A pure relationship type in which the relator model is uniquely determined (up to isomorphism) by all but one of its endpoint models is called a *transformation*. More formally, for pure relationship type defined by $\langle R, h_1, h_2, \ldots, h_n\rangle$, let the endpoint models identified by $\langle h_1, h_2, \ldots, h_{n-1}\rangle$ be the ones that uniquely determine the relator model. We call these the *inputs* to the transformation and the endpoint model identified by h_n is the *output* of the transformation. Since the relator model is determined by the input models we can the project the output model from this using the reduct $Mod(h_n)$. If the transformation is a mapped relationship type then the mapping content in the relator model represents traceability information for the transformation.

Partiality relationship types

In Table 4.1, we identified the formal condition for when logical consequence holds between models. A binary relationship type that guarantees that this holds between its endpoint models is called a *partiality* relationship type. Formally, $\langle R, h_1, h_2\rangle$ is a partiality relationship when

$\forall \underline{m}_1{:}T_1, \underline{m}_2{:}T_2, \underline{r}{:}R \cdot \ \underline{r} \in R(\underline{m}_1, \underline{m}_2) \Rightarrow \underline{m}_2 \vDash \underline{m}_1$. Intuitively, when $\underline{r}{:}R$ holds between models $\underline{M}_1$ and $\underline{M}_2$, then $\underline{M}_1$ is a semantically partial model relative to $\underline{M}_2$ and so anything $\underline{M}_1$ says can be inferred from what $\underline{M}_2$ says given the mapping information in $\underline{r}$.

Most of the interesting model relationship types encountered in software engineering are partiality relationship types. For example, if $\underline{M}_1$ is an *abstraction* of $\underline{M}_2$ then we expect $\underline{M}_2 \vDash \underline{M}_1$ since the abstraction should contain some derived form of the information in $\underline{M}_2$ where the mapping defines the relationships between the abstract elements in $\underline{M}_1$ and the refined elements in $\underline{M}_2$. As a more concrete example, we define *submodelOf*($\underline{M}_1$, $\underline{M}_2$) in Section 4.3 as the case where $\underline{M}_1 \subseteq \underline{M}_2$ and $\underline{M}_2 \vDash \underline{M}_1$ since a submodel should not only be a syntactic subset of the whole model but should also semantically carry a part of the information in the whole model. Many common partiality relationship types are listed in Section 4.3.2.

Partiality relationship types have a number of characteristics that make them interesting for expressing the relationships between model roles. First, due to logical consequence, they are *truth preserving* and so we can propagate truth between models. For example, if we know that what $\underline{M}_2$ says is true about its subject and $\underline{M}_1$ is a correct abstraction of $\underline{M}_2$ then we know that what $\underline{M}_1$ says is also true. Second, partiality relationships are often closed under compositions. For example, if $\underline{M}_1$ is an abstraction of $\underline{M}_2$ via mapping $\underline{r}_1$ and $\underline{M}_2$ is an abstraction of $\underline{M}_3$ via mapping $\underline{r}_2$ then $\underline{M}_1$ is an abstraction of $\underline{M}_3$ via a mapping that can be formed from a composition of $\underline{r}_1$ and $\underline{r}_2$. Third, we can often decompose a model using a partiality relationship. For example, we can decompose a model into submodels (this is the focus of Chapter 7) or into abstractions. Conversely, we can compose submodels or abstractions into larger models. Note that since it depends on the kind of partiality relationship used, the notion of decomposition/composition here is more general than just “parts” based. The last two properties relate to the fact that partiality relationship types often form categories in the sense of

Category Theory[12]. Although it brings a rich mathematical foundation to models and their relationships, exploring the Category Theoretic perspective is beyond the scope of this thesis.

4.3.3 Abstract model relationship types

In a way similar to model types, we can define abstract relationship types that have no specific relator metamodel but have properties that its instances must satisfy. In general, the relationship types required for a particular modeling project can often be seen as concrete specializations of typical abstract relationship types encountered in software engineering such as *submodelOf*, *abstractionOf*, *aspectOf*, etc. These abstract relationship types carry specific semantics and play distinct roles within software development that are inherited by their specializations. If formally characterized, they can also be used to provide guidance in developing their concrete specializations. Some authors have proposed candidate lists of abstract relationship types useful in modeling [F05, BCE06]; however, determining a complete set of such types is an open problem.

We present some common abstract binary relationship types in Figure 4.6. Note that almost all of them are shown to be partiality relationship types[13]. The partiality transformations all construct an output model M_1 that has a particular partiality relationship to the input model M_2 and the type of this relationship is indicated by the dashed line. Formal characterizations of most of these are beyond the scope of this thesis and we leave this to future work.

[12] A good introduction for computer scientists can be found in [P93].

[13] Since we are not giving the formal definitions of each of these types, we are appealing to the readers intuition about the validity of the classification. As a result, some may be disputed (e.g., is *refactoringOf* necessarily a partiality relationship?)

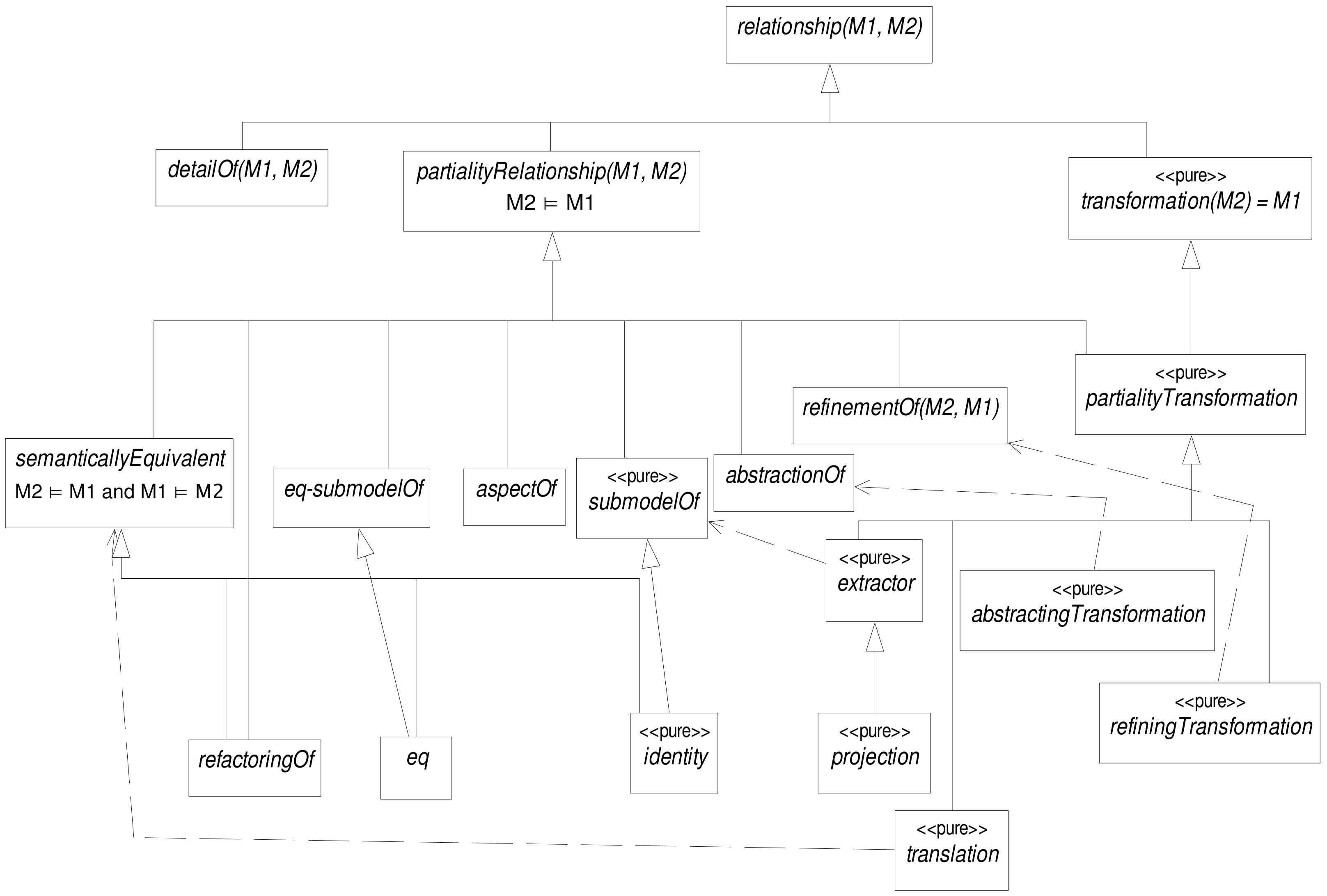


Figure 4.6. A taxonomy of abstract relationship types.

Eq-submodelOf and *eq* are the names we give to the commonly used homomorphism and isomorphism mappings where the element correspondences are interpreted as equivalence [e.g., SE05]. We formalize *eq* below and *submodelOf* is defined in Section 4.3. *Extractors* extract a submodel from a model and these are defined in Chapter 5 along with *detailOf*.

Constructed relationship types

In a similar way to model types, some abstract relationship types can be concretized in a generic way for any model type by defining a type constructor and these are called constructed relationship types. The benefit of constructed relationship types is that they are obtained "for free" from the other information. For example, we define *eq*[*T*] for model type *T* as follows: *eq*[*T*]($\underline{M}_1$, $\underline{M}_2$) 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. The construction of relator metamodel is as follows:

```
eq[T](M1:T, M2:T) =  M1.T + M2.T +
    **func**
    *for each sort* $S \in sorts_T$,
        map$S$ : M1.$S \rightarrow$ M2.$S$
    **constraints**
  // each map function is bijective
  *for each sort* $S \in sorts_T$,
        $\forall x_1, x_2$:M1.$S$ · map$S(x_1)$ = map$S(x_2) \Rightarrow x_1 = x_2$
        $\forall x_2$: M2.$S$ $\exists x_1$:M1.$S$ · map$S(x_1) = x_2$
```

```
// preserve incidence
 for each function $F$:$S_1 \times \ldots \times S_{n-1} \rightarrow S_n \in func_T$
      $\forall x_1$:M1.$S_1$, …, $x_{n-1}$:M1.$S_{n-1}$ ·
           M1.$F$($x_1$, …, $x_n$) = M2.$F$(map$S_1$($x_1$), …, map$S_{n-1}$($x_{n-1}$))
 for each predicate $P$:$S_1 \times \ldots \times S_n \in pred_T$
      $\forall x_1$:M1.$S_1$, …, $x_n$:M1.$S_n$ ·
           M1.$P$($x_1$, …, $x_{n-1}$) $\Leftrightarrow$ M2.$P$(map$S_1$($x_1$), …, map$S_n$($x_n$))
```

In a similar way, we can define the submodel relationship type *submodelOf*[*h*](T_1, T_2) given any signature morphism h:$\Sigma_{T1} \rightarrow \Sigma_{T2}$. This construction is described in the next section.

4.4 Submodels and Diagrams

One of the most basic types of relationships between models is the *submodelOf* relationship. A commonly occurring example of a submodel in practice is the *diagram*. A model is often manifested as a set of diagrams, possibly of different types, that decompose and structure the content of the model in a coherent way. The prototypical example of this is the UML which defines a single metamodel for UML models and identifies thirteen types of diagrams that can be used with it [UML2]. Diagrams are normally associated with concrete syntax but since in this thesis we are interested only in content (i.e., abstract syntax), a diagram reduces to the submodel it picks out from a model.

Intuitively, given a metamodel T, if $\underline{M_1}$:T is a submodel of $\underline{M_2}$:T then the elements of $\underline{M_1}$ are a subset of the elements of $\underline{M_2}$ and any predicate or function instance that holds between elements in $\underline{M_1}$ also holds in $\underline{M_2}$. This is the syntactic criterion and we can write it as $\underline{M_1} \subseteq \underline{M_2}$; however, there is also the semantic criterion $\underline{M_2} \vDash \underline{M_1}$ since *submodelOf* is a partiality relationship. This is based on the intuition that a "reasonable" submodel should carry a subset of the information that the whole model carries and so it should not say anything that the whole model does not say. Having $\underline{M_1} \subseteq \underline{M_2}$ alone is not sufficient because there are cases when it could lead to confusion for model consumers.

Figure 4.7 illustrates the issue using the transportation system example. Note that the designator “{complete}” means that the set of subclasses indicated covers the superclass – i.e., every instance of *Vehicle* must also be an instance of *Car*, *SUV* or *Truck*. This designator is a valid part of UML 2.2 class diagram syntax. Although both $\underline{DTollPriceA}$ and $\underline{DTollPriceB}$ are well-formed class diagrams and it is the case that $\underline{DTollPriceA} \subseteq \underline{DTollPrice}$ and $\underline{DTollPriceB} \subseteq \underline{DTollPrice}$, only $\underline{DTollPriceA}$ satisfies the semantic criterion. A consumer reading $\underline{DTollPriceA}$ would correctly learn that the class *Vehicle* is covered by its subclasses *Car*, *SUV* and *Truck*. However, a consumer reading $\underline{DTollPriceB}$ would be misled into thinking that *Vehicle* is covered by *Car* and *Truck* alone.

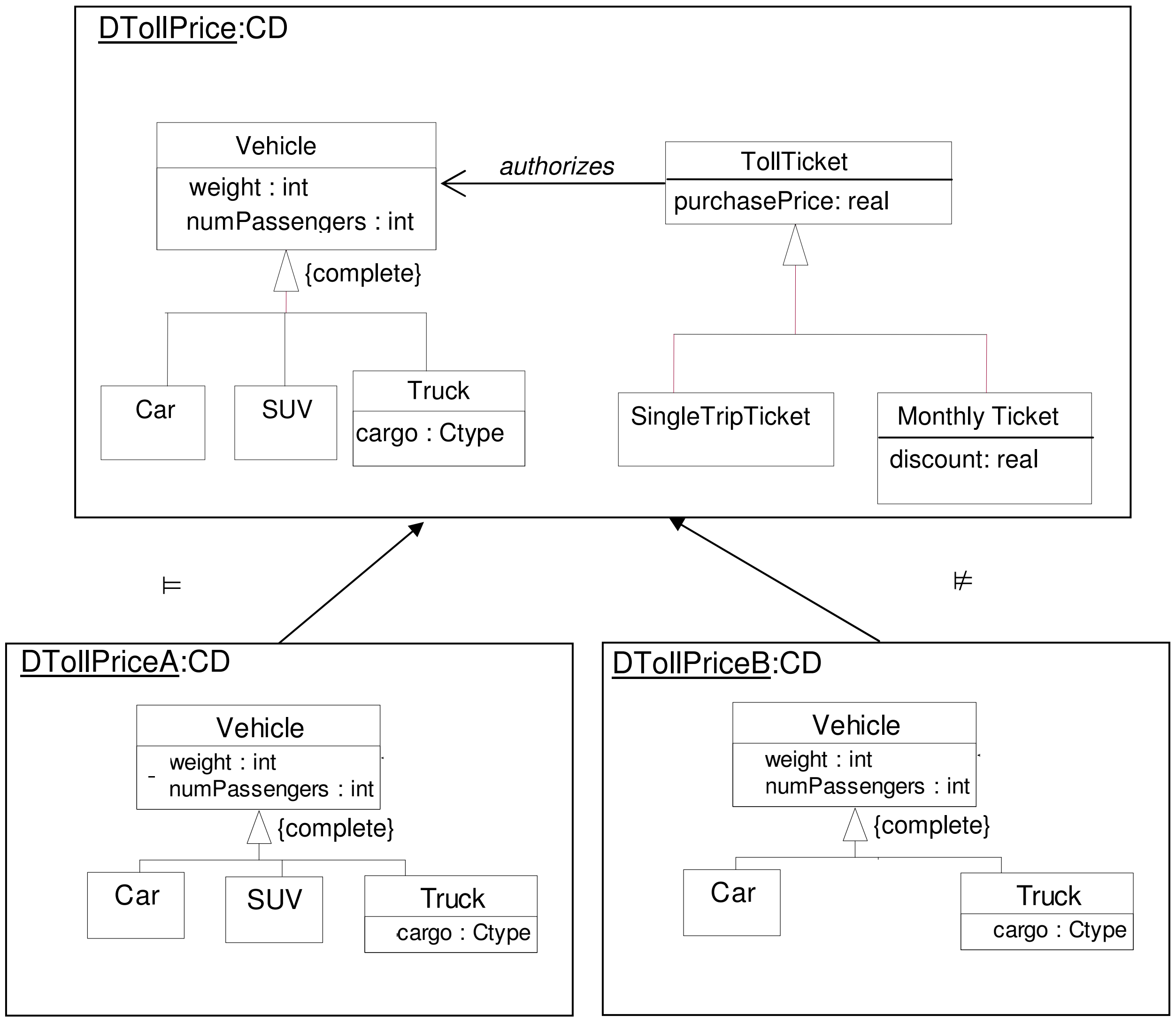


Figure 4.7. Examples of valid and invalid submodels.

4.4.1 The submodelOf relator metamodel

The metamodel for the submodel relationship type over the version of *CD* shown in Figure 4.1 can be expressed as:

```
submodelOf(M1:CD,M2:CD) =   M1.CD + M2.CD +
      subsort  M1.Class ≤ M2.Class
               M1.Association ≤ M2.Association
               M1.Attribute ≤ M2.Attribute
               M1.Operation ≤ M2.Operation
      constraints
         // M1.CD is a subset of M2.CD
               ∀x:M1.Association · M1.startClass(x) = M2.startClass(x)
               ∀x:M1.Association · M1.endClass(x) = M2.endClass(x)
               ∀x:M1.Attribute · M1.attrClass(x) = M2.attrClass(x)
               ∀x:M1.Operation · M1.opClass(x) = M2.opClass(x)
               ∀x1, x2 :M1.Class · M1.subclassOf(x1 , x2) ⇒ M2.subclassOf(x1 , x2)
```

p_{M1}:*CD* → *submodelOf*(*CD*, *CD*), p_{M2}:*CD* → *submodelOf*(*CD*, *CD*) are metamodel morphisms into *M1.CD* and *M2.CD*, respectively. With the submodel relationship, 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 the submodel as in the whole model. Thus, the submodel could be considered to simply be a delineation of a portion of the whole model. Note that this addresses both the syntactic and semantic criteria for submodel for this simplified version of class diagram.

The submodel relationship can also hold between models of different types. For example, in a *UML* model, we consider the diagrams to pick out submodels of type *CD*, *AD*, *SMD*, etc. In general, we would like to define the submodel relationship that can hold between two models $\underline{Msub}:Tsub$ and $\underline{M}:T$.

In order to accomplish this we assume we have a signature morphism $h:\Sigma_{\mathrm{Tsub}}\rightarrow\Sigma_{\mathrm{T}}$ that maps the signature of *Tsub* to a semantically equivalent subsignature of *T*. We can then generically define a constructor for the submodel relationship type defined by *h* as:

```
submodelOf[$h$] (Msub:T1, M:T) =  Msub.Tsub + M.T + $\Psi_{\mathrm{TsubSem}}$+
    **subsort**  *for each sort*  $S \in sorts_{Tsub}$,
                    Msub.$S \leq$ M.$h(S)$
    **constraints**
            *for each function* $F{:}S_1 \times \ldots \times S_{n-1} \rightarrow S_n \in func_{Tsub}$
                $\forall x_1$:Msub.$S_1$, …, $x_{n-1}$:Msub.$S_{n-1}$ ·
                    Msub.$F(x_1, \ldots, x_{n-1})$ = M.$h(F)(x_1, \ldots, x_{n-1})$
            *for each predicate* $F{:}S_1 \times \ldots \times S_n \in pred_{Tsub}$
                $\forall x_1$:Msub.$S_1$, …, $x_n$:Msub.$S_n$ ·
                    Msub.$P(x_1, \ldots, x_n) \Rightarrow$ M.$h(P)(x_1, \ldots, x_n)$
```

With metamodel morphisms $p_{\mathrm{sub}}:Tsub \rightarrow submodelOf[h](Tsub, T)$ and $p_{\mathrm{base}}:T \rightarrow submodelOf[h](Tsub, T)$. The set of constraints Ψ_{TsubSem} represent the additional constraints that we need to add to satisfy the semantic criterion.

For diagram types, it is typically the case that $\Sigma_{Tsub} \subseteq \Sigma_T$ as is the case with the diagram types of a UML model. In this case, *h* maps each sort, function and predicate of Σ_{Tsub} to the same sort, function and predicate in Σ_T. However, we need the more general form of *h* when we consider the case that a relationship is a submodel of a model. For example, consider the *extends*(*det*:*CD*, *gen:CD*) relationship type that can hold between two class diagrams when every class in *det* is a subclass of a class in *gen*. Now consider the case where we have <u>*E12*</u>(<u>*Veh1*</u>, <u>*Veh2*</u>):*extends* for two class diagrams <u>*Veh1*</u> and <u>*Veh2*</u> as shown in Figure 4.8. Not only are both <u>*Veh1*</u> and <u>*Veh2*</u> submodels of the same UML model *TransportationSystem* but the *extends* mapping <u>*E12*</u> is as well since *TransportationSystem* contains the subclass relationships between the classes of <u>*Veh1*</u> and the classes of <u>*Veh2*</u>.

Figure 4.9 shows the signature morphisms mapping *CD*, *UML* (only partially shown) to the *extends* relator metamodel. All the morphisms map in the obvious way except h_{dU} for which we show a few of the correspondences to get the idea of how the morphism works. Note that the following conditions must also hold:

$$h_{CU} = h_{dU} \circ p_{det}$$

$$h_{CU} = h_{dU} \circ p_{gen}$$

This ensures that arguments *det* and *gen* of *extends* map to *UML* consistently with the way *CD* maps to *UML*.

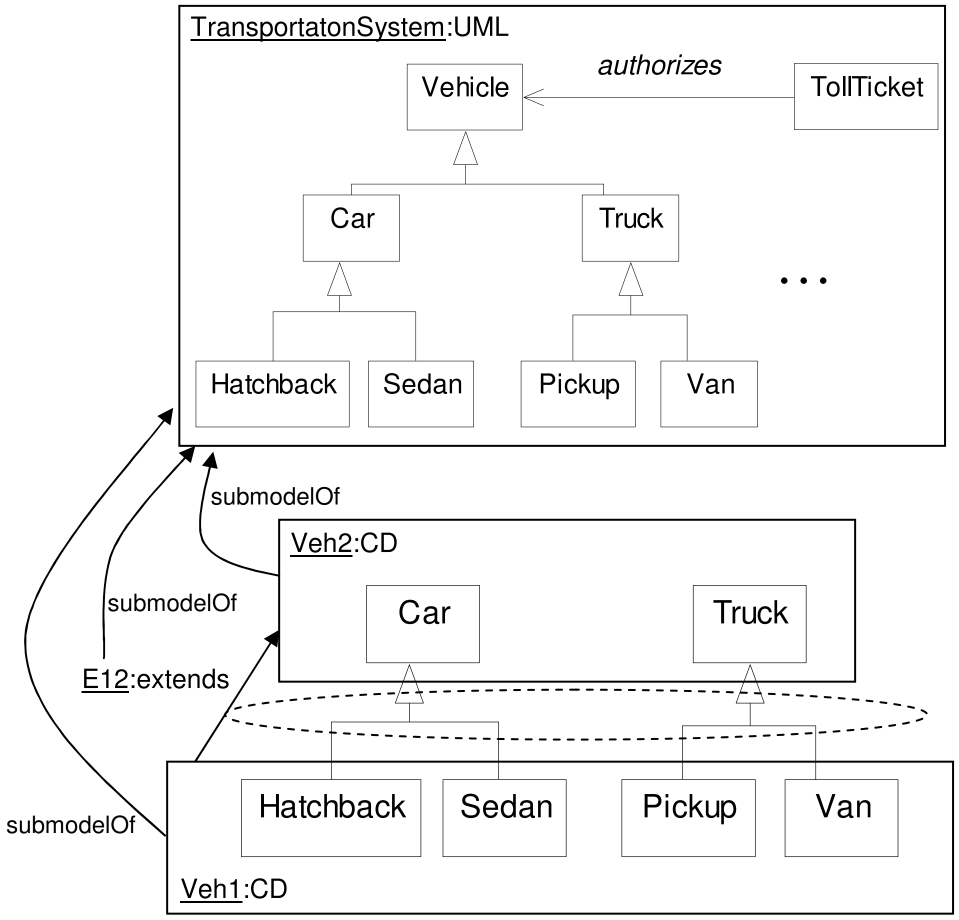


Figure 4.8. An example of how a relationship can be a submodel.

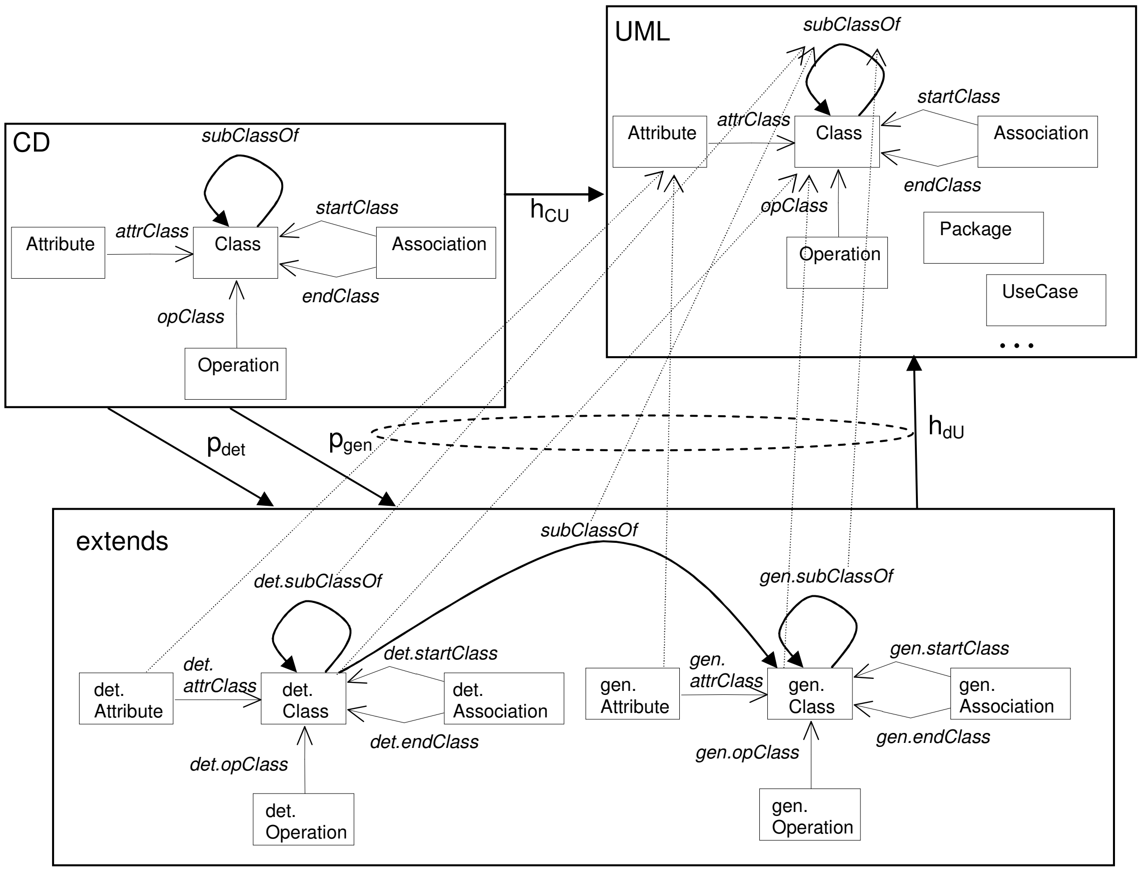


Figure 4.9. Morphisms used for submodel relationships.

4.4.2 Submodel types

The signature morphism $h:\Sigma_{Tsub}\rightarrow\Sigma_T$ shows how *Tsub* models can be submodels of *T* models. Now we consider whether there are any additional criteria that *Tsub* must satisfy. Note that in general, the constraints of *Tsub* may be stronger than the constraints of *T*. For example, in UML, communication diagrams can only represent interactions in which message overtaking does not take place [UML2] and so the constraints on communication diagrams are stronger than on the interactions they represent within a UML model. Thus, a communication diagram is a type of submodel of a UML model that is more constraining than UML.

A plausible criterion for *Tsub* would be rule out situations where we can have $\underline{Msub}$: *Tsub* with no possible $\underline{M}$:*T* for which $\underline{Msub}$ can be a submodel. For example, let *Tsub* be that same as *CD* but without the acyclicity constraint on the *subClassOf* relationship. Thus, the class diagram that consists of a single class that is a subclass of itself is a valid *Tsub* model but it clearly cannot occur as a submodel of any valid *CD*. Thus, we would reject *Tsub* as being a valid submodel type for *CD*. We can express the submodel type validity condition formally as follows:

$$\forall \underline{m_1}:\mathrm{Tsub}\ \exists \underline{m}:\mathrm{T} \cdot \mathrm{submodel}\mathit{Of}(\underline{m_1}, \underline{m})$$

This expresses the requirement that *Tsub* is extensionally consistent with *T*. That is any *Tsub* model can be extended to a *T* model.

4.4.3 Submodel sum

An important operation is the *sum* of a set of submodels of a common model. Intuitively, if $\underline{M}_1$ and $\underline{M}_2$ are submodels of $\underline{M}$ then the sum $\underline{M}_1 + \underline{M}_2$ is the smallest submodel of M that contains both $\underline{M}_1$ and $\underline{M}_2$ as submodels[14]. The computation that achieves this is to take the union of the content of the models for each sort, function and predicate. First consider sums of submodels of the same type. Formally, the semantics of the sum $\underline{M}_1 + \underline{M}_2$ of submodels $\underline{M}_1{:}T$ and $\underline{M}_2{:}T$ is defined as follows[15]:

$$\text{for each sort } S \in \mathit{sorts}_T,\ [[(\underline{M}_1{:}T + \underline{M}_2{:}T).S]] = [[\underline{M}_1.S]] \cup [[\underline{M}_2.S]] \tag{4.1}$$

$$\text{for each function } F \in \mathit{func}_T,\ [[(\underline{M}_1{:}\mathrm{T} + \underline{M}_2{:}\mathrm{T}).F]] = [[\underline{M}_1.F]] \cup [[\underline{M}_2.F]]$$

$$\text{for each predicate } P \in \mathit{pred}_T,\ [[(\underline{M}_1{:}\mathrm{T} + \underline{M}_2{:}\mathrm{T}).P]] = [[\underline{M}_1.P]] \cup [[\underline{M}_2.P]]$$

This can be generalized to the sum of an arbitrary finite set of submodels of type T in the natural way by extending the unions. Although the types of $\underline{M}_1$ and $\underline{M}_2$ are T, we take the type of $(\underline{M}_1 + \underline{M}_2)$ to be $\langle \Sigma_T, \varnothing \rangle$ since we cannot in general guarantee that $(\underline{M}_1 + \underline{M}_2) \models \Psi_T$. Figure 4.10 illustrates a simple example of this. The submodels taken individually are well-formed *CD* models but the sum violates the acyclicity constraint. Note that we still consider *Cars* to be a valid *sum* of *Cars1* and *Cars2* even though the result is not a well-formed *CD*. The reason why we “tolerate” this will become clear in Chapter 6 when we discuss macromodels. The idea is

[14] This corresponds to the *colimit* operation in Category Theory.

[15] We treat functions and relations here as sets of tuples and union is set union.

that the modeler intends both that <u>*Car*</u> = <u>*Car1*</u> + <u>*Car2*</u> and that <u>*Car*</u> $\models \Psi_{CD}$ but these constraints are checked independently.

Now consider the situation where we are taking the sum of models having multiple submodel types $\{T_1, \ldots, T_n\}$ - then we must have a way to determine the result type. To do this we assume that the sum operation is typed by a result type and there exist signature morphisms from each submodel type to the result type. For example, consider the situation depicted in Figure 4.11. This expresses the fact that <u>*M*</u> is the sum of submodels <u>*Veh1*</u>, <u>*Veh2*</u> and the relationship <u>*E12*</u>. In this case, we seek the sum operation that takes models/relationships of types {*CD*, *extends*} and produces a submodel with signature $\langle \Sigma_{UML}, \varnothing \rangle$. The signature morphisms shown in Figure 4.9 show how to construct such a sum: first transform the models

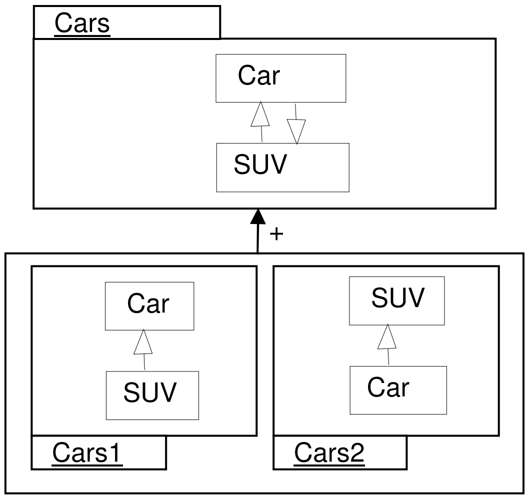


Figure 4.10. An example of inconsistent union.

to signature Σ_{UML} and then take the sum as defined in formula 4.1 above. The signature morphism $h{:}\Sigma_{T1} \rightarrow \Sigma_{T2}$ defines a corresponding transformation of models $Up(h){:}Mod(\Sigma_{T1}) \rightarrow Mod(\Sigma_{T2})$ where for any $\underline{M}{:}T1$, the semantics of $Up(h)(\underline{M})$ are as follows:

for each sort $S \in sorts_{T2}$, $[[\mathrm{Up}(h)(\underline{\mathrm{M}}).S]] = \cup\{\, [[\underline{\mathrm{M}}.s]] \mid S = h(s)\}$

for each predicate $F \in func_{T2,}$ $[[\mathrm{Up}(h)(\underline{\mathrm{M}}).F]] = \cup\{\, [[\underline{\mathrm{M}}.f]] | \; F = h(f)\}$

for each predicate $P \in pred_{T2,}$ $[[\mathrm{Up}(h)(\underline{\mathrm{M}}).P]] = \cup\{\, [[\underline{\mathrm{M}}.p]] \mid P = h(p)\}$

That is, we rename and union the sets of elements, function and predicate instances according to the mappings in *h*. Thus, in Figure 4.11, $\underline{M} = +\{Up(h_{CU})(\underline{Veh2}), Up(f_{CU})(\underline{Veh1}), Up(f_{dU})(\underline{E12})\}$. This computation applied to the example of Figure 4.8 is shown in Figure 4.12.

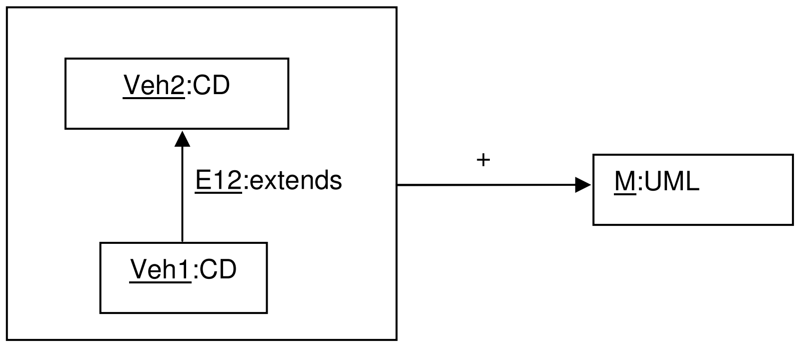


Figure 4.11. The sum of submodels having different types.

Source models	Models transformed to Σ_{UML}

Veh2 = {
class = {Car, Truck},
all other Σ_{CD} sets empty
}
	Up(h_{CU})(Veh2) = {
	class = {Car, Truck},
	all other Σ_{UML} sets empty
	}

Veh1 = {
class = {Hatchback, Sedan, Pickup, Van},
all other Σ_{CD} sets empty
}
	Up(h_{CU})(Veh1) = {
	class = {Hatchback, Sedan, Pickup, Van},
	all other Σ_{UML} sets empty
	}

E12 = {
det.class = { Hatchback, Sedan, Pickup, Van },
gen.class = { Car, Truck },
subClassOf = {⟨Hatchback, Car⟩,
⟨Sedan, Car⟩,
⟨Pickup , Truck⟩,
⟨Van , Truck⟩},
all other $\Sigma_{extends}$ sets empty
}
	Up(h_{dU})(E12) = {
	class = {Hatchback, Sedan, Pickup, Van,
	Car, Truck}
	subClassOf = {⟨Hatchback, Car⟩,
	⟨Sedan, Car⟩,
	⟨Pickup , Truck⟩,
	⟨Van , Truck⟩},
	all other Σ_{UML} sets empty
	}

	M = {
	class = {Hatchback, Sedan, Pickup, Van,
	Car, Truck}
	subClassOf = {⟨Hatchback, Car⟩,
	⟨Sedan, Car⟩,
	⟨Pickup , Truck⟩,
	⟨Van , Truck⟩},
	all other Σ_{UML} sets empty
	}

Figure 4.12. An application of *Up* transformation to perform a sum.

4.5 Summary

In this chapter, we provide a formal account of model types, model relationships and model relationship types. Sort ordered first order logic with transitive closure (FO+) is used as the metamodeling formalism and the metamodel morphism is proposed for relating metamodels in sound and metamodeling formalism independent way. Various characteristics of relationship types are discussed. Among these, partiality is a key property that a relationship type could have and carries the semantic property of logical consequence. It turns out that many of the common relationship types used in software engineering are partiality relationship types and we show these in a taxonomy of abstract relationship types. Moreover, we can define these commonly occurring types as abstract relationship types that are concretized for particular modeling cases by defining the appropriate metamodel. Finally, the *submodelOf* relationship type is explored in detail as a precursor to its use in subsequent chapters.

Chapter 5

Content Criteria

As discussed in Section 2.3, our concern in this thesis is with the content requirements of a model rather than its quality requirements. Thus, in the intent framework, we define the notion of *content criteria* as expressing the modeler's intentions about what information a model ought to contain. In this chapter, we explore the problem of expressing content criteria and make the following contributions. We define a general formal approach to the problem and then develop it in detail for the special case of content criteria for diagrams and submodels. We then show how expressing content criteria can be used to support model comprehension, improve model quality by identifying six new types of defects and support automation and model evolution.

5.1 A general formulation of content criteria

How can we characterize what information a model role *M:T* ought to contain? Our approach will be to assume that *M* is a partial model with respect to some maximal model *G*:*T* (the *base model role*) and then we will characterize the information that ought to be in *M* as the part of *G* that must be contained in *M*. We will express this semantic condition syntactically by characterizing the type of partiality relationship that must hold between realizations of *M* and the corresponding realizations of *G*. To help motivate this, consider the following example from the transportation system scenario.

Say that we identify the need for the model role *MTollGUI:CD* that shows the structural model of the user interface for a toll booth control system we intend to build. Our objective is to formally express what information about the toll booth control system should be found within a realization of *MTollGUI* by expressing role constraints that it must satisfy. To do this, assume that we can identify another model that can act as a surrogate for the toll booth control system. For example, let the surrogate be *MTollProg*:*Java* which represents a (possibly unknown) complete Java program that will implement the toll booth control system. By "complete" we mean that for any Java program $\underline{P}$ that realizes *MTollProg* there exists no Java program $\underline{P1}$ that implements more of the required behaviours of the toll booth control system than $\underline{P}$. Furthermore, for the purposes of this example, we will assume that any information that can be expressed in a class diagram of a system can also be expressed in a Java program of that system. That is, we assume the Java language is at least as expressive as the class diagram language[16].

We can characterize the content of *MTollGUI:CD* relative to the content of *MTollProg*:*Java* since we know that the information about the structure of the user interface must be a found as part of any complete Java implementation. That is, the content criterion of *MTollGUI* can be expressed as a relationship to *MTollProg*

[16] In general, we need the modeling language of the base model to be at least as expressive as the modeling language of the model for which we are defining content criteria.

containing the role constraints that must hold between these model roles. We will call this relationship $CC_{MTollGUI}$ and make the following observations about it:

O1. Since we know that *MTollGUI* is a partial model relative to *MTollProg*, this means that $CC_{MTollGUI}$ is a type of partiality relationship and guarantees the semantic condition $MTollProg \models MTollGUI$ as discussed in Chapter 4. This limits $CC_{MTollGUI}$ to relationships such as submodel, abstraction, etc. (or some combination of these).

O2. The fact that we say that the toll booth control system has a model *MTollGUI* means that we are assuming that it has a user interface - otherwise a model showing the user interface would be meaningless. This implies that the relationship $CC_{MTollGUI}$ imposes some constraints on *MTollProg* that follow from the fact that there is a user interface. For example, this may be the constraint that each realization of *MTollProg* must have a main window class that will contain the user interface. We call such constraints, the *preconditions* due to the required existence (i.e., the existential intent) of *MTollGUI*.

O3. $CC_{MTollGUI}$ satisfies the following uniqueness principle: for each realization of *MTollProg* that satisfies the preconditions, there must exist a unique realization of *MTollGUI* that satisfies the constraints of $CC_{MTollGUI.}$. Thus, $CC_{MTollGUI}$ is a kind of transformation from the set of realizations of *MTollProg* that satisfy the preconditions to the set of realizations of *MTollGUI* .

Observation (O1) follows from our assumptions that *MTollProg* is complete, that the GUI is part of the system and that a Java program can express anything that a class diagram can. Observation (O2) follows from the fact that simply asserting the existential intent for a model can be enough to say something about the subject. This is a case of the tangling that can occur between the intent about the subject and the intent about the model as discussed in Section 2.1.3. To justify observation (O3), consider the following thought experiment. Assume we have a realization *MTollProg*. Now assume we ask the modeler to create the corresponding realization of *MTollGUI* according to what they think it should be (i.e., according to what they believe is the intent of the definer of *MTollGUI*). Since they will always produce *some* specific model *MTollGUI*, they must have followed some selection principle for deciding on a particular class diagram amongst all possible class diagrams. Furthermore they could follow this principle for any realization of *MTollGUI* that satisfies the preconditions. This suggests that $CC_{MTollGUI}$ is a transformation and it is an expression of this selection principle.

Before we fully accept this principle, let us consider some nuanced aspects of it. First note that when we say that the realization of *MTollGUI* is unique we always mean unique "up to isomorphism" since two models are considered the same if they differ only in the identity of their symbols but retain all attributes and relationships of these symbols.

Second, consider that we naturally expect the modeler intent about content to correspond to the "optimal" realization that would satisfy the purpose. In this case an optimal realization of a model role satisfying its purpose is one that contains all the information required by the purpose and no information not required by the purpose. However the purpose alone may underdetermine the realization of a model since there may be many possible realizations that are equally optimal.

For example, say that the purpose of *MTollAttr*:*CD* is to show that there exist toll related attributes in *MTollProg*. In this case, a class diagram that shows a single toll-related attribute is optimal for satisfying the purpose. Now, if there are several toll related attributes then there will be several corresponding class diagrams that are all optimal. In this example, each of the alternative optimal realizations are semantically distinct. Another case of underdetermination is where the purpose uniquely determines the required information but there are multiple semantically equivalent ways to express this in the modeling language. In both these situations, the modeler must still select one realization among the alternate equivalent ones and so the modeler intent must carry additional constraints in order to collapse the underdetermined choice. Thus, although model purpose may not select a unique realization, the modeler intent will and so this must appeal to factors that go beyond model purpose when necessary.

Third, note is that even though the content criterion expresses the selection principle, this may always not be formalizable. For example, model role *MTollComplex* may be intended to show the parts of the toll system that are deemed to be complex and the selection principle here may be a subjective assessment of complexity by the modeler. Although it is not formalizable it is still correct to say that there is content criterion that selects a unique realization of *MTollComplex* for each realization of *MTollProg*. Our interest in this thesis lies primarily with those content criteria that are formalizable since these are amenable to automation and tool support.

Fourth, consider the case where the content criterion is formalizable but there is not enough information in the base model to do it. For example, let *MTollTeamA* show the classes of *MTollProg* that team *A* must implement. Although there is a unique realization of *MTollTeamA* that could be extracted from any realization of *MTollProg*, the criteria for extracting this requires additional information about which team each class of *MTollProg* is assigned to. In this case we say that the content criterion is underdetermined by the base model. Deciding whether or not this is an acceptable situation depends on the usage context and is one way in which content criteria could be used to assess model quality. We discuss this in greater detail for the case of submodels in Section 5.2.

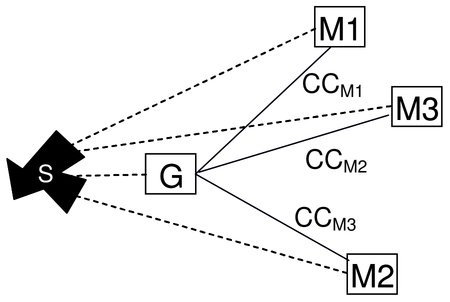
Finally, note that the uniqueness principle depends on the assumption that a base model G *can* exist, even if it is not actually created. In the case of *MTollGUI* this is a reasonable assumption since it is a model at the design stage of software development and there is a well defined resultant artifact that it describes a part of. At the requirements phase, however, this assumption may not be reasonable. When there are multiple stakeholders and they express incommensurable viewpoints on the requirements, then there is no single global base model that can combine these partial models. Thus, we recognize that our approach to content criteria is only applicable to contexts when this assumption holds.

5.1.1 Formalization

Figure 5.1 shows the general case when there are several partial models. The base model role $G{:}T_G$ acts as a surrogate of the thing *S* being modeled and represents the complete T_G

model of *S*. This means that every realization $\underline{G}$ satisfies the condition that for every other T_G model $\underline{M}$ that says true things about *S*, $\underline{G} \vDash \underline{M}$.

The content criteria for all model roles are defined relative to *G*. Specifically, the content criterion CC_{Mi} of *Mi* is defined as follows.



CC_{Mi} means $Pre_{Mi}(G) \wedge Mi = F_{Mi}(G)$

Where

$F_{Mi};T_G \rightarrow T_{Mi}$ is a transformation

$Pre_{Mi} \subseteq T_G$ is the precondition

$G \vDash Mi$ is the partiality condition

Figure 5.1. A general approach to defining content criteria.

Definition 5.1. *Content criterion.* Given model roles $Mi{:}T_{Mi}$ and $G{:}T_G$, the content criterion CC_{Mi} of *Mi* is defined as the constraint $Pre_{Mi}(G) \wedge Mi = F_{Mi}(G)$ where

- *G* is the base model
- Pre_{Mi} (the precondition) is a property over T_G models and is an existential constraint for the role *Mi*
- $F_{Mi}{:}T_G \rightarrow T_{Mi}$ is a partial transformation that is defined for all T_G models that satisfy Pre_{Mi} and satisfies the partiality condition:

$$\forall \underline{m}{:}\mathrm{T_G} \cdot Pre_{Mi}(\underline{m}) \Rightarrow \underline{m} \vDash \mathrm{F_{Mi}}(\underline{m})$$

Thus, F_{Mi} is a transformation that defines a particular realization of *Mi* for each realization of *G* that satisfies the precondition and the realization of *Mi* is guaranteed to be a partial model relative to *G*. In general, the problem of expressing the content criterion of *Mi* is exactly the problem of finding a property that uniquely identifies a T_{Mi} model among all T_{Mi} models that are partial relative to *G*. Note that although we refer to a base model *G* it may never actually be created within a modeling project. In this case, it is considered to be "unrealized" – the idea that a model role may be unrealized within a project is addressed in Chapter 6.

In Section 5.2 we elaborate the detailed implications of this approach to content criteria for the special case where the partiality relation is restricted to the submodel relation. However, in order to strengthen the intuitions behind this we first sketch some examples that are not submodels.

5.1.2 Example 1: Content criteria of a sequence diagram

Once again we use the transportation system example but this time we want to define the content criteria for the sequence diagram role *BuyMonthlyTicket:SD* that shows how to purchase a monthly toll ticket. Assume that *G* = *TransSystemControl:Java* representing a complete implementation of the system controller (i.e., it includes the toll ticket purchase functionality). A sequence diagram can express a set of positive and a set of negative traces [UML2]. Now consider the partiality relationship type between sequence diagrams and Java programs that holds when the sequence diagram depicts a set of traces in the

Java program[17]. In order to define CC_{M1} we must find a property among all sequence diagrams of *TransSystemControl* that only *BuyMonthlyTicket* satisfies. We can proceed by asking what system traces should be included and what should be excluded and define the informal requirements due to the purpose as follows:

- *BuyMonthlyTicket* should include only traces that begin with a customer initiating the purchase of a monthly ticket.
- *BuyMonthlyTicket* should include only "happy path" traces. We define a happy path trace as one that ends in the customer actually getting a ticket.
- *BuyMonthlyTicket* should include *all* traces that satisfy the above conditions

Using more formal language we could say the following:

$F_{BuyMonthlyTicket}:Java \rightarrow SD$:=

For objects *Tk*:*MonthlyTicket* and *C*:*Customer*, transformation $F_{BuyMonthlyTicket}$ extracts from *TransSystemControl* the complete set of distinct traces of object calls satisfying condition that it begins with *Tk.buy* being invoked by *C* and ends with *Tk.sold* = *TRUE.*

$Pre_{BuyMonthlyTicket}$:=

- *TransSystemControl* contains classes *Customer* and *MonthlyTicket* with the latter having method *buy* and attribute *sold*:*boolean*

[17] We do not define this partiality relationship type here but it should be clear that such a relationship type is possible since sequence diagrams are intended to be used to depict traces in programs.

- When *TransSystemControl* runs it is possible to have an instance of *MonthlyTicket* and for all instances *t* of *MonthlyTicket*, if *t.buy* is invoked then eventually it is possible that *t.sold = TRUE*

This is a case where the purpose underdetermines the content criteria because while no specific objects and classes where specified in the requirements, some specific choices were made in the content criteria (i.e., object *Tk*, class *MonthlyTicket*, etc.). While these choices are necessary to specify a particular sequence diagram, they are semantically equivalent to other alternatives that could have been made (e.g., alternate naming). Note that we assume that the set of traces extracted by F_{M1} is expressible by a single sequence diagram – this is a validity condition that F_{M1} must satisfy. Whether or not it is desirable to do so is a different issue and this is addressed by the part of the intent framework dealing with decomposition in Chapter 7.

5.1.3 Example 2: Content criteria of a state machine model

In example 1, although *BuyMonthlyTicket* was a different type of model than *TransSystemControl*, it represented the same level of abstraction – i.e., the traces expressed by *BuyMonthlyTicket* are all directly visible as paths through the program *TransSystemControl*. In this example, we consider the content criteria for the state machine[18] *CustMonthlyTicket:SM* that shows the customer's view of the procedure for purchasing a monthly toll ticket. This time we assume *G* represents the most detailed

[18] There are many state machine modeling languages (e.g., finite state machines, StateCharts, etc.). We don't specifiy the particular one here but we assume the existence of states and transitions.

state machine that expresses the procedure for purchasing a monthly toll ticket. Note that this model is sufficiently complete to contain the information in *CustMonthlyTicket* and has its own content criteria defined relative to *TransSystemControl:Java*.

Intuitively, the content criterion of *CustMonthlyTicket* says that it is the most refined state machine abstraction of *G* in which the states and transitions are distinguishable to a customer. Here we define a state machine abstraction as a partiality relationship in which sets of states (transitions) in the more refined state machine are mapped to particular states (transitions) in the more abstract state machine that semantically represent the disjunction of the elements in the refined sets. We define customer distinguishability as follows:

- Two states s_1 and s_2 are distinguishable by the customer iff there is some customer visible property that holds in s_1 and not in s_2.
- Two transitions t_1 and t_2 are distinguishable by the customer iff the start states are distinguishable or the end states are distinguishable or t_1 involves a different customer interaction than t_2.

We can define $F_{CustMonthlyTicket}:SMD\rightarrow SMD$ by saying that $F_{CustMonthlyTicket}(G)$ is the state machine abstraction of *G* that is constructed by disjunctively combining the *G*-states that are customer indistinguishable and disjunctively combining the *G*-transitions that are customer indistinguishable. We can use this definition to check some cases of non-conformance even though *G* is unknown. For example, in Figure 5.2, state machine *M2* is definitely not conformant because it distinguishes some states that are not distinguishable to the customer. It can be "repaired" by producing the abstraction *M1* that does satisfy the indistinguishability criterion.

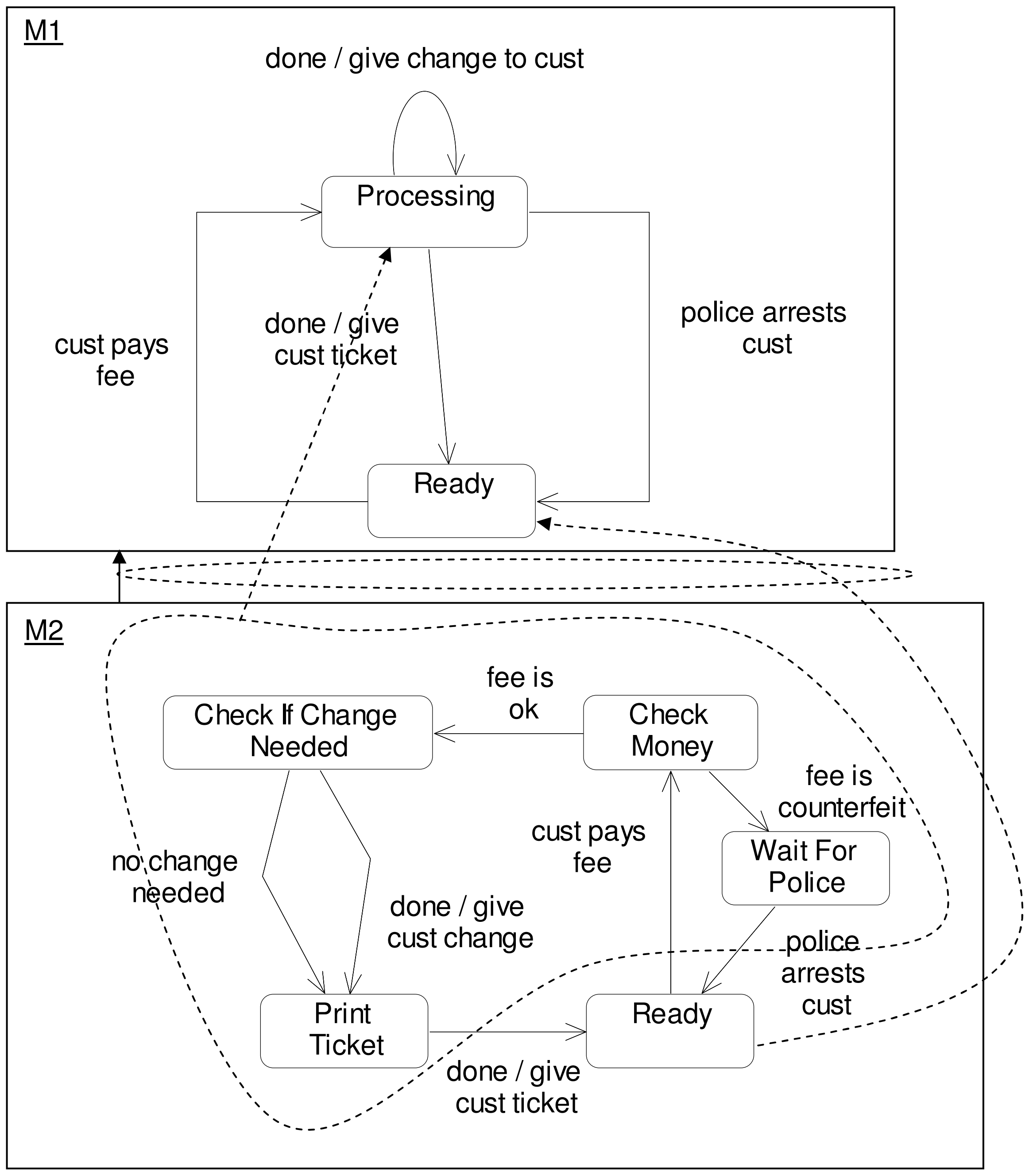


Figure 5.2. Examples of candidate versions of *CustMonthlyTicket.*

5.1.4 Aspects of modeler intent that content criteria do not address

Although the set of content criteria of a model captures much of the modeler intent about the model, it is missing some aspects. In particular, since it only contains role constraints that hold between a model role and a base model role *G*, it does not address the following:

1. How particular partial models of *G* are intended to be related to each other within *G*
2. How different (complete) models *G1*, *G2, …* are intended to be related to each other
3. How *G* is intended to be hierarchically decomposed into partial models

Cases (1) and (2) are addressed using relationships between models as discussed in Chapter 4. Case (3) is addressed by decomposition criteria in Chapter 7.

5.2 The content criteria of submodels

We now turn to the investigation of the content criteria for a common class of model role: a model role that is intended to be a particular submodel of the base model role. We will refer to this type of model role as a *view* and the content criterion of the view characterizes this submodel. This is an important special case, since in practice, a model is often manifested as a set of diagrams and as discussed in Chapter 4, we take a diagram to be a submodel. We now extend this and state that a diagram is more specifically, a view, since the purpose of a diagram is to present a particular part of the model.

For example, in the UML model of the transportation system, one class diagram may be intended to show the different types of vehicle classes in the transportation system while another is intended to show the different classes involved with road tolls. The content criteria of diagrams are not typically modeled and even when they are made explicit, it is only done through informal means such as comments or as part of the name of the diagram. However, as discussed in Chapter 1, the explicit and precise expression of content criteria is a fruitful activity because it improves model quality, model comprehension and provides support for automation and model evolution. We first illustrate the content criteria of submodels using the transportation system example and then develop the formal details.

5.3 Illustration

Consider the diagram *DTollTicket:CD* from the transportation system example as shown in Figure 5.3. The intent of this class diagram is "to show the information relating to toll tickets." Assume that this intent implies that the following constraints should hold between *DTollTicket* and the UML model role *TransportationSystem*:

TT1. All and only the classes *TollTicket*, its direct or indirect subclasses and the classes related to *TollTicket* by a navigable association are included in *DTollTicket.*

TT2. All and only the attributes of *TollTicket* are included in *DTollTicket*.

TT3. All and only the navigable associations from *TollTicket* are included in *TollTicket*.

