Merging and Consistency Checking of Distributed Models

by

Mehrdad Sabetzadeh

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

Abstract

Merging and Consistency Checking of Distributed Models

Mehrdad Sabetzadeh

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2008

Large software projects are characterized by distributed environments consisting of teams at different organizations and geographical locations. These teams typically build multiple overlapping models, representing different perspectives, different versions across time, different variants in a product family, different development concerns, etc. Keeping track of the relationships between these models, constructing a global view, and managing consistency are major challenges.

Model Management is concerned with describing the relationships between distributed models, i.e., models built in a distributed development environment, and providing systematic operators to manipulate these models and their relationships. Such operators include, among others, *Match*, for finding relationships between disparate models, *Merge*, for combining models with respect to known or hypothesized relationships between them, *Slice*, for producing projections of models and relationships based on given criteria, and *Check-Consistency*, for verifying models and relationships against the consistency properties of interest.

In this thesis, we provide automated solutions for two key model management operators, Merge and Check-Consistency. The most novel aspects of our work on model merging are (1) the ability to combine arbitrarily large collections of interrelated models and (2) support for toleration of incompleteness and inconsistency. Our consistency checking technique employs model merging to reduce the problem of checking *inter-*

model consistency to checking *intra*-model consistency of a merged model. This enables a flexible way of verifying *global* consistency properties that is not possible with other existing approaches.

We develop a prototype tool, TReMer+, implementing our merge and consistency checking approaches. We use TReMer+ to demonstrate that our contributions facilitate understanding and refinement of the relationships between distributed models.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Models

As software becomes ever more complex, there has been a surge of interest in modelling as a way to bring more discipline to software development, and to raise the level of abstraction at which software engineers perform their tasks. A model is an abstract representation of a certain aspect of a system (existing or to-be), expressed in a notation with certain syntax and semantics. The system in question may be a domain of interest, some software, some software and its environment, a data schema, a user interface, a software architecture, a process to be followed, and so on (Mylopoulos, 1998; van Lamsweerde, 2000).

Analysts may build models of a system to understand and communicate the concepts in the system, or to reason about the system's structure, behaviour, and function. For large-scale projects, modelling is often a distributed endeavour involving multiple teams spread across multiple sites. These teams build multiple overlapping models, representing different perspectives, different versions across time, different variants in a product family, different development concerns, etc. Keeping track of the relationships between these models, managing consistency, and constructing a global view of the models are major challenges.

1.2 Model Management

Model management is concerned with describing the relationships between *distributed models*, i.e., models built in a distributed development environment, and providing systematic operators to manipulate these models and their relationships (Bernstein, 2003; Melnik, 2004; Brunet *et al.*, 2006). Such operators include, among others, *Match*, for finding relationships between disparate models, *Merge*, for combining models with respect to known or hypothesized relationships between them, *Slice*, for producing projections of models and relationships based on given criteria, and *Check-Consistency*, for verifying models and relationships against the consistency properties of interest.

Model management is complicated by a number of factors (CASCON Workshop on Model Fusion, 2006):

F1. ***Discrepancy***: Models may refer to the same or closely related concepts, but they may have differences in how they represent these concepts.

F2. ***Partiality (Incompleteness)***: Models may offer varying degrees of certainty about their content. They may have unknown or under-explored aspects which need to be completed through incremental elaboration.

F3. ***Inconsistency***: Models may be built by teams with different responsibilities and goals. Hence, models may have conflicting purposes, or disagreements over the choice of terminology, design, and usage.

F4. ***Evolution***: Models may evolve over time, so model transformations (e.g., merge) need to be updated if the source models are updated.

F5. ***Scale***: Models may be large and complex. Further, the number of models and model interconnections may be large if development is distributed across many geographical location, organizations, and stakeholder groups.

1.3 Contributions

Our ultimate goal is to develop a suite of synergistic model management operators that are robust in the face of the factors described in Section 1.2. This thesis takes a step towards this goal by providing automated tool-supported solutions for two major operators, Merge and Check-Consistency.

Foundations

We use category theory (Barr & Wells, 1999) as a theoretical foundation for our work. Intuitively, a category is an algebraic structure consisting of a collection of objects and a collection of mappings between these objects. Of specific interest to us is when the objects represent some species of models, and the mappings express potential overlaps between the models. Category theory provides an ideal machinery for dealing with distributed models by making the notion of mapping *explicit*. This enables relating models with different vocabularies and frames of reference (see F1 in Section 1.2).

Further, category theory has a built-in abstraction, called interconnection diagram, for describing *systems of (interrelated) models*. Interconnection diagrams allow model management operators to be defined over systems with an arbitrarily large number of models and mappings rather than just over individual models, or pairs of models related by a single mapping. This contributes to the scalability of our operators (see F5 in Section 1.2).

Context and Scope

Our work on model management operators is motivated primarily by the need to improve the applicability of models for *exploration* of complex systems. Exploration allows developers to build insights and learn about a system that is not yet fully understood. Both model merging and consistency checking are instrumental to exploration: Merging

is a prerequisite to exploratory activities such as brainstorming (Osborn, 1979), walk-throughs (Blum, 1992) and negotiation (Easterbrook, 1994), which typically require the construction of a global perspective. Consistency checking is the basis for decision impact analysis – a technique that helps developers think through their decisions by analyzing how these decisions affect the desired properties (e.g., integrity) of a set of models and the relationships defined between them.

While exploration is not limited to a particular stage of development, it is arguably most crucial in the requirements elicitation stage, where the requirements for a proposed systems are being gathered from its stakeholders, and the relationships between these requirements are being specified.

Models built during requirements elicitation tend to be informal or semi-formal in nature, and their exact meaning relies heavily on the tacit beliefs, perceptions, and assumptions of the people who built them. Since this tacit information is not easily expressible or available to other people, one can never conclusively determine how the contents of models originating from different human sources overlap. Hence, the relationships defined between elicitation models are merely hypotheses, not established facts. The validity of these hypotheses cannot be proved mathematically, but can only be *probed* through exploratory analysis.

Given the importance of exploration in requirements elicitation, we ground our work on elicitation models. Elicitation models are usually specified in simple visual notations such as goal, entity-relationship, and high-level class diagrams; therefore, we concentrate on *graph-based* modelling languages.

Our focus on requirements elicitation in this thesis does not mean that our contributions are inapplicable in other contexts – in fact, the theoretical basis for our work is very general, making it applicable to a variety of problems. This is, for example, evidenced by (Niu *et al.*, 2005), where we apply our merge technique to combine models of software code represented as call graphs.

Despite this generality, one must recognize that the design of tool support for any theoretical solution is inevitably guided by a specific methodology of use for the solution. In this respect, the implementation we describe in this thesis for our merge and consistency checking operators is guided by how these operators are applied during requirements elicitation. This has influenced our decisions about several practical factors, such as specification and representation of models and mappings, balancing flexibility and rigour, visualization, etc. Providing a customizable implementation of our operators that can be instantiated for different contexts and methodologies of use is a significant engineering challenge and is the subject of ongoing research (Salay *et al.*, 2007).

With the above remarks, we highlight in the remainder of this section the specific contributions of this thesis.

1.3.1 Model Merging

We propose a general operator for merging graph-based models expressed in the same notation. Our merge operator is characterized by a category-theoretic concept called *colimit*. The general intuition behind colimits is that they glue objects together with nothing essentially new added and nothing left over (Goguen, 1991). In addition to providing a mathematically precise notion for merge, colimits have the advantage that they directly apply to systems, rather than pairs, of models.

In theory, if a binary merge operator is commutative and associative, then generalization to more than two models can be achieved by repeated merges, in any order. However, such a generalization is often complicated by the need to specify the relationships in a series of steps. Figure 1.1 shows a common way for merging multiple models using binary analysis: We first specify a relationship between two of the models (here, M_1 and M_2) and construct a merge ($Merge_{12}$). In the next step, we relate the merge from the previous step with a new input model (here, M_3), and construct a new merge ($Merge_{123}$), and so on. This step-by-step strategy has two drawbacks:

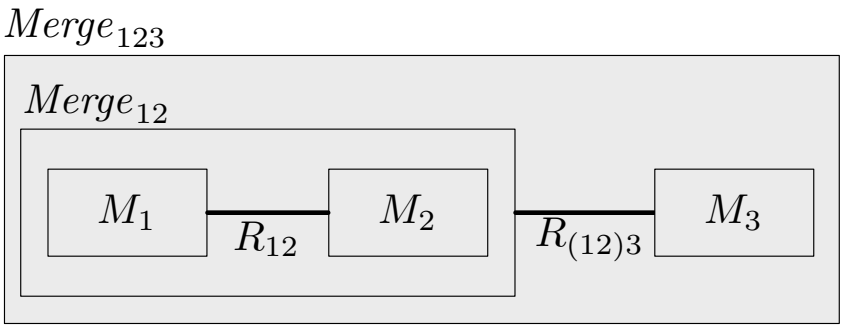


Figure 1.1: Model merging using a binary merge operator

Firstly, it tangles relationship building with model merging. Hence, the relationships built for merge cannot be easily reused for other model management tasks (e.g., checking inter-model consistency), and vice versa.

Secondly, merges may become non-associative even if the merge operator is theoretically associative. This is because in many contexts, e.g., requirements elicitation, relationship building is an inexact process and relies largely on human judgment. In such cases, the order of binary merges can introduce a bias. For example, suppose M_1, M_2, and M_3 in Figure 1.1 are independently-developed perspectives from three different stakeholders; and suppose we are at the step where we want to specify $R_{(12)3}$. While specifying this relationship, one is very likely to see $Merge_{12}$ as being more definitive than M_3, because $Merge_{12}$ encompasses the views of two stakeholders. Hence, $Merge_{12}$ may be implicitly favoured over M_3; and, $R_{(12)3}$ may be defined such that it overrides the inconsistent design choices in M_3, without ever properly analyzing these choices. Therefore, the final merge may be adversely affected by the order of binary merges.

Instead, we provide a merge operator that is *agnostic* to the relationship building process and works directly over systems with an arbitrary number of models and relationships. For example, one can define pairwise relationships between the original models in any order and merge the resulting system in a single shot. This is illustrated in Figure 1.2 for three models.

A second novel characteristic of our model merging approach is its ability to tolerate

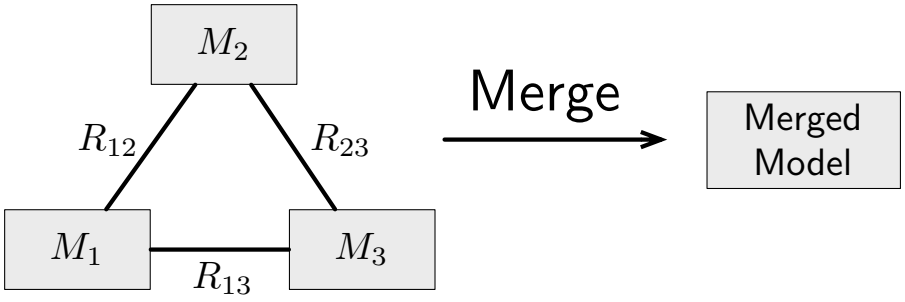


Figure 1.2: Merging systems of models

incompleteness and inconsistency (see F2 and F3 in Section 1.2). This represents a major advance over current practices which typically require that incompleteness and inconsistency should be resolved as soon as it arises. Immediate resolution of incompleteness and inconsistency can be disruptive in projects where ambiguities and conflicts tend to occur frequently (Nentwich *et al.*, 2003). Further, maintaining completeness and consistency at all times can be counter-productive because it may lead to premature commitment to design decisions that have not yet been sufficiently analyzed (Nuseibeh *et al.*, 2001).

Instead, our merge operator provides explicit means for describing incompleteness and inconsistency, enabling developers to merge models without having to make them complete and consistent first. To make incompleteness and inconsistency explicit, we use multiple-valued logics (Ginsberg, 1988) – logics that extend the classical (binary) logic of true and false with additional values.

One of the most famous multiple-valued logics is the one proposed by Belnap (Belnap, 1977), which extends binary logic with two new values: a value unknown to represent "a lack of knowledge", and a value disagreement to represent "too much knowledge", i.e., we can infer that something is both true and false. We use a variant of this logic to demonstrate how one can cope with uncertain or contradictory models originating from different information sources. In general, our work is not restricted to any particular logic and can be parameterized by an arbitrary multiple-valued logic expressed as a lattice.

Our merge operator further incorporates a mechanism to produce the traceability

information required for tracing the elements of a merged model back to their sources. This makes it possible to keep track of how merges change as the source models evolve over time (see F4 in Section 1.2).

1.3.2 Consistency Checking

We provide a flexible operator for consistency checking of distributed models. Traditionally, the problem of consistency checking has been seen as that of checking the consistency of individual models and of individual relationships between pairs of models (Nuseibeh *et al.*, 1994). In large distributed projects, however, we are usually faced with *many* models interrelated by *many* relationships (see F5 in Section 1.2). Therefore, we not only need to check the consistency of individual models and relationships, but also the consistency of a system of models as a whole. This problem, called *global consistency checking*, is known *not* to be reducible to checking consistency of individual models and relationships (van Lamsweerde *et al.*, 1998; Nuseibeh *et al.*, 2001).

Our consistency checking operator provides a solution for global consistency checking when models are described in the same notation. The idea behind our technique is to employ model merging to reduce the problem of checking *inter*-model consistency to checking *intra*-model consistency of a merged model. The realization of this approach requires a merge operator that is well-defined for *any* system of interrelated models even when they are not consistent. This is achieved by the merge operator outlined in Section 1.3.1.

Figure 1.3 shows an overview of our consistency checking approach: Having defined a system of interrelated models, we begin by applying our merge operator to the system. This yields a (potentially inconsistent) merged model along with traceability links from it to the source system. In the next step, we check the consistency of this merged model against the desired (intra-model) constraints, and generate appropriate diagnostics for any violations found. By utilizing the traceability data for the merge, the diagnostics

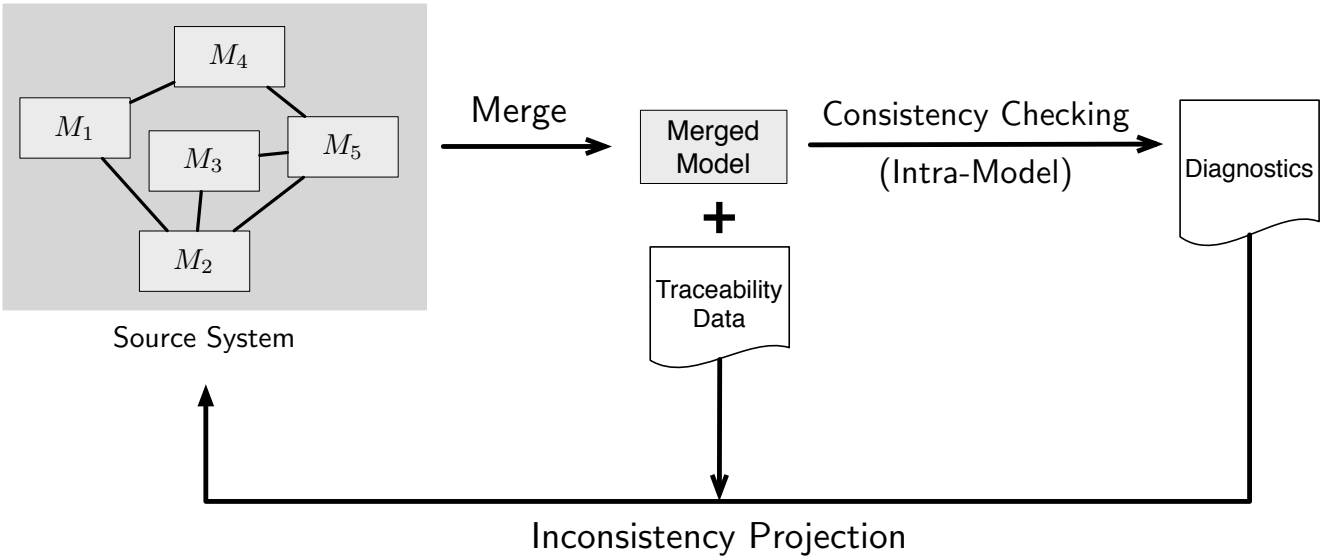


Figure 1.3: Overview of global consistency checking

are automatically projected onto the source system, allowing developers to detect and resolve anomalies in the originating models and relationships.

To describe the consistency constraints of interest, we use a rich logical language, RML (Beyer *et al.*, 2005), based on first-order relational calculus. This language provides enough expressive power to characterize the main structural and behavioural properties of models. A major advantage of our consistency checking operator is that it requires only a single logical formula to be written for each consistency property – the formula applies to any system of models, no matter how many models are involved and how they are related to one another.

Our consistency checking operator comes equipped with a set of generic and reusable expressions capturing recurrent patterns across the consistency constraints of different modelling notations. These expressions can be easily customized to describe a variety of consistency properties over entity-relationship, class, goal, and hierarchical state machine models.

1.3.3 Tool Support

As part of the research undertaken in this thesis, we have implemented our merge and consistency checking operators into a prototype model management tool, TReMer+, developed in collaboration with Shiva Nejati, Steve Easterbrook, and Marsha Chechik.

We use TReMer+ to demonstrate the usefulness of our operators for exploratory analysis. Specifically, we study how our operators help developers refine their understanding of the relationships between independently-developed requirements models.

TReMer+ has been presented to the academic and practitioner communities at various stages of its development. The tool and the case studies we have conducted with it are available at

`http://www.cs.toronto.edu/~mehrdad/tremer/`

1.4 Organization

The remainder of this thesis is organized as follows:

We review in Chapters 2 and 3, respectively, the conceptual background, and the mathematical foundations of our work. In Chapter 4, we present our merge operator, reported in (Sabetzadeh & Easterbrook, 2006), for combining incomplete and inconsistent models. In Chapter 5, we describe our consistency checking operator, reported in (Sabetzadeh *et al.*, 2007a), for verifying global consistency properties. Implementation is discussed in Chapter 6, using material from (Sabetzadeh *et al.*, 2007b; Sabetzadeh *et al.*, 2008). We conclude in Chapter 7 with a summary of the thesis, an outline of important limitations, and a number of directions for future research.

Chapter 2

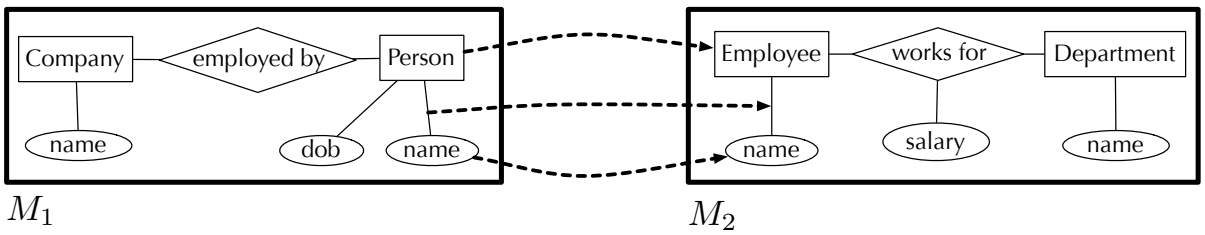
Background

Model merging and consistency checking have been studied for many years. In this chapter, we present background information on these two operations and review the relevant literature.

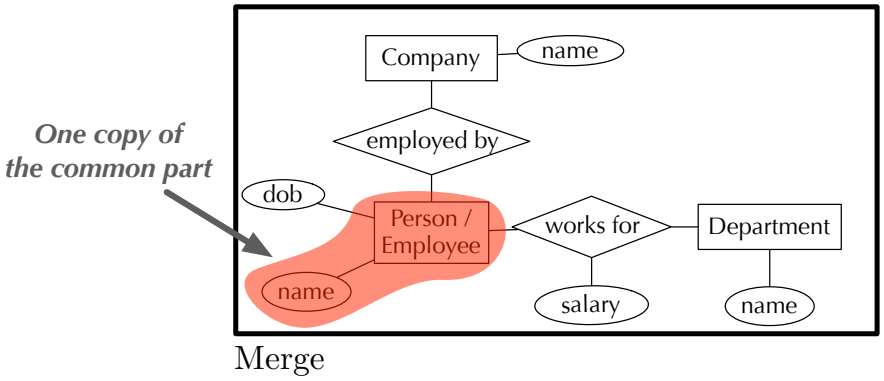
2.1 Model Merging

The primary goal of model merging is to unify the overlaps between a set of models. More precisely, if a concept appears in more than one source model, only one copy of it should be included in the merged model. This property is known as *non-redundancy*.

The overlaps between different models can be specified either implicitly (e.g., using name equivalence if models have a common vocabulary, or identifier equivalence if models have common ancestors), or explicitly via mappings between models. Recent research on model merging, e.g., (Bernstein, 2003; Brunet *et al.*, 2006), favours explicit mappings and indeed suggests that mappings should be treated as first-class artifacts. Such treatment is crucial for models that are developed independently. These models usually have different vocabularies and might be structured differently. To merge a set of independently-developed models, one needs to be able to explicitly specify how the vocabularies and structures in the models overlap.



(a)



(b)

Figure 2.1: Example of model merging

Example 2.1 (model merging) Figure 2.1(a) shows two independently-developed perspectives, represented as entity-relationship diagrams, and a mapping that equates the corresponding elements of these perspectives. Figure 2.1(b) shows the merge. This merge is redundancy-free because it has only one copy of the common parts.

Non-redundancy is only the most basic requirement for merge. Model merging is often subject to additional correctness criteria. The most notable of these criteria are:

- **Completeness**: Merge should not lose information, i.e., it should represent all the source models completely.
- **Minimality**: Merge should not introduce information that is not present in or implied by the source models.

- **Semantic Preservation**: Merge should support the expression and preservation of semantic properties. For example, if models are expressed as state machines, one may want to preserve their temporal behaviours to ensure that the merge properly captures the intended meaning of the source models.

- **Totality**: Merge should be well-defined for *any* set of models, whether or not they are consistent. This property is of particular importance if one wants to tolerate inconsistency between the source models.

These additional criteria are not universal to all model merging problems. For example, completeness and minimality, in the strong sense described above, may be undesirable if model merging involves conflict resolution as well, in which case the final merge can potentially add or delete information (Noy & Musen, 2000). Semantic preservation may be undesirable when one wants to induce a design drift or perform an abstraction during merge. Such manipulations are usually not semantics-preserving (Kalfoglou & Schorlemmer, 2005). And, totality may be undesirable when the source models are expected to seamlessly fit together. In such a case, the source models should be made consistent before they are merged (Uchitel & Chechik, 2004).

2.1.1 Classification of Merge Based on Input and Output

In (Darke & Shanks, 1996), three types of model merging are distinguished based on the input expected by and the output produced by the merge operation:

- **Single representation scheme**: A single notation is used for specifying all the source models. Merge is done by an automated or semi-automated algorithm and the result is represented in the same notation as that of the source models.

- **Pre-merge translation**: The source models are in different notations. The models are translated into a common notation first. The merge is computed and represented in this common notation.

- **Post-merge translation**: This is identical to the single representation scheme except that the result is translated into a different notation after the merge operation.

An observation that follows from this classification is that merge algorithms are typically defined over a single notation – adapting the input to and the output from a merge algorithm is left to pre- and post-merge processing.

Despite its usefulness, the classification must be treated with some care. In particular, the classification assumes that if the source models are represented in the same notation (i.e., are homogeneous), then the merge can be performed in that notation as well. This is not always true: A translation into a new notation may be required even when models are homogeneous. For example, hierarchical data definition languages like nested-relational and XML schemata may be transformed to flat structures before being merged (Melnik *et al.*, 2003). Another example is resolution of parallelism in state machine merging where parallel states are replaced with their semantically equivalent non-parallel structures prior to the merge operation (Nejati *et al.*, 2007).

2.1.2 Merge Strategies

The number of models to be merged may be more than two. Hence, every merge framework needs a strategy for sequencing and grouping the source models during merge. Batini et al. classify merge computation strategies into two groups (Batini *et al.*, 1986):

- **Binary strategies** allow merging two models at a time. To merge more than two models, the result of merging two models is considered as a new model and then merged with another model.

- **n-ary strategies** allow merging n models at a time ($n > 2$). The most general case is when n is not fixed, and can assume an arbitrary value.

For a binary merge operator, certain algebraic properties might be expected (Brunet *et al.*, 2006):

Idempotence

$\mathsf{Merge}(M_1, M_1, R_{11}) = M_1$ (where R_{11} is the identity relationship)

Merging a model with itself should return the same model, assuming that the given relationship completely maps the model onto itself. In practice, this property may be too strong, as one may be willing to accept a result that is isomorphic, rather than identical, to the original.

Commutativity

$\mathsf{Merge}(M_1, M_2, R) = \mathsf{Merge}(M_2, M_1, R)$

This simply states that, for a given relationship, it should not matter which order the models are presented in.

Associativity

$\mathsf{Merge}(\mathsf{Merge}(M_1, M_2, R_{12}), M_3, R_{(12)3}) = \mathsf{Merge}(M_1, \mathsf{Merge}(M_2, M_3, R_{23}), R_{1(23)})$

Together with commutativity, associativity ensures that generalization to multiple models can be achieved by repeated merges, in any order. However, as we argued in the Chapter 1, such generalization is complicated by the need to specify a new relationship at each step. Hence, it is usually more practical to define a general merge operator using an n-ary strategy.

2.1.3 Qualitative Aspects of Merge

To have practical value, a merge framework may be expected to have certain qualitative capabilities, including, among others, the following:

- **Understandability**: Merges should be easy to understand for both analysts and end-users.

- **Support for traceability**: The elements of a merged model should be traceable back to their origins. This is necessary to support cooperative work among teams and to ensure that the contributions of all the stakeholders are properly taken into consideration (Gotel & Finkelstein, 1994).

- **Support for exploration of alternatives**: When models are developed by distributed teams, one can never be sure how different models should relate to one another. A merge framework should make it possible to describe different alternatives for relating a set of models and to configure different merges according to these alternatives.

2.1.4 Other Notions for Combining Models

Merge is not the only way in which a set of models may be put together. Despite a lack of consensus on terminology, the literature distinguishes three different notions for combining models: One is merge, which was already discussed. The other two are *composition* and *weaving* which we describe below. In this thesis, we address only merge.

- **Composition** refers to the process of assembling a set of components and verifying that the result fulfills a desired function. In contrast to merge, composition treats models as black-box artifacts, and as such, its line of sight into the contents of the models is limited to the interfaces that the models expose to the outside world. A second difference between composition and merge is that, in composition, multiple copies of the same model may be included in a single system to denote the fact that the system has several components of the same type (Fiadeiro & Maibaum, 1992).

  In composition, each model is considered as an object with an interface and, optionally, a set of rules describing the assumptions about the environment and the properties that the model guarantees when it is part of a system that satisfies the environmental assumptions. Composition has been mainly studied as a way to

tackle the state explosion problem in verification by breaking down the construction of a large system into smaller self-contained modules (Clarke *et al.*, 1999); but more recently, it has also been considered for analysis of interactions between different features of a system (Hay & Atlee, 2000). For example, in the telecommunication domain, several features such as call-forwarding, auto-answering, and three-way-calling may need to be composed to control the establishment of a connection between two end-points. It is important to verify that these compositions do not exhibit undesirable behaviours (Zave *et al.*, 2004).

- **Weaving** is the predominant notion of model combination in Aspect-Oriented Software Development (AOSD) (Filman *et al.*, 2004). AOSD is an attempt to provide better separation of concerns in software design. This can in part be achieved by means of encapsulating different concerns into distinct models. However, some concerns defy encapsulation as they cut across many models. A classic example of a crosscutting concern is logging which affects all logged activities in a system. Given a set of crosscutting concerns, weaving is the process of incorporating these concerns into an existing set of models. Weaving can be done in various ways depending on the nature of the models and concerns involved. For example, at the source code level, weaving may be implemented through pre-processing (Laddad, 2003).

2.1.5 Literature Review

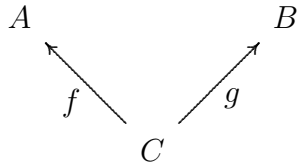
We provide a survey of recent model merging approaches in the areas of database development, software engineering, and the Semantic Web. A critical comparison between our model merging approach and other existing approaches is given in Chapter 4.

Database Development

References to model merging in the database literature go as far back as the late 1970's when an explicit conceptual step, called conceptual schema design, was introduced into database development practices. Conceptual schema design aims to create an abstract global schema that captures the data requirements of a proposed application. Batini et al. (Batini *et al.*, 1986) note the distributed and evolutionary nature of data requirements acquisition, and the fact that this often gives rise to several partial schemata. To create a global schema, these partial schemata need to be merged. Numerous approaches to schema merging have been proposed. A survey of several early approaches is given in (Batini *et al.*, 1986). These early approaches have been superseded by the more recent work we review here.

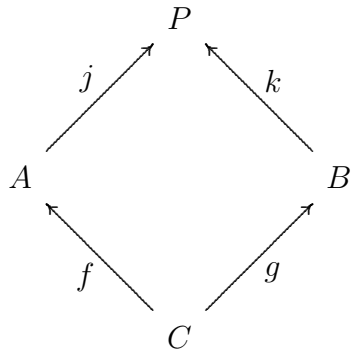
(Buneman *et al.*, 1992) provides a lattice-theoretic approach for merging schemata represented as directed graphs. A merge is constructed by inducing an ordering on the source schemata and computing a least upper bound. Relationships between schemata are based on name equivalences, hence users may need to alter the source schemata as needed to unify their vocabularies prior to merge. **(Pottinger & Bernstein, 2003)** generalizes (Buneman *et al.*, 1992) by making inter-schema relationships explicit and providing a richer mechanism for describing overlaps between the source schemata.