These constraints constitute the content criteria for *DTollTicket*. Expressing the content criteria explicitly and precisely is useful for all three of the modeling roles: definer, producer and consumer. The definer can articulate the intent of the diagram and effectively communicate this to the producer. The producer can use this to assess whether they are conforming to this intent by making sure that nothing is included that does not belong in the diagram and that everything that does belong is included. The consumer can use the constraints to properly interpret the content of the diagram. For example, without (TT3) it may not be clear to the consumer whether or not the diagram is showing all the associations between the classes or that there may be more that have been omitted from the diagram. Thus, while diagrams are typically assumed to be incomplete relative to the model, the content criteria provide the consumer with information about the ways in which the diagram *is* complete.

If formalized, the content criteria are useful for automated support of the management of the diagram content in order to ensure that the intent of the diagram is maintained as the specification model evolves. For example, if the producer adds a class to the model via the realization of *DTollTicket* and does not make it a subclass of *TollTicket*, this violates constraint (TT1) and can be flagged as such. On the other hand, if a subclass of *TollTicket* is added to the realization of *TransportationSystem* by some other means, such as manually through another diagram, change propagation, round-trip engineering, etc., the violation of constraint (TT1) can trigger the "repair" action of adding it to diagram *DTollTicket*.

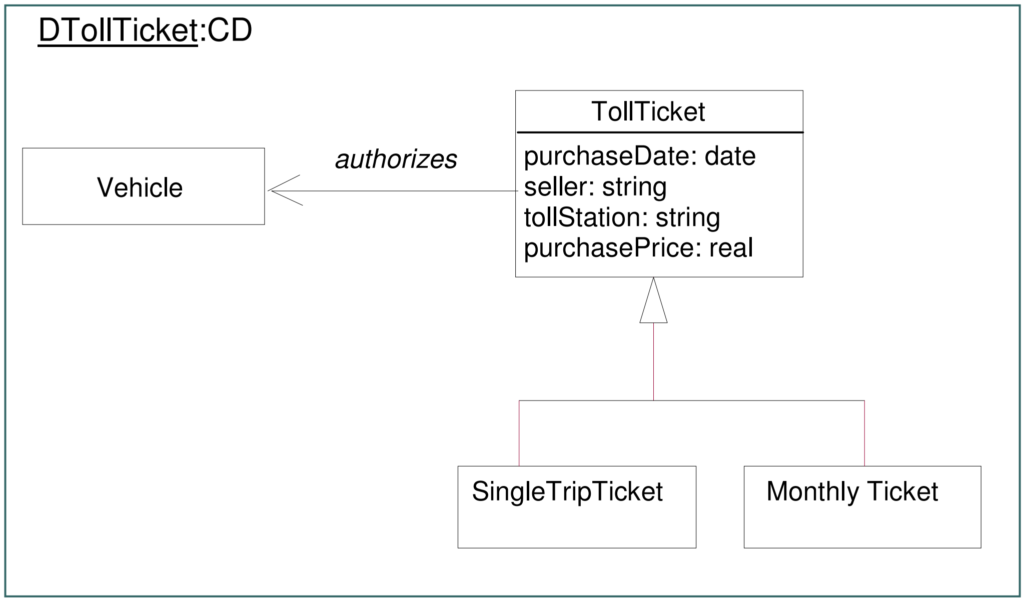


Figure 5.3. A diagram showing the details of the class TollTicket.

The precondition of *DTollTicket* is that the class *TollTicket* exists in *TransportationSystem* since without the presence of this class, none of the criteria (TT1), (TT2) or (TT3) would be well defined. At a more conceptual level, it does not make sense to have a diagram “to show the information relating to toll tickets” without assuming that some implementation of the concept of “toll ticket” exists in the model. Thus, a realization of *TransportationSystem* has a corresponding realization of *DTollTicket* iff it contains the class *TollTicket*.

Now consider diagram *DTollPrice* shown in Figure 5.4. The intent of this diagram is to show the information related to toll price within *TransportationSystem*. We interpret this as the following content criteria:

TP1. Include only the class *Vehicle* and its direct subclasses and the class *TollTicket* and all of its subclasses.

TP2. Include all and only the attributes of the included classes that affect toll price.

TP3. Include all associations between the included classes.

Like the content criteria for *DTollTicket*, this lists a set of diagram *inclusion constraints* that together extract a unique submodel for each realization of *TransportationModel*; however, unlike *DTollTicket*, it is unclear whether the truth of these conditions can be fully determined from the content of the model *TransportationSystem* alone. In particular, constraint (TP2) requires that attributes be included only if they affect toll price but no means for determining this property is evident. This highlights another benefit of articulating the content criteria – they expose contextual information that is assumed when interpreting the diagram and that may be missing from the base model.

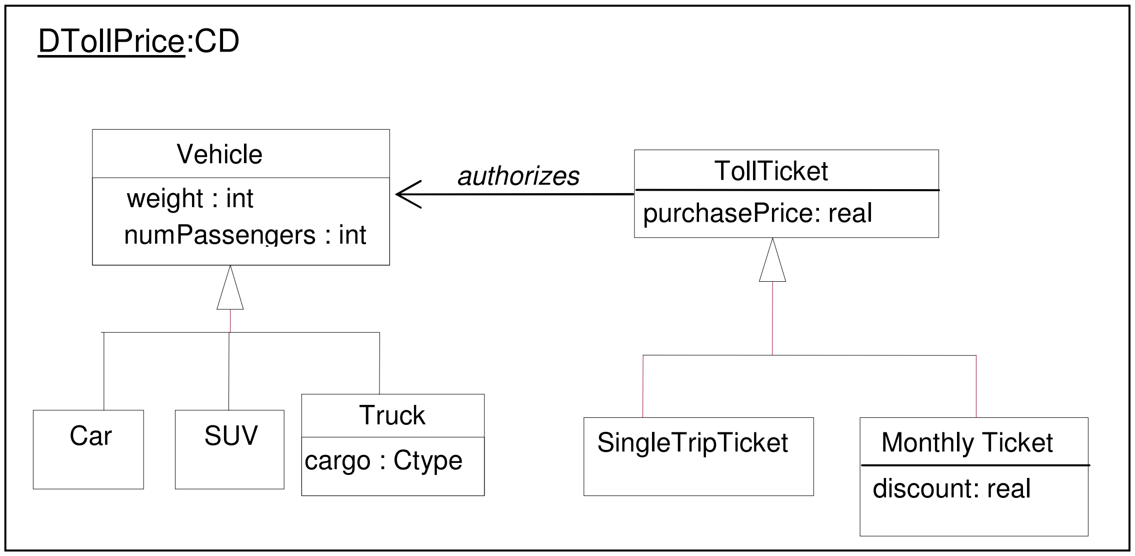


Figure 5.4.Transportation system diagram dealing with toll ticket price.

One response to this is to extend the model to include this information. In this case, this could be done in several ways ranging from formal to informal including: including an OCL expression in the model that computes toll price and then determining what attributes are used in this expression, adding a stereotype to attributes that indicates when they affect toll price, using a naming convention on attributes to indicate when they affect toll price, annotating the classes and attributes with comments, etc.

Another response to this situation is to treat the inclusion of an attribute in the diagram as an *assertion* that the attribute affects toll price. In this case, diagrams are not only used as views on the model but also to extend the model itself. Since diagrams are typically considered to only be relevant to the presentation of a model and not its content, this approach has the drawback that the information may not be preserved in further refinements of the model (e.g., into the code) and hence would be lost. This suggests that the first response may be preferred if this information is needed in downstream processes – i.e., missing context information should be viewed as a case of model incompleteness.

To summarize what we have seen in this illustration:

- The content criteria of a view (such as a diagram) can be expressed as inclusion constraints that together pick out a unique submodel for each realization of the base model.
- There may be preconditions that must hold for the inclusion conditions to be well defined and hence are required for the existence of the view.

- The inclusion constraints may be not be fully expressible in terms of information in the base model.

We now turn to the formalization of these concepts and then discuss their implications.

5.4 Formalization

We follow the general formulation for content criteria introduced in Section 5.1.1. First consider partial transformations of the form $F{:}T \times T_1 \times \ldots \times T_n \rightarrow T_0$ where the following holds[19]:

$$\forall \underline{m}{:}\mathrm{T}, \underline{m}_1{:}\mathrm{T}_1, \ldots, \underline{m}_n{:}\mathrm{T}_n \cdot \mathrm{DEFINED}(F(\underline{m}, \underline{m}_1, \ldots, \underline{m}_n)) \Rightarrow \mathit{submodelOf}(F(\underline{m}, \underline{m}_1, \ldots, \underline{m}_n), \underline{m})$$

These extract a submodel of $\underline{m}$ and so satisfies the partiality condition. We will refer to this kind of transformation as an *extractor*[20]. Since an extractor is a partial function we can define the precondition predicate $Pre_F \subseteq T \times T_1 \times \ldots \times T_n$ as the condition under which F is defined. Thus, we have:

$$\forall \underline{m}{:}\mathrm{T}, \underline{m}_1{:}\mathrm{T}_1, \ldots, \underline{m}_n{:}\mathrm{T}_n \cdot \mathit{Pre}_F(\underline{m}, \underline{m}_1, \ldots, \underline{m}_n) \Rightarrow \mathit{submodelOf}(F(\underline{m}, \underline{m}_1, \ldots, \underline{m}_n), \underline{m})$$

We now define content criteria in terms of an extractor.

[19] The 2nd order predicate DEFINED($F(x)$) used for partial functions is *TRUE* iff $F(x)$ has a value for x.

[20] From a database perspective, an extractor is similar to a parameterized query except that the query is not defined for all database or parameter values and the parameters can include types of information not found in the database.

Definition 5.2. ***Content criterion of a view.*** Given model roles *Msub*:*Tsub* and *M*:*T*, if *Msub* is a view of *M*, then it has an associated content criterion of the form:

$$CC_{Msub} := Pre_F(M, M_1, \ldots, M_n) \wedge Msub = F(M, M_1, \ldots, M_n)$$

where *F* is an extractor and Pre_F is defined as:

$$\forall \underline{m}:T, \underline{m}_1:T_1, \ldots, \underline{m}_n:T_n \cdot Pre_F(\underline{m}, \underline{m}_1, \ldots, \underline{m}_n) \Leftrightarrow \mathrm{DEFINED}(F(\underline{m}, \underline{m}_1, \ldots, \underline{m}_n))$$

Since Pre_F is defined in terms of *F*, we can use the abbreviated form for content criterion:

$$CC_{Msub} := F(M, M_1, \ldots, M_n)$$

Here, *M* is the base model role and the other model roles M_1:T_1, …, M_n:T_n are referred to as the *generators* of *Msub*. We think of the generators as jointly “generating” the view of the base model. This intuition is elaborated further in Section 5.4.3.

From an existential perspective, the view *Msub* can exist only when the views *M*, M_1, …, M_n exist and $Pre_F(M, M_1, \ldots, M_n)$ holds. From a conformance perspective, a tuple of realizations for *Msub*, *M*, M_1, …, M_n satisfy this constraint iff the realization of *Msub* is the same submodel of *M* as $F(M, M_1, \ldots, M_n)$. When this constraint is satisfied we say that *Msub* satisfies the intent expressed by its content criteria.

Note that the same extractor can be used to express different content criteria. For example, we may say that CC_{M1} := *subsOf*(*TransportationSystem*, *MVehicle*) or CC_{M1} := *subsOf*(*TransportationSystem*, *MPayment*). In the first case we are saying that *M1* is intended to be the submodel of *TransportationSystem* that consists of the classes in model

MVehicle and its subclasses. In the second case we are saying that *M1* is intended to be the submodel of *TransportationSystem* that consists of the classes in model *MPayment* and its subclasses. These express different intents for the content of *M1*. This is even the case if it turns out that the content of *MVehicle* is the same as the content of *MPayment* (and so the content of *M1* is the same) for some or all realizations of *TransportationSystem.*

We can define the content criteria as a partial transformation formally using the relationship type approach described in Chapter 4. Thus, it is a relator metamodel on the combined signatures of *Tsub, T*, T_1, …, T_n that includes the metamodels of these and additional constraints showing how these models are related. As an example, we will express content criteria $CC_{DTollTicket}$:= F_{toll}(*TransportationSystem*) of *DTollTicket* where extractor F_{toll} is defined as follows.

F_{toll}(M:UML): CD = Out.CD + M.UML + (1)

subsort Out.Class ≤ M.Class, Out.Association ≤ M.Association, Out.Attribute ≤ M.Attribute (2)

constraints (3)

// precondition

∃ *mc*:M.Class · M.className(*c*) = "TollTicket" ∧ (4)

// inclusion constraints defining extractor

(∀*c*:M.Class · (∃*c1*: Out.Class · *c1* = *c*) ⇔ ((*c* = *mc* ∨ TC(M.subClassOf(*c*, *mc*) ∨ (5)
(∃*a*:M.Association · M.endClass(*a*) = *c* ∧
M.startClass(*a*) = *mc*))) ∧

(∀*a*:M. Association · (∃*a1*: Out. Association · *a1* = *a*) ⇔ M.startClass(*a*) = *mc*) ∧ (6)

(∀*a*:M. Attribute · (∃*a1*: Out. Attribute · *a1* = *a*) ⇔ M.attrClass(*a*) = *mc*) ∧ (7)

(∀*c1*, *c2*: Out. Class · Out.subClassOf(*c1*, *c2*) ⇔ M.subClassOf(*c1*, *c2*)) ∧ (8)

$(\forall a\text{: Out. Attribute} \cdot \text{Out.attrClass}(a) = \text{M.attrClass}(a)) \wedge$ (9)

$(\forall a\text{: Out. Association} \cdot \text{Out.startClass}(a) = \text{M.startClass}(a)) \wedge$ (10)

$(\forall a\text{: Out. Association} \cdot \text{Out.endClass}(a) = \text{M.endClass}(a)) \wedge$ (11)

$(\forall c\text{: Out. Class} \cdot \text{Out.className}(c) = \text{M.className}(c))$ (12)

Recall that line (2) indicates that F_{toll} imports the signature and constraints for *CD* and *UML* and to avoid name clashes these are "namespaced" with "Out" and "M", respectively. *Out* represents the class diagram that is the result of applying $F_{toll}(M)$. Line (3) asserts that the elements in *Out* are subsets of the elements in *M*. Line (4) asserts the precondition that *M* must contain a class named *TollTicket*.

The inclusion constraints are what must hold between the content of *Out* and *M* and are defined in the scope of the precondition so that the variable *mc* is bound. These encode the constraints for diagram *DTollTicket* expressed in words in Section 5.3. Constraint (TT1) is expressed by (5), constraint (TT2) is expressed by (7) and (9) and constraint (TT3) is expressed by (6), (10) and (11). Note that an instance of *Out* cannot exist if the precondition does not hold in *M* and this encodes the fact that the inclusion constraints are not evaluable without *mc*. However, if it does hold then *Out* is uniquely determined by *M* and so F_{toll} is a partial transformation.

The content criteria above are written in a standardized form. If we assume that we are expressing content criteria $CC_{Msub} := F(M, M_1, \ldots, M_n)$ where $Msub$:$Tsub$ and M:T, M_1:T_1, …, M_n:T_n, then the form is:

```
F(M:T):Tsub =  Out.Tsub + M.T + $M_1.T_1$ + … + $M_n.T_n$
  subsort  for each $S \in sorts_{Tsub}$, $Out.S \leq M.S$
  constraints
  // precondition
    precondition $\wedge$
  // inclusion constraints
    for each $S \in sorts_{Tsub}$,
      $(\forall x{:}M.S \cdot (\exists x1{:}Out.S \cdot x1 = x) \Leftrightarrow Q_S(x)) \wedge$
    for each function $H{:}S_1 \times \ldots \times S_{n-1} \rightarrow S_n \in func_{Tsub}$
        $(\forall\ x_1{:}Out.S_1, \ldots, x_n{:}Out.S_{n-1} \cdot$
            $Out.H(x_1, \ldots, x_{n-1}) = M.H(x_1, \ldots, x_{n-1})) \wedge$
    for each predicate $P{:}S_1 \times \ldots \times S_n \in pred_{Tsub}$
        $(\forall\ x_1{:}F.S_1, \ldots, x_n{:}F.S_n \cdot$
            $Out.P(x_1, \ldots, x_n) \Leftrightarrow M.P(x_1, \ldots, x_n) \wedge\ Q_P(x_1, \ldots, x_n)) \wedge$
    TRUE
```

In each inclusion constraint, Q_i represents a formula called the *inclusion condition* that may involve bound variables in the precondition. Intuitively, the inclusion conditions pick out the parts of *M* that belong in *Msub* and provides a systematic way of defining content criteria. Based on this form, the content criteria can be seen to consist more simply of the precondition and a set of inclusion conditions. For example, the extractor representing the content criteria for *DTollTicket* could be expressed more compactly as the set of definitions:

```
$F_{toll}$(M:UML): CD := [
  precondition := $\exists\ mc{:}M.Class \cdot M.className(c)$ = "TollTicket",
  $Q_{Class}(c)$ :=   $(c = mc \vee TC(M.subClassOf(c, mc) \vee$
                    $(\exists a{:}M.Association \cdot M.endClass(a) = c \wedge$
                                        $M.startClass(a) = mc$ )),
  $Q_{Association}(a)$ := $M.startClass(a) = mc$,
```

$Q_{Attribute}(a)$:= M.attrClass(a) = mc

]

For further compactness, we assume that by default if a *sort* inclusion condition is not mentioned then it must be *FALSE* (i.e., no instances are included) and if a *predicate* inclusion condition is not mentioned then it must be *TRUE* (i.e., all instances are included). When the content criteria CC_{Msub} is expressed in terms of inclusion conditions it is clear that for every *M* that satisfies the precondition, the inclusion constraints specify a unique submodel *Msub* of *M*. This is because there is a constraint for each sort and predicate of *Msub* that determines exactly what subset of these from *M* are included in *Msub*[21]. To ensure that the resulting submodel *F*(*M*) is also always a well formed *Tsub* model – i.e., that it satisfies the constraints Ψ_{Tsub} – we must add the following validity conditions:

Consistency. $\quad M.\Psi_T \cup Out.\Psi_{Tsub} \cup \Psi_F \nvDash FALSE$

Well-formedness. $\quad M.\Psi_T \cup \Psi_F \vDash Out.\Psi_{Tsub}$

Here, $M.\Psi_T$ and $Out.\Psi_{Tsub}$ are the imported versions of the constraints of *T* and *Tsub* found in the relator metamodel for *F* and Ψ_F are the set of subsort, precondition and inclusion constraints. The consistency condition says that all of the constraints in *F* must be consistent and well-formedness condition guarantees that the submodel extracted by the content criteria from each *T*-model is a well formed *Tsub*-model.

[21] A function instance must always included in *Msub* if the input elements are included and the function is in *Tsub*.

5.4.1 Generalizing content criteria

Note that although F_{toll} is defined as a relationship type it is actually a very specialized partial transformation because it makes reference to the class *TollTicket*. This suggests that this relationship type would only ever be used to express the content criteria of *DTollTicket*. However, in many cases, it is possible to generalize the diagram intent and content criteria by replacing certain constants by parameters representing generators. For example, we can generalize the intent for diagram *DTollTicket* by defining the extractor *classDetails*(*M*:*UML*, *C*:*One*[*class*]) as:

```
classDetails(M:UML, C:One[class]) CD := [
    precondition := ∃x:C.Class, mc:M.Class · mc = x,
    $Q_{Class}(c)$ :=  (c = mc ∨ TC(M.subClassOf(c, mc) ∨
                     (∃a:M.Association · M.endClass(a) = c ∧
                                       M.startClass(a) = mc )),
    $Q_{Association}(a)$ := M.startClass(a) = mc,
    $Q_{Attribute}(a)$ := M.attrClass(a) = mc
]
```

classDetails is identical to F_{toll} except that it takes the class for which it shows the details as an argument. *classDetails* is a generic extractor in the sense that for any class *C* it extracts the submodel from *M* that details *C* in the particular way described by the inclusion conditions. Given this generic extractor we can more simply and compactly express the content criteria of *DTollTicket* as the expression $CC_{DTollTicket}$:= *classDetails*(*TransportationSystem*, *Class*(*TransportationSystem*, "*TollTicket*")).

An obvious benefit of using parameterized extractors is reuse since it reduces the incremental effort to define the content criteria for different diagrams when the content criteria have the same form. In addition, a predefined set of parameterized extractors can be composed algebraically to define more complex content criteria at the "macroscopic level" without having to specify inclusion conditions. For example, if we want view *DVehToll* to represent the class detail of both *Vehicle* and *TollTicket* we can express this as $CC_{DVehToll}$:= *classDetails*(*TransportationSystem*, *Class*(*TransportationSystem*, "*TollTicket*")) + *classDetails*(*TransportationSystem*, *Class*(*TransportationSystem*, "*Vehicle*"))].

There are other benefits of parameterized extractors as well. They can be used to define a library of common view types, like *classDetails* that are domain-independent and hence are meaningful in many contexts.

5.4.2 Content criteria and naming

As discussed in Section 3.2, the name is typically the source of informally expressed information regarding the modeler's intent about the model's content. Thus, we can view content criteria as a formalized version of the model name and so there ought to be a semantic correspondence between the name and the content criteria. In particular, the name should reflect some meaningful abstraction of the content criteria. For example, a better name for *DTollTicket* might be "The details of class TollTicket." Note that the content criteria would typically contain more information about the content than the name. For example, we could also generate a name like "The attributes, navigable associations and subclasses of class TollTicket" but this may be too unwieldy for a name.

When this correspondence between name and content criteria does not hold, then it negatively impacts the quality of the model because the name is misleading. We discuss this further in the context of defects detectable using content criteria in Section 5.5.2. The correspondence between name and content criteria also suggests another possibility: that it should be possible to generate a "good" name for a model from its content criteria. Furthermore, with generic content criteria as in Section 5.4.1, the name generation could be reused in many contexts. For example, *classDetails*(*M*, *C*) can have the corresponding name schema "The details for class *C*". The benefit of generating the name from the content criteria is that names can be assigned in a consistent way (e.g., naming conventions enforced) and it ensures the semantic correspondence holds.

5.4.3 The role of generators and detail views

We have used the term *generator* to represent the parameters of an extractor. This is because an extractor should be thought of as a function that "generates" a view of the base model for each combination of generator values. From a pragmatic perspective, we expect to use extractors where the resultant view has a meaningful relationship to its generators and so this relationship is useful for expressing modeler intent about the view.

One kind of relationship between the generators and the view is when the generator defines an "aspect"[22] of the base model. For example, the extractor *Proj*(M:T, Σ) that returns the projection of M on the subsignature $\Sigma \subseteq \Sigma_T$ is this kind of extractor. Another

[22] We are using the term "aspect" more loosely than the specialized meaning in aspect oriented modeling.

common kind of relationship between generators and the view is *aboutness*. Thus, although the view is of the base model it is often the case that it is about the generators in some way – i.e., the generators represent the subject of the view.

A special case of this occurs when the view shows some details about another model. In this case, we call it a *detail view*. For example, *DTollTicket* is a detail view since it shows the class details for class *TollTicket*. In general, we say that *M1* is the *F1* detail view of *M* for some model *Mg* when its content criterion has the form $\mathrm{CC}_{\mathrm{M1}} := F1(M, Mg)$ and *F1* shows some details associated with *Mg*. Detail views form a natural "refining" navigation path between views because a consumer can "expand" an element from one view into its detail view(s). This style of navigation has been leveraged by modeling tools such as GME [LMB01] and MetaEdit+ [MEdit], although views are not formalized.

Detail views give rise to a corresponding class of relationship types that can hold between views that we call *detailOf* relationships. This was presented as a common abstract relationship type in Figure 4.6. Informally, we say that *M1* is a *detailof M2* iff *M1* is a detail view for some element in *M2*. Formally we define it as follows.

Definition 5.3. *DetailOf relationship*. Given an extractor $F1: T \times T_E \rightarrow T1$ and a submodel type *T2*, where $\Sigma_{T_E} \subseteq \Sigma_{T2} \subseteq \Sigma_T$, then we can define the *F1*(*E*)-*detailOf*(*M1*:*T1*, *M2*:*T2*) relationship type as follows:

$$\mathrm{F1(E)}\text{-}detailOf(\mathrm{M1, M2}) \Leftrightarrow (CC_{M1} := F1(M, E)) \wedge \exists x{:}\mathrm{M2}.T_E \cdot x = \mathrm{E}$$

That is, *M1* is an *F1* view of *M* generated by *E* and *M2* contains *E*. Typically, *M2* is also a view of *M*. Since *detailOf* relationship types can hold between views, they provide a convenient way to express the intended detailing relationships that define the structure of a set of views.

5.5 Value of content criteria

5.5.1 Impact on model comprehension

As discussed in Chapter 1, as a linguistic entity, a model has a pragmatic aspect in addition to syntactic and semantic aspects. The intent of the modeler about what information the model should contain is a source of pragmatic information that can affect a consumer's interpretation of the model. Here we consider the ways in which the content criteria of a submodel can affect the interpretation of it.

Based on the general form of the content criterion described in Section 5.4 for the arbitrary submodel *Msub* of *M*, if the content criterion is satisfied then we know the following facts:

F1. The precondition holds for *M*, Q_S holds for all elements of sort *S* in *Msub* and Q_P holds for all instances of predicate *P* in *Msub*.

F2. *Msub* contains all the elements of *M* of sort *S* for which Q_S holds and all the instances of predicate *P* in M for which Q_P holds.

Fact (F1) says that knowing the content criteria of *Msub* allows a consumer to infer information from it that may not be visible explicitly in the content of the model. For example, it is only through the content criteria of diagram *DTollPrice* shown in Figure 5.4 that we know that the attributes shown for vehicles are all related to the toll price computation. As another example, consider diagram *DFamily* in Figure 5.5 with content criteria given by the following extractor:

```
$F_{DFamily}$(M:UML): CD := [
    precondition := $\exists$veh, fam:M.Class, cr:M.Association ·
            M.name(veh) = "Vehicle" $\wedge$
            M.name(fam) = "Family" $\wedge$
            M.name(cr) = "Carries",
    $Q_{Class}$(c) :=  (c = veh $\vee$ TC(subClassOf(c, veh))) $\wedge$
            (c = startClass(cr) $\vee$ TC(M.subClassOf(c, startClass(cr)))) $\wedge$
            (fam = endClass(cr)  $\vee$ TC(M.subClassOf(fam, endClass(cr))) $\wedge$
            $\neg\exists$cr1 : M.Association ·  // no restriction of cr to non-fam
                    M.subsets(cr1, cr) $\wedge$
                    (startClass(cr1)  = startClass(cr)  $\vee$
                     TC(M.subClassOf(startClass(cr1), startClass(cr)))) $\wedge$
                      (fam $\neq$ endClass(cr1) $\wedge$
                      $\neg$TC(M.subClassOf(fam, endClass(cr1)))
]
```

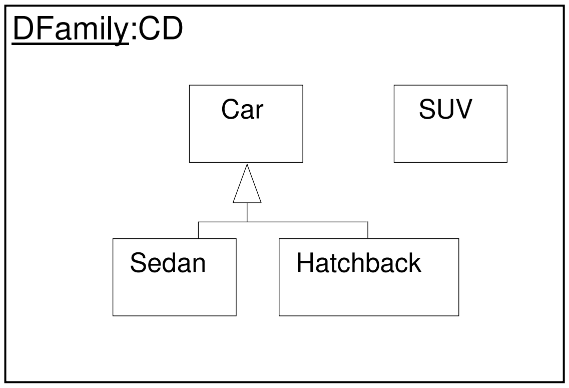


Figure 5.5. Diagram showing vehicle types that can carry families.

This expresses the condition that *DFamily* contains all vehicle types that can carry a family. From the precondition we can infer the implicit information that there exist classes *Vehicle* and *Family* and association *Carries*. From Q_{Class} we can infer that the classes in *DFamily* are all subclasses of *Vehicle* and that have a (possibly inherited) *Carries* relationship to *Family*[23]. None of this information is evident from the content of *DFamly*. This contrasts with *DTollTicket* shown in Figure 5.3 having content criteria given in (13). In this case, the precondition that there exists a class *TollTicket* is also explicitly evident in the diagram itself. Thus, the precondition is only sometimes a source of additional implicit information.

Now consider fact (F2). This says that Msub is *complete* with respect to M for properties Q_S and Q_P for all sorts S and predicates P in Tsub. Thus, although the interpretation of

[23] Note that we are using the *subsets* relationship that can hold between associations in UML 2.2 to define association restrictions on more specialized classes.

diagrams typically assumes they are incomplete and hence takes an open world interpretation for all properties[24], the content criteria say that one can safely interpret *Msub* as closed world with respect to properties Q_S and Q_P. Specifically this means that it is possible to answer queries involving $\neg Q_S$ and $\neg Q_P$ directly using the information in *Msub*. For example, consider *DTollTicket* in Figure 5.3 and the query "Does *TollTicket* have a subclass *AnnualTicket*?" Without knowing the content criteria in (13) this query cannot be decided by the information in *DTollTicket*; however, with the content criteria, the answer can be determined to be "no."

Finally, we consider how content criteria can be used to compensate for submodels that violate the semantic partiality criterion of logical consequence for submodels defined in Chapter 4. Recall that we argued that a "reasonable" submodel should not say anything that the whole model does not say when interpreted using the standard semantics associated with a model type. In Figure 5.6 we have reproduced the example that shows a case where the submodel *DTollPriceB* does not satisfy this condition and so, in isolation, would be misinterpreted. However, if the content criteria are known, then the implicit information in them can be used to compensate for the misreading of a submodel since it can be used to adjust the consumer's interpretation to take the modeler's intent into account. For example, if we know that the content criteria of *DTollPriceB* extract only the vehicles that can be two-seaters, then the consumer can infer that the set of subclasses

[24] Of course this depends on the modeling language. For example, for a state machine, the semantics assumes that it shows all the states and transitions and not just a subset of them.

of Vehicle shown may not be complete and hence the constraint "{complete}" should not be interpreted in the standard way.

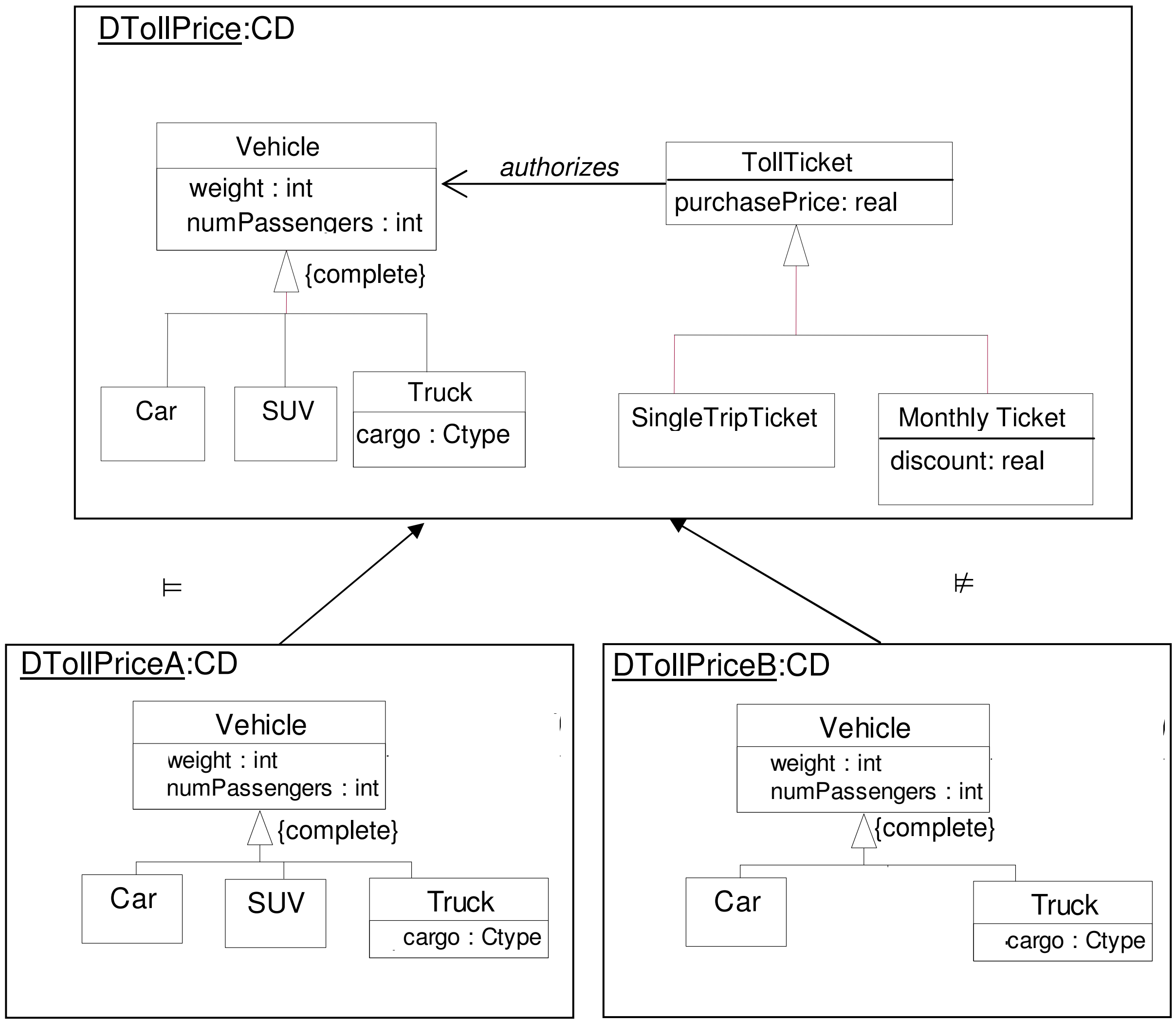


Figure 5.6. Examples of valid and invalid submodels (reproduced from Figure 4.7).

5.5.2 *Impact on model quality*

We can directly relate the formal structure of content criteria to the types of defects that expressing content criteria can be used to detect as shown in Table 5.1. The first two types of defects relate to the coherence between content criteria and submodel names. As discussed in Section 5.4.2, the name is typically the carrier of informally expressed information about the submodel's intent. Two potential problems are identified here. Naming inconsistency refers to the fact that there is a potential for confusion if different diagrams with similar intents are named in ways that do not reflect their similarity. Naming inaccuracy occurs when the name does not express the intent as given by the content criteria and thus leads to confusion for model consumers and producers.

For example, consider the two diagrams "M1 – details of toll tickets" and "M2 – the kinds of vehicles" with corresponding content criteria CC_{M1}:= *classDetails*(*TransportationSystem*, ClassSet(*M*, "*TollTicket*")) and CC_{M2} := *classDetails*(*TransportationSystem*, ClassSet(*M*, "*Vehicle*")), respectively. The similarity of the intent that is evident from their content criteria is not reflected in the naming. A more consistent naming scheme might be "M1 – details of toll tickets" and "M2 – details of vehicles". On the other hand, if the name of *M2* was "M2 – details of cars" this would be case of naming inaccuracy since *M2* contains more than just the details of cars.

Content exclusion and inclusion defects can occur when an instance of the submodel violates the content criteria – either by excluding intended information or by including unintended information. For example, *DTollTicket* would have an exclusion defect if it

Table 5.1. Defect types detectable by formally defining content criteria.

Defect Type	Description	Occurrence criteria
Naming inconsistency	The form of the name of *Msub* differs from other submodels with similar intent.	The form of the name of *Msub* differs from other submodels with same extractor and different parameters.
Naming inaccuracy	The intent of *Msub* does not mean the same thing as its name.	The extractor of *Msub* does not produce the submodel expressed by the name of Msub.
Content exclusion	*Msub* does not contain some information from *M* that it is intended to.	An instance of an inclusion constraint in which right hand side is satisfied but the left hand side is not.
Content inclusion	*Msub* contains some unintended information.	An instance of an inclusion constraint in which the left hand side is satisfied but the right hand side is not.
Unmodeled information	The intent of *Msub* cannot be (fully) expressed using the content of *M.*	One or more formulas Q_i cannot be formally expressed in terms of information in M.
Weakly modeled information	The intent of *Msub* can only be expressed by using informal information in *M.*	One or more formulas Q_i are expressed using content in M that is not modeled in the metamodel of M.

omitted the class *SingleTripTicket* and it would have an inclusion defect if it included the class *Car* as a subclass of *Vehicle.*

The final two types of defects are only "potential" defects in the sense that even a violation may be considered acceptable. The value here is in the fact that the modeler is forced to consider the situation and make a decision on how to deal with it. The case of unmodeled information occurs when the content criteria is underdetermined by the information in M. This was the case with diagram *DTollPrice* discussed above since the attribute inclusion condition Q_{Attr} requires a determination of whether the attribute affected toll price and *TransportationSystem* does not have enough information to determine this. As discussed there, the corrective action required depends on whether or not the information represented by the inclusion conditions is considered to be needed by downstream processes using the model.

The last type of defect is the case where the inclusion condition *can* be specified using information in the model but this information is only "weakly modeled" using some informal scheme such as naming conventions. For example, if a convention is used to prefix all attributes that affect toll price with the string "Toll_", this would allow the inclusion condition to be defined by checking for this prefix. The potential problem with this is that the semantics of these conventions may be lost in downstream uses of the model unless they are recorded with the model in some way. Thus, it may be preferable to promote this information to "first class" status, when possible, by modeling it directly – e.g., putting all toll related attributes in a single class called *TollCalculation*.

5.5.3 Support for automation and evolution

Since content criteria are expressed as role constraints between a submodel and its model, all the opportunities for automation discussed in Section 2.3.3 are applicable. This includes conformance checking, extension to conformance and change propagation. The details of these scenarios are discussed in general for role constraints in Chapter 6.

Automation and naming defects

Since name related defects require an interpreter of natural language, neither type of defect can be reliably detected in an automated way. However, one way to avoid these kinds of defects is to always generate diagram names from the content criteria as suggested in Section 5.4.2. This ensures consistency and accuracy of naming. In particular, as the model evolves, generating names from content criteria provides a way to update the names in an automated way.

5.6 Summary

In this chapter we have explored the concept of content criteria as introduced in the intent framework. The content criterion of a model characterizes the kind of information the modeler intends to be in the model. We approach the expression of this by assuming that a model is always intended to be some partial view of the maximal model of the subject that can be expressed using a particular model type. This idea is developed in detail for the special case of submodels, and in particular, diagrams of a model. The general form of content criteria for submodels is given as consisting of a precondition and a set of inclusion conditions. Based on this form, the various ways in which content criteria can provide value is elaborated. Model comprehension is shown to be augmented by the implicit information that content criteria provides about the semantic interpretation of the view. Improved model quality is supported by allowing six new types of defects to be detected that are based on content criteria. Finally, we discuss the use of content criteria for supporting automation and model evolution.

Chapter 6

Macromodeling

As discussed in Chapter 2, modeler intent is expressed at the role level. In this chapter we detail one of the main contributions of this thesis: a modeling language for the role level called the *macromodel* language. We define the syntax (abstract and concrete), semantics and the usage modes of macromodels. Macromodels allow model roles to be defined with role constraints and relationship types can be used for expressing role constraints that hold between models. Special support is given for views and the expression of content criteria. In addition, special support is given for expressing model decompositions; however, we defer the complete treatment of this to Chapter 7 where we discuss decomposition criteria.

6.1 Macromodels

We begin this exposition by using examples to informally introduce the notation and concepts of macromodels. Consider Figure 6.1. This is a diagram of the transportation system macromodel *TransportationProject* that identifies all the model roles in the transportation system project. A macromodel is represented by a box with a thick border. The name and other attributes are given in the top compartment – we refer to this information as the *tag*. Every element type in a macromodel has a tag[25]. The designator "{inc}" as part of the macromodel tag is used to

[25] Although sometimes it is optional.

indicate that the diagram does not show all the contents of the macromodel. Boxes with thin borders represent model roles while lines represent intended binary relationships between these roles (intended *n*-ary relationships are expressed using the diamond notation as with UML class diagrams). In both cases, their types are shown in the tag.

In general, we use the term "relationship" loosely to refer either to an actual relationship between particular models or to a relationship between roles that expresses the intent that a particular model relationship must hold between models that play the roles. Whenever there is potential for confusion we will refer to the former as a *model relationship* and the latter as a *role relationship*. The relationships shown in a macromodel are always role relationships. The

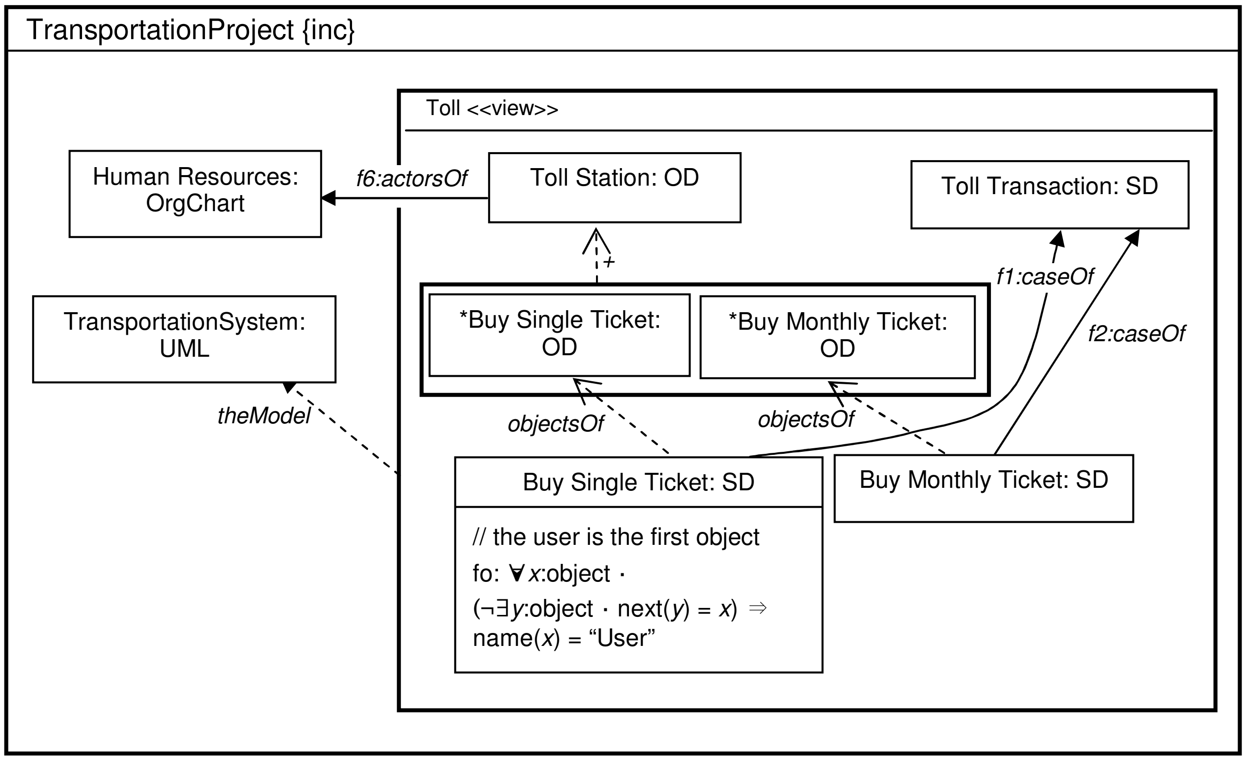


Figure 6.1. A macromodel of the transportation system example.

notation for relationships varies depending on their properties. If the direction of a relationship is important, it has an arrow head. The arrow head is filled unless the relationship is a transformation in which case the arrow head is open and points to the result of the transformation. Normally, a relationship has a name and a type and the name is used to identify the relator model representing the mapping. However, there are cases when the name can be omitted and then we say that the relationship is *anonymous*. One such case is when the relationship type is pure and this is the case with the occurrences of *objectsOf* . Another case is when the mapping is unrealized. We discuss the case of unrealized models and mappings below.

Since macromodels can contain macromodels, these are represented by a contained box as is illustrated with the macromodel *Toll*. This containment hierarchy forms a tree with the root referred to as the *root macromodel* (in this case this is *TransportationProject*). Any non-root macromodel is called a *contained macromodel*. A contained macromodel could be named or be anonymous like the one containing roles *BuySingleTicket:OD* and *BuyMonthlyTicket:OD*. For example, here we use an anonymous macromodel to express the fact that *TollStation* is intended to be the sum of the two object diagrams. Note that, the name/type pair for any non-anonymous model role or relationship must be unique within its root macromodel.

A model role or relationship can be marked as *unrealized* by beginning its tag with an asterisk. Saying that a role is unrealized means that there is no corresponding model/mapping in the project that plays it but it must be possible for there to exist a model/mapping that could satisfy its constraints. This is used to express more complex constraints graphically by introducing intermediate models or mappings. For example, in Figure 6.1, we want to show that *TollStation:OD* contains all and only the objects and links required by *BuySingleTicket:SD* and

BuyMonthlyTicket:*SD*. This is achieved by showing that it is the sum of the pair of unrealized object diagrams *BuySingleTicket*:*OD* and *BuyMonthlyTicket*:*OD* which each represent the minimal object diagram of their corresponding sequence diagram.

6.1.1 Views, presentations and decompositions

The boxes *Human Resources* and *TransportationSystem* in Figure 6.1 represent simple model roles while the other ones are *views* as defined in Chapter 5. The generators and the base model can be shown connected to the view via dashed arrows. In the case of a generator, the arrow points to the view and in the case of the base model it points to the base model. The generator endpoints can optionally be given a name (not shown in this example) and the base model endpoint always has the name *theModel.* When the extractor defining the content criteria of the view takes a single generator as an argument then it can be indicated on the dashed arrow for the generator – this is case with the instances of *objectsOf.* Views are indicated by the stereotype "<<view>>" in the tag and have the base model designator *theModel*; however, both of these may be omitted in certain cases as we discuss next.

Toll is a kind of view of *TransportationSystem* that is also a macromodel. Views that are macromodels are called *presentations* and these can only contain other views. Thus, a view contained within a presentation may omit its stereotype since it must also be a view. Furthermore, if the base model designator is omitted then it is considered to be inherited from the containing presentation. Thus, the base model for *BuySingleTicket*:*SD*, *BuySingleTicket*:*OD* and for the anonymous presentation containing *BuySingleTicket*:*OD* is *TransportationSystem*:*UML*. This feature reduces the effort and complexity of expressing collections of views of the same model.

The notation also allows a presentation to be combined with its base model into a single box rather than having separate boxes connected with the *theModel* designator. This more compact notation is useful for the common case where a model has only a single presentation. This is illustrated with the model *TransportationSystem:UML* in Figure 6.2. The tag of the base model *TransportationSystem:UML* is shown on the first line of the top compartment and the tag of its presentation is shown on the second line. In this case, it is an anonymous presentation that additionally is a *decomposition* and this is indicated by the view stereotype variant "<<+view>>." A decomposition is a presentation that decomposes its base model. That is, it includes the implicit constraint that the base model is the sum of the views within the presentation. Decompositions are a commonly occurring structure within a macromodel and these are studied in detail in Chapter 7.

6.1.2 Owned constraints

Figure 6.1 shows another important feature of macromodels. Although relationship types provide one way to express role constraints, there are cases when this is not sufficient because a role requires a "one-off" role constraint. In this case, it can be expressed within a compartment of the role box. This is illustrated by *BuySingleTicket* : *SD* which adds the constraint that the name of the first object in the sequence diagram must be "User". In this case, the constraint is expressed using first order logic (as indicated by the language qualifier "fo:") but any constraint language is allowable. In particular, an owned constraint can be an entire macromodel – this is particularly meaningful for role types described next.

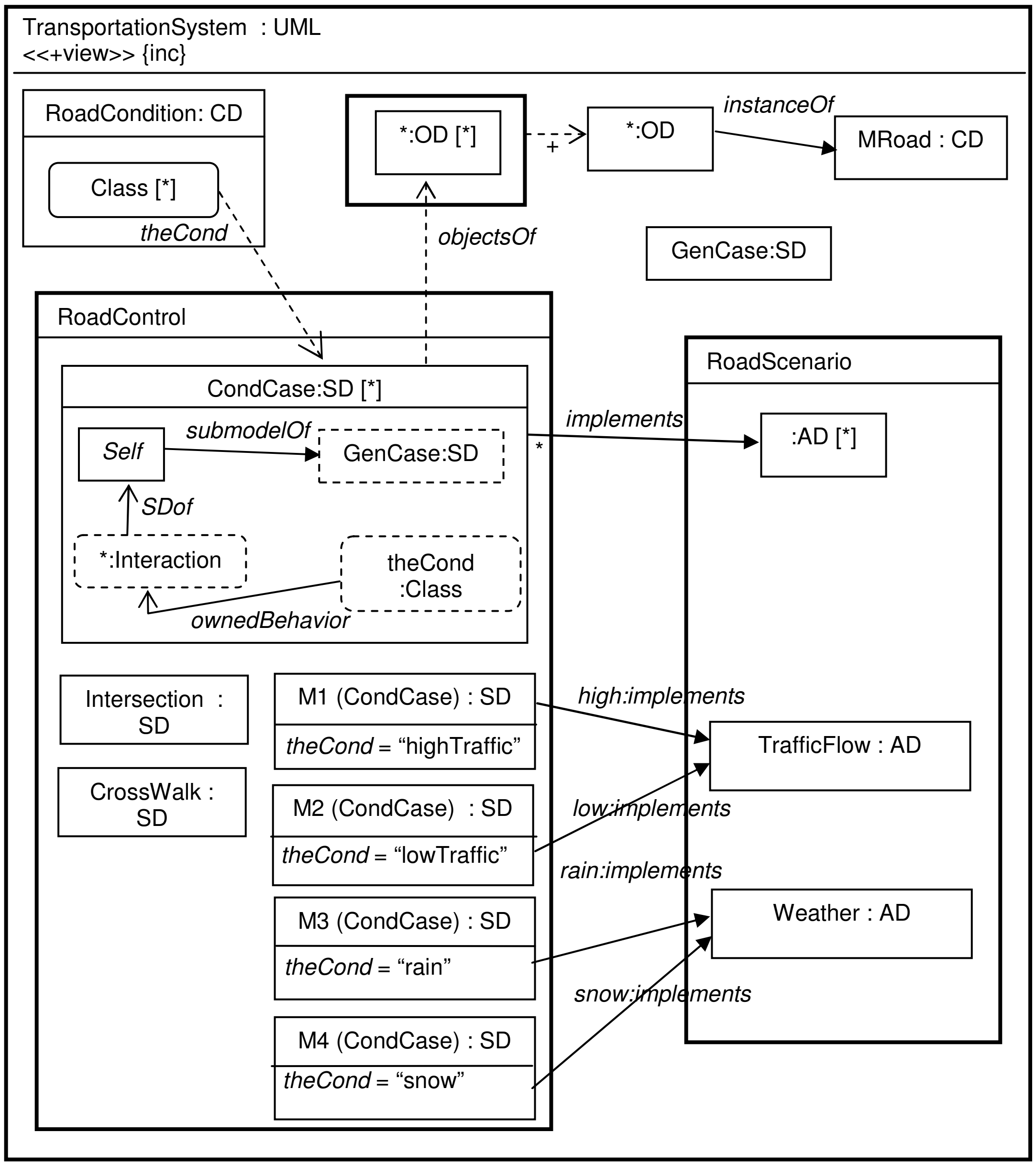


Figure 6.2. Another view of *TransportationSystem*.

6.1.3 Role types

In Figure 6.2, some of the tags for model roles have a multiplicity specifier such as "[*]". This represents a *model role type* rather than a model role. The multiplicity can be expressed more generally as [*upperLimit*] or [*lowerLimit*..*upperLimit*]. A model role type represents a subset of model roles within its containing macromodel and provides a way to classify model roles in order to efficiently express role constraints they have in common. For example, *CondCase* represents a subset of the sequence diagrams within the macromodel *RoadControl*. If the name of a role type is omitted and only a model type is given in the tag then it is the anonymous role type representing the set of all model roles of that type within the macromodel. For example, :*AD* [*] represents the set of all activity diagrams in *RoadScenario*. A model role indicates that it is in a role type by adding the role type name in parentheses following the model role name in the tag. For example, we have *M1 (Cond)* : *SD*, *M2 (Cond)* : *SD*, etc. The set of model roles must conform to the multiplicity of the role type.

A line (or diamond for the *n*-ary case) with a model role type on one or more of its ends represents a role relationship type. The multiplicity is given on its endpoints in the same manner as for a UML associations. The multiplicity is assumed to be “1” if it is omitted. Role relationship types can also be anonymous (i.e., type only) and relationships designate their membership in a role relationship type in the same manner as model roles. Note that if a model role is on an end of a role relationship type then it is treated like a model role type with multiplicity “1” with regard to the relationship type multiplicity. Thus, Figure 6.2 says that *RoadControl* contains a set of sequence diagrams called *CondCase*, each of which has an *objectsOf* relationship to an unrealized object diagram in the anonymous macromodel and an *implements* relationship to a *RoadScenario* activity diagram.

Any owned constraints in a role type or role relationship type are assumed to be replicated for each of its instances. Thus, the constraints owned by the type are taken to apply to any instance of the type. An owned constraint expressed as macromodel in a role type is a graphical way to express constraint reuse. For example, in Figure 6.2, the role type *CondCase* contains an owned macromodel representing a constraint that should hold for each of its instances. In this macromodel, the instance is referred to using the pseudo role *Self*, the associated generator *theCond* is shown as an element role and additional roles *GenCase* and an unrealized *Interaction* are present as well. The dashed border on a role indicate that they are actually external to the *CondCase* and that they are just shown within its box to allow a constraint to be expressed. The constraint expressed by the macromodel is that each *CondCase* is a submodel of *GenCase* and is generated from *theCond* by composing the transformation *ownedBehavior* with *SDof*. Note that only pure relationships can be used in an owned macromodel. Using an macromodel in this way is semantically equivalent to replicating it for each instance of *CondCase* by using the bindings of the local roles *Self* and *theCond* for each instance. In the instances of *CondCase* we see another notational convention – the value of generator *theCond* is given textually as an attribute rather than graphically.

To summarize, role types are an abstraction mechanism that both add information about the intent regarding how the macromodel is structured and simplifies the macromodel by allowing common information to be expressed for all the elements of a similar set of roles.

6.1.4 Relationships to model elements

In some cases it is useful to refer to particular model elements or sets of model elements directly from within a macromodel. For example, in Figure 6.2, the classes of *RoadCondition* : *CD* indicate different conditions that could hold in traffic context. The dashed arrow from the element set *Class* representing the set of classes in *RoadCondition* to *CondCase* indicates that the classes are generators for the sequence diagrams. Thus, each class of *RoadCondition* generates a unique corresponding *CondCase* sequence diagram. This kind of relationship is fundamental to the expression of decomposition criteria discussed in Chapter 7. Note that these representations of model elements are only intended for expressing model role constraints that require reference to model elements and are not intended for expressing the detailed content of particular models. The latter activity occurs at the model level rather than the role level and utilizes the particular notations that are appropriate for the model type.

6.1.5 Ground and general macromodels

Recall that in Chapter 2 we made a distinction between the project level and the method level and noted that model roles and role constraints can exist at both levels. Correspondingly, macromodels can also be used at either level to express the intended roles and their constraints. However, to understand the differences between these two uses of macromodels we must elaborate this distinction further.

The project level corresponds to the actual set of models used in a particular project. We will assume that each project has a macromodel that describes the project's intended structure and contains a model role symbol for each required model in the project. Furthermore, each such model role symbol is mapped to the model that plays the role within the project and this allows conformance to modeler intent to be assessed. We will call a macromodel with such a mapping, a *ground macromodel*. Despite its distinguished role within a project, the project macromodel should be a model like any other within the project in the sense that it evolves over time and when the project is non-conformant to it then a valid way to resolve this is to change the macromodel.

The method level spans many possible occurrences of a method and hence there is no specific set of models that are mapped to model roles. Thus, we consider a macromodel used at this level to be a *general macromodel*. In the same way that a project may represent an occurrence of a method specialized to the needs of the project, the project macromodel may be a specialization of a general macromodel containing generalized expressions of intent regarding models. Thus, general macromodels are used to define classes of similar ground macromodels. However it is important to note that although the ground-general macromodel relationship seems like an instance-type relationship it is actually a specialization relationship.

Figure 6.3 shows a case where the ground macromodel *TransportationProject* is taken to be a specialization of a generalized macromodel *Dev*. The notational convention is that a general macromodel has the stereotype "<<gen>>" in its tag. *Dev* defines a general development process that identifies two singleton model role types, *ReqSpecification* and *Design* related by a relationship *RD* and the set of model roles *PlatformImpl* related to *Design* by a set of relationships *ID*. Note that the model and relationship types given are abstract.

The ground macromodel *TransportationProject* specializes *Dev*. The element *TransReq*(*ReqSpecification*) : *UML* indicates that the model role *TransReq* specializes *ReqSpecification*. All of the abstract model and relationship types in *Dev* are concretized in *TransportationProject* in this way. In general, a specializing macromodel inherits all of the model roles or relationships (and sets) in the more general macromodel and can specialize these further as well as adding new model roles and relationships.