Several schema merging approaches have proposed the use of category theory and the concept of colimit (Barr & Wells, 1999) for characterizing merge[1]. A special colimit construction is for the case where the common parts of a pair of models A, B are captured by a third model C and two mappings f, g that describe how this shared model is represented in each of the source models:



[1]We formally introduce category theory and colimits in Chapter 3.

The colimit of this diagram, called a *pushout*, is a model P that combines A and B with respect to the overlaps specified by C, and mappings j, k that show how each of A and B is embedded into P:



In **(Cadish & Diskin, 1996)**, the categorical notion of sketch – intuitively, a directed graph augmented with logical equations – is used to capture schemata. A mapping between sketches is a mapping of their underlying graphs compatible with their equations. Merges are characterized using pushouts.

(Alagic & Bernstein, 2001; Goguen, 2005) use *institutions* as a unifying theory for schema merging. Institution theory, introduced by Goguen and Burstall (Goguen & Burstall, 1992), is a category-theoretic framework for studying algebraic specification languages[2]. Every specification language has a logical formalism for expressing the desired properties (i.e., axioms) in a system. Briefly, the goal of institution theory is to provide a general specification framework that is independent of the choice of logical formalism. To obtain a specification language suitable for a given task, this general framework must be instantiated with a specific logical formalism.

The core components of a logical formalism are semantic objects (i.e., models, in model-theoretic terms) and axioms. The presentation of these components is parameterized by an arbitrary *signature*. Intuitively, a signature describes the structure of the semantic objects associated with it and provides a vocabulary for expressing axioms about these objects. Signatures change quite frequently throughout development.

[2](Tarlecki, 1999) provides an excellent introduction to institution theory.

To deal with signature changes, one needs an appropriate basis for moving from one signature to another. This is achieved by *signature morphisms*.

In both (Alagic & Bernstein, 2001; Goguen, 2005), a schema is defined as a signature and a set of axioms. These axioms describe the integrity constraints for the databases (i.e., the semantic objects) over the schema. (Alagic & Bernstein, 2001) and (Goguen, 2005) respectively use Horn logic and equational logic for expressing axioms. A schema morphism is a signature morphism which extends to a semantically sound mapping of axioms. Schema merges are computed using colimits. Under certain conditions, these merges have counterpart constructions at the level of databases. This means that individual databases of the source schemata can be integrated to form a database of the merged schema.

(Alagic & Bernstein, 2001) assumes that schemata are homogeneous. That is, signatures are drawn from the same category and axioms are expressed within the same logical formalism. (Goguen, 2005) provides a generalization to the heterogeneous case where different species of signatures and logical formalisms may be used for describing schemata. The machinery used for this purpose is that of *institution morphisms*, providing a way for translating between different notions of signature and logic (Goguen & Rosu, 2002).

Software Engineering

Software engineering research deals extensively with model merging. In their survey (Darke & Shanks, 1996), Darke and Shanks identify merge as one of the core activities in distributed development. Several papers study model merging for use cases, class diagrams, state machines, transformation systems, etc. We review some recent approaches here[3].

[3]In software engineering, there is also a large volume of work on merging software code. We do not discuss this work. See (Mens, 2002) for a survey on the subject.

(Fiadeiro & Maibaum, 1992) describes a framework for modular development of reactive systems using category theory. With respect to the classification of model combination activities in Section 2.1.4, this work falls under composition. Nevertheless, the work is particularly relevant to model merging because of its use of colimits for composing specifications – as discussed earlier, colimits are used for model merging, too. It is interesting to see how the same construct can be employed for characterizing two different activities.

The framework utilizes many of the same algebraic specification principles used in institution theory [4], but the formalism introduced by the framework for expressing temporal specifications and their morphisms is not directly characterizable by institutions. Each specification is made up of a signature and a set of temporal formulas. Temporal specification signatures are more complex than the type of signatures used in (Alagic & Bernstein, 2001; Goguen, 2005), surveyed earlier, for capturing database schemata. This is due to the addition of new programming constructs such as variables, arrays, and actions. A specification morphism is a signature morphism that extends to a semantically sound mapping of temporal formulas. Here, morphisms are not used for expressing conceptual overlaps between specifications, but rather to prescribe the relative behaviours of these specifications using program composition primitives such as sequencing, iteration, parallelism, and the like. A composed specification is arrived at by colimit computation.

Intuitively, the reason why colimits can be used for both merge and composition lies in an interesting property of colimits, which is that they never combine any two concepts unless the concepts are explicitly related by the morphisms. In merge, this property is used to avoid the unification of homonyms – concepts that share the same name label while being semantically distinct. In composition, the property is used to avoid interference between the internal structures of the participating components.

[4] An informal introduction to institution theory was given in our survey of schema merging approaches.

(Heckel *et al.*, 1999) provides a framework for merging graph transformation systems. Each transformation system is given by a type graph and a set of rewrite rules. A mapping between a pair of graph transformation systems is made up of a mapping between their type graphs and a set of mappings between their rules. Rule mappings are required to satisfy certain properties to avoid undesirable interactions between different rules. The merge operation is characterized using pushouts.

(Easterbrook & Chechik, 2001) describes an approach for merging incomplete and inconsistent state machines. A state machine is represented as a set of states, a set of transitions between states, and a set of variables whose values vary from state to state. To denote incompleteness and inconsistency, each state machine is extended with a logic with multiple truth values (Belnap, 1977). A mapping between a pair of state machines consists of two parts, a signature map and a truth map. The signature map describes the correspondences between the state machines and further establishes a common vocabulary for the merge. The truth map specifies how the truth values of the source state machines should be combined in the merge. The notion of mapping defined by the approach results in a straightforward binary merge algorithm for state machines.

(Sabetzadeh & Easterbrook, 2003) uses category theory to provide an algebraic characterization of the merge algorithm in (Easterbrook & Chechik, 2001). State machines are defined as in (Easterbrook & Chechik, 2001) but with their underlying structure based on graphs rather than sets and relations. Mappings between state machines are described using truth-preserving homomorphisms. State machine merges are computed using pushouts.

(Richards, 2003) provides a method for reconciling natural language use case models. The approach works by constructing a cross-table where the rows are use case sentences and the columns are the keywords and phrases of these sentences. A cell in the table is marked if the sentence in question has the corresponding keyword or phrase in it. This table is processed using Formal Concept Analysis (FCA) (Ganter & Wille, 1998)

– a technique commonly used for finding and visualizing relationships between different concepts in a domain. The result is a structure, called a concept lattice, that can be used for comparing use case models and merging common concepts.

(Alanen & Porres, 2003) provides a generic approach for merging diagrams in the UML notation. Given a pair of diagrams, the approach first finds the differences (deltas) between the diagrams and their common ancestor. The deltas are described as a sequence of elementary transformations for creating, deleting, and modifying diagram elements. To construct a merge, the deltas are applied to the common ancestor. This is straightforward when the deltas do not overlap; however, conflicts may arise when the deltas refer to the same elements of the common ancestor. The approach provides a semi-automated conflict resolution algorithm for dealing with such conflicts. **(Letkeman, 2005)** independently develops an approach similar to that of (Alanen & Porres, 2003) for merging UML diagrams. The work provides a practical tool for merge and offers interesting insights about the challenges presented by model merging in a production environment, e.g., integration with model-based development processes, interaction with model repositories and local histories, visualization, etc. **(Mehra *et al.*, 2005)**, also independently, proposes a tool-supported approach for merging graphical diagrams based on computing deltas and incorporating them into a common ancestor. But, in contrast to (Alanen & Porres, 2003; Letkeman, 2005), the approach conceives of conflict resolution during merge as an entirely manual process.

(Uchitel & Chechik, 2004; Nejati *et al.*, 2007) address the problem of behavioural model merging, with the main goal being the preservation of temporal properties of the source models in their merges. In (Uchitel & Chechik, 2004), mappings between models are implicit and are induced by behavioural bisimulation (Milner, 1989); whereas in (Nejati *et al.*, 2007), mappings are explicit and are described by binary relations over the states of the source models. In both approaches, models are compared via a notion of ordering, called refinement (Larsen & Thomsen, 1988). A merge is charac-

terized as a common refinement of the source models. This characterization allows one to prove a number of interesting results about what temporal properties of the source models are preserved in their merges.

There has also been work on defining languages for model merging, e.g., the Epsilon Merging Language (EML) **(Kolovos *et al.*, 2006)**. EML is a rule-based language for merging models with the same or different meta-models. The language distinguishes four phases in the merge process and provides flexible constructs for defining the rules that should be applied in each phase. These phases are (1) comparison (identification of correspondences), (2) compliance checking (examination of corresponding elements to identify potential conflicts), (3) merging (computation of a duplicate-free union of two models), and (4) reconciliation (resolution of inconsistencies and restructuring). Despite its versatility, the current version of EML does not formalize the conditions and consequences of applying the merge rules, and hence, in contrast to our approach described in Chapter 4, (Kolovos *et al.*, 2006) does not provide a mathematical characterization of the merge operation.

Semantic Web

Model merging has been studied in the Semantic Web as a way to consolidate ontologies originating from different communities. Below, we outline some of the proposed approaches to ontology merging. For a more comprehensive treatment, see (Kalfoglou & Schorlemmer, 2005).

(Noy & Musen, 2000) provides an incremental method for ontology merging. The merge process begins with creating a list of potential pairwise matches between the elements of the source ontologies based on linguistic similarities between element names. Then, the following cycle happens: (1) The user triggers the merge of a pair of elements either by accepting a match from the list of potential matches, or by defining a new match. Depending on the type of elements being merged, an appropriate unification

operation is performed. This may trigger further unification operations. (2) Conflicts introduced by the operations in step (1) are resolved. Conflicts that cannot be resolved automatically are reported to the user. (3) The list of potential matches is updated to account for the additional structure imposed by steps (1) and (2).

Similarly to (Richards, 2003) reviewed earlier, **(Stumme & Maedche, 2001)** characterizes merge using ideas from Formal Concept Analysis. Here, the input to the merge process is a pair of ontologies and a set of natural language documents. A merge is carried out in three steps: (1) The documents are analyzed to identify which concepts of the source ontologies appear in them. The results are used to generate a pair of cross-tables, one for each source ontology. The rows in each table are the names of the documents, and the columns are the concepts of the respective source ontology. (2) These tables are combined into a new table by taking the disjoint union of their columns. This combined table is processed to generate a concept lattice. (3) A merged ontology is created through a semi-automated analysis of individual elements in the concept lattice. The approach does not provide a mechanism for conflict resolution – possible anomalies and duplicates have to be resolved by a domain expert.

Several papers, e.g., **(Jannink *et al.*, 1998; Schorlemmer *et al.*, 2002; Goguen, 2005; Hitzler *et al.*, 2005)** formalize ontology merging using pushouts. In recent work, Zimmermann et al. **(Zimmermann *et al.*, 2006)** argue that pushouts are not expressive enough for ontology merging, because in addition to the overlaps between the source ontologies, one often has to deal with information that is missing from the ontologies, too. For example, to merge a pair of ontologies one of which elaborates the concept of Physician and the other which – the concept of Nurse, one first needs to introduce the missing abstract concept MedicalStaff. Pushouts do not provide a way to address this missing information during merge. In contrast, as we see in Chapter 4, colimits in their general form provide enough expressiveness for bridging such information gaps between the source models.

2.2 Consistency Checking

Consistency checking is a major research topic with various nuances and complexities. In software engineering, consistency checking has been studied since the 1980's and techniques have been developed to address the problem for a wide range of development artifact including logical specifications (Nuseibeh *et al.*, 1994; Easterbrook & Nuseibeh, 1996; van Lamsweerde *et al.*, 1998; Spanoudakis *et al.*, 1999; Bowman *et al.*, 2002; Jackson, 2006), conceptual graphs (Delugach, 1992), state diagrams (Glinz, 1995; Easterbrook & Chechik, 2001), XML documents (Nentwich *et al.*, 2003), UML models (Zisman & Kozlenkov, 2001; Elaasar & Briand, 2004; Egyed, 2006), natural language documents (Gervasi & Zowghi, 2005), and so on.

2.2.1 Notions of Consistency

Formal approaches to consistency checking build on one of the following two notions:

Let $\mathcal{L}$ be a logical language, and let $\models$ denote the satisfaction relation in $\mathcal{L}$.

- ***Logical Consistency.*** Given a set $S = \{\varphi_1, \ldots, \varphi_n\}$ of formulas in $\mathcal{L}$, the formulas are said to be logically consistent if there exists an instance object O admitted by the semantics of $\mathcal{L}$ such that:

$$O \models \varphi_1,\ O \models \varphi_2,\ \ldots,\ O \models \varphi_n.$$

  The formulas are said to be logically inconsistent if no such instance object exists.

- ***Conformance.*** Given an instance object O (admitted by the semantics of $\mathcal{L}$) and a formula φ in $\mathcal{L}$, the object O is said to be consistent with respect to (or, conformant to) φ if $O \models \varphi$. Otherwise, O is said to be inconsistent (or, nonconformant).

2.2.2 Consistency Checking of Individual Descriptions

We illustrate logical consistency and conformance over individual descriptions before generalizing the notions to multiple descriptions in Section 2.2.3.

Example 2.2 (checking logical consistency) Consider the following propositional specification:

$$S = \{\mathsf{flies} \Rightarrow \mathsf{bird},\ \mathsf{bird} \Rightarrow \mathsf{has\text{-}bill},\ \mathsf{bat} \Rightarrow \mathsf{flies},\ \mathsf{bat} \Rightarrow \neg\mathsf{has\text{-}bill},\ \mathsf{bat}\}$$

S is logically inconsistent because there is no truth assignment satisfying all the formulas in S.

Example 2.3 (checking structural conformance) Consider the class diagram M in Figure 2.2 and the following first order well-formedness property:

$$\varphi = \neg\exists\, x, y, z\ (y \neq z) \wedge \mathsf{Inherits}(x, y) \wedge \mathsf{Inherits}(x, z).$$

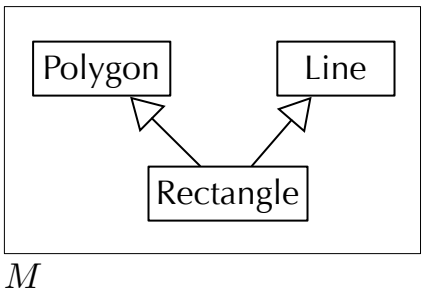


Figure 2.2: Illustration for structural conformance checking

The property states that a class cannot have more than one parent class (e.g., to ensure that the class diagram can be easily translated to Java code). The model in Figure 2.2 is inconsistent with respect to φ because $M \not\models \varphi$.

Example 2.4 (checking behavioural conformance) Consider the state machines M_1 and M_2 in Figure 2.3 describing two variant versions of a traffic light: A simple version M_1 that is always on, and a second version M_2 that can also be off. The property

$\varphi = \mathbf{G}(\mathsf{red} \Rightarrow \mathbf{X}\,\mathsf{green})$, expressed in Linear Temporal Logic (Clarke *et al.*, 1999), stating that red is always followed by green holds over M_1 but not over M_2 (because of the path $\mathsf{red}, \mathsf{off}, \mathsf{green}$). Hence, M_2 is inconsistent with respect to φ.

Both M_1 and M_2 will be consistent if the property is changed to $\varphi = \mathbf{G}(\mathsf{red} \Rightarrow \mathbf{F}\,\mathsf{green})$, saying that one always eventually gets from red to green.

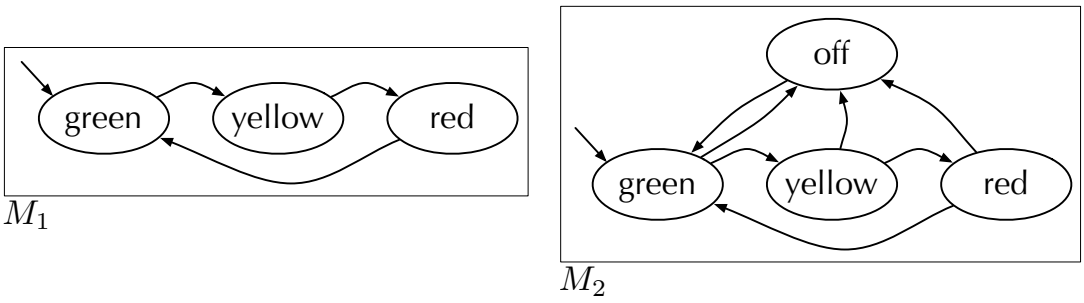


Figure 2.3: Illustration for behavioural conformance checking

2.2.3 Consistency Checking of Multiple Descriptions

For multiple descriptions that are related, one not only needs to check the consistency of individual descriptions but also their consistency according to the relationships defined between them. Consistency checking of interrelated descriptions is done in one of the following ways (Finkelstein & Sommerville, 1996; van Lamsweerde *et al.*, 1998; Nuseibeh *et al.*, 2001):

1. Relationships have specific consistency rules associated with them to ensure well-formedness, compatibility, process compliance, etc. Consistency checking is performed by checking the corresponding rules on pairwise relationships between the descriptions.

2. Descriptions are first combined according to the relationships between them. Checking consistency of the descriptions then amounts to checking consistency of their combination.

Example 2.5 (pairwise rules – homogeneous models) Consider class diagrams M_1, M_2 in Figure 2.4 and the relationship R between them, which equates Rect in M_1 and Rectangle in M_2. If multiple inheritance is undesired, R should be deemed inconsistent because it equates two concepts without equating their parent concepts. The rule for checking this property is as follows:

$$\varphi = \neg\exists x, y, z, t \;\; \mathsf{Inherits}(x, y) \wedge \mathsf{Inherits}(z, t) \wedge R(x, z) \wedge \neg R(y, t).$$

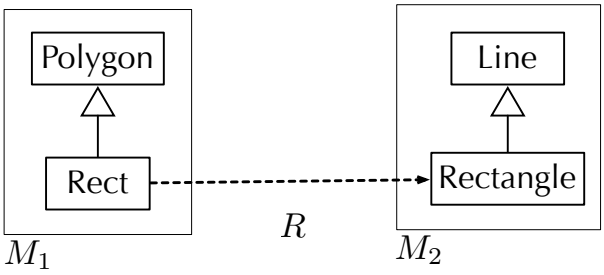


Figure 2.4: Pairwise checking of homogeneous models

Example 2.6 (pairwise rules – heterogeneous models) Consider the class diagram M_1, the sequence diagram M_2, and the relationship R between the two in Figure 2.5. The models, adopted from (Egyed, 2006), represent an early-stage design snapshot of a video-on-demand system. Unlike our previous example where the pairwise relationship between the models described element equalities, the relationship in Figure 2.5 denotes instantiation, i.e., $R(x, y)$ does not mean that x and y are equivalent, but rather means y is an instance of x.

A consistency rule that one might want to check over the models in Figure 2.5 is whether messages in the sequence diagram have corresponding methods in the class diagram. This is captured by the following:

$$\varphi = \neg\exists msg, obj, cls \;\; \mathsf{Target}(msg, obj) \wedge R(cls, obj) \wedge \neg\mathsf{Contains}(cls, msg.\mathsf{name}).$$

The models in the figure fail to satisfy the above rule because the doStream message in M_2 does not have a corresponding method in the Streamer class of M_1.

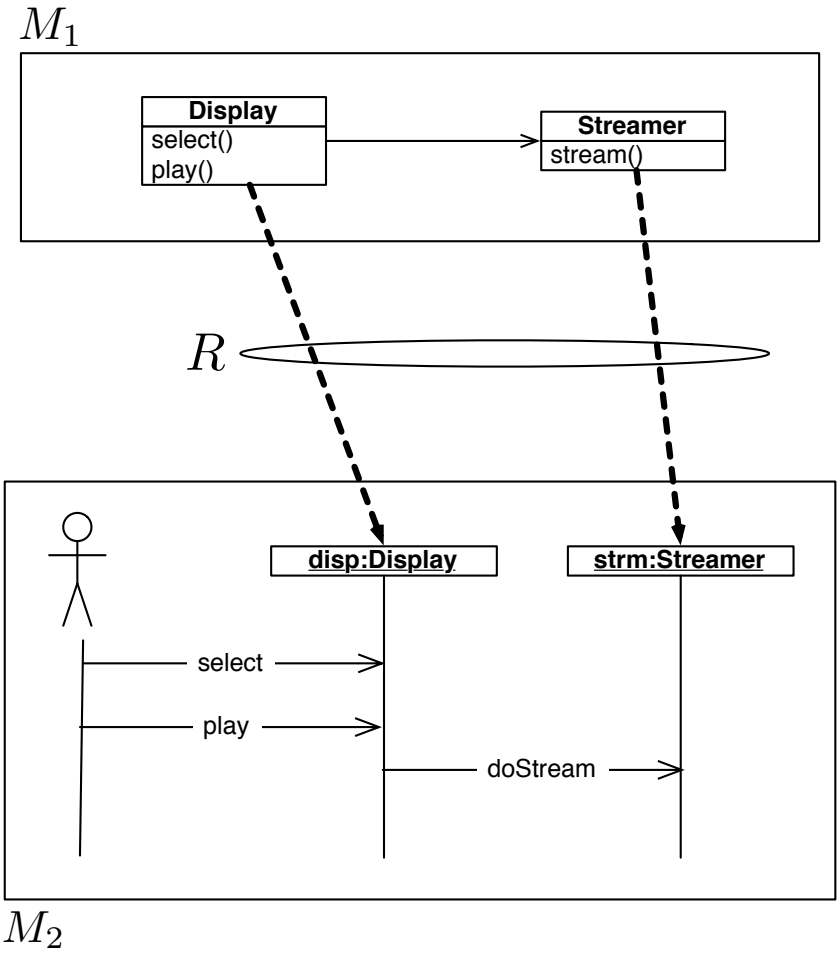


Figure 2.5: Pairwise checking of heterogeneous models

Example 2.7 (consistency checking via combination) Suppose that the specifications S_1, S_2 in Figure 2.6 are different perspectives on the terms used for referring to children at different ages. The relationship R is a mapping between the vocabularies of the two specifications, hypothesizing that (1) Child in S_1 is a more general concept than Baby in S_2, and (2) Newborn in S_2 is a more general concept than Neonate in S_1. For simplicity, we assume absence of homonyms (i.e., terms that are lexically the same but have different meanings). Otherwise, we would have had to introduce a separate namespace for each specification and explicitly specify all vocabulary mappings, even for lexically equivalent terms.

To check the consistency of the two specifications, we construct a combined specifi-

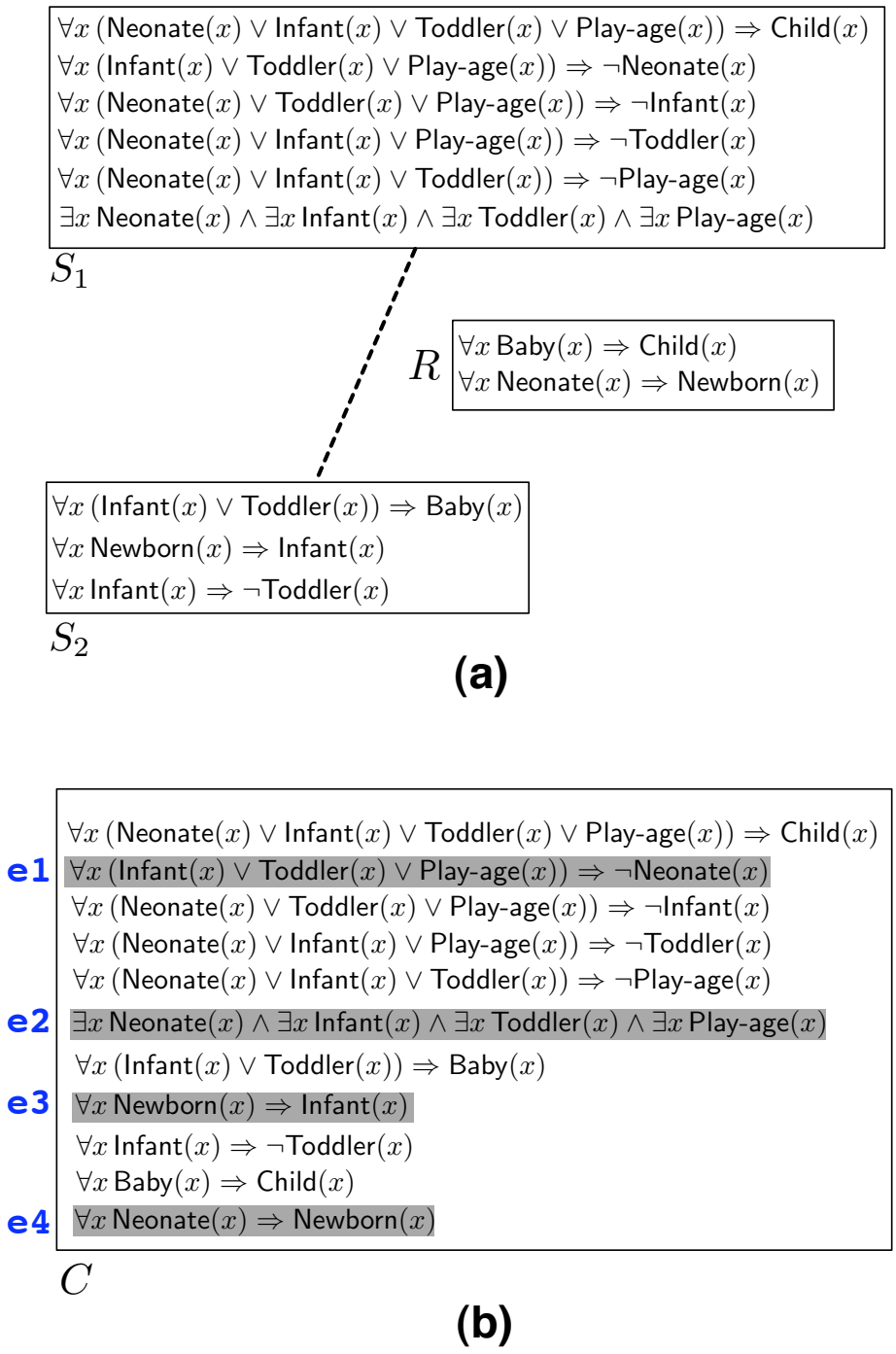


Figure 2.6: Consistency checking of specifications via conjunction

cation, shown in Figure 2.6(b), by taking the conjunction C of S_1, S_2, and R, and then check C for logical consistency. It turns out that C is unsatisfiable and hence inconsistent. A minimally unsatisfiable set of formulas is shaded gray in Figure 2.6(b): pick an

element x_0 for which we have $\mathsf{Neonate}(x_0)$. The existence of such an element is necessary to satisfy e2. By e4, we have $\mathsf{Newborn}(x_0)$, and then by e3 – $\mathsf{Infant}(x_0)$. Now, we get a contradiction by e1, because we cannot have both $\mathsf{Neonate}(x_0)$ and $\mathsf{Infant}(x_0)$.

Consistency checking via combination has the advantage that it enables detection of *global* inconsistencies (van Lamsweerde *et al.*, 1998; Nuseibeh *et al.*, 2001). For example, consider the following three specifications expressed in first order logic:

$$\begin{aligned} S_1 &= \{\ \forall x\, P(x)\ \} \\ S_2 &= \{\ \exists x\, P(x), \\ &\qquad \forall x\, Q(x)\ \} \\ S_3 &= \{\ \forall x\, P(x) \Rightarrow \neg Q(x)\ \} \end{aligned}$$

The global conjunction of S_1, S_2, S_3 is inconsistent, whereas all pairwise conjunctions are consistent.

To enable consistency checking via combination, one first needs to define an operator for putting descriptions together. In logical specification languages, both the specifications and their relationships are usually described in the same logic, as illustrated by Example 2.7. Hence, the combination can be defined using some form of logical conjunction (Zave & Jackson, 1993).

In contrast, in conceptual modelling languages like goal, entity-relationship, class and sequence diagrams, the models are expressed using graphical notations, and the relationships – using simple tuples of the form (e_i, e_j), where e_i and e_j are elements of different models (Spanoudakis & Zisman, 2001). Here, the exact definition of what it means for two elements e_i and e_j to be mapped is left implicit; and, in different contexts, these mappings could mean equivalence, synchronization, instantiation, refinement, implementation, and so on. As a result, any attempt to combine a set of conceptual models has to account for the (implicit) semantics of the relationships between the models.

When element correspondences denote equivalence, the combination operation coincides with model merging. Thus, provided with a merge operator, one can check the

global consistency of a set of models interrelated by equivalence mappings. This is exactly what we do in Chapter 5 by utilizing the merge operator defined in Chapter 4.

2.2.4 Expressive Power Considerations

Given the choice, developers would always rather describe their desired properties in convenient-to-use and highly expressive logics. But, such flexibility comes at a cost:

Checking logical consistency becomes undecidable if the expressive power of the logic being used is not sufficiently restricted. Undecidability effectively means that full automation is impossible. The main goal in many areas of knowledge representation and reasoning, e.g., Description Logics (Baader *et al.*, 2003), is to identify fragments of formal logics that are decidable. If decidability is established, one has to also consider the complexity of the reasoning procedure for the logic in question to determine whether the logic scales to the goals and tasks at hand.

Alternatively, one can restrict logical consistency checking to structures of bounded size. That is, instead of trying to produce a general proof that a specification S is logically inconsistent, one could show that S cannot be satisfied by any structure with $\leq k$ elements, where k is a fixed number. The advantage of this so-called "bounded" approach is that it allows logical consistency to be verified by exhaustive search. One of the most notable and useful tools that implements this approach is the Alloy Analyzer (Jackson, 2006). Alloy provides a rich specification language based on first order logic, and can check the logical consistency of specifications written in this language for universes of bounded size. Despite their demonstrated usefulness for exploration, applying Alloy and similar tools is still a challenge when the bound (k) is large.