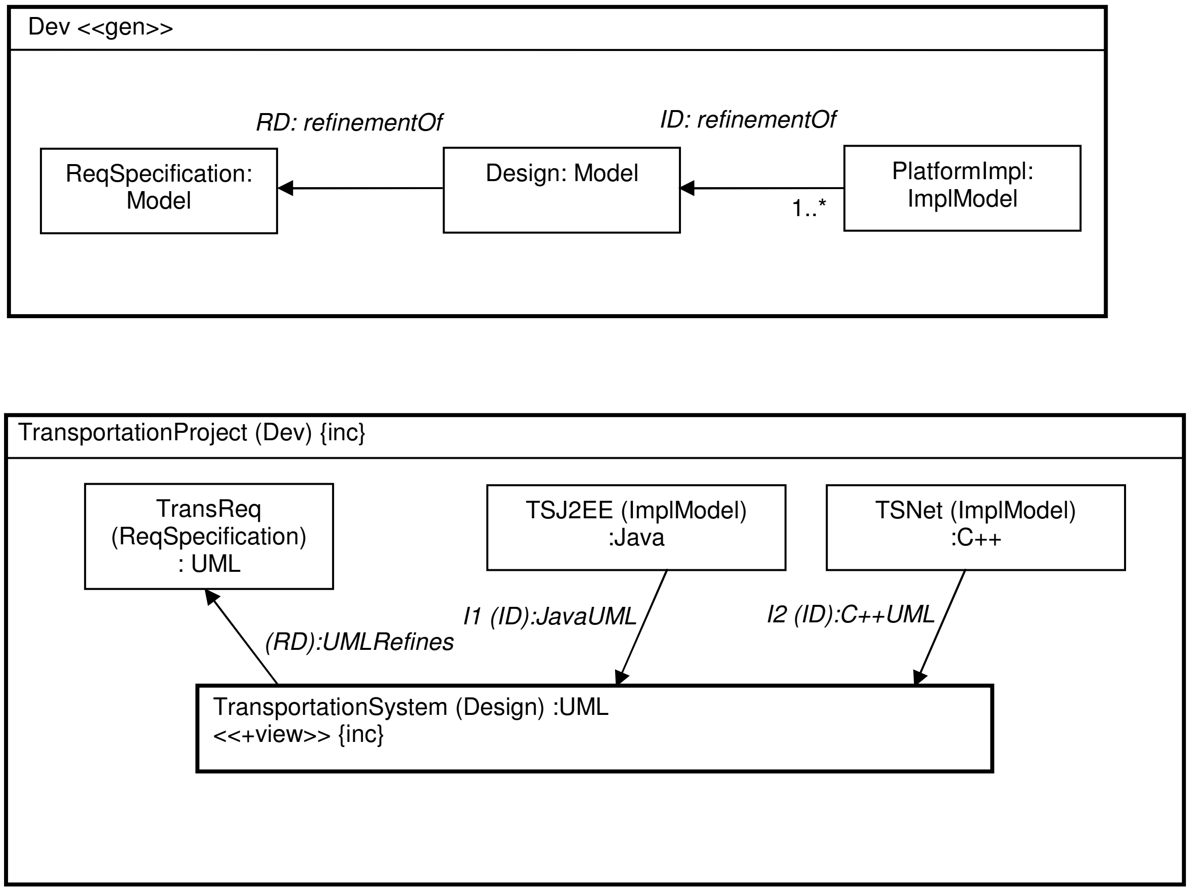


Figure 6.3. A general macromodel and an instance of it.

Specializations of a role/ relationship must satisfy the usual conformance condition that the specialization is a more constraining role/relationship in terms of model/relationship type and multiplicity. The difference with a ground macromodel is that it must specialize *all* the model roles and relationships in the general macromodel. *TransportationProject* satisfies these conditions – it correctly specializes the elements of *Dev* as well as adding various other model roles and relationships.

6.1.6 Type diagrams

A macromodel includes references to model and relationship types and these too can be shown in a diagram that depicts this part of a macromodel. Figure 6.4 is a *type diagram* showing the types portion of the macromodel depicted in Figure 6.1. A type diagram is shown simply as a UML style class diagram with no multiplicities. Types have stereotype “<<type>>” in their tag. For clarity, Figure 6.4 omits the stereotypes of relationship types. Relationship type arrow conventions are the same as for relationships in a macromodel diagram. Metamodel morphisms are shown with stereotype “<<morphism>>”. Note that in the macromodel, the concrete type elements depicted in this diagram contain an attribute (not shown) that references the metamodel artifacts defining these types.

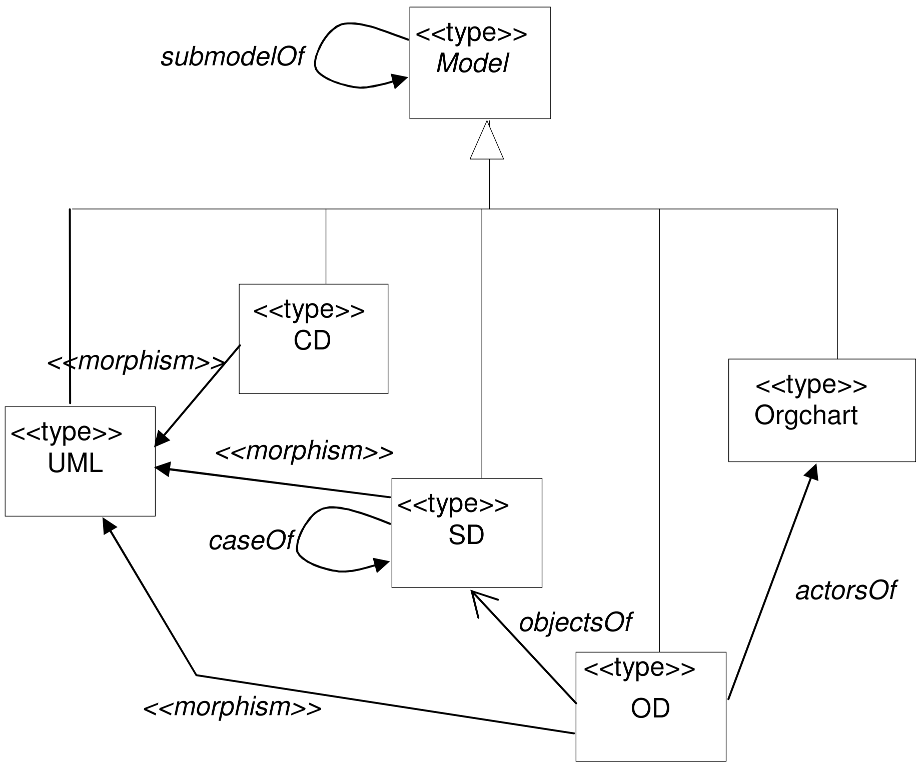


Figure 6.4. The type diagram corresponding to Figure 6.1.

6.2 Formalization

6.2.1 Abstract syntax

Figures 6.5 to 6.7 show the metamodel for the macromodeling language. *MacromodelKind* is the abstract element type that represents all variants of macromodels. Each instance consists of a set of *ContainedMember* elements. When a macromodel is not contained in another then it is a *RootMacromodel* and as discussed above, it can either be general or ground and this is indicated by the attribute *use*. A ground macromodel contains elements that map to the actual model and relationship artifacts in the project. Different subclasses of *ContainedMember* exist for model roles, role relationships and macromodels and each of these have their ground and "type" versions shown in Figure 6.6. Thus we have *ModelRole*/*ModelRoleType*, *SimpleRel*/*SimpleRelType* and *ContainedMacromodel*/*ContainedMacromodelType*. A relationship has *EndType* elements where it attaches to its endpoints. In addition to these, a *ModelRole* can have *ElementSet* and *ElementRole* elements showing the elements within a model as in Figure 6.2. A *View* is a special kind of member that "wraps" other members to turn them into views as defined in Chapter 5. Finally, in Figure 6.7 we have the elements that represent model and relationship types.

We defer any further formal description of macromodel syntax to the specification in Section 6.4. Instead we turn to the issue of defining the formal semantics.

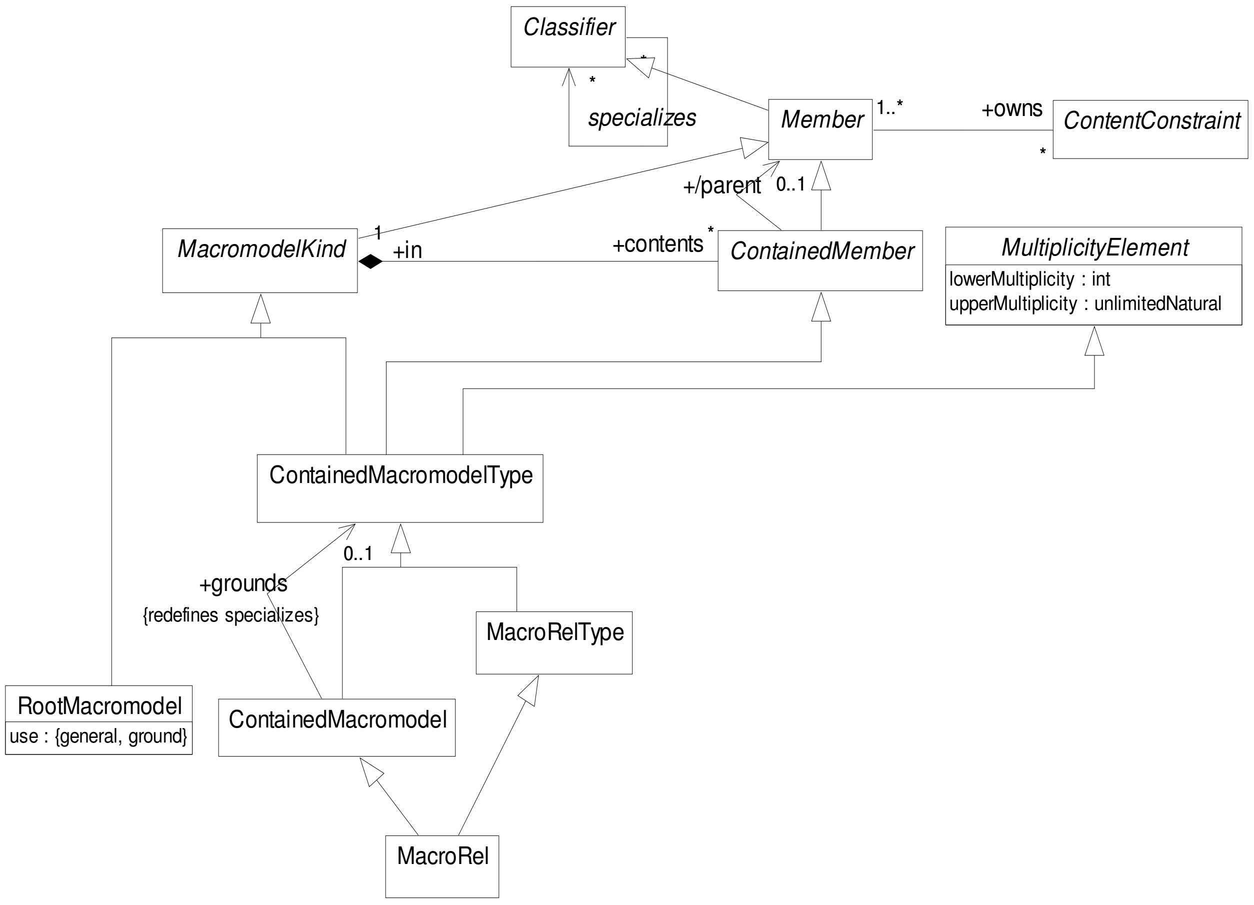


Figure 6.5. The core abstract syntax of the macromodeling language.

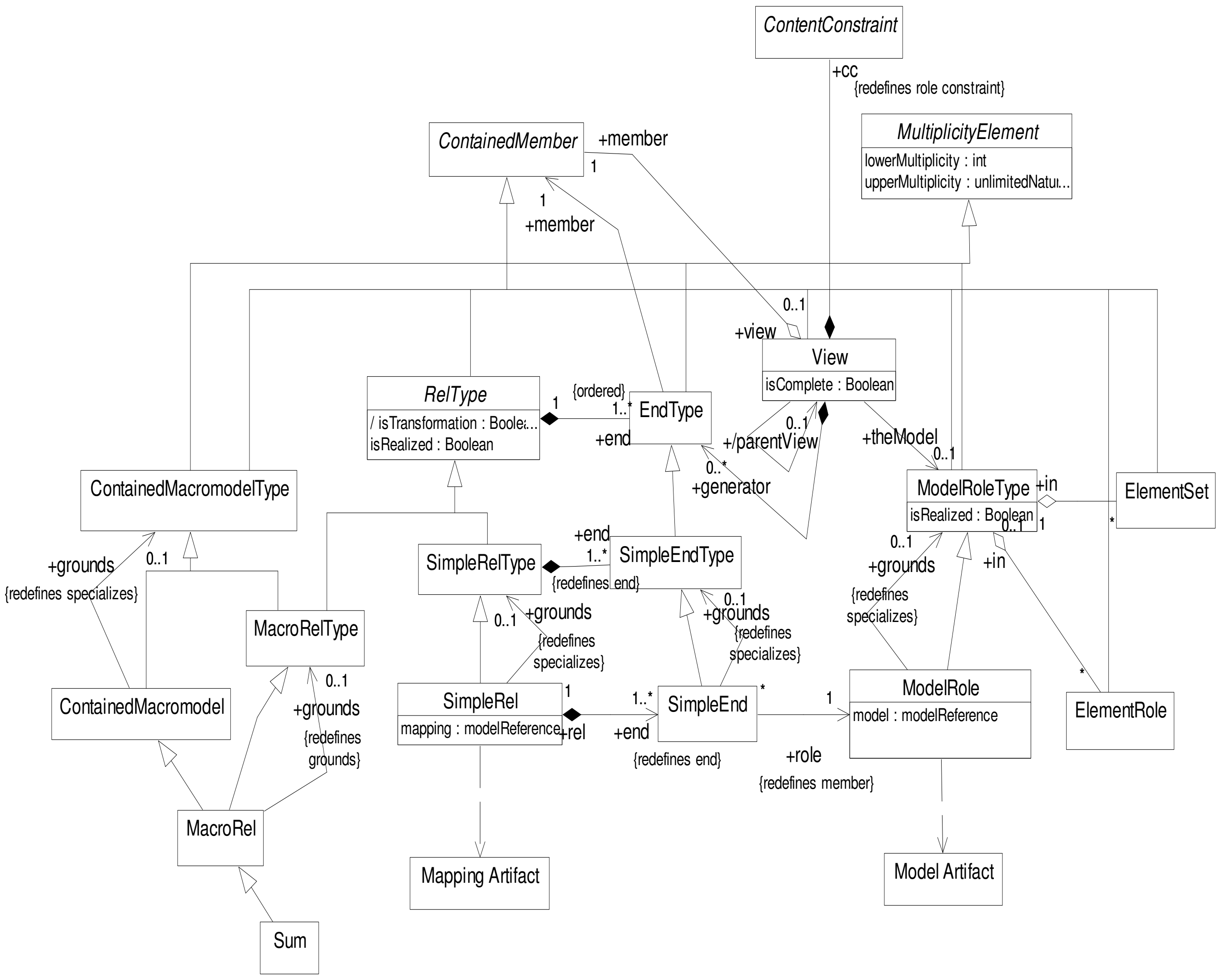


Figure 6.6. The elements contained in a macromodel.

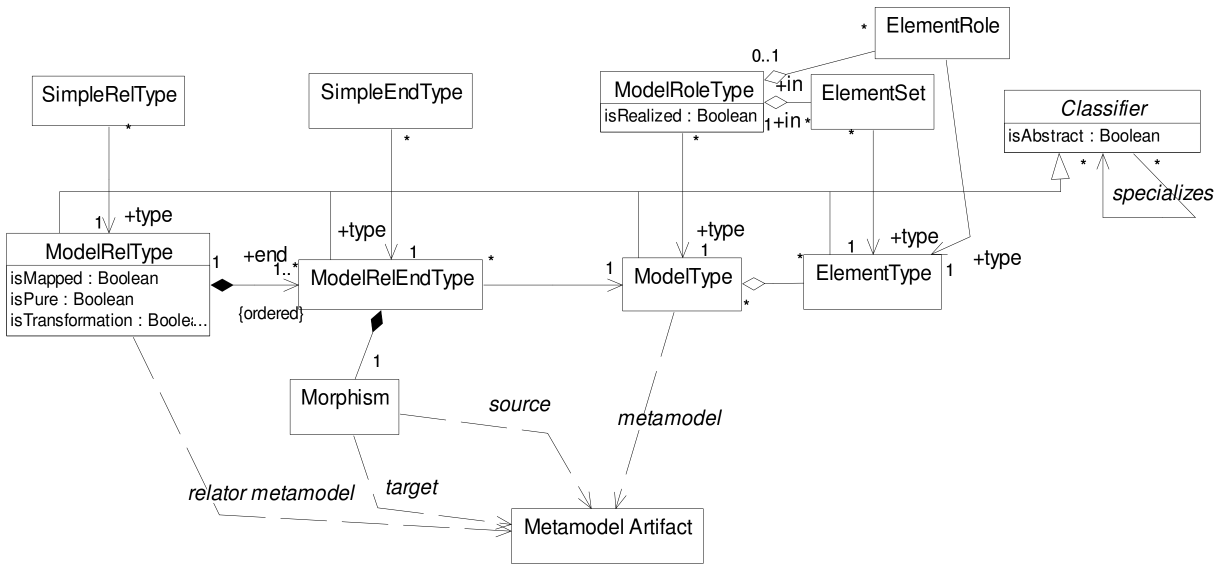


Figure 6.7. Model type related elements.

6.2.2 Macromodel semantics

In this section we give the semantics for ground macromodels . Our focus in defining the semantics is to identify the ways in which a macromodel expresses modeler intent and what the conditions of its satisfaction are. Thus we are interested in conformance to modeler intent. First we define the notion of a *configuration*.

Definition 6.1. *Configuration*. For a project *P* we define a *configuration* δ of *P* as consisting of a pair $\langle K_P, \Delta_P \rangle$ where K_P is the project macromodel and Δ_P is the partial function from the realized *ModelRole* or *SimpleRel* elements K_P to the model and mapping artifacts that they reference.

Now, the conditions for the conformance of a configuration can be formally defined.

Definition 6.2. ***Conformance conditions*****.** We say that configuration $\delta = \langle K_P, \Delta_P \rangle$ is conformant to the modeler intent expressed in K_P iff the following three conditions hold:

C1. Δ_P is a total bijective function
C2. K_P conforms to its specialization constraints
C3. Δ_P conforms to K_P

The semantics of a general macromodel are based on the rules for specializing it and these are implicit in the description of condition (C2).We now discuss each of these conditions in detail.

C1: Δ_P is a total bijective function

The presence of a realized *ModelRole* or *SimpleRel* element in K_P represents an existential intent on the part of the modeler since this means that it is expected to be played by an artifact within *P*. Δ_P is defined to be a partial function because it may not be the case at all times in the project that every such element has a corresponding player. When this is the case, the existential intent of the modeler is violated and thus, the condition that Δ_P is a total function says that δ conforms to existential intent.

We also expect Δ_P to also be injective and surjective (i.e., bijective). The fact that it is injective means that the role of an artifact within the project is represented by a unique symbol within the macromodel – although, in a macromodel diagram, this may appear multiple times and in multiple diagrams. The fact that it is surjective means that the role of every artifact in the project is represented within the macromodel. These requirements are not an issue of conformance to modeler intent but rather a condition of the correct usage of project macromodels.

C2: K_P conforms to its specialization constraints

The *ContainedMember* elements within a macromodel are classifiers and hence a member may specialize another member. There are constraints on what constitute valid specializations. We refer to these constraints as specialization constraints and K_P must conform to them. Although specialization constraints are well-formedness constraints for macromodels, we distinguish them from other well-formedness constraints because in this case the specification of one part of the macromodel (e.g., a model role type) is intended to constrain another part of the macromodel (e.g., a model role). In this sense, specialization constraints are constraints that a user of a macromodel specifies and hence comprises part of the modeler intent. Thus, a violation of specialization constraints constitutes a violation of the modeler intent while a violation of the other well-formedness constraints of the macromodeling language just implies that the model is ill-formed as a macromodel.

The conformance conditions for specialization constraints are given in Figure 6.8. Here, Mem_{RT} is the set of members in K that specialize RT either directly or indirectly, $RMem_{RT}$ are the endpoint tuples of the relationships in Mem_{RT} and $EMem_i[r_1, \ldots, r_{i-1}, r_{i+1}, \ldots, r_n]$ is the set of endpoints of type RTi when we fix the remaining n-1 endpoints to $r_1, \ldots, r_{i-1}, r_{i+1}, \ldots, r_n$.

For every *MacromodelKind* K where either $K = K_P$ or K is in K_P at any depth,

For any specializable member RT referenced within K,

For *ContainedMacromodelType* RT, the following must hold:

- The set of RT contained macromodels must conform to the multiplicities of RT
  - $\text{lowerMultiplicity}(RT) \leq |\,Mem_{RT}| \leq \text{upperMultiplicity}(RT)$

For *ModelRoleType* RT, the following must hold:

- The model types of all model roles in RT must specialize the model type of RT
  - $\forall r \in Mem_{RT} \cdot \text{specializes}(\text{type}(r), \text{type}(RT))$
- The set of RT model roles must conform to the multiplicities of RT
  - $\text{lowerMultiplicity}(RT) \leq |\,Mem_{RT}| \leq \text{upperMultiplicity}(RT)$

For *RelType* RT, with *EndType* elements $RT1, \ldots, RTn$, the following must hold:

- If RT is a *SimpleRelType* then the model types of all model roles in an endpoint of RT must specialize the model type of the endpoint
  - $\forall i \in \{1, \ldots, n\} \forall r \in Mem_{RTi} \cdot \text{specializes}(\text{type}(\text{role}(r)), \text{type}(RTi))$
- The set of model roles in an endpoint of RT must conform to the multiplicities of the endpoint
  - $\forall i \in \{1, \ldots, n\} \forall r_1 \in Mem_{RT1}, \ldots, r_{i-1} \in Mem_{RTi-1}, r_{i+1} \in Mem_{RTi+1}, \ldots, r_n \in Mem_{RTn} \cdot$

    $\text{lowerMultiplicity}(RTi) \leq |\,EMem_i[r_1, \ldots, r_{i-1}, r_{i+1}, \ldots, r_n]| \leq \text{upperMultiplicity}(RTi)$

Where,

$Mem_{RT} = \{m \mid \text{in}(m) = K \wedge \text{TC}(\text{specializes}(m, RT))\}$

$RMem_{RT} = \{\langle r_1, \ldots, r_n\rangle \mid \exists x \in Mem_{RT}\ \forall i \in \{1, \ldots, n\} \cdot \text{end}(x, r_i) \wedge r_i \in Mem_{RTi}\}$

$EMem_i[r_1, \ldots, r_{i-1}, r_{i+1}, \ldots, r_n] = \{r \mid \langle r_1, \ldots, r_{i-1}, r, r_{i+1}, \ldots, r_n\rangle \in RMem_{RT}\}$

Figure 6.8. Specialization constraint conformance conditions.

C3: Δ_P conforms to K_P

To define the formal conformance relationship between Δ_P and K_P we proceed by first translating K_P to an equivalent first order theory $\langle \Sigma_K, \Psi_K \rangle$. Then we construct a particular interpretation J of this theory so that (Δ_P conforms to K_P) $\Leftrightarrow J \vDash \Psi_K$.

Note that the contained macromodel structure is ignored for this conformance relationship because it only concerns the conformance of the project artifacts with the role constraints. Figure 6.9 shows an example of the translation of K_P applied to the macromodel *Toll* in Figure 6.1. First consider the construction of the signature Σ_K. Each model type and relationship type T has a corresponding sort S_T representing artifacts with signature Σ_T and each realized model role and relationship C is represented by a constant R_C of the appropriate sort. For example, $R_{TollStation}$: S_{OD} represents the model role *TollStation* of type *OD*.

The type and role constraints that apply to these roles are represented separately as unary predicates on these sorts. For model types, a unary predicate $TC_T \subseteq S_T$ represents the set of type constraints Ψ_T. For relationship types, S_T represents the relator model instances and we define a unary predicate $TC_T \subseteq S_T$ to represent Ψ_T.

sorts $S_{objectsOf}$, S_{caseOf}, S_{OD}, S_{SD}, S_{UML}, $S_{submodelOf[f:ODUML]}$, …

pred $TC_{objectsOf}$: $S_{objectsOf}$

TC_{caseOf}: S_{caseOf}

$TC_{submodelOf[fODUML]}$: $S_{submodelOf[f:ODUML]}$ …

TC_{OD}: S_{OD}

TC_{SD}: S_{SD}

$RC_{BuySingleTicket}$: S_{SD}

func p_{od}: $S_{objectsOf} \to S_{OD}$

p_{sd}: $S_{objectsOf} \to S_{SD}$

p_{sd1}: $S_{caseOf} \to S_{SD}$

p_{sd2}: $S_{caseOf} \to S_{SD}$

$R_{BuySingleTicket}$, $R_{BuyMonthlyTicket}$, $R_{TollTransaction}$: S_{SD}

$R_{TollStation}$: S_{OD}

$R_{TransportationSystem}$: S_{UML}

R_{f1}, R_{f2} : S_{caseOf}

$+: S_{OD} \times S_{OD} \to S_{OD}$

constraints

$\exists$*f3, f4*: $S_{objectsOf}$, *buySingleTicket, buyMonthlyTicket*: S_{OD}, *f5* : $S_{submodelOf[f:ODUML]}$, …

$TC_{OD}(buySingleTicket) \wedge TC_{OD}(buyMonthlyTicket) \wedge$

$TC_{OD}(R_{TollStation}) \wedge$

$TC_{SD}(R_{BuySingleTicket}) \wedge RC_{BuySingleTicket}(R_{BuySingleTicket}) \wedge TC_{SD}(R_{BuyMonthlyTicket}) \wedge$

$TC_{SD}(R_{TollTransaction}) \wedge$

$TC_{caseOf}(f1) \wedge p_{sd1}(f1) = R_{BuySingleTicket} \wedge p_{sd2}(f1) = R_{TollTransaction} \wedge$

$TC_{caseOf}(f2) \wedge p_{sd1}(f2) = R_{BuyMonthlyTicket} \wedge p_{sd2}(f2) = R_{TollTransaction} \wedge$

$TC_{objectsOf}(f3) \wedge p_{sd}(f3) = R_{BuySingleTicket} \wedge p_{od}(f3) = buySingleTicket \wedge$

$TC_{objectsOf}(f4) \wedge p_{sd}(f4) = R_{BuyMonthlyTicket} \wedge p_{od}(f4) = buyMonthlyTicket \wedge$

$R_{TollStation} = buySingleTicket + buyMonthlyTicket$

$TC_{submodelOf[fODUML]}(f5) \wedge \pi_{sub}(f5) = R_{TollStation} \wedge \pi_{whole}(f5) = R_{TransportationSystem}$

Figure 6.9. Result of applying translation algorithm to macromodel in Figure 6.1.

Each model role or relationship C of type T may also have (or participate in) additional role constraints, either directly or through inheritance from their role types. Thus, we add a unary predicate $RC_C \subseteq S_T$ to represent these when they are not expressed using existing relationship types. For example, *BuySingleTicket*:*SD* contains an additional role constraint and so we have the predicate $RC_{BuySingleTicket} \subseteq S_{SD}$.

For each use of the model sum operator "+" we need to add a version with the appropriate signature to Σ_K. In this case, we are only summing *OD* models into *OD* models. Finally, the metamodel morphisms of relationship types correspond to functions between the appropriate sorts to represent the reducts. For example, the *objectsOf* relationship type is represented by sort $S_{objectsOf}$, role constraints $RC_{objectsOf}$ and the two reduct functions p_{od}: $S_{objectsOf} \rightarrow S_{OD}$ and p_{sd}: $S_{objectsOf} \rightarrow S_{SD}$.

The separation of sorts and constants (representing artifacts and roles, respectively) from the constraints they must satisfy allows us to represent the fact that a project can contain models and mappings that are not necessarily conformant to their constraints. For example, constant $R_{TollStation} : S_{OD}$ is assigned to a model with signature Σ_{OD} that plays this role and this model is conformant to its type constraints when $TC_{OD}(R_{TollStation})$ holds.

Now consider the construction of the constraints Ψ_K. Since an unrealized model role means "there must exist such a model" we represent unrealized model roles as existentially quantified variables. Unrealized or pure relationships are treated similarly.

Thus Ψ_K is actually constructed as a single sentence that begins with a set of existentially quantified variables for the unrealized model roles and relationships. The body of the sentence places constraints on these variables and the constants representing the realized model roles and relationships.

Each relationship translates to a clause that asserts that the relator maps to the correct endpoint models via the reducts and that the relator satisfies its constraints. For example, the constraint expressed by *f1*:*caseOf* in Figure 6.1 produces $RC_{caseOf}(R_{f2}) \wedge p_{sd1}(R_{f2}) = R_{BuyMonthlyTicket} \wedge p_{sd2}(R_{f2}) = R_{TollTransaction}$. Each model role is translated to a clause that asserts that it satisfies its type and role constraints. Thus, *BuySingleTicket*:*SD* produces $TC_{SD}(R_{BuySingleTicket}) \wedge RC_{BuySingleTicket}(R_{BuySingleTicket})$. Each model sum is produces a corresponding sum expression such as $\mathrm{R}_{\mathrm{TollStation}} = buySingleTicket + buyMonthlyTicket$. The translation algorithm described below gives additional details that are not illustrated in this example. Note also that since *Toll* is a view, there is an implicit submodel relationship between each constituent model and *theModel.* For conciseness we only show one of these in Figure 6.8 and put ellipses to indicate the remaining ones.

Translation algorithm

Figures 6.10 to 6.13 show the full algorithm used for the translation. In the algorithm we translate K_P to FO+ theory $\langle \Sigma_K, \Psi_K \rangle$. Note that quantification over macromodel elements is assumed to be over the elements of K_P. In step 1 we initialize the *namemap* mapping that we build incrementally in the algorithm. The purpose of this mapping is to maintain a correspondence between the elements in the macromodel and the names we use for them in the theory being constructed. Steps 3-7 build the components of Σ_K that represent the model and relationship types used by the macromodel. Steps 3-5 add the sorts and unary predicates for types and steps 6-7, define the function between sorts to represent the reducts for relationship types.

Steps 8-30 build the parts of $\langle\Sigma_K, \Psi_K\rangle$ that represent the content of the macromodel itself. Steps 8-9 add unary predicates to hold owned role constraints if they are present. Step 13 adds constants to Σ_K for each realized *ModelRole* and *SimpleRel* element in K_P. Thus, each constant represents an artifact that must be in project Δ_P. Steps 8-9 add the unary predicate representing role constraints if needed. The constraints Ψ_K are then constructed as a single constructed sentence *X*. As discussed above, each unrealized *ModelRole* or *SimpleRel* element and each pure *SimpleRel* element is expressed as an existentially quantified variable and these prefix *X* (steps 12-16). Note that we use the helper function *MakeUniqueName* to construct a globally unique name. We need this because pure relationships do not have a name and unrealized members may not either. Steps 20 – 26 construct the conjuncts of *X* to express the fact that type constraints and role constraints must be satisfied. Each *Sum* element or decomposition *View* yields a conjunct that computes the sum of the flattened content (using sub algorithm *FlattenMacromodel*) of a macromodel since we assume that sum ignores the hierarchical structure (Steps 24-26). An *ElementRole* element *E* in a *ModelRole M* expresses the constraint that *M* contains an element of a given type and name (Steps 27-28). Finally, if a member contains an owned constraint expressed as a macromodel this is translated and added to *X*. We discuss the details of the sub algorithm *TranslateOwnedMacromodel* next.

Recall that an owned constraint expressed as a macromodel only contains pure relationships and may have model roles with local names. The idea is that this macromodel is instantiated (and replicated) for each binding of the local names. If the owning member that is a type (e.g., ModelRoleType, etc.) then it is replicated once for each instance of the type. The algorithm proceeds for each instance by first building a mapping *localmap* that defines a correspondence between the local names and the macromodel elements that instantiate them (steps 1-20). Then it uses the *namemap* mapping from the main algorithm *TranslateMacromodelToFO* to add the clauses expressing the constraints in the macromodel (steps 21-34).

It should be apparent from the algorithm that each member of K_P introduces a constant number of elements into Σ_K and Ψ_K; thus the complexity of algorithm is $O(n)$ where n is the number of members in K_P.

Algorithm. TranslateMacromodelToFO

Input: RootMacromodel K_P

Output: Theory $\langle \Sigma_K, \Psi_K \rangle$

```
1:     $\Sigma_K = \varnothing$, $\Psi_K = \varnothing$, $namemap = \varnothing$
2:     // Build the signature from the macromodel
3:     for every ModelType or ModelRelType $T$ referenced by a ModelRole or SimpleRel
       element and with associated metamodel $\langle \Sigma_T, \Psi_T \rangle$:
4:         add sort $S_T$ to $\Sigma_K$
5:         // predicate representing type constraints
           add unary predicate $TC_T : S_T$ to $\Sigma_K$
6:     for every metamodel morphism $f:\langle \Sigma_T, \Phi_T \rangle \rightarrow \langle \Sigma_{T1}, \Phi_{T1} \rangle$ denoted by a
       ModelRelEndType element:
7:         add a function $p_f:S_{T1} \rightarrow S_T$ to $\Sigma_K$
8:     for every ModelRole or SimpleRel element $C$ with type($C$) = $T$ and owned role
       constraints:
9:         // predicate representing role constraints
           add unary predicate $RC_C : S_T$ to $\Sigma_K$
10:    // Build sentence $X$ by adding quantified variables and conjuncts
       let sentence $X$ = $TRUE$
11:    // for the realized const elements we add constants and for the unrealized
       // elements we add variables that will be existentially quantified
       for every ModelRole or SimpleRel element $C$ with type($C$) = $T$:
12:        if ¬isRealized($C$) or pure($C$) :
13:            // generate a unique variable name
14:            let $namemap(C)$ := MakeUniqueName($C$)
16:            add existentially quantified variable $\exists namemap(C):S_T$ as a prefix to $X$
17:        else
18:            add the constant $R_C:S_T$ to $\Sigma_{MX'}$
19:            let $namemap(C)$ := $R_C$
20:    for every ModelRole or SimpleRel element $C$ with type($C$) = $T$:
21:            conjoin the following to $X$:
                 $TC_T(namemap(C))$
```

Figure 6.10. Translation algorithm – part 1.

```
22:   for every SimpleEnd element $E$ with type($E$) = $f$, rel($E$) = $R$ and role($E$) = $M$ :
23:     conjoin the following to $X$ :
          $p_f(namemap(\mathrm{R})) = namemap(\mathrm{M})$
24:   for every Sum element from MacromodelKind $M$ to ModelRole $C$ or complete
      View $V$ with member($V$) = MacromodelKind $M$ and theModel($V$) = ModelRole $C$:
25:     conjoin the following to $X$ where $\{C_1, \ldots, C_n\}$ = FlatttenMacromodel($M$):
26:       $C = namemap(C_1) + \ldots + namemap(C_n)$
27:   for every ElementRole $C$ with name($C$) = $n$ and ModelRole $C_1$ with in($C$) = $C_1$ :
28:     conjoin the following to $X$ :
          $hasElementWithName(C_1, n)$
29:   for every ContainedMember $C$ that has a MacromodelKind $M$ as an owned
      constraint:
30:     conjoin the result of $TranslateOwnedMacromodel(M, C, namemap)$ to $X$
```

Algorithm. FlattenMacromodel

Input: MacromodelKind M

Output: Set F_M

```
1:    $F_M = \emptyset$
2:    for each realized ModelRole element $C$ such that TC(parent($C$, $M$)):
3:      add $C$ to $F_M$
4:    for each realized SimpleRel element $C$ in $M$ relating ModelRoles $C_1, \ldots, C_n$:
5:      if $(\forall i \in \{1,\ldots, n\} \cdot \mathrm{TC}(\mathrm{parent}(C_i, M)))$ : add $C$ to $F_M$
```

Figure 6.11. Translation algorithm – part 2.

Algorithm. TranslateOwnedMacromodel

Input: MacromodelKind *M,* ContainedMember *C,* name mapping *namemap*

Output: Sentence X_M

```
1:   // first build mapping of local names to model roles
2:   localmap = ∅, $X_M$ = TRUE
3:   // add pseudo role Self to the mapping if it exists in the macromodel
4:   for every ContainedMember $C_1$ such that grounds($C_1$, $C$):
5:      if there exists ModelRoleType $M_1$ in $M$ with name = “Self”
6:        let localmap(“Self”) := $C_1$
7:      if $C_1$ is a View
8:         // add each generator to the mapping that is in the macromodel
9:         for every ModelRoleType $M_1$ in $M$ :
10:           if there exists generator $E$ of $C$ with name($E$) = name($M_1$)
11:              // get corresponding instance of the generator for $C_1$
12:              if there exists generator $E_1$ of $C_1$ such that grounds($E_1$, $E$)
13:                 let localmap(name($E$)) := member($E_1$)
14:     if $C_1$ is a RelType
15:        // add each endtype to the mapping that is in the macromodel
16:        for every ModelRoleType $M_1$ in $M$ :
17:           if there exists EndType $E$ of $C$ with name($E$) = name($M_1$)
18:              // get corresponding instance of the generator for $C_1$
19:              if there exists EndType $E_1$ of $C_1$ such that grounds($E_1$, $E$)
20:                 let localmap(name($E$)) := member($E_1$)
```

Figure 6.12. Translation algorithm – part 3.

```
21:  // translate macromodel
22:  // for unrealized elements or pure rels add unique existentially quantified variables
     **for** every ModelRoleType or SimpleRelType element $M_1$ in $M$ with type($M_1$) = $T$:
23:    **if** ¬isRealized($M_1$) or pure($M_1$) :
24:        // generate a unique variable name
25:        **let** $namemap(M_1)$ := MakeUniqueName($M_1$)
26:        add existentially quantified variable $\exists namemap(M_1){:}S_T$ as a prefix to $X_M$
27:  // add constraint clauses for all pure relationships in $M$
28:  **for** every pure SimpleRelType element $R$ in $M$ with type($R$) = $T$:
29:       conjoin the following to $X_M$ :
           $TC_T(namemap(R))$
30:      **for** every SimpleEndType $E$ in $M$ with type($E$) = $f$, rel($E$) = $R$ and role($E$) = $M_1$:
31:          **if** $localmap(M_1)$ exists // name is local
32:              conjoin the following to $X_M$ :
                   $p_f(namemap(R)) = namemap(localmap(M_1))$
33:          **else**
34:              conjoin the following to $X_M$ :
                   $p_f(namemap(R)) = namemap(M_1)$
```

Figure 6.13. Translation algorithm – part 4.

We now construct an interpretation J of $\langle\Sigma_K, \Psi_K\rangle$ as defined in Table 6.1. The idea is to use the model and relationship types in the macromodel and to define J as containing all possible models on their signatures and define the predicates to in terms of the constraints in the metamodels. Then we assign the constants to the existing models in the project P. Thus, the only way that this set of models will be conformant with the macromodel K_P is if $J \vDash \Psi_K$. This gives the conformance condition that (Δ_P conforms to K_P) $\Leftrightarrow J \vDash \Psi_K$.

Although we don't discuss it here, we can also formulate the semantics by extending $\langle\Sigma_K, \Psi_K\rangle$ to encode all of the above semantic conditions directly in first order logic. The benefit of this is that logic based tools can then be used to provide automation using macromodels. In Chapter 9, we use this approach to implement a subset of the macromodeling language semantics with the Kodkod model finder [TJ07].

6.3 Value of macromodels

6.3.1 Impact on model comprehension

A macromodel provides a means for the expression of modeler intent at the role level. In Chapter 2, we discussed how the expression of modeler intent is a type of summarizing abstraction of model content that helps to manage complexity and aids comprehension. Thus, one way a macromodel supports comprehension is by providing this level of abstraction on a collection of models.

Table 6.1. Definition of the interpretation *J*.

// $[[S_T]]_J$ is the set of all finite FO+ structures on signature Σ_T For each sort $S_T \in sorts_K$, $[[S_T]]_J = [[\Sigma_T, \varnothing]]$
// $[[TC_T]]_J$ is the set of all finite FO+ structures on signature Σ_T that satisfy Ψ_T For each predicate $TC_T \in pred_K$, $[[TC_T]]_J = [[\Sigma_T, \Psi_T]]$
// $[[RC_C]]_J$ is the set of all finite FO+ structures on signature Σ_T that satisfy Ψ_C For each predicate $RC_C \in pred_K$, $[[RC_C]]_J = \{m \mid m \in [[\Sigma_T, \varnothing]]$ and $m \vDash \Psi_C$ where T is the type of role $C\}$
// $[[p_f]]_J$ is the reduct corresponding to f For each function $(p_f : S_{T1} \rightarrow S_T) \in func_K$ corresponding to a metamodel morphism $f{:}\langle\Sigma_T, \Phi_T\rangle \rightarrow \langle\Sigma_{T1}, \Phi_{T1}\rangle$, $[[p_f]]_J = Mod(f)$ with domain restricted to $[[\Sigma_T, \Psi_T]]$
For each function $(+ : S_{T1} \times S_{T2} \rightarrow S_T) \in func_K$, $[[+]]_J$ is given by the algorithm in Section 4.4.3
// $[[R_C]]_J$ is the model or mapping artifact that plays the role C in the project. For each constant $R_C \in func_K$, $[[R_C]]_J = \Delta_P(C)$

Another key way that a macromodel supports the comprehension of model collections is that it allows structural information about the collection to be expressed. In particular, we can identify two types of structural information that a macromodel can express: the relationships between models and the hierarchically structure of model collections. Both of these types of structure support the improved cognitive manageability of information, particularly for decompositions of large conceptual models into collections of views[M06]. A macromodel can also be used to express the intent behind decompositions but we defer discussion of this to Chapter 7.

6.3.2 Support for automation and model evolution

Macromodel semantics can be implemented by tools such as model checkers, model finders and theorem provers to automate the satisfaction of the semantic conditions C1-C3. In Chapter 9 we describe a prototype developed using a model finder. In this section we discuss various usage modes of macromodels to support automation and model evolution. We illustrate these modes using the example in Figure 4.4 reproduced here in Figure 6.14.

Conformance checking mode

Using a macromodel in conformance checking mode means that the three semantic conditions C1-C3 are checked for satisfaction. Condition C1 must hold in order for condition C3 to be checkable. The example configuration δ_x in Figure 6.14 can be seen to satisfy all the semantic conditions. However, if we define configuration δ_{x1} by removing all *sentOver* relation instances from *R1* then the configuration is non-conformant because the constraint that *sentOver* is a total function from *Message* to *Link* was violated (thus, C3 is not satisfied). Note that, in C3, unrealized roles result in a sentence with existentially quantified variables. Thus, if a model finder is used for checking conformance then it requires the construction of models or relationships to assign to each of these variables and this requires extension-to-conformance mode described next.

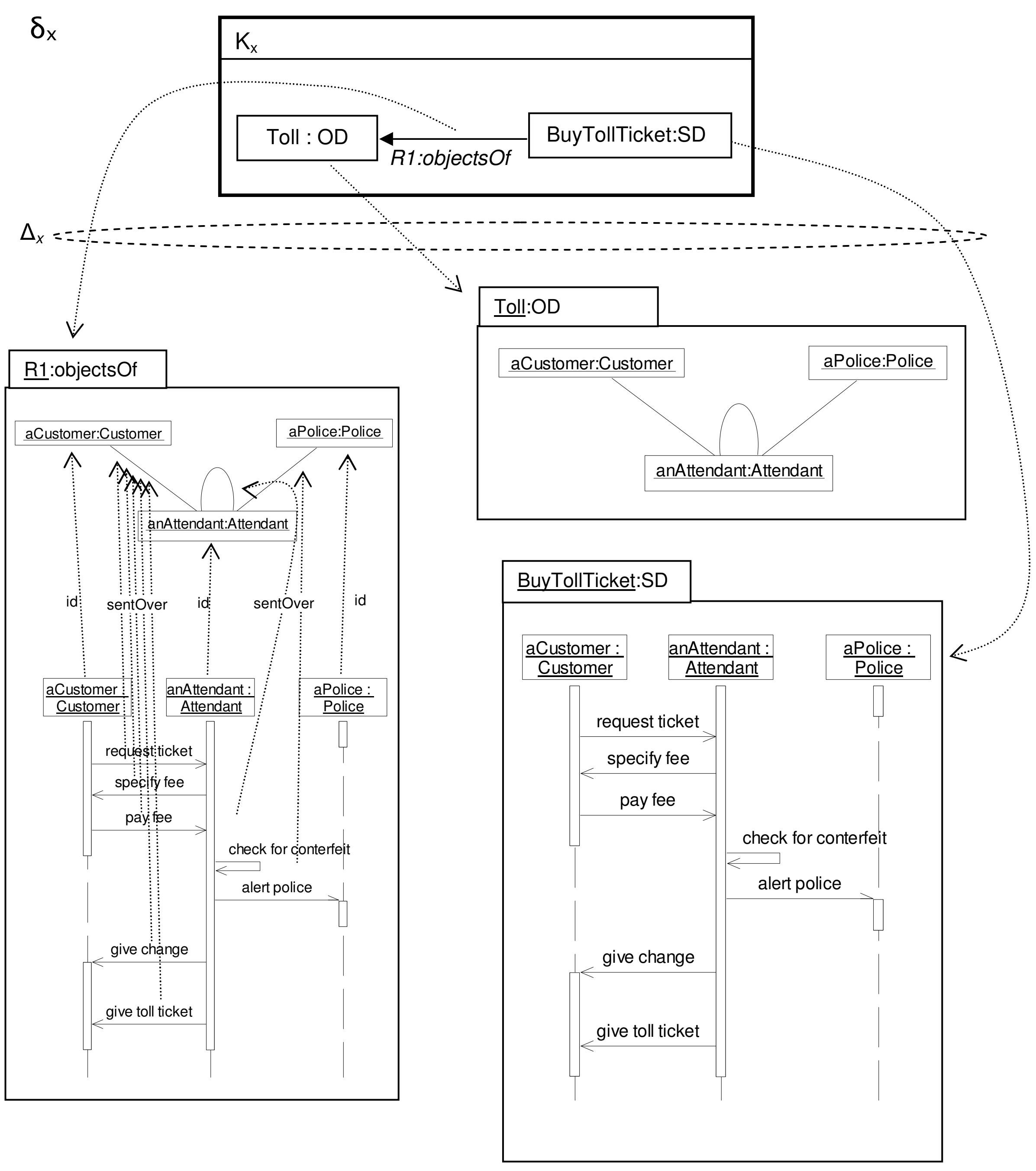


Figure 6.14. Example configuration δ_x .

Extension-to-conformance mode

In conformance checking mode, when the existential intent is not satisfied in condition C1 because some required models or mappings do not exist, it is still possible to do limited checking of condition C3 by extending the existing configuration δ to one that is conformant. This can be further generalized to allow existing but incomplete models and mappings to be extended to conformance. There are two important ways to use this mode:

- Synthesis. When the set of models and mappings are extended as part of finding a conformant solution they are guaranteed to be consistent with the existing models and relationships (in the sense that all constraints are satisfied) but this does not mean that what they express is necessarily correct since there may be many possible consistent extensions. When the solution is unique, however, then it *must* be correct and hence this provides a way to do model and mapping synthesis. For example, if we consider the mapping in the configuration δ_{x1} above to be incomplete (and thus, extendable), it is clear that there is only one possible extension that will satisfy the *objectsOf* constraints – the one that restores it to δ_x by putting back the missing *sentOver* links
- Conformance checking with incomplete information. If a conformant extension cannot be found, this indicates that there is no way to consistently extend the incomplete models and relationships and hence the existing artifacts can be determined to be definitely non-conformant even though they are incompletely specified. For example, consider configuration δ_{x2} constructed from δ_{x1} by modifying the object diagram to remove the link from *anAttendant* to *aPolice*. Now

there is no conformant extension of the mapping and we can conclude that the models (and partial mapping) are definitely non-conformant with the constraint that the *objectsOf* relationship holds between the models.

The examples in both these cases show that it is possible to work usefully with incomplete (or even non-existent) mappings. This is significant since the creation of mappings is often given little attention in the modeling process because it requires significant additional effort.

Change propagation mode

Say that configuration δ satisfies the semantic conditions. Now assume a change is made to a subset of the content in the models and mappings to form $δ_1$ that is no longer conformant. In this case, we want to find a minimum "repair" of the remainder to form a new configuration $δ_2$ that is conformant. In general, the minimum repair may not be unique but there always exists a unique (possibly empty) subset that is common to all minimum repairs. This subset represents the change that is propagated due to the constraints expressing modeler intent. As an example, consider that we begin with configuration $δ_x$ and then construct $δ_{x1}$ by changing the mapping <u>*R1*</u> to remove all the *sentOver* links as before. Now, every minimal repair that preserves this changed mapping must remove all links in <u>*Toll*</u> and all messages in <u>*BuyTollTicket*</u>. Thus, these changes are the result of propagating the effect of the change to <u>*R1*</u>.

Top-down modeling mode

Another mode of use for a macromodel is to support model evolution by using it to drive top-down modeling. In this case, the definer can use the macromodel to define the intended structure of a collection of models before they exist and hence provide a template for modelers to follow. For example, if configuration δ_x is missing both models and the mapping then the macromodel K_x still provides information about what these are intended to be.

6.4 Macromodel language specification

This section is a reference guide to the macromodeling language and provides detailed descriptions of the elements in the metamodel (Figures 6.5 – 6.7). Each subsection describes one or more related types of elements, gives their well-formedness conditions ("Constraints" subsection) and their concrete syntax ("Notation" subsection).

6.4.1 MacromodelKind and its subclasses

A *MacromodelKind* consists of a set of *ContainedMember* elements which represent the different kinds of constituents of a macromodel. A *RootMacromodel* has no container and could be used either as a general macromodel (*use* = *general*) or as ground macromodel for a particular project (*use* = *ground*). A *ContainedMacromodel* is one that is within another macromodel and *ContainedMacromodelType* represents a set of contained macromodels with upper and lower multiplicity bounds within the containing macromodel.

Constraints

// A *MacromodelKind* cannot contain itself
$\forall m$:MacromodelKind · $\neg$TC(contents(m, m))

// Only a *MacromodelKind* can specialize a *MacromodelKind*
$\forall m$:MacromodelKind,r:Member ·specialize(r, m) $\Rightarrow$ $\exists m_1$:MacromodelKind · $m_1 = r$

// Nothing can specialize a ground *RootMacromodel*
$\forall m$:RootMacromodel · use(m) = *ground* $\Rightarrow$ $\neg\exists m_1$:MacromodelKind · specialize(m_1, m)

The constraints for macromodel specialization are given in Section 6.2.2.

Notation

A macromodel is represented as a box with a thick border containing its constituent roles. A box with a single compartment containing a *macromodel-tag* is a reference to a contained macromodel detailed in another diagram. If there are multiple compartments there are several additional notational options:

- An optional *macromodel-tag* is found in the top compartment of the box. If a *macromodel-tag* is not given or if the *macromodel-name* is omitted, then the macromodel is considered *anonymous*.
- The contents of a macromodel can optionally appear within a compartment of the box. In this case, the compartment is assumed to show the complete contents of the macromodel unless the string "{inc}" appears in the *macromodel-tag* indicating that there is additional content not shown in the diagram.
- The contents of a macromodel can optionally be shown using links with a diamond arrow head from the boxes representing constituents to a macromodel box. In this case, the set of links is assumed to show the complete contents of the

contained macromodel unless the string “{inc}” appears in the *macromodel-tag* indicating that there is additional content not shown in the diagram.

Only the compartment method of showing content can be used to show a contained macromodel that is empty (i.e., the macromodel is complete but has no contents). In general, the compartment method is the preferred approach and the link method is only used in cases where it is visually difficult to use the compartments – e.g., if there are overlapping contained macromodels. Figure 6.15 illustrates the variations here. *TransportationProject* is a root macromodel showing the incomplete content using a compartment. Within *TransportationProject* there is the contained macromodel *Toll* using the compartment method, the contained macromodel *Family Travellers* using the link method, a referenced macromodel *Roads* and an anonymous macromodel consisting of two object diagrams.

If the macromodel specializes another macromodel then a *generalization* section is included in the *macromodel-tag* to reference the specialized macromodel. Specialization of macromodels is indicated by specifying the more general macromodel in parentheses. The attribute *use* = *general* is indicated with the stereotype <<gen>>, otherwise the macromodel is assumed to be ground. Figure 6.3 shows a use of a general macromodel and a specialization of it. Figure 6.17 shows a macromodel type and a specialization of it. The other stereotypes <<view>> and <<+view>> are discussed further in the Section 6.4.8.

tag format:

```
<macromodel-tag> ::= [<macromodel-name>]
                      [<generalization>]
                     [<multiplicity>]
```

```
                        [<stereotype>]
                        ["{inc}"]
<stereotype> ::= [<macroview-stereotype>]["<<gen>>"]
<macromodel-name> ::= STRING
```

6.4.2 Member, ContainedMember

Member is an abstract metaclass that defines a type of role within a macromodel. A *Member* asserts a set of role constraints that express modeler intent. Semantically a role is played by an entity – the type depends on the kind of role. Conformance semantics require that the entity playing the role satisfy the constraints owned by the role type. More specific semantics are given for the concrete subtypes.

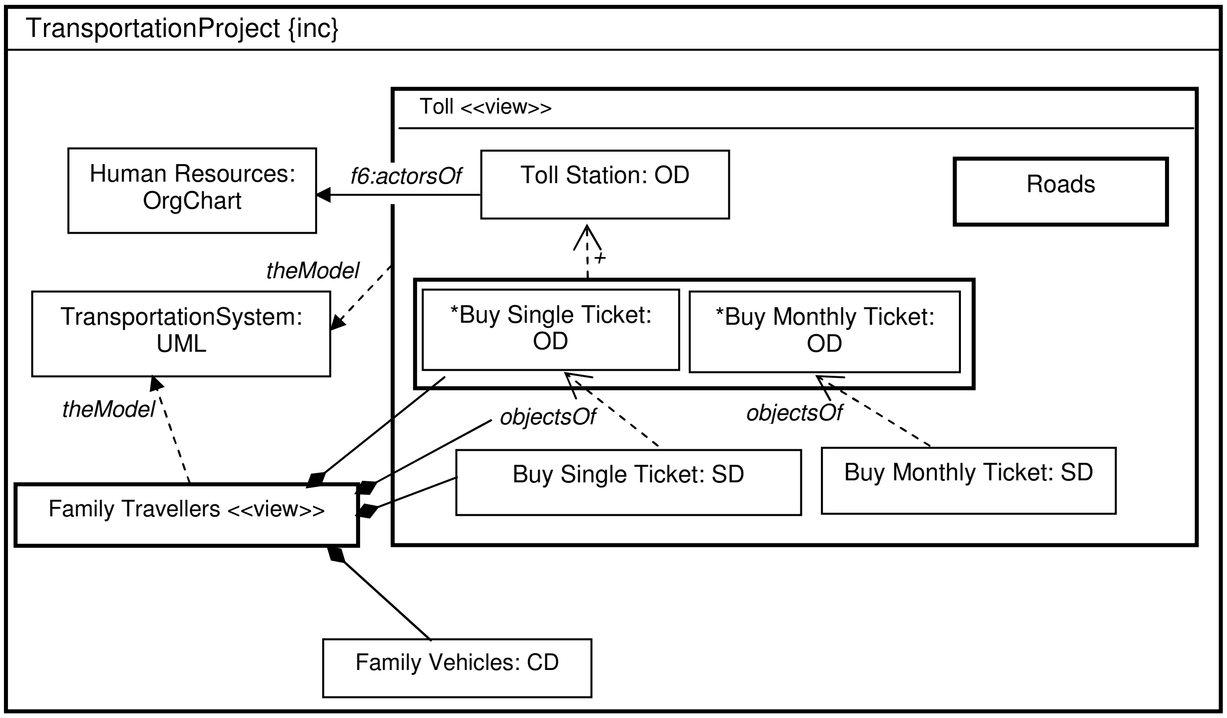


Figure 6.15. Examples of macromodels.

Constraints

// isAnonymous is *TRUE* iff no name is given
$\forall z$:ContainedMember · isAnonymous(z) $\Leftrightarrow$ DEFINED(name(z))

// If this specializes another role type then the multiplicities must at least as restrictive
$\forall z, z1$:MultiplicityElement, ContainedMember· specializes(z,$z1$) $\Rightarrow$ upperMultiplicity(z) $\leq$ upperMultiplicity($z1$) $\wedge$ lowerMultiplicity(z) $\leq$ lowerMultiplicity($z1$)

Notation

The notation is given for the concrete subtypes. Since a *MacromodelKind* is a namespace, the name can be the same as in other *MacromodelKind*. The name can also be omitted and then it is an *anonymous* role type. The rules for anonymous roles depends on the concrete subclass. A role can optionally specialize another role. Specialization is indicated by specifying the name of the more general role in parentheses as part of the role tag. The precise position of this is given by the concrete subclasses.

tag format:

```
<generalization> ::= “(“ <role-name> “)”
<role-name> ::= STRING
```

6.4.3 ModelRole, ModelRoleType

A *ModelRole* represents the use of a model with a given model type for a particular purpose. If the model role is realized (*isRealized* = *TRUE*) then it should contain a reference to the model in the project that plays it. If it is unassigned then this is considered to be a violation of the existential intent that such a model must exist in the project. To be conformant with modeler intent this model must conform to the model type and the role constraints associated with the model role.

A *ModelRoleType* represents a set of model roles within its containing macromodel with size that is within the specified multiplicity. A model role is a "ground" model role type in the sense that its multiplicity is "1" and the multiplicity is "1" for every macromodel in its containment path to the root macromodel and the root has *use* = *ground*. A ground model role type cannot have any specializations. Note that it is possible for the multiplicity of a model role type to be "1" and it not be ground. For example in Figure 6.3, *ReqSpecification* and *Design* have multiplicity "1" but are not ground since they occur in a general macromodel.. A model role type element also contains *ElementSet* and *ElementRole* elements and is a namespace for these.

Constraints

// For a ModelRole, the lowerMultiplicity and upperMultiplicity must be 1
$\forall z$:ModelRole$\cdot$ lowerMultiplicity(z) = 1 $\wedge$ upperMultiplicity(z) = 1

The constraints for model role type specialization are given in Section 6.2.2.

Notation

A *ModelRole* is represented as a box with a thin border containing a *model-tag* that identifies the role type, its generalization and model type. Starting the tag with an asterisk indicates that the *isRealized* attribute is set to *FALSE*. When a role is unrealized, its name is optional and if missing, the role is referred to as *anonymous*. The name/type pairs for model roles are unique within the root macromodel. If multiple occurrences of the same name/type pair occur in a macromodel diagram, they refer to the same model; however, different anonymous model roles are considered to represent distinct models.

A *ModelRoleType* is represent as a model role except that it has a multiplicity specifier. The name of a role type is unique only within the macromodel in which it occurs. If the name is missing, the role type is referred to as *anonymous* and is identified by its model type – i.e., it is understood to be the entire set of model roles of the given model type occurring in its containing macromodel. Thus, there can only be at most one anonymous model role type with a given model type.