Unlike the case for checking logical consistency, undecidability is usually not an issue for conformance checking: For many interesting variants of temporal and first order logics, conformance checking over finite structures is decidable[5]. The main factor to be

[5]In the formal verification literature, conformance checking is known as *model checking*.

considered here is the computational complexity of conformance checking, which as one may expect, is inversely related with the expressive power of the logic being used. Finding the right balance between scalability and expressiveness in conformance checking is an important engineering question that must be addressed based on the size of the models and the types of analyses involved.

2.2.5 Inconsistency Management

Management of inconsistency involves a number of activities, of which consistency checking is one. Other important activities in inconsistency management include (Nuseibeh *et al.*, 2001):

- Inconsistency *classification*, focusing on identifying the kinds of inconsistencies detected. Inconsistencies may be classified according to their causes, or according to pre-defined kinds prescribed by developers.

- Inconsistency *handling*, focusing on acting in the presence of inconsistencies. For example, when an inconsistency is detected, the appropriate response may be to ignore, tolerate, resolve, or circumvent it.

- Inconsistency *measurement*, focusing on calculating the impact of inconsistencies on different aspects of development, and prioritizing the inconsistencies according to the severity of their impact. The actions taken to handle an inconsistency typically depend on the results of inconsistency measurement.

Our work in this thesis concerns only the consistency checking activity. The background information we provided in this chapter was therefore specifically directed towards this activity. For a comprehensive overview of inconsistency management, see (Nuseibeh *et al.*, 2000; Spanoudakis & Zisman, 2001).

2.2.6 Literature Review

We briefly review the literature on automated consistency checking. A critical comparison between our consistency checking approach and other existing approaches is given in Chapter 5. Our review focuses on checking consistency *between* descriptions, i.e. *inter*-description consistency checking. The bulk of the research on this topic has been done in the requirements engineering community, motivated by the problem of inconsistency detection between requirements originating from different stakeholders.

With the rise of global software engineering (Herbsleb, 2007), applications of inter-description consistency checking are expanding beyond their original roots in requirements engineering. In particular, software design is increasingly being conducted by distributed teams located in different companies and countries. Ensuring consistency between the artifacts built by these teams is a major problem (Egyed, 2006). This makes the continuation of research on inter-description consistency checking very important for the future of software development.

One of the early approaches to consistency checking of multiple descriptions is **(Easterbrook *et al.*, 1994)** which promotes the use of explicit rules to capture the consistency relationships between different descriptions. These rules are described as first order logic formulas over description pairs. The notion of consistency used is conformance (see Section 2.2.1). A generalization of the work is provided in **(Easterbrook & Nuseibeh, 1996)** where a broader range of inconsistency management tasks, including inconsistency resolution and monitoring, are considered. This later work also provides more flexibility in defining the consistency rules. In particular, it allows the rules to be supplemented with a domain-specific strategy for identifying the overlaps between different descriptions.

(Finkelstein *et al.*, 1994) proposes an approach for checking the logical consistency of a set of related requirements viewpoints. The approach works by constructing a first order knowledge-base containing formulas representing the contents of the viewpoints and environmental information such as inter-viewpoint relations. A classical theorem prover

is then used to detect inconsistencies in the knowledge-base, under the assumption that the knowledge-base is closed, i.e., if a fact A is not in the knowledge-base then $\neg A$ holds. The approach further provides a way to resolve inconsistencies using an action-based language based on temporal logic.

(Hunter & Nuseibeh, 1998) presents a technique for reasoning about inconsistent requirements. The goal here is to enable generation of useful inferences from an inconsistent knowledge-base of requirements viewpoints. The facts in the knowledge-base are written in first order logic without existential quantification and function symbols. This makes checking logical consistency decidable. To reason in the presence of inconsistency, the approach distinguishes between inferences derivable from (1) some consistent subset of the knowledge-base, (2) all consistent subsets of the knowledge-base, and (3) the intersection of all consistent subsets. This provides a measure for the quality of each inference: the highest degree of confidence is provided by inferences of type (3), and the lowest – by inferences of type (1).

(van Lamsweerde *et al.*, 1998) identifies various kinds of inconsistency in requirements engineering, covering both the centralized and distributed development settings. The kinds of inconsistency identified are:

- *Deviation*: This corresponds to what we called non-conformance in Section 2.2.1.

- *Terminological and structural clash*: Terminological clash refers to the situation where different terms are used for expressing the same concept (synonymy), or the same term is used for expressing different concepts (homonymy). Structural clash refers to the situation where a concept is described using different structures in different descriptions. For example, if the descriptions are database schemata, the concept of "gender" might be modelled as an attribute of the entity Person in one model, and as two sub-entities, Male and Female, of Person in another model.

  Detecting terminological and structural clashes is by and large a manual process,

because this usually requires knowledge of the tacit (i.e., unstated) semantics of the descriptions. Our characterization in Section 2.2.1 concerns only the formal notions of consistency. In this thesis, we treat terminological and structural clashes as inevitable facts of distributed development, whose handling requires first-class support from the underlying modelling framework. As we argued in Chapter 1 and shall describe in more detail in Chapter 4, category theory provides an abstract solution for managing these clashes by making the relationships between descriptions explicit.

- *Conflict*: This corresponds to logical inconsistency as described in Section 2.2.1.

- *Divergence*: This captures the situation where a set of descriptions are not conflicting per se, but there exists a feasible condition, called a boundary condition, which, if conjoined with the original descriptions, leads to a conflict. Obviously, the underlying notion of inconsistency for divergence is the same as that for conflict, i.e., logical inconsistency.

In addition to providing the above classification, (van Lamsweerde *et al.*, 1998) elaborates specifically on divergence and develops two formal techniques for derivation of boundary conditions in goal-oriented requirements engineering. The first technique uses backward chaining – an AI strategy for working back from possible conclusions to the evidence supporting them. The second technique uses divergence patterns. Essentially, a divergence pattern is a set of formula templates together with a generic boundary condition. Detecting divergence between a set of logical formulas is achieved by matching the formulas to the formula templates in each divergence pattern. If a match is found, the generic boundary condition associated with the matching pattern is instantiated accordingly. In this way, divergence patterns alleviate the need to manually carry out the formal derivations required in the backward chaining technique.

(Fradet *et al.*, 1999) develops a graph-based language for describing software ar-

chitectures from multiple viewpoints, and proposes a framework for checking intra- and inter-viewpoint consistency. The approach deals with both logical consistency and conformance. In particular, it translates multiplicity constraints into a system of linear inequalities and uses satisfiability checking to ensure that the system has a solution. In addition, the approach provides a simple constraint language based on first order logic for describing process-level constraints over viewpoints and their relationships. The notion used for verifying these constraints is conformance. The approach recognizes the need for combining viewpoints before checking global constraints, but it does not specify how this combination should be performed.

(Easterbrook & Chechik, 2001) describes an approach for verifying global consistency properties of state machines. The underlying notion of consistency is behavioural conformance. The approach first constructs a merge of the source state machines and then checks the (temporal) properties of interest over the merge. The model merging component of the approach was already surveyed in Section 2.1.5. The main characteristic of the consistency checking component of the approach is its use of multiple-valued semantics for evaluation of properties. This enables toleration of inconsistency and also leads to a solution for classifying inconsistencies based on their impact. Such classification is not easily possible with classical logic, because the presence of a single contradiction results in trivialization – anything follows from $A \wedge \neg A$. Hence, all inconsistencies are treated as equally bad. Multiple-valued semantics instead permits some contradictions without the resulting trivialization of classical logic.

(Bowman *et al.*, 2002) proposes a formal framework for consistency checking of viewpoints expressed in languages like LOTOS (Language of Temporal Ordering Specifications) (Bolognesi & Brinksma, 1987) and Z (Spivey, 1992). The notion of consistency used is logical consistency. Specifically, a collection of viewpoints are consistent if there is a non-empty implementation of all the viewpoints. The approach describes several strategies for checking pairwise consistency between viewpoints and further, shows how

global consistency properties can be checked by iteratively merging viewpoints through logical unification and applying binary consistency checks.

(Nentwich *et al.*, 2003) develops an end-to-end framework, called xlinkit, for consistency checking of distributed XML documents. The framework includes a document management mechanism, a language based on first order logic for expressing consistency rules, and an engine for checking documents against these rules and generating diagnostics. The notion of consistency used in xlinkit is conformance. Consistency rules in xlinkit do not directly reference where the elements indicated in the rules should be retrieved from. Hence, the rules can be applied to different sets of documents. Another characteristic of xlinkit is that it does not use Boolean semantics for evaluating consistency rules. Instead, it provides an extended semantics in terms of hyperlinks. This enables linking consistent elements if a consistency rule holds, and linking inconsistent elements otherwise. The semantics of linkage is defined such that only elements that have directly contributed to an inconsistency are included in the resulting diagnostics. For example, in the formula $a \wedge b$, if a holds and b does not, the formula fails due to b and only due to b, and hence b would be included in the diagnostics whereas a would not.

(Egyed, 2006) presents a method for checking consistency between heterogeneous UML diagrams. The work focuses on well-formedness constraints and uses (structural) conformance as the underlying notion of consistency. Consistency rules are written in an imperative programming language. The behaviour of each rule is dynamically profiled during its evaluation to keep track of the elements accessed by the rule. The resulting profiling information is used to establish a correlation among elements, consistency rules, and inconsistencies. Based on this correlation, the approach can decide when to re-evaluate each consistency rule and when to display inconsistencies.

(Paige *et al.*, 2007) investigates the problem of verifying consistency constraints imposed by the UML meta-model (Unified Modeling Language, 2003). These constraints can be over either individual UML diagrams or pairs of (heterogeneous) UML diagrams.

Two consistency checking techniques are explored, one using a general-purpose theorem prover and the other – an imperative object-oriented programming language. The overall observation made by the work is the following: Most meta-model constraints can be verified via model checking (conformance checking, in our terminology). However, there are certain constraints whose verification involves checking logical consistency. For example, UML requires that each message in a sequence diagram should correspond to an enabled routine call, i.e., a routine call for which the pre-conditions are true. A pre-condition of interest can be true only if the post-conditions of previous routine calls combine to enable the pre-condition. Therefore, the pre-condition has to be logically consistent with the post-conditions of previous calls.

2.3 Summary

In this chapter, we discussed the conceptual background for the thesis and reviewed the literature relevant to our contributions. An overall understanding of the material in this chapter is assumed throughout the remainder of the thesis.

Chapter 3

Mathematical Foundations

This chapter is intended to serve as a detailed reference for the mathematical machinery in our work. Subsequent chapters will provide a less formal exposition of the relevant mathematical concepts as needed. A reader more interested in the practical applications of the thesis may wish to skip this chapter. References to this chapter in later chapters are limited to side remarks.

In this chapter, we provide a formal introduction to algebras, graphs, lattice theory, category theory, fuzzy structures, and formal logic. The topics covered in Sections 3.1 – 3.4 and 3.7 are standard and can be found in mathematical textbooks. Our presentation of these topics is narrowed down to the aspects needed for the purposes of this thesis. The results presented in Section 3.5 on fuzzy sets are well-known in the broader literature on toposes (see, e.g., (Barr & Wells, 1984)); however, we have been unable to find a reference that covers these results in a way that is easily accessible to an audience not familiar with topos theory. Our presentation in Section 3.5 spells out the required details in a concise and easy-to-understand manner. Categories of fuzzy graphs, as introduced in Section 3.6, are a contribution of the author (Sabetzadeh, 2003; Sabetzadeh & Easterbrook, 2003).

3.1 Many-Sorted Sets and Algebras

In this section, we review the basic definitions concerning many-sorted algebras. The notation we use here is based on (Goguen *et al.*, 1987; Sannella & Tarlecki, 1999).

Definition 3.1 (many-sorted set) Let S be a set of sorts. An ***S-sorted set*** is an S-indexed family of sets $X = \langle X_s \rangle_{s \in S}$, which is empty if X_s is empty for all $s \in S$.
For S-sorted sets $X = \langle X_s \rangle_{s \in S}$ and $Y = \langle Y_s \rangle_{s \in S}$:

1. $X \cup Y = \langle X_s \cup Y_s \rangle_{s \in S}$ (***Union***)
2. $X \cap Y = \langle X_s \cap Y_s \rangle_{s \in S}$ (***Intersection***)
3. $X \times Y = \langle X_s \times Y_s \rangle_{s \in S}$ (***Cartesian Product***)
4. $X \uplus Y = \langle X_s \uplus Y_s \rangle_{s \in S}$ (***Disjoint Union***)
5. $X \subseteq Y \iff (\forall s \in S : X_s \subseteq Y_s)$ (***Inclusion***)
6. $X = Y \iff (X \subseteq Y \wedge Y \subseteq X)$ (***Equality***)

Definition 3.2 (many-sorted function) An ***S-sorted function*** $f \colon X \to Y$ is an S-indexed family of functions $f = \langle f_s \colon X_s \to Y_s \rangle_{s \in S}$. We respectively call X and Y the ***source*** and the ***target*** of f.

Definition 3.3 (function composition) If $f \colon X \to Y$ and $g \colon Y \to Z$ are S-sorted functions, then their ***composition*** $g \circ f : X \to Z$ is the S-sorted function defined by $(g \circ f)_s(x) = g_s(f_s(x))$ for $s \in S$ and $x \in X_s$.

Definition 3.4 (binary relation) An S-sorted ***binary relation*** on X, written $R \subseteq X \times X$, is an S-indexed family of binary relations $R = \langle R_s \subseteq X_s \times X_s \rangle_{s \in S}$. For $s \in S$ and $x, y \in X_s$, xR_sy, also written xRy, means $\langle x, y \rangle \in R_s$.

Definition 3.5 (equivalence relation) Let R be an S-sorted relation on X. R is an S-sorted ***equivalence*** on X if it is reflexive (xR_sx), symmetric ($xR_sy \implies yR_sx$), and transitive ($xR_sy \wedge yR_sz \implies xR_sz$) for each sort s. The symbol $\equiv$ is used for (S-sorted) equivalence relations.

Definition 3.6 (quotient set) Let $\equiv$ be an S-sorted equivalence on X. If $s \in S$ and $x \in X_s$, then the ***equivalence class of*** x ***modulo*** $\equiv$ is the set $[x]_{\equiv_s} = \{y \in X_s \mid x \equiv_s y\}$. The ***quotient*** of X modulo $\equiv$, denoted $X/\equiv$, is the S-sorted set $\langle\{[x]_{\equiv_s} \mid x \in X_s\}\rangle_{s\in S}$.

Definition 3.7 (many-sorted signature) An S-sorted ***signature*** is a pair $\Sigma = \langle S, \Omega\rangle$, where S is a set of sorts and Ω is an $(S^* \times S)$-sorted set of operations. By S^* we mean the set of all finite strings from S, including the empty string λ. We call f an ***operation symbol*** of ***arity*** $s_1 \ldots s_n$ and of ***result sort*** s if $f \in \Omega_{s_1\ldots s_n,s}$.

Definition 3.8 (many-sorted algebra) Let $\Sigma = \langle S, \Omega\rangle$ be a signature. A **Σ-*algebra*** A consists of an S-sorted set $|A|$ of ***carrier sets***; and for each $f \in \Omega_{s_1\ldots s_n,s}$, a function

$$f_A \colon |A|_{s_1} \times \cdots \times |A|_{s_n} \to |A|_s.$$

For an operator symbol $f \in \Omega_{\lambda,s}$, the function $f_A \in |A|_s$ (also written $f_A :\to |A|_s$) is a ***constant*** of A of sort s.

Definition 3.9 (many-sorted homomorphism) Let $\Sigma = \langle S, \Omega\rangle$ be a signature and let A and B be Σ-algebras. A **Σ-*homomorphism*** $h \colon A \to B$ is an S-sorted function $h \colon |A| \to |B|$ such that if $f \in \Omega_{s_1\ldots s_n,s}$ and if $a_1 \in |A|_{s_1}, \ldots, a_n \in |A|_{s_n}$, then

$$h_s(f_A(a_1, \ldots, a_n)) = f_B(h_{s_1}(a_1), \ldots, h_{s_n}(a_n)).$$

3.2 Graphs and Graph Homomorphisms

In this section, we introduce a specific kind of directed graph commonly used in the algebraic literature that deals with graph-based structures. The importance of graphs in

this thesis is two-fold: On the one hand, graphs are the basis for defining diagrams in category theory (see Section 3.4). And, on the other hand, graphs and their mappings are core to the model merging approach developed in Chapter 4.

Definition 3.10 (graph) A ***graph*** is a quadruple $G = (N, E, \mathsf{source}_G, \mathsf{target}_G)$ where N is a set of nodes, E is a set of edges, and $\mathsf{source}_G, \mathsf{target}_G : E \to N$ are functions respectively giving the source and the target for each edge. A ***graph homomorphism*** from a graph $G = (N, E, \mathsf{source}_G, \mathsf{target}_G)$ to a graph $G' = (N', E', \mathsf{source}_{G'}, \mathsf{target}_{G'})$ is a pair of functions $h = \langle h_{\text{node}} : N \to N', h_{\text{edge}} : E \to E' \rangle$ such that:

$$\boxed{h_{\text{node}} \circ \mathsf{source}_G = \mathsf{source}_{G'} \circ h_{\text{edge}}} \quad \textbf{and} \quad \boxed{h_{\text{node}} \circ \mathsf{target}_G = \mathsf{target}_{G'} \circ h_{\text{edge}}}$$

Example 3.11 Figure 3.1 illustrates four graphs A, B, C, and D along with four possible homomorphisms $h : A \to B$, $k : B \to B$, $l : A \to C$, and $p : B \to D$.

An equivalent definition for graph and graph homomorphism can be given by noticing that a graph is a (two-sorted) algebra and a graph homomorphism is a (two-sorted) homomorphism:

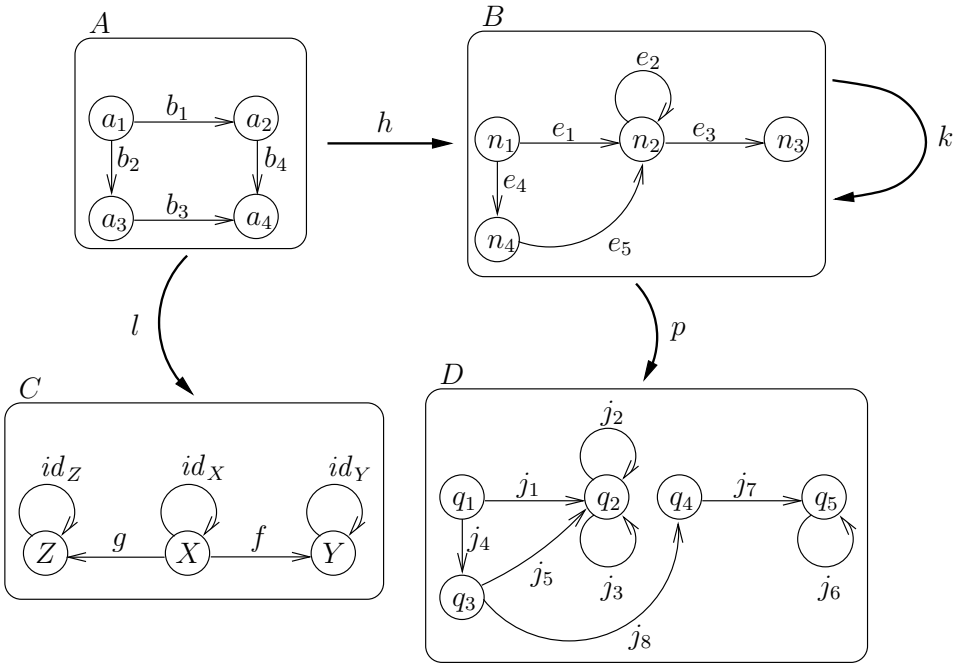
Definition 3.12 A ***graph*** is a Σ_G-algebra and a ***graph homomorphism*** is a Σ_G-homomorphism where $\Sigma_G = \langle S_G, \Omega_G \rangle$ is defined as followed:

$S_G = \{\text{node}, \text{edge}\}$;

$\Omega_{\text{edge,node}} = \{\mathsf{source}, \mathsf{target}\}$;

$\Omega_{w,s} = \emptyset$ for all other $w \in S_G^*$ and $s \in S_G$.

NOTATION We usually drop the sort subscripts of the functions that comprise a graph homomorphism and use the name of the homomorphism as an overloaded operator that acts on both nodes and edges. For example, for a node a_1, we write $h(a_1) = n_1$ instead of $h_{\text{node}}(a_1) = n_1$; and for an edge b_1, we write $h(b_1) = e_1$ instead of $h_{\text{edge}}(b_1) = e_1$.



$$h_{\text{node}} = \left\{ \begin{array}{l} a_1 \mapsto n_1, a_2 \mapsto n_2, a_3 \mapsto n_4 \\ a_4 \mapsto n_2 \end{array} \right\} \qquad h_{\text{edge}} = \left\{ \begin{array}{l} b_1 \mapsto e_1, b_2 \mapsto e_4, b_3 \mapsto e_5, \\ b_4 \mapsto e_2 \end{array} \right\}$$

$$k_{\text{node}} = \left\{ \begin{array}{l} n_1 \mapsto n_1, n_2 \mapsto n_2, n_3 \mapsto n_2, \\ n_4 \mapsto n_4 \end{array} \right\} \qquad k_{\text{edge}} = \left\{ \begin{array}{l} e_1 \mapsto e_1, e_2 \mapsto e_2, e_3 \mapsto e_2, \\ e_4 \mapsto e_4, e_5 \mapsto e_5 \end{array} \right\}$$

$$l_{\text{node}} = \left\{ \begin{array}{l} a_1 \mapsto X, a_2 \mapsto Y, a_3 \mapsto X \\ a_4 \mapsto Y \end{array} \right\} \qquad l_{\text{edge}} = \left\{ \begin{array}{l} b_1 \mapsto f, b_2 \mapsto id_X, b_3 \mapsto f, \\ b_4 \mapsto id_Y \end{array} \right\}$$

$$p_{\text{node}} = \left\{ \begin{array}{l} n_1 \mapsto q_1, n_2 \mapsto q_2, n_3 \mapsto q_2, \\ n_4 \mapsto q_3 \end{array} \right\} \qquad p_{\text{edge}} = \left\{ \begin{array}{l} e_1 \mapsto j_1, e_2 \mapsto j_2, e_3 \mapsto j_3, \\ e_4 \mapsto j_4, e_5 \mapsto j_5 \end{array} \right\}$$

Figure 3.1: Examples of graphs and graph homomorphisms

3.3 Partial Orders and Lattices

In this section, we recall some basic definitions and results from lattice theory (Birkhoff, 1979; Davey & Priestley, 2002). Lattice theory provides a unified framework for the study of ordered structures. It finds one of its major applications in fuzzy set theory (Goguen, 1974; Höhle & Rodabaugh, 1999), which we discuss in Section 3.5.

Definition 3.13 (partial order relation) A ***partial order*** $\leq$ on a set A is a binary relation on A such that the following conditions hold:

1. $\forall a \in A : a \leq a$ ***(reflexivity)***
2. $\forall a, b \in A : a \leq b \wedge b \leq a \implies a = b$ ***(anti-symmetry)***
3. $\forall a, b, c \in A : a \leq b \wedge b \leq c \implies a \leq c$ ***(transitivity)***

We call $\leq$ a ***total order*** on A if the following condition holds, as well:

4. $\forall a, b \in A : a \leq b \vee b \leq a$ ***(totality condition)***

Definition 3.14 (partially ordered set) A non-empty set with a partial order on it is called a ***partially ordered set*** or a ***poset*** for short. If the relation is a total order then the set is called a ***totally ordered set*** or more conveniently a ***chain***. In a poset A, we use the expression $a < b$ to indicate that $a \leq b$ but $a \neq b$.

Definition 3.15 (bottom and top) Let P be a poset. P has a ***bottom*** element if there exists $\bot \in P$ such that $\bot \leq x$ for all $x \in P$. Dually, P has a ***top*** element if there exists $\top \in P$ such that $x \leq \top$ for all $x \in P$.

Definition 3.16 (covering relation) Let P be a poset and let $x, y \in P$. We say x ***is covered by*** y (or y ***covers*** x), and write $x \prec y$ or $y \succ x$, if $x < y$ and $x \leq z < y$ implies $z = x$.

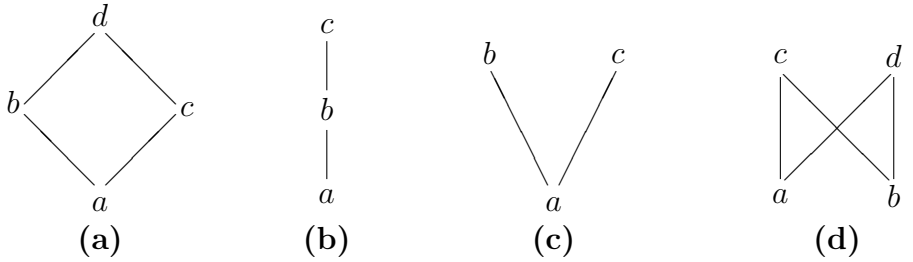


Figure 3.2: Examples of Hasse diagrams

It simply follows that for elements x, y in a finite poset P: $x < y$ if and only if there exists a finite sequence of covering relations $x = x_0 \prec x_1 \prec \ldots \prec x_n = y$. This is the underlying observation for visualizing finite posets by ***Hasse diagrams***. A Hasse diagram is a graphical rendering of a poset displayed via the covering relation of the poset with an implied upward orientation. In a Hasse diagram, the elements of a finite poset P are displayed in such a way that for every $a, b \in P$: if $a \prec b$, then b is located above a and the two elements are connected with a line segment. Figure 3.2 illustrates Hasse diagrams. In Figure 3.2(a), for example, the covering relation is: $\{a \prec b, a \prec c, b \prec d, c \prec d\}$. It can be verified that for a finite poset P, the relation $\leq$ is equal to $\prec^*$ (the closure of $\prec$). Therefore, $\leq$ can be reconstructed from the Hasse diagram corresponding to P.

Definition 3.17 (upper bound and lower bound) Let P be a poset and $A \subseteq P$. An element $p \in P$ is an ***upper bound*** (resp. ***lower bound***) of A if

$$\forall a \in A : a \leq p \qquad (\text{resp. } \forall a \in A : p \leq a)$$

Definition 3.18 (supremum and infimum) Let P be a poset and $A \subseteq P$. An element $p \in P$ is the ***least upper bound*** or the ***supremum*** of A, written **sup** A, if p is an upper bound of A, and for all upper bounds x of A, $p \leq x$. Dually, an element $p \in P$ is the ***greatest lower bound*** or the ***infimum*** of A, written **inf** A, if p is a lower bound of A, and for all lower bounds x of A, $x \leq p$.

NOTATION We write $x \sqcup y$, read as "x ***join*** y", in place of $\mathbf{sup}\{x, y\}$ when it exists; and $x \sqcap y$, read as "x ***meet*** y", in place of $\mathbf{inf}\{x, y\}$ when it exists. Similarly, we write $\bigsqcup A$ ("the ***join of*** A"), and $\bigsqcap A$ ("the ***meet of*** A") in place of $\mathbf{sup}\ A$ and $\mathbf{inf}\ A$, respectively, when they exist.

Definition 3.19 (lattice) Let P be a poset. If $x \sqcup y$ and $x \sqcap y$ exist for all $x, y \in P$, then P is called a ***lattice***. If $\bigsqcup A$ and $\bigsqcap A$ exist for all $A \subseteq P$, then P is called a ***complete lattice***.

Theorem 3.20 *(e.g., see (Davey & Priestley, 2002)) Every finite lattice is complete.*

Theorem 3.21 *(e.g., see (Davey & Priestley, 2002)) Every complete lattice has a bottom ($\bot$) and a top ($\top$) element.*

Example 3.22 In Figure 3.2, **(a)** and **(b)** are lattices, but **(c)** and **(d)** are not: in **(c)**, $\mathbf{sup}\{b, c\}$ does not exist; and in **(d)**, the set $\{a, b\}$ fails to have a least upper bound.

3.4 Category Theory

In this section, we review the basics of category theory, focusing on the aspects most relevant to our work. For a more comprehensive treatment, see (Barr & Wells, 1999).

3.4.1 Categories

Definition 3.23 (category) A category $\mathscr{C}$ consists of:

1. a collection of objects denoted $|\mathscr{C}|$;
2. for every $A, B \in |\mathscr{C}|$, a collection $Hom_{\mathscr{C}}(A, B)$ of ***morphisms*** (also called ***arrows***) from A to B. We write $f : A \to B$ when $f \in Hom_{\mathscr{C}}(A, B)$ and call A the ***source*** and B the ***target*** of f;

3. for every $A, B, C \in |\mathscr{C}|$, a ***composition*** operation

$$\circ : Hom_{\mathscr{C}}(A, B) \times Hom_{\mathscr{C}}(B, C) \to Hom_{\mathscr{C}}(A, C)$$

such that:

I. (*composition associativity*) any morphisms $f \in Hom_{\mathscr{C}}(A, B)$, $g \in Hom_{\mathscr{C}}(B, C)$, and $h \in Hom_{\mathscr{C}}(C, D)$ satisfy: $h \circ (g \circ f) = (h \circ g) \circ f$;

II. (*existence of identity*) for every $A \in |\mathscr{C}|$, there exists a morphism $id_A \in Hom_{\mathscr{C}}(A, A)$, called the ***identity*** of A, such that $f \circ id_A = f$ for any morphism $f \in Hom_{\mathscr{C}}(A, B)$; and $id_A \circ g = g$ for any morphism $g \in Hom_{\mathscr{C}}(B, A)$.