Figure 6.16 shows various examples of model role types. The anonymous :*OD* [1..*] indicates the set consisting of all object diagram model roles in the macromodel and it also asserts that there must be at least one of them. *Buy Monthly Ticket* specializes the set *Basic* and is related to two unrealized model roles, one of which is anonymous.

tag format:

```
<modelroletype-tag> ::= [<role-name>]
                        [<generalization>]
                        “:”<model-type>
                        <multiplicity>
                        [<view-stereotype>]
<modelrole-tag> ::= ((“*” [<role-name>] ) | <role-name>)
                    [<generalization>]
                    “:” <model-type>
                    [<view-stereotype>]
<role-name> ::= STRING
<model-type> ::= STRING
```

6.4.4 SimpleRel, SimpleRelType, RelType

A *SimpleRel* identifies an intended relationship between a tuple of model roles. A relationship expresses a set of constraints that must hold between a tuple of models and optionally, a mapping that relates their content. The relationship carries a reference to a *ModelRelType* element representing a relationship type. If the relationship is realized (*isRealized* = *TRUE*) then it should contain a reference to the mapping (i.e., relator model) in the project. If it is unassigned then this is considered to be a violation of the existential intent that such a mapping must exist in the project. In addition, since every

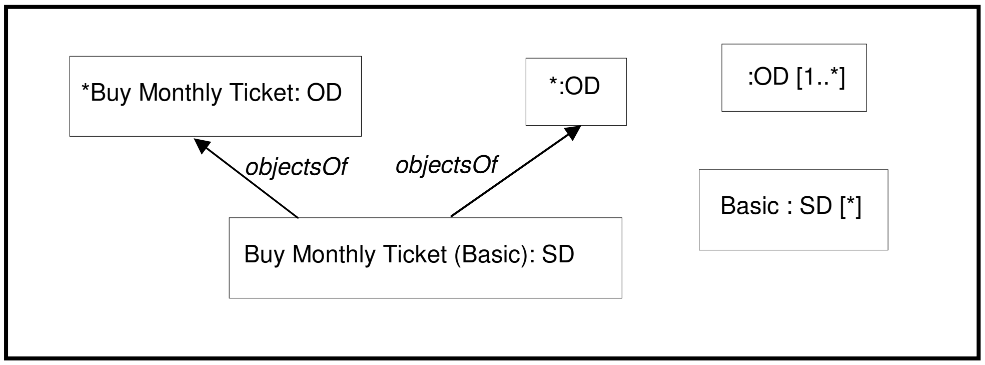


Figure 6.16. Examples of notation for model roles and role types.

role can have owned role constraints, these are considered to be conjunctively combined with the constraints in the relationship type and with the constraints in *SimpleRelType* elements that this relationship (optionally) specializes. The tuple of models and mappings conforms to the relationship iff they satisfy this resultant constraint.

A *SimpleRelType* represents a set of *SimpleRel* elements within a macromodel. The endpoints specify model role types and can give the multiplicities (multiplicity "1" is assumed if omitted). A *RelType* is an abstract class representing sets of relationships between model roles.

Constraints

```
// If any of its ends are unrealized then the role relationship must be unrealized
  ∀r :SimpleRel ∃e:SimpleEnd.
     end(e, r) ∧ isRealized(role(e)) = FALSE ⇒  ¬isRealized(role(r)) = FALSE
```

Notation

The notation of a relationship is determined by the characteristics of the associated relationship type. It is represented with a line, if binary, or as a diamond with *n* ends, if the relationship type is *n*-ary. A unary relationship type (i.e., a model property) is expressed as a diamond with one end. The relationship role has an associated *relationship-tag* that contains its name, type, generalization and an asterisk is used to indicate the *realized* attribute. In addition, the following apply:

- The ends can be decorated with optional name labels corresponding to the roles on the ends of the associated relationship type.
- Since ends are ordered, one end may optionally be decorated with a filled-in arrow head to indicate that is the last end. For binary relationships this indicates directionality for the relationship if the direction is meaningful.

- When the relationship type is *pure* then the name and colon is optional. This is the case because there cannot be multiple instances of the relationship type between a given set of arguments and so type name is sufficient to uniquely identify it.
- When the relationship type is a *transformation* then the name and colon is optional since it is *pure*. In addition, the output end(s) are decorated with an open arrow head.

Figure 6.17 shows some example variants. Here, all relationship types are pure except for *caseOf*. The class diagram *DVehicle* is constrained by having a *submodelOf* relationship to *DTransport*, being the result of the transformation *classDetails* that takes two arguments and having the unary relationship (property) *isRooted*. Since the macromodel *MX* specializes *Standard,* this introduces model role types *Types*, *Basic* and *Core* that are specialized by some model roles of *MX*. *Standard* contains the constraint that every *Basic* sequence diagram must have a *caseOf* relationship to a *Core* sequence diagram.

tag format:

```
<relationship-tag> ::= <mapping-tag> | <pure-tag>
                        [<view-stereotype>]
<mapping-tag> ::= ((“*” [<relationship-name>])
                     | <relationship-name>)
                     [<generalization>]
                     “:” <relationship-type>
<pure-tag> ::=  <relationship-type>
<relationship-type> ::= STRING
<relationship-name> ::= STRING
```

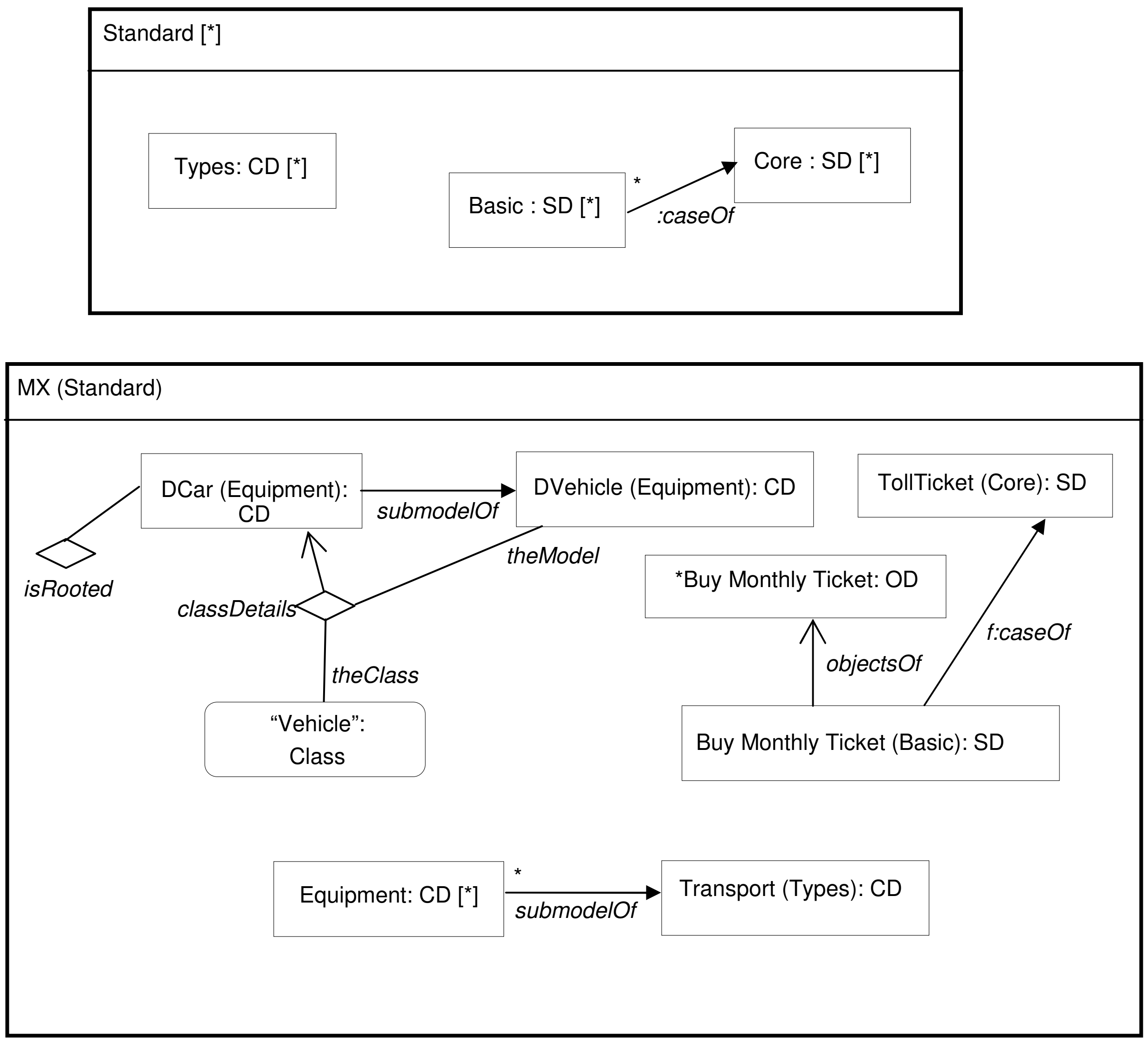


Figure 6.17. Examples of notation for relationships.

6.4.5 EndType, SimpleEnd, SimpleEndType

An *EndType* represents an endpoint of a role relationship type. It carries the multiplicity of the end as well as the name used to identify then end. A specialization *SimpleEnd* is the special case when it is an endpoint of a relationship that directly references a model relationship type. In this case, it has multiplicity “1” and it carries a reference to the *ModelRelEndType* for the relationship type.

Constraints

// For a SimpleEnd, the lowerMultiplicity and upperMultiplicity must be 1
$\forall e$:SimpleEnd $\cdot$ lowerMultiplicity(e) = 1 $\wedge$ upperMultiplicity(e) = 1

Notation

The notation of endpoints is part of the notation of relationships. See *SimpleRel* and *RelType* for examples.

6.4.6 MacroRelType, MacroRel

A *MacroRel* represents a macromodel that acts like a relationship – a macrorelationship. A macrorelationship has ends like a simple relationship except that it aggregates a collection of model roles and relationships within it that express the details of the macrorelationship. The endpoints of macrorelationship need not be macromodels but they could be. The endpoints of all relationships contained within macrorelationship must either be in the macrorelationship or within one of its endpoints. A *MacroRelType* represents a set of macrorelationships with the same structure. The set size is bounded by the upper and lower multiplicity within the containing macromodel. In addition to its benefits as an aggregating abstraction, a macrorelationship type provides a way of defining new relationship types from existing ones using macromodels rather than creating a relator metamodel.

Constraints

```
// The endpoints of contained rels must be in the macrorel or its endpoints
    $\forall r$:MacroRelType, $r_1$:RelSet · in($r_1$) = $r \Rightarrow$
        $\forall e$:EndType · end($e$, $r_1$) $\Rightarrow$
            TC(parent(member($e$), $r$)) $\vee$
            $\exists e_1$ :EndType · end($e_1$, $r$) $\wedge$ TC(parent(member($e$), member($e_1$)))

// define derived association parent
    $\forall r_1$ : ContainedMember, $r_2$ : Member · parent($r_1$, $r_2$) $\Leftrightarrow$ $r_2$ = in($r_1$)
```

Notation

A macrorelationship is represented using the same conventions as a relationship except that the line is thick (same thickness as macromodel borders). The content of a macrorelationship, when it is shown, is indicated with a dotted lasso. Any element that is contained within or intersects with the lasso is considered to be within the macrorelationship. If the content is shown but is incomplete then the "{inc}" designator is used in the tag. Figure 6.18 illustrates the use of a macrorelationship. The lower macromodel diagram shows two macrorelationships *objectsOf* and *caseOf* with their content not shown. The upper diagram shows the content using lassos. Note that *objectsOf* is a composition of transformations and so it is a transformation itself and it is shown as such. If a relationship endpoint intersects with the lasso then the macrorelationship must also have an endpoint on it or on a container of it. This is shown in both macrorelationships since they begin on a macromodel containing the source endpoints and end on models that are the target endpoints.

tag format:

```
<macrorelationship-tag> ::= [<relationship-name>]
                           [<generalization>]
                            [<multiplicity>]
                            [<macroview-stereotype>]
                            ["{inc}"]
<relationship-name> ::= STRING
```

6.4.7 ElementSet, ElementRole

An *ElementRole* represents a specific model element, optionally, with a given name. However, since macromodels can only reference models, the element role is really a shorthand for a singleton model reference. Specifically, element role $E:T_E$ is equivalent to the unrealized model role $*E:One[T_E]$ where the element is constrained to have the same name as E if it has a name (otherwise it is an anonymous element role). If the element role is put inside a model role $M:T$ then this expresses the constraint that there must exist a an element of type T_E with name $name(E)$ in the model playing M and this element is considered to be the player of element role E. Thus, T_E must also be a sort in the metamodel corresponding to T.

An *ElementSet* represents the entire set of elements of a given type within a model role. The intent is to provide a way to reference the set of elements for a given sort in the metamodel of the model. An element set has no name, only a type and no multiplicity can be specified. Note that unlike a model role type which represents a set of model roles within a macromodel, an element set represents the set of elements in the model playing a model role and not the set of element roles within the model role itself. We give an example below in the notation section to clarify this distinction.

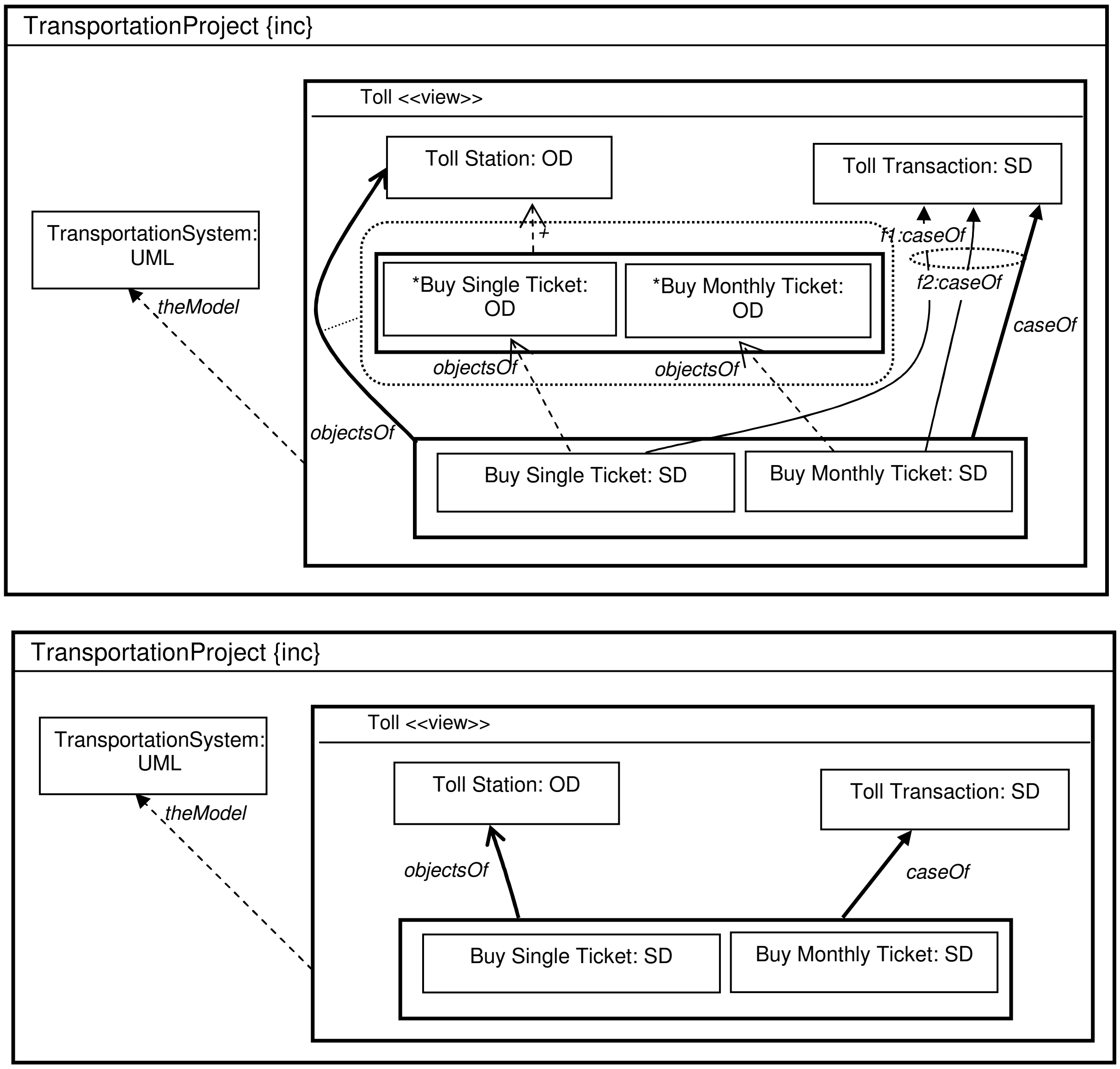


Figure 6.18. Examples of macrorelationships.

Notation

Both element roles and sets are expressed as a rectangle with rounded corners within a compartment of the containing model role. Element roles are always considered to be unrealized but it can be prefixed with "*" for clarity. Although multiplicities cannot be expressed for an *ElementSet*, a pseudo multiplicity indicator "[*]" is used to identify it as a set and distinguish it from an *ElementRole*. Figure 6.19 shows an example in which the sequence diagrams in set *Cond* within *RoadControl* are generated by classes in a class diagram *RoadCondition*. The element set *Class* represents the set of all classes in the model that plays the role *RoadCondition*. Two specific classes are represented by element roles and linked to specific sequence diagrams. Thus, any model that realizes *RoadCondition* is required to have at least two classes with these names. In contrast, the type *CondCase* represents the set of all model roles within *RoadControl* and in this case this set consists of {*M1*, *M2*}.

tag format:

```
<elementrole-tag> ::= ["*"][<element-name>] ":" <element-type>
<elementset-tag> ::= <element-type> "[*]"
<element-name> ::= STRING
<element-type> ::= STRING
```

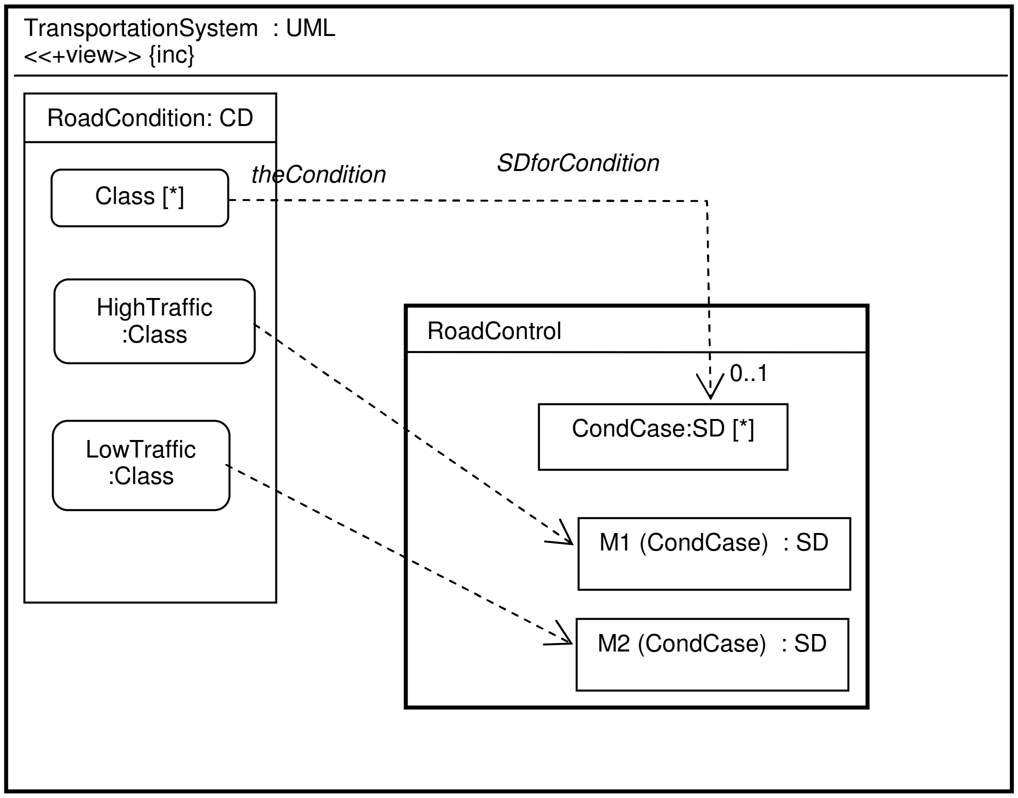


Figure 6.19. Examples of element roles and element role types.

6.4.8 View

Any *ContainedMember* in a macromodel can behave as a view of a base model. To express this, a *View* acts as a container that holds another role and "converts" it into a view. In this case it means that the full content of role (i.e., the content in any models or relationships playing this role or roles within this role) is a subset of the content of the base model. A view is associated with a set of generators, a base model given by *theModel* and a distinguished role constraint representing the content criterion of the view. A view that holds a macromodel is a presentation and if *isComplete* = *TRUE* then it is a decomposition. The *theModel* designator is optional only if the view is within another element that is a view. In this case *theModel* for the inner view is assumed to be the same as *theModel* for the containing view.

Constraints

// define the derived association parent

$\forall v_1, v_2\text{:View} \cdot \text{parentView}(v_1, v_2) \Leftrightarrow v_2 = \text{in}(\text{view}(v_1))$

// theModel is optional only for nested views

$\forall v_1\text{: View} \cdot (\neg\exists r\text{:ModelRoleType} \cdot \text{theModel}(r, v_1)) \Rightarrow$

$\quad\quad \exists v_2\text{:View}, r\text{:ModelRoleType} \cdot \text{TC}(\text{parentView}(v_1, v_2)) \wedge \text{theModel}(r, v_2)$

// isComplete can only be true for Macromodels

$\forall v\text{:View} \cdot \text{isComplete}(v) = \mathit{TRUE} \Rightarrow \exists r\text{:Macromodel} \cdot r = \text{member}(v)$

Notation

A view is designated by adding the stereotype "<<view>>" to the tag for the element. If *isComplete* = *TRUE* then the stereotype "<<+view>>" is used instead. The stereotypes can be omitted if the view is within a role that is itself in a *View*. The generators and the base model can be shown connected to the view via dashed arrows. In the case of a generator, the arrow points to the view and in the case of the base model it points to the base model. The generator endpoints can optionally be given a name and the base model endpoint always has the name *theModel.* When the extractor defining the content criteria of the view takes a single generator as an argument then it can be indicated on the dashed arrow for the generator.

A view that is a macromodel (i.e., a presentation or decomposition) can also be expressed in a way that combines the base model with the macromodel. In this case, the box has a thick border to indicate a macromodel but the tag compartment contains two lines. The first line is the base model tag and the second line is the macromodel tag. Additional compartments can apply to either the base model or the macromodel and should be ordered to show the base model compartments first and then the macromodel compartments. If the base model participates in a relationship then the relationship end is attached to the top compartment while if the macromodel participates in a relationship then the end is attached to the compartment showing the macromodel content. Figure 6.20 illustrates this.

tag format:

```
<macroview-stereotype> ::= <view-stereotype> | "<<+view>>"
<view-stereotype> ::= "<<view>>"
<combined-tag> ::= <modelrole-tag> EOL <macromodel-tag>
```

6.4.9 ModelRelType, ModelType, ModelRelEndType, ElementType, Morphism

The first four of these represent artifact types used by the roles in a macromodel. A *Morphism* contains a signature morphism between two metamodels. A *ModelType* that is concrete (i.e., *isAbstract* = *FALSE*) has a reference to the metamodel artifact that defines the model type. Similarly, a concrete *ModelRelType* carries a reference to a relator metamodel and the corresponding *ModelRelEndType* elements contain *Morphism* elements that map the relationship endpoint metamodels. An ElementType is a proxy for a sort in a metamodel of a model type and it can occur in multiple metamodels.

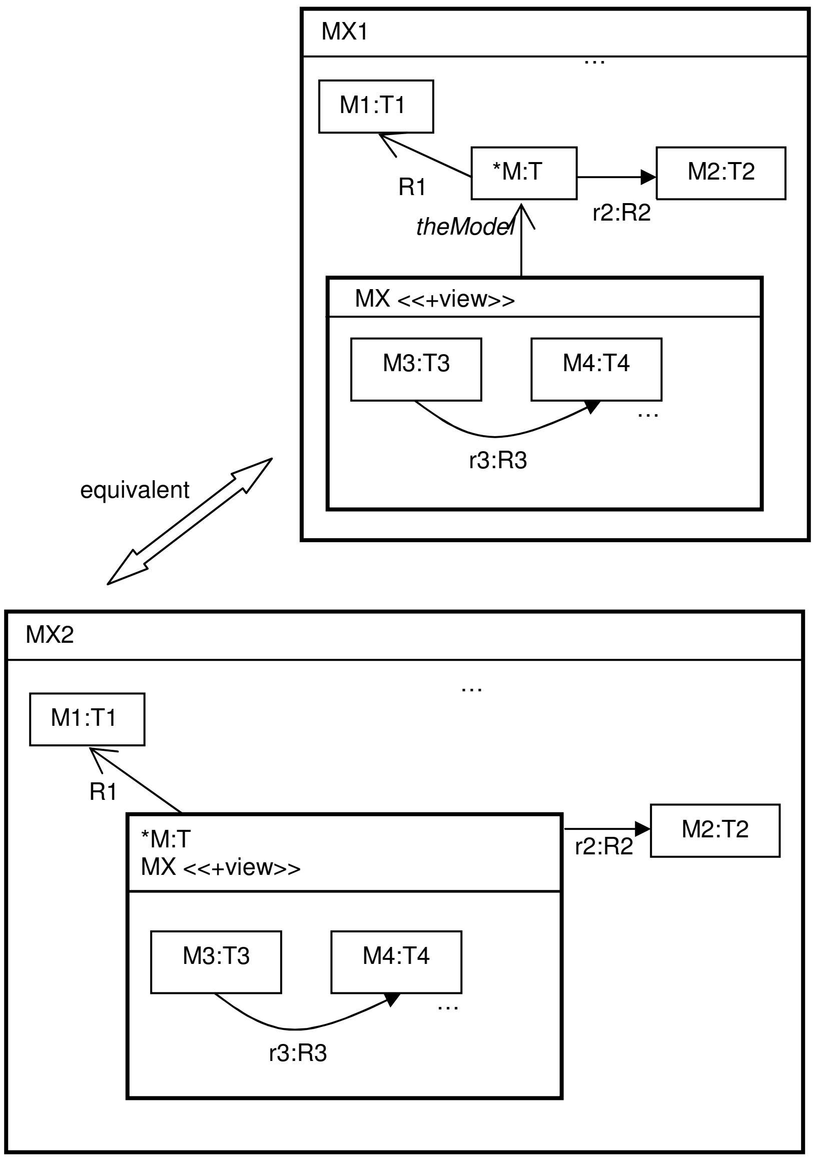


Figure 6.20. Example of a combined base model and decomposition view.

Notation

These elements are normally shown in a separate diagram from macromodels as a *type diagram*. A type diagram is shown simply as a UML style class diagram with no multiplicities. Abstract types have names italicized and subtyping is shown with the triangle headed arrow. Figure 6.4 is an example of a type diagram. Types can show the stereotype "<<type>>" when they are shown within a macromodel diagram but may omit them in a type diagram. Relationship type arrow conventions are the same as for relationships in a macromodel diagram. Metamodel morphisms are shown with stereotype "<<morphism>>". The concrete elements can show the reference to the corresponding metamodel artifact (not shown in Figure 6.4) .

6.4.10 ContentConstraint

This is an abstract metaclass for constraints owned by a role. Each type of constraint language may define its own concrete subclass to define a constraint notation. Owned constraints are assumed to be conjoined with each other and with the constraints that are inherited from roles that are specialized.

Notation

Constraints owned by a role can optionally be specified directly within a notation element. The approach varies with the kind of role. For a model role, the constraints appear within a compartment in the model role box. Similarly for a macromodel. For a relationship role, constraints appear in brace brackets near the *relationship-tag*. Each constraint is prefixed by a *language-identifier* that indicates the constraint language used. If an owned constraint is prefixed with “cc:” then it is assumed to be part of the content criteria for a view.

We do not elaborate the many different possible constraint languages as it is beyond the scope of this thesis. Rather we focus on two. The first is to use FO+ as a constraint language. An example of this is shown in Figure 6.1. Note that the sentence is assumed to be defined over the signature of the model type of the owning role (in this case, *SD*). The second constraint language of interest is to use the macromodel language itself as an owned constraint. Figure 6.2 shows an example. A macromodel such as this uses the following conventions:

- Local roles may be expressed along side other model roles. A local role is a role that is named relative to the owning role. The pseudo role *Self* is always available as a local role and it refers to the owning role. If the owning role is a view then each generator name can be used as a local role.

- Roles that are referenced but are external to the owning role have a dashed border.
- Only pure (or unrealized) relationships can be used to express the constraint.

tag format:

```
<constraint> ::= language-identifier [“cc”] “:” constraint-body
<language-identifier> ::= STRING
<constraint-body> ::= STRING
```

6.4.11 MultiplicityElement

An abstract class representing elements with multiplicities.

Constraints

//The lowerMultiplicity must be less than the upper multiplicity
$\forall$ *z*:MultiplicityElement$\cdot$ DEFINED(upperMultiplicity(*z*)) $\Rightarrow$ lowerMultiplicity(*z*) $\leq$ upperMultiplicity(*z*)

Notation

A multiplicity specifier can optionally occur in the name of instances of concrete subclasses of this class. The particular context of occurrence is given by the concrete subclass. An omitted multiplicity specifier indicates the multiplicity: *upperMultiplicity* = 1, *lowerMultiplicity* = 1

tag format:

```
<multiplicity> ::= “[“ <multiplicity-val> “]”
<multiplicity-val> ::= “*” | “0..*” | “1..*”
                     | NUMBER | NUMBER “..” NUMBER
```

6.5 Summary

In this chapter we have presented the macromodeling language as a way of modeling the role level. As such it both provides a mechanism for expressing modeler intent at the project or method level and allows this to be represented graphically. Special support is given for the various aspects of modeler intent discussed in this thesis. This includes:

views, content criteria, decomposition criteria, model roles, role types, relationships and relationship types. Macromodels can be used to support comprehension of a collection of models in a project, to guide their evolution and to support automation.

Chapter 7

Decomposition Criteria

In the intent framework, a consumer's need for information gives rise to an existential intent for a model. However, this usually does not mean that the information must be delivered as single artifact – rather, it is often presented as a decomposition consisting of a collection of interrelated models. Thus, decomposition is a natural and ubiquitous part of modeling activity. As discussed in the Chapter 2, there are several reasons why a modeler may want to decompose a model: the model may be decomposed into smaller parts and into different levels of abstraction in order to manage complexity, it may be split into other types of models with well defined notations in order to render it when it has no single notation (e.g., with UML), it may be partitioned in order to support some task such as assigning the parts to different teams, etc. Furthermore, this reason may underdetermine the particular decomposition chosen and so the intent may include design decisions about the decompositional structure as well.

In this chapter, we explore the part of the intent framework that addresses how the modeler can express their intent about model decompositions and refer to expressions of this intent as *decomposition criteria*. In Chapter 4 we noted that different partiality relationship types can be used to decompose a model in different ways such as into “parts” (i.e., submodels), abstractions, etc. Here we follow Chapter 5 by focusing on submodels and study the common case of decompositions into submodels or diagrams. As with other types of modeler intent discussed in this thesis, explicit and formal decomposition criteria provides value by improving the consumer’s comprehension of a model decomposition, by improving the quality of the model and its decompositions and by supporting automation and model evolution. In particular, we develop the notion of *indexed* decomposition criteria as a way of decomposing a model that has similar comprehension benefits to classification schemes in information classification systems. We then identify three types of modeling defects that indexed decomposition criteria helps detect. Finally, we discuss the applicability of decomposition criteria to a variety of model automation and evolution scenarios.

7.1 Basic approach

In Chapter 5, we developed the idea that when a modeler is creating a diagram (or any submodel) of a model, she is following some principle for deciding what information from the model belongs in the diagram and what does not. Thus, the modeler intends that the diagram represents a particular view of the model and this is expressed using a query-like role constraint called the content criterion. In a similar manner, we argue here that whenever a modeler decides to decompose a model, she does not break it arbitrarily into submodels but rather, she is following some principle that guides how to do this. Furthermore, it is not enough that this principle simply yield submodels; rather, it must also define the content criteria for each of these, since, like all models, each must have a well defined purpose. Thus, the principle must actually yield a set of views of the model.

As a simple illustration, consider the decomposition shown in Figure 7.1 from the transportation system example. Here, the macromodel *DTollPriceX* expresses the decomposition of the model role *DTollPrice* into the following set of views: *BaseClasses* shows its base classes, *DVehicle* shows the details for the class *Vehicle* and *DTollTicket* shows the details of the class *TollTicket*. *DVehicle* is further decomposed into two views: the vehicles related to the class *Commercial User* and the vehicles related to the class *Non-commercial User*. The definitions of the extractors *top*, *relatedTo* and *classDetail* used in these views are given in Appendix E.

While Figure 7.1 does depict a decomposition of *DTollTicket*, it does not give any indication of what the intention was regarding how the constituent views in *DTollPriceX* where chosen – i.e., it does not contain decomposition criteria. Now consider the assertion "*DTollPriceX* is a presentation of *DTollPrice* containing: the *top* view *BaseClasses* and a *classDetail* view for each class in *BaseClasses*". This assertion is the decomposition criterion for *DTollPriceX* and it does two things:

1. It specifies how to construct a particular set of constituent views in *DTollPriceX* for each realization of *DTollPrice*.

It provides a guarantee that the realizations of these views will decompose the realization of *DTollPrice* – i.e., that the *DTollPriceX* is a decomposition.

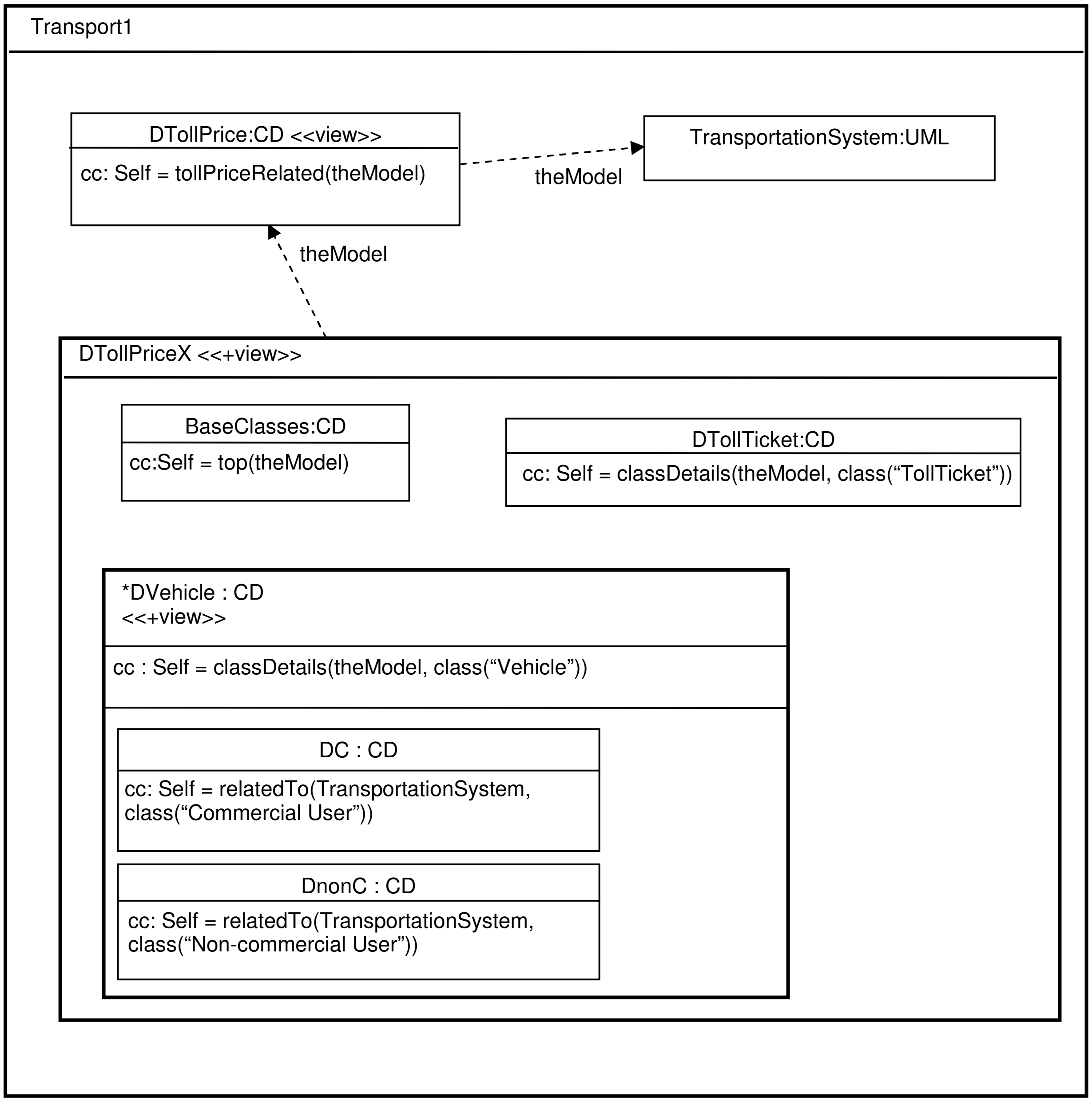

Transport1
DTollPrice:CD <<view>>
cc: Self = tollPriceRelated(theModel)
TransportationSystem:UML
theModel
theModel
DTollPriceX <<+view>>
BaseClasses:CD
cc:Self = top(theModel)
DTollTicket:CD
cc: Self = classDetails(theModel, class("TollTicket"))
*DVehicle : CD
<<+view>>
cc : Self = classDetails(theModel, class("Vehicle"))
DC : CD
cc: Self = relatedTo(TransportationSystem, class("Commercial User"))
DnonC : CD
cc: Self = relatedTo(TransportationSystem, class("Non-commercial User"))

Figure 7.1. A simple decomposition from the transportation system example.

Point (1) expresses the fact that the basis of the decomposition is “base class” and when the set of base classes change then so does the set of views. Point (2) comes from the fact that any class diagram can be decomposed on the basis of its base classes into a set of *classDetails* views. This suggests that a generic approach to decomposition is being applied here. Note that the decomposition criteria do not just define how each realization is *DTollPriceX* decomposed into submodels – it gives the content criteria of these as well since it defines the views they realize. In order to better understand this, consider a criterion like “*DTollPriceX* contains the set of all class diagram submodels of *DTollPrice* containing two or fewer classes.” This criterion defines a unique set of submodels for each realization of *DTollPrice* that decomposes it. However, it does not define the content criterion for each of these submodels and hence gives no indication of the intent behind the content for each. Thus, it would not count as a valid decomposition criterion.

Expressing the decomposition criteria provides a number of benefits and we discuss these in Section 7.3. We now develop the concept of decomposition criteria more formally.

7.2 Formalization

The objective of this section is to encode a decomposition criterion formally in order to be precise about its structure, allow the definition of validity conditions that must hold and for characterizing the types of defects that can be detected. As discussed in Chapter 6, a presentation is a macromodel that is also a view of some base model role *M* and it consists of a collection of views. A decomposition is a presentation that has the additional constraint that the sum of its constituent views is equal to *M*. We now define the following related concepts.

Definition 7.1. ***Presenters and decomposers*.** A *presenter* (*decomposer*) is a transformation that when applied to a model *<u>M</u>* produces a presentation (decomposition) of *<u>M</u>*. Thus, it is a transformation that generates a macromodel from a model.

While our main focus will be on decomposers and how these can be used to define decomposition criteria for particular decompositions, it is useful to understand these as being special cases of presenters and presentations. Thus, although a presentation is a kind of view of *M* that consists of other views, it alone contains no information about the *basis* on which the presentation is structured. In a way analogous to views, where an extractor was used to characterize modeler intent by specifying how a submodel is constructed from each realization of *M*, a presenter can be used to characterize modeler intent about a presentation by defining how to construct it from each realization of *M*. When we assert the constraint that a particular presentation *MX* is constructed from its base model *M* using a presenter *P*, then we call this constraint the *presentation criteria* of *MX*. When *MX* is a decomposer, this becomes its *decomposition criteria*. We now extend this analogy further by extending the inclusion constraint approach from views to presenters.

7.2.1 Inclusion constraints

With views, the content criteria operationalize the modeler's principle for deciding what information to include into a submodel and a natural way to express this is as a set of inclusion constraints. If we extend this idea to presentations, we could ask what sort of inclusion constraints would operationalize the modeler's principle for deciding what views to include into the presentation. To elaborate this we must be clear about what set we are including views from. Since we define views in terms of extractors, we need only focus on how to select the correct extractors. Let $Extractor_L$ define the set of all extractors in some language L for defining extractors. In this case, if we are defining the macromodel MX as a presentation of M, we can consider the inclusion constraint to be of the form:

$$\begin{aligned}&\forall f \in Extractor_L \cdot\\ &\quad \mathrm{DEFINED}(Q(f, \mathrm{M})) \Rightarrow\\ &\quad (\forall v{:}\mathrm{MX.view} \cdot (\mathrm{CC}_v := \text{‘}f(\mathrm{M})\text{’}) \Rightarrow Q(f, \mathrm{M})) \wedge\\ &\quad (Q(f, \mathrm{M}) \wedge f(\mathrm{M}) \neq \varnothing) \Rightarrow \exists v{:}\mathrm{MX.view} \wedge (\mathrm{CC}_v := \text{‘}f(\mathrm{M})\text{’})\end{aligned} \tag{7.1}$$

Here, $Q(f, M)$ is the condition that extractor f must satisfy according to the modeler's intention in order for a view with content criterion "$f(M)$" to be included in MX. Thus, the inclusion constraint says: if f satisfies condition Q and $f(M)$ results in a non-empty subset of M then MX must contain a view with content criterion "$f(M)$" and these are the only views it contains. The non-emptiness part of the condition is due to the intuition that we typically don't express submodels that are empty. For example, in Figure 7.1, even if our intent was to have a *classDetails* view for each base class, if this turned out to be empty for class TollTicket then it would be "ok" to not have a diagram for it at all (i.e., this should not violate the modeler's intent). This inclusion constraint defines the presentation criterion for MX and when MX and M are replaced by parameters it defines a presenter.

In Chapter 5, we considered the case of defining views with L = FO+. In that case, the language for expressing Q would need to express properties of FO+ theories. Although this is a general approach, it may be too complex for a practical application. Instead, we take a simpler approach by considering the classes of views that can be defined using parameterized extractors. In particular, we consider extractors with a single generator.

Given an extractor $F{:}T \times T_1 \rightarrow T_2$, we can define a class of F-views $Msub$ of M with content criterion $\mathrm{CC}_{Msub} := F(M, m_1)$ where m_1 is another model role. Thus, we can reduce the problem of defining Q to that of defining the set of model roles m_1. Although, this doesn't appear to reduce the problem significantly (because one set of roles is defined in terms of another), in practice it is useful. In particular, we can do the following::

1. Let m_1 be the set of element roles corresponding to the elements in some model role M_1 that is a view with content criteria Q
2. Let m_1 be the set of model roles in a macromodel MX_1

In both cases, we say that the set of views in MX is *indexed* by another model. In the first approach, the views are indexed by the set of elements of a given type within model M_1. This allows us to define Q simply as the content criteria of a model as described in Chapter 5. In the second approach, the views are indexed by "simpler" roles in another macromodel. Since this macromodel can itself be indexed by another model, it allows a composition of indexing that could be ultimately grounded in a simple model (i.e., the first approach).

When presentation criteria are given in this form we call them *indexed presentation criteria*. Correspondingly, if we add the decomposition constraint then they become *indexed decomposition criteria*. The example discussed in Figure 7.1 has indexed decomposition criteria at two levels. *DTollPriceX* is a decomposition indexed by *BaseClasses* and the decomposition of *DVehicles* is indexed by the set of “user types”. We elaborate this further below.

Although indexed decomposers are very restricted compared to the general form shown in formula 7.1, we have found that it sufficient for most common expressions of decomposition criteria. Furthermore, they have two important features. First, they are simple and natural to express graphically using the macromodel notation as we shall see below. This makes them practical to express during the modeling process. Second, they provide a level of abstraction that aids model comprehension because the elements of the index set could be seen as “classifying” the content of the corresponding view. We discuss this further in Section 7.3.1.

7.2.2 Graphical expression of decomposition criteria

Decomposition criteria as we have described it above can be directly expressed within the macromodeling language. The general form is shown in Figure 7.2. The dashed arrow from the role type $M_R.T_1$ to the role type $MX.R$ expresses the indexing relationship. Specifically, it indicates that:

1. There is an R view generated by each element $x{:}T_1$ of M_R for which $F(theModel, x)$ is non-empty.

2. Each *R* view is generated by a T_1 element of M_R using *F*.

3. The set of views in *MX* decomposes *M*.

Thus, *MX* can have a different set of *R* views for different realizations of M_R. Note that here we assume that M_R is a model role type and T_1 is an element role type (indicated by the rounded box); however, a similar pattern can be had with M_R as a macromodel and T_1 a model role type within it.

We now apply this graphical approach to the example in Figure 7.1. First consider the indexed decomposer *ByBaseClasses* in Figure 7.3. expressed as a macromodel type. This consists of a singleton view type *BaseClasses* and a view type *BaseDetail*, both having

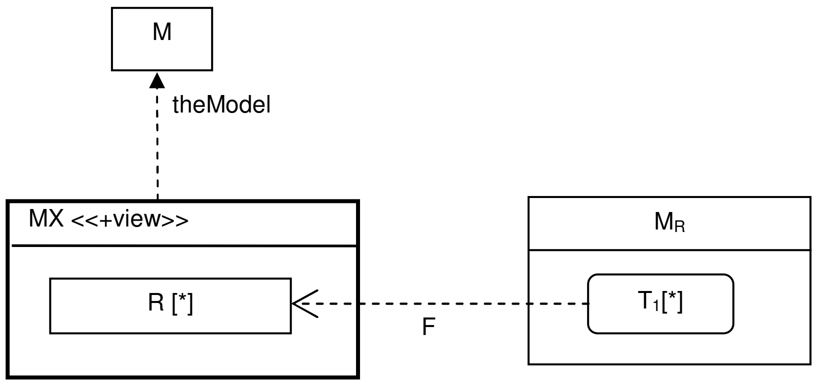


Figure 7.2. The general form for expressing decomposition criteria in a macromodel.

model type *CD*. Given any class diagram realizing *theModel*, there is a unique[26] conformant instance of *ByBaseClasses* and this contains the view *BaseClasses* showing the set of base classes in *theModel* and a set of *BaseDetail* views indexed by the classes in *BaseClasses* the show the details for each of these base classes. Furthermore, this macromodel is required to be a decomposition of *theModel*.

The *ByBaseClasses* decomposer is applied to *DTollPrice* in Figure 7.4 to express the decomposition criteria of the first level of the hierarchical decomposition. Once again *DTollPriceX* shows a decomposition of *DTollPrice*, but this time it is marked as an instance of the *ByBaseClasses* decomposer and its top level constituents instantiate the constituents of this decomposer. Both the views *DTollTicket* and *DVehicles* are instances

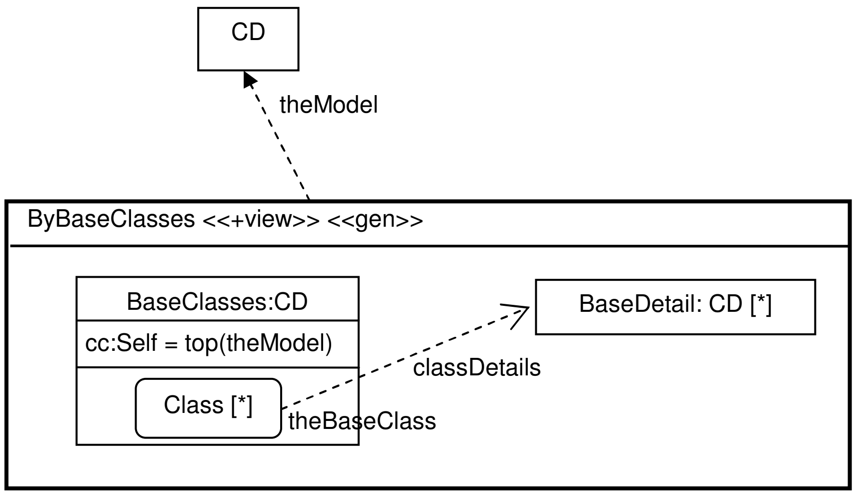


Figure 7.3. The generic base classes decomposer.

[26] Note that we mean uniqueness up to renaming of views – an arbitrary number of equivalent macromodels

of the view type *BaseDetail.* The decomposition criteria of *DVehicles* are also expressed using a decomposer but whereas *ByBaseClasses* is a global decomposer, this one is only locally defined. There is a single anonymous view type *CD*[*] representing the constituent views and this is indexed by the unrealized view *UserTypes* that shows the main types of users of a vehicle in the transportation system. The content criterion of *UserTypes* shows that it is the set of direct subclasses of the class *User* in the *TransportationSystem*. In the current example, this consists of *Commercial User* and *Non-commercial User* (see Appendix E).

The key difference between Figure 7.1 and Figure 7.4 is that the definition of the decomposer *ByBaseClasses* and the local decomposer of *DVehicles* make the basis for these decompositions explicit. Thus, it is now clear that the intent behind *DTollPriceX* is that it is decomposed first by its base classes and then *DVehicles* is further decomposed by the user type. In both these decompositions, the indexing set and its content criteria are made explicit whereas in Figure 7.1 it was not clear what these were. For example, if a new user type is added by adding a subclass to class *User* in *TransportationSystem*, the indexing constraint implies that *DVehicles* should have another view – this implication was not evident in Figure 7.1.

can be produced that have the same views but are named differently.

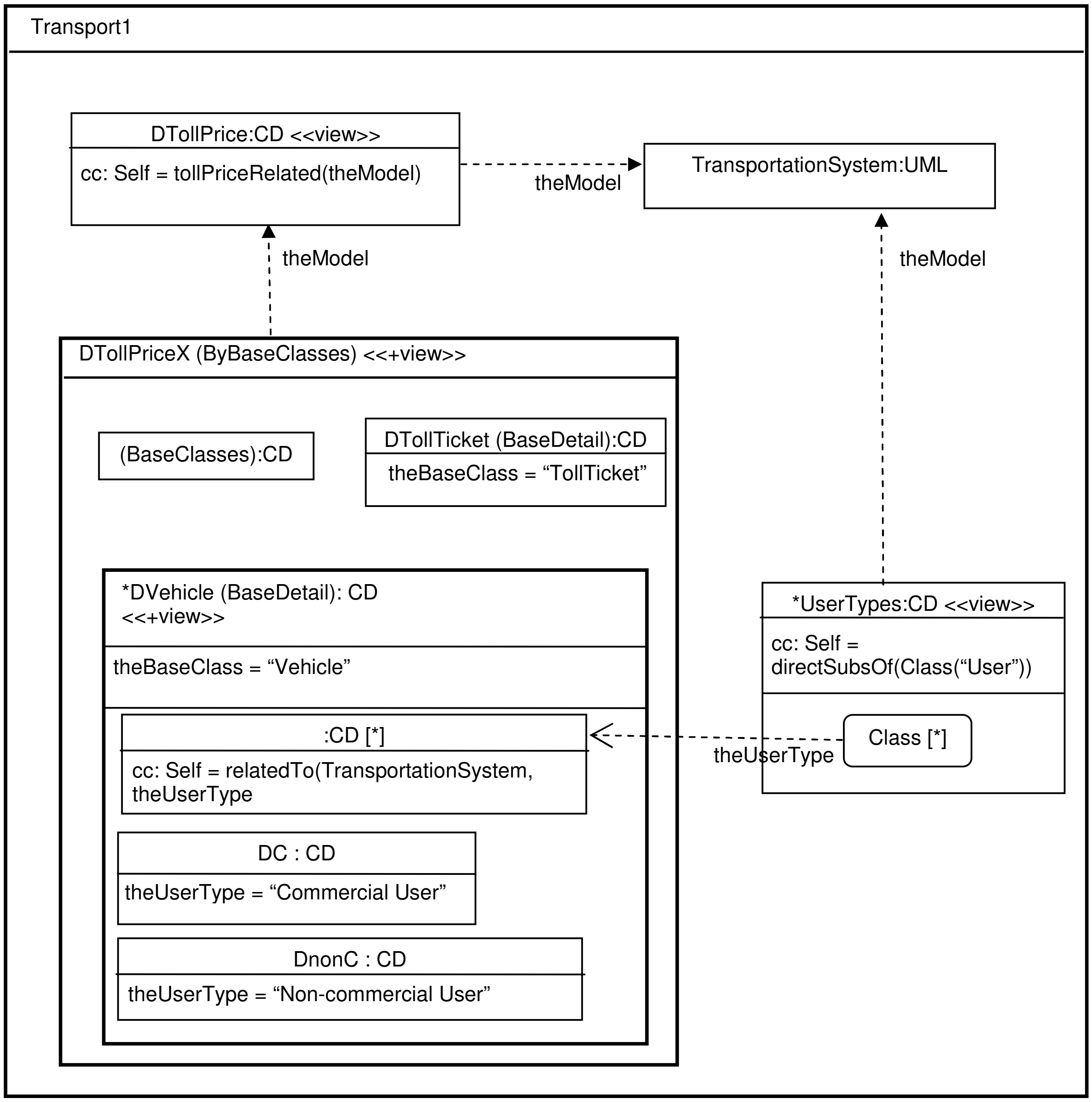

Transport1
DTollPrice:CD <<view>>
cc: Self = tollPriceRelated(theModel)
TransportationSystem:UML
theModel
theModel
theModel
DTollPriceX (ByBaseClasses) <<+view>>
(BaseClasses):CD
DTollTicket (BaseDetail):CD
theBaseClass = "TollTicket"
*DVehicle (BaseDetail): CD <<+view>>
theBaseClass = "Vehicle"
:CD [*]
cc: Self = relatedTo(TransportationSystem, theUserType
DC : CD
theUserType = "Commercial User"
DnonC : CD
theUserType = "Non-commercial User"
*UserTypes:CD <<view>>
cc: Self = directSubsOf(Class("User"))
Class [*]
theUserType

Figure 7.4. DTollPriceX augmented with decomposition criteria.

Another way that Figure 7.4 is different from Figure 7.1 is that in the former, the content criteria for the sets of the indexed views are specified only once in the view type whereas in the latter it is specified for each view. This yields two standard benefits of using typing for views: it reduces the effort required to specify the content criteria and it makes it clear that the content criteria for multiple views are intended to have the same form.

7.2.3 Characteristics of decomposers

Local vs. Global scope

Our example illustrates some important characteristics of decomposers. First we noted that *ByBaseClasses* was specified in a global scope independently of any particular use whereas the anonymous decomposer used in *DVehicles* was specified in a local scope and applies only to that one macromodel. Since the *DVehicles* decomposer is in a local scope it can reference any other model in the same scope. For example, this allows the content criterion of the view type to reference *TransportationSystem*. In contrast, *ByBaseClasses* can only reference its generators and *theModel*. However, this also means that *ByBaseClasses* can be reused in multiple contexts.

Generic vs. domain-specific decomposers

We can consider *ByBaseClasses* to be a generic decomposer because it can be applied to any class diagram to produce a *ByBaseClasses* decomposition of it. This is because every class diagram has a set of base classes, the set of *BaseDetails* views can always be constructed of these and they are guaranteed to decompose the class diagram. In contrast, the decomposer used in *DVehicles* is specific to the transportation system domain since it requires that the *UserType* model exists and that the set of views that are indexed by *UserType* yields a decomposition of *DVehicles*. Such a set of requirements are only satisfied by certain UML models.

The degree of generality of a decomposer is governed by its preconditions. A generic decomposer like *ByBaseClasses* has no preconditions. The benefit of generic decomposers is that they can be designed with good quality properties and then used in arbitrary contexts where a systematic decomposition is needed. The drawback is that they convey no domain semantics. It is typical to use a combination of generic and domain specific decomposers to structure a model and this is evident from the example applications in Chapter 8.

Tangling of subject with model

Decomposers allow the content of particular models in a project (i.e., the index models) to affect the macromodel structure and thus they represent a kind of tangling between the intention about the subject and intention about the model discussed in Chapter 2. For example, in the *ByBaseClasses* decomposer, the content of the submodel that realizes *BaseClasses* is used to determine what *BaseDetail* views will be in the resultant decomposition. This kind of tangling is not surprising because it is often the case that the intent of a modeler is to detail an element that occurs in another model and this gives rise to *detailOf* relationships discussed in Chapter 5.

7.3 Value of decomposition criteria

7.3.1 Impact on model comprehension

In Chapter 2, we discussed how expressions of modeler intent provides a summarizing abstraction of model content that helps manage complexity and aid comprehension. Here we apply this concept to macromodels rather than to domain models in order provide a summarizing abstraction of macromodel content. Decomposition criteria add missing (i.e., unexpressed) information about the modeler intent regarding the decomposition and this allows a consumer to understand the underlying rationale for the way the collection of views that form the decomposition is structured.

Indexed decomposers also provide a level of abstraction in another way because the elements of the index set could be seen as classifying the content of their corresponding views. In this sense, an indexed decomposer defines a *classification scheme* [ISOC] that structures the content of a model where the index set represents the principle of division that conveys the basis for the decomposition. When indexed decomposers are combined to form hierarchical decompositions they correspondingly form hierarchical classification schemes. The consistent use of a meaningful principle of division at each level of decomposition facilitates comprehension of this hierarchy. Classification schemes are widely used to organize information in order to aid comprehension, search and convey semantics. Thus, these same benefits also accrue for indexed model decompositions.

7.3.2 Impact on model quality

In Chapter 5 we identified six types of defects that content criteria could be used to identify. The notion of decomposition criteria builds on that of content criteria and adds new constraints that must hold between the content of models and the structure of the macromodel. In the case of indexed decomposition criteria, the indexing constraint says that there must be a view generated by each index value that produces a non-empty submodel and all the views must be generated by elements from the index set. Table 7.1 lists three possible defects that are detectable due to this constraint.

The first two are similar to inclusion/exclusion defects that occur with content criteria but here we consider the inclusion/exclusion of views in the macromodel. View exclusion violations are interesting since they indicate that new views with specific content criteria must be created. Thus, they could be seen as "proposing" diagrams that are implied by the modeler's intent. This is particularly relevant during model evolution and is discussed in Section 7.3.3 below. The final type of defect occurs when an index value produces an empty submodel and so a view is not required. This may signal an unusual situation because if the modeler intended the decomposition of the model into views according to an index set, then a typical expectation is that there is some content in the view for each index value (i.e., otherwise, why not just exclude the "irrelevant" values from the index set?). Nevertheless, there may be cases when it is reasonable for this situation to occur and so we refer to this only as "potential" incompleteness. We will see some of these cases in the example applications in Chapter 8.

Relationship to decomposition quality

Wand and Weber's work on system decomposition [WW89] suggests that of the many possible system decompositions, not all are of equal quality and that measures such as coupling and cohesion can be used to assess decomposition quality. Moody [M00] applies this approach to conceptual model decompositions and gives a set of principles that characterize a good decomposition. These include: *integration* – that the entities of a constituent model are all related to one another; *unity* – that the content of each constituent model should be related to a single central entity; *cognitive manageable* – that each constituent model should contain at most nine entities; etc.