Example 3.24

- A single object together with a single morphism (which must be the identity morphism) constitutes a category, denoted **1**.
- The category of sets, denoted **Set**, has sets as objects and functions as morphisms. Notice that the source and the target of every function is explicitly given.
- For a given signature Σ, the category of Σ-algebras, denoted $\mathbf{Alg}(\Sigma)$, has Σ-algebras as objects and Σ-homomorphisms as morphisms.
- For the signature Σ_G given in Definition 3.10, $\mathbf{Alg}(\Sigma_G)$ is the category of graphs which will hereafter be denoted **Graph**.
- A poset P can be viewed as category whose objects are the elements of P; and for any $x, y \in P$ satisfying $x \leq y$, there is a unique morphism that has x as source and y as target.
- A set can be viewed as a category whose objects are the elements of the set and whose only morphisms are the identity morphisms.

Definition 3.25 (small, locally small, and large categories) A category $\mathscr{C}$ is ***small*** if $|\mathscr{C}|$ is a set and for any $A, B \in |\mathscr{C}|$, the collection $Hom_{\mathscr{C}}(A, B)$ is a set, as well. A category is ***locally small*** if $Hom_{\mathscr{C}}(A, B)$ is a set for any $A, B \in |\mathscr{C}|$. If a category is not small, then it is said to be ***large***.

Remark 3.26 All the categories referred to in this thesis are locally small.

3.4.2 Functors

Definition 3.27 (functor) A functor $F : \mathscr{C} \to \mathscr{D}$ consists of:

- A function $F_{Obj} : |\mathscr{C}| \to |\mathscr{D}|$;
- For every $A, B \in |\mathscr{C}|$, a function $F_{A,B} : Hom_{\mathscr{C}}(A, B) \to Hom_{\mathscr{D}}\big(F_{Obj}(A), F_{Obj}(B)\big)$.

such that:

1. Identities are preserved: $F_{A,A}(id_A) = id_{F_{Obj}(A)}$ for every $A \in |\mathscr{C}|$.
2. Composition is preserved: $F_{A,C}(g \circ f) = F_{B,C}(g) \circ F_{A,B}(f)$ for all morphisms $f : A \to B$ and $g : B \to C$ in $\mathscr{C}$.

Example 3.28 (identity functor) For a category $\mathscr{C}$, there exists a functor $I_{\mathscr{C}} : \mathscr{C} \to \mathscr{C}$, known as the ***identity functor*** on $\mathscr{C}$, that maps every object and morphism in $\mathscr{C}$ to itself.

Example 3.29 (Cartesian product functor) There is functor $T : \mathbf{Set} \to \mathbf{Set}$, known as the ***Cartesian product functor***, that maps every set N to $N \times N$ and every function $f : N \to N'$ to $f \times f : N \times N \to N' \times N'$ where $f \times f$ is the function such that $(x, y) \overset{f \times f}{\longmapsto} \big(f(x), f(y)\big)$ for all $x, y \in N$.

Definition 3.30 (category of locally small categories) The category of locally small categories, denoted **Cat**, has locally small categories as objects and functors as morphisms. If $F : \mathscr{A} \to \mathscr{B}$ and $G : \mathscr{B} \to \mathscr{C}$ are **Cat**-morphisms (i.e., functors), $G \circ F : \mathscr{A} \to \mathscr{C}$ is defined as follows:

- $(G \circ F)_{Obj} = G_{Obj} \circ F_{Obj}$;
- $(G \circ F)_{A,B} = G_{F(A),F(B)} \circ F_{A,B}$ for all $A, B \in |\mathscr{A}|$.

The identity morphism for every $\mathscr{A} \in |\mathbf{Cat}|$ is the identity functor $I_{\mathscr{A}} : \mathscr{A} \to \mathscr{A}$.

Example 3.31 (underlying graph functor) Let $\mathscr{C}$ be a category. By forgetting how morphisms in $\mathscr{C}$ are composed and forgetting which morphisms are the identities, we obtain the ***underlying graph*** of $\mathscr{C}$. This yields a functor $U : \mathbf{Cat} \to \mathbf{Graph}$ that maps every $\mathscr{C} \in |\mathbf{Cat}|$ to the underlying graph of $\mathscr{C}$ and every **Cat**-morphism F to the graph homomorphism induced by F. Notice that, by definition, every functor is a graph homomorphism but the converse is not true.

3.4.3 Diagrams

Definition 3.32 (diagram) Let $\mathscr{C}$ be a category and let G be a graph. A ***diagram*** of ***shape*** G in $\mathscr{C}$ is a graph homomorphism $D : G \to U(\mathscr{C})$, where $U : \mathbf{Cat} \to \mathbf{Graph}$ is the underlying graph functor (see Example 3.31).

Example 3.33 In Figure 3.1, graph C can be thought of as the underlying graph of a category $\mathscr{C}$ with objects X, Y, Z and identities id_X, id_Y, id_Z. The non-identity morphisms are $f : X \to Y$ and $g : X \to Z$. The graph homomorphism $l : A \to C$ in Figure 3.1 identifies the following diagram in $\mathscr{C}$:

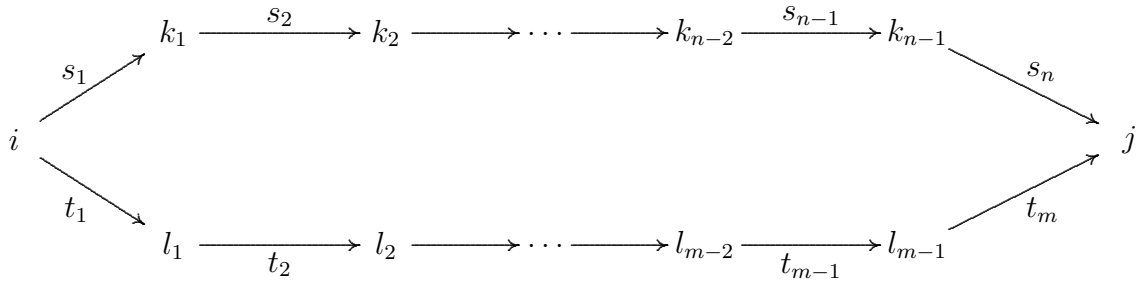
$$\begin{array}{ccc} X & \xrightarrow{f} & Y \\ {\scriptstyle id_X}\downarrow & & \downarrow{\scriptstyle id_Y} \\ X & \xrightarrow[f]{} & Y \end{array}$$

The shape graph for the above diagram is graph A as shown in Figure 3.1.

Definition 3.34 (finite diagram) A diagram is said to be ***finite*** if its shape graph is finite, that is, if its shape graph has a finite number of nodes and edges.

Definition 3.35 (discrete diagram) A diagram is said to be ***discrete*** if its shape graph has no edges.

Definition 3.36 (commutative diagram) A diagram $D : G \to U(\mathscr{C})$ is said to ***commute*** (or be ***commutative***) if for each pair of nodes i, j in G and any two paths:



from i to j in G, we have:

$$D(s_n) \circ D(s_{n-1}) \cdots \circ D(s_2) \circ D(s_1) = D(t_m) \circ D(t_{m-1}) \circ \cdots \circ D(t_2) \circ D(t_1).$$

3.4.4 Basic Category-Theoretic Definitions

Let $\mathscr{C}$ be an arbitrary category.

Definition 3.37 (isomorphism) A $\mathscr{C}$-morphism $f : A \to B$ is an ***isomorphism*** if there exists a morphism $f^{-1} : B \to A$ such that $f^{-1} \circ f = id_A$ and $f \circ f^{-1} = id_B$. The morphism f^{-1} is called the ***inverse*** of f; and objects A and B are called ***isomorphic***.

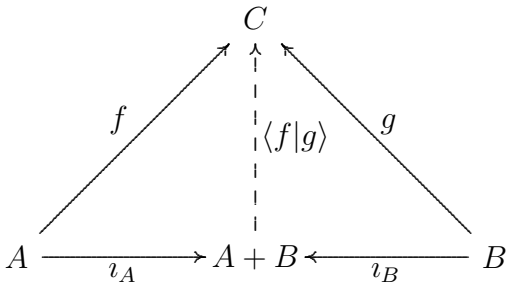
Remark 3.38 (isomorphic categories) Categories $\mathscr{A}$ and $\mathscr{B}$ are said to be isomorphic if there exists an isomorphism $F : \mathscr{A} \to \mathscr{B}$ in **Cat**.

Definition 3.39 (initial object) An object $\mathbf{0} \in |\mathscr{C}|$ is ***initial*** if for every $A \in |\mathscr{C}|$, there exists a unique morphism $\langle\rangle : \mathbf{0} \to A$.

Example 3.40 (initial objects in Set and Graph) The initial object in **Set** (resp. **Graph**) is the empty set (resp. the empty graph).

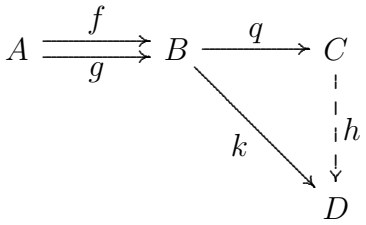
NOTATION In the rest of this section, a dashed morphism in a diagram indicates the uniqueness of that morphism.

Definition 3.41 (binary coproduct) A ***binary coproduct*** of $A, B \in |\mathscr{C}|$ is an object $A+B \in |\mathscr{C}|$ together with a pair of morphisms $\imath_A : A \to A+B$ and $\imath_B : B \to A+B$, called the ***injection*** morphisms, such that for any $C \in |\mathscr{C}|$ and pair of morphisms $f : A \to C$ and $g : B \to C$ there is a unique morphism $\langle f|g\rangle : A + B \to C$ making the following diagram commute:



Example 3.42 (binary coproducts in Set) The (canonical) binary coproduct of two sets A and B is the set $A \uplus B$ together with the obvious injections $\imath_A : A \to A \uplus B$ and $\imath_B : B \to A \uplus B$.

Definition 3.43 (coequalizer) A ***coequalizer*** of a pair of parallel $\mathscr{C}$-morphisms $f : A \to B$ and $g : A \to B$ is an object $C \in |\mathscr{C}|$ together with a morphism $q : B \to C$ such that $q \circ f = q \circ g$ and for any morphism $k : B \to D$ satisfying $k \circ f = k \circ g$, there is a unique morphism $h : C \to D$ such that $h \circ q = k$.



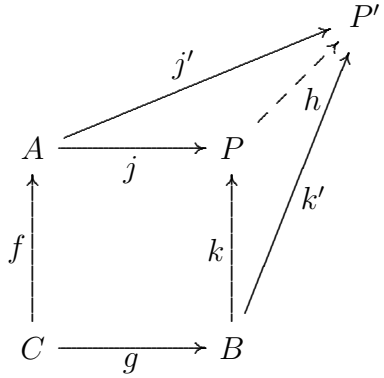
Example 3.44 (coequalizers in Set) Let $f : A \to B$ and $g : A \to B$ be a pair of **Set**-morphisms (i.e., functions) and let $R = \big\{\big(f(a), g(a)\big) \mid a \in A\big\}$. Assuming $\equiv_R$ is the smallest equivalence relation that includes R, the (canonical) coequalizer of f and g is

$B/\equiv_R$ (i.e., the quotient of B modulo $\equiv_R$) together with the function $q : B \to B/\equiv_R$ such that $q(b) = [b]_{\equiv_R}$ for all $b \in B$.

Remark 3.45 It is easy to verify that for a binary relation R on a (single-sorted) set S, the smallest equivalence relation that includes R, denoted $\equiv_R$, can be constructed as follows: consider an *undirected graph* G in which the set of nodes is S, and for any nodes x, y in G, there is an undirected edge between x and y if and only if $(x, y) \in R$. Then, $B/\equiv_R$ will be the set of G's connected components. Thus, for any $x, y \in S$: $x \equiv_R y$ if and only if x and y belong to the same connected component of G.

Definition 3.46 (pushout) A ***pushout*** of a pair of $\mathscr{C}$-morphisms $f : C \to A$ and $g : C \to B$ is an object $P \in |\mathscr{C}|$ together with a pair of morphisms $j : A \to P$ and $k : B \to P$ such that:

- $j \circ f = k \circ g$
- for any $P' \in |\mathscr{C}|$ and pair of morphisms $j' : A \to P'$ and $k' : B \to P'$ satisfying $j' \circ f = k' \circ g$, there is a unique morphism $h : P \to P'$ such that the following diagram commutes:



Remark 3.47 (construction of pushouts) A pushout of any pair of $\mathscr{C}$-morphisms with common source can be constructed if every pair of $\mathscr{C}$-objects has a coproduct and every pair of parallel $\mathscr{C}$-morphisms has a coequalizer: let $f : C \to A$ and $f : C \to B$ be a pair of morphisms and let $A + B \in |\mathscr{C}|$ together with the injections $\imath_A : A \to A + B$

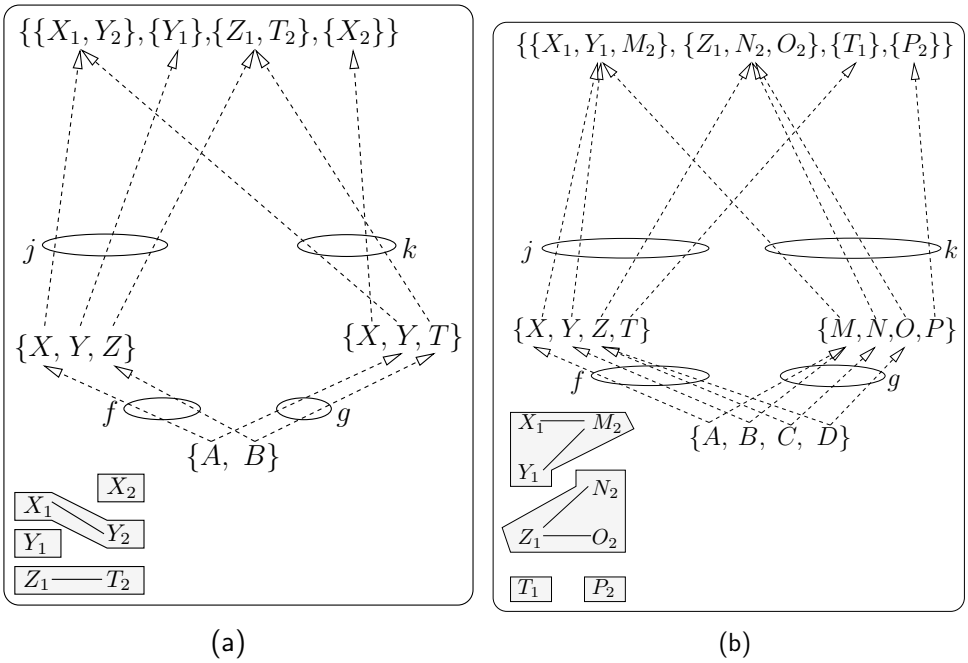
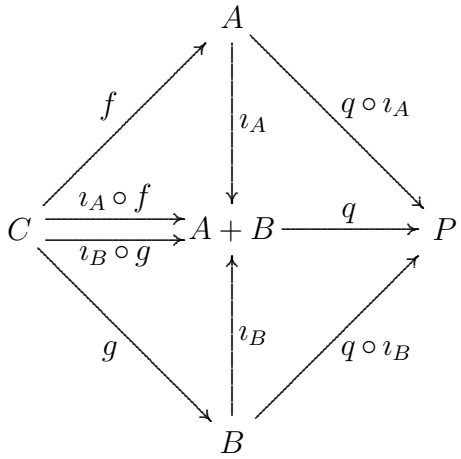


Figure 3.3: Pushout examples in **Set**

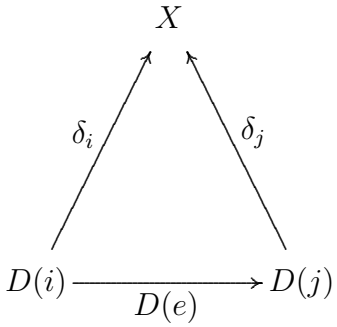
and $\imath_B : B \to A + B$ be a binary coproduct of A and B. Let $P \in |\mathscr{C}|$ together with a morphism $q : A + B \to P$ be a coequalizer of $\imath_A \circ f$ and $\imath_B \circ g$. It can be verified that the outer square in the following diagram is indeed a pushout square (Ehrig & Pfender, 1972):



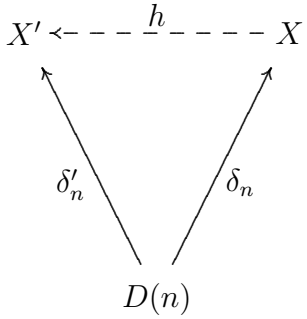
Example 3.48 (Pushouts in Set) Figure 3.3 shows two examples of pushout computation in **Set**. The maps corresponding to the morphisms f, g, j, and k of the pushout square have been marked in both examples. The figure also illustrates how the required coequalizer for each example has been computed (see Remark 3.45).

3.4.5 Colimits

Definition 3.49 (cocone and colimit) Let D be a diagram of shape G in a category $\mathscr{C}$ and let N and E denote the set of G's nodes and edges, respectively. A ***cocone*** δ over D is a $\mathscr{C}$-object X together with a family of $\mathscr{C}$-morphisms $\langle \delta_n : D(n) \to X \rangle_{n \in N}$ such that for every edge $e \in E$ with $\mathsf{source}_G(e) = i$ and $\mathsf{target}_G(e) = j$, the following diagram commutes:



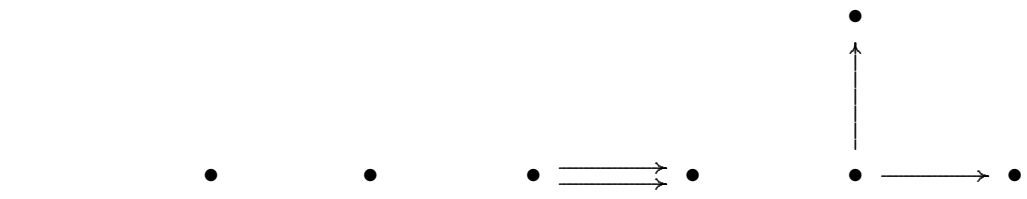
A ***colimit*** of D is a cocone $\langle \delta_n : D(n) \to X \rangle_{n \in N}$ such that for any cocone $\langle \delta'_n : D(n) \to X' \rangle_{n \in N}$, there is a unique morphism $h : X \to X'$ that makes the following diagram commute for all $n \in N$:



Theorem 3.50 *(e.g., see (Barr & Wells, 1999)) Colimits are unique up to isomorphism.*

Definition 3.51 (cocompleteness) A category $\mathscr{C}$ is (finitely) cocomplete if every (finite) diagram in $\mathscr{C}$ has a colimit.

Example 3.52 Initial objects, binary coproducts, coequalizers, and pushouts are colimits over diagrams of shapes **(a)**, **(b)**, **(c)**, and **(d)** respectively:



(a) **(b)** **(c)** **(d)**

The shape graph **(b)** suggests the generalization of the definition of binary coproduct in the following sense:

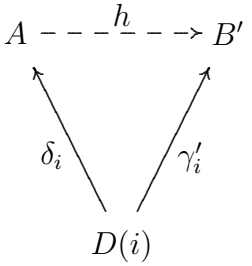
Definition 3.53 (coproduct) The colimit of a diagram D is called a ***coproduct*** if D is discrete.

Lemma 3.54 *A category with an initial object and binary coproducts has all finite coproducts.*

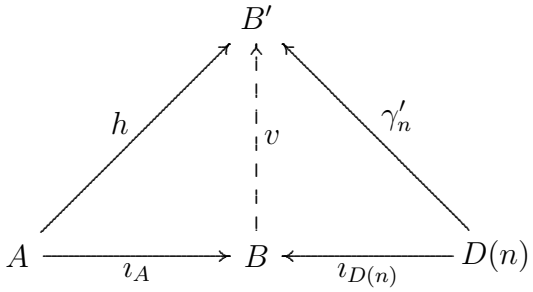
Proof (Rydeheard & Burstall, 1988) Let $D : G \to U(\mathscr{C})$ be a finite *discrete* diagram in a category $\mathscr{C}$ and let N denote the set of G's nodes. If N is empty, then the coproduct is the initial object; otherwise, there exists some $n \in N$. Let D' be the same diagram as D with node n removed from its shape graph. Inductively, construct the coproduct $\langle \delta_i : D'(i) \to A \rangle_{i \in N \setminus \{n\}}$; and further let B together with $\imath_A : A \to B$ and $\imath_{D(n)} : D(n) \to B$ be a binary coproduct of A and $D(n)$.

Construct a cocone $\langle \gamma_i : D(i) \to B \rangle_{i \in N}$ by letting $\gamma_n = \imath_{D(n)}$ and $\gamma_i = \imath_A \circ \delta_i$ for $i \neq n$. We claim that $\langle \gamma_i : D(i) \to B \rangle_{n \in N}$ is a coproduct of D: suppose $\langle \gamma'_i : D(i) \to B' \rangle_{i \in N}$ is a cocone over D. Then, $\langle \gamma'_i \rangle_{i \in N \setminus \{n\}}$ is a cocone over D'. Since $\langle \delta_i \rangle_{i \in N \setminus \{n\}}$ is a colimiting cocone, there is a unique morphism $h : A \to B'$ such that the following diagram commutes

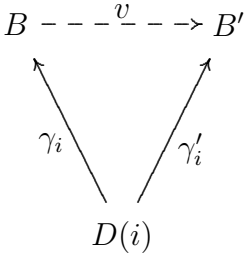
for all $i \in N \backslash \{n\}$:



Now, by the definition of binary coproduct, there is a unique morphism $v : B \to B'$ such that the following diagram commutes:



It follows from the previous two diagrams that $v : B \to B'$ makes the following diagram commute for all $i \in N$:



The uniqueness conditions on h and v ensure that v is the only such morphism. ■

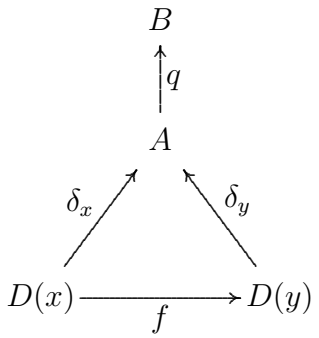
Among the numerous results on (finite) cocompleteness of categories, the following theorem is of particular interest because its constructive proof leads to a straightforward algorithm for computing (finite) colimits (see Remark 3.58).

Theorem 3.55 *A category $\mathscr{C}$ is finitely cocomplete if it has an initial object, binary coproducts of all object pairs, and coequalizers of all parallel morphism pairs.*

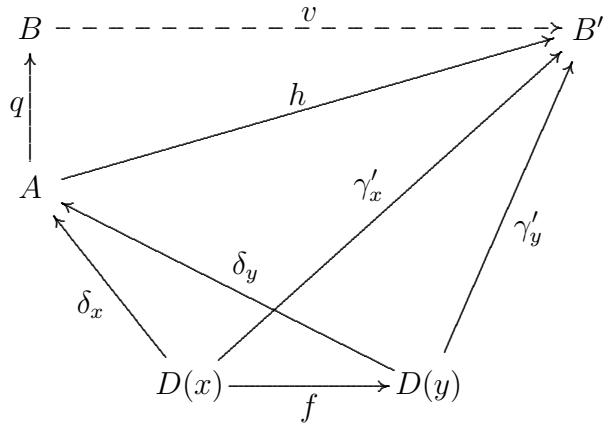
Proof (Rydeheard & Burstall, 1988) Let $D : G \to U(\mathscr{C})$ be a finite diagram in a category $\mathscr{C}$. If D is discrete then the colimit of D is the coproduct constructed in Lemma 3.54;

otherwise, suppose $e : x \to y$ is an edge in G with associated morphism $f : D(x) \to D(y)$ in $\mathscr{C}$. Let D' be the same diagram as D with edge e removed from its shape graph. Inductively, construct the colimiting cocone $\langle \delta_i : D'(i) \to A \rangle_{i \in N}$ on D' where N is the set of nodes in D's shape graph (notice that the shape graphs of D and D' have the same set of nodes).

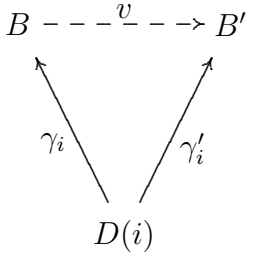
Now, consider the parallel pair of morphisms $\delta_x : D(x) \to A$ and $\delta_y \circ f : D(x) \to A$ and let B together with $q : A \to B$ be a coequalizer of δ_x and $\delta_y \circ f$.



Construct a cocone $\langle \gamma_i : D(i) \to B \rangle_{i \in N}$ over D by letting $\gamma_i = q \circ \delta_i$. We claim that $\langle \gamma_i \rangle_{i \in N}$ is colimiting: suppose $\langle \gamma'_i : D(i) \to B' \rangle_{i \in N}$ is a cocone over D. Then, $\langle \gamma' \rangle_{i \in N}$ is a cocone over D', as well. Since $\langle \delta_i : D'(i) \to A \rangle_{i \in N}$ is colimiting, there is a unique morphism $h : A \to B'$ such that for all $i \in N : h \circ \delta_i = \gamma'_i$. Hence, we have: $h \circ \delta_x = \gamma'_x = \gamma'_y \circ f = h \circ \delta_y \circ f$, and by the definition of coequalizer, there is a unique morphism $v : B \to B'$ such that $v \circ q = h$.



Therefore, by letting $\gamma_i = q \circ \delta_i$, the following diagram commutes for all $i \in N$:



The uniqueness of v follows from the uniqueness of q and h. ■

Example 3.56 Set is finitely cocomplete (see Examples 3.40, 3.42, and 3.44).

Remark 3.57 It can also be shown that $\mathbf{Alg}(\Sigma)$ is finitely cocomplete for any signature Σ, but the proof is more difficult. The interested reader can consult standard textbooks on Categorical Algebra (such as (Borceux, 1994)) for the proof of cocompleteness in the single-sorted case. The proof for the many-sorted case is analogous.

The intuition behind colimits is that they put structures together, with nothing essentially new added, and nothing left over (Goguen, 1991). A pushout of two morphisms $f : C \to A$ and $g : C \to B$ in **Set**, for example, can be interpreted as the merge of A and B with respect to a shared part C such that only one copy of C is included in the merge. This was already illustrated in Example 3.48. More generally: "*Given a species of structure, say widgets, the result of interconnecting a system of widgets to form a super-widget corresponds to taking the colimit of the diagram of widgets in which the morphisms show how they are interconnected.*" (Goguen, 1991). This observation is the main reason for our interest in cocompleteness results.

Remark 3.58 (algorithm for computing colimits) In Figure 3.4, we show an algorithmic view of the proof of Theorem 3.55. This algorithm is the basis for our model merging technique in Chapter 4.

Algorithm. Compute-Colimit

Input: Diagram $D : G \to U(\mathscr{C})$

Output: Colimiting cocone δ over D

▷ Let N and E denote the node and edge sets of G, respectively;

▷ Let D' be the same as D but with the edges of the shape graph (i.e., G) removed;

▷ Construct the coproduct $\langle \delta_i : D'(i) \to A \rangle_{i \in N}$; */* using Lemma 3.54 */*

▷ Let $e_1, e_2, \ldots, e_m$ enumerate the elements of E;

▷ **for** $j = 1$ **to** m **do**

  ▷ Let $f = D(e_j)$, let $x = \mathsf{source}_G(e_j)$, and let $y = \mathsf{target}_G(e_j)$;

  ▷ Construct the coequalizer object B and morphism $q : A \to B$ of δ_x and $\delta_y \circ f$;

  / Now, reconstruct δ so that B replaces A as the apex. */*

  ▷ **for** every $n \in N$ **do**

    ▷ $\delta_n := q \circ \delta_n$;

Figure 3.4: Algorithm for computing colimits

For reasons that will become clear in Section 3.4.6, we are also interested in functors that preserve (finite) colimits:

Definition 3.59 (cocontinuity) A functor $F : \mathscr{C} \to \mathscr{D}$ is said to be (finitely) ***cocontinuous*** if it preserves the existing colimits of all (finite) diagrams in $\mathscr{C}$, that is, if for any (finite) diagram D in $\mathscr{C}$, the functor F maps any colimiting cocone over D to a colimiting cocone over $F(D)$.

Theorem 3.60 *If $\mathscr{C}$ is a finitely cocomplete category and if a functor $F : \mathscr{C} \to \mathscr{D}$ preserves initial objects, binary coproducts of all object pairs, and coequalizers of all parallel morphism pairs, then F is finitely cocontinuous.*

Proof Follows from Theorem 3.55. ∎

3.4.6 Comma Categories [1]

Definition 3.61 (comma category) Let $\mathscr{A}$, $\mathscr{B}$, and $\mathscr{C}$ be categories and $L : \mathscr{A} \to \mathscr{C}$ and $R : \mathscr{B} \to \mathscr{C}$ be functors. The comma category $(L \downarrow R)$ has as objects, triples $(A, f : L(A) \to R(B), B)$ where A is an object of $\mathscr{A}$, and B is an object of $\mathscr{B}$. A morphism from (A, f, B) to (A', f', B') is a pair $(s : A \to A', t : B \to B')$ such that the following diagram commutes in $\mathscr{C}$:

$$\begin{array}{ccc} L(A) & \xrightarrow{\quad f \quad} & R(B) \\ {\scriptstyle L(s)}\downarrow & & \downarrow{\scriptstyle R(t)} \\ L(A') & \xrightarrow[\quad f' \quad]{} & R(B') \end{array}$$

Identities are pairs of identities and composition is defined component-wise, i.e., for $(L \downarrow R)$-morphisms (s, t) and (s', t'), we have: $(s, t) \circ (s', t') = (s \circ s', t \circ t')$.

It is easy to verify that the above definition indeed gives rise to a category.

Remark 3.62 (projection functors) Every comma category $(L \downarrow R)$ is equipped with a pair of projection functors $\pi_1 : (L \downarrow R) \to \mathscr{A}$ and $\pi_2 : (L \downarrow R) \to \mathscr{B}$. The former projects objects and morphisms onto their first coordinates; and the latter projects objects onto their third coordinates and morphisms onto their second.