Decomposition criteria can be seen to help improve decomposition quality in two ways. First, the complexity of the decomposition criteria can be used as an indicator of decomposition quality since the more complex this is, the more difficult it is for a consumer to comprehend a decomposition. Second, predefined generic decomposers could be developed that are guaranteed to meet certain quality levels by adhering to Moody's (or other) principles of good decomposition. For example, the *ByBaseClasses* decomposer guarantees the satisfaction of the integration principle (all related through the base class) and the unity principle (base class is the central entity) but not the cognitive manageability principle (can have more than nine entities related to the base class). These "well-behaved" decomposers can then be mandated as standards to be used during modeling activities.

Table 7.1. Defect types detectable due to indexed decomposition criteria.

Defect Type	Occurrence criteria
View inclusion violation	Views exist in the macromodel that do not correspond to an index value or are not of the type specified in the decomposer. $\exists v{:}MX.R \cdot \neg\exists x{:}\underline{M}_R{:}T_1 \cdot CC_v := F(M, x)$
View exclusion violation	An element exists in $\underline{M_R}$ that generates a non-empty submodel but no corresponding view exists in *MX.R*. $\exists x{:}\underline{M}_R{:}T_1 \cdot F(\underline{M}, x) \neq \emptyset \wedge \neg\exists v{:}MX.R \cdot CC_v := F(M, x)$.
Potential incompleteness	Some views are empty. $\exists x{:}\underline{M}_R{:}T_1 \cdot F(\underline{M}, x) = \emptyset$

7.3.3 Support for automation and model evolution

Figure 7.5 shows the basic modes for using macromodels discussed in Chapter 6 applied specifically to decomposition. In conformance checking mode, there is a known decomposition and the question is whether it conforms to given decomposition criteria in order to assess the correctness of the decomposition.

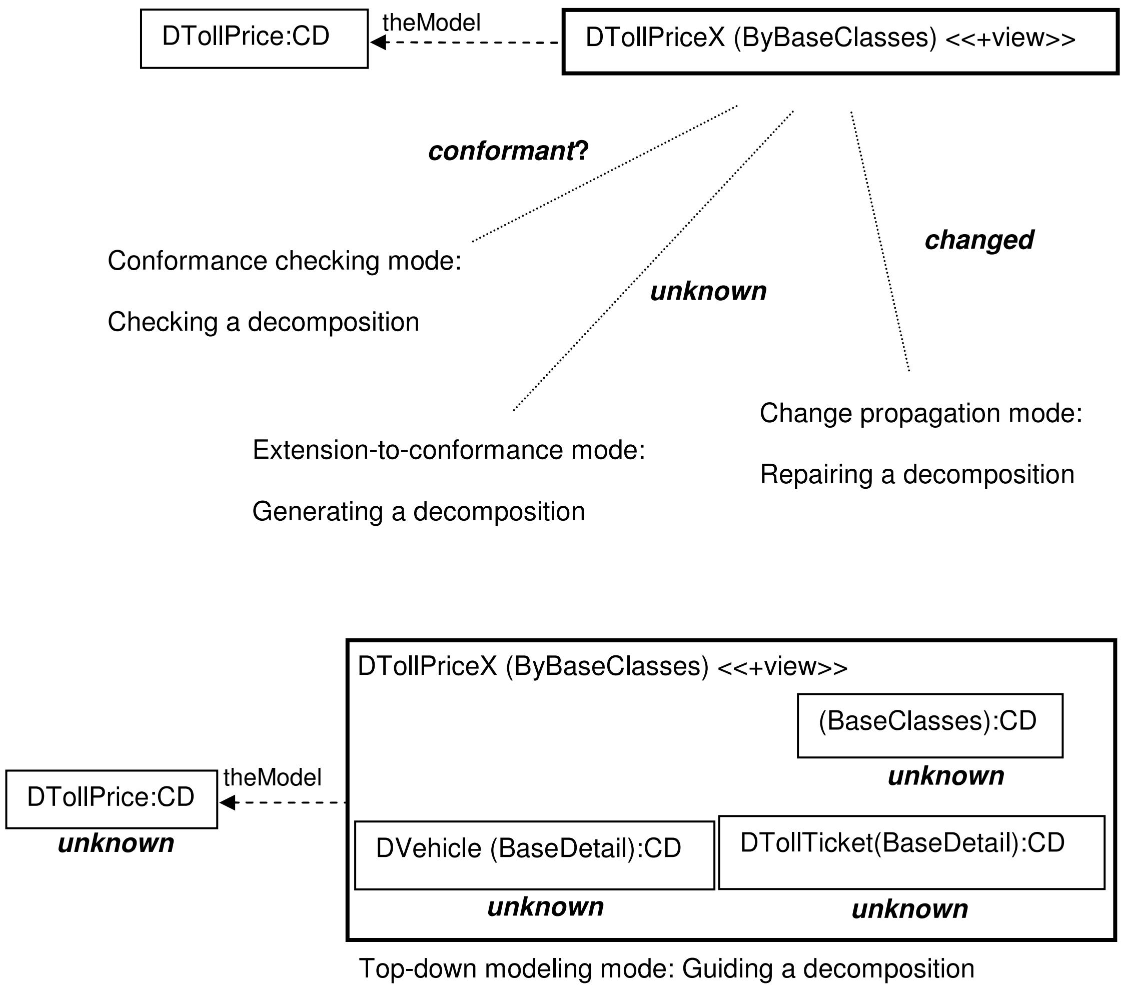

DTollPrice:CD
theModel
DTollPriceX (ByBaseClasses) <<+view>>
conformant?
Conformance checking mode:
Checking a decomposition
unknown
changed
Extension-to-conformance mode:
Generating a decomposition
Change propagation mode:
Repairing a decomposition
DTollPriceX (ByBaseClasses) <<+view>>
(BaseClasses):CD
unknown
DTollPrice:CD
theModel
unknown
DVehicle (BaseDetail):CD
unknown
DTollTicket(BaseDetail):CD
unknown
Top-down modeling mode: Guiding a decomposition

Figure 7.5. Three main modes for using decomposition criteria.

In the extension-to-conformance mode, a particular model is given and a decomposition of it is synthesized according to certain decomposition criteria. Predefined decomposers can be used to produce decompositions with good quality properties. In change propagation mode, a change can cause view inclusion/exclusion violations and repairs to resolve them require views to be removed/added. For example, in Figure 7.4, if a new class *Road* is added to the submodel realizing *BaseClasses* then a new view *Road*:*CD* must appear in macromodel *DTollPriceX.BaseDetail* and a corresponding submodel created to contain this class and its details. Finally, in top-down modeling mode, before a model is created, a particular decompositional structure is imposed on it using decomposition criteria and this directs subsequent model development to refine this structure. This may occur due to method-level intent when a development process mandates the existence of submodels playing certain roles. It occurs due to project-level intent if the modeler (as definer) designs the decompositional structure. For example, a UML model may be decomposed on the basis of package, major component, use case, etc. and the modeler may decide one of these bases at the outset in order to manage the complexity of the model creation process.

During model evolution all four modes may apply. As a submodel becomes too large to handle easily, extension-to-conformance can be used to synthesize a decomposition of it that is more manageable. Top-down modeling may apply when a major new section of the model is identified and the content must be pre-structured. As content and submodels are being added and removed, conformance checking and change propagation can be used to ensure that it is still consistent with the decomposition criteria.

The identification of the decomposition criteria exposes the different ways that model evolution can affect diagram structure. One dimension of change is horizontal: a model change may cause index values to be added/removed and this adds/removes corresponding views. The addition of the class *Road to BaseClasses,* as discussed above, is an example of this. Another dimension of change is vertical: any view can be decomposed further (or a decomposition removed) if more content is added (removed). For example, if *DTollPrice* evolves so that *DTollTicket* becomes too complex, it could be replaced by a decomposition into smaller views in the way that *DVehicles* is. Finally, another form of change is to reorder the indices. For example, in 7.4, we could alternatively use decomposition by *UserType* at the top level in *DTollPriceX* and then use *ByBaseClasses* within either or both of these views. This does not affect the content of diagrams but re-groups them.

7.4 Summary

Chapter 5 introduced the notion that the modeler intent about a diagram or submodel of a model can be characterized as a view of the model defined using content criteria. In this chapter, we extended this concept to collections of diagrams/submodels that decompose a model and called this the decomposition criteria of the decomposition. In the same way that an extractor is used to express content criteria, a decomposer is used to express decomposition criteria by defining how to generate a collection of views that decompose a larger view or model. In particular, we have focused on indexed decomposers because they support comprehension by defining a classification scheme over the decomposed model and they can be expressed graphically within a macromodel. Furthermore, our examples in Chapter 8 suggest that these are a commonly occurring form of decomposition criterion.

Although decomposition criteria are not typically expressed explicitly for collections of models, we propose that doing so can bring various benefits to modeling practice. Model comprehension by stakeholders can improve because indexed decomposition criteria reveal the basis for a decomposition and thus provide a meaningful level of abstraction over a macromodel. Violations of the constraints expressed in decomposition criteria reveal structural defects in the decomposition and hence help to improve the quality of the model and decomposition. Finally, automation and model evolution is supported because decomposition criteria are expressed formally and can be used to guide and synthesize the structure of a decomposition as a model changes.

Chapter 8

Evaluation

In this section, we evaluate the approaches defined in this thesis for expressing modeler intent by reverse-engineering the role level for two detailed example modeling projects and assessing the value this brings. In Chapter 1, we discussed four ways that value can be obtained from expressing modeler intent: improving model quality, improving model comprehension, supporting automation and supporting model evolution. The improvement of model quality is due to the ability to identify various types of defects exposed by expressing modeler intent. Thus, we enumerate the occurrences of these defects for both examples and discuss how to improve their quality by correcting them.

We evaluate the ability for modeler intent to improve model comprehension in the examples by enumerating the quantity and variety of information about modeler intent that we make explicit through the reverse engineering process. As discussed at various points in this thesis, modeler intent can be shown to aid model comprehension. We briefly review the basis for this claim here from the perspective of linguistics and from the perspective of complexity management. A dominant view from linguistics (pragmatics), regarding the comprehension of texts is that this depends on the reader's ability to determine the author's intent [G01]. Rhetorical Structure Theory [RST] extends this further to include the author's intended relationships between texts. Correspondingly, since a model is kind of text, we expect that a consumer's comprehension of a model depends on their knowledge of the modeler's intent and when there are multiple related models, this includes the intended relationships between the models.

Modeler intent expressed in a macromodel also provides complexity management mechanisms such as abstraction and hierarchical decomposition that aid the comprehension of the information in a collection of models. Abstraction is a powerful mechanism for managing complexity by supporting top-down understanding but in order for a level of abstraction to be useful it must provide some form of summarization of the details that are omitted [M09]. As we discussed in Chapter 2, modeler intent is a type of summary of model content that is particularly well suited to supporting comprehension because it explains why a model contains the information it contains. Thus, one way a macromodel supports comprehension is by providing this level of abstraction on a collection of models. Another key way that a macromodel supports the comprehension of model collections is that it allows structural information about the collection to be expressed that are otherwise missing. In particular, we can identify three types of structural information that a macromodel can express: the relationships between models, the hierarchically structure of model collections and the sets that index model collections. The first two of these types of structure have been shown to support the improved cognitive manageability of information, particularly for decompositions of large conceptual models into collections of views[M06]. The third kind of information represents a classification scheme [ISOC] over models and is similar to the kind of abstraction that is widely used to organize information in order to aid comprehension, search and navigation.

The ability to support automation arises as a result of the fact that the modeler intent is expressed formally and we provide a formal expression of this for both examples. We take this as a partial justification of this claim and we defer to Chapter 9 for an actual demonstration of the feasibility of automation. Finally, since our examples are static sets of models rather than a project in progress, we were not able to directly assess the use of the formalism for supporting model evolution. However, we discuss various plausible evolution scenarios to suggest how this could be applied.

For each example, we present the results of the evaluation using the following structure:

- The macromodel for the example is presented that formally expresses the modeler's intent. This includes the model relationships, content criteria and decomposition criteria. Note that since we did not have access to the original modelers we have inferred the modeler intent from an analysis of the model and the supporting documentation.
- In addition to analyzing the kinds of content criteria that are used for the example, each of the following types of added structural information is examined: hierarchical model collections, relationships and model collection indexing. The occurrences of these types of information is summarized in a table that is constructed according to Table 8.1. In addition to the summary, the particular

interesting aspects of the structural information in the example are discussed in detail.

- Each of the nine types of defects that have been identified (in chapters 5 and 7) as being detectable due to the expression of content and decomposition criteria are examined. The occurrences of these are presented in summary and detailed form. Table 8.2 lists the defect types and how their occurrences are counted.
- Model evolution scenarios are discussed.

Table 8.1. Types of structural information and how occurrences are counted.

Type of structural information	Counting of occurrences
Hierarchical model collections	Number of diagram groups : The main macromodel for each model is not counted. Each contained macromodel found within a main macromodel at any containment depth is counted as one group. Number of hierarchies (depth) : Each macromodel containment tree is counted as one hierarchy. The depth of the hierarchy is the level of the deepest contained macromodel in the hierarchy.
Relationships	Number of relationships : Each occurrence of a relationship is counted as one. Content criteria are not counted as relationships.
Index sets	Index set (uses): Each index set is counted. Each case in which a macromodel is indexed using this type is counted as a use of the index set.

Table 8.2. Types of defects and how occurrences are counted.

Defect type	Description	How counted
Naming inconsistency	The same intent is expressed in multiple ways.	If the intent is expressed in a single way then there is no inconsistency. If the intent is expressed in *k* ways then we count this as *k*-1 cases.
Naming inaccuracy	The name does not express the same concept as the content criteria.	Each diagram is a case.
Content inclusion	The diagram includes information that it should not according to its content criteria.	Each diagram is a case even if multiple incorrect inclusions occur.
Content exclusion	The diagram excludes information that it should not according to its content criteria.	Each diagram is a case even if multiple incorrect exclusions occur.
Weakly modeled info	The inclusion conditions are weakly modeled in M.	Each diagram is a case.
Unmodeled info	The inclusion conditions are not modeled in M.	Each diagram is a case.
Potential incompleteness	Content in M may be missing according to the decomposition criteria. The diagram's content is empty based on the content criteria.	Each diagram is a case.
Diagram inclusion	A diagram is included in a macromodel that does not satisfy the decomposition criteria.	Each diagram is a case.
Diagram exclusion	A diagram is missing from a macromodel that should be present according to the decomposition criteria.	Each diagram is a case.

8.1 Application to PUMR Example

The Private User Mobility dynamic Registration service (PUMR) is given as an example protocol developed and modeled using UML according to the standard set out by the European Telecommunications Standards Institute (ETSI) for protocol specifications [ET08]. The PUMR example consists of three UML models: a context model (4 diagrams), a requirements model (6 diagrams) and a specification model (32 diagrams) and defines a protocol for integrating telecommunications networks in order to support mobile communications. Thus, for example, PUMR allows an employee using a mobile phone at her home company with a private exchange to roam to other private exchanges seamlessly. 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." [ET08, pg. 43]

8.1.1 Expressing modeler intentions

In this section we discuss the results of our analysis to express the modeler's intentions for the 42 diagrams over the three UML models in the PUMR example. Since we did not have access to the original definers of these diagrams, we relied on the documentation in [ET08] of the diagrams to infer their intentions. Fortunately, the documentation is substantial and detailed so that we have a high level of confidence that our results are reasonable. In the next section we present the macromodel for PUMR. This presentation includes the relationships, decomposition criteria and some of the content criteria for the individual models. The full results of the content criteria analysis is given in Appendix B and is omitted from the presentation when it leads to excessive complexity.

Macromodel: type diagrams

Figures 8.1 and 8.2 are type diagrams show the model and relationship types used in the PUMR example. The model and relationship types used are defined in detail in Appendix B. The relationship types are described informally in Table 8.3. Note that the first three types are constructed types that concretize abstract types.

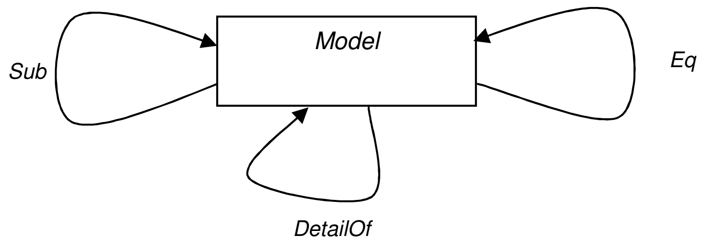


Figure 8.1. Abstract types.

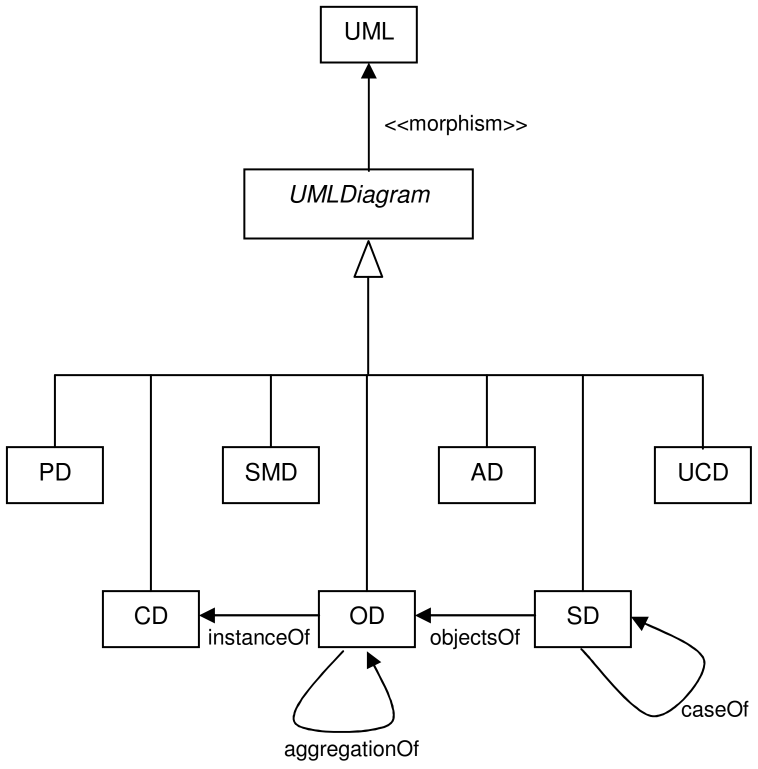


Figure 8.2. Concrete types.

Table 8.3. Summary of relationship types used in PUMR.

Relationship type	Description
detailOf(*T1*, *T2*)	detailOf(M1, M2) indicates that M1 expands on or details an element of M2. This relationship holds iff the following conditions are true: ○ M1 and M2 are submodels of the same model. ○ All the generators of M1 are elements of M2. detailOf() is a constructed relationship type and does not contain a mapping.
submodelOf(*T1*, *T2*)	sub(M1, M2) indicates that M1 is a submodel of M2. sub() is a constructed relationship type and does not contain a mapping.
eq(*T*, *T*)	eq(M1, M2) indicates that the content of M1 and M2 is identical but are distinct copies of this content. Eq() is a constructed relationship type and contains a mapping.
instanceOf(OD, CD)	instanceOf(M1, M2) indicates that M2 contains the type information for the elements of M1. This relationship holds iff every object and link in M1 is an instance of a class and association, respectively, in M2. instanceOf() is a defined relationship type and does not contain a mapping.
objectsOf(SD, OD)	objectsOf(M1, M2) indicates that M2 contains the minimal set of objects and links required for the behaviour described by M1. This relationship holds iff the following conditions are true: ○ M1 and M2 have the same set of objects. If a message is passed from object *X* to object *Y* in M1 then there is a corresponding link in M2 between *X* and *Y* that the messages is sent over. objectsOf() is a defined transformation and contains a mapping.
aggregationOf(OD, OD)	refines(M1, M2) indicates that the elements of M1 are aggregates of elements in M2. This relationship holds iff ○ Every object in B is expressed as an aggregation of a set of objects in A and every object in A is in exactly one such aggregation. ○ Every link in B is expressed as an aggregation of links in A and every link in A is in exactly one such aggregation. ○ The link aggregations must be consistent with the object aggregations of their endpoints.
caseOf(SD, SD)	caseOf(A, B) holds iff the following conditions are true: ○ 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. ○ 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. ○ The message refinements must be consistent with the object refinements of their endpoints.

Macromodel: macromodel diagrams

The macromodel for the PUMR example is presented in Figures 8.3 to 8.10. Figure 8.3 is the overview showing the three UML models and how they are related. Figure 8.4 shows the presentation of the *ContextModel* while Figure 8.5 shows this for the *ReqModel*. Figure 8.6 shows the overview of the *SpecModel* presentation and the subsequent figures show the content of particular contained macromodels within this.

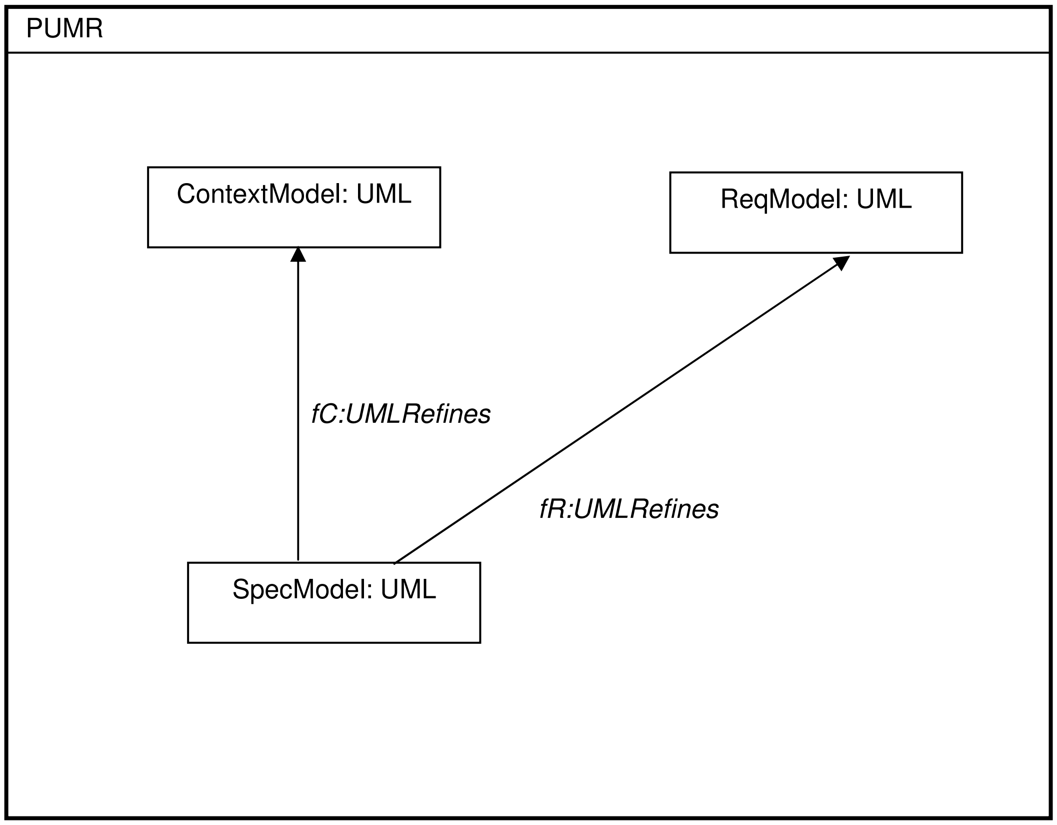


Figure 8.3. The main macromodel for PUMR.

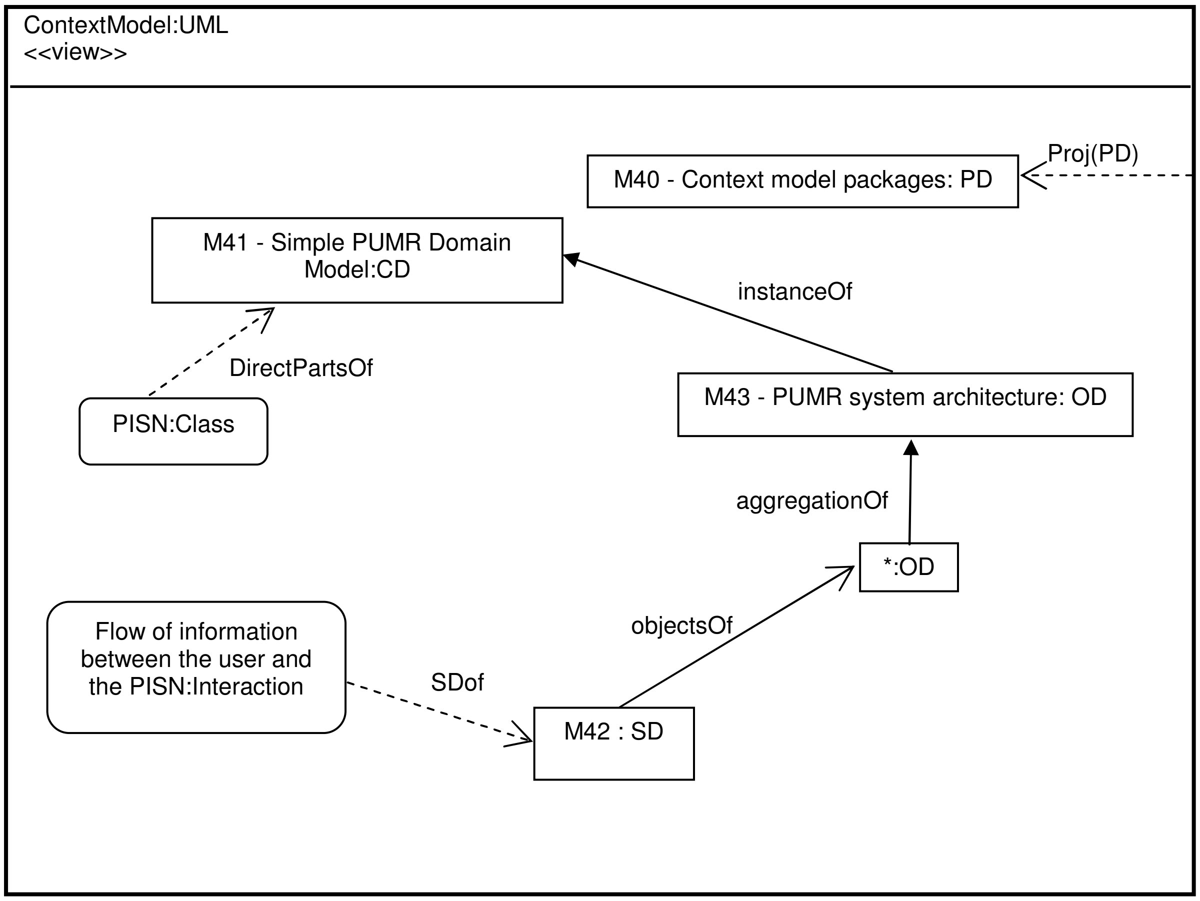

ContextModel:UML
<<view>>
M40 - Context model packages: PD
Proj(PD)
M41 - Simple PUMR Domain Model:CD
instanceOf
DirectPartsOf
PISN:Class
M43 - PUMR system architecture: OD
aggregationOf
*:OD
objectsOf
Flow of information between the user and the PISN:Interaction
SDof
M42 : SD

Figure 8.4. Presentation of the Context Model.

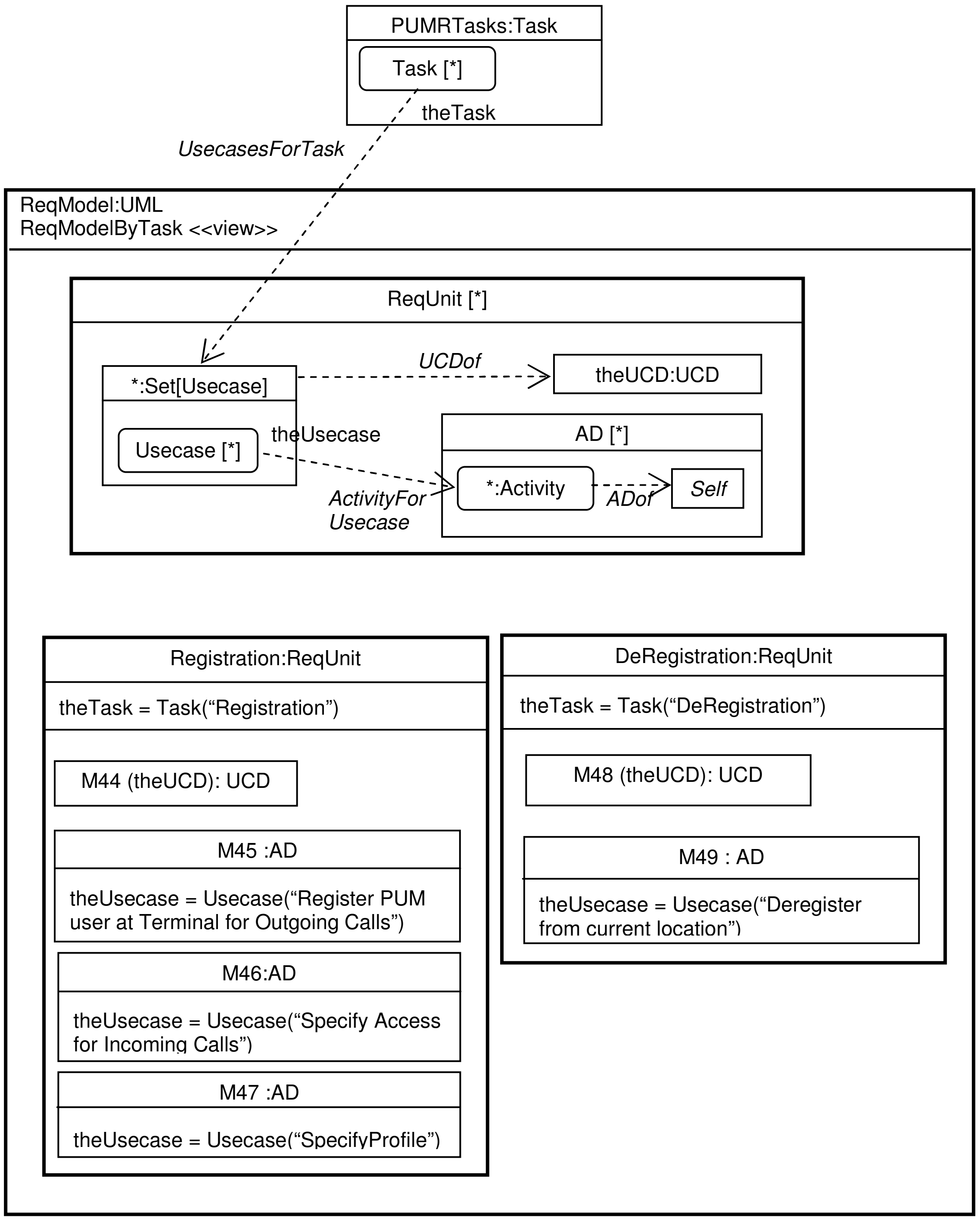


Figure 8.5. Presentation of the ReqModel.

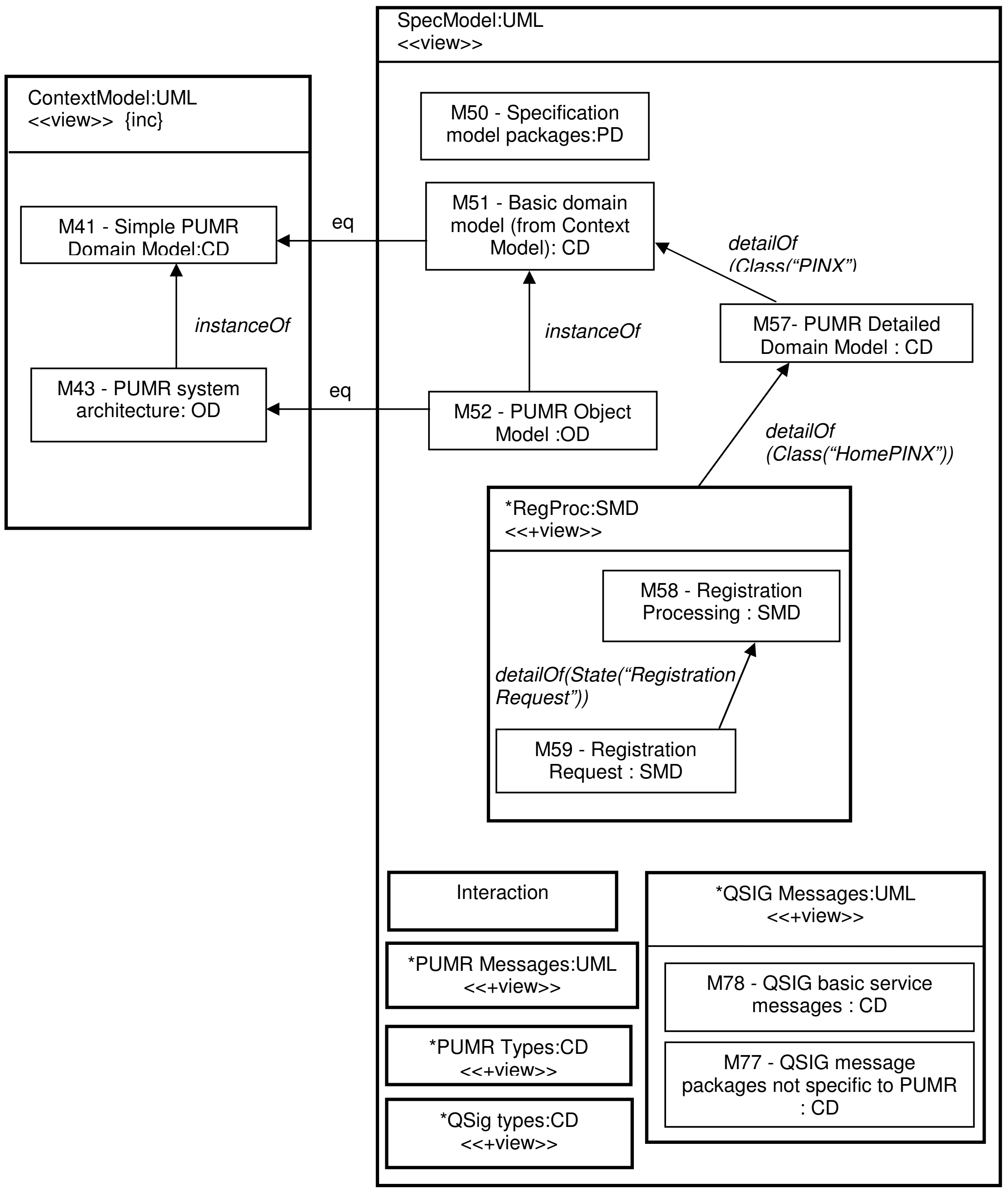

SpecModel:UML
<<view>>
ContextModel:UML
<<view>> {inc}
M50 - Specification model packages:PD
M51 - Basic domain model (from Context Model): CD
M41 - Simple PUMR Domain Model:CD
eq
detailOf (Class("PINX")
instanceOf
instanceOf
M57- PUMR Detailed Domain Model : CD
M43 - PUMR system architecture: OD
eq
M52 - PUMR Object Model :OD
detailOf (Class("HomePINX"))
*RegProc:SMD
<<+view>>
M58 - Registration Processing : SMD
detailOf(State("Registration Request"))
M59 - Registration Request : SMD
Interaction
*QSIG Messages:UML
<<+view>>
*PUMR Messages:UML
<<+view>>
M78 - QSIG basic service messages : CD
*PUMR Types:CD
<<+view>>
M77 - QSIG message packages not specific to PUMR : CD
*QSig types:CD
<<+view>>

Figure 8.6. Top level of SpecModel presentation.

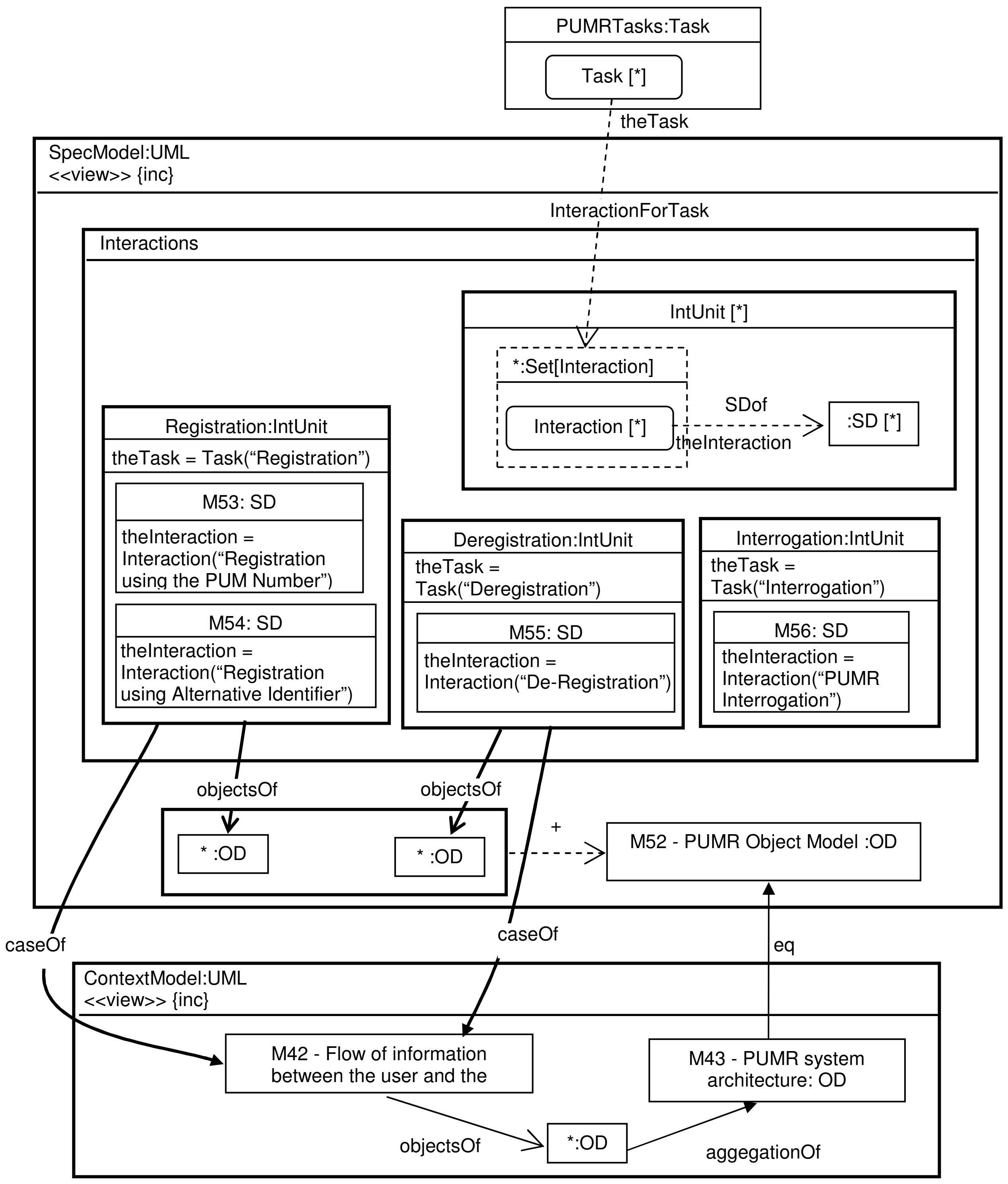

PUMRTasks:Task
Task [*]
theTask
SpecModel:UML
<<view>> {inc}
InteractionForTask
Interactions
IntUnit [*]
*:Set[Interaction]
Interaction [*]
SDof
theInteraction
:SD [*]
Registration:IntUnit
theTask = Task("Registration")
M53: SD
theInteraction = Interaction("Registration using the PUM Number")
M54: SD
theInteraction = Interaction("Registration using Alternative Identifier")
Deregistration:IntUnit
theTask = Task("Deregistration")
M55: SD
theInteraction = Interaction("De-Registration")
Interrogation:IntUnit
theTask = Task("Interrogation")
M56: SD
theInteraction = Interaction("PUMR Interrogation")
objectsOf
objectsOf
* :OD
* :OD
+
M52 - PUMR Object Model :OD
caseOf
caseOf
eq
ContextModel:UML
<<view>> {inc}
M42 - Flow of information between the user and the
M43 - PUMR system architecture: OD
objectsOf
*:OD
aggegationOf

Figure 8.7. SpecModel - Interactions.

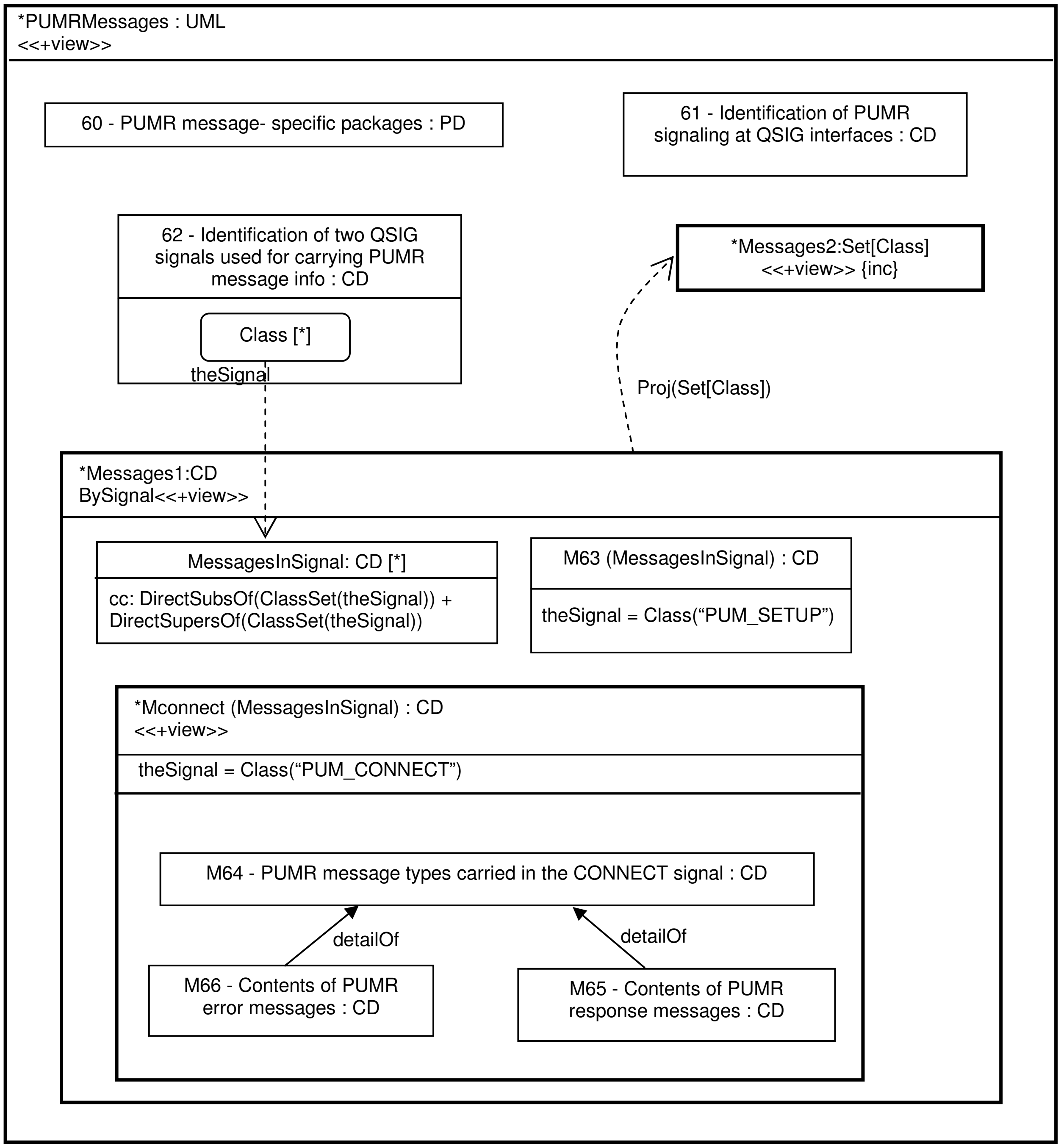

*PUMRMessages : UML
<<+view>>
60 - PUMR message- specific packages : PD
61 - Identification of PUMR signaling at QSIG interfaces : CD
62 - Identification of two QSIG signals used for carrying PUMR message info : CD
Class [*]
theSignal
*Messages2:Set[Class] <<+view>> {inc}
Proj(Set[Class])
*Messages1:CD BySignal<<+view>>
MessagesInSignal: CD [*]
cc: DirectSubsOf(ClassSet(theSignal)) + DirectSupersOf(ClassSet(theSignal))
M63 (MessagesInSignal) : CD
theSignal = Class("PUM_SETUP")
*Mconnect (MessagesInSignal) : CD <<+view>>
theSignal = Class("PUM_CONNECT")
M64 - PUMR message types carried in the CONNECT signal : CD
detailOf
detailOf
M66 - Contents of PUMR error messages : CD
M65 - Contents of PUMR response messages : CD

Figure 8.8. SpecModel – decomposition of messages on the basis of signal.

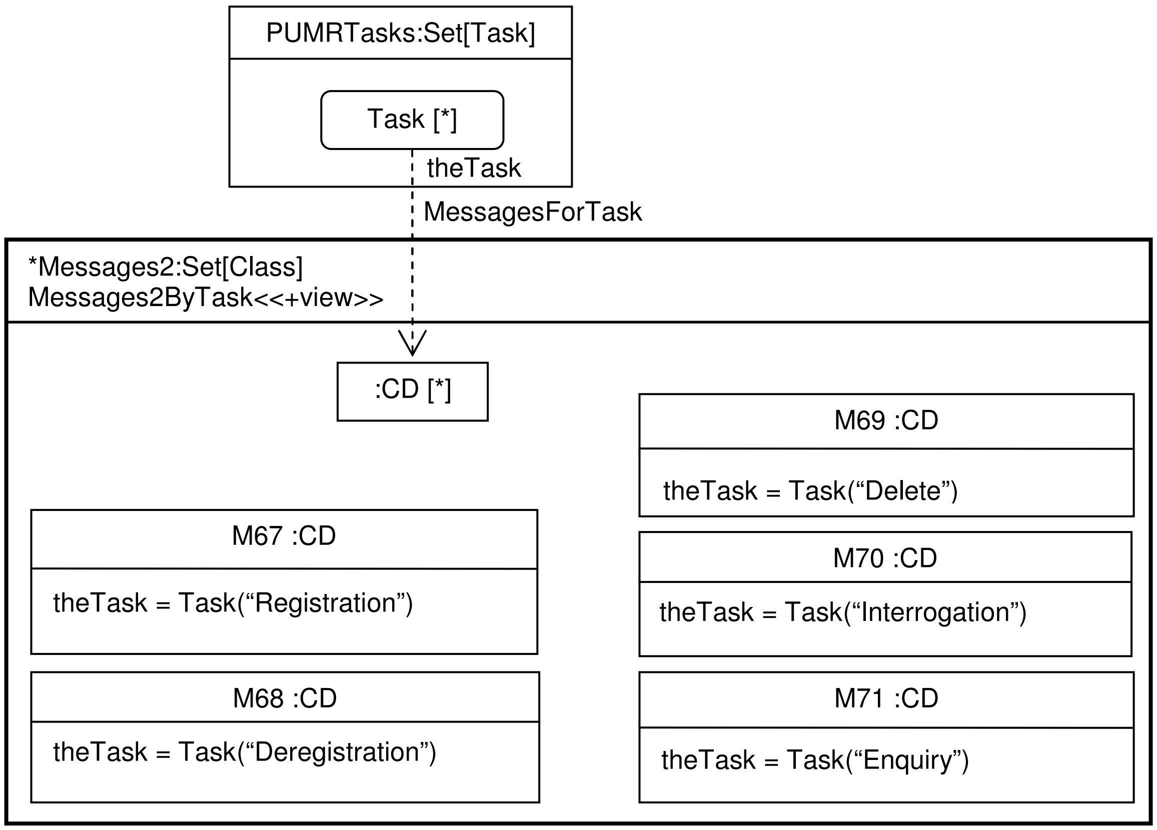


Figure 8.9. SpecModel – decomposition of messages on the basis of task.

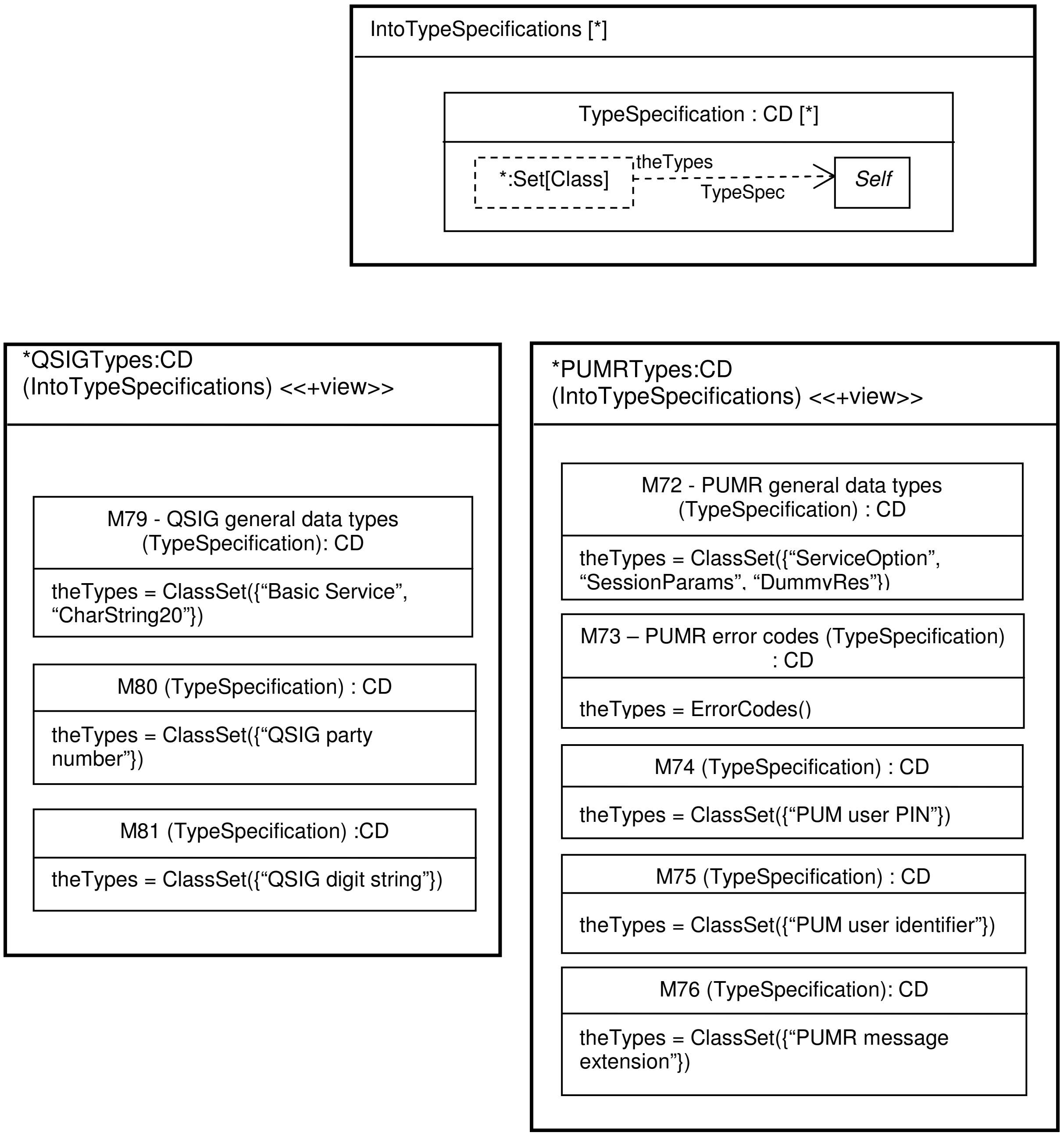

IntoTypeSpecifications [*]
TypeSpecification : CD [*]
*:Set[Class]
theTypes
TypeSpec
Self
*QSIGTypes:CD (IntoTypeSpecifications) <<+view>>
M79 - QSIG general data types (TypeSpecification): CD
theTypes = ClassSet({“Basic Service”, “CharString20”})
M80 (TypeSpecification) : CD
theTypes = ClassSet({“QSIG party number”})
M81 (TypeSpecification) :CD
theTypes = ClassSet({“QSIG digit string”})
*PUMRTypes:CD (IntoTypeSpecifications) <<+view>>
M72 - PUMR general data types (TypeSpecification) : CD
theTypes = ClassSet({“ServiceOption”, “SessionParams”, “DummyRes”})
M73 – PUMR error codes (TypeSpecification) : CD
theTypes = ErrorCodes()
M74 (TypeSpecification) : CD
theTypes = ClassSet({“PUM user PIN”})
M75 (TypeSpecification) : CD
theTypes = ClassSet({“PUM user identifier”})
M76 (TypeSpecification): CD
theTypes = ClassSet({“PUMR message extension”})

Figure 8.10. SpecModel – type specifications.

Content criteria

To give a sense for the diversity of content criteria of the PUMR diagrams, we summarize a few in Table 8.4. We could not infer the content criteria for three of the diagrams because the documentation was too vague. Of the remaining 39 diagrams, only seven required a detailed expression using inclusion conditions such as is used for diagram 62 while the remainder could be expressed compactly as expressions using a small set of predefined extractors.

The predefined extractors are listed in Table 8.5 and their definitions can be found in Appendix B. They range from generic to domain-specific. The first group of three are the most generic and represent extractor constructors. The next group of eleven are particular extractors that apply to any UML model. The final group of four extractors are specific to the PUMR project. Within the generic UML group there are natural extractors corresponding to "large objects" within a UML model that have dedicated diagram types - e.g., activities, interactions and statemachines. The most common content criteria is to show the full content of these objects in a diagram (e.g., *ADof*(*M*:*UML*, *K*:*One*[*Activity*]). The variety of content criteria that can be associated with these depend on their ability to show partial information. For example, statemachine diagrams can also be used to show the content of a single composite state and so we have *SMDofState*(*M*:*UML*, *K*:*One*[*State*]). This capability can be used to decompose the presentation of a large statemachine across several diagrams. This is the case with diagrams 58 and 59 – together they depict the statemachine showing registration processing. A similar possibility exists with large interactions and activities but no instances of these occur in the PUMR example.

Table 8.4. Examples of content criteria from PUMR analysis.

Diagram	Content Criteria
40 - Context model packages : PD	CC_{M40} := Proj(ContextModel, PD)
44 - PUM Registration use case diagram: UCD	CC_{M44} := UCDof(ReqModel, UsecasesForTask(ReqModel, “Registration”), UCD)
56 - Example sequence diagram showing PUMR interrogation: SD	CC_{M55} := SDof(SpecModel, Interaction(SpecModel, “De-Registration”))
62 - Identification of the two QSIG signals used for carrying PUMR message info : CD	CC_{M62} := [*precondition* := ∃*qsig*, *pumr*:SpecModel.Package · SpecModel.name(*qsig*) = “QSIG” ∧ SpecModel.name(*pumr*) = “PUMR” Q_{Class}(*c*) := ∃*c1*: SpecModel.Class, *p, p1*: SpecModel.Package · TC(SpecModel.ownedMember(*p*, *qsig*)) ∧ SpecModel.ownedMember(*c*, *p*) ∧ TC(SpecModel.ownedMember(p1, *pumr*)) ∧ SpecModel.ownedMember(*c1*, *p1*) ∧ TC(SpecModel.general(*c1*, *c*)) ∧ SpecModel.name(stereotype(*c*)) = “communication message”)]
51 - Basic Domain Model (from Context Model): CD	CC_{M51} := DirectPartsOf(SpecModel, ClassSet(SpecModel, {“PISN”}))
58 - Statechart diagram showing the registration processing at the Home PINX: SMD	CC_{M58} := RegProc – M59
RegProc : SMD	$CC_{RegProc}$:= SMDof(SpecModel, StateMachineForClass(SpecModel, Class(SpecModel, “Home PINX”)))
59 - Statechart sub-diagram showing the detailed processing of a registration request at the Home PINX: SMD	CC_{M59} := SMDofState(RegProc, State(SpecModel, “RegistrationRequest”))

Table 8.5. Predefined extractors used to express content criteria in the PUMR example.

Predefined Extractor	Result Type	Summary	Number of uses
Proj(M:*T*, *T1*)	*T1*	Extracts the maximal *T1* submodel of M. Assumes that $T1 \subseteq T$.	1
T(M:UML, S:One[String]	One[*T*]	Extracts the *T* element with name S	10
*T*Set(M:UML, S:Set[String])	Set[*T*]	Extracts the *T* set model with names in S	15
SDof(M:UML, K:One[Interaction])	SD	Extracts the SD that contains the full content of interaction K	4
WithStereotype(M:UML, M1:Set[string])	Set[Class]	Extracts the set of classes with stereotype in set M1	2
UCDof(M:UML, M1:Set[Usecase])	UCD	Extracts the UCD that shows all the relationships that the usecases in M1 participate in.	2
ADof(M:UML, K:One[Activity])	AD	Extracts the AD that contains the full content of activity K	4
SMDof(M:UML, K:One[Statemachine])	SMD	Extracts the SMD that contains the full content of statemachine K	1
SMDofState(M:UML, K:One[State])	SMD	Extracts the SMD that contains the full content of state K	1
DirectPartsOf(M:UML, K:Set[Class])	CD	Extracts the CD consisting of the classes K and all their direct aggregated classes	2
DirectSubsOf(M:UML, K:Set[Class])	CD	Extracts the CD consisting of the classes K and all their direct subclasses	4
DirectSupersOf(M:UML, K:Set[Class])	CD	Extracts the CD consisting of the classes K and all their direct superclasses	1
ActivityForUsecase(M:UML, U:One[Usecase])	One[Activity]	Extracts the activity owned by the use case U	4
StateMachineForClass(M:UML, K:One[Class])	One[Statemachine]	Extracts the statemachine owned by class K	1
TypeSpec(M:UML, M1:Set[Class])	CD	Extracts the subclasses and aggregated classes of the classes in M1	8
MessagesForTask(M:UML, T:string)	CD	Extracts the set of message classes that relate to task T	5
InteractionsForTask(M:UML, T:string)	CD	Extracts the set of interactions that relate to task T	3
UsecasesForTask(M:UML, T:string)	Set[Usecase]	Extracts the set of use cases that relate to task T	2

8.1.2 Findings

Added structural information

Table 8.6 summarizes the quantity of structural information made explicit of the kinds identified in Table 8.1. The main contained macromodel for each UML model (i.e., three macromodels for: Context, Req, Spec) is are not counted. We make the following observations regarding the structural information.

Rationale for diagram collection identification

Although the PUMR example in [ET08] does not identify diagram collections beyond the main grouping based on the *ContextModel*, *ReqModel* and *SpecModel*, the PUMR macromodel has many additional contained macromodels. In this section, we discuss the rationale for identifying these.

Table 8.6. Summary of the structural information made explicit in the PUMR example

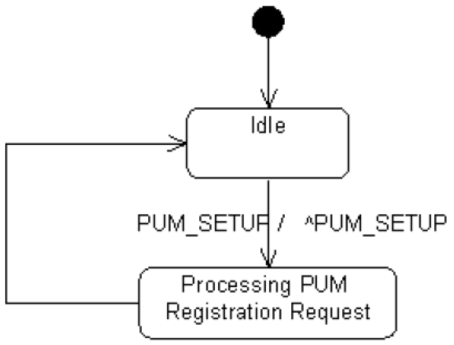
Structural information measure	Value
Number of diagram groups	15
Number of hierarchies (depth)	5 (1), 1(2), 1(3)
Number of relationships	16
Index sets (uses)	Task (3), Signal (1), Usecase (2)

Eight of the fifteen identified collections are included to indicate the implicit decomposition of a required, but unrealized, submodel. For example, consider the diagrams *M58* and *M59* that are reproduced in Figure 8.11. It is clear that these two actually represent the decomposition of the full state machine diagram for registration processing in the class *HomePINX* but this state machine diagram doesn't actually occur explicitly in the PUMR example. Thus. we presume that the actual modeler intent is for *M58* and *M59* to implicitly decompose an unrealized state machine diagram *RegProc*:*SMD*[27]. This arrangement is shown in Figure 8.6 along with the fact that *M59* shows a detail of *M58*. It is important to note that the *M58* and *M59* are defined as views of *RegProc* rather than *SpecModel* and thus they are existentially dependent on *RegProc*. This is the case with all decompositions.

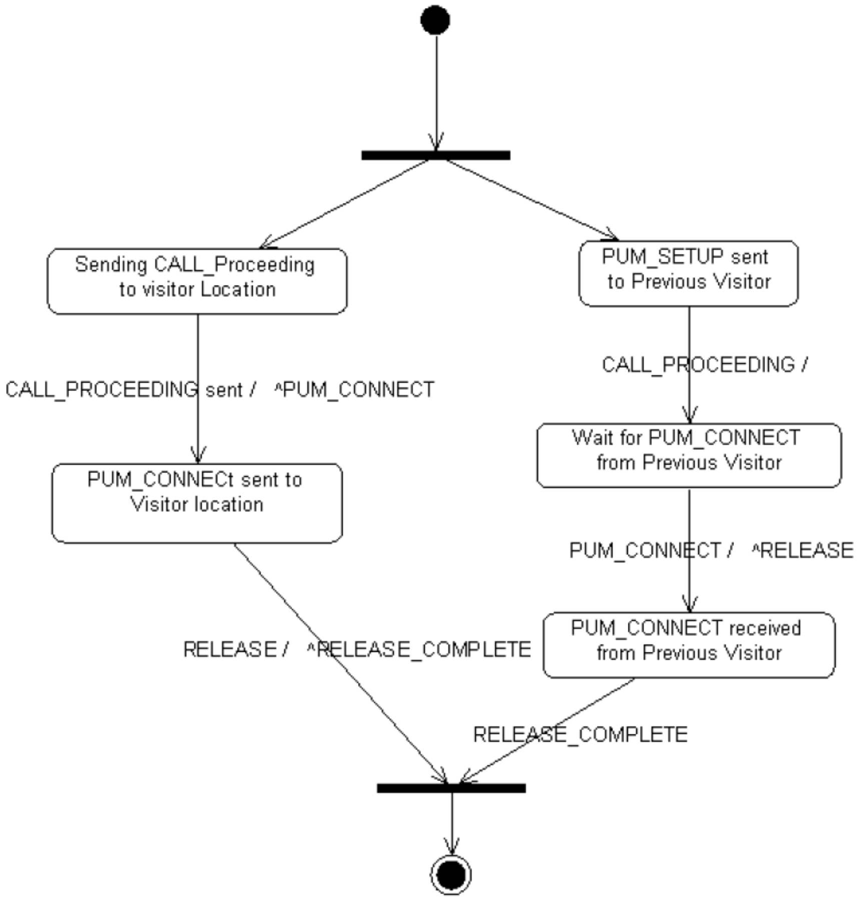
Five of the remaining identified collections arise as a result of the evident intended application of indexed presenter. For example, in Figure 8.5., two collections are identified with the same structure but corresponding to different tasks. Note that the diagram numbers correspond to the order in which they occur in the document [ET08]; thus, it should be clear that these were intended to be grouped together even though there was no explicit indicator of this. The interactions in Figure 8.7 is another example of this.

Of the remaining two collections, both found in Figure 8.7, *Interactions* was assumed to exist to represent the set of consecutive sequence diagrams indexed by task and the anonymous collection of object diagrams occurs in order to express M52 as their sum.

[27] Our choice of name is evocative but arbitrary.



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


Sending CALL_Proceeding to visitor Location
PUM_SETUP sent to Previous Visitor
CALL_PROCEEDING sent / ^PUM_CONNECT
CALL_PROCEEDING /
PUM_CONNECt sent to Visitor location
Wait for PUM_CONNECT from Previous Visitor
PUM_CONNECT / ^RELEASE
RELEASE / ^RELEASE_COMPLETE
PUM_CONNECT received from Previous Visitor
RELEASE_COMPLETE

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

Figure 8.11. Diagrams M58 and M59 that decompose an unrealized state machine diagram.

Index sets

There are three index sets used by decomposers in the PUMR macromodel:

- *Task* used three times (See Figures 8.5, 8.7 and 8.9)
- *Signal* (See Figure 8.8)
- *Usecase* used twice (See Figure 8.5)

In Section 7.2.3 we characterized decomposers as having local vs. global scope and being generic vs. domain-specific. Based on this classification we can observe that *Task* and *Signal* both represent domain-specific (i.e., PUMR-specific) indices but *Task* has a more global scope than *Signal* since it used in three places to decompose different parts of the *ReqModel* and *SpecModel*. This suggests that *Task* acts like a “dimension” that cuts across all of PUMR. In contrast *Usecase* is a generic basis for decomposing a set of activity diagrams but it has local scope.

Alternate Decomposition

While it is often the goal of a decomposition to minimize redundancy, sometimes this is foregone in order to achieve an informative rhetorical effect in the presentation of the model. PUMR has a good example of this. The macromodels *Messages1* in Figure 8.8 and *Messages2* in Figure 8.9 show alternate decompositions of the same set of communication message classes, presumably to emphasize different aspects of these. *Messages1*, containing diagrams *M63 – M66*, decomposes the message classes on the basis of the signal that carries the message while *Messages2*, containing class diagrams *M67 – M71*, decomposes the message classes on the basis of the task type that uses the message. This juxtaposition of decompositions would not have been evident if the decomposition criteria had not been clearly defined.

Defects detected

Figure 8.12 summarizes the defects found due to our analysis. The number of each type of defect is determined according to the criteria given in Table 8.1. The complete list of the defects found is given in Appendix B. Here we discuss some of the noteworthy cases.

Naming inconsistencies show up in with the pairs of diagrams "M43 - PUMR system architecture"/"M52 - PUMR Object Model" and "M41 - Simple PUMR Domain Model"/"M51- Basic domain model (from Context Model)". These are named differently even though they are intended to be equivalent as shown by the *eq* relationships in Figure 8.6. Another naming inconsistency occurs in the two groups of diagrams that show the

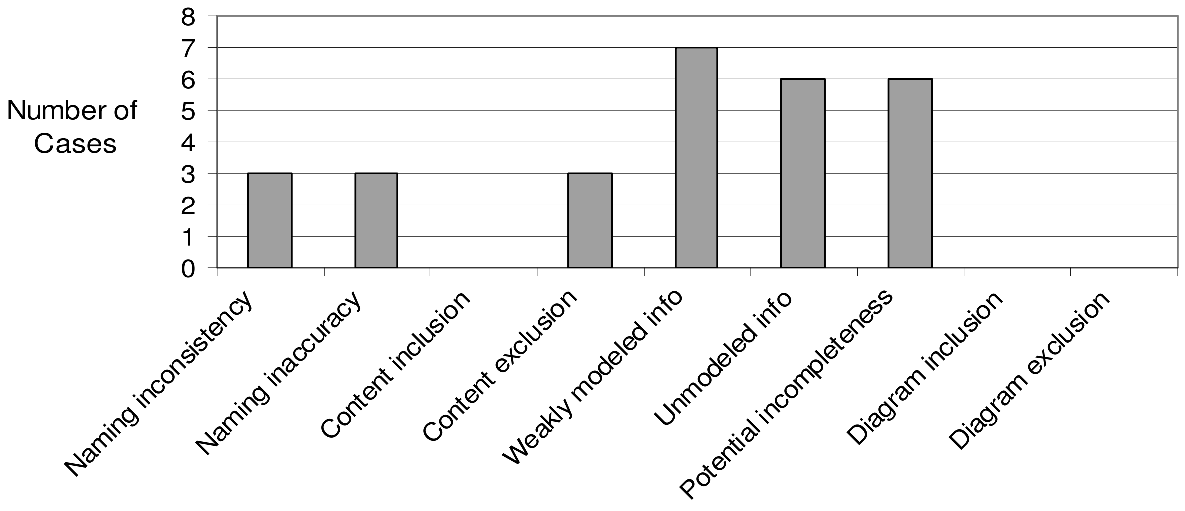


Figure 8.12. Defects found in PUMR example.

details for various data types in Figure 8.10. All of these have the same form for their content criteria – the application of the extractor *TypeSpec* to detail a set of data type (or enumeration) classes. When a single type *T* is detailed, the diagram is named "Type specification of *T*" – in all these cases we have omitted the name from the macromodel diagram since it is implied by the *theTypes* generator. However, when a diagram shows a group of types there is no fixed convention: e.g., "M73 – PUMR error codes","M72 - PUMR general data types", etc.

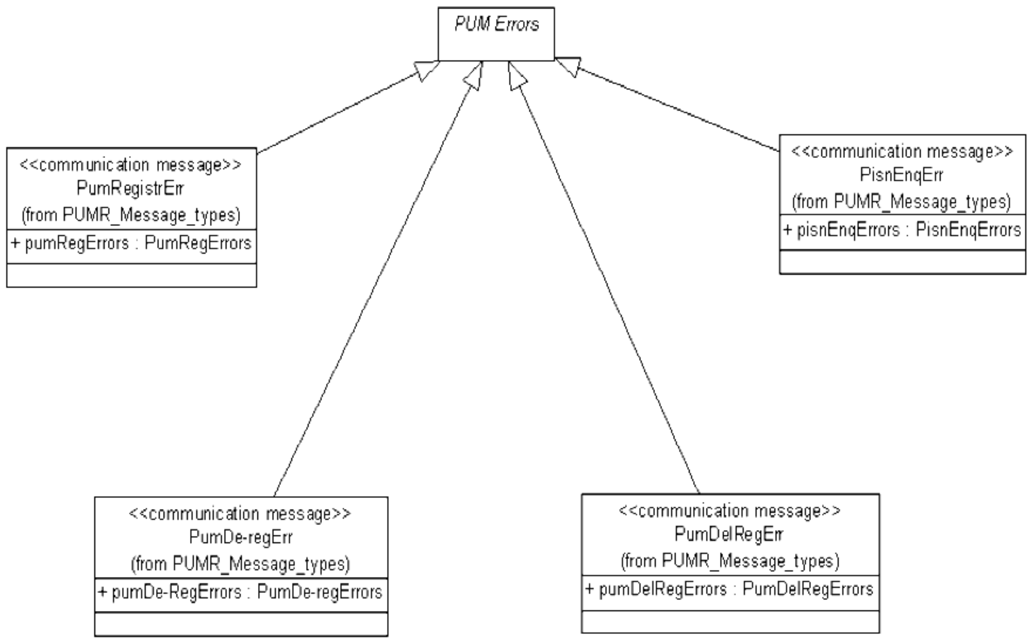
Naming inaccuracies are subtle defects that can cause confusion for a consumer. One example of this is diagram "M50 - Specification model packages." Like, "M40 - Context model packages", this suggests that the intent is to show all the packages in the model. With diagram *M40*, this is this case since its content criterion is defined by the extractor *Proj*(*ContextModel*, *PD*) that projects out all package diagram information from *ContextModel*. However with diagram *M50*, only a subset of packages of *SpecModel* are shown and it is not clear what the inclusion criterion is.

To some extent, the low number of content inclusion/exclusion defects could be attributed to the fact that in expressing the content criteria we were trying to find the criteria that would best fit the existing diagrams. It is interesting that despite this, there were some clear defects that we were able to find. For example, consider Figure 8.13 showing diagrams *M66* and *M70*. We inferred that the intent of *M66* was to show the set of PUMR error messages; however, in attempting to define the content criteria for this we discovered that these classes seem to be identified in two ways: as subclasses of the class

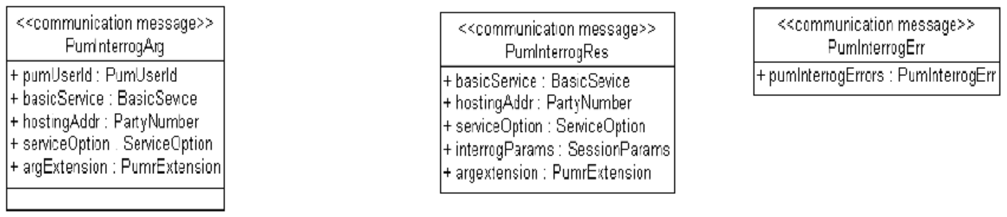
PUM Errors and as classes with the suffix “Err” in their name. According to this combined criterion, *M66* has an exclusion defect since it is missing the class *PumInterrogErr* found in diagram *M70*. In considering this we realized that this is probably *a defect of the SpecModel itself* and that *PumInterrogErr* ought to be a subclass of *PUM Errors* along with the other error classes. This would then make the content criterion of M66 simply be the set of subclasses of *PUM Errors*. This is an example of how the process of determining the correct content criterion could help detect defects not only in the diagrams but also in the model.

All of the examples of weak modeling defects relied on naming conventions to identify a type of element in the inclusion conditions. For example, diagram *M70* in Figure 8.13 shows the different interrogation message classes. These are all identifiable by having the stereotype “communication message” and by having a name with the prefix “PumInterrog” in their name. Cases that exhibited the unmodeled inclusion condition defect included diagram *M44*:*UCD* (in Figure 8.5) which shows the use cases for registration; however, there was no way to express the inclusion condition for registration use cases using only information in *SpecModel*.

Cases of potential incompleteness in the model occur when the modeler intent for diagrams suggests that a certain diagram ought to exist even though there is no content in the model for it. There are several examples of this in PUMR and two can be seen in


PUM Errors
<<communication message>>
PumRegistrErr
(from PUMR_Message_types)
+ pumRegErrors : PumRegErrors
<<communication message>>
PisnEnqErr
(from PUMR_Message_types)
+ pisnEnqErrors : PisnEnqErrors
<<communication message>>
PumDe-regErr
(from PUMR_Message_types)
+ pumDe-RegErrors : PumDe-regErrors
<<communication message>>
PumDelRegErr
(from PUMR_Message_types)
+ pumDelRegErrors : PumDelRegErrors

66: Contents of the PUMR error messages


<<communication message>>
PumInterrogArg
+ pumUserId : PumUserId
+ basicService : BasicSevice
+ hostingAddr : PartyNumber
+ serviceOption : ServiceOption
+ argExtension : PumrExtension
<<communication message>>
PumInterrogRes
+ basicService : BasicSevice
+ hostingAddr : PartyNumber
+ serviceOption : ServiceOption
+ interrogParams : SessionParams
+ argextension : PumrExtension
<<communication message>>
PumInterrogErr
+ pumInterrogErrors : PumInterrogErr

70: PUMR interrogation message types

Figure 8.13. Diagrams M66 and M70 showing a discrepancy on error messages.

Figure 8.5. The decomposition of *ReqModel* is indexed by *Task*, however, although there are five tasks, there are only *ReqUnit* contained macromodels for two of these: *Registration* and *Deregistration*. The remaining three tasks have no corresponding use cases or activity diagrams and this is probably an omission in the model. Another

probable omission is the fact that the use case diagram *M44* lists four use cases but there are only three activity diagrams (*M45-M47*) that elaborate the use cases. The modeler intent suggests that the set of activity diagrams is indexed by the set of use cases and thus there ought to be an activity diagram for each use case.

8.2 Application to UML Metamodel

The UML 2.0 specification [UML2] documents the large and complex UML metamodel using over one hundred diagrams. Thus, is a good example of a situation in which we can apply the techniques described in this thesis. In this example, we focus on the portion of the UML metamodel (henceforth, *UMLMeta*) relating to actions (Chapter 11 in the specification) and call this submodel *ActionMeta*. An *Action* is the fundamental unit of behavioural specification within a UML model and is used by other more complex behaviour specifications such as *Interactions*, *StateMachines* and *Activities*.

8.2.1 Expressing modeler intent

In total, nineteen diagrams are used to present *ActionMeta*. The macromodel reveals that this set of diagrams represents a three level hierarchical decomposition of the content of *ActionMeta*. We first discuss the decomposition criteria of the macromodel and then the macromodel itself.

Action classes, action types and entity operation types

The model *ActionMeta* is structured around the class *Action* and its subclasses which are the different types of actions supported. For example, action classes include *CallAction*, *SendSignalAction*, *DestroyObjectAction*, etc. In all, there are 46 action classes. Based on our analysis of the nineteen diagrams, it is apparent that these action classes are further classified into categories we call *action types*. We expect that as the metamodel evolves the action classes within an action type may change and the set of action types itself may change but action types will remain as a primary kind of classification of the content of *ActionMeta.*

A subset of the action classes deals with actions relating to entities such as objects, variables, links, etc. and for these, another kind of classification is evident as well. Entity action classes can mostly be categorized as either being a read operation or a write operation on the entity. Thus, we define entity operation types *read*, *write* and *base* for the entity actions not of the first two types.

Macromodel: top level (decomposition criteria)

Figure 8.14 shows that the macromodel of *ActionMeta* is completely structured as a hierarchy of indexed decompositions. At the topmost level, the content of *ActionMeta* is decomposed into four submodels on the basis of package using the generic *ByPackage* decomposer for *EMOF* models. Then, the content of each package submodel is further decomposed on the basis of the type of action described using the *ByActionType* decomposer. There are ten action types but not all are represented in each package. Finally, for some action types, the submodel is further decomposed. Here two different decomposers are used. The *ByClassType* is a generic decomposer that breaks the content of an *EMOF* model into submodels on the basis of the base classes much like the *ByBaseClasses* decomposer described in Chapter 7. The *ByOperType* decomposer breaks the content into submodels based on the type of entity operation being performed. This decomposer is only applicable to action types that operate on entities.

Macromodel: specific level

Figure 8.15 shows the remainder of the macromodel. This expands each of the *ByPackage* decompositions shown at the bottom of Figure 8.14.

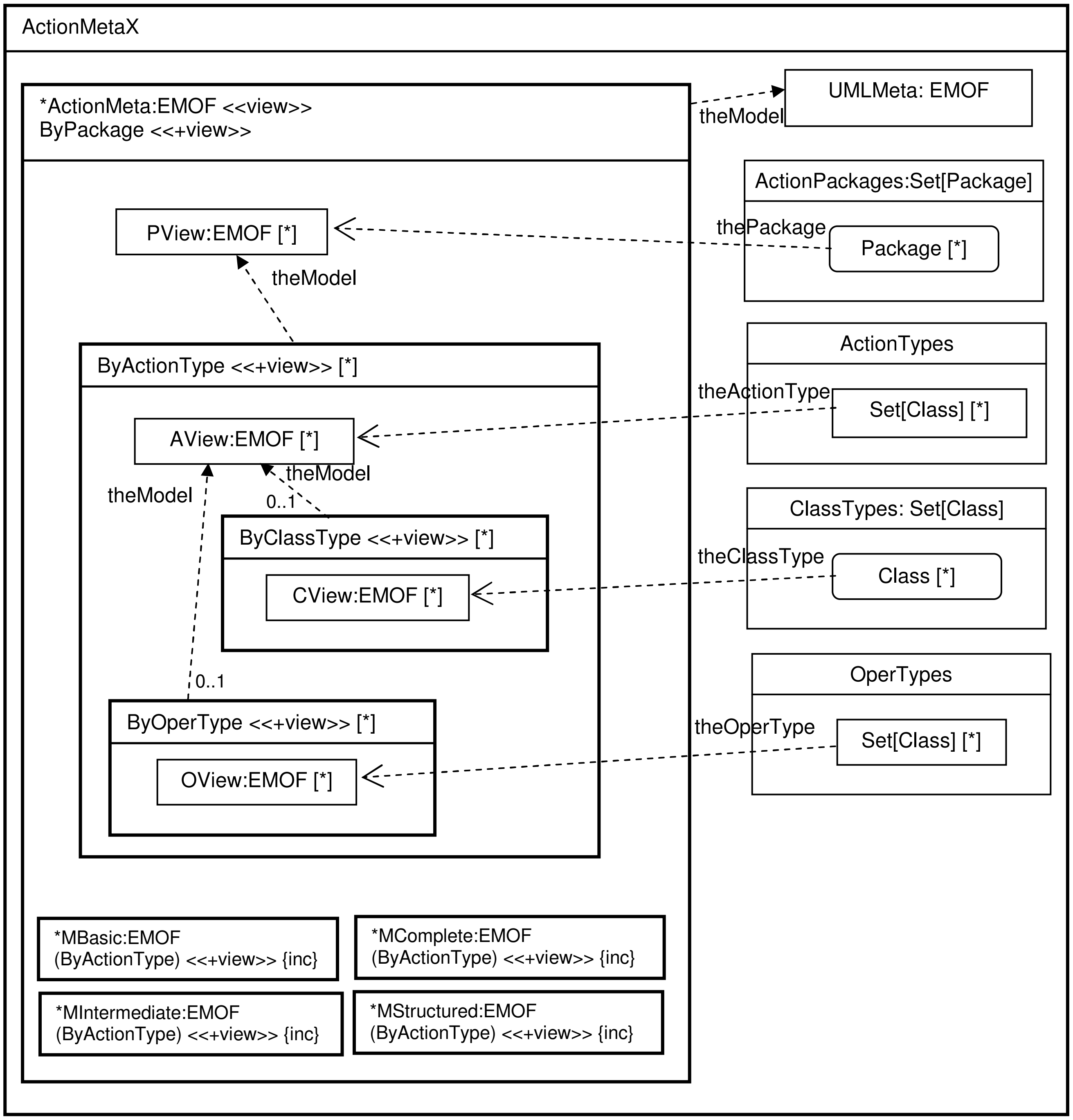


Figure 8.14. ActionMeta macromodel – top level.

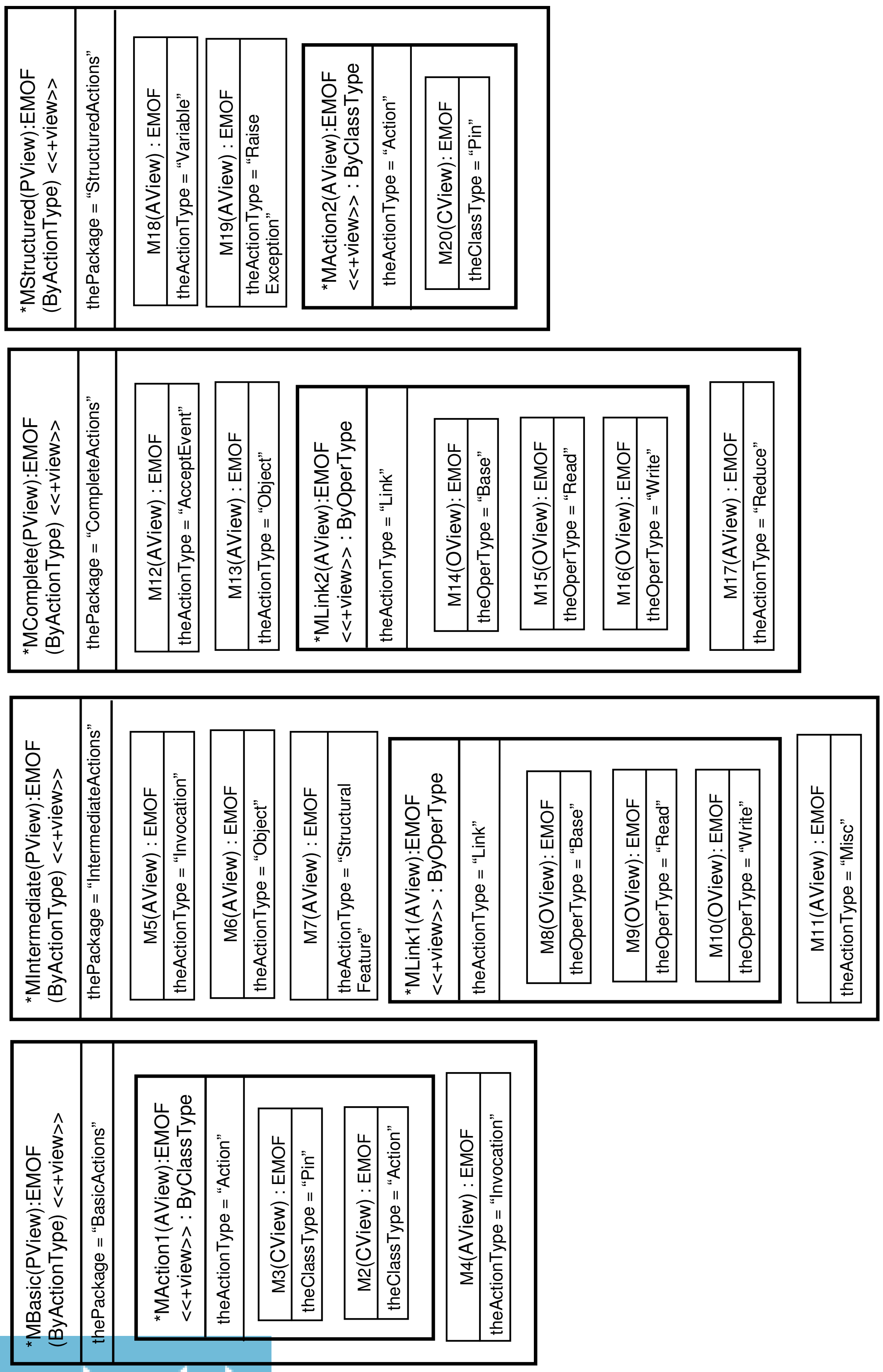


Figure 8.15. ActionMeta macromodel – all packages.

Content criteria

The content criterion of *ActionMeta* is given as follows:

$CC_{ActionMeta} := expandClasses(\text{UMLMeta}, classesInPkg(\text{UMLMeta}, \text{ActionPackages}))$

The definitions of the primitive extractors used here are defined in Appendix A. The index set *ActionPackages* = {*BasicActions*, *IntermediateActions*, *StructuredActions, CompleteActions*} contains the four packages within *UMLMeta* in which the core content of *ActionMeta* is found. Thus, the content criterion of *ActionMeta* first extracts the set of classes in these packages and then expands these to include their attributes, superclasses and navigable associations.

Table 8.7 shows the definitions of extractors used in defining the decomposition criteria for the four levels of decomposition. To get a better understanding of how these work to produce all the diagrams in decomposition of *ActionMeta*, we illustrate the generation process for diagram *M5*. First note that diagram *M5* is part of the decomposition of *MIntermediate* which is the view of *ActionMeta* representing the contents of the intermediate package. The content criterion of *MIntermediate* uses the first extractor *PDetail* and is defined as:

$CC_{\text{MIntermediate}} := PDetail(UMLMeta, Package(\text{“IntermediateActions”}))$

Table 8.7. Extractors used for decompostion criteria in the ActionMeta example.

Index model	Definition
ActionPackages	PDetail(theModel, thePackage) := *expandClasses*(theModel , *classesInPkg*(theModel, thePackage))
ActionClassTypes	ADetail(theModel, theActionType) := restrictTo(theModel, *expandClasses*(ActionMeta, *supportingClasses*(ActionMeta, theActionType)))
ClassTypes	CDetail(theModel, theClassType) := restrictTo(theModel, *expandClasses*(ActionMeta, *classSubsOf*(ActionMeta, theClassType)))
OperTypes	ODetail(theModel, theOperType) := restrictTo(theModel, *expandClasses*(ActionMeta, *supportingClasses*(ActionMeta, theOperType)))

Figure 8.16 shows the construction of *MIntermediate* using this content criteria. The extractor *classesInPkg*(*UMLMeta*, Package("*IntermediateActions*")) first produces the submodel *P1* that contains the classes in the "intermediate" package. Note that we only show a portion of this submodel here. Then the application of *expandClasses* to this submodel extends it to include the immediate superclasses and navigable associations and this results in *MIntermediate*. Diagram *M5* is a view of *MIntermediate* and its content criterion is:

CC_{M5} := *ADetail*(*MIntermediate*, *ActionTypes.Invocation*)

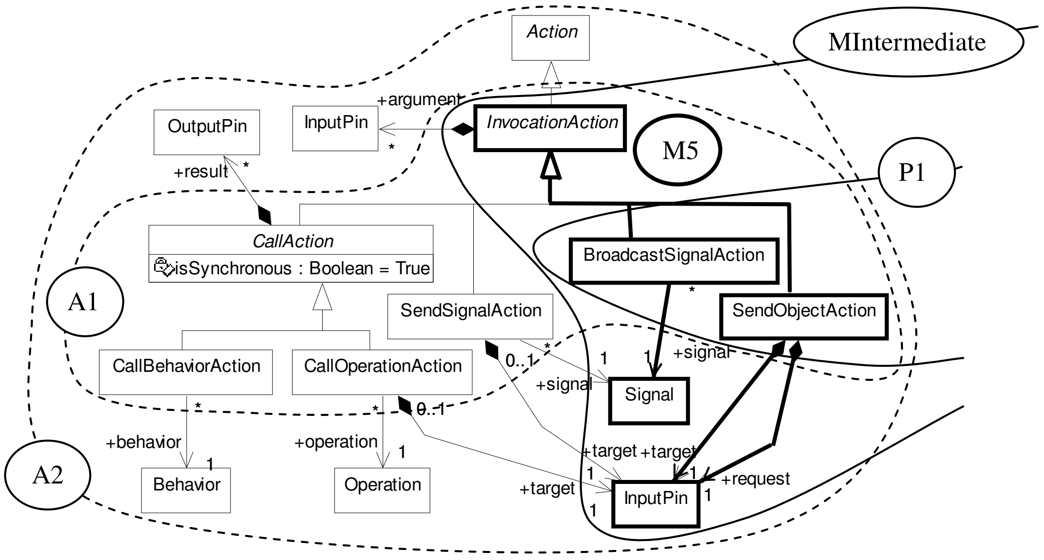


Figure 8.16. The generation of diagram M10.

First *supportingClasses* (*ActionMeta*, *ActionTypes.Invocation*) produces the submodel *A1* consisting of all the subclasses of the abstract class *Action* that are "invocation" actions expanded to include supporting classes. Then *expandClasses* adds the immediate superclasses and navigable associations to produce *A2*. Finally, the *restrictTo* operation yields diagram *M5* shown with bold lines.

8.2.2 Findings

Added structural information

Table 8.8 summarizes the amount of structural information made explicit by the macromodel.

Table 8.8. Summary of the structural information made explicit in the UML example.

Structural information measure	Value
Number of diagram groups	8
Number of hierarchies (depth)	1 (2)
Number of relationships	0
Index sets (uses)	Package (1), Action type (4), Class type (2), Entity operation (2)

Index sets

It is interesting to note that this macromodel is completely expressed as a hierarchical decomposition and any diagram in it can be generated from its tuple of “indices.” For example, the indices of diagram *M5* are [“Intermediate”, “Invocation”]. Thus, all the diagrams can be generated from relatively little information. One use of this is that it provides a simple way of generating or regenerating decompositions when the model changes.

Another observation is that, as with the PUMR example, a combination of domain-specific and generic decomposers are used. In this case, *ByPackage* and *ByClassType* are generic and could be used to decompose any EMOF model. In contrast, *ByActionType* and *ByOperType* are specific to the action class domain.

Level Alignment

One way in which the structural information added by the macromodel helped to explain the rationale for the diagrams of *ActionMeta*, was by showing the "alignment" between the diagrams of different packages – there is a comprehension advantage to decomposing each package using the same principles of division. For example, the *Link* action type is further decomposed using *OperTypes* both in the *IntermediateActions* and *CompleteActions* packages. This could account for the fact that although diagrams *M9* and *M16* are small enough (3 classes each) that they might be combined with their sibling diagrams, they are nevertheless kept separate. It seems that the value of maintaining a consistent decompositional structure for the same action types in different packages outweighs the value of having diagrams of balanced size. This kind of alignment holds across the entire decomposition and may be a new facet of decomposition quality not addressed by Moody's quality principles [M00].

Defects detected

Figure 8.17 summarizes the defects found in the UML metamodel example. See Appendix A for the full description of these. Here we discuss some of the more representative and interesting findings.

Since every diagram of *ActionMeta* can be generated from a small number of indices, it is reasonable to define a uniform naming scheme that reflects this. Each diagram is generated by a package, action type and optionally a class type or entity operation.

Despite this, there are significant naming inconsistencies. For example, the way the package is mentioned in the diagram name is not consistently followed. In the *BasicActions* package, every diagram name is prefixed with the word “Basic” while in the *CompleteActions* package, some diagrams have a suffix of “(CompleteActions)” – usually (but not always) when it has the same action type as a diagram in *IntermediateActions* or *BasicActions*.

Although the content criteria were determined from the existing content of the diagrams, the indexing structure forces multiple diagrams to share the same content criteria (differing only in the generator values). Since this means that this shared content criteria is a “best fit” to the set, it is possible to discover defects with diagrams that do not fit as well (i.e., are “outliers”). This allowed us to detect two cases of content inclusion defects.

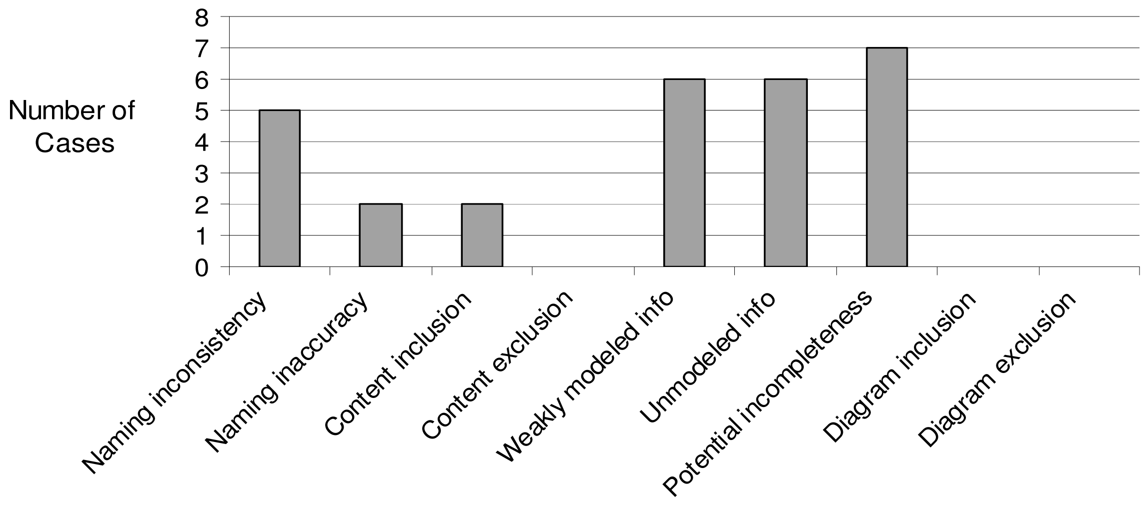


Figure 8.17. Defects found in UML metamodel example.

Specifically, diagram *M3* contains three extra elements that are not accounted for by the content criteria and diagram *M10* contains two extra elements.

Given the index models, all the extractors for diagrams in *ActionMetaX* are completely determined. Thus, all of the causes for weakly modeled or unmodeled information defects in *ActionMeta* arise due to the index models. Furthermore, both *ByPackage* and *ByClassType* are generic decomposers and so the extractors for *ActionPackage* and *ClassTypes* , respectively, are completely determined. The index model *OperTypes* is fully determined but relies on weakly modeled information since it uses naming conventions rather than a modeled relationship to determine which action classes correspond to operation types. For example, the nine actions classes with operation type *Read* can be identified by having the prefix "Read" or "Test" on their name. The drawback of weakly modeled information is that it relies on informally enforced conventions and these may not be properly managed as the UML metamodel evolves.

The content criteria of *ActionTypes* are partially underdetermined since there is not enough information in *ActionMeta* in all cases to determine to which action type an action class belongs. In particular, of the ten action types, the membership in six of them can be determined by whether a class extends an abstract base class. For example, the set of action classes of action type "Invocation" are exactly those that are the subclasses of the abstract class *Invocation*. However, the classes of action type "Object" have no such abstract class and so these must be explicitly enumerated within the *Object* action type in *ActionTypes*. The fact that most of the action types are determined by an abstract base class suggests that a plausible and consistent mode of repair is to add an abstract base class to *ActionMeta* for all “Object” action classes. Fixing this for the other three action types with this problem (“Link”, “Misc” and “AcceptEvent”) would ensure that the concept of an action type is modeled within *ActionMeta* and not just within the diagrams of it.

The strong indexing of the diagrams in *ActionMetaX* provides several opportunities to check for potential incompleteness of *ActionMeta*. For example, within the decomposition of package *CompleteActions*, there is no diagram (or group) corresponding to the action types “Action”, “Invocation”, “StructuralFeature” or “Misc”. Similarly, there is no class type *Pin*, *EndData* or *Qualifier* in the *Action* group within *StructuredActions*.

8.3 Summary

In this chapter we applied the techniques for expressing modeler intent described in this thesis to two example modeling projects. A key objective here was to assess whether the claims about improving model quality and comprehension could be substantiated. Table 10.1 summarizes the results. We take these results to support the claim that expressing the role level can improve model quality because in both examples a significant number of defects were found that could not have been identified without expressing the modeler intent. We also take the results to support the claim of improved comprehension because in both cases a substantial amount of intent-related information is added and as discussed at the start of this chapter, this kind of information has been shown to support comprehension.

Some additional interesting findings that impact comprehension include the following:

- In the PUMR example, the same information about messages is decomposed in two alternative ways in order to highlight different aspects of it. This was not evident until the macromodel was constructed.
- There are many examples of unrealized views whose presence is implied by the fact that a subset of diagrams are intended to decompose them. These unrealized views are only evident in the macromodel but are crucial to understanding why certain subsets of diagrams exist.
- In the UML example, the entire set of diagrams forms a tight decomposition hierarchy. In fact, the content criteria for all the diagrams is formed from composing just four extractors by using different combinations of index values. It is quite surprising that the content of all the 19 diagrams could be constructed from such little information in a systematic way. This systematicity helps to explain some of the "unintuitive" aspects of the diagrams such as why some of the diagrams seem unusually small or large.
- Both examples use a mixture of generic and domain-specific extractors and decomposers. This shows that the use of generic expressions of intent predefined at the method-level is useful and can reduce the effort required to express and understand the expressions of intent. However it also shows that this by itself is insufficient to express all the modeler intent at the project-level.

In addition to quality and comprehension, the fact that we were able to formally express most of the modeler intent in these examples supports the claim that this could be used to drive automation. The prototype described next in Chapter 9 further supports this conclusion.

Table 8.9. Summary of evaluation results

Finding	PUMR	UML
Defects or potential defects	28	28
Content Criteria	42	19
Relationships	16	0
Indexed decompositions	6	4

Chapter 9

Tool Support

In this chapter we present the details of the prototype implementation MCAST (Macromodel Creation and Solving Tool)[28] that uses a model finder to solve macromodel conformance problems. The objective in creating the prototype was to show that the formal expressions of modeler intent found in macromodels can be feasibly used to support automation.

MCAST is built in Java on the Eclipse modeling frameworks (EMF, GMF, GEF). Figure 9.1 shows the architecture of MCAST. Macromodels can be created or edited and then the Solver module, that incorporates the Kodkod [TJ07] model finding engine, can be used to solve macromodel conformance problems using simplified macromodels.

[28] An earlier implementation of macromodels was the Model Management Tool Framework (MMTF) [SCE07] but this has since evolved in a different direction.

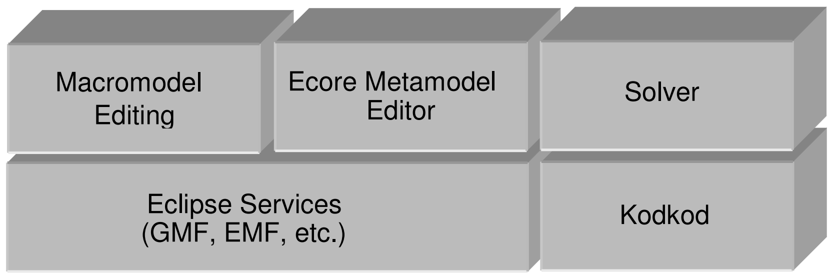


Figure 9.1. MCAST Architecture.

As discussed in Chapter 6, there are different automation usage modes for macromodels including conformance checking and extension to conformance where the latter allow can be used to synthesize models and mappings and do conformance checking with incomplete information. In this chapter we describe the details of the Solver module and its use in these modes of operation. Since a macromodel assumes the existence of a set of model and relationship types, we begin by describing the implementation of these within MCAST.

9.1 Defining model and relationship types

MCAST leverages the modeling infrastructure provided by Eclipse. The EMF (Eclipse Modeling Framework) provides a metametamodel called Ecore. A simplified view of this is shown in Figure 9.2. The sorts and predicates of a metamodel signature are represented by *EClass* and *EReference* elements, respectively. An *EReference* represents a binary predicate from its containing *EClass* to the target *EClass* given by the value of *eType*. The multiplicity information of an *EReference* applies to the target *EClass*.

EAnnotations allow additional application-specific information to be included within a metamodel and can be attached to any model element. MCAST assumes that every metamodel *T* contains a root *EClass* with name *T* and marked with the annotation “Root”. In addition, it can have the annotation “Model”, “Relationship” or “Macromodel”. Annotations are also used to store metamodel constraints within the metamodel and these are expressed in the Kodkod relational logic-based rule language. Since Kodkod does not provide a textual notation for its rule language, MCAST provides one. To accomplish this, the ANTLR parser generator for *LL*(*k*) parsers was used [ANTL] to construct a parser and translator of the MCAST textual notation to internal Kodkod rules expressed as Java objects. The grammar for the MCAST notation for the Kodkod rule language is given in Appendix D.

In the formal treatment given in Chapter 4, relationship types are expressed using a relator model plus metamodel morphisms. In MCAST we exploit the fact that within EMF, metamodels can directly reference other metamodels and thus, rather than replicate the elements of the endpoint model types within the relator metamodel, they are instead expressed as external references into the endpoint metamodels. The result is that the relator metamodel contains only the parts that are unique to the relationship type – i.e., the mapping content.

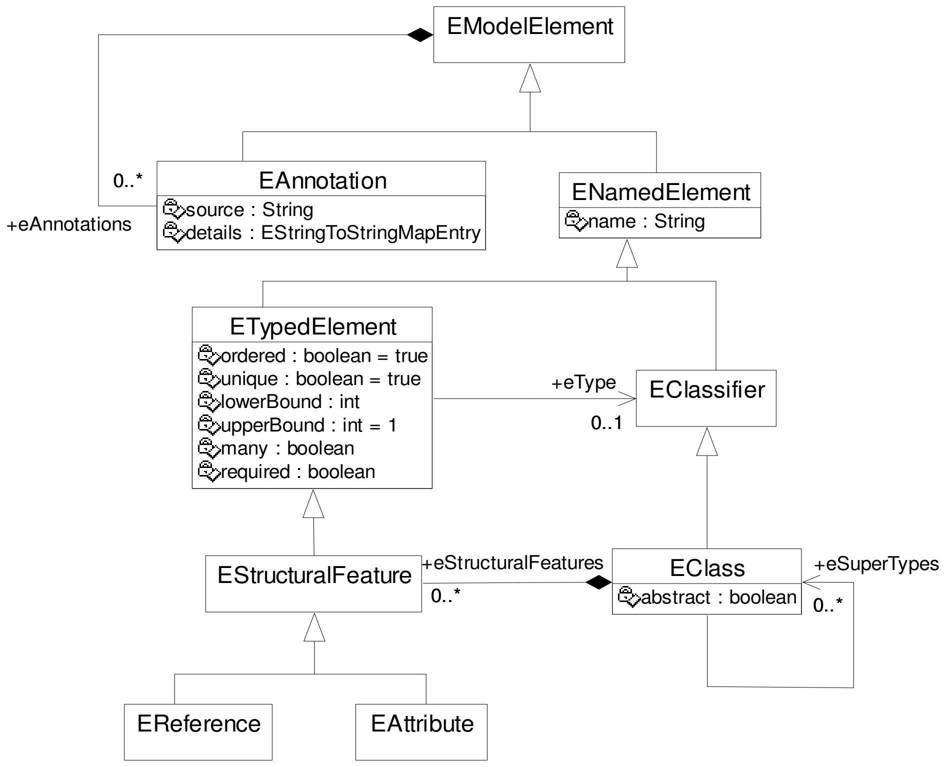


Figure 9.2. The simplifed Ecore metametamodel.

Figures 9.3 shows the signature portion of the metamodel for the *ObjectsOf* relationship type defined in this way[29]. Note that although the signatures of the *OD* and *SD* metamodels appear to be included in Figure 9.3, these are actually only "shortcuts" to the external metamodels. The shortcut symbol is indicated in the lower left corner of each

[29] This is based on the definition of *objectsOf* in Chapter 4 and not the one used in the PUMR example of Chapter 8.

external element by the small arrow in a box. Figure 9.4 shows the constraints portion of the *ObjectsOf* metamodel using our textual version of the Kodkod language.

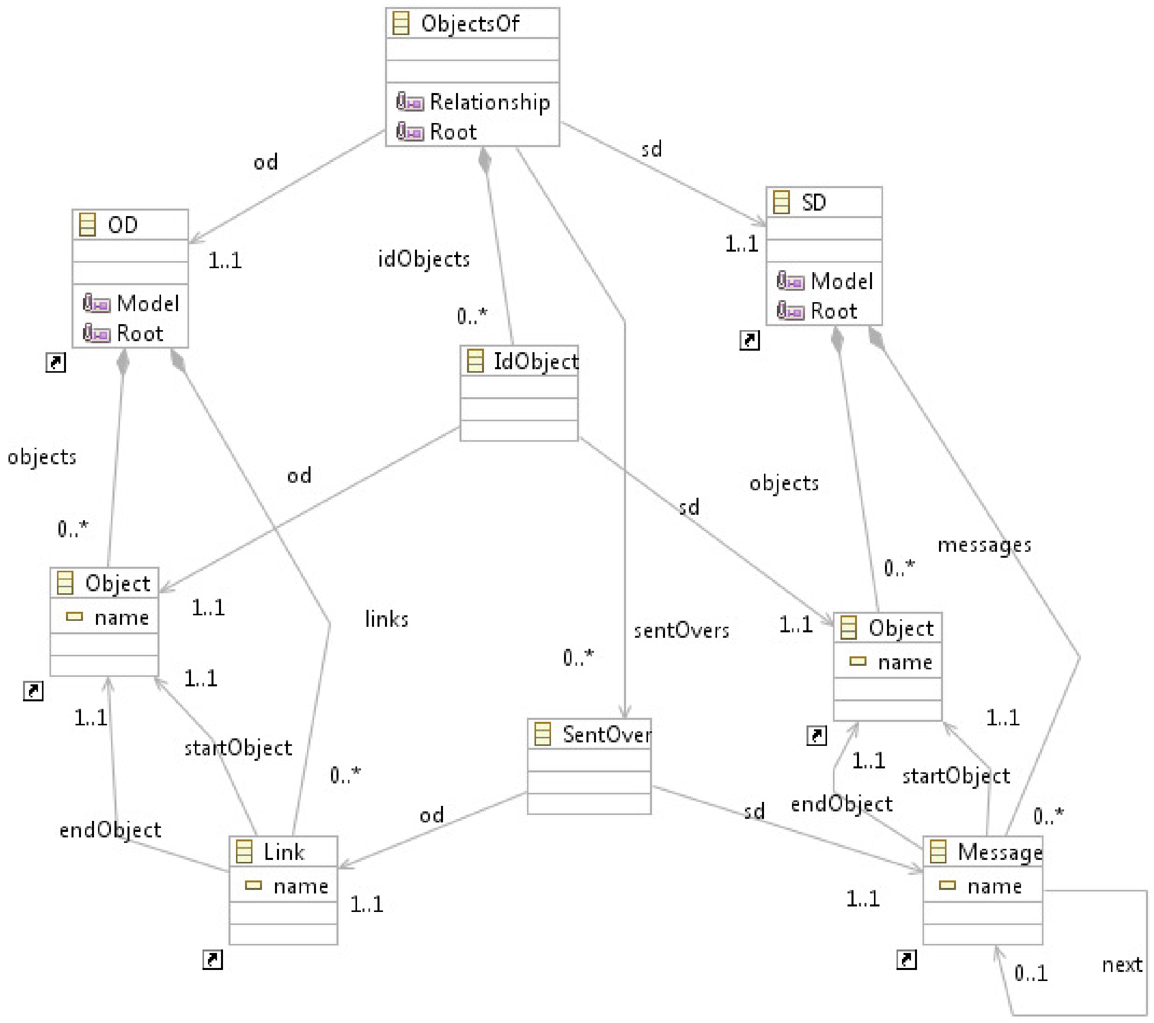


Figure 9.3. The signature part of the *ObjectsOf* metamodel.

```
Constraints

// every object in the sd is mapped to an object in the od
[all of:ObjectsOf][all o:((of.ObjectsOf_Sd).SD!SD_Objects)] o in
     ((IdObject_Od.((of.ObjectsOf_Od).OD!OD_Objects)).IdObject_Sd);
[all of:ObjectsOf][all o:((of.ObjectsOf_Sd).SD!SD_Objects)] one
      IdObject_Sd.o;

// every message in the sd is SentOver to a link in the od
[all of:ObjectsOf][all m:((of.ObjectsOf_Sd).SD!SD_Messages)] m
    in ((SentOver_Od.((of.ObjectsOf_Od).OD!OD_Links)).SentOver_Sd);
[all m:SD!Message]one SentOver_Sd.m;


// incidence preservation
[all so:SentOver][some ios:IdObject][some ioe:IdObject]
   {(so.SentOver_Sd).SD!Message_StartObject = (ios.IdObject_Sd)} and
   {(so.SentOver_Sd).SD!Message_EndObject = (ioe.IdObject_Sd)} and
   {
     {
      {(so.SentOver_Od).OD!Link_StartObject = (ios.IdObject_Od)} and
      {(so.SentOver_Od).OD!Link_EndObject = (ioe.IdObject_Od)}
     } or {
      {(so.SentOver_Od).OD!Link_StartObject = (ioe.IdObject_Od)} and
      {(so.SentOver_Od).OD!Link_EndObject = (ios.IdObject_Od)}
     }
    };

// every object in od has an object in sd mapped to it
[all o:OD!Object] some IdObject_Od.o;

// every link in od has a message in sd mapped to it
[all l:OD!Link] some SentOver_Od.l;
```

Figure 9.4. The constraints part of the *ObjectsOf* metamodel.

9.2 Using the solver with macromodel conformance problems

MCAST allows the type diagram of a macromodel to be defined as an Ecore metamodel that extends the base metamodel shown in Figure 9.5. Each model and relationship type is given as a subclass of classes *Model* and *Relationship*, respectively. These carry a static attribute *typeHref* to reference the Ecore metamodels they denote. A macromodel based on this type diagram is then defined as an instance of this Ecore metamodel.

The Solver takes as input, a macromodel with a subset of the model and relationship elements containing references to existing artifacts (i.e., models and relator models) via the attribute *contentHref*. In addition, any model or relationship element may be marked *incomplete* by setting attribute *isComplete* = *FALSE* (default = *TRUE*). A solution to the macromodel conformance problem for this macromodel is then a set of artifacts assigned to the model and relationship elements that includes the existing artifacts and extensions to artifacts marked *incomplete* that satisfy the constraints expressed by the macromodel.

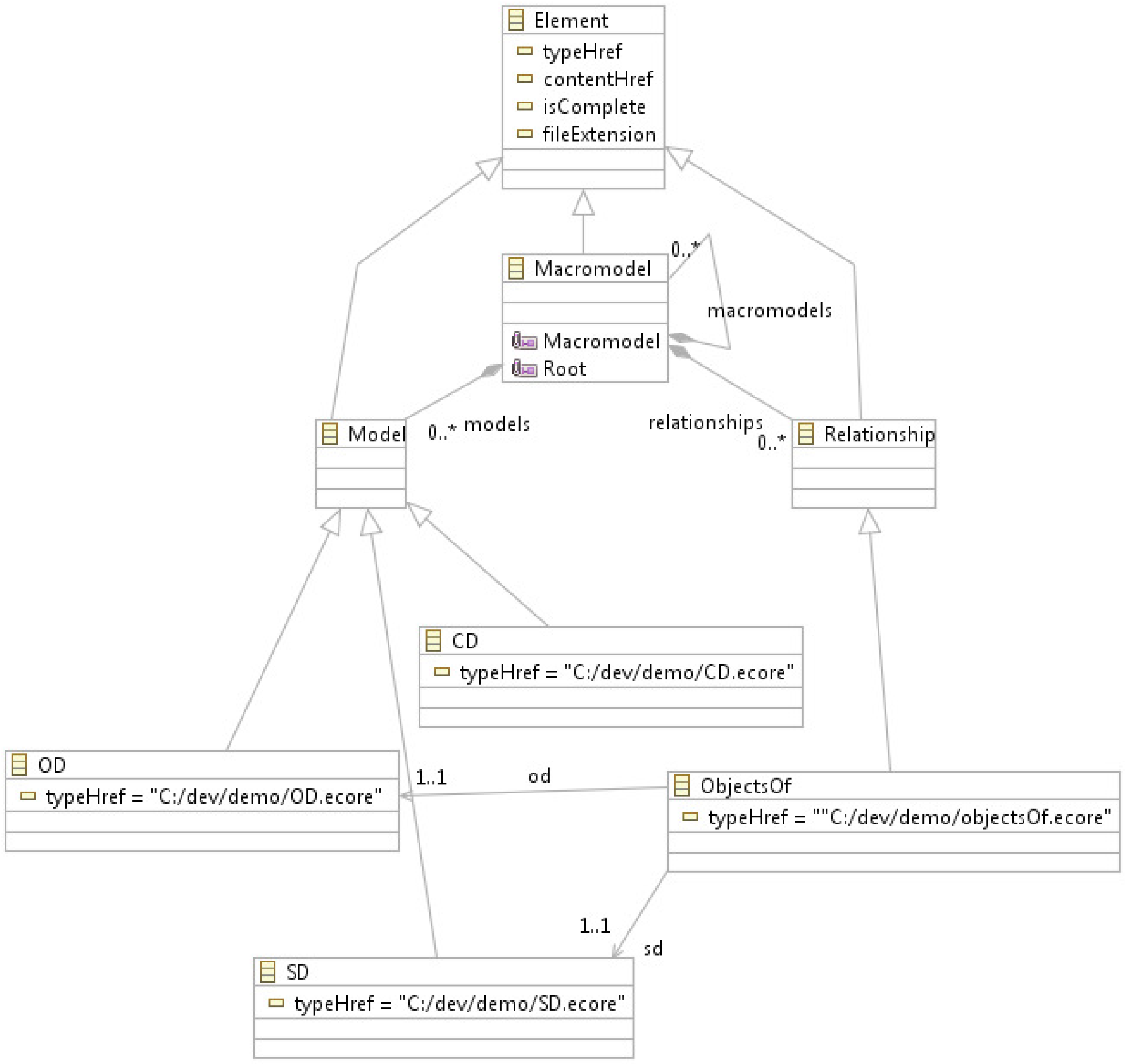


Figure 9.5. Example type diagram as an extension of a base metamodel.

The Solver finds such a solution by constructing and solving a corresponding Kodkod model finding problem. This can be used with the different conformance modes as follows:

- Conformance checking mode: If the input consists has all of the realized elements marked as *complete* and assigned to existing artifacts then a solution exists to the Kodkod problem iff this is a conformant collection of artifacts.
- Extension-to-conformance mode: If some of the model and relationship elements are marked as incomplete, then Kodkod will attempt to synthesize extensions to these that result in a conformant collection of artifacts. Recall from Chapter 6 the two ways to use this mode:
    - Synthesis. When a solution is found and models and relationships are extended as part of finding a conformant solution they are guaranteed to be consistent with the existing models and relationships (in the sense that all constraints are satisfied) but this does not mean they are necessarily correct since there may be many possible consistent extensions. When the solution is unique, however, then it *must* be correct and hence this provides a way to do model and relationship synthesis.
    - Conformance checking with incomplete information. If a solution cannot be found, this indicates that there is no way to consistently extend the incomplete models and relationships and hence the existing artifacts are definitely non-conformant even though they are incompletely specified.

9.3 Solver

MCAST solves macromodel conformance problems by translating them to Kodkod model finding problems. The overall steps of this algorithm are as follows:

1. Translate macromodel conformance problem to Kodkod problem.
2. Perform the Kodkod model finding operation
3. Reconstruct EMF models from the Kodkod solution.

Of these steps, the key algorithm is step (1). The reconstruction of the resultant EMF model in step (3) is a direct reversal of the construction of the Kodkod problem in step (1). We first describe what a Kodkod problem is and then we present the algorithm for translating a macromodel conformance problem to a Kodkod problem.