NOTATION For an arbitrary category $\mathscr{C}$, any $\mathscr{C}$-object C can be considered a functor $1_C : \mathbf{1} \to \mathscr{C}$. Assuming $F : \mathscr{A} \to \mathscr{C}$ is an arbitrary functor, we usually write $(F \downarrow C)$ in place of $(F \downarrow 1_C)$ and $(C \downarrow F)$ in place of $(1_C \downarrow F)$.

[1] Comma categories are not covered in (Barr & Wells, 1999). See (Goguen & Burstall, 1984; Rydeheard & Burstall, 1988) for a detailed discussion of the properties of comma categories.

Example 3.63 (many-sorted sets over an index set) Let S be a fixed set. The category S-**Set** whose objects are S-indexed families of disjoint sets and whose morphisms are S-indexed families of functions is isomorphic to the comma category $(I_{\mathbf{Set}} \downarrow S)$, where $I_{\mathbf{Set}} : \mathbf{Set} \to \mathbf{Set}$ is the identity functor on **Set**.

Example 3.64 (morphism category) For an arbitrary category $\mathscr{C}$, the ***morphism category*** of $\mathscr{C}$, denoted $\mathscr{C}^{\to}$, has the morphisms of $\mathscr{C}$ as objects. A $\mathscr{C}^{\to}$-morphism from $f : A \to B$ to $f' : A' \to B'$ is a pair of $\mathscr{C}$-morphisms $(h : A \to A', k : B \to B')$ making the following diagram commute in $\mathscr{C}$:

$$\begin{array}{ccc} A & \xrightarrow{\quad f \quad} & B \\ {\scriptstyle h}\downarrow & & \downarrow{\scriptstyle k} \\ A' & \xrightarrow[\quad f' \quad]{} & B' \end{array}$$

It is easy to verify that $\mathscr{C}^{\to}$ is isomorphic to $(I_{\mathscr{C}} \downarrow I_{\mathscr{C}})$, where $I_{\mathscr{C}}$ is the identity functor on $\mathscr{C}$.

Example 3.65 (category of graphs, revisited) Denote every graph G as a triple $G = (E, f : E \to N \times N, N)$ where E is a set of edges, N is a set of nodes, and f is a function taking every $e \in E$ to a tuple $\big(\mathsf{source}_G(e), \mathsf{target}_G(e)\big) \in N \times N$. Then, a graph homomorphism from a graph $G = (E, f, N)$ to a graph $G' = (E', f', N')$ consists of a pair of functions $(h_{\text{edge}} : E \to E', h_{\text{node}} : N \to N')$ making the following diagram commute in **Set**:

$$\begin{array}{ccc} E & \xrightarrow{\quad f \quad} & N \times N \\ {\scriptstyle h_{\text{edge}}}\downarrow & & \downarrow{\scriptstyle h_{\text{node}} \times h_{\text{node}}} \\ E' & \xrightarrow[\quad f' \quad]{} & N' \times N' \end{array}$$

It is now easy to verify that **Graph** is isomorphic to the comma category $(I_{\mathbf{Set}} \downarrow T)$ where $I_{\mathbf{Set}} : \mathbf{Set} \to \mathbf{Set}$ is the identity functor on **Set** and $T : \mathbf{Set} \to \mathbf{Set}$ is the Cartesian product functor (see Example 3.29).

Theorem 3.66 *(Tarlecki, 1986; Rydeheard & Burstall, 1988; Borceux, 1994)*[2] *Let $L : \mathscr{A} \to \mathscr{C}$ and $R : \mathscr{B} \to \mathscr{C}$ be functors with L finitely cocontinuous. If $\mathscr{A}$ and $\mathscr{B}$ are finitely cocomplete, so is the comma category $(L \downarrow R)$; moreover, the projections $\pi_1 : (L \downarrow R) \to \mathscr{A}$ and $\pi_2 : (L \downarrow R) \to \mathscr{B}$ preserve finite colimits.*

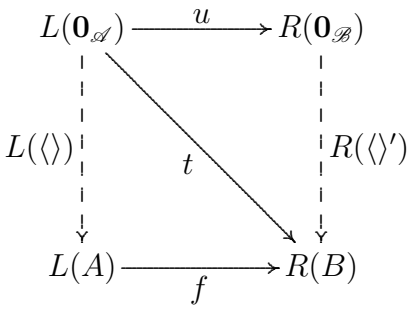
Proof We constructively prove that when the conditions stated in Theorem 3.66 are met, the category $(L \downarrow R)$ has an initial object, binary coproducts of all object pairs, and coequalizers of all parallel morphism pairs. Colimit preservation property of the projection functors follows directly from the constructions.

Initial object: By assumption, $\mathscr{A}$ and $\mathscr{B}$ are finitely cocomplete, so both $\mathscr{A}$ and $\mathscr{B}$ have initial objects. Let $\mathbf{0}_{\mathscr{A}}$ be an initial object in $\mathscr{A}$ and $\mathbf{0}_{\mathscr{B}}$ be an initial object in $\mathscr{B}$. By finite cocontinuity of L, we know that $L(\mathbf{0}_{\mathscr{A}})$ is an initial object in $\mathscr{C}$; therefore, for each $\mathscr{C}$-object C, there is a unique morphism from $L(\mathbf{0}_{\mathscr{A}})$ to C. Particularly, there is a unique morphism $u : L(\mathbf{0}_{\mathscr{A}}) \to R(\mathbf{0}_{\mathscr{B}})$.

We claim that $I = (\mathbf{0}_{\mathscr{A}}, u, \mathbf{0}_{\mathscr{B}})$ is an initial object in $(L \downarrow R)$: suppose $C = (A, f, B)$ is a $(L \downarrow R)$-object. Let $\langle\rangle : \mathbf{0}_{\mathscr{A}} \to A$ be the unique $\mathscr{A}$-morphism from $\mathbf{0}_{\mathscr{A}}$ to A and let $\langle\rangle' : \mathbf{0}_{\mathscr{B}} \to B$ be the unique $\mathscr{B}$-morphism from $\mathbf{0}_{\mathscr{B}}$ to B. Since $L(\mathbf{0}_{\mathscr{A}})$ is an initial object in $\mathscr{C}$, there exists a unique $\mathscr{C}$-morphism $t : L(\mathbf{0}_{\mathscr{A}}) \to R(B)$; hence, $t = f \circ L(\langle\rangle)$; and for the same reason, $t = R(\langle\rangle') \circ u$. Therefore, $f \circ L(\langle\rangle) = R(\langle\rangle') \circ u$, that is, $(\langle\rangle, \langle\rangle') : I \to C$

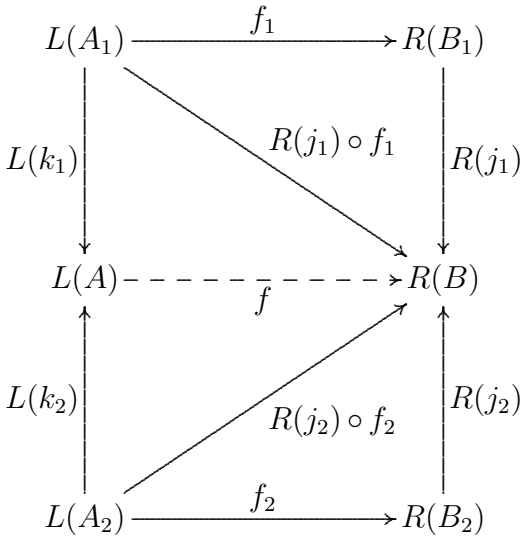
[2]The cited references take the non-finite case into account and prove a stronger result.

is a $(L \downarrow R)$-morphism.



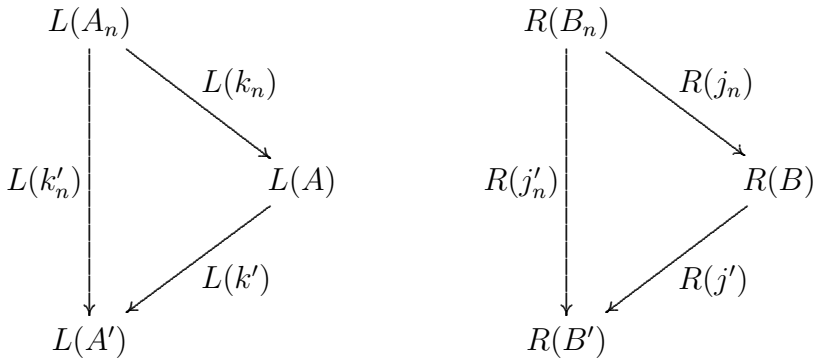
The uniqueness of $(\langle\rangle, \langle\rangle') : I \to C$ follows from the uniqueness of $\langle\rangle$ in $\mathscr{A}$ and the uniqueness of $\langle\rangle'$ in $\mathscr{B}$.

Binary coproduct: Suppose $C_1 = (A_1, f_1, B_1),\ C_2 = (A_2, f_2, B_2)$ are a pair of $(L \downarrow R)$-objects. Let $A = A_1 + A_2$ with injections $k_n : A_n \to A$ and let $B = B_1 + B_2$ with injections $j_n : B_n \to B$ for $n = 1, 2$. By finite cocontinuity of L, we know that $L(A)$ with $\mathscr{C}$-morphisms $L(k_n) : L(A_n) \to L(A)$ for $n = 1, 2$ is a binary coproduct of $L(A_1)$ and $L(A_2)$. Therefore, there exists a unique morphism $f : L(A) \to R(B)$ such that the following diagram commutes:

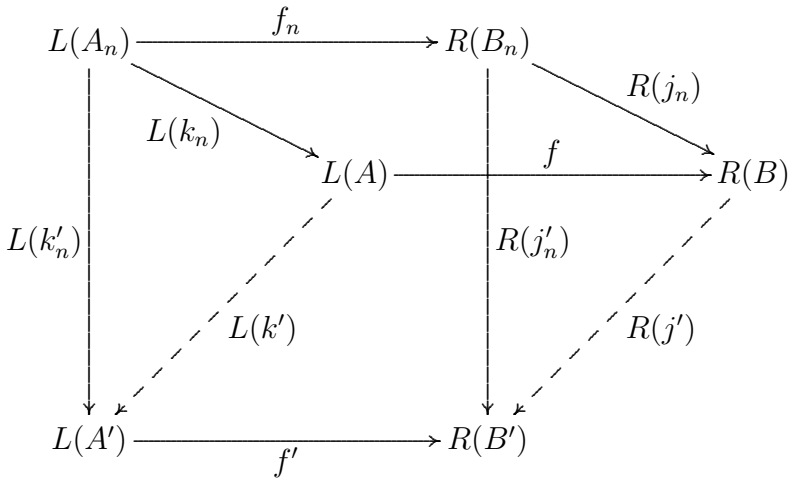


We claim that $C = (A, f, B)$ with $(k_n, j_n) : C_n \to C$ for $n = 1, 2$ is a binary coproduct of C_1 and C_2: suppose $C' = (A', f', B')$ is a $(L \downarrow R)$-object and $(k'_n, j'_n) : C_n \to C'$ for $n = 1, 2$ are $(L \downarrow R)$-morphisms. By the properties of A and B in their respective categories, there are unique morphisms $k' : A \to A'$ and $j' : B \to B'$ such that $k' \circ k_n = k'_n$

and $j' \circ j_n = j'_n$. Therefore, the following two diagrams commute in $\mathscr{C}$ for $n = 1, 2$:



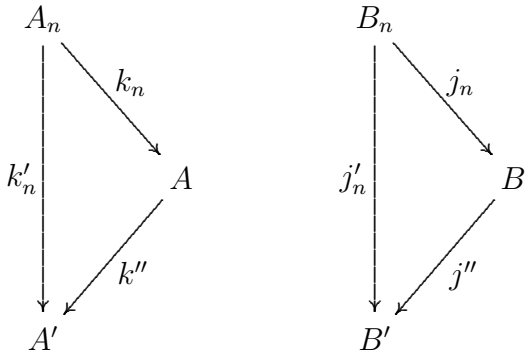
We now show that the following diagram commutes in $\mathscr{C}$ for $n = 1, 2$, as well:



$$f' \circ L(k'_n) = R(j'_n) \circ f_n \implies$$
$$f' \circ L(k') \circ L(k_n) = R(j') \circ R(j_n) \circ f_n \implies$$
$$f' \circ L(k') \circ L(k_n) = R(j') \circ f \circ L(k_n)$$

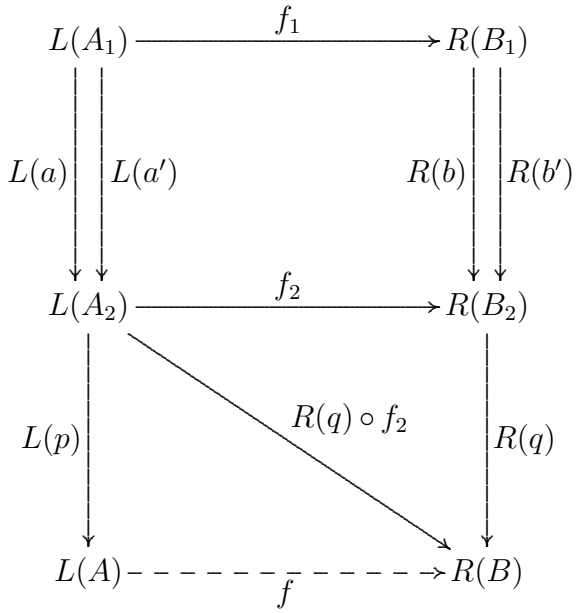
By the properties of $L(A)$ in $(L \downarrow R)$, there is a unique morphism $h : L(A) \to R(B')$ such that $f' \circ L(k'_n) = h \circ L(k_n)$. Therefore, $f' \circ L(k') = h = R(j') \circ f$. Thus, (k', j') is a morphism from C to C' in $(L \downarrow R)$. Uniqueness of (k', j') follows from the fact that any morphism (k'', j'') such that $(k'', j'') \circ (k_n, j_n) = (k'_n, j'_n)$ will make the following two

diagrams commute for $n = 1, 2$:



Therefore, by the properties of A and B in their respective categories, k'' has to be the same as k' and j'' has to be the same as j'.

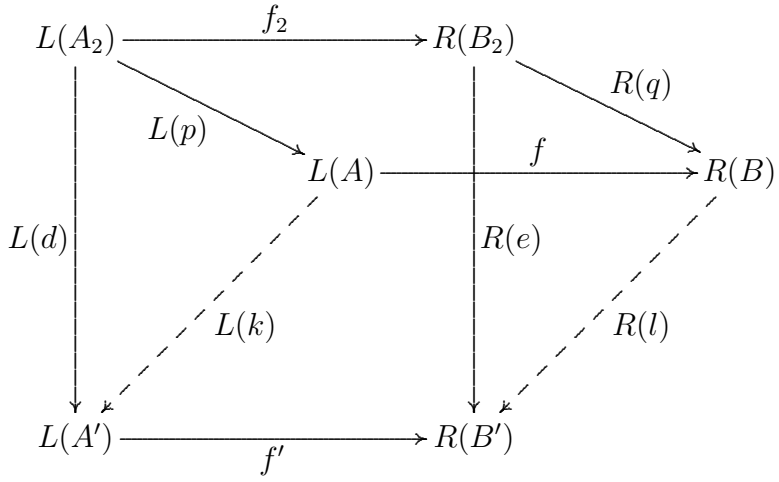
Coequalizer: Suppose $C_1 = (A_1, f_1, B_1)$, $C_2 = (A_2, f_2, B_2)$ are a pair of $(L \downarrow R)$-objects; and $(a, b), (a', b') : C_1 \to C_2$ are a pair of parallel $(L \downarrow R)$-morphisms. Let A with $p : A_2 \to A$ be a coequalizer of $a : A_1 \to A_2$ and $a' : A_1 \to A_2$ (in $\mathscr{A}$) and let B with $q : B_2 \to B$ be a coequalizer of $b : B_1 \to B_2$ and $b' : B_1 \to B_2$ (in $\mathscr{B}$). By finite cocontinuity of L, coequalizers in $\mathscr{A}$ are mapped to coequalizers in $\mathscr{C}$; therefore, there exists a unique morphism $f : L(A) \to R(B)$ such that $f \circ L(p) = R(q) \circ f_2$.



We claim that $C = (A, f, B)$ together with (p, q) is a coequalizer of (a, b) and (a', b'). It is clear from the above diagram that (p, q) is indeed a morphism in $(L \downarrow R)$;

moreover, by the properties of A and B in their respective categories, we have: $(p, q) \circ (a, b) = (p, q) \circ (a', b')$.

Assuming $C' = (A', f', B')$ is a $(L \downarrow R)$-object and $(d, e) : C_2 \to C'$ is a $(L \downarrow R)$-morphism, there exists a unique morphism $k : A \to A'$ (in $\mathscr{A}$) and a unique morphism $l : B \to B'$ (in $\mathscr{B}$) such that: $k \circ p = d$ and $l \circ q = e$. We now show that the following diagram commutes in $\mathscr{C}$:



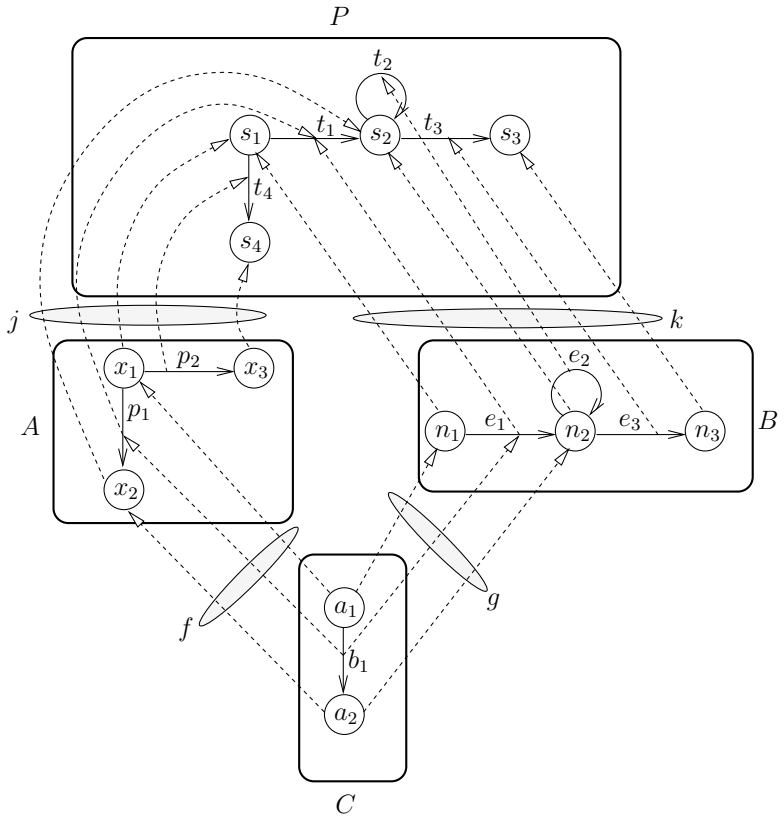
$$f' \circ L(d) = R(e) \circ f_2 \implies$$

$$f' \circ L(k) \circ L(p) = R(l) \circ R(q) \circ f_2 \implies$$

$$f' \circ L(k) \circ L(p) = R(l) \circ f \circ L(p)$$

By the properties of $L(A)$ in $\mathscr{C}$, there exists a unique morphism $h : L(A) \to R(B')$ such that $f' \circ L(d) = h \circ L(p)$. Therefore, $f' \circ L(k) = h = R(l) \circ f$. Hence, (k, l) is a $(L \downarrow R)$-morphism from C to C'. The uniqueness of (k, l) can be proved in exactly the same way as how the uniqueness of (k', j') was proved in the construction of binary coproducts. ■

Remark 3.67 Notice that the colimit preservation property of the projection functors (established in Theorem 3.66) implies that finite colimits in a comma category $(L \downarrow R)$ are inherited from those in the constituent categories (i.e., $\mathscr{A}$ and $\mathscr{B}$) when L is finitely cocontinuous.



$f_{\text{node}} =$	$\{a_1 \mapsto x_1, a_2 \mapsto x_2\}$	$f_{\text{edge}} =$	$\{b_1 \mapsto p_1\}$
$g_{\text{node}} =$	$\{a_1 \mapsto n_1, a_2 \mapsto n_2\}$	$g_{\text{edge}} =$	$\{b_1 \mapsto e_1\}$
$j_{\text{node}} =$	$\{x_1 \mapsto s_1, x_2 \mapsto s_2, x_3 \mapsto s_4\}$	$j_{\text{edge}} =$	$\{p_1 \mapsto t_1, p_2 \mapsto t_4\}$
$k_{\text{node}} =$	$\{n_1 \mapsto s_1, n_2 \mapsto s_2, n_3 \mapsto s_3\}$	$k_{\text{edge}} =$	$\{e_1 \mapsto t_1, e_2 \mapsto t_2, e_3 \mapsto t_3\}$

Figure 3.5: Pushout example in **Graph**

Example 3.68 Since **Set** is finitely cocomplete and the identity functor is finitely cocontinuous, Theorem 3.66 implies that $(I_{\mathbf{Set}} \downarrow T)$, i.e., the category of graphs, is finitely cocomplete as well. Moreover, *finite colimits in the category of graphs are computed component-wise for nodes and edges.*

Example 3.69 (pushouts in Graph) Figure 3.5 shows an example pushout computation in **Graph**. The naming of the graphs and homomorphisms in this figure is compatible with the naming of the objects and morphisms in the pushout square of Definition 3.46.

3.5 Categories of Fuzzy Sets

In this section, we introduce fuzzy sets in a category-theoretic setting. Fuzzy sets are the building blocks for fuzzy graphs introduced in Section 3.6. Our presentation in this section follows (Goguen, 1968; Goguen, 1974).

3.5.1 Fuzzy Sets and Fuzzy Set Morphisms

Definition 3.70 (fuzzy set) Let Q be a poset. A ***Q-valued set*** is a pair (S, σ) consisting of a set S and a function $\sigma : S \to Q$. We call S the ***carrier set*** of (S, σ) and Q the ***truth set*** of σ. For every $s \in S$, the value $\sigma(s)$ is interpreted as the ***degree of membership*** of s in (S, σ).

Definition 3.71 (fuzzy set morphism) Let Q be a poset and let (S, σ) and (T, τ) be a pair of Q-valued sets. A morphism $\mathbf{f} : (S, \sigma) \to (T, \tau)$ is a function $f : S \to T$ such that $\sigma \leq \tau \circ f$, i.e., the degree of membership of s in (S, σ) does not exceed that of $f(s)$ in (T, τ). The function $f : S \to T$ is called the ***carrier function*** of $\mathbf{f}$.

Note In classical fuzzy set theory, it is implicitly assumed that the poset Q is the closed real interval $[0, 1]$ with the obvious linear ordering. We emphasize that we *do not* make such an assumption in this thesis.

Example 3.72 Figure 3.6 (informally) shows two $\mathbf{Fuzz}(\mathbf{A_4})$ objects (S, σ) and (T, τ) along with the carrier function $f : S \to T$ of a $\mathbf{Fuzz}(\mathbf{A_4})$-morphism $\mathbf{f} : (S, \sigma) \to (T, \tau)$ where $\mathbf{A_4}$ is the lattice shown in the same figure.

Definition/Proposition 3.73 (fuzzy set category) For a fixed poset Q, the objects and morphisms defined above together with the obvious identities give rise to a category, denoted $\mathbf{Fuzz}(Q)$.

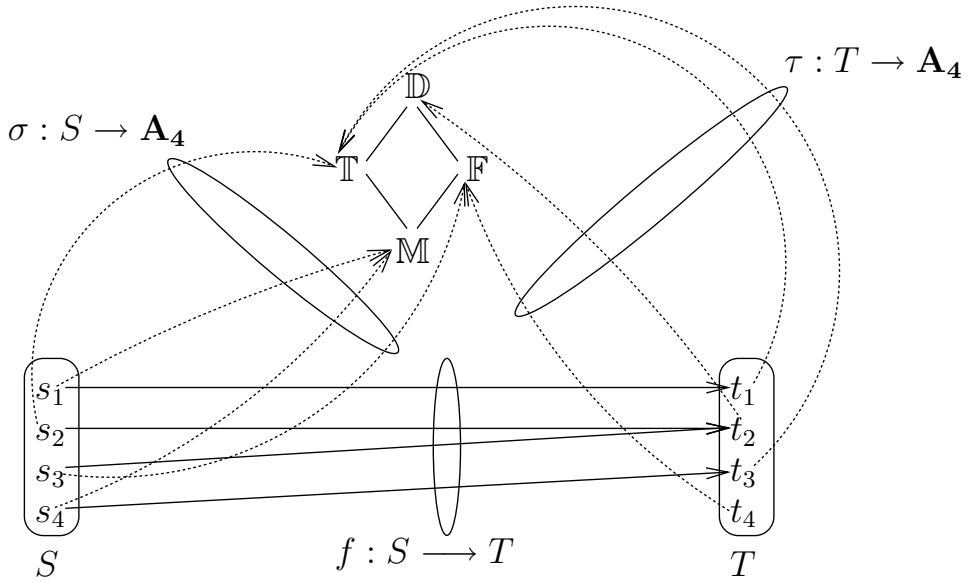


Figure 3.6: Example of fuzzy sets

3.5.2 Cocompleteness Results for Fuzzy Set Categories

Theorem 3.74 *(Goguen, 1968; Goguen, 1974)* **Fuzz**(Q) *is finitely cocomplete when* Q *is a complete lattice.*

Proof (*sketch*) [3] We show how to construct the initial object, binary coproducts, and coequalizers. Finite cocompleteness of **Fuzz**(Q) then follows from Theorem 3.55.

Initial object: $\mathbf{0} = (\emptyset, \lambda)$ where $\lambda : \emptyset \to Q$ is the empty function.

Binary coproduct: given objects $X_1 = (S_1, \sigma_1)$ and $X_2 = (S_2, \sigma_2)$, a coproduct is $X_1 + X_2 = (S_1 + S_2, \kappa)$ where $S_1 + S_2$ is a **Set**-coproduct (disjoint union) of S_1 and S_2 with injections $\imath_n : S_n \to S_1 + S_2$ for $n = 1, 2$; and $\kappa\big(\imath_n(s)\big) = \sigma_n(s)$ for $s \in S_n$ and $n = 1, 2$.

Coequalizer: given objects $X = (A, \sigma)$ and $Y = (B, \tau)$ with parallel morphisms $\mathbf{h_1} : X \to Y$ and $\mathbf{h_2} : X \to Y$, we first take the **Set**-coequalizer of the carrier functions $h_1 : A \to B$ and $h_2 : A \to B$ to find a set C and a function $q : B \to C$. Thus, C is the quotient of B by the smallest equivalence relation $\equiv$ on B such that $h_1(a) \equiv h_2(a)$

[3] I gratefully acknowledge Andrzej Tarlecki for sketching the proof.

for all $a \in A$; and q is the function such that $q(b) = [b]_{\equiv}$ for all $b \in B$. Then, we put $Z = (C, \mu)$ where $\mu([b]_{\equiv}) = \bigsqcup_Q \{\tau(b') \mid b' \equiv b\}$. This lifts the function $q : B \to C$ to a morphism $\mathbf{q} : Y \to Z$, which is a coequalizer of $\mathbf{h_1}$ and $\mathbf{h_2}$. ∎

Remark 3.75 (pushouts in Fuzz(Q)) Let Q be a complete lattice. For computing the pushout of a pair of $\mathbf{Fuzz}(Q)$-morphisms $\mathbf{f} : (C, \gamma) \to (A, \sigma)$ and $\mathbf{g} : (C, \gamma) \to (B, \tau)$, we first compute the **Set**-pushout of the carrier functions $f : C \to A$ and $g : C \to B$ (as discussed in Example 3.48) to find a set P along with functions $j : A \to P$ and $k : B \to P$. Then, we compute a membership degree for every $p \in P$ by taking the supremum of the membership degrees of all those elements in (A, σ) and (B, τ) that are mapped to p. This yields an object (P, ρ) and lifts j and k to $\mathbf{Fuzz}(Q)$-morphisms which together with (P, ρ), constitute the pushout of $\mathbf{f}$ and $\mathbf{g}$ in $\mathbf{Fuzz}(Q)$.

Note In all the figures in the remainder of this section, a fuzzy set (S, σ) is depicted as a set $\big\{\big(s, \sigma(s)\big) \mid s \in S\big\}$ of tuples.

Example 3.76 Figure 3.7 shows two example pushouts in $\mathbf{Fuzz}(\mathbf{A_4})$. The carrier function for each $\mathbf{Fuzz}(\mathbf{A_4})$-morphism $\mathbf{f}$, $\mathbf{g}$, $\mathbf{j}$, $\mathbf{k}$ in Figure 3.7(a) (resp. Figure 3.7(b)) is the same as the corresponding function in Figure 3.3(a) (resp. Figure 3.3(b)).

Definition 3.77 (carrier functor) The map that takes every $\mathbf{Fuzz}(Q)$-object (S, σ) to its carrier set S and every $\mathbf{Fuzz}(Q)$-morphism $\mathbf{f} : (S, \sigma) \to (T, \tau)$ to its carrier function $f : S \to T$ yields a functor $K_Q : \mathbf{Fuzz}(Q) \to \mathbf{Set}$, known as the ***carrier functor***.

Proposition 3.78 *The carrier functor $K_Q : \mathbf{Fuzz}(Q) \to \mathbf{Set}$ is finitely cocontinuous when Q is a complete lattice*[4].

[4]The carrier functor is finitely cocontinuous even when Q is only a poset; however, a separate proof is required.

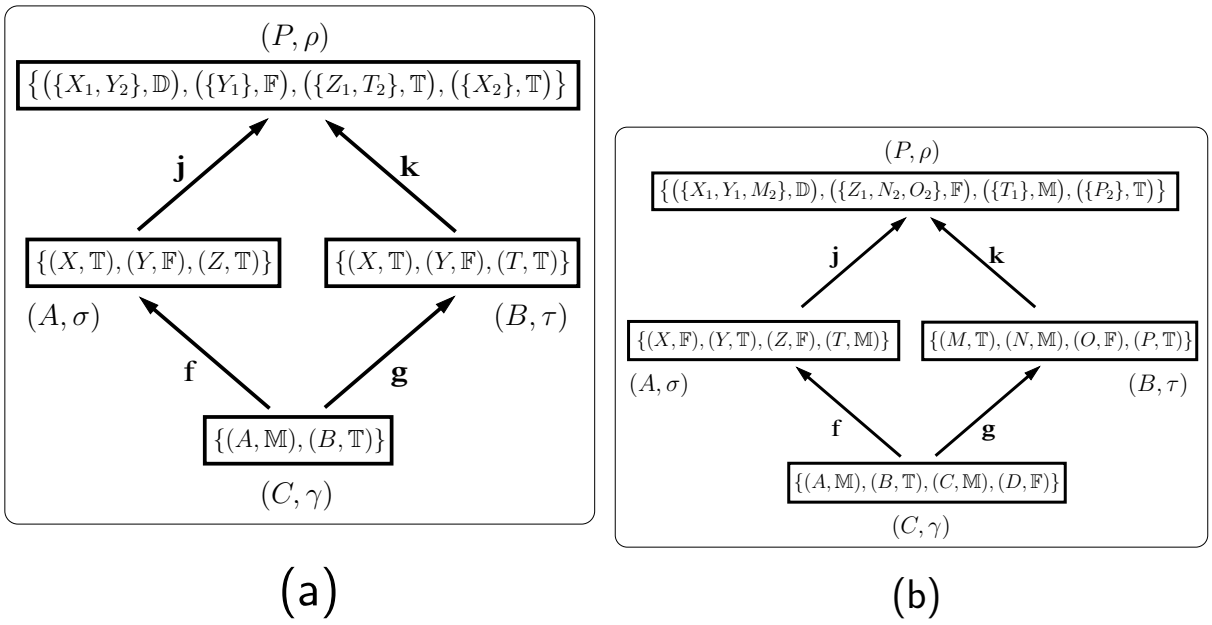


Figure 3.7: Pushout examples in $\mathbf{Fuzz}(\mathbf{A_4})$

Proof Based on the proof of Theorem 3.74, it is obvious that $K_Q : \mathbf{Fuzz}(Q) \to \mathbf{Set}$ preserves the initial object, binary coproducts, and coequalizers. Finite cocontinuity of K_Q then follows from Theorem 3.60. ∎

3.5.3 Fuzzy Powersets

Definition 3.79 (fuzzy powerset) Let Q be a poset and let $Z = (S, \sigma)$ be a $\mathbf{Fuzz}(Q)$-object. The powerset of Z, denoted $\mathcal{P}(Z)$, is the set of all $(C, \xi) \in |\mathbf{Fuzz}(Q)|$ such that $C \subseteq S$ and for every $c \in C$: $\xi(c) \leq \sigma(c)$.

Theorem 3.80 *(Goguen, 1968; Goguen, 1974) The powerset of any* $\mathbf{Fuzz}(Q)$*-object is a complete lattice when* Q *is.*

Proof (*sketch*) (Goguen, 1968; Goguen, 1974) For an index set I, the supremum of an I-indexed family of $\mathcal{P}(Z)$ elements $\langle (S_i, \sigma_i) \rangle_{i \in I}$ is a fuzzy set (X, θ) where $X = \bigcup_{i \in I} S_i$ and $\theta : X \to Q$ is a function such that for every $x \in X$: $\theta(x) = \bigsqcup_Q \{\sigma_i(x) \mid i \in I; x \in S_i\}$. The infimum is computed dually. ∎

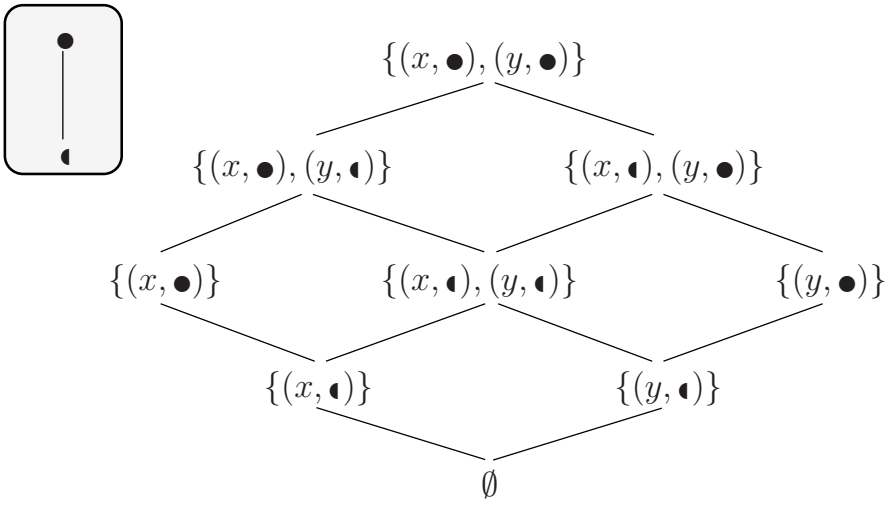


Figure 3.8: Example of a powerset lattice

Example 3.81 Suppose the truth-set is the lattice $\mathbf{L_2} = \{◖, ●\}$ with $◖ < ●$. Then, the powerset of the $\mathbf{Fuzz}(\mathbf{L_2})$-object $Z = \big(\{a, b\}, \{a \mapsto ●, b \mapsto ●\}\big)$ is the lattice shown in Figure 3.8.

We use powerset lattices in Chapter 4 for keeping track of the decisions made by different modellers about the elements of an individual model.

3.6 Categories of Fuzzy Graphs

In this section, we introduce a general notion of fuzzy graphs. We use this notion in Chapter 4 for expressing graph-based models and merging them. The results we present here may also be used to develop graph transformation systems for fuzzy graphs, but we do not explore this application in this thesis.

Definition 3.82 (FGraph category) Let I and J be a pair of posets. The category $\mathbf{FGraph}(I, J)$ is defined as the comma category $(K_J \downarrow T \circ K_I)$ where $K_I : \mathbf{Fuzz}(I) \to \mathbf{Set}$ and $K_J : \mathbf{Fuzz}(J) \to \mathbf{Set}$ are the appropriate carrier functors and $T : \mathbf{Set} \to \mathbf{Set}$ is the Cartesian product functor as defined in Example 3.29.

Theorem 3.83 $\mathbf{FGraph}(I, J)$ *is finitely cocomplete when* I *and* J *are complete lattices.*

Proof By Theorem 3.74, both $\mathbf{Fuzz}(I)$ and $\mathbf{Fuzz}(J)$ are finitely cocomplete, and by Theorem 3.78, K_J is finitely cocontinuous. Finite cocompleteness of $\mathbf{FGraph}(I, J)$ then follows from Theorem 3.66. ∎

Definition 3.84 There is a functor $W : \mathbf{FGraph}(I, J) \to \mathbf{Graph}$, called the ***carrier graph functor***, that maps every object $\big((E, E \to J), f : E \to N \times N, (N, N \to I)\big)$ to $(E, f : E \to N \times N, N)$ and every morphism $(s_{\text{edge}}, t_{\text{node}})$ to $\big(K_J(s_{\text{edge}}), K_I(t_{\text{node}})\big)$.

Example 3.85 Let $\mathcal{L}$ be a linear four-point lattice: {**White**, **Light Grey**, **Dark Grey**, **Black**} ordered by increasing shade intensity, and let $\mathbf{1}$ be the one-point lattice. Figure 3.9 illustrates two $\mathbf{FGraph}(\mathcal{L}, \mathbf{1})$-objects along with a $\mathbf{FGraph}(\mathcal{L}, \mathbf{1})$-morphism.

Example 3.86 Figure 3.10 illustrates an example pushout in $\mathbf{FGraph}(\mathcal{L}, \mathbf{1})$. The morphisms corresponding to f, g, j, and k in the pushout square (Definition 3.46) have been marked in the example.

Example 3.87 Figure 3.11 illustrates an example pushout in $\mathbf{FGraph}(2^S, \mathbf{A_4})$, where S is the set $\{p, q, r\}$ and $\mathbf{A_4}$ is Belnap's four-valued lattice (Belnap, 1977). Here, the nodes and edges of the carrier graphs are without name labels, and only have annotations. Notice that we can replace S with any finite or infinite S' such that $\{p, q, r\} \subseteq S'$ and yet characterize the objects and morphisms in Figure 3.11 as $\mathbf{FGraph}(2^{S'}, \mathbf{A_4})$ objects and morphisms.

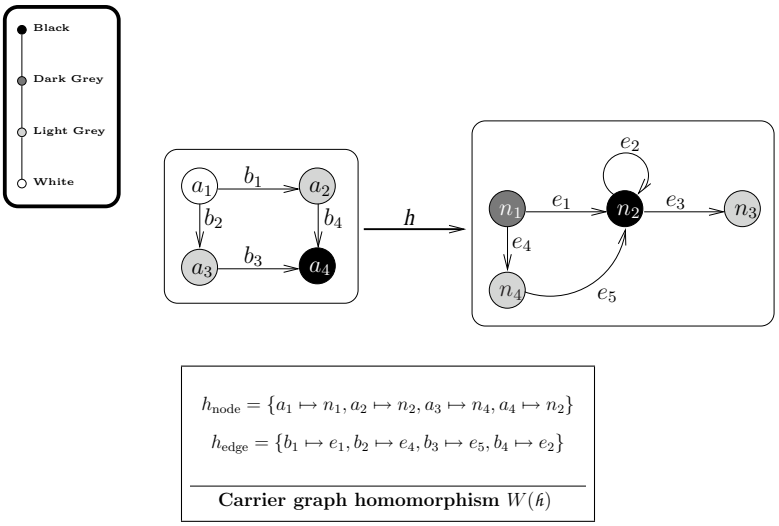


Figure 3.9: Example of $\mathbf{FGraph}(\mathcal{L}, \mathbf{1})$ objects and morphisms

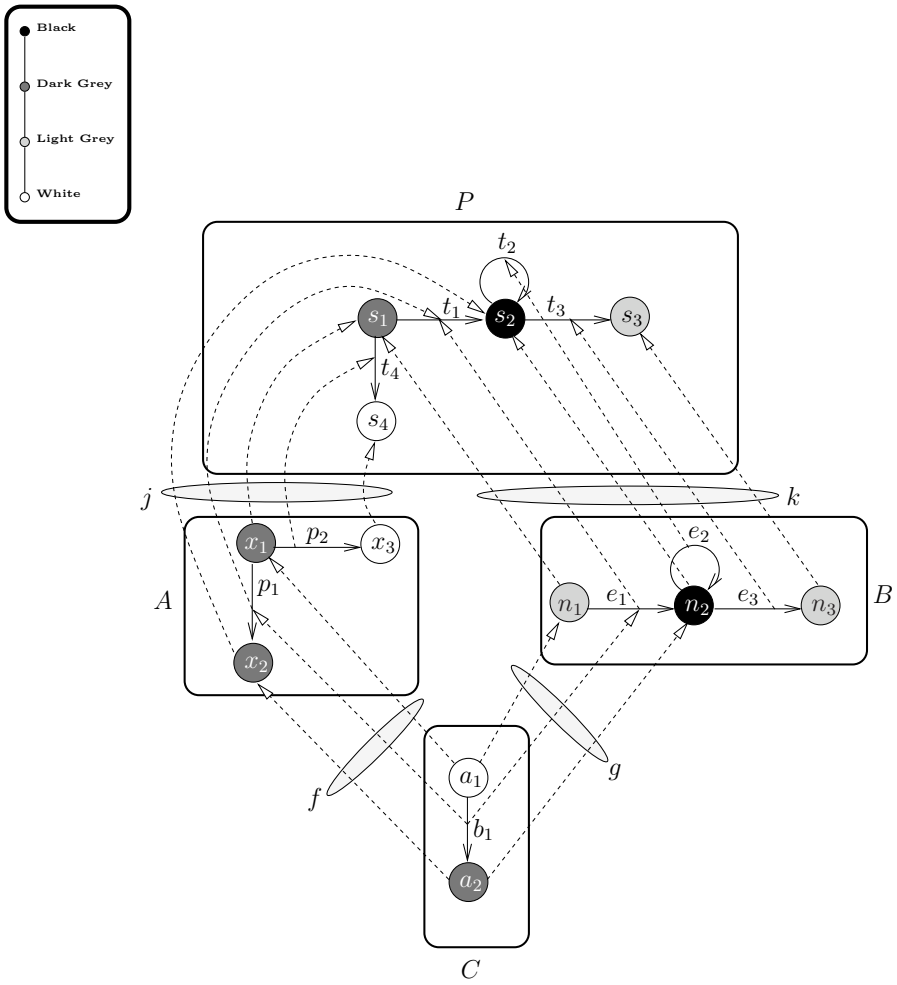


Figure 3.10: Pushout computation in $\mathbf{FGraph}(\mathcal{L}, \mathbf{1})$

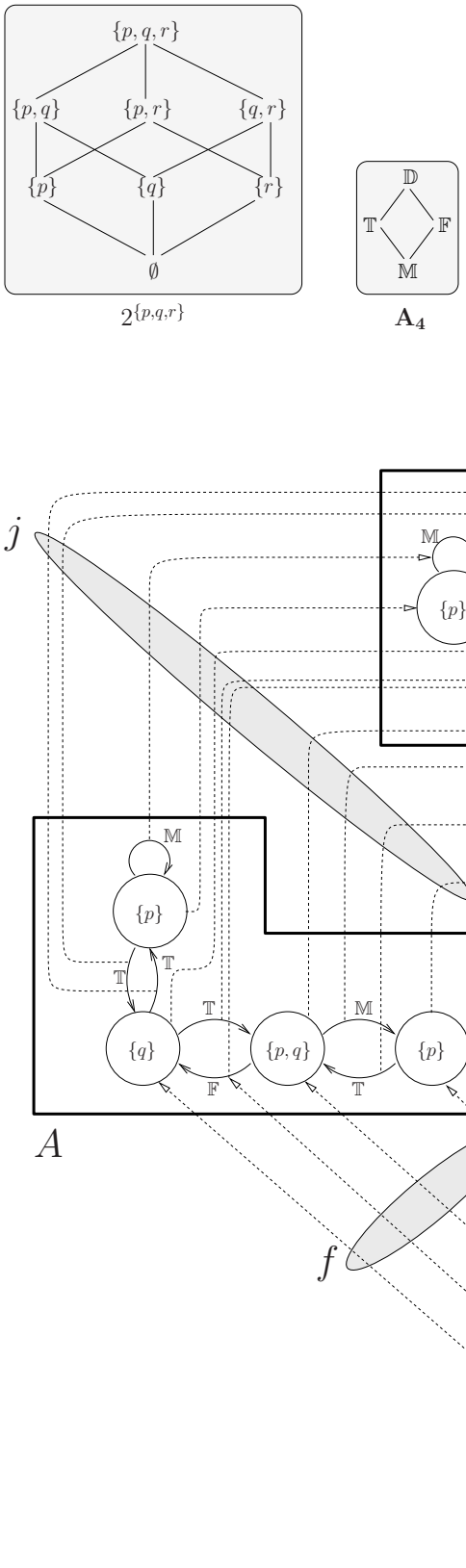


Figure 3.11: Pushout computation in $\mathbf{FGraph}(2^{\{p,q,r\}}, \mathbf{A_4})$

3.7 Background on Formal Logic

In this section, we review the logic background for our consistency checking approach in Chapter 5. Our presentation follows (Libkin, 2004; Rossman, 2008).

3.7.1 The Basics

Definition 3.88 (relational structure) A ***(relational) structure*** is an object $\mathfrak{A} = (A, R_1, \ldots, R_m)$ where A is a nonempty set, m is a natural number, $R_1, \ldots, R_m$ are abstract relation symbols with associated arities $k_1, \ldots, k_m$ (nonnegative integers), and each R_i is a k_i-ary relation on A.

The set A is called the ***universe*** of $\mathfrak{A}$ and may in general be infinite. We consider only *finite* structures, i.e., we assume that A is finite. The sequence of relation symbols $R_1, \ldots, R_m$ together with corresponding arities $k_1, \ldots, k_m$ comprise the ***vocabulary*** of $\mathfrak{A}$. Relation $R_i^{\mathfrak{A}}$ is called the ***interpretation*** of relation symbol R_i in $\mathfrak{A}$. We usually consider structures with a common vocabulary, denoted σ.

In general, relational structures may contain functions and constants as well as relations. We consider only purely relational structures, i.e., structures whose vocabulary consists of only relation symbols. This is not a major restriction, as functions and constants can always be turned into relations. Specifically, a function f with arity k can be written as a relation with tuples $(x_1, \ldots, x_k, f(x_1, \ldots, x_k))$. Constants can be treated as functions with arity zero, and hence, can be written in relational form.

Definition 3.89 (relational structure homomorphism) Let $\mathfrak{A} = (A, R_1^{\mathfrak{A}}, \ldots, R_m^{\mathfrak{A}})$ and $\mathfrak{B} = (B, R_1^{\mathfrak{B}}, \ldots, R_m^{\mathfrak{B}})$ be structures in the same vocabulary. A ***(relational structure) homomorphism*** from $\mathfrak{A}$ to $\mathfrak{B}$ is a function $h : A \to B$ such that $h(R_i^{\mathfrak{A}}) \subseteq R_i^{\mathfrak{B}}$, i.e., if $(a_1, \ldots, a_{k_i}) \in R_i^{\mathfrak{A}}$ then $(h(a_1), \ldots, h(a_{k_i})) \in R_i^{\mathfrak{B}}$ for every $1 \leq i \leq m$.

Remark 3.90 (algebraic graphs as logical relational structures) A graph $G = (N, E, \mathsf{source}_G, \mathsf{target}_G)$, as given by Definition 3.10, can be seen as a relational

structure $\mathfrak{A}$ with universe $A = N \cup E$, and vocabulary $\{\mathsf{Node}, \mathsf{Edge}, \mathsf{Source}, \mathsf{Target}\}$, where Node and Edge are unary relation symbols, and Source and Target are binary relation symbols. The interpretations of these symbols are given as follows:

$$n \in \mathsf{Node}^{\mathfrak{A}} \text{ iff } n \in N; \qquad (e, n) \in \mathsf{Source}^{\mathfrak{A}} \text{ iff } \mathsf{source}_G(e) = n$$

$$e \in \mathsf{Edge}^{\mathfrak{A}} \text{ iff } e \in E; \qquad (e, n) \in \mathsf{Target}^{\mathfrak{A}} \text{ iff } \mathsf{target}_G(e) = n$$

It also easily follows that a graph homomorphism (Definition 3.10) induces a relational structure homomorphism.

Remark 3.91 (graphs in logic versus graphs in algebra) The standard definition of graph in logic differs from that in algebra. Specifically, in logic, the universe is usually assumed to be the set of nodes. Edges are captured using a binary relation symbol E. This simplified treatment is inadequate for our work because it does not allow parallel edges between nodes. For example, it cannot capture the conceptual model in Figure 3.12, saying that a Company plays the role of a supplier for some Parts and the role of a consumer for some others.

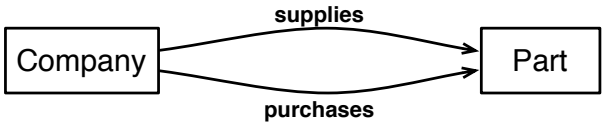


Figure 3.12: Example of parallel edges between nodes

Unless stated otherwise, in this thesis, the term graph refers to the algebraic notion in Definition 3.10. The logical edge relation, $E(x, y)$, can be easily defined as a query in first order logic (see Example 3.92).

3.7.2 First Order Logic

First order logic (FO) formulas in a vocabulary σ are built up from atomic formulas using negation, conjunction, disjunction, and existential and universal quantification:

$$\varphi \quad ::= \quad x = y \mid R(x_1, \ldots, x_n) \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \exists x\varphi(x) \mid \forall x\varphi(x)$$

Here, x, y and $x_1 \ldots, x_n$ are variables, R is a n-ary relation symbol in σ, and φ_1 and φ_2 are formulas. We assume the reader is familiar with the notions of free and bound variables, and the semantics of first order logic, i.e., what it means for a formula to hold over a structure for a given assignment of free variables. Formulas are written out followed by an ordered list of free variables, in the style of $\varphi(x_1, \ldots, x_k)$. For a structure $\mathfrak{A}$ and a tuple $\vec{a} \in A^k$, the notation $\mathfrak{A} \models \varphi(\vec{a})$ asserts that the formula $\varphi(\vec{a})$ holds over $\mathfrak{A}$ with variables $x_1, \ldots, x_k$ taking values $a_1, \ldots, a_k$. Formulas with no free variables are called ***sentences***.

Example 3.92 (edge relation in graphs) To exemplify first order logic, we define a formula $E(x, y)$ capturing the edge relation in graphs (see Remark 3.91).

$$E(x, y) = \exists e \, \mathsf{Source}(e, x) \wedge \mathsf{Target}(e, y).$$

3.7.3 Least Fixpoint Logic

FO does not have sufficient expressive power to describe properties that involve reachability or cycles. To address this limitation, one can add to FO a least fixpoint operator, obtaining the least fixpoint logic (LFP). Below, we first formally define the concept of least fixpoint and then show how FO can be extended with a least fixpoint operator.

Given a set U, let $\mathcal{P}(U)$ denote its powerset. A set $X \subseteq U$ is said to be a ***fixpoint*** of a mapping $F : \mathcal{P}(U) \to \mathcal{P}(U)$ if $F(X) = X$. A set $X \subseteq U$ is a ***least fixpoint*** of F if it is a fixpoint, and for every other fixpoint Y of F, we have $X \subseteq Y$. The least fixpoint of F is denoted by $\mathbf{lfp}(F)$. Least fixpoints are guaranteed to exist only if F is ***monotone***. That is,

$$X \subseteq Y \text{ implies } F(X) \subseteq F(Y).$$

Theorem 3.93 (Knaster-Tarski) *Every monotone mapping $F : \mathcal{P}(U) \to \mathcal{P}(U)$ has a least fixpoint $\mathbf{lfp}(F)$ which can be defined as*

$$\mathbf{lfp}(F) = \bigcap \{Y \mid Y = F(Y)\}$$

Further, $\mathbf{lfp}(F) = \bigcup_{i=0}^{\infty} X^i$ *where* $X^0 = \emptyset$ *and* $X^{i+1} = F(X^i)$.

We now add a least fixpoint operator to FO. Suppose we have a vocabulary σ, and an additional relation symbol $R \notin \sigma$ of arity k. Let $\varphi(R, x_1, \ldots, x_k)$ be a formula of vocabulary $\sigma \cup \{R\}$. For a structure $\mathfrak{A}$ with vocabulary σ, the formula $\varphi(R, \vec{x})$ yields a mapping $F_\varphi : \mathcal{P}(A^k) \to \mathcal{P}(A^k)$ defined as follows:

$$F_\varphi(X) = \{\vec{a} \mid \mathfrak{A} \models \varphi(X/R, \vec{a})\}$$

The notation $\varphi(X/R, \vec{a})$ means that X is substituted for R in φ. More precisely, if $\mathfrak{A}'$ is a $(\sigma \cup \{R\})$-structure expanding $\mathfrak{A}$, in which R is interpreted as X, then $\mathfrak{A}' \models \varphi(\vec{a})$.

To ensure that F_φ is monotone, we impose certain restrictions. Given a formula φ that may contain a relation symbol R, we say that an occurrence of R is ***negative*** if it is under the scope of an odd number of negations, and ***positive***, otherwise. We say that a formula is ***positive in*** R if there are no negative occurrences of R in it, i.e., either all occurrences of R are positive, or there are none at all.

Lemma 3.94 *If* $\varphi(R, \vec{x})$ *is positive in* R*, then* F_φ *is monotone.*

Definition 3.95 (least fixpoint logic) The ***least fixpoint logic (LFP)*** extends FO with the following formula building rule:

- if $\varphi(R, \vec{x})$ is a formula positive in R, where R is k-ary, and $\vec{t}$ is a tuple of terms, where $|\vec{x}| = |\vec{t}| = k$, then

  $$[\mathbf{lfp}_{R,\vec{x}}\varphi(R, \vec{x})](\vec{t})$$

  is a formula, whose free variables are those of $\vec{t}$.

The semantics is defined as follows:

$$\mathfrak{A} \models [\mathbf{lfp}_{R,\vec{x}}\varphi(R, \vec{x})](\vec{a}) \quad \text{iff} \quad \vec{a} \in \mathbf{lfp}(F_\varphi).$$

Example 3.96 (reachability) Consider graphs whose edge relation is E, and let

$$\varphi(R, x, y) = E(x, y) \vee \exists z \left(E(x, z) \wedge R(y, z)\right).$$

Reachability, i.e., the transitive closure of E, is characterized by the formula

$$\psi(x, y) = [\mathbf{lfp}_{R,x,y}\varphi(R, x, y)](x, y).$$

That is, $\psi(a, b)$ holds over a graph G iff there is a path from a to b in G.

3.7.4 Transitive Closure Logic

We saw in Example 3.96 that one of the standard properties expressible in LFP is transitive closure. Below, we introduce an extension of FO, named FO(TC), that is based on a transitive closure operator rather than a general least fixpoint operator. We see in Chapter 5 that FO(TC) provides sufficient expressive power for expressing the consistency constraints we deal with in this thesis.

Definition 3.97 (transitive closure logic) The ***transitive closure logic*** FO(TC) is defined as the extension of FO with the following formula building rule: if $\varphi(\vec{x}, \vec{y}, \vec{z})$ is a formula, where $|\vec{x}| = |\vec{y}| = k$, and $\vec{t_1}, \vec{t_2}$ are tuples of terms of length k, then

$$[\mathbf{trcl}_{\vec{x},\vec{y}}\varphi(\vec{x}, \vec{y}, \vec{z})](\vec{t_1}, \vec{t_2})$$

is a formula whose free variables are $\vec{z}$ plus the free variables of $\vec{t_1}$ and $\vec{t_2}$.

The semantics is defined as follows. Given a structure $\mathfrak{A}$, values $\vec{a}$ for $\vec{z}$ and $\vec{a_i}$ for $\vec{t_i}$, $i = 1, 2$, construct the graph G on A^k with the set of edges

$$\left\{(\vec{b_1}, \vec{b_2}) \mid \mathfrak{A} \models \varphi(\vec{b_1}, \vec{b_2}, \vec{a})\right\}$$

Then

$$\mathfrak{A} \models [\mathbf{trcl}_{\vec{x},\vec{y}}\varphi(\vec{x}, \vec{y}, \vec{a})](\vec{a_1}, \vec{a_2})$$

iff $(\vec{a_1}, \vec{a_2})$ is in the transitive closure of G.

Example 3.98 (graph connectivity) The connectivity of graphs can be expressed by the FO(TC) formula $\forall u \forall v[\mathbf{trcl}_{x,y}\,(E(x,y) \vee E(y,x))](u,v)$.

The following result gives us bounds on the complexity of model checking in FO(TC).

Theorem 3.99 (data complexity of FO(TC)) *(Vardi, 1982) Given a FO(TC) sentence φ, model checking φ is NLOGSPACE in the size of the structure against which φ is evaluated.*

This theorem implies that model checking FO(TC) can be done in manageable space, and that efficient incremental evaluation of formulas is possible (Immerman & Vardi, 1997).

3.7.5 Property Preservation under Homomorphisms

Property preservation is a strong tool for reasoning about correctness of manipulations performed over models. The main question that property preservation tackles is the following: If a property (formula) φ in some logic holds over a structure $\mathfrak{A}$, will φ also hold over a structure $\mathfrak{B}$ obtained from $\mathfrak{A}$ via some manipulation? If the manipulation in question is model merging, then we may want to know whether the consistency properties of the source models are preserved in their merge.

Our merge framework in Chapter 4 uses algebraic colimits (Definition 3.49) for combining models. If the source models are described as graphs, colimits ensure that each source model is embedded into the merge through a homomorphism. As we shall see in Chapter 5, the existence of these homomorphisms leads to preservation of certain consistency properties. Below, we review the theoretical results underlying our discussion of property preservation in Chapter 5. The first result, which dates back to the 1950's, is the Los-Tarski-Lyndon Theorem:

Theorem 3.100 (homomorphism preservation theorem) *(e.g., see (Rosen, 2002; Rossman, 2005)) A first order formula is preserved under homomorphisms on all structures (finite and infinite) if and only if it is equivalent to an existential positive formula, i.e., a formula without negation and universal quantification.*

The existential positive fragment of FO is denoted $\exists\text{FO}^+$. In this thesis, we are interested in finite structures only, and like many classical mathematical logic results that fail in the finite case (e.g., compactness), there is the danger that the above result may fail as well when restricted to finite structures. Fortunately, this is not the case.

Proving sufficiency (i.e., the forward direction of the if-and-only-if) in Theorem 3.100 is trivial for finite structures (and for infinite ones as well). This gives us the following:

Lemma 3.101 *Every $\exists FO^+$ formula is preserved under homomorphisms [on finite structures].*

Proving necessity (i.e., the backward direction) over finite structures has been an open problem for decades and was settled only recently by Rossman (Rossman, 2005).

Theorem 3.102 (homomorphism preservation theorem in the finite case) *(Rossman, 2005) A first order formula is preserved under homomorphisms on finite structures if and only if it is equivalent to an $\exists FO^+$ formula.*

For the finite case, the extension of Lemma 3.101 to existential positive FO(TC) follows trivially from Definition 3.97. More generally, we prove that Lemma 3.101 extends to $\exists\text{LFP}^+$, the existential positive fragment of LFP.