9.3.1 Kodkod problems

A Kodkod problem is a 3-tuple $K = \langle Rels, Formula, Universe \rangle$ consisting of the following:

- *Universe* : A set of atoms from which all tuples in a solution are built.
- *Rels* : A set of relation declarations which includes the arity of the relation as well as its lower bound and the upper bound. The lower bound is the set of tuples of atoms that must occur in every solution. The upper bound is the set of tuples of atoms that may occur in a solution.
- *Formula* : A relational logic expression over the relations *Rels* expressed using Kodkod's rule language.

A solution $Sol_K = \{T_R \mid R \in Rels\}$ to K consists of a collection of sets T_R of tuples assigned to the relations such that *Formula* is satisfied. Note that the tuples in T_R are drawn from the upper bound for R and will always contain the tuples in the lower bound for R. Kodkod finds solutions by translating K into a SAT problem and uses one of the many available SAT solvers to solve this problem.

9.3.2 Macromodel translation algorithm

A solution to a macromodel conformance problem $Q = \langle M{:}T, C\rangle$ is obtained by extending the models that are marked incomplete with additional elements and predicate instances so that the resulting set of models is conformant with the macromodel constraints. The macromodel conformance problem Q can be translated to an equivalent Kodkod problem K_Q as follows:

1. Translate all signature elements in metamodels referenced by T to relation declarations to form $Rels_Q$
2. Translate all constraints in metamodels referenced by T to construct the formula $Formula_Q$
3. Translate the existing content of models/relationships referenced by M to define atoms in $Universe_{Q.}$ and to form the lower bounds on the relations in $Rels_{Q.}$
4. Construct the upper bounds on the relations in $Rels_Q$ as the lower bounds plus additional tuples to accommodate the possible extensions of the models/relationships marked incomplete. This also requires additional atoms in $Universe_Q$ that represent possible new elements in these models/relationships.

Making the existing content of models/relationships the lower bound in step (3) ensures that every solution will be an extension to the existing content. In step (4), the upper bounds define which models/relationships can be extended and by how much.

The detailed algorithm for translating a macromodel conformance problem into a Kodkod problem is shown in Figures 9.6 and 9.7. The first part handles steps (1) and (2) and the second part handles steps (3) and (4). Since Kodkod is a bounded model finder, it is necessary to specify an amount by which incomplete models/relationships can be extended. For the current prototype this is accomplished by using a single integer extension parameter n_{ext} as an input to the translation algorithm. This is interpreted as saying that each type of element in an incomplete model/relationship can be extended by n_{ext} elements. For example, if $n_{ext} = 3$ and there is an incomplete model M_1:*ObjectDiagram* in *M*, then in a solution the extended version of M_1 may have up to 3 more *Object* elements and up to 3 more *Link* elements.

In the first part of the algorithm, from each metamodel *T*, *EClasses* (i.e., sorts) are translated to unary relations and *EReferences* (i.e., predicates) are translated to binary relations and are named using the names from the metamodels (lines 5, 7 and 11). For example for *T* = *ObjectDiagram* we get unary relations *ObjectDiagram*!*ObjectDiagram*, *ObjectDiagram*!*Object*, *ObjectDiagram*!*Link*, and binary relations *ObjectDiagram*!*ObjectDiagram_links*, *ObjectDiagram*!*ObjectDiagram_objects*, *ObjectDiagram*!*Link_startObject* and *ObjectDiagram*!*Link_endObject*. These are the names used to refer to relations within rules expressed using the MCAST rendering of the

Kodkod rule language. The qualifier "*ObjectDiagram*!" is used to avoid name clashes since many metamodels are merged into one set of relations, however this can be omitted within rules when there is no referential ambiguity. As a convenience for expressing rules, when *EClass S* is not an abstract class and has subclasses, a special unary relation named *T!@@S* is defined that represents the instances of *S* that are not found in any of its subclasses (line 7).

The value of *Formula* is constructed as a conjunction of constraints that come from three sources: subclass relationships (line 9), multiplicities on *EReferences* (line 12) and explicitly expressed rules as annotations within the metamodel using the MCAST rendering of the Kodkod rule language.

Part 2 of the algorithm translates the models referenced by the input macromodel M_{inp} and builds the *Universe,* lower bounds and upper bounds on the relations in *Rels.* EObjects are translated to atoms in the *Universe* in line (17) - atoms are qualified with the model name so that they are unique across M_{inp}. For incomplete models/relationships, n_{ext} additional "extension" atoms are defined for each *EClass* and added to the *Universe* (lines 20-21). The lower bounds for relations corresponding to *EClasses* and *EReferences* consist of the content of the existing models (line 19 and 29). Note that relationships may contain *EReferences* whose target elements are outside of the relationship and in one of the endpoint models that are related by the relationship. This is why in line 29 and 31, the expression *model*(S_2) is used as the model for the target element rather than *M.* The upper bounds for complete models/relationships are the same as the lower bounds and thus will not be extended (lines 25 and 33). The upper bounds for incomplete model/relationships use the additional extension atoms (lines 23 and 31).

If we consider the worst-case complexity of this algorithm, part 1 is $O(n_c+n_r)$ where n_c is the number of *EClasses* and n_r is the number of *EReferences.* Part 2 has complexity $O(n_o + n_c n_{ext} + n_r(n_c + n_{ext})^2)$ where n_o is the number of *EObjects* in the existing models. The second and third terms are due to lines 21 and 31, respectively. These are required in order to build a sufficiently large upper bound for incomplete models and relationships.

9.4 Discussion

The exercise of defining and implementing the macromodel conformance problem solving algorithm using Kodkod validates the idea that support for these problems can be automated; however, it also highlights a number of problems with this approach to automation. First, since the (bounded) relational model finding problem is NP-complete, the problem of finding a model that satisfies a set of first order constraints (e.g. role constraints as we have discussed them) is intractable. This fundamentally limits the scalability of the approach. Second, since determining whether a set of first order constraints has a model, is in general undecidable, we cannot determine when to stop increasing the bound on a model finder and "give up" because there exists no model.

In our case, the bound represents the amount by which we can extend an incomplete model and is defined by the parameter n_{ext}. The first problem can be "managed" by providing user control over the bound that is used. For the second problem, fortunately, there are some common cases in which it is possible to compute maximal bounds for model extension in the sense that if a conformant extension cannot be found within the bounds then one does not exist. For example, a mapping such as *objectsOf* defined in Figures 9.3 and 9.4 is bounded by the size of the models on its endpoints since it can have no more *IdObject* elements than *Object* elements in the source *SD* and no more *SentOver* elements than messages in the source *SD*. Thus, if the *SD* is considered complete and the *objectsOf* mapping is incomplete then no conformant extensions of the mapping can exceed the bounds defined by the *SD*.

In general, the only way to surmount both of these problems is to limit the class of role constraints that can be expressed to one for which the model finding algorithm is tractable and the maximal bound is decidable. We leave the further investigation of this to future work.

9.5 Summary

In this chapter we describe in detail the MCAST prototype for solving macromodel conformance problems using the Kodkod model finder. The objective of creating MCAST is to validate the claim that macromodels can be used to support automation in modeling. To this end, we describe the algorithm that is used for translating a subset of the macromodel language into Kodkod and discuss how MCAST is used to automate the conformance checking and extension-to-conformance usage modes of macromodels.

Algorithm. TranslateMacromodel

Input: Macromodel M_{inp} with metamodel T_{inp} and extension parameter n_{ext}

Output: Kodkod problem $K = \langle Rels, Formula, Universe \rangle$

```
1:   $Rels = \emptyset$, $Formula = TRUE$, $Universe = \emptyset$
2:   // Translate metamodels to build Rels and Rules
3:   for every metamodel $T$ referenced by $T_{inp}$ :
4:     for every EClass $S \in T$ :
5:       add unary relation "T!S" to Rels
6:       if $S$ is not an abstract class and has subclasses:
7:         add unary relation "T!@@S" to Rels
8:       if $S$ has subclasses $\{S_1, S_2, \ldots, S_n\}$ :
9:         add constraint "$T!S = T!S_1 + T!S_2 + \ldots + T!S_n$" to Formula
10:    for every EReference $P(S_1, S_2) \in T$ :
11:      add binary relation "$T!S_1\_P$" to Rels
12:      add constraints on $T!S_1\_P$ to Formula to implement the multiplicities of $P$
13:    for every constraint $R$ in an annotation $A \in T$ : add $R$ to Formula
```

Figure 9.6. The algorithm for translating a macromodel to a Kodkod problem (part 1).

```
14: // Translate models to build $Universe$, relation upper bounds and lower bounds
15: for every model or relationship $M$ with type $T$ referenced by $M_{inp}$ :
16:   for every EObject $o \in M$ :
17:     add atom $Atom[o, M]$ to $Universe$
18:   for every EClass $S \in T$ :
19:     $LowerBound[S, M] = \{Atom[o, M] \mid o \in M$ and $o$ is of type $S\}$
20:     if $M$ is marked $incomplete$ :
21:       $Ext[S, M] = \{Atom[\mathbf{S\_\_1}, M], Atom[\mathbf{S\_\_2}, M], \ldots, Atom[\mathbf{S\_\_}n_{ext}, M]\}$
22:       add $Ext[S, M]$ to $Universe$
23:       $UpperBound[S, M] = \cup\{Ext[S_1, M] \mid S_1$ is $S$ or a subclass of $S\} \cup$
                 $LowerBound[S, M]$
24:     else
25:       $UpperBound[S, M] = LowerBound[S, M]$
26:     add $LowerBound[S, M]$ to lower bound for relation $\mathbf{T!S}$ in $Rels$
27:     add $UpperBound[S, M]$ to upper bound for relation $\mathbf{T!S}$ in $Rels$
28:   for every EReference $P(S_1, S_2) \in T$ :
29:     $LowerBound[P, M] = \{\langle Atom[o_1, M], Atom[o_2, model(S_2)]\rangle \mid P(o_1, o_2) \in M\}$
30:     if $M$ is marked $incomplete$ :
31:       $UpperBound[P, M] = UpperBound[S_1, M] \times UpperBound[S_2, model(S_2)]$
32:     else
33:       $UpperBound[P, M] = LowerBound[P, M]$
34:     add $LowerBound[P, M]$ to lower bound for relation $\mathbf{T!S_1\_P}$ in $Rels$
35:     add $UpperBound[P, M]$ to upper bound for relation $\mathbf{T!S_1\_P}$ in $Rels$
36: return $Rels, Formula, Universe$
```

Figure 9.7. The algorithm for translating a macromodel to a Kodkod problem (part 2).

Chapter 10

Conclusion

Modeling is used widely within software engineering and it has been studied from many perspectives; however, a perspective that has received little attention is the role of modeler intent in modeling. Since a model is a linguistic artifact, the modeler is seen as both the utterer of the model (as linguistic expression) and the creator of the model (as artifact). Correspondingly, knowing the intent of the modeler is valuable both for determining how the model should be interpreted semantically and for assessing its quality. Furthermore, formal expressions of this intent allow automated support for these. Despite the value that the knowledge of modeler intent would provide, there are no adequate means in the current state of modeling practice for expressing this information. The focus of this thesis is to address this gap by providing mechanisms for expressing modeler intent both explicitly and formally.

10.1 Summary of approach

We approached this problem by first recognizing that the purpose of a model is always to provide a particular set of information required by stakeholder activities. Then when modelers create a collection of models in order to satisfy the requirements of stakeholders, they do so with an idea of what role each model plays in the collection – i.e., what information it should contain and how this information is related to the information in other models. The specification of these roles is what we refer to as the expression of modeler intent. A key contribution of this thesis is the identification of this "role level" as being an important aspect of a modeling project alongside the "model level" at which the content of models is created.

We presented a framework that incorporates four components of modeler intent at the role level: the existential intent for a model that arises in response to the need for information by stakeholders, the content criteria that express what information the model is intended to contain, the model relationships that express how models are intended to constrain one another and the decomposition criteria that express the intent behind how a model is decomposed into a collection of models. A major contribution of this thesis is the specification of the macromodeling language as a new modeling language designed for the role level that supports the expression of all four aspects of modeler intent.

The macromodeling language is based on a formal approach to expressing model relationships and relationship types using metamodels. Relationship types combine the definition of mapping information and constraints in order to define standard ways that models can be related. This also allows various properties of relationships to be rigorously defined. In particular, a relationship type property identified as having central importance is "partiality" and many commonly used relationship types such as *submodelOf*, *abstractionOf*, *refinementOf*, etc. in software engineering have this property. Partiality relationship types represent different ways that one model can carry a part of the information in another model and are the basis for our approach to expressing content criteria and decomposition criteria.

A general approach to formally defining content criteria was proposed based on the idea that the information in model could be characterized in terms of the precise partiality relationship it has to a larger base model. Furthermore, the fact that a partiality relationship can provide the means for model decomposition allows us to use it as a general basis for defining decomposition criteria. These were both developed in detail for the special case of the submodel partiality relationship type. This case is of particular relevance in modeling practice because a model is typically presented as a set of interrelated diagrams that represent different submodels of the model.

The content criterion of a submodel (or diagram) is defined as the specification of the particular portion of the base model that is intended to be contained within the submodel. This is expressed at the role level as a view that includes a query-like transformation, called an extractor, that can extract the intended submodel from the base model. Extractors can be composed so that more complex extractors can be defined from simpler ones and they can be parameterized so that different submodels with similar intent can be defined by providing different parameter values.

The approach to defining decomposition criteria is based on specifying collections of views that decompose their base model. Such a specification is called a decomposer. Decomposers can be composed to define decomposition criteria for complex hierarchical decompositions. Of particular interest are “indexed decomposers” that use parameterized extractors to define a collection of views by using an “index model” to provide a set of parameter values. Indexed decomposers have natural properties that support model comprehension. In addition, they can be defined in a simple graphical way using macromodels.

10.2 Value and contributions

Throughout this thesis we have discussed four areas of value that explicit and formal expressions of modeler intent could provide: improving model *quality* by allowing modelers to confirm that the intent is correct and that the models conform to the intent; improving model *comprehension* by helping model consumers to understand the context and the basis for model content; supporting model *evolution* because expressions of intent can precede and guide the creation of model content and supporting *automation* by using formal expressions of intent to drive tools.

We can summarize our contributions to these areas of value as follows:

- Existential intent.
    - (quality, evolution) A model role represents the intent that a given model should exist in a project. When a model role exists without a model playing it we consider this to be a violation of an existential intent. Depending on the context this can either be seen as indicator of incompleteness (quality) or as a directive to modelers to create such a model (evolution).

- Content criteria.
  - (quality) We identified six types of model defects that require content criteria to detect: inclusion/exclusion defects on diagram content, inconsistency and inaccuracy defects on model names, and two kinds of potential defects on model completeness.
  - (comprehension) We also showed that content criteria is important because it reveals implicit information and context assumptions that are needed by stakeholders in order to interpret a model correctly.
  - (automation) Finally, we defined a systematic approach to defining formal content criteria that are amenable to tool support.
- Relationships. The definition and use of relationship types as first class entities is a contribution of this thesis.
  - (automation) We defined a generic and formal approach to defining relationship types based on Institution theory [GB92] that is applicable to any metamodeling language.
  - (comprehension) Relationship types represent a commonly recurring kind of intent about how models can be related. As such they are provide a meaningful level of abstraction over sets of constraints that can hold between models.
- Decomposition criteria.
  - (quality) We identified three types of defects can only be detected by using decomposition criteria. Diagram exclusion/inclusion defects are due to the violation of intent by the exclusion or unnecessary inclusion of

diagrams while the existence of intended, but empty, diagrams can indicate the incompleteness of the base model for the diagram.

  - (quality, evolution) In addition to defining decomposition criteria, decomposers also provide a mechanism for enforcing standards and best practices since they can be designed to yield desirable decomposition qualities such as non-redundancy, unity, etc. [M00].
  - (comprehension) Indexed decomposers have the desirable property that the index model acts as a kind of classification scheme over the base model by providing the "principle of division" by which it is decomposed. This provides a meaningful level of abstraction over hierarchical decompositions.
  - (quality, evolution) Indexed decomposers also enforce a consistent application of content criteria for different views within a decomposition.
  - (automation) Finally, since decomposition criteria are built using content criteria they enjoy the same level of formality in their definition.

- Macromodels. Macromodels represent two central contributions of this thesis: the identification of the role level as a key part of modeling and the definition of a modeling language for the role level by providing a unified means for expressing the different types of modeler intent in the intent framework.
  - (comprehension) Macromodels provide a level of abstraction on the collection of models in a project that is particularly appropriate for the task of understanding the purpose of models and how information is distributed over the models.

- (evolution) A macromodel also provides a means for top-down modeling by defining the information architecture of a set of models that guides and constrains team-based model development.
- (quality, automation) Several reasoning scenarios based on constraint satisfaction can be defined in terms of macromodel semantics including: conformance checking, synthesis, change propagation, etc.

10.3 Summary of the evaluation

In order to evaluate some of the claimed benefits of the techniques described in this thesis, we “reverse-engineered” the role level for two example modeling projects. The objectives here were twofold. First we could assess the claim about improved model quality by determining if defects were found. Second we could assess the claim about improved model comprehension by determining the quantity of implicit intent that is revealed, since as discussed in Chapter 8, this information could be shown to correlate with improved model comprehension.

The first example is a UML project used to define a telecommunication protocol for mobile telephone roaming called the Private User Mobility dynamic Registration service (PUMR). The second example is a MOF (Meta-Object Facility) metamodeling project that represents the portion of the UML 2.2 metamodel dealing with the UML concept of actions. In both cases, the macromodel including content criteria, relationships and decomposition criteria was specified and analyzed. Table 10.1 (reproduced from Table 8.9) summarizes the results. We take these results to support the claim that expressing the role level can improve model quality because in both examples a significant number

Table 10.1. Summary of evaluation results (reproduced from Table 8.9).

Finding	PUMR	UML
Defects or potential defects	28	28
Content criteria	42	19
Relationships	16	0
Indexed decompositions	6	4

of defects were found that could not have been identified without expressing the modeler intent. We also take the results to support the claim of improved comprehension because in both cases a substantial amount of intent-related information is added.

In order to assess the claim about formal intent supporting automation, a prototype was developed for solving macromodel conformance problems in order to test the viability of using a macromodel in various automation scenarios. The prototype translates the constraints expressed within a macromodel into a Kodkod [TJ07] model finding problem in order to automate the checking of a project for conformance to the modeler intent expressed in a macromodel. An interesting result here is that in some cases it is possible to check conformance even with incomplete models and mappings in the project. From our success in creating such a prototype we conclude that it is possible to use macromodels as a basis for automation; however, using a general model finder such as Kodkod to do it is impractical.

Finally, although the claim that expressing modeler intent supports model evolution via top-down modeling seems reasonable, we were not able to evaluate this as part of the thesis work. To do so would require a case study of an actual multi-person modeling project in progress in order to observe the effect that modeler intent has on the evolution of the models. Unfortunately, due to the complexities of such a study, we were unable to include it within the scope of the research in this thesis and we leave it to future work.

10.4 Future work

Here we discuss some promising future directions for this work.

10.4.1 Content and decomposition criteria for other partiality relationship types

In this thesis the notion of content criteria and decomposition criteria was detailed only for the submodel partiality relationship. Content criteria based on the submodel relationship alone assumes that all views of the base model are at the same level of abstraction as the base model. A more general scenario is that different views can represent different kinds and levels of abstraction. A decomposition into abstractions would be one where the base model is the greatest common refinement of the abstractions. As part of future work, content/decomposition criteria for other common partiality relationship types could be explored individually or in combination. One complexity here is that unlike the submodel type which can be defined in a generic way for any model types, abstraction and other partiality relationship types are typically model type specific.

10.4.2 Automatically inferring modeler intent

The claim in this thesis is that modeling the role level for a project is a fruitful thing to do since it brings value in several ways. However, since the expression of modeler intent places an additional burden of effort on the modeler, it may be neglected in the same way that other useful information such as software requirements and documentation are often neglected. To address this, we suggest that it is useful to consider techniques for automatically inferring intent from modeler actions. For example, when a modeler creates a set of diagrams, it may be possible to use machine learning methods such as inductive logic programming to make a guess at the content criteria of a the diagram by inferring the simplest query that would extract the diagram. Another possibility is to automatically enumerate the compositions of a fixed set of extractors to find the one that best approximates the content of a diagram. These guesses could then be refined with interaction from the modeler. Clearly there are many possibilities one could explore in this avenue.

10.4.3 A formalized taxonomy of abstract relationship types

In Chapter 4, we proposed a taxonomy of abstract relationship types. Of these, only the *detailOf*, *submodelOf* and *eq* relationship types were formally defined. There are several reasons why there is value in completing the formal characterization of the abstract relationship types in this taxonomy. First it would provide a well defined standardized vocabulary for discussing relationships between models within software engineering. Many terms such as “refinement” and “refactoring” representing relationships are in wide usage but have meanings that vary significantly across usages. Second, it would provide guidance to researchers who are defining formalizations of the particular concretized versions of these relationship types. This may also lead to the discovery of new constructable relationships types such as *submodelOf*. Finally, it would help identify errors and gaps in the taxonomy itself.

10.4.4 Making formal expressions of intent more understandable

Since expressions of modeler intent are used both for supporting model automation and model comprehension, they must satisfy the conflicting requirements of both being formal and also being easily understandable by non-technical stakeholders. Although an attempt was made to strike a balance between these two objectives, some of the approaches described in this thesis, such as for relationship types and content criteria, have tended more toward formal expressiveness rather than ease of understanding. Thus, further work must be done on finding ways to express formal role constraints in a way that can be readily understood by model consumers.

10.4.5 A case study for model evolution

As discussed above and in Chapter 8, the claim that modeler intent could be used to support model evolution was not assessed in this thesis. In order to do such an assessment a case study could be conducted in which a project macromodel is created and maintained throughout the lifetime of a modeling project. Such a study could be used to observe how effective top-down modeling is in helping to manage the development process by guiding the evolution of the models. Furthermore it could be used to evaluate the effectiveness of the different ways in which the role constraints could be used to control model content.

10.4.6 Additional evaluation studies

We note that while this thesis explores modeler intent in the context of software engineering, we expect our techniques to be readily applicable to other contexts in which graphical models are used. This includes areas such as business modeling, hardware design, etc. Studies must be done to evaluate the approach in these domains. In addition, more studies are needed even within software engineering in order to fully understand the limitations of our assumptions. For example, in Chapter 5 we discussed the fact that our approach to content criteria assumes that a global base model can exist and that this assumption may not be valid in contexts such as requirements gathering. Additional studies will help uncover other such issues and may point to ways to further generalize our approach.

References

[AGM08] Ardagna, D. and Ghezzi, C. and Mirandola, R.: Rethinking the use of models in software architecture. In: Proc. 4th International Conference on the Quality of Software Architectures, pp. 1--27, Springer (2008)

[AKK99] Astesiano, E., Kreowski, H. J., and Krieg-Bruckner, B.: Algebraic Foundations of Systems Specification. Springer-Verlag New York Inc. (1999)

[ANR06] Aizenbud-Reshef, N., Nolan, B.T., Rubin, J., Shaham-Gafni, Y.: Model Traceability. IBM Systems Journal, vol. 45, no. 3, pp. 515--526 (2006)

[ANTL] ANTLR parser generator website. July 2010, http://www.antlr.org

[B03] Bernstein, P.: Applying Model Management to Classical Meta Data Problems. In: Proc. of the Conference on Innovative Database Research, pp. 209--220 (2003)

[BCR05] Boronat, A., Carsí, J. A., Ramos, I.: An Algebraic Baseline for Automatic Transformations in MDA. Electronic Notes Theoretical Computer Science, vol. 127, no. 3, pp. 31--47 (2005)

[BCE06] Brunet, G., Chechik, M., Easterbrook, S., Nejati, S., Niu, N., Sabetzadeh., M.: A Manifesto for Model Merging. In: Proc. 1st Int. Workshop on Global Integrated Model Management (associated with ICSE'06), pp. 5--12 (2006)

[BJRV05] Bezivin, J., Jouault, F., Rosenthal, P., and Valduriez, P.: Modeling in the Large and Modeling in the Small. In: Aßmann U., Aksit, M., Rensink A. (eds.), Model Driven Architecture, LNCS vol. 3599, , pp. 33--46, Springer-Verlag GmbH (2005)

[CH06] Czarnecki, K. and Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal, vol. 45, no. 3, pp. 621--645 (2006)

[D05] Diskin, Z.: Mathematics of Generic Specifications for Model Management I and II. In Encyclopedia of Database Technologies and Applications. pp. 35--366, Idea Publishing Group (2005)

[DBV06] Del Fabro, M., Bezivin, J. and Valduriez, P.:Weaving models with the Eclipse AMW plugin. In Proc. of Eclipse Modeling Symposium, Eclipse Summit Europe (2006)

[DC] Dublin Core metadata standards initiative website. July 2010, http://dublincore.org/specifications/

[EFKN92] Easterbrook, S. A. Finkelstein, J. Kramer, and B. Nuseibeh, Coordinating distributed ViewPoints: the Anatomy of a Consistency Check. Concurrent Engineering: Research and Applications, vol. 2, no. 3, pp. 209--222 (1994)

[ET08] 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). July 2010, http://portal.etsi.org/mbs/Referenced%20Documents/eg_201_872.pdf

[EMF] Eclipse Modeling Framework website. July 2010, http://www.eclipse.org/modeling/emf

[F05] Favre, J. M.: Megamodelling and Etymology. In: Proc. of Dagstuhl Seminar 05161 on Transformation Techniques in Software Engineering (2005)

[G01] Gibbs, R. W.: Authorial intentions in text understanding. Discourse Processes, vol. 32, no. 1, pp. 73--80 (2001)

[G93] Giunchiglia, F.: Contextual reasoning. Epistemologia - Special Issue on "I Linguaggi e le Macchine", vol. 16, pp. 345--364 (1993)

[G98] Gurr, C.: On the isomorphism, or lack of it, of representations. In: Marriott, K. and Meyer, B. (eds.), Visual Language Theory, Springer Verlag, pp. 293--306 (1998)

[GB92] Goguen, J.A. and Burstall, R.M.: Institutions: Abstract Model Theory for Specification and Programming. Journal of the ACM, vol. 39, no. 1, pp. 95--146 (1992)

[GM92] Goguen, J.A. and Meseguer, J.: Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Theoretical Computer Science, vol. 105, no. 2, pp. 217--273 (1992)

[GMF] Eclipse Graphical Modeling Framework website. July 2010, http://www.eclipse.org/gmf/

[GR04] Guenther, R. and Radebaugh, J.: Understanding metadata. National Information Standard Organization (NISO) Press, Bethesda, MD (2004)

[GS04] Greenfield, J. and Short, K.: Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools. J. Wiley and Sons Ltd. (2004)

[HKR05] Huzar, Z., Kuzniarz, L., Reggio, G. and Sourrouille, J.L.: Consistency problems in uml-based software development. In: Jardim , N., Selic, B., Rodrigues da Silva, A., Alvarez A. T. (eds.), UML Modeling Languages and Applications, 2004 UML Satellite Activities, LNCS vol. 3297, pp 1--12. Springer (2005)

[HR04] Harel, D. and Rumpe, B.: Meaningful modeling: what's the semantics of" semantics"? IEEE Computer, vol. 37, no. 10, pp. 64--72 (2004)

[ISOC] ISO/IEC 11179-2:2005 Metadata registries (MDR) - Part 2: Classification. July 2010, http://standards.iso.org/ittf/PubliclyAvailableStandards/c035345_ISO_IEC_11179-2_2005(E).zip

[J00] Jones, C.: Software Assessments, Benchmarks, and Best Practices. Information Technology Series, Addison-Wesley, Boston, MA (2000)

[J09] Jackson, M.: Some Notes on Models and Modelling. In: Borgida, A.T. and Chaudhri, V.K. and Giorgini, P. (eds.), Conceptual Modeling: Foundations and Applications: Essays in Honor of John Mylopoulos, Springer, pp. 68--81 (2009)

[KH06] Karagiannis, D. and Höfferer, P.: Metamodels in Action: An Overview. In: J. Filipe, B. Shishkov and M. Helfert (eds.), First International Conference on Software and Data Technologies, pp. IS27--36, Insticc Press (2006)

[KLMM97] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin. J.: Aspect-oriented programming. In: Proc. of the European Conference on Object-Oriented Programming, pp. 220--242 (1997)

[KMS05] Kagdi, H., Maletic, JI and Sutton, A.: Context-free slicing of UML class models. In: Proc. of the 21st IEEE International Conference on Software Maintenance, pp. 635--638 (2005)

[K00] Kruchten, P.: The rational unified process: an introduction. Addison-Wesley Inc. Boston, MA (2000)

[KS03] Kalfoglou, Y. and Schorlemmer, M.: Ontology mapping: the state of the art. The Knowledge Engineering Review Journal, vol. 18, no. 1, pp. 1--31 (2003)

[KPP06] Kolovos, D.S., Paige, R.F. and Polack, F.A.C.: In: Eclipse Development Tools for Epsilon. Eclipse Summit Europe, Eclipse Modeling Symposium (2006)

[Lad97] Ladkin, P.: Abstraction and modeling, research report RVS-Occ-97-04, University of Bielefeld (1997)

[L00] Leveson, N.G.: Intent specifications: An approach to building human-centered specifications. IEEE Transactions on Software Engineering, vol. 26, no. 1, pp. 15--35, (2000)

[LC05] Lange, C.F.J. and Chaudron, M.R.V.: Managing Model Quality in UML-Based Software Development. In: Proc. 13th IEEE International Workshop on Software Technology and Engineering Practice, pp.7--16 (2005)

[LE03] Lohmann, D. and Ebert, J.: A Generalization of Hyperspace Approach Using Meta-Models. In: Araújo, J., Rashid, A., Tekinerdoğan, B., Moreira, A. and Clements P.

(eds.), Early Aspects 2003: Aspect-Oriented Requirements Engineering and Architecture, Boston, MA (2003)

[LMB01] Ledeczi A., Maroti M., Bakay A., Karsai G., Garrett J., Thomason IV C., Nordstrom G., Sprinkle J., Volgyesi P.: The Generic Modeling Environment. In: Proc. of the IEEE International Workshop on Intelligent Signal Processing, May (2001)

[LMT09] Lucas, F. J., Molina, F., Toval, A.: A systematic review of UML model consistency management. Information and Software Technology, vol. 51, no. 12, pp.1631--1645 (2009)

[LS06] Liu, Y.U. and Stoller S.D.: Querying Complex Graphs. In: Van Hentenryck, P. V. (ed.), Practical Aspects of Declarative Languages. LNCS vol. 3819, pp. 199--214 (2006)

[LSS94] Lindland, O., Sindre, G. and Solvberg, A.: Understanding quality in conceptual modeling. IEEE Software, vol. 11, no. 2, pp. 42--49 (1994)

[MDA] Model Driven Architecture (MDA) website. Object Management Group. July 2010, http://www.omg.org/mda/

[Mel04] Melnik, S.: Generic Model Management: Concepts and Algorithms. Springer-Verlag (2004)

[MMW02] Mens, K., Mens, T. and Wermelinger, M.: Maintaining software through intentional source-code views. In: Proc. of the 14th international conference on software engineering and knowledge engineering, pp. 289--296 (2002)

[M98] Moody, D. L.: Metrics for evaluating the quality of entity relationship models. In: T.W. Ling, S. Ram, M.L. Lee (eds.), Proc. of the 17th International Conference on Conceptual Modeling, LNCS vol. 1507, pp. 211--225 (1998)

[M00] Moody, D. L.: A Decomposition Method for Entity Relationship Models: A Systems Theoretic Approach. In: Altmann, G., Lamp, J., Love, P.E.D., Mandal, P., Smith, R., Warren, M. (eds), Proc. International Conference on Systems Thinking in Management, pp. 462--469 (2000)

[M05] Moody, D. L.:Theoretical and practical issues in evaluating the quality of conceptual models: current state and future directions. Data & Knowledge Engineering, vol. 55, no. 3, pp. 243--276 (2005)

[M06] Moody, D. L. : Dealing with “Map Shock”: A Systematic Approach for Managing Complexity in Requirements Modelling. In: Proc. of 12th International Workshop on Requirements Engineering Foundations for Software Quality, Luxembourg, (2006)

[M09] Moody, D. L. : The “Physics” of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. IEEE Transactions on Software Engineering, vol. 35, no. 6, pp. 756--779 (2009)

[MB02] Madhavan, J., Bernstein, P. A., Domingos, P., Halevy A. Y.: Representing and Reasoning about Mappings between Domain Models. In: Proceedings of the 18th National Conference on Artificial Intelligence, pp. 80--86 (2002)

[MEdit] MetaEdit+ website. July 2010, www.metacase.com

[MLPS97] Mikk, E., Lakhnech, Y., Petersohn, C. and Siegel, M.: On formal semantics of Statecharts as supported by STATEMATE. In Proc. of Second BCS-FACS Northern Formal Methods Workshop (1997)

[MOF06] Meta-Object Facility (MOF™) – Core specification V2.0. Object Management Group. July 2010, http://www.omg.org/technology/documents/formal/MOF_Core.htm.

[MBJK90] Mylopoulos, J., A. Borgida, M. Jarke, and M. Koubarakis, Telos: Representing Knowledge About Information Systems. ACM Transactions on Information Systems, vol. 8, no. 4, pp. 325--362 (1990)

[NCE02] Nentwich, C., Capra, L., Emmerich, W., Finkelstein, A.: xlinkit: a consistency checking and smart link generation service. ACM ACM Transactions on Internet Technology, vol. 2, no. 2, pp. 151--185 (2002)

[NKF94] Nuseibeh, B., Kramer J. and Finkelstein, A. : A Framework for Expressing the Relationships Between Multiple Views in Requirements Specifications. IEEE Transactions on Software Engineering, vol. 20, no. 10, pp. 760--773 (1994)

[NKF03] Nuseibeh, B., Kramer J. and Finkelstein, A.: Viewpoints: meaningful relationships are difficult. In: Proc. of 25th International Conference on Software Engineering, pp. 676--681 (2003)

[OCL] Object Constraint Language specification. Object Management Group. July 2010, http://www.omg.org/spec/OCL/2.0/

[P93] B. C. Pierce: Basic Category Theory for Computer Scientists. The MIT Press, Cambridge, Massachusetts (1993)

[QVT05] MOF™ Query / Views / Transformations (QVT) specification. Object Management Group. July 2010, http://www.omg.org/cgi-bin/doc?ptc/2005-11-01

[RJ01] Ramesh, B., Jarke, M.: Toward Reference Models for Requirements Traceability. IEEE Transactions on Software Engineering, vol. 27, no. 1, pp. 58--93 (2001)

[RST] Rhetorical Structure Theory website. July 2010, http://www.sfu.ca/rst/index.html

[S03] Seidewitz, E.: What Models Mean. IEEE Software, vol. 20, no. 5, pp. 26--32 (2003)

[S06] Schmidt, D.C.: Model-Driven Engineering. IEEE Computer, vol. 39, no. 2 (2006)

[SCE07] Salay, R., Chechik, M., Easterbrook, S.,Diskin, Z., McCormick, P., Nejati, S., Sabetzadeh, M. and Viriyakattiyaporn, P.: An Eclipse-Based Tool Framework for Software Model Management. In: Proc. of OOPSLA'07 Workshop on Eclipse Technology, pp. 55--59 (2007)

[SME08] Salay, R., Mylopoulos, J., Easterbrook, S. Managing Models through Macromodeling. In: Proc. IEEE/ACM International Conference on Automated Software Engineering 2008, pp. 447--450 (2008)

[SM09] Salay, R., Mylopoulos, J.: Improving Model Quality Using Diagram Coverage Criteria. In Proc. International Conference on Advanced Information Systems Engineering 2009, pp. 186--200 (2009)

[SME09] Salay, R., Mylopoulos, J., Easterbrook, S.M.: Using Macromodels to Manage Collections of Related Models. In Proc. International Conference on Advanced Information Systems Engineering 2009, pp.141--155 (2009)

[SE05] Sabetzadeh, M. and Easterbrook, S.: An Algebraic Framework for Merging Incomplete and Inconsistent Views. In: Proc. 13th IEEE Requirements Engineering Conference, Paris (2005)

[SEPa] "Artifact" in Stanford Encyclopedia of Philosophy. July 2010, http://plato.stanford.edu/entries/artifact/

[SEPb] "Pragmatics" in Stanford Encyclopedia of Philosophy. July 2010, http://plato.stanford.edu/entries/pragmatics/

[SCC06] Simonyi, C., Christerson, M. and Clifford, S.: Intentional software. ACM SIGPLAN Notices, vol. 41, no. 10 (2006)

[SNEC07] Sabetzadeh, M., Nejati, S., Easterbrook, S. and Chechik, M.: A Relationship-Driven Framework for Model Merging. In: Proc. of the Workshop on Modeling in Software Engineering (associated with ICSE 2007) (2007)

[SFT99] Spanoudakis, G., Finkelstein, A., Till, D.: Overlaps in Requirements Engineering. Automated Software Engineering Journal, vol. 6, pp. 171--198 (1999)

[SPEM] Software Process Engineering Metamodel V2.0 specification. Object Management Group. July 2010, http://www.omg.org/technology/documents/formal/spem.htm

[SSR06] Schauerhuber, A., Schwinger, W., Retschitzegger, W., Kapsammer, E. and Wimmer, M.: A Survey on Aspect-Oriented Modeling Approaches. Technical Report, Vienna University of Technology, Oct. (2007)

[SubV] Subversion website. July 2010, http://subversion.apache.org/

[T03] Tarlecki, A.: Abstract specification theory: an overview. In: Broy M. and Pizka, M. (eds.), Models, Algebras, and Logics of Engineering Software, vol. 191 of NATO Science Series, III: Computer and System Sciences, pages 43--79, IOS Press (2003)

[TOH99] Tarr, P., Ossher, H., Harrison, W. and Sutton Jr., S.M.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: Proc. of the International Conference on Software Engineering 1999, May (1999)

[TJ07] Torlak E. and Jackson, D.: Kodkod: A Relational Model Finder. Tools and Algorithms for Construction and Analysis of Systems. In: Proc. 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Braga, Portugal (2007)

[UML2] UML 2.2 Metamodel specification. Object Management Group. July 2010, http://www.omg.org/spec/UML/2.2/Superstructure/PDF/

[WW89] Wand, Y. and Weber, R. : A model of systems decomposition. In: DeGross, J. I., Henderson, J. C. and Konsynski, B. R. (eds.), Proc. of the 10th International Conference on Information Systems. ACM, New York, pp. 41--51 (1989)

[YM97] Yu, E.S.K. and Mylopoulos, J.: Modelling organizational issues for enterprise integration. In: Proc. International Conference On Enterprise Integration and Modelling Technology 1997. Turin, Italy, October (1997)

Appendix A

Supplementary information for the UML example

A.1 Extractors

The definitions in this section are based on the EMOF metamodel defined in Section C.1.

Generic extractors

These extractors are applicable to any EMOF model.

```
classesInPkg($m$:EMOF, $m1$:Set[Package]):Set[Class] := [
        // the classes of $m$ that occur in the packages of $m1$
        $precondition := m1 \subseteq m$,
        $Q_{Class}(c) := \exists p{:}m1.\text{Package} \cdot m.\text{package}(c, p)$
]

classSubsOf($m$:EMOF, $m1$:Set[Class]):Set[Class] := [
        // expand $m1$ to include all direct or indirect subclasses of $m1$
        $precondition := m1 \subseteq m$,
        $Q_{Class}(c) := (\exists c1{:}m1.\text{Class} \cdot \text{TC}(m.\text{superClass}(c, c1))$
]

expandClasses($m$:EMOF, $m1$:Set[Class]):MOF := [
        // expand classes in $m1$ to include their attributes, navigable associations
        // and superclasses
        $precondition := m1 \subseteq m$,
        $Q_{Property}(p) := \exists c{:}m1.\text{Class} \cdot m.\text{ownedAttribute}(c, p)$,
        $Q_{Class}(c) := (c \in m1.\text{Class}) \vee (\exists p{:}m.\text{Property} \cdot Q_{property}(p) \wedge m.\text{type}(p, c)\ )$
```

```
                              ∨ ($\exists c1$:$m1$.Class · $m$.superClass($c1$, $c$),
    $Q_{Datatype}(d)$ := $\exists p$:$m$.Property · $Q_{property}(p)$ ∧ $m$.type($p$, $d$),
    $Q_{EnumerationLiteral}(l)$ := $\exists e$:$m$.Enumeration · $Q_{datatype}(e)$ ∧ $m$.ownedLiteral($d$, $e$),
    $Q_{ownedAttribute}(c, p)$ := $TRUE$,
    $Q_{superClass}(c1, c2)$ := $TRUE$,
    $Q_{type}(p, t)$ := $TRUE$,
    $Q_{ownedLiteral}(d, l)$ := $TRUE$,
]

// gets the classes for the class types in m1 – this is the same as classSubsOf
classType($m$:EMOF, $m1$:Set[Class]):Set[Class] = classSubsOf($m$, $m1$)

partOf($m$:MOF, C1:Class, C2:Class):Boolean := [
        // determine if C1 is a part of C2 in m
        precondition := C1, C2 $\in$ $m$.class,
        partOf := $\exists p$:Property · $m$.ownedAttribute(C2, $p$) ∧
                    $m$.type($p$, C1) ∧
                    $m$.isComposite($p$, $TRUE$)
]
```

ActionMeta-specific extractors

These extractors are based on specific knowledge about ActionMeta.

```
supportingClasses($m$:EMOF, $ac$:Set[Class]):Set[Class] := [
    // gets the supporting classes for action classes in AC
    precondition := $ac \subseteq m$
    $Q_{Class}(c)$ :=
        ($\exists c1$: $ac$.Class · name($c1$, "Action") ∧
            $c \in$ classSubsOf($m$, {Pin}) ∧
        ($\exists c1$: $ac$.Class · name($c1$, "Link") ∧
                    c ∈ classSubsOf(m, {LinkEndData, QualifierValue}) ∧
        (∃c1: ac.Class · ¬(name(c1, "Action") ∨ name(c1, "Link") ) ∧
                    //  c must be a "base" part of some c1 ∈ ac
                    // i.e., c must be a part of c1 and not a part of any superclass of c1
                    partOf(c, c1) ∧
                    ¬∃c2:m.class · TC(superClass(c, c2)) ∧ partOf(c1, c2))
]
```

ActionClasses := subsOf(*m*, model(Set[Class], class = {Action}))

Index Sets

// Presentation *ActionTypes* and the coverage criteria for its constituents

// theModel is taken to be *ActionMeta*

```
ActionTypes = {
        action:Set[Class].cc = id(model(Set[Class], class = {Action, Opaque}))

        object:Set[Class].cc = classSubsOf(
                Class({CreateObjectAction,   DestroyObjectAction, TestIdentityAction,
                ReadSelfAction, ReadExtentAction, ReclassifyObjectAction,
                ReadIsClassifiedObjectAction, StartClassifierBehaviorAction})),

        link:Set[Class].cc = classSubsOf(
                Class({LinkAction, ClearAssociationAction, ReadLinkObjectEndAction,
                ReadLinkObjectEndQualifierAction})),

        acceptEvent:Set[Class].cc = classSubsOf(
                Class({AcceptEventAction, ReplyAction, UnmarshallAction })),

        invocation:Set[Class].cc = classSubsOf(
                Class({InvocationAction})),

        structuralFeature:Set[Class].cc = classSubsOf(m,
                Class({StructuralFeatureAction})),

        variable:Set[Class].cc = classSubsOf(m,
                Class({VariableAction})),

        raiseException:Set[Class].cc = classSubsOf(m,
                Class({RaiseExceptionAction})),

        misc:Set[Class].cc = classSubsOf(m,
                Class({ValueSpecificationAction})),

}
```

// Presentation *OperTypes* and the coverage criteria for its constituents

// theModel is taken to be *ActionMeta*

```
OperTypes =  {

                base:Set[Class].cc = id(
                        Class({StructuralFeatureAction, LinkAction, VariableAction })),

                read:Set[Class].cc = classSubsOf(
                        Class({TestIdentityAction, ReadSelfAction,
                        ReadExtentAction,ReadIsClassifiedObjectAction,
                ReadStructuralFeatureAction, ReadLinkAction,
                ReadLinkObjectEndAction, ReadLinkObjectEndQualifierAction,
                ReadVariableAction})),

        write:Set[Class].cc = classSubsOf(
                Class({CreateObjectAction, DestroyObjectAction,
                ReclassifyObjectAction, StartClassifierBehaviorAction,
                ClearAssociationAction, WriteLinkAction, WriteVariableAction,
                ClearVariableAction }))
}
```

A.2 List of defects for UML example

- Naming inconsistencies
    - The information about the package is incorporated into the diagram name in different ways at different places.
        - In the BasicActions package, the word "Basic" prefixes we name.
        - In the IntermediateActions package, the word "Intermediate" prefixes any name where the diagram has the same action type as in BasicActions
        - In the CompleteActions package, the string "(CompleteActions)" is used as a suffix on any diagram that has an action type the same as in IntermediateActions or BasicActions. However, even this scheme is used inconsistently since M17 uses the suffix but does not occur in either of the two packages.

        - In the StructuredActions package, the scheme is not followed: M20 has action type Action and this occurs in BasicActions.
    - The information about the action type is incorporated into the diagram name in different ways at different places.
        - The general scheme of the string "*actionType* actions" as part of the name is used consistently with the following exceptions:
            - M17 says "ReduceAction" instead of "Reduce actions"
            - In cases where the diagram associated with an action type is decomposed further some deviations occur because the action type is associated with the group and the diagrams within it are decomposed based on a different principle of division. For example, the action type "Action" is decomposed further on the basis of class type and so we get diagram M3 with the name "M3 – Basic pins" that gives no indication of the action type. This is a naming inconsistency due to an implicit group.
- Naming inaccuracies
    - Diagrams M8 and M14 both contain the string "Link identification" to indicate the type of entity operation however this is inconsistent with the manner in which the diagram is constructed – it is a detailing of the class LinkAction.
- Inclusion
    - Diagrams M3 and M10

- Weak modeling
  - EntOper (6 diagrams affected)
- Unmodeled information
  - ActionTypes: Object, Link, Misc and AcceptEvent (6 diagrams affected)
- Potential incompleteness (due to diagram structure constraints)
  - possibly missing diagrams due to empty indices
    - ActionType: Basic (8) (0 – real), Intermediate (5) (1 – real), Complete (6) (4 – real), Structured (7) (1 - real)
      - this may be misleading since we really only incrementally introduce new action types so we only expect to add to action types that occurred in earlier packages.
    - ClassType
      - The action type "Action" in BasicActions (2) (0 – real)
      - The action type "Action" in StructuredActions (3) (1 – real)
      - This may be misleading since not all class types are applicable to action type Action

Appendix B

Supplementary information for the PUMR example

The definitions in this section are based on the diagram type metamodels for UML2.2 defined in Appendix C. For additional elements that are not mentioned in these metamodels, please refer to the UML specification [UML2].

B.1 Generic constructed extractors

type(M:UML, S:One[String]):One[*type*] := [

    // extract the *type* element with name S

    precondition := $\exists x$:M.*type* · M.name(x) = s,

    $Q_{type}(k) := k = x$

    ]

*type*Set(M:UML, S:Set[String]):Set[*type*] := [

    // extract the *type* set model with names in S

    precondition := $\forall s$:S $\exists c$:M.*type* · M.name(c) = s,

    $Q_{type}(k)$:= $\exists s$:S · M.name(k) = s

    ]

Examples:

ClassSet(M:UML, S:Set[String]):Set[Class] := [

    // extract the Class set model with names in S

    precondition := $\forall s$:S $\exists c$:M.Class · M.name(c) = s,

```
        $Q_{Class}(k) := \exists s{:}S \cdot M.name(k) = s$
        ]

    Interaction(M:UML, S:One[String]):One[Interaction] := [
        // extract the Interaction with name S
        precondition := $\exists x{:}M.Interaction \cdot M.name(x) = s$,
        $Q_{Interaction}(k) := k = x$
    ]
```

B.2 Generic extractors

```
CDof(M:UML, K:Set[Class]):CD := [
    // Extracts the CD surrounding the classes in K
    precondition := submodelOf(K, M),
    $Q_{Association}(x) =$
    $Q_{Class}(x) = \exists k{:}K.Class \cdot (k = x) \vee$
                $\exists a{:}M.Association, e1, e2{:}M.Property \cdot$
                  $memberEnd(e1, a, 1) \wedge type(e1) = k \wedge$
                  $memberEnd(e2, a, 2) \wedge type(e2) = x \wedge$


SDof(M:UML, K:One[Interaction]):SD := [
    // Extracts the SD that contains the full content of interaction K
    precondition := submodelOf(K, M),
    $Q_{Interaction}(x) := \exists k : K.Interaction \cdot k = x$,
    $Q_{Lifeline}(x) := \exists k : K.Interaction \cdot k = M.interaction(x)$,
    $Q_{Message}(x) := \exists k : K.Interaction \cdot k = M.interaction(x)$,
    $Q_{InteractionFragment}(x) := \exists k : K.Interaction \cdot k = M.enclosingInteraction(x)$,
    $Q_{GeneralOrdering}(x) := \exists x1{:} M.InteractionFragment \cdot Q_{InteractionFragment}(x1) \wedge$
                                            $generalOrdering(x1, x)$,
    … and so on tracing elements to K for all sorts in SD
    ]

withStereotype(M:UML, M1:Set[String]):Set[Class] := [
    // extract classes from M with stereotypes in set M1
    precondition:= TRUE,
    $Q_{Class}(c)$ := $\exists s$:M1.String · M.name(stereotype($c$)) = $s$
    ]

UCDof(M:UML, M1:Set[Usecase]):UCD := [
    // Extracts the UCD that shows all the relationships that the usecases in M1 participate in
    precondition := submodelOf(M1, M),
    $Q_{Usecase}(u)$ := ($\exists x$:M.Extend · $Q_{Extend}(x)$ $\wedge$ $u$ = extendedCase($x$)) $\vee$
                ($\exists x$:M.Include · $Q_{Include}(x)$ $\wedge$ $u$ = includedCase($x$)) $\vee$
                ($\exists x$: $u1$:M1.Usecase · $u = u1$),
    $Q_{Extend}(x)$ := $\exists u1$:M1.Usecase · extension($x$, $u1$),
    $Q_{Include}(x)$ := $\exists u1$:M1.Usecase · addition($x$, $u1$),
    $Q_{Actor}(a)$ := $\exists u1$:M1.Usecase · ownedUseCase($u1$, $a$)
]

ADof(M:UML, K:One[Activity]): AD := [
    // Extracts the AD that contains the full content of activity K
    precondition := submodelOf(K, M),
    $Q_{Activity}(a)$ := $\exists a1$:K.Activity · $a = a1$,
    $Q_{ActivityNode}(a)$ := $\exists a1$:K.Activity · M.containedNode($a$, $a1$),
    $Q_{ActivityEdge}(e)$ := $\exists a1$:K.Activity · $e$ = M.containedEdge($e$, $a1$),
    $Q_{ActivityGroup}(g)$ := $\exists a1$:K.Activity · $g$ = M.subGroup($g$, $a1$),

    … and so on tracing elements to K for all sorts in AD

]

SMDof(M:UML, K:One[Statemachine]):SMD := [
    // Extracts the SMD that contains the full content of statemachine K
    precondition := submodelOf(K, M),
    $Q_{Statemachine}(x)$ := $\exists k$ : K.Statemachine · $k = x$,
    $Q_{Region}(x)$ := $\exists k$ : K.Statemachine · $k$ = M.statemachine($x$),
    $Q_{Vertex}(x)$ := $\exists k$ : K.Statemachine · $k$ = M.statemachine($x$),
    … and so on tracing elements to K for all sorts in SMD

]

SMDofState(M:UML, K:One[State]):SMD := [
    // Extracts the SMD that contains the full content of state K
    precondition := submodelOf(K, M),
    in($x$, $k$) := DEFINED(M.state(M.container($x$))) $\wedge$ $k$ = M.state(M.container($x$))
    $Q_{State}(x)$ := ($\exists k$ : K.State · $k = x$) $\vee$ TC(in($x$, $k$)),
    $Q_{Region}(x)$ := $\exists k$ : K.State · DEFINED(M.state($x$)) $\wedge$ $k$ = M.state($x$),

    … and so on tracing elements to K for all sorts in SMD
]

DirectPartsOf(M:UML, K:Set[Class]):CD := [
    // Extracts the CD consisting of the classes K and all their direct aggregated classes
    precondition:= submodelOf(K, M),
    $Q_{Association}(a)$ := $\exists c1$:K.Class, p1:M.Property ·
                    M.memberEnd($p1$, $a$, 1) $\wedge$ $c1$ = M.type($p1$) $\wedge$
                    $\neg$(M.aggregation($p1$) = none)
    $Q_{Class}(c)$ := ($\exists c1$:K.Class · ($c = c1$) $\vee$
                $\exists a$:M.Association, $p$:M.Property · $Q_{Association}(a)$ $\wedge$
                  M.memberEnd($p$, $a$, 2) $\wedge$ $c$ = M.type($p$)),
    $Q_{Property}(a)$ := $\exists c1$:M.Class · $Q_{Class}(c1)$ $\wedge$ ownedAttribute($a$, $c1$)
    ]

DirectSubsOf(M:UML, K:Set[Class]):CD := [
    // Extracts the CD consisting of the classes K and all their direct subclasses and
    // attributes
    precondition:= submodelOf(K, M),
    $Q_{Class}(c) := (\exists c1{:}\mathrm{K.Class} \cdot (c = c1) \vee \mathrm{M.general}(c, c1)),$
    $Q_{Property}(a) := \exists c1{:}\mathrm{M.Class} \cdot Q_{Class}(c1) \wedge \mathrm{ownedAttribute}(a, c1)$
    ]

DirectSupersOf(M:UML, K:Set[Class]):CD := [
    // Extracts the CD consisting of the classes K and all their direct superclasses and
    //  attributes
    precondition:= submodelOf(K, M),
    $Q_{Class}(c) := (\exists c1{:}\mathrm{K.Class} \cdot (c = c1) \vee \mathrm{M.general}(c1, c)),$
    $Q_{Property}(a) := \exists c1{:}\mathrm{M.Class} \cdot Q_{Class}(c1) \wedge \mathrm{ownedAttribute}(a, c1)$

]

TypeSpec(M:UML, K:Set[Class]):CD := [
    // Extracts the subclasses and aggregated classes of the classes in M1
    precondition:= submodelOf(K, M),
    $\mathrm{partOf}(c, c1) := \exists a{:}\mathrm{M.Association}, p, p1{:}\mathrm{M.Property} \cdot$
                            $\mathrm{M.memberEnd}(p, a, 2) \wedge c = \mathrm{M.type}(p) \wedge$
                            $\mathrm{M.memberEnd}(p1, a, 1) \wedge c1 = \mathrm{M.type}(p1) \wedge$
                            $\neg\mathrm{M.aggregation}(p1) = \mathrm{none},$
    $Q_{Class}(c) := (\exists c1{:}\mathrm{K.Class} \cdot \mathrm{TC}(\mathrm{general}(c, c1)) \vee \mathrm{TC}(\mathrm{partOf}(c, c1)))$
    $Q_{Property}(a) := TRUE$
    ]

ActivityForUsecase(M:UML, U:One[Usecase]): One[Activity] := [
    // Extracts the activity owned by the use case U
    precondition := $u \in m \wedge \exists a{:}m.\mathrm{Activity} \cdot m.\mathrm{context}(a, u)$

    $Q := a$
    ]

MessagesForTask(M:UML, T:One[String]):Set[Class] := [
    // Extracts the set of message classes that relate to task T
    ]

InteractionsForTask(M:UML, T:One[String]):Set[Interaction] := [
    // Extracts the set of interactions that relate to task T
]

UsecasesForTask(M:UML, T:One[String]): Set[Usecase] := [
    // Extracts the set of use cases that relate to task T
    …
    ]

StateMachineForClass(M:UML, K:One[Class]):One[Statemachine] := [
    // Extracts the statemachine owned by class K
    precondition:= submodelOf(K, M),
    $Q_{Statemachine}(x) := \exists k{:}K.\text{Class} \cdot x = \text{classifierBehavior}(k)$

]

SubsOf(M:UML, M1:Set[Class]):CD := [
    // expand m1 to include all direct or indirect subclasses of m1
    precondition:= $m1 \subseteq m$,
    $Q_{Class}(c) := (\exists c1{:}m1.\text{Class} \cdot \text{TC}(\text{superClass}(c, c1)),$
    $Q_{Property}(a) := TRUE$
    ]
```

B.3 Diagram content criteria

40 - Context model packages : PD

CC_{M40} := Proj(ContextModel, PD)

41 - Simple PUMR Domain Model : CD

CC_{M41} := DirectPartsOf(ContextModel, ClassSet(ContextModel, {“PISN”}))

42 – Sequence diagram indicating the flow of information between the user and the PISN : SD

CC_{M42} := SDof(ContextModel,
Interaction(ContextModel,
"Flow of information between the user and the PISN")

43 - PUMR system architecture : OD

CC_{M43} is unclear

44 - PUM Registration use case diagram: UCD

CC_{M44} := UCDof(ReqModel,
UsecasesForTask(ReqModel, "Registration"), UCD)

45 - Activity diagram describing the "Register PUM User at Terminal for Outgoing Calls" use case : AD

CC_{M45} := ADof(ReqModel, ActivityForUsecase(ReqModel,
Usecase(ReqModel, "Register PUM user at Terminal for Outgoing Calls")))

46 - Activity diagram describing the "Specify Access for Incoming Calls" use case : AD

CC_{M46} := ADof(ReqModel, ActivityForUsecase(ReqModel,
Usecase(ReqModel, "Specify Access for Incoming Calls")))

47 - Activity diagram describing the "Specify Profile" use case: AD

CC_{M47} := ADof(ReqModel, ActivityForUsecase(ReqModel,
Usecase(ReqModel, "Specify Profile")))

48 - PUM De-registration use case diagram: UCD

CC_{M48} := UCDof(ReqModel,
UsecasesForTask (ReqModel, "De-Registration"), UCD)

49 - Activity diagram describing the "De-register from current location" use case: AD

CC_{M49} := ADof(ReqModel, ActivityForUsecase(ReqModel,
Usecase(ReqModel, "Deregister from current location")))

50 - Specification model packages : PD

CC_{M50} is unclear – some submodel of Proj(SpecModel, PD)

51 - Basic Domain Model (from Context Model): CD

CC_{M51} := DirectPartsOf(SpecModel, ClassSet(SpecModel, {“PISN”}))