Lemma 3.103 *Every $\exists LFP^+$ formula is preserved under homomorphisms on finite structures.*

Proof Let $\mathfrak{A} = (A, R_1^{\mathfrak{A}}, \ldots, R_m^{\mathfrak{A}})$ and $\mathfrak{B} = (B, R_1^{\mathfrak{B}}, \ldots, R_m^{\mathfrak{B}})$ be a pair of relational structures over vocabulary $\sigma = (R_1, \ldots, R_m)$. Let $h : \mathfrak{A} \to \mathfrak{B}$ be a homomorphism (Defini-

tion 3.89). We show that for every $\varphi \in \exists LFP^+$ and for every $\vec{a} \in A^k$

$$\mathfrak{A} \models \varphi(\vec{a}) \Rightarrow \mathfrak{B} \models \varphi(h(\vec{a}))$$

where $h(\vec{a}) = (h(a_1), \ldots, h(a_k))$.

The proof for $\varphi \in \exists FO^+ \cap \exists LFP^+$ follows from Lemma 3.101. Below, we provide a proof for least fixpoint formulas.

Let $\varphi(\vec{x}) = [\mathbf{lfp}_{R,\vec{y}}\alpha(R,\vec{y})](\vec{x})$. By the definition of **lfp**, for every structure $\mathfrak{A}$, the formula φ yields a mapping $F_{\alpha,\mathfrak{A}} : \mathcal{P}(A^k) \to \mathcal{P}(A^k)$ defined as follows:

$$F_{\alpha,\mathfrak{A}}(X) = \{\vec{a} \mid \mathfrak{A} \models \alpha(X/R, \vec{a})\}$$

By Definition 3.95 and Theorem 3.93, for every $\vec{a} \in A^k$ we have:

$$\vec{a} \in \bigcup_{i=0}^{\infty} F^i_{\alpha,\mathfrak{A}}(\emptyset) \Leftrightarrow \mathfrak{A} \models [\mathbf{lfp}_{R,\vec{y}}\alpha(R,\vec{y})](\vec{a})$$

We first prove by induction that $h(F^i_{\alpha,\mathfrak{A}}(\emptyset)) \subseteq F^i_{\alpha,\mathfrak{B}}(\emptyset)$.

Base case: Let $\vec{a} \in F_{\alpha,\mathfrak{A}}(\emptyset)$. Then, $\mathfrak{A} \models \alpha(\emptyset, \vec{a})$. Since $\mathfrak{A}$ is a substructure of $\mathfrak{B}$ by h and since $h(\emptyset) = \emptyset$, we have $\mathfrak{B} \models \alpha(\emptyset, h(\vec{a}))$. Thus, $h(\vec{a}) \in F_{\alpha,\mathfrak{B}}(\emptyset)$.

Inductive step: Let $\vec{a} \in F^i_{\alpha,\mathfrak{A}}(\emptyset)$. Then, $\mathfrak{A} \models \alpha(F^{i-1}_{\alpha,\mathfrak{A}}(\emptyset), \vec{a})$. Since $\mathfrak{A}$ is a substructure of $\mathfrak{B}$ by h, we have $\mathfrak{B} \models \alpha(h(F^{i-1}_{\alpha,\mathfrak{A}}(\emptyset)), h(\vec{a}))$. Thus, $h(\vec{a}) \in F_{\alpha,\mathfrak{B}}(h(F^{i-1}_{\alpha,\mathfrak{A}}(\emptyset)))$. By the inductive hypothesis and since $F_{\alpha,\mathfrak{B}}$ is monotone, $h(\vec{a}) \in F^i_{\alpha,\mathfrak{B}}(\emptyset)$.

Thus,

$$h(\textstyle\bigcup_{i=0}^{\infty} F^i_{\alpha,\mathfrak{A}}(\emptyset)) \subseteq \bigcup_{i=0}^{\infty} F^i_{\alpha,\mathfrak{B}}(\emptyset) \qquad (\mathbf{1})$$

Therefore,

$\mathfrak{A} \models \varphi(\vec{a}) \quad \Leftrightarrow$

(By assumption $\varphi(\vec{a}) = [\mathbf{lfp}_{R,\vec{y}}\alpha(R,\vec{y})](\vec{a})$)

$\mathfrak{A} \models [\mathbf{lfp}_{R,\vec{y}}\alpha(R,\vec{y})](\vec{a}) \quad \Leftrightarrow$

(By Definition 3.95 and Theorem 3.93)

$\vec{a} \in \bigcup_{i=0}^{\infty} F^i_{\alpha,\mathfrak{A}}(\emptyset) \quad \Leftrightarrow$

(Since h is homomorphism)

$h(\vec{a}) \in h(\bigcup_{i=0}^{\infty} F^i_{\alpha,\mathfrak{A}}(\emptyset)) \quad \Rightarrow$

(By **(1)**)

$h(\vec{a}) \in \bigcup_{i=0}^{\infty} F^i_{\alpha,\mathfrak{B}}(\emptyset) \quad \Leftrightarrow$

(By Definition 3.95 and Theorem 3.93)

$\mathfrak{B} \models [\mathbf{lfp}_{R,\vec{y}}\alpha(R,\vec{y})](h(\vec{a})) \quad \Leftrightarrow$

(By definition of **lfp**)

$\mathfrak{B} \models \varphi(h(\vec{a}))$

■

3.8 Summary

In this chapter, we presented the mathematical background for the thesis. In Chapter 4, we employ colimits (Section 3.4.5) for characterizing the merge operation, and fuzzy graphs (Section 3.6) for formalizing incomplete and inconsistent models. And, in Chapter 5, we use the least fixpoint and transitive closure logics (Sections 3.7.3 and 3.7.4) for consistency checking, and use property preservation (Section 3.7.5) for reasoning about consistency properties of merged models.

Chapter 4

Merging Incomplete and Inconsistent Models

In this chapter, we describe an approach for merging incomplete and inconsistent models, focusing on how model merging can facilitate requirements elicitation from multiple perspectives. Furthermore, the technical results developed in this chapter are a prerequisite for the consistency checking approach in Chapter 5.

4.1 Introduction

Model merging is useful in any conceptual modelling language as a way of consolidating a set of models to gain a unified perspective, to understand interactions among models, or to perform various types of end-to-end analysis.

Numerous approaches to model merging have been proposed, some of the most recent of which were surveyed in Chapter 2. These approaches are limited in two major ways: Firstly, they treat merge as a binary operator, leaving generalization to multiple models to repeated merges. In practice, such a generalization is complicated by the need to construct a new relationship at each step of the merge process. Secondly, the approaches typically assume the set of models are complete and consistent prior to merging. However,

for most interesting applications, the models are likely to be incomplete and inconsistent (Finkelstein *et al.*, 1994). Hence, existing approaches to model merging can be used only if considerable effort is put into detecting and repairing incompleteness and inconsistency.

In this chapter, we present a framework for merging arbitrarily large collections of models that tolerates incompleteness and inconsistency between the models. The framework can be adapted to any graph-based modelling language, as it treats the mappings between models in terms of mappings between nodes and edges in the underlying graphs. We demonstrate the application of the framework to the early requirements modelling language ***i**** (Yu, 1997) and to entity-relationship models.

Our approach to model merging is based on the observation that in exploratory modelling, one can never be entirely sure how concepts expressed in different models should relate to one another. Each attempt to merge a set of models can be seen as a hypothesis about how to put the models together, in which choices have to be made about which concepts overlap, and how the terms used in different models are related. If a particular set of choices yields an unacceptable result, it may be because we misunderstood the nature of the relationships between the models, or because there is a real disagreement between the models over either the concepts being modeled, or how they are best represented. In any of these cases, it is better to perform the merge and analyze the resulting inconsistencies, rather than restrict the available merge choices.

We use category theory (Barr & Wells, 1999) as a theoretical basis for our merge framework. We treat models as structured objects, and the intended relationships between them as structure-preserving mappings. To model incompleteness and inconsistency, we annotate model elements with labels denoting the amount of knowledge available about them. To ensure proper evolution of annotations, we constrain how these labels can be treated in the mappings that interrelate models. We provide a mathematically rigorous merge algorithm based on an algebraic concept called *colimit*. This treatment offers both scalability to arbitrary numbers of models, and adaptability to different conceptual

modelling languages.

After computing a merge, we may need to know how the original models and the defined mappings between them participated in producing the result. Our framework provides the ability to trace the elements of the merged model back to the originating models, to the contributing stakeholders, and to the relationship assumptions relevant to the elements. We discuss how the information required for addressing each of these traceability concerns can be generated and represented in our framework.

4.2 Motivating Examples

We use two working examples throughout this chapter, one involving goal models represented in the ***i**** notation, and another involving database schemata captured by entity-relationship diagrams. Through these applications, we demonstrate how the ideas we present here can be used for managing requirements elicitation artifacts, and to support the exploratory model merging process. This section briefly explains these examples, and uses them to illustrate the main challenges in model merging.

4.2.1 Merging *i** Models

Suppose two stakeholders Mary and Bob want to develop a goal model for a meeting scheduler (van Lamsweerde *et al.*, 1995; Feather *et al.*, 1997), with the help of a requirements analyst, Sam. To ensure that their contributions are adequately captured, each stakeholder first models their perspective separately, using the ***i**** notation. Sam then merges these perspectives to study how well the stakeholders' goals fit together.

Figures 4.1(a) and 4.1(b) show the initial models of Mary and Bob. At first sight, there appears to be no overlap, as Mary and Bob use different terminologies. However, Sam suspects there are some straightforward correspondences: Schedule meeting in Mary's model is probably the same task as Plan meeting in Bob's. Mary's Available dates be obtained

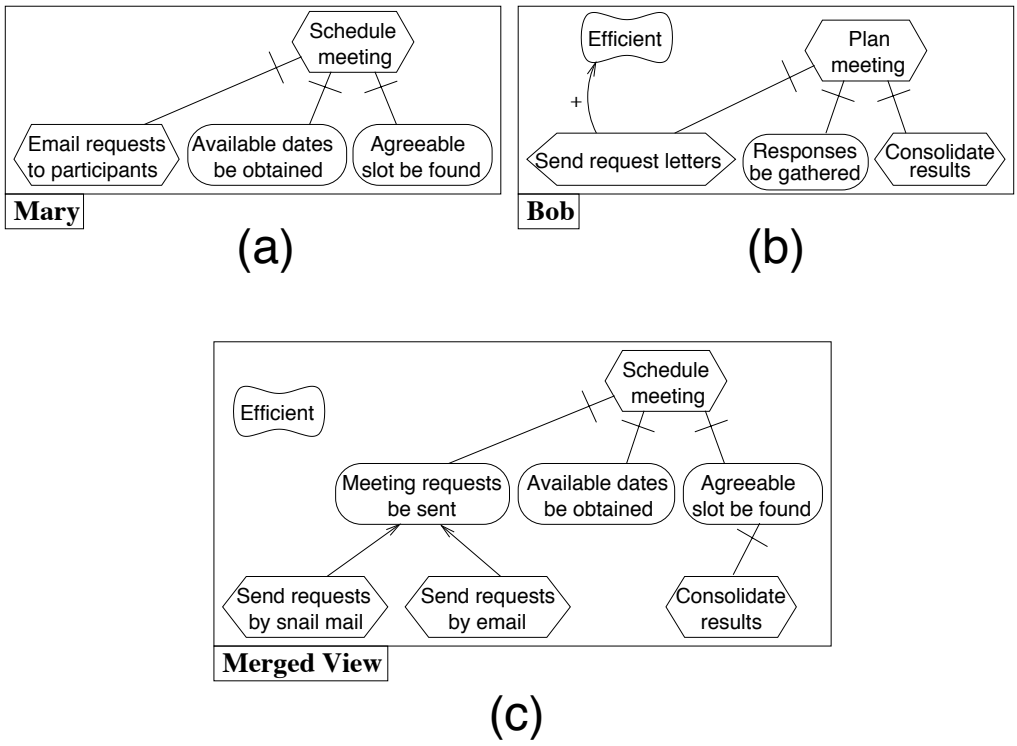


Figure 4.1: Merging ***i**** models

may be the same goal as Bob's Responses be gathered. Sam also thinks it makes sense to treat Mary's Email requests to participants and Bob's Send request letters as alternative ways of satisfying an unstated goal, Meeting requests be sent. Bob's Consolidate results task appears to make sense as a subtask of Mary's Agreeable slot be found goal. Finally, after seeing both models, Mary points out that Bob's positive contribution link from Send request letters to the Efficient soft-goal is inappropriate, although she believes the Efficient soft-goal itself is important.

For a problem of this size, Sam would likely just do an ad-hoc merge with a result such as Figure 4.1(c), and show this to Bob and Mary for validation. This (ad-hoc) merge has a number of drawbacks:

- There is no separation between hypothesizing a relationship between the original models, and generating a merged version based on that relationship. Hence, it is hard for Sam to test out alternative hypotheses, and it will be very hard for Bob

and Mary to check Sam's assumptions individually.

- In an ad-hoc merge, Sam will naturally tend to repair inconsistencies implicitly and align the stakeholders' models with his own vision of the merge. Hence, we lose the opportunities to analyze inconsistencies that arise with a particular choice of merge.

- We have lost the ability to trace conceptual contributions. If it is important to capture stakeholders' contributions in individual models, then it must be equally important to keep track of how these contributions get adapted into the merged model.

4.2.2 Merging Entity-Relationship Models

In the ***i**** model merging example in Section 4.2.1, the merged model (Figure 4.1(c)) would most likely turn out to be agreeable to both Bob and Mary. However, in a more realistic elicitation problem, arriving at a viable consolidation is seldom as easy: Model merging is an iterative and evolutionary process where stakeholders constantly refine their perspectives as a result of gaining more knowledge about the problem, and looking back at previous merges and studying how their models affect and are affected by other parties' intentions. To illustrate this, consider the following example: Suppose Sam, the analyst, now wants to develop a database schema for a payroll system based on Bob's and Mary's perspectives. Models are described using entity-relationship diagrams.

After sketching Mary's and Bob's initial perspectives (Figure 4.2), Sam will merge them to produce a unified schema. He identifies the following correspondences between the two models: Employee in Mary's model is likely to be the same entity as Person in Bob's; and consequently, their name attributes are probably the same. Merging Mary's and Bob's models with respect to these correspondences results in a schema like the one shown Figure 4.3. For naming the elements of the merged schema, Sam favoured Mary's

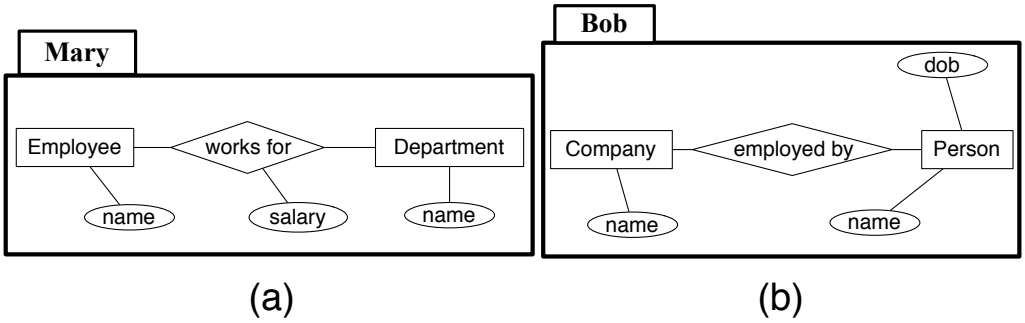


Figure 4.2: Initial perspectives of stakeholders

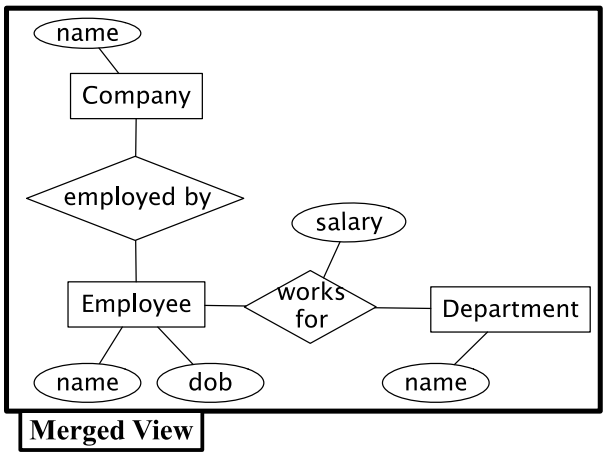


Figure 4.3: First merge attempt

naming choices over Bob's.

When this merge is presented to Mary, she notices Company, an entity she had not included in her original model. She finds the entity to be important; however, she prefers to call it Corporation. She also decides to add an aggregation link from Corporation to Department. Further, she deems Bob's employed by relationship to be redundant in the light of the works for relationship and the aggregation link from Corporation to Department. The new merged schema addressing Mary's concerns is shown in Figure 4.4.

When this new schema is shown to Bob, he finds out that the employed by relationship has been dropped from the merge; however, he argues that there is no redundancy, as it is possible for some employees not to be attached to a particular department. Therefore, he insists that the relationship be added back to the merge!

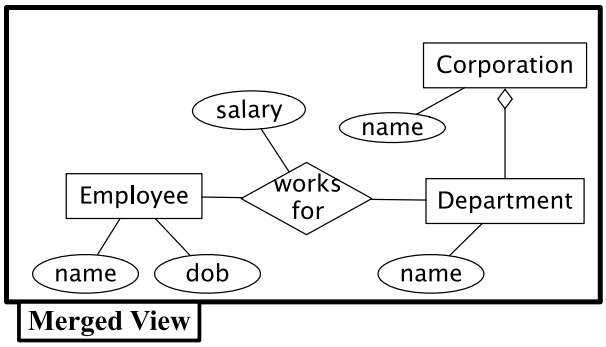


Figure 4.4: Second merge attempt

An ad-hoc merge, or even a structured one computed in a classical framework would fail in at least two respects when faced with a problem such as the one described above:

- It is not possible to describe how sure stakeholders are about elements of their models, and how their beliefs evolve over time. If we later need to know *how flexible* a stakeholder is with respect to a certain decision, we have no way of discovering how strongly the stakeholder argued for (or against) the decision.

- Disagreements between stakeholders would need to be resolved immediately after being identified because we have no means to model such disagreements explicitly. This is unsatisfactory – previous work suggests that toleration of inconsistencies and disagreements, and being able to delay their resolution is basis for flexible development (Easterbrook & Nuseibeh, 1996).

Our model merging framework addresses all the problems motivated by the examples in Sections 4.2.1 and 4.2.2.

4.3 Model Merging as an Abstract Operation

Our model merging framework is based on a category-theoretic concept called *colimit* (Barr & Wells, 1999). We already provided a formal introduction to category theory in Chapter 3. Here, we focus on the intuitions that motivate our use of category theory.

Intuitively, a *category* is an algebraic structure consisting of a collection of *objects* together with a collection of *mappings* (also known as *arrows* or *morphisms*). Each mapping connects a pair of objects, known as its *source* and *destination*. Typically, the objects will have some internal structure, and the mappings express ways in which the structure of one object maps onto that of another. For example, if the objects are geometric shapes, then the mappings could be transformations that preserve shape, such as rotation and scaling. This gives rise to a number of familiar constructs – for example, if a mapping between two objects has an inverse, then we say the two objects are *isomorphic*, i.e., the objects have the same structure.

The appeal of category theory is that it provides a formal foundation for manipulating collections of objects and their mappings. In our case, the objects are models, and the mappings are known or hypothesized relationships between them. We describe a "system" of interrelated objects using an *interconnection diagram* – a directed graph whose nodes and edges are labelled respectively with objects and mappings from a category. The label of each edge in an interconnection diagram has to be consistent with the labels of its endpoints, i.e., if an edge has mapping $m : O_1 \rightarrow O_2$ as its label, then the source and target nodes of the edge should be labelled by objects O_1 and O_2, respectively.

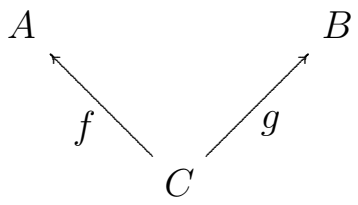
The *colimit* of an interconnection diagram is a new object, called the *colimiting object*, together with a family of mappings, one from each object in the diagram onto the colimiting object[1]. Since each mapping expresses how the internal structure of its source object is mapped onto that of its destination object, the colimit expresses the merge of all the objects in the interconnection diagram. Furthermore, the colimit respects the mappings in the diagram: The intuition here is that the image of each object in the colimit is the same, no matter which path through the mappings in the diagram you follow. By definition, the colimit is also minimal – it merges the objects in the diagram

[1]In the remainder of the chapter, with a slight abuse of terminology, we use the term "colimit" to refer to the colimiting object for a given interconnection diagram.

without adding anything essentially new (Goguen, 1991).

Each of the merge algorithms we discuss in this chapter corresponds to colimit computation in a category: Merging sets (Section 4.4.1) is based on colimit construction in the category of sets[2]; merging graphs (Section 4.4.2) is based on colimit construction in the category of graphs[3]; and merging annotated graphs (Section 4.5) is based on colimit construction in the category of fuzzy graphs[4].

To merge a set of models, we first express how they are related in an interconnection diagram, and then compute the colimit. For example, if we want to merge two models, A and B, that overlap in some way, we can express the overlap as a third model, C, with mappings from C to each of A and B:



In this interconnection diagram, the two mappings f and g specify how the common part, C, is represented in each of A and B. The colimit of this diagram is a new model, P, expressing the union of A and B, such that their overlap, C, is included only once. This simple interconnection pattern is known as a *three-way merge*.

The reason why we hypothesize merges explicitly and define the merge operation in terms of specific interconnection diagrams, rather than in terms of *all* given models and mappings is because, from time to time, we may want to create merges using only a subset of the existing models. We therefore need to be able to specify which models are involved in each merge. Further, we may have several competing versions of mappings between any two participating models making it necessary also to specify which mappings

[2]See Example 3.56 and Remark 3.58.
[3]See Example 3.65 and Theorem 3.66.
[4]See Theorem 3.83 and Examples 3.86, 3.87.

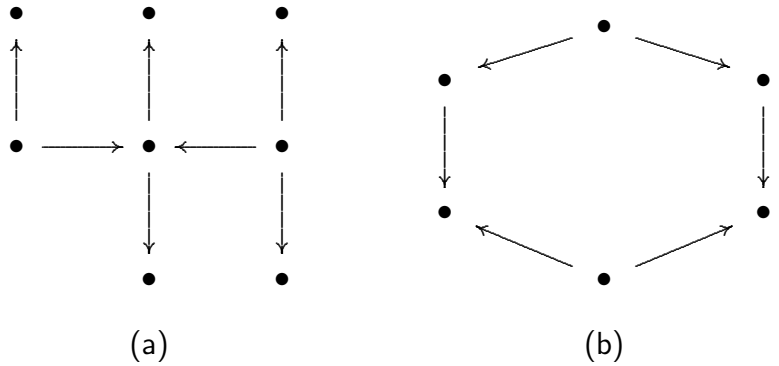


Figure 4.5: Examples of interconnection patterns

are to be used for computing a particular merge.

In practice, interconnection diagrams often have more complex patterns than that of three-way merge. Figure 4.5 shows two examples used later in this chapter: 4.5(a) is used for capturing the relationships between the i^* meta-model fragments in Figure 4.10, and 4.5(b) is used for capturing the relationships between the models in Figure 4.13.

4.4 Interconnecting and Merging Graphs

In our framework, we assume that the underlying structure of each model can be treated as a graph. This section introduces graphs, and describes how they can be interconnected and merged. Further, it explains how graphs can be equipped with a typing mechanism. The merge algorithm for graphs is built upon that for sets; therefore, we begin with a discussion of how sets can be merged.

4.4.1 Merging Sets

A system of interrelated sets is given by an interconnection diagram whose objects are sets and whose mappings are (total) functions. Rather than treating functions as general mapping rules between arbitrary sets, we consider each function to be a map with a unique domain and a unique codomain. Each function can be thought of as an embedding: each element of the domain set is mapped to a corresponding element in the codomain set.

For example, in a three-way merge, the mappings would show how the set C is embedded in each of A and B.

To describe the algorithm for merging sets, we need to introduce the concept of *disjoint union*: The disjoint union of a given family of sets $S_1, S_2, \ldots, S_n$, denoted $S_1 \uplus S_2 \uplus \ldots \uplus S_n$, is (isomorphic to) the following set: $S_1 \times \{1\} \cup S_2 \times \{2\} \cup \ldots \cup S_n \times \{n\}$. For conciseness, we construct the disjoint union by subscripting the elements of each given set with the name of the set and then taking the union. For example, if $S_1 = \{x, y\}$ and $S_2 = \{x, t\}$, we write $S_1 \uplus S_2$ as $\{x_{S_1}, y_{S_1}, x_{S_2}, t_{S_2}\}$ instead of $\{(x, 1), (y, 1), (x, 2), (t, 2)\}$.

To merge a system of interrelated sets, we start with the disjoint union as the largest possible merged set, and refine it by grouping together elements that get unified by the interconnections. To identify which elements should be unified, we construct a *unification graph* U, a graphical representation of the symmetric binary relation induced on the elements of the disjoint union by the interconnections. We then combine the elements that fall in the same connected component of U. Figure 4.6 shows the merge algorithm for an interconnection diagram whose objects are sets $S_1, \ldots, S_n$ and whose mappings are functions $f_1, \ldots, f_k$.

Figure 4.7 shows an example of three-way merge for sets: 4.7(a) shows the interconnection diagram; 4.7(b) shows the induced unification graph and its connected components; and 4.7(c) shows the merged set.

The example shows that simply taking the union of two sets A and B might not be the right way to merge them as this may cause name-clashes (e.g., according to the interconnections, the y elements in A and B are not the same although they share the same name), or duplicates for equivalent but distinctly-named elements (e.g., according to the interconnections, w in A and t in B are the same despite having distinct names).

In the above set-merging example, the elements of each set were uniquely identifiable by their names within the set. This is not necessarily the case in general because we may have unnamed or identically-named, but distinct elements. For example, in Sections 4.2.1

Algorithm. Set-Merge

Input: Sets $S_1, \ldots, S_n$

Functions $f_1, \ldots, f_k$

Output: Merged set P

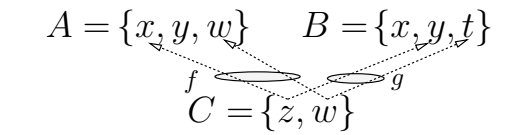
- ▷ Let U be an initially discrete graph with node-set $S_1 \uplus \ldots \uplus S_n$;
- ▷ For every function $f_i \quad (1 \leq i \leq k)$:
  - ▷ For every element a in the domain of f_i:
    - ▷ Add to U an undirected edge between the elements corresponding to a and $f_i(a)$;
- ▷ Let P be the set of the connected components of U;
- ▷ Return P as the result of the merge operation.

Figure 4.6: Algorithm for merging sets

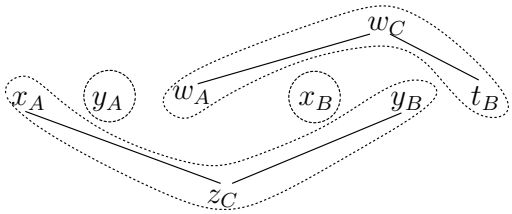
and 4.2.2, most edges in the models were unnamed; and in Section 4.2.2, the name node appeared more than once in Bob's and Mary's models as well as the merges. To avoid ambiguity, our implementation of the merge framework (discussed in Chapter 6) uses *unique identifiers* (uid's) instead of names to distinguish between model elements.

Name Mapping

To assign a name to each element of the merged set in Figure 4.7, we combined the names of all the elements in A, B, and C that are mapped to it. For example, "$\{x_A, y_B, z_C\}$" indicates an element that *represents* x of A, y of B, and z of C. A better way to name the elements of the merged set is assigning *naming priorities* to the input sets. For example, in three-way merge, it makes sense to give priority to the element names in the connector, C, and write the merged set in our example as $P = \{z_C, y_A, w_C, x_B\}$. In this particular example, there are no name-clashes in the merged set, so we could drop the element subscripts and write $P = \{z, y, w, x\}$; however, in general, the subscripts are



(a)



(b)

$$P = \{\underline{\{x_A, y_B, z_C\}}, \underline{\{y_A\}}, \underline{\{w_A, t_B, w_C\}}, \underline{\{x_B\}}\}$$

(c)

Figure 4.7: Three-way merge example for sets

needed to avoid name clashes that arise when models use the same terms to describe different concepts.

This naming convention is of no theoretical significance, but it provides a natural solution to the name mapping problem: in most cases, we would like the choice of names in *connector objects*, i.e., objects solely used to describe the relationships between other objects, to have precedence in determining the element names in the merged object. We will use this convention in the rest of this chapter.

4.4.2 Graphs and Graph Merging

The notion of graph as introduced below is a specific kind of directed graph used in algebraic approaches to graph-based modelling and transformation (Ehrig & Taentzer, 1996), and has been successfully applied to capture various graphical formalisms including UML, entity-relationship diagrams, and Petri nets (Rozenberg, 1997).

Definition 4.1 (graph) A ***(directed) graph*** is a tuple $G = (N, E, \mathsf{source}, \mathsf{target})$ where N is a set of nodes, E is a set of edges, and $\mathsf{source}, \mathsf{target} : E \to N$ are functions respectively giving the source and the target of each edge.