52 - PUMR Object Model : OD

CC_{M52} := objectsOf(SpecModel, M54) + objectsOf(SpecModel, M55)

53 - Example sequence diagram showing registration using the PUM Number: SD

CC_{M53} := SDof(SpecModel,
Interaction(SpecModel, “Registration using the PUM Number”))

54 - Example sequence diagram showing registration using Alternative Identifier: SD

CC_{M54} := SDof(SpecModel,
Interaction(SpecModel, “Registration using Alternative Identifier”))

55 - Example sequence diagram showing de-registration: SD

CC_{M55} := SDof(SpecModel, Interaction(SpecModel, “De-Registration”))

56 - Example sequence diagram showing PUMR interrogation: SD

CC_{M56} := SDof(SpecModel, Interaction(SpecModel, “PUMR Interrogation”))

57- PUMR Detailed Domain Model : CD

CC_{M57} := [
// Expand class PINX to containing classes, their attributes
and methods, their interfaces and attributes and methods

precondition := ∃*c*:SpecModel.Class · SpecModel.name(*c*) = “PINX”,

partOf(*c*, *c1*) := ∃*a*: SpecModel.Association, p, p1: SpecModel.Property ·
SpecModel.memberEnd(*p*, *a*, 2) ∧ *c* = SpecModel.type(*p*) ∧
SpecModel.memberEnd(*p1*, *a*, 1) ∧ *c1* = SpecModel.type(*p1*) ∧

$\neg$SpecModel.aggregation($p1$) = none,

$Q_{class}(x) := (x = c) \vee$ partOf(x, c),

$Q_{Interface}(x) := \exists r$:InterfaceRealization ·

SpecModel.implementingClassifier(r) = $c \wedge$

SpecModel.contract(r) = x

$Q_{attribute}(x) := (\exists c1$:SpecModel.Class · ownedAttribute(x, $c1$) $\wedge$ $Q_{class}(c1)) \vee$

($\exists c1$:SpecModel.Interface · ownedAttribute(x, $c1$) $\wedge$ $Q_{Interface}(c1)$),

$Q_{operation}(x) := (\exists c1$:SpecModel.Class · ownedOperation(x, $c1$) $\wedge$ $Q_{class}(c1)) \vee$

($\exists c1$:SpecModel.Interface · ownedOperation(x, $c1$) $\wedge$ $Q_{Interface}(c1)$),

]

RegProc : SMD

$CC_{RegProc}$:= SMDof(SpecModel,

StateMachineForClass(SpecModel, Class(SpecModel, “Home PINX”)))

58 - Statechart diagram showing the registration processing at the Home PINX: SMD

CC_{M58} := RegProc – M59

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

CC_{M59} := SMDofState(RegProc, State(SpecModel, “RegistrationRequest”))

60 - PUMR message-specific packages : PD

CC_{M60} := [

precondition := *TRUE*,

$Q_{Package}(x)$ = … // unmodeled condition for PUMR message-specific packages

]

61 - Identification of PUMR signaling at QSIG interfaces : CD

CC_{M61} is unclear

62 - Identification of the two QSIG signals used for carrying PUMR message info : CD

// communication message classes of QSIG that are extended by PUMR classes

CC_{M62} := [

precondition := $\exists$*qsig*, *pumr*:SpecModel.Package ·

SpecModel.name(*qsig*) = “QSIG” $\wedge$

SpecModel.name(*pumr*) = “PUMR”

$Q_{Class}(c)$:= $\exists$*c1*: SpecModel.Class, *p, p1*: SpecModel.Package ·

TC(SpecModel.ownedMember(*p*, *qsig*)) $\wedge$

SpecModel.ownedMember(*c*, *p*) $\wedge$

TC(SpecModel.ownedMember(p1, *pumr*)) $\wedge$

SpecModel.ownedMember(*c1*, *p1*) $\wedge$

TC(SpecModel.general(*c1*, *c*)) $\wedge$

SpecModel.name(stereotype(*c*)) = “communication message”)

]

Messages1 : CD

$CC_{Messages1}$:= [

precondition := *TRUE,*

$Q_{Class}(c)$:= SpecModel.name(stereotype(*c*)) = “communication message”,

$Q_{Attribute}(x)$:= $\exists$*c*:SpecModel.Class · ownedAttribute(*x*, *c*) $\wedge$ $Q_{Class}(c)$

]

63 - PUMR message contents carried in the SETUP signal : CD

CC_{M63} := DirectSubsOf(SpecModel, ClassSet(SpecModel, {“PUM_SETUP”})) +
DirectSupersOf(SpecModel, ClassSet(SpecModel, {“PUM_SETUP”}))

64 - PUMR message types carried in the CONNECT signal : CD

CC_{M64} := DirectSubsOf(SpecModel, ClassSet(SpecModel, {“PUM_CONNECT”})) +
DirectSupersOf(SpecModel, ClassSet(SpecModel, {“PUM_SETUP”}))

65 - Contents of PUMR response messages : CD

CC_{M65} := DirectSubsOf(SpecModel, ClassSet(SpecModel, {“Response”}))

66 - Contents of PUMR error messages : CD

CC_{M66} := DirectSubsOf(SpecModel, ClassSet(SpecModel, {“Errors”}))

67 - PUMR registration message types : CD

CC_{M67} := MessagesForTask(SpecModel, “Registration”)

68 - PUMR de-registration message types : CD

CC_{M68} := MessagesForTask(SpecModel, “De-Registration”)

69 - PUMR delete registration message types : CD

CC_{M69} := MessagesForTask(SpecModel, “Delete”)

70 - PUMR interrogation message types : CD

CC_{M70} := MessagesForTask(SpecModel, “Interrogation”)

71 - PISN enquiry message types : CD

CC_{M71} := MessagesForTask(SpecModel, “Enquiry”)

72 - PUMR general data types : CD

CC_{M72} := TypeSpec(SpecModel,
ClassSet(SpecModel, {“ServiceOption”, “SessionParams”, “DummyRes”}))

ErrorCodes : CD

// all enums that in a PUMR package that has name ending with “Errors”

$CC_{ErrorCodes}$:= [

precondition := ∃*pumr*:SpecModel.Package · SpecModel.name(*pumr*) = “PUMR” ,

Q_{Class}(*c*) := ∃*p*: SpecModel.Package, *s1*, *s2*: SpecModel.String ·

TC(SpecModel.ownedMember(*p*, *pumr*)) ∧

SpecModel. ownedMember(*c*, *p*) ∧

SpecModel.name(stereotype(*c*)) = “enumeration” ∧

SpecModel.name(*c*) = *s1*) ∧

s1 = *s2* + “Errors”,

$Q_{Property}(p)$:= *TRUE*
]

73 - PUMR error codes : CD

// all enums that in a PUMR package that has name ending with "Errors"

CC_{M73} := TypeSpec(SpecModel, ErrorCodes)

74 - Type specification of PUM user PIN : CD

CC_{M74} := TypeSpec(SpecModel, ClassSet(SpecModel, {"PUM user PIN"}))

75 - Type specification of PUM user identifier : CD

CC_{M75} := TypeSpec(SpecModel,
ClassSet(SpecModel, {"PUM user identifier"}))

76 - Type specification of PUMR message extension : CD

CC_{M76} := TypeSpec(SpecModel,
ClassSet(SpecModel, {"PUMR message extension"}))

77 - QSIG message packages not specific to PUMR : PD

```
CC_M77 := [
   precondition := TRUE,
   Q_Package(x) = … // unmodeled condition for QSIG message-specific packages
]
```

78 - QSIG basic service messages : CD

// communication messages in QSIG package

```
CC_M78 := [
   precondition := ∃qsig:SpecModel.Package · SpecModel.name(qsig) = "QSIG",
   Q_Class(c) := ∃p: SpecModel.Package · TC(SpecModel.ownedMember(p, qsig)) ∧
                  SpecModel.ownedMember(c, p) ∧
                  SpecModel.name(stereotype(c)) = "communication message",
   Q_Property(p) := TRUE
   ]
```

79 - QSIG general data types : CD

CC_{M79} := TypeSpec(SpecModel,
ClassSet(SpecModel, {“Basic Service”, “CharString20”}))

80 - Type specification of QSIG party number : CD

CC_{M80} := TypeSpec(SpecModel,
ClassSet(SpecModel, {“QSIG party number”}))

81 - Type specification of QSIG digit string : CD

CC_{M81} := TypeSpec(SpecModel, ClassSet(SpecModel, {“QSIG digit string”}))

B.4 Relationship type definitions used in PUMR

instanceOf(OD, CD)

```
instanceOf(Inst:OD,Type:CD) =   Inst.OD + Type.CD +
    subsort  Inst.Class ≤ Type.Class
             Inst.Association ≤ Type.Association
    constraints
      // all objects (ConnectableElement) and links (Connector) in Inst must have types
        ∀x:Inst.Connector · DEFINED(Inst.type(x))
        ∀x:Inst.ConnectorElement · DEFINED(Inst.type(x))

      // link (connector) endpoints conform to association endpoint types  at the same index
        ∀x:Inst.Connector ∃a:Inst.Association ·
          a = Inst.type(x) ∧
          ∀e:Inst.ConnectorEnd, i :INT ·  Inst.end(x, e, i) ⇒
             ∃p:Type.Property · Type.memberEnd(a, p, i) ∧
                Inst.type(Inst.role(e)) = Type.type(p)
      // minimality: Type only contains Classes and Assocs corresponding to Inst
        ∀x:Type.Class ∃y:Inst.ConnectableElement · x = Inst.type(y)
        ∀x:Type.Association ∃y:Inst.Connector · x = Inst.type(y)
```

objectsOf(SD, OD)

Note that this version of *objectsOf* introduced in Chapter 4 is adapted to the UML 2.2 abstract syntax. The *sentOver* function is replaced by the *connector* function which exists with the UML 2.2. metamodel.

```
objectsOf(theSD:SD,theOD:OD) =   theSD.SD + theOD.OD +
    subsort  theSD.ConnectableElement ≤ theOD.ConnectableElement
    func connector: theSD.Message → theOD.Connector
    constraints
      // all Lifelines in theSD must have an ConnectableElement
        ∀x:theSD.Lifeline · DEFINED(theSD.represents(x))
      // connector preserves endpoint incidence
        ∀x:theSD.Message ∃source, target:theOD.ConnectorEnd ·
          theOD.end(source, connector(x), 1)  ∧
          theSD.represents(theSD.covered(theSD.sendEvent(x))) = theOD.role(source) ∧
          theOD.end(target, connector(x), 2)  ∧
          theSD.represents(theSD.covered(theSD.eceiveEvent(x))) = theSD.role(target)

      // minimality: theOD only contains Connectors and ConnectableElements corresponding
        to theSD
        ∀x:theOD.ConnectableElement ∃y:theSD.Lifeline · x = theSD.represents(y)
        ∀x:theOD.Connector ∃y:theSD.Message· x = connector(y)
```

caseOf(SD, SD)

Here, the skeleton of the relationship type is defined but the details of the constraints (i.e., “additional constraints”) that define this as an SD specialization relationship are not developed as this is beyond the scope of this thesis.

```
caseOf(Spec:SD,Gen:SD) =   Spec.SD + Gen.SD +
      func  map: Spec.Lifeline → Gen.Lifeline
             map: Spec.Message → Gen.Message
      constraints
         // map preserves endpoint incidence
            ∀x: Spec.Message ·
             Gen.covered(Gen.sendEvent(map(x))) =
                map(Spec.covered(Spec.sendEvent(x))) ∧
             Gen.covered(Gen.receiveEvent(map(x))) =
                map(Spec.covered(Spec.receiveEvent(x))) ∧
         // minimality: Gen only contains Lifelines and Messages corresponding to Spec
            ∀x: Gen.Lifeline ∃y:Spec.Lifeline · x = map(y)
            ∀x: Gen.Message ∃y:Spec.Message · x = map(y)
         // other constraints that define an SD specialization
            additional constraints
```

aggregationOf(OD, OD)

```
instanceOf(Agg:OD, Det:OD) =   Agg.OD + Det.OD + OD +
      sort Property, AggregationKind
      func none: AggregationKind, composite: AggregationKind, shared: AggregationKind
      pred aggregates:ConnectableElement × ConnectableElement
            aggregation:Property → AggregationKind
      constraints
         // Agg and Det are both submodels of OD
         submodelOf(Agg.OD, OD)
         submodelOf(Det.OD, OD)
```

```
// helper predicate ‘aggregates(x, y)’ says that x has an aggregation link to y
∀x, y: ConnectableElement · aggregates(x, y) ⇔
  ∃z: Connector, ex, ey:ConnectorEnd, w:Property ·
    x = role(ex) ∧ y = role(ey) ∧
    end(ex, z, 1) ∧ end(ey, z, 2) ∧
    memberEnd(w, type(z), 1) ∧
    ¬(aggregation(w) = none)
// there must be a chain of aggregation Connectors from each ConnectableElement
// in Det and one in Agg
  ∀x:Agg.ConnectableElement ∃y:Det.ConnectableElement · TC(aggregates(x, y))
```

B.5 List of defects for the PUMR example

Naming inconsistencies

- “41 - Simple PUMR Domain Model” and “51- Basic domain model (from Context Model)” have the same CC but are named differently
- “43 - PUMR system architecture” and “52 - PUMR Object Model” have the same CC but are named differently
- There are two groups of diagrams that show the details for various data types. When the diagram shows this for a single type *T* it is named “Type specification of *T*” however when it shows it for a group types there is no fixed convention: “73 – PUMR error codes”,“72 - PUMR general data types” and “79 - QSIG general data types.” Since these all represent the same intent (TypeSpecification) with different generators we might expect a more standardized form, e.g.,: “73 – Type specification of PUMR error codes”,“72 – Type specification of PUMR general data types” and “79 – Type specification of QSIG general data types.”

Naming inaccuracies

- "50 - Specification model packages" suggests the CC Proj(SpecModel, PD) as is the case with "40 - Context model packages" but it is actually a subset of the packages
- "58 – Statechart diagram showing the registration processing at the Home PINX" only contains part of the state machine diagram for registration processing. This is actually decomposed over diagrams 58 and 59. However, because there is no means for referring to the unrealized combined diagram that would correspond to the statemachine element, it is attached to the "top" model in a (detailOf) hierarchical decomposition.
- "62 - Identification of the two QSIG signals used for carrying PUMR message info"
    - Furthermore, I have generalized this to "the QSIG signals …" but maybe this is my error. The alternative is incompleteness.

Content exclusions

- 66 is missing PumInterrogErr
- 65 is missing PumDelRegRes
    -

Weakly modeled information

- partial weak modeling of the concept of a "normative interface" since normative interfaces are identified either by a stereotype or by a comment in diagram 61

- in diagram 73, a class is identified as being an "error code" by using the naming convention that it has "Errors" as a suffix.
- for diagrams 67-71, to identify the task associated with a message class using, *MessagesForTask* is implemented using a naming convention that the class name has prefix "PumRegistr", "PumDereg", "PumInterrog", "PumDelReg" and "PisnEnq" for tasks "Registration", "Deregistration", Interrogation", "DeleteRegistration" and "Enquiry", respectively.
- For diagrams 63-66, in order to identify whether a message class is an argument, response or error, the naming convention is used that the class name has the suffix "Arg", "Res" or "Err", respectively. However, in this case it is not necessary to rely on this informal source of information since the type can also be identified by base classes "PUM_SETUP", "Response" and "Errors." Thus this is a case of unnecessary redundancy of information.

Unmodeled information

- UsecasesForTask (44, 48)
- InteractionForTask
- in diagram 60, it is not evident how to determine whether a PUMR package is message specific
- in diagram 50, it is not evident how the subset of specification model packages in this diagram is determined
- in diagram 77, it is not evident how to determine which QSIG message packages are not specific to PUMR

Potential incompleteness

- There is no architecture corresponding to 56
    - this is an incompleteness defect
- There are no interactions corresponding to tasks DeleteRegistration and Enquiry
- There are is no use case diagram or activity diagrams corresponding to tasks Interrogation, DeleteRegistration or Enquiry
- There are four use cases in 44 and only three activity diagrams are shown

  this is an example of incompleteness detection but also an exclusion defect in the Registration : ReqUnitT group.

Appendix C

Metamodels

The metamodels in this section are all expressed graphically as EMOF models [MOF06]. These are excerpts from specifications and are intended for use in the definitions of the extractors and relationship types in the examples. Note that not all the constraints defined in the specifications are shown here for these metamodels. We assume these metamodels are to FO+ as follows with examples shown from the EMOF metamodel (Section C.1):

- Each element class becomes a sort and the subclass relationship between element classes becomes a subsort relationship
  - e.g., we have sorts: Enumeration, Datatype, EnumerationLiteral, etc. with Enumeration $\leq$ Datatype
- Each association end with a multiplicity of 1 becomes a function and with multiplicity 0..1, a partial function, having one argument, named by the role name of the end and with result type given by the end class.
  - e.g., we have partial functions:
    - class:Property o$\rightarrow$ Class
    - package:Type o$\rightarrow$ Package
- Each association end with a multiplicity of *k*..* becomes a binary predicate named by the role name of the end and with end are the second argument. In the case of a multiplicity of *k*..* with $k > 0$, a corresponding constraint is added. If the end is ordered, then the predicate has an additional integer argument with the order index.
  - e.g., we have predicates:
    - superClass:Class $\times$ Class (with specific class first)
    - ownedAttribute:Class $\times$ Property $\times$ INT
    - annotatedElement:Comment $\times$ NamedElement
- Each attribute becomes a function with the element class as the first argument.

- e.g., we have functions:
  - name:NamedElement → String
  - isReadOnly:Property → Boolean

C.1 EMOF

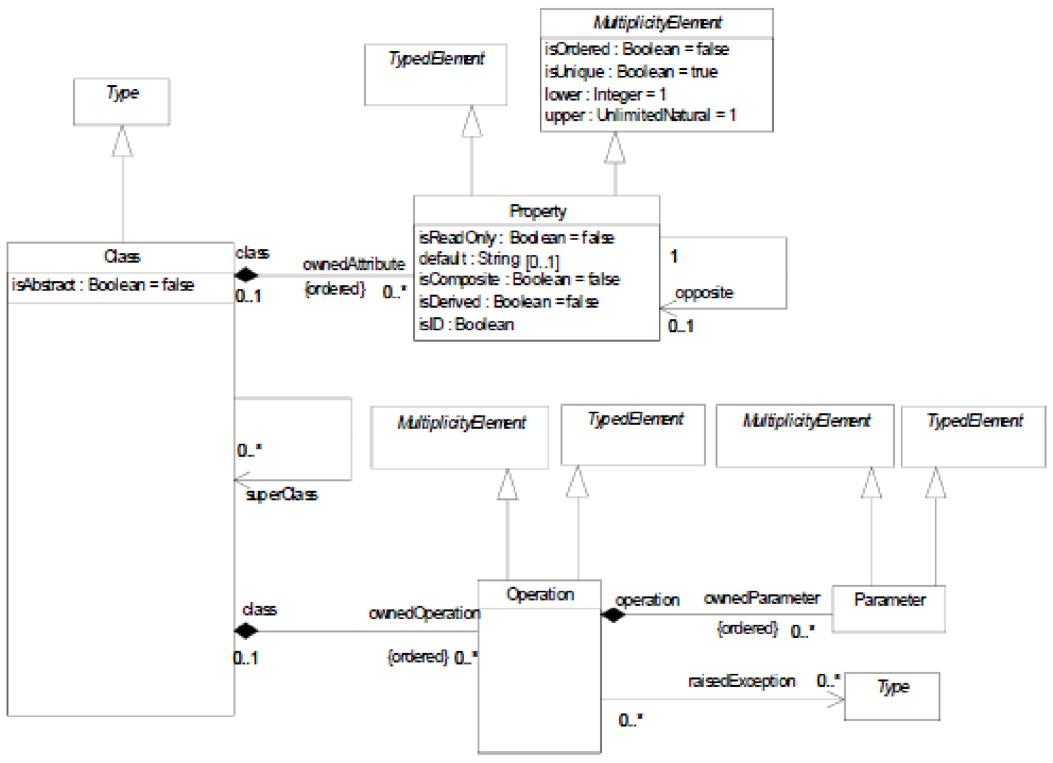


Figure 12.2 - EMOF Classes

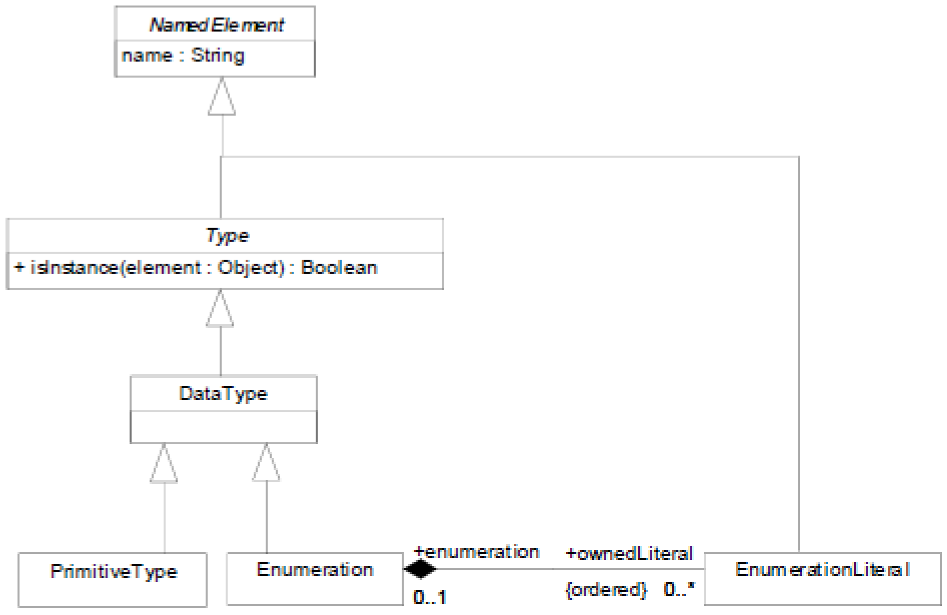


Figure 12.3 - EMOF Data Types

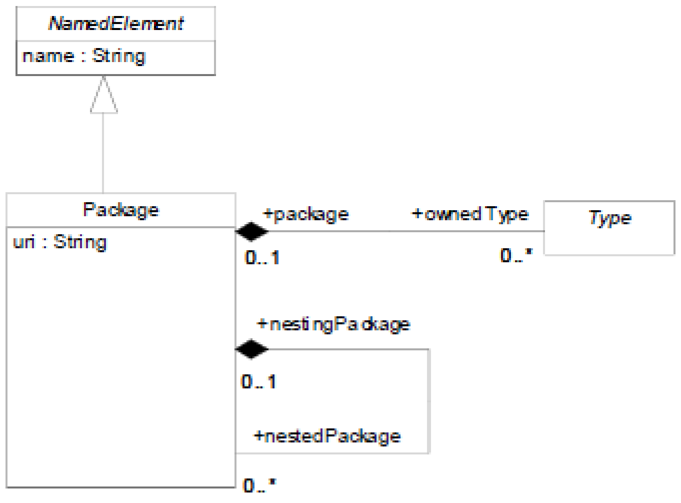


Figure 12.4 - EMOF Packages

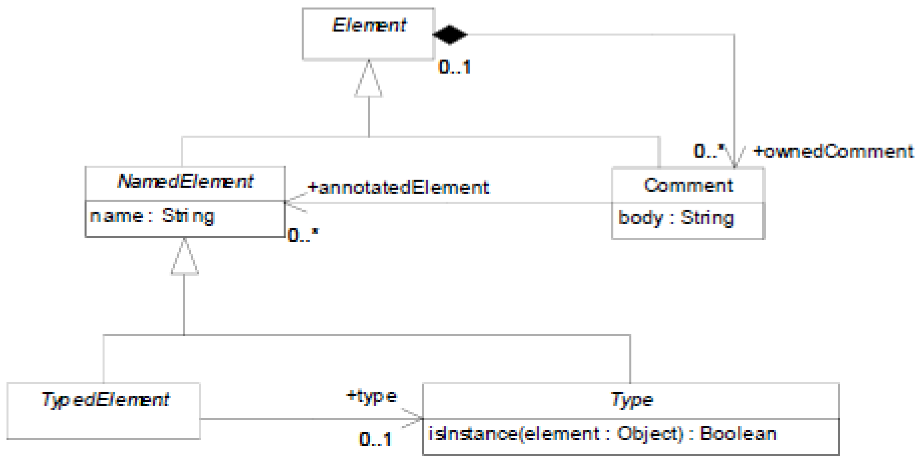
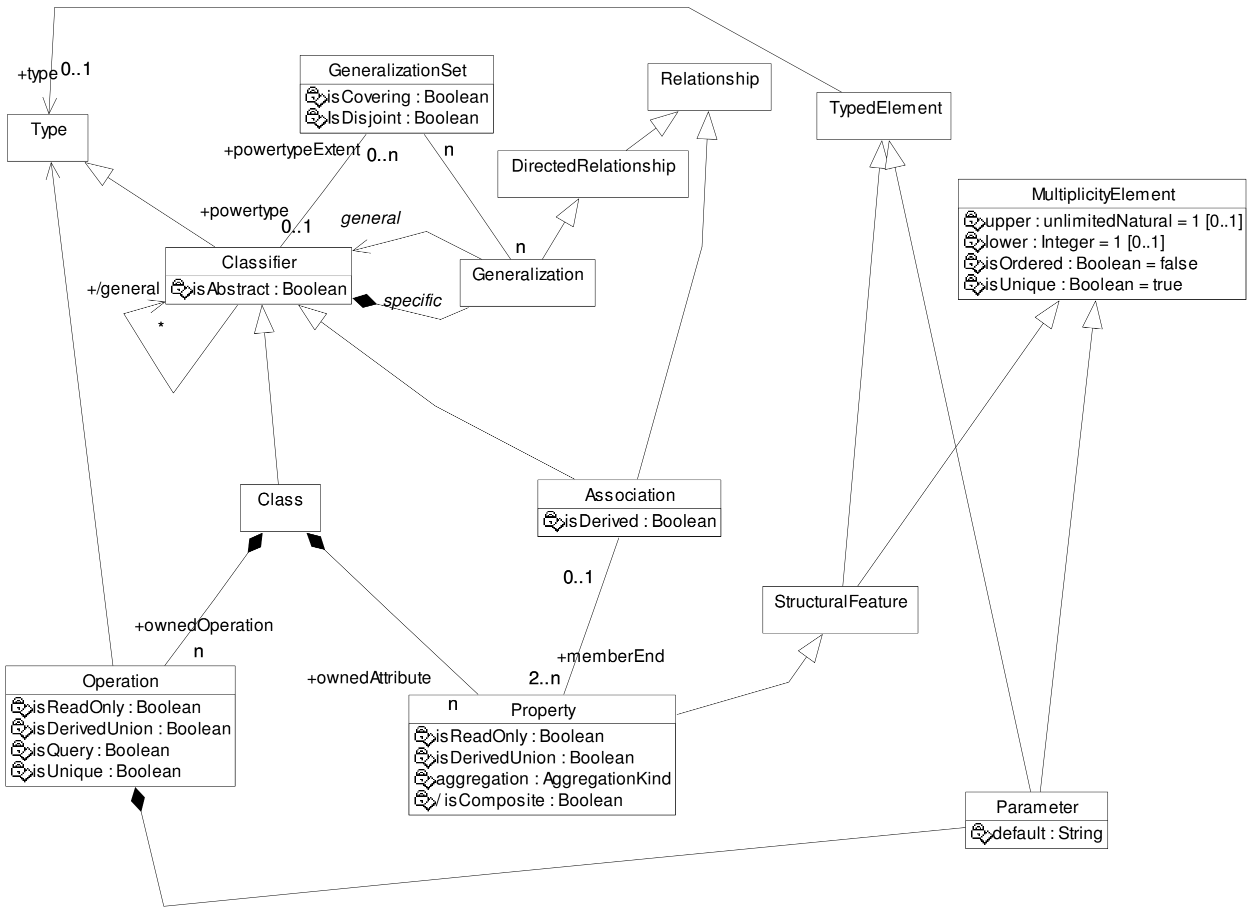
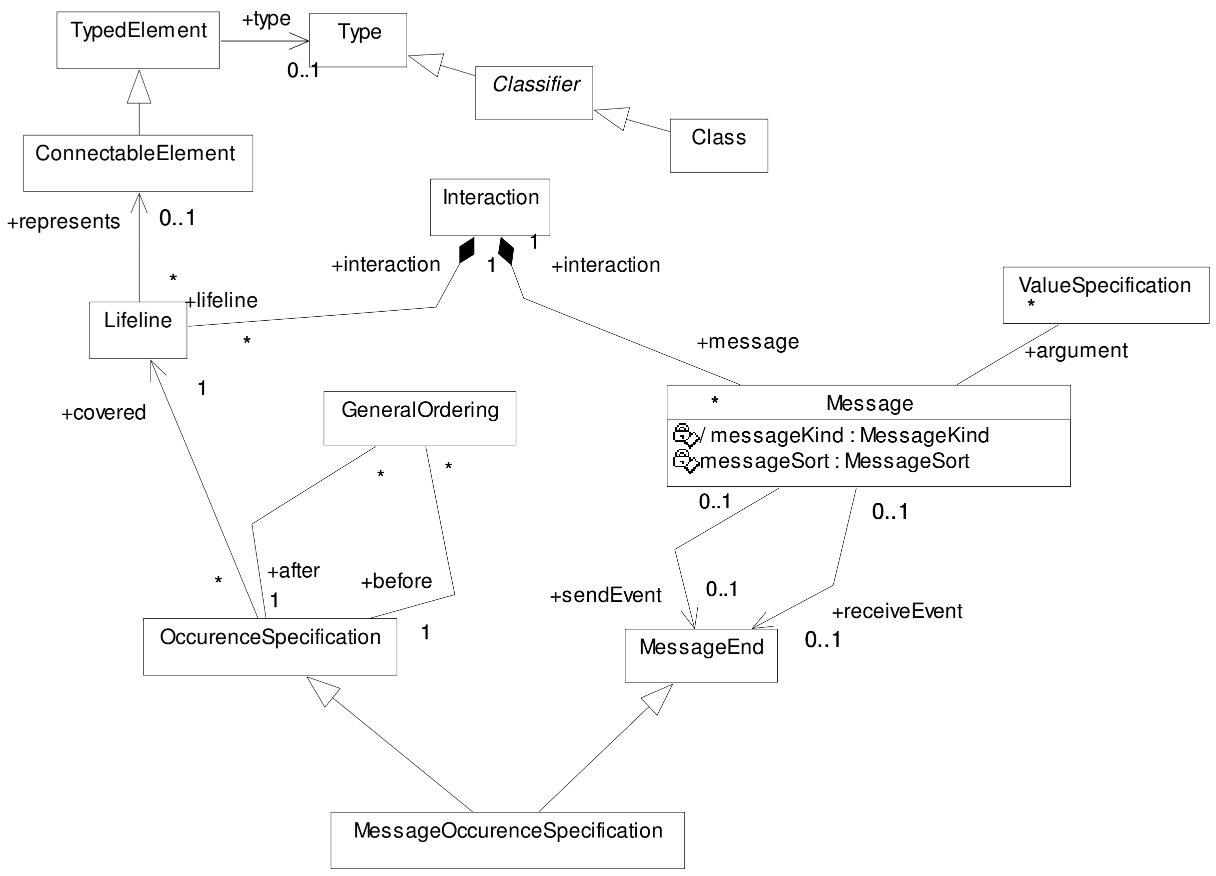


Figure 12.5 - EMOF Types

C.2 CD (ClassDiagram)

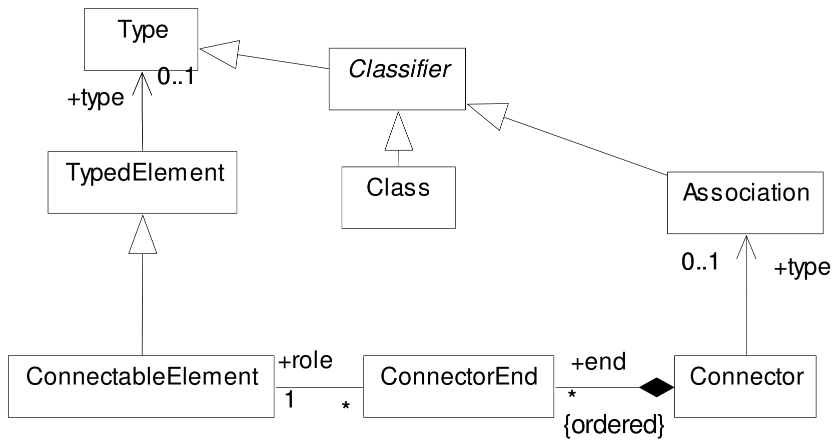


C.3 SD (Sequence Diagram)

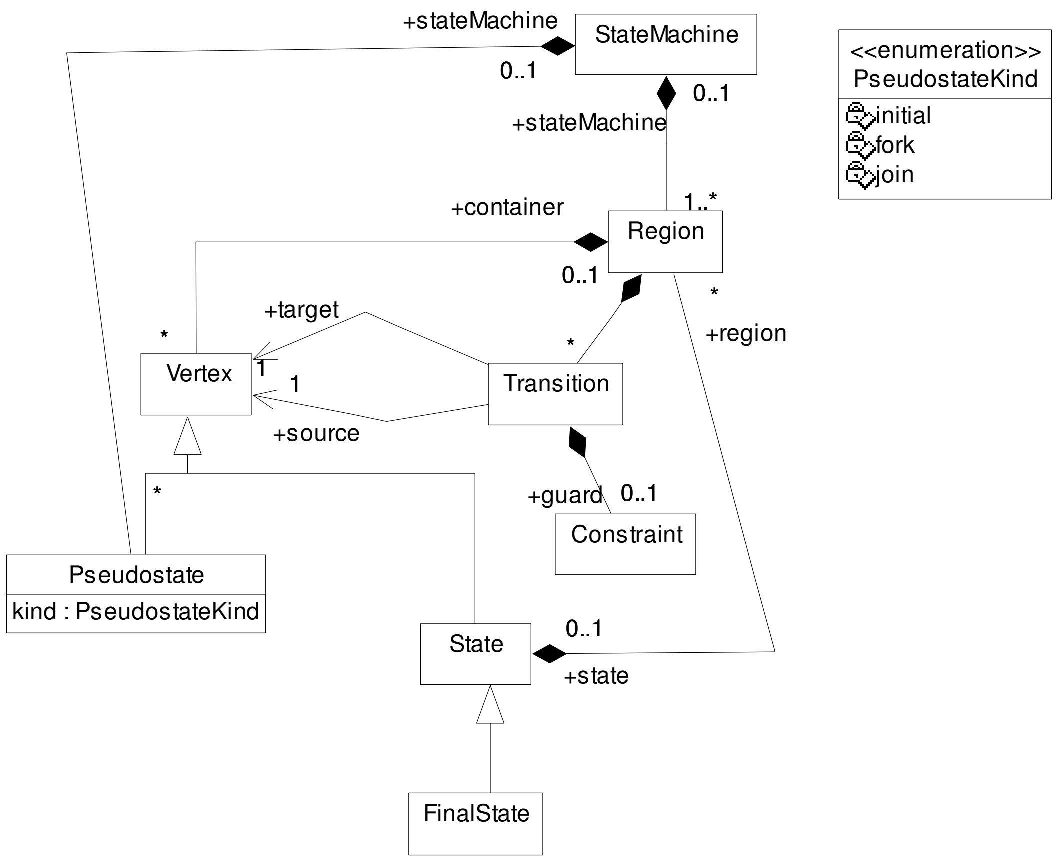


C.4 OD (Object Diagram)

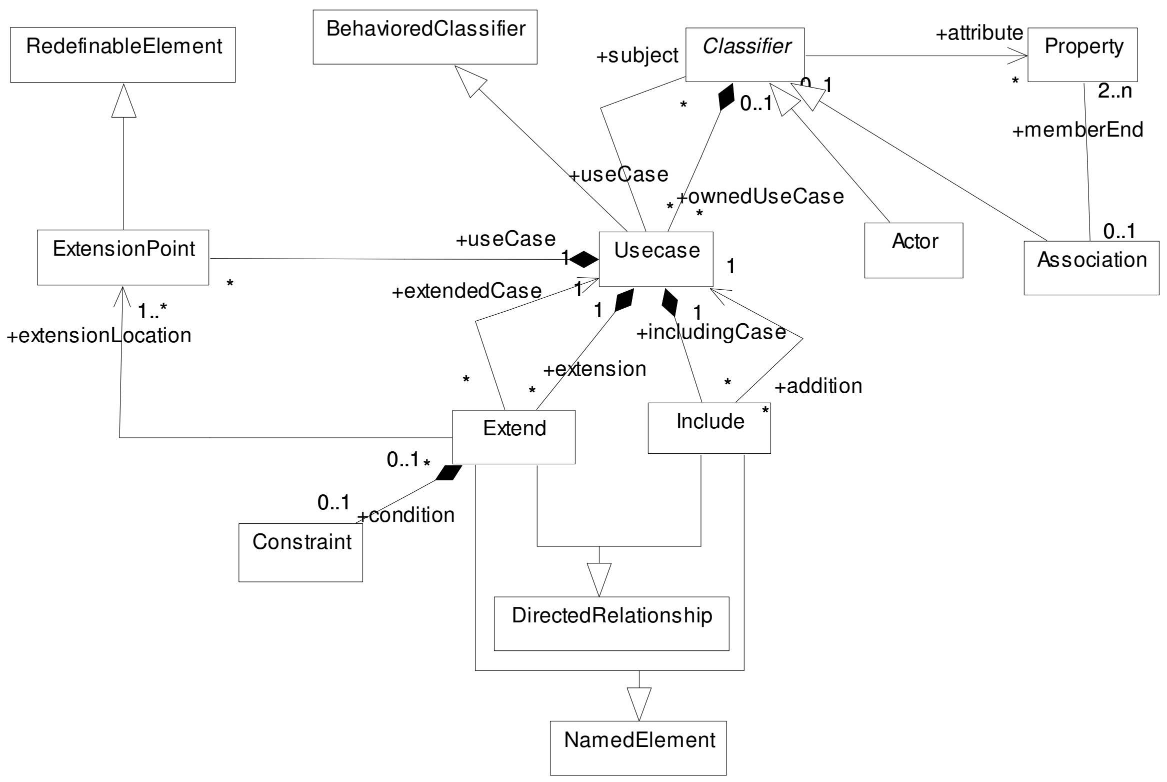
Note: we define OD using UML 2.2 ConnectableElements and Connectors rather than InstanceSpecifications since this better fits the intended us in PUMR - they used UML 1.4 which didn't have these concepts.



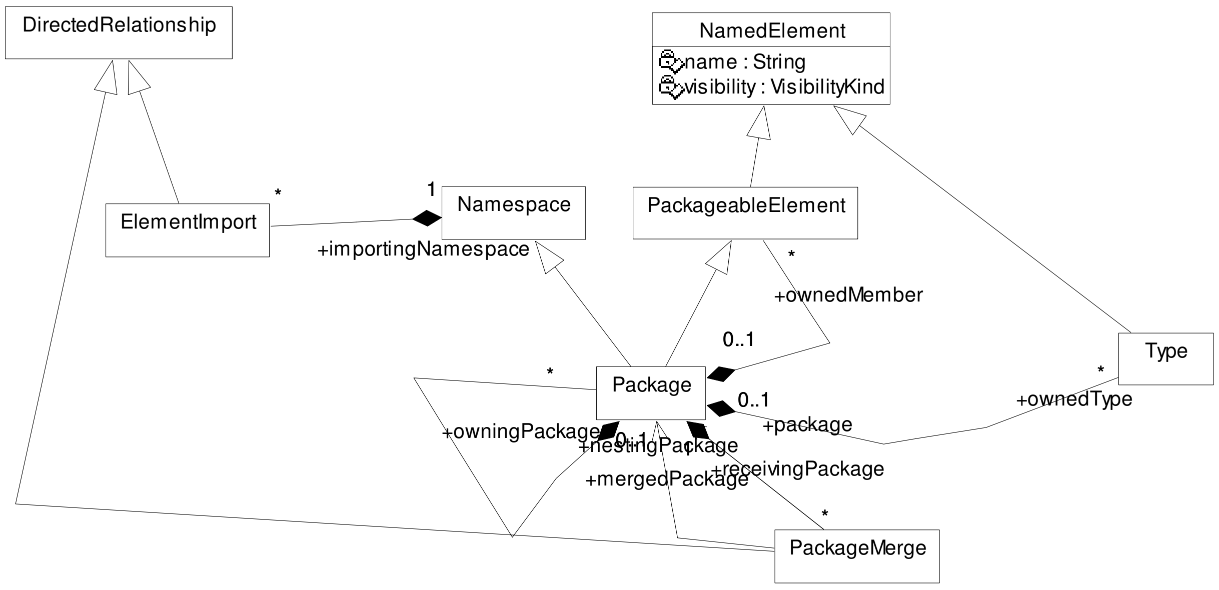
C.5 SMD (Statemachine Diagram)



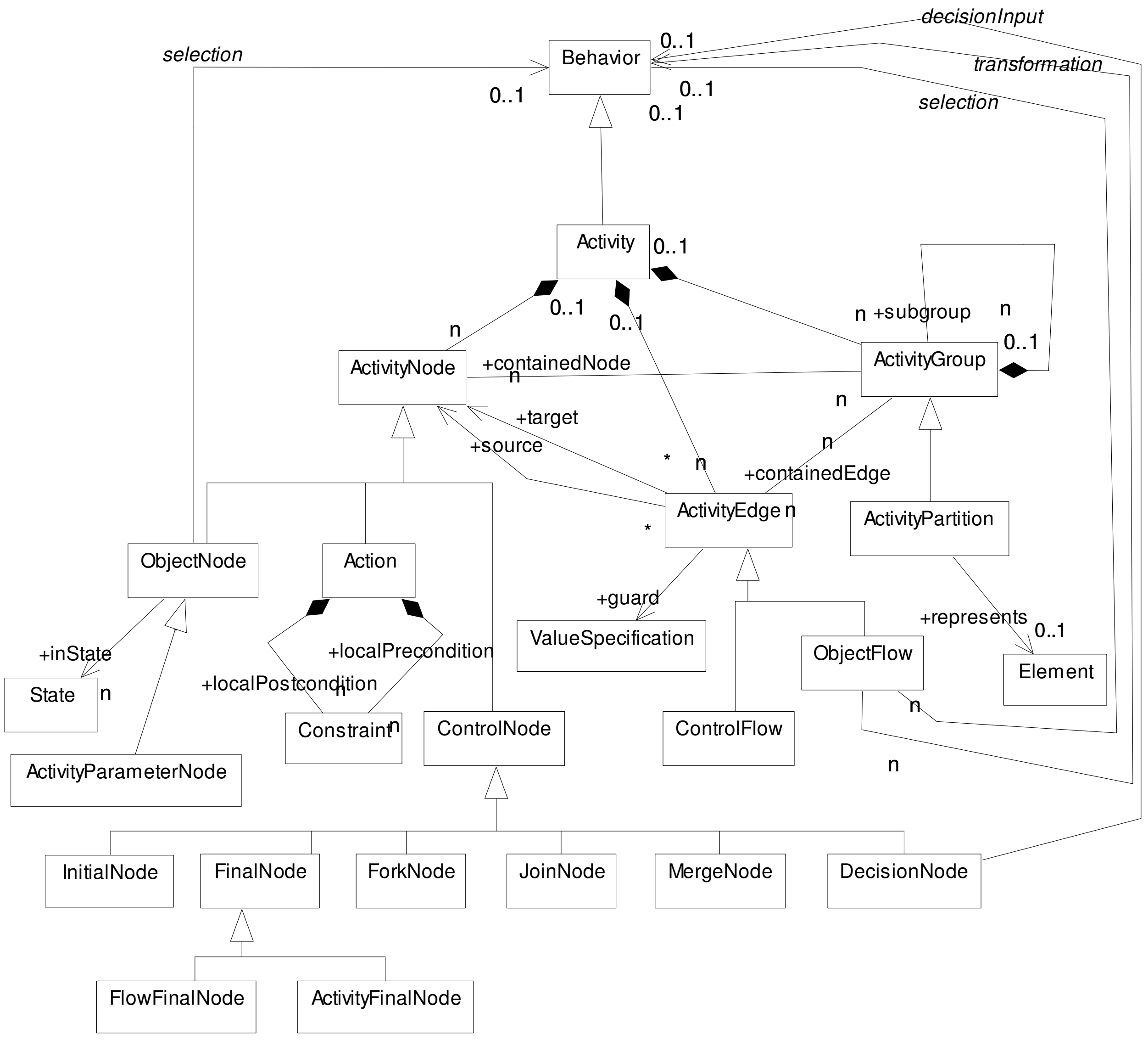
C.6 UCD (Usecase Diagram)



C.7 PD (Package Diagram)



C.8 AD (Activity Diagram)

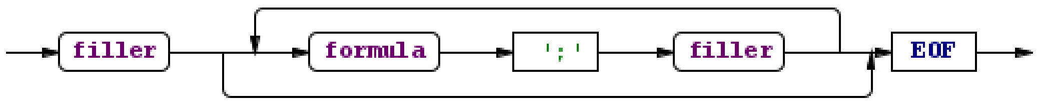


Appendix D

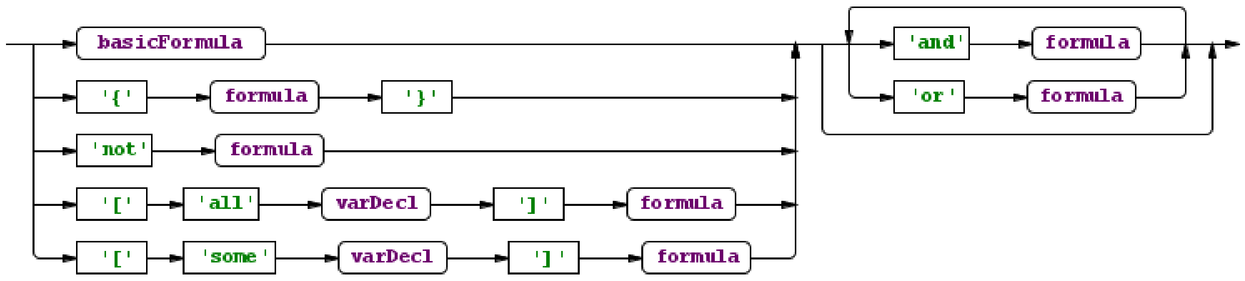
Antlr grammar for Kodkod

D.1 Grammar for Kodkod using railroad diagrams

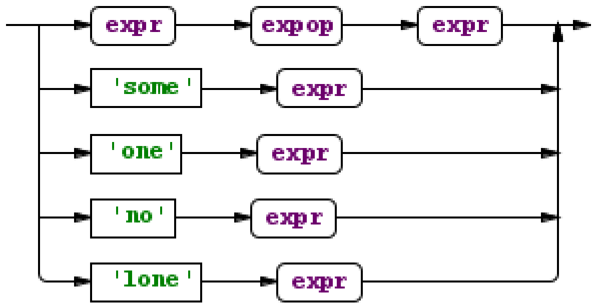
start



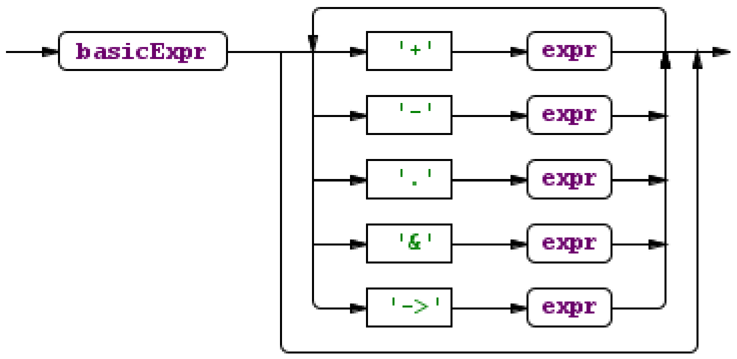
formula



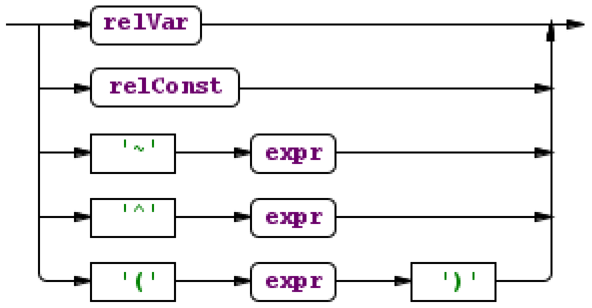
basicFormula



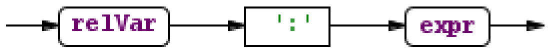
expr



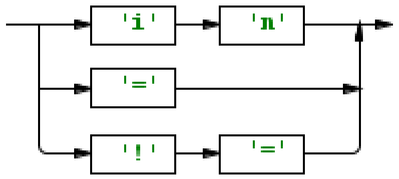
basicExpr



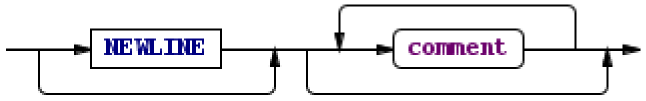
varDecl



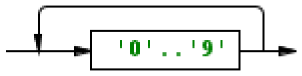
expop



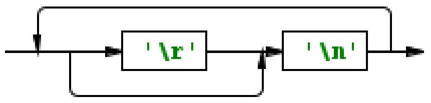
filler



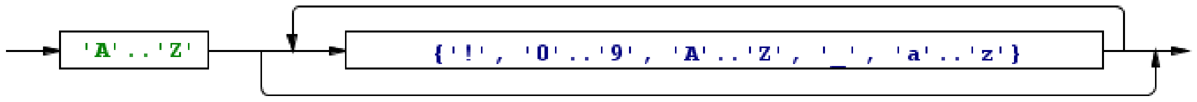
int



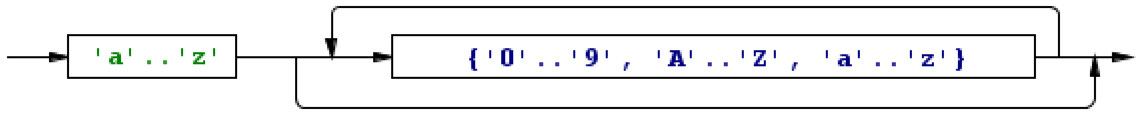
NEWLINE



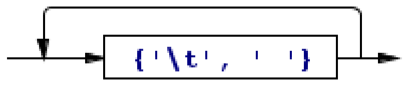
relConst

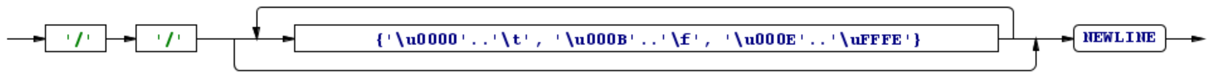


relVar



ws





comment

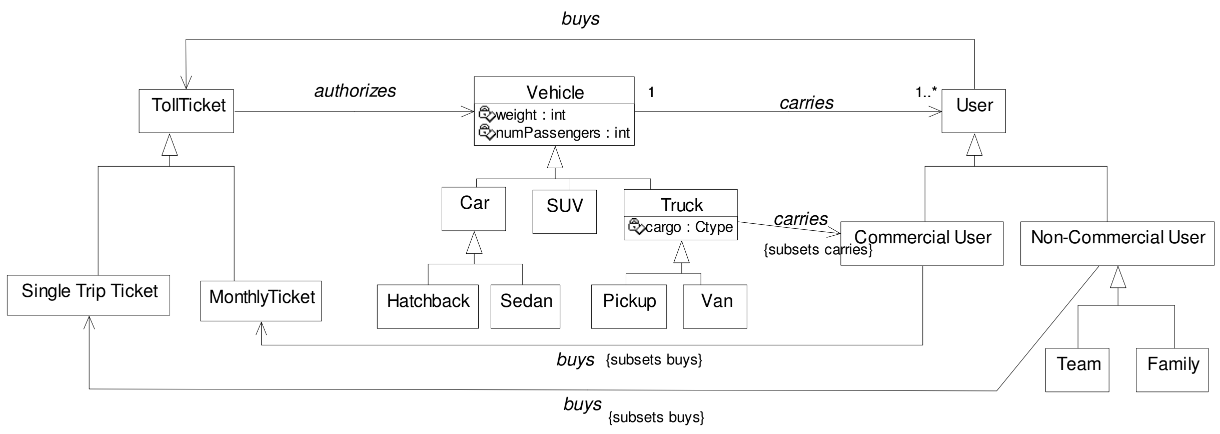
D.2 *Listing of grammar source code*

```
grammar kodkod;


@header {
package mainstuff;

import java.util.HashMap;
import java.util.Map;

import kodkod.ast.Decl;
import kodkod.ast.Expression;
import kodkod.ast.Formula;
import kodkod.ast.Variable;
import kodkod.ast.Relation;



}

@members {
/** Map variable name to Integer object holding value */
private Map<String, Variable> vars = new HashMap<String, Variable>();
public Map<String, Relation> rels = new HashMap<String, Relation>();
public String modelTypeName;

    public class RelationNotFoundException extends RecognitionException{
      private String relName;

      RelationNotFoundException(String rn) {
            relName = rn;
      }

      public String getMessage() {
            return "No relation found with name " + relName;
      }

    }


    private Relation getRel(String relName) throws
RelationNotFoundException {
      if (!relName.contains(new String("!"))) // must be qualified
            relName = modelTypeName + "!" + relName;

      Relation rel = rels.get(relName);
      if (rel != null) return rel;

      RelationNotFoundException rnfe =
                              new RelationNotFoundException(relName);
      throw rnfe;
    }

}



start returns [Formula form] :
      filler {$form = Formula.TRUE;}
      (f=formula  {$form = $form.and($f.form);} ';' filler)*
      EOF;

basicFormula      returns [Formula form] :
       (e1=expr expop e2=expr) {if ($expop.text.equals("in"))
                                  $form = $e1.exp.in($e2.exp);
                              else if ($expop.text.equals("="))
                                  $form = $e1.exp.eq($e2.exp);
                              else if ($expop.text.equals("!="))
                              $form = $e1.exp.eq($e2.exp).not(); }
      | 'some' e=expr {$form = $e.exp.some();}
      | 'one' e=expr {$form = $e.exp.one();}
      | 'no' e=expr {$form = $e.exp.no();}
      | 'lone' e=expr {$form = $e.exp.lone();};

formula returns [Formula form] :
      (f=basicFormula {$form = $f.form;}
      | '{' f=formula '}' {$form = $f.form;}
      | 'not' f=formula {$form = $f.form.not();}
      | '[' 'all' d=varDecl ']' f=formula {$form =
$f.form.forAll($d.decl);}
      | '[' 'some' d=varDecl ']' f=formula {$form =
$f.form.forSome($d.decl);})
      ('and' f=formula {$form = $form.and($f.form);}
      | 'or' f=formula {$form = $form.or($f.form);} )*;


varDecl     returns [Decl decl] :
      v=relVar ':' e=expr {
            vars.put($v.text, Variable.nary($v.text, $e.exp.arity()));
            $decl = vars.get($v.text).oneOf($e.exp);};

expr  returns [Expression exp] :
      r=basicExpr {$exp = $r.exp;}
      ('+' r=expr {$exp = $exp.union($r.exp);}
      | '-' r=expr {$exp = $exp.difference($r.exp);}
      | '.' r=expr {$exp = $exp.join($r.exp);}
      | '&' r=expr {$exp = $exp.intersection($r.exp);}
      | '->' r=expr {$exp = $exp.product($r.exp);})*;

basicExpr returns [Expression exp] :
       relVar {$exp = vars.get($relVar.text);}
      | relConst {$exp = getRel($relConst.text);}
      | '~' r=expr {$exp = $r.exp.transpose();}
      | '^' r=expr {$exp = $r.exp.closure();}
      | '(' r=expr ')' {$exp = $r.exp;};


filler : NEWLINE? comment*;
comment : SINGLE_COMMENT;
relVar      : RELVAR;
relConst: RELCONST;
expop       : EXPOP;

EXPOP       : 'in' | '=' | '!=';
RELVAR  : ('a'..'z')('a'..'z' | 'A'..'Z' |'0'..'9')* ;
INT :   '0'..'9'+ ;
NEWLINE:('\r'? '\n')+ ;
SINGLE_COMMENT: '//' ~('\r' | '\n')* NEWLINE { skip(); };


WS  :   (' '|'\t')+ {skip();} ;
RELCONST: ('A'..'Z')('a'..'z' | 'A'..'Z'| '_' | '!' |'0'..'9')* ;
```

Appendix E

Supplementary information for the transportation example

E.1 UML model



E.2 Extractors

top(M:UML): CD := [

// the set of base classes of M

precondition := *TRUE*,

$Q_{Class}(c)$:= $\neg\exists c_1$: M.Class · M.subClassOf(c, c_1)

]

relatedTo(M:UML, UType:One[Class]): CD := [

// vehicles related to a user type

precondition := submodelOf(UType, M) $\wedge$

```
                    ∃veh:M.Class, user:UType.Class, cr:M.Association ·
                    M.name(veh) = “Vehicle” ∧
                    M.name(user) = “User” ∧
                     M.name(cr) = “Carries”,

    $Q_{Class}(c)$ :=  (c = veh ∨ TC(subClassOf(c, veh))) ∧
                  (c = startClass(cr) ∨ TC(M.subClassOf(c, startClass(cr)))) ∧
                  (user = endClass(cr)  ∨ TC(M.subClassOf(user, endClass(cr))) ∧
                  ¬∃cr1 : M.Association ·  // no restriction of cr to non-UType
                      M.subsets(cr1, cr) ∧
                      (startClass(cr1)  = startClass(cr)  ∨
                          TC(M.subClassOf(startClass(cr1), startClass(cr)))) ∧
                       (user ≠ endClass(cr1) ∧
                        ¬TC(M.subClassOf(user, endClass(cr1)))

]

classDetails(M:UML, C:One[class]): CD := [
   precondition := ∃x:C.Class, mc:M.Class · mc = x,
  $Q_{Class}(c)$ :=   (c = mc ∨ TC(M.subClassOf(c, mc) ∨
                  (∃a:M.Association · M.endClass(a) = c ∧
                                M.startClass(a) = mc )),
   $Q_{Association}(a)$ := M.startClass(a) = mc,
   $Q_{Attribute}(a)$ := M.attrClass(a) = mc
]
```