To interconnect graphs, a notion of mapping needs to be defined. A natural choice of mapping between graphs is homomorphism – a structure-preserving map describing how a graph is embedded into another:

Definition 4.2 (graph homomorphism) Let $G = (N, E, \mathsf{source}, \mathsf{target})$ and $G' = (N', E', \mathsf{source}', \mathsf{target}')$ be graphs. A ***(graph) homomorphism*** $h : G \to G'$ is a pair of functions $\langle h_{\mathbf{node}} : N \to N', h_{\mathbf{edge}} : E \to E' \rangle$ such that for all edges $e \in E$, if $h_{\mathbf{edge}}$ maps e to e' then $h_{\mathbf{node}}$ respectively maps the source and the target of e to the source and the target of e'; that is: $\mathsf{source}'(h_{\mathbf{edge}}(e)) = h_{\mathbf{node}}(\mathsf{source}(e))$ and $\mathsf{target}'(h_{\mathbf{edge}}(e)) = h_{\mathbf{node}}(\mathsf{target}(e))$. We call $h_{\mathbf{node}}$ the ***node-map function***, and $h_{\mathbf{edge}}$ the ***edge-map function*** of h.

A system of interrelated graphs is given by an interconnection diagram whose objects are graphs and whose mappings are homomorphisms. Merging is done component-wise for nodes and edges. For a graph interconnection diagram with objects $G_1, \ldots, G_n$ and mappings $h_1, \ldots, h_k$, the merged object P is computed as follows: The node-set (resp. edge-set) of P is the result of merging the node-sets (resp. edge-sets) of $G_1, \ldots, G_n$ with respect to the node-map (resp. edge-map) functions of $h_1, \ldots, h_k$.

To determine the source (resp. target) of each edge e in the edge-set of the merged graph P, we pick, among $G_1, \ldots, G_n$, some graph G_i that has an edge q which is represented by e. Let s (resp. t) denote the source (resp. target) of q in G_i; and let s' (resp. t') denote the node that represents s (resp. t) in the node-set of P. We set the source (resp. target) of e in P to s' (resp. t'). Notice that an edge in the merged graph may represent edges from several input graphs. In a category-theoretic setting, it can be shown that the source and the target of each edge in the merged graph are uniquely

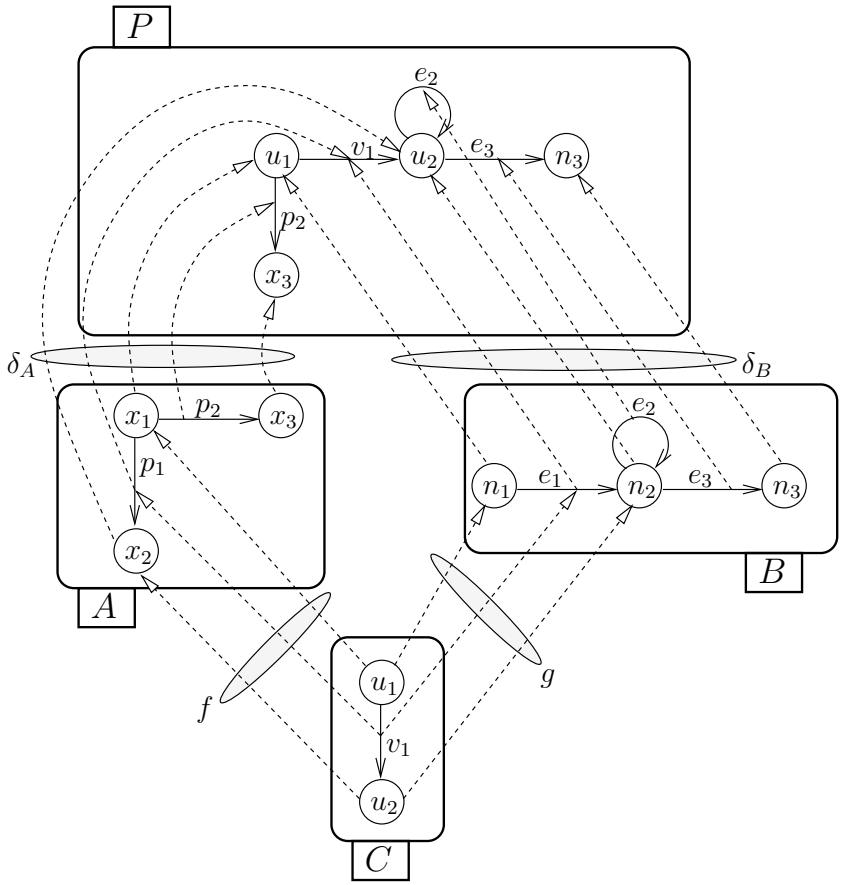


Figure 4.8: Three-way merge example for graphs

determined irrespective of which G_i we pick[5].

Figure 4.8 shows an example of three-way merge for graphs. In the figure, each homomorphism has been visualized by a set of directed dashed lines. In addition to the homomorphisms of the interconnection diagram, i.e., f and g, we have shown the homomorphisms δ_A and δ_B specifying how A and B are represented in P. The homomorphism from C to P is implied and has not been shown.

To compute the graph P in Figure 4.8, we first separately merged the node-sets and the edge-sets of A, B, C. That is, we merged sets $\{x_1, x_2, x_3\}, \{n_1, n_2, n_3\}, \{u_1, u_2\}$ with respect to functions $f_{\mathbf{node}} = \{u_1 \mapsto x_1, u_2 \mapsto x_2\}, g_{\mathbf{node}} = \{u_1 \mapsto n_1, u_2 \mapsto n_2\}$; and merged $\{p_1, p_2\}, \{e_1, e_2, e_3\}, \{v_1\}$ with respect to $f_{\mathbf{edge}} = \{v_1 \mapsto p_1\}, g_{\mathbf{edge}} = \{v_1 \mapsto e_1\}$. This yielded

[5]See Theorem 3.66.

two sets $N = \{u_1, u_2, x_3, n_3\}$, $E = \{v_1, p_2, e_2, e_3\}$ constituting the node-set and the edge-set of P respectively. For naming the elements of N and E, we gave priority to the choice of names used in graph C (name mapping was already discussed in Section 4.4.1). After computing N and E, we assigned to each edge in E a source and a target node from N using the method described earlier. We illustrate this with two examples: 1) To determine the source and target of v_1 in E, we need to pick, among A, B, C, a graph that has an edge represented by v_1. In this case, any of the three graphs will do because v_1 has a pre-image in each of them – the edge represents p_1 of A, e_1 of B, and v_1 of C. Regardless of which graph we pick, the computed source and target will be the same. Suppose we pick A. Edge p_1 has x_1 as source and x_2 as target. The two nodes are represented in N by u_1 and u_2 respectively; therefore, v_1 is assigned u_1 as source and u_2 as target. 2) Now, consider e_3 in E. The edge has a pre-image in graph B only. Thus, we pick B. Edge e_3 in B has n_2 as source and n_3 as target. Nodes n_2 and n_3 are respectively represented by u_2 and n_3 in N. Thus, e_3 in E is assigned u_2 as source and n_3 as target.

4.4.3 Enforcement of Types

Graph-based modelling languages typically have typed nodes and edges. The definitions of graph and homomorphism given earlier do not support types; therefore, we need to extend them for typed graphs. We can then restrict the admissible mappings to those that preserve types.

In (Corradini *et al.*, 1996), a powerful typing mechanism for graphs has been proposed using the relation between the models and the meta-model for the language. Assuming that the meta-model for the language of interest is given by a graph $\mathcal{M}$, every model is described by a pair $\langle G, t : G \to \mathcal{M} \rangle$ where G is a graph and t is a homomorphism, called the *typing map*, assigning a type to every element in G. Notice that a typing map is a homomorphism, offering more structure than an arbitrary pair of functions assigning types to nodes and edges. A typed homomorphism $\underline{h} : \langle G, t \rangle \to \langle G', t' \rangle$ is simply

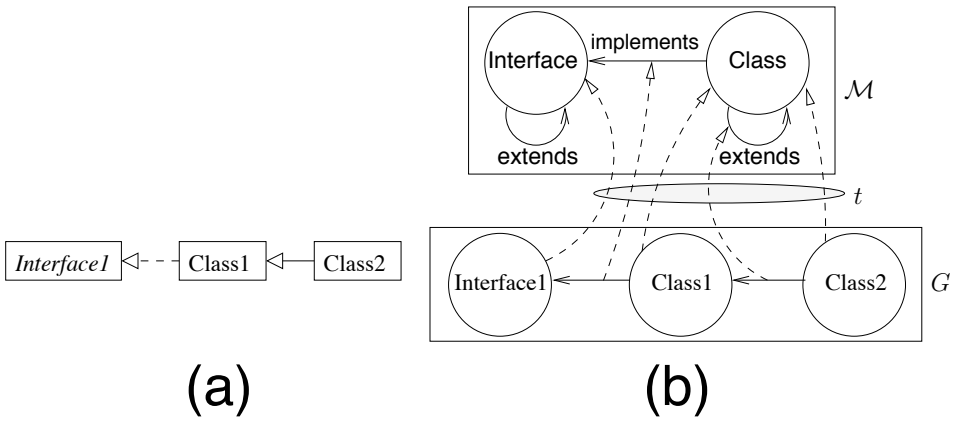


Figure 4.9: Example of typed graphs

a homomorphism $h : G \to G'$ that preserves types, i.e., $t'(h(x)) = t(x)$ for every element x in G. This typing mechanism is illustrated in Figure 4.9: 4.9(a) shows a Java class diagram in UML notation and 4.9(b) shows how it can be represented using a typed graph. The graph $\mathcal{M}$ in 4.9(b) is the extends–implements fragment of the meta-model for Java class diagrams.

The meta-model for a graph-based language can be much more complex than that of Figure 4.9. Figure 4.10 shows some fragments of the ***i**** meta-model extracted from the visual syntax description of ***i**** 's successor GRL (Goal-oriented Requirement Language, 2004). Instead of showing the whole meta-model in one graph, we have broken it into a number of *views*, each of which represents a particular type of relationship (means-ends, decomposition, etc.). Our graph merging framework allows us to describe the meta-model without having to show it monolithically: the ***i**** meta-model, $\mathcal{M}_{\boldsymbol{i}^*}$, is the result of merging the interconnection diagram in Figure 4.10. To describe the relations between the meta-model fragments, a number of connector graphs (shaded gray) have been used. Each mapping (shown by a thick solid line) is a homomorphism giving the obvious mapping. Notice that the connector graphs are *discrete* (i.e., do not have any edges) as no two meta-model fragments share common edges of the same type.

The $\wedge$- and $\vee$-contribution structures in ***i**** convey a relationship between a group of

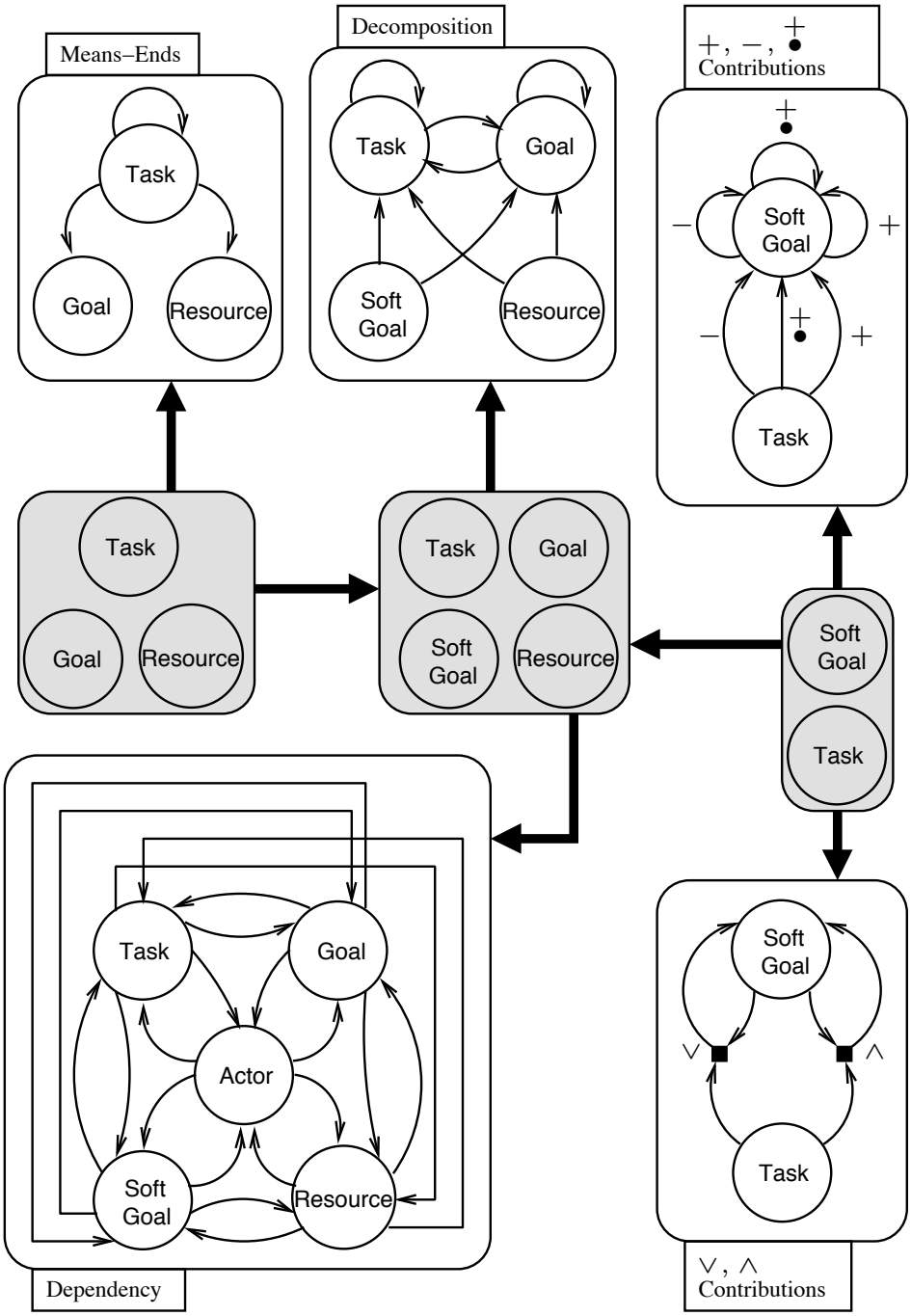


Figure 4.10: Some meta-model fragments of ***i****

edges. To capture this, we introduced helper nodes (shown as small rectangular boxes) in the meta-model to group edges that should be related by $\wedge$ or $\vee$. Figure 4.11(a) shows how we normally draw an $\vee$-contribution structure in an ***i**** model and Figure 4.11(b)

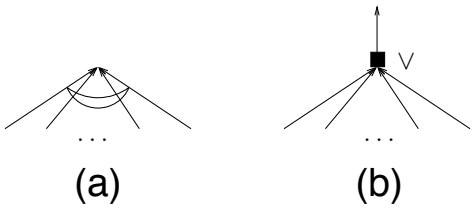


Figure 4.11: Adaptation of ∨-contribution

shows the adaptation of the structure to typed graphs. Structures conveying relationships between a combination of nodes and edges can be modeled similarly.

The merge operation for typed graphs is the same as that for untyped graphs. The only additional step required is assigning types to the elements of the merged graph: each element in the merged graph inherits its type from the elements it represents. In a category-theoretic setting, it can be proven that every element of the merged graph is assigned a unique type in this way and that a typing map can be established from the merged graph to the meta-model[6].

4.5 Merging in the Presence of Incompleteness and Inconsistency

In this section, we show how incompleteness and inconsistency can be modeled by an appropriate choice of *annotation* for model elements. Using the motivating examples in Section 4.2, we demonstrate how incomplete and inconsistent models can be represented, interconnected, and merged.

[6]To see why, notice that going from untyped graphs to graphs typed by a meta-model $\mathcal{M}$ means going from the category **Graph** to the comma category $(L \downarrow R)$ where L : **Graph** $\rightarrow$ **Graph** is the identity functor, and R : **1** $\rightarrow$ **Graph** is the functor from the singleton category **1** to **Graph**, mapping the single object in **1** to the type graph $\mathcal{M}$ in **Graph**.

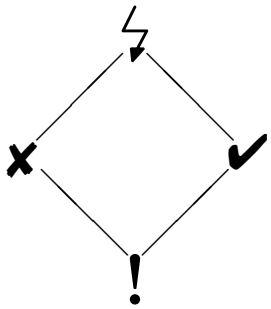


Figure 4.12: Belnap's knowledge order variant

4.5.1 Annotated Models

The classical approach in Section 4.4 provides no means to express the stakeholders' beliefs about the *fitness* of model elements, and the possible ways in which these beliefs can *evolve*. Consequently, we cannot describe *how sure* stakeholders are about each of the decisions they make. Further, we cannot express inconsistencies and disagreements that arise due to discrepancies between stakeholders' decisions about either the structure or the contents of models.

To model stakeholders' beliefs, we attach to each model element an annotation denoting the *degree of knowledge* available about the element. We formalize knowledge degrees using *knowledge orders*. A knowledge order is a partially ordered set specifying the different levels of knowledge that can be associated to model elements, and the possible ways in which this knowledge can grow.

One of the simplest and most useful knowledge orders is Belnap's four-valued knowledge order (Belnap, 1977). The knowledge order $\mathcal{K}$ shown in Figure 4.12 is a variant of this: assigning ! to an element means that the element has been *proposed* but it is not known if the element is indeed well-conceived; ✘ means that the element is known to be ill-conceived and hence *refuted*; ✔ means that the element is known to be well-conceived and hence *confirmed*; and ϟ means there is conflict as to whether the element is well-conceived, i.e., the element is *disputed* [7].

[7] Belnap's original lattice refers to ! as maybe, ✘ as false, ✔ as true, and ϟ as disagreement.

An upward move in a knowledge order denotes a growth in the amount of knowledge, i.e., an evolution of specificity. In $\mathcal{K}$, the value **!** denotes uncertainty; ✘ and ✔ denote the conclusive amounts of knowledge; and ϟ denotes a disagreement, i.e., too much knowledge – we can infer something is both ill- and well-conceived.

To augment graph-based models with the above-described annotation scheme, the definitions of graph and homomorphism are extended as follows. Let K be a knowledge order:

Definition 4.3 (annotated graph) A ***K-annotated graph*** $\mathbf{G}$ is a graph each of whose nodes and edges has been annotated with an element drawn from K.

Definition 4.4 (annotation-respecting homomorphism) Let $\mathbf{G}$ and $\mathbf{G}'$ be K-annotated graphs. A ***K-respecting homomorphism*** $\mathbf{h} : \mathbf{G} \to \mathbf{G}'$ is a homomorphism subject to the following condition: For every element (i.e., node or edge) x in $\mathbf{G}$, the image of x under $\mathbf{h}$ has an annotation which is *larger than or equal to* the annotation of x.[8]

The condition in Definition 4.4 ensures that knowledge is preserved as we traverse a mapping between annotated models. For example, if we have already decided an element in a model is *confirmed*, it cannot be embedded in another model such that it is reduced to just *proposed*, or is changed to a value not comparable to *confirmed* (i.e., *refuted*).

For a fixed knowledge order K, the merge of an interconnection diagram whose objects $\mathbf{G}_1, \ldots, \mathbf{G}_n$ are K-annotated graphs and whose mappings $\mathbf{h}_1, \ldots, \mathbf{h}_k$ are K-respecting homomorphisms is an object $\mathbf{P}$ computed as follows: First, disregard the annotations of $\mathbf{G}_1, \ldots, \mathbf{G}_n$ and merge the resulting graphs with respect to $\mathbf{h}_1, \ldots, \mathbf{h}_k$ to get a graph P. Then, to construct $\mathbf{P}$, attach an annotation to every element x in P by taking the *least upper bound*[9] of the annotations of all the elements that x represents.

Intuitively, the least upper bound of a set of knowledge degrees $S \subseteq K$ is the least

[8] For a given knowledge order K, the collection of all K-annotated graphs and K-respecting homomorphisms is characterized by the category $\mathbf{FGraph}(K, K)$ as given in Definition 3.82.

[9] See Definition 3.18.

specific knowledge degree that refines (i.e., is more specific than) all the members of S. To ensure that the least upper bound exists for any subset of K, we assume K to be a *complete lattice*[10]. The knowledge order $\mathcal{K}$ in Figure 4.12 is an example of a complete lattice.

As an example, suppose the graphs in Figure 4.8 were annotated with $\mathcal{K}$ in such a way that the homomorphisms f and g satisfied the condition in Definition 4.4. Assuming that the nodes u_1 of C, x_1 of A, n_1 of B are respectively annotated with **!**, ✔, and ✘, the annotation for the node u_1 of P, which represents the aforementioned three nodes, is calculated by taking the least upper bound of the set $S = \{!, ✔, ✘\}$ resulting in the value ϟ.

Incorporating types into annotated graphs is independent of the annotations and is done in exactly the same manner as described in Section 4.4.

4.5.2 Example I: Merging i^* Models

We can now demonstrate how to merge the i^* models of Figure 4.1. We assume models are typed using the i^* meta-model $\mathcal{M}_{i^*}$ (see Section 4.4), and will use the lattice $\mathcal{K}$ (Figure 4.12) for annotating model elements. We therefore express relationships between models by ($\mathcal{M}_{i^*}$-typed) $\mathcal{K}$-respecting homomorphisms. Figure 4.13 depicts one way to express the relationships between the models in Figures 4.1(a) and 4.1(b). For convenience, we treat 'proposed' (**!**) as a default annotation for all nodes and edges, and only show annotations for the remaining values. For example, some edges in the revised versions of Bob's and Mary's models are annotated with ✘ to indicate they are refuted.

The interconnections in Figure 4.13 were arrived at as follows. First, Sam creates a connector model **Connector1** to identify synonymous elements in Bob's and Mary's models. Notice that even if Bob and Mary happened to use the same terminology in their models, defining a connector would still be necessary because our merge framework does

[10]See Definition 3.19 and Theorem 3.20.

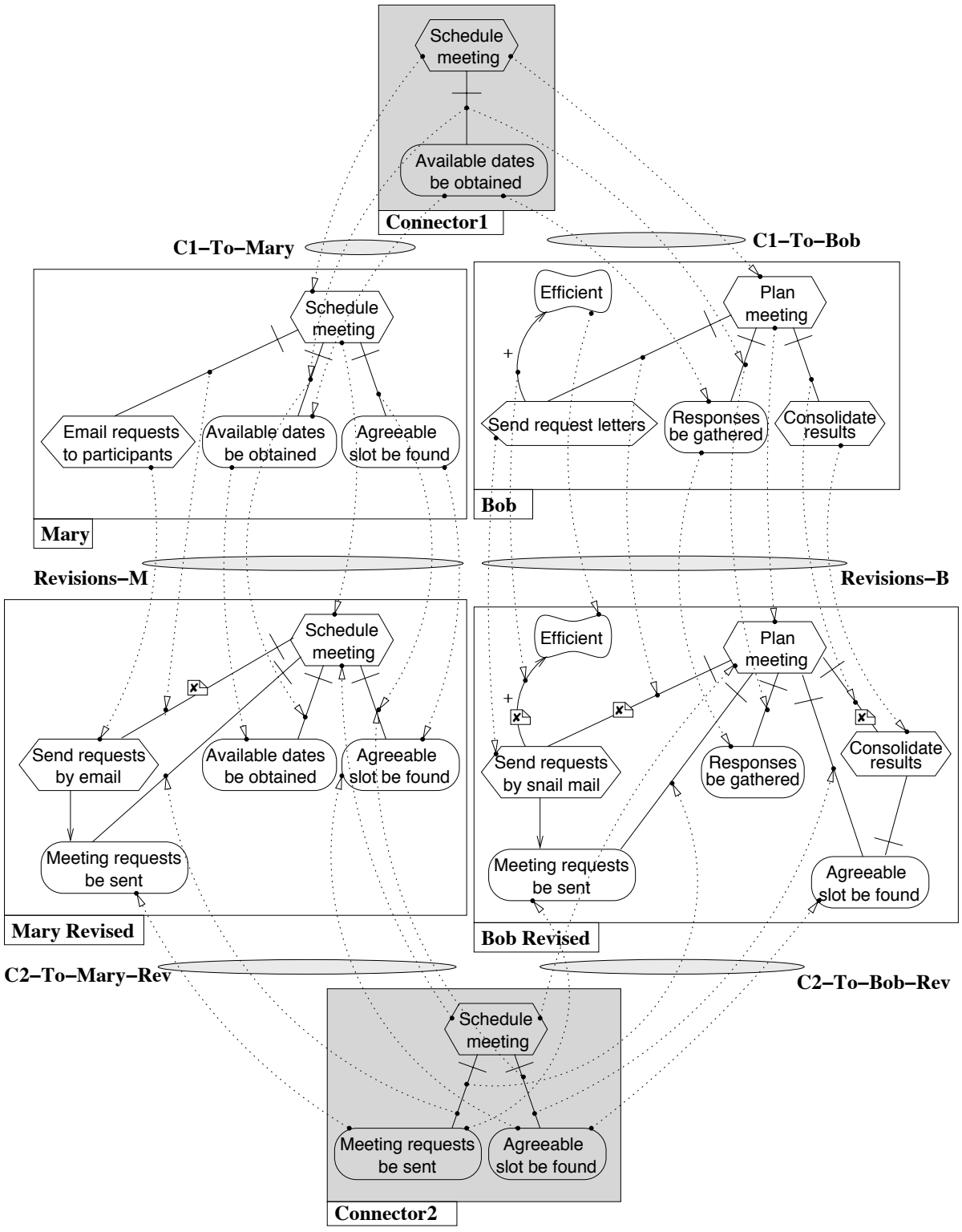


Figure 4.13: i^* example: Interconnections

not rely on naming conventions to describe the desired unifications – all correspondences must be identified explicitly prior to the merge operation.

To build **Connector1**, Sam merely needs to declare which nodes in the two models are equivalent. Because ***i**** does not allow parallel edges of the same type between any pair of nodes, the edge interconnections are identified automatically once the node interconnections are declared. For example, when Mary's Schedule meeting and Available dates be obtained are respectively unified with Bob's Plan meeting and Responses be gathered, the decomposition links between them in the two models should also be unified.

Next, Sam elaborates each of Bob's and Mary's models to obtain **Mary Revised** and **Bob Revised**. In these models, Sam has refuted the elements he wants to replace, and proposed additional elements that he needs to complete the merge. Sam could, of course, confirm all the remaining elements of the original models, but he preferred not to do so because the models are in very early stages of elicitation. Finally, Sam keeps track of cases where the same element was added to more than one model using another connector model, **Connector2**.

With these interconnections, the models in Figure 4.13 can be automatically merged, to obtain the model shown in Figure 4.14. To name the elements of the merged model priority has been given to Sam's choice of names. For presentation, we may want to *mask* the elements annotated with ✘. This would result in the model shown in Figure 4.1(c).

In the above scenario, we treated the original elements of Mary's and Bob's models as being at the *proposed* level, allowing further decisions to be freely made about any of the corresponding elements in the revised models. At any time, Mary or Bob may wish to insist upon or change their minds about any elements in their models. They can do this by elaborating their original models, confirming (or refuting) some elements. In this case, we simply add the new elaborated models to the merge hypothesis with the appropriate mappings from Mary's or Bob's original models. When we recompute the merge, the new annotations may result in disagreements. We illustrate this in Section 4.5.3.

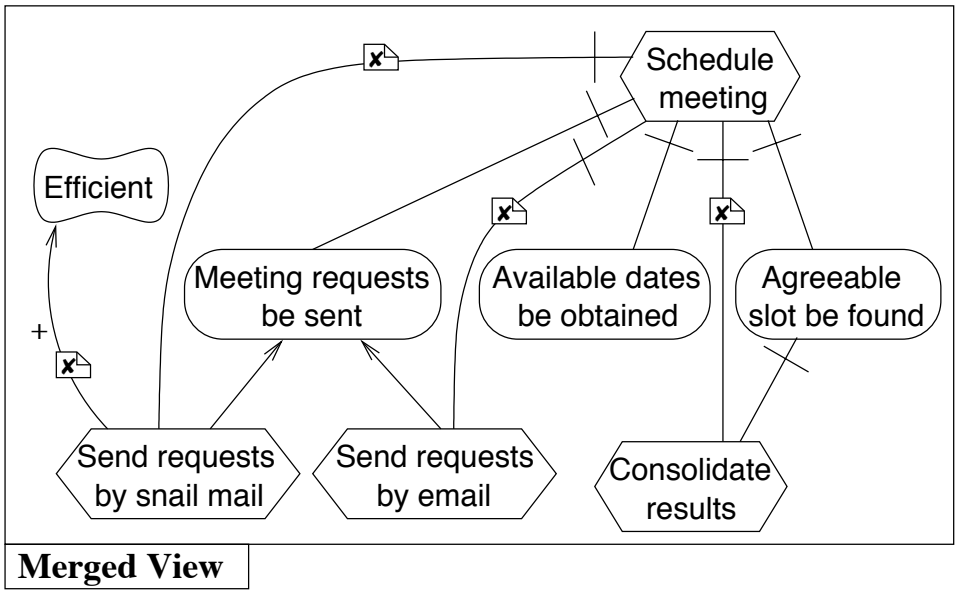


Figure 4.14: i^* example: The merged model

4.5.3 Example II: Merging Entity-Relationship Models

To merge the entity-relationship models of Figure 4.2, we assume them to be typed by a meta-model $\mathcal{M}_{\mathbf{ER}}$. We chose to omit this meta-model because the process of constructing it is similar to that described in Section 4.4 for constructing $\mathcal{M}_{i^*}$. As in the previous example, the lattice $\mathcal{K}$ (Figure 4.12) is used to annotate model elements, and 'proposed' (**!**) will be treated as a default annotation. Relationships between models will be expressed by $\mathcal{K}$-respecting homomorphisms.

First attempt: In the first iteration, Sam describes the correspondences between Bob's and Mary's models using a connector model, **Connector1** (Figure 4.15), and two mappings **C1-To-Bob** and **C1-To-Mary**. Merging the interconnection diagram made up of models **Bob**, **Mary** and **Connector1**, and mappings **C1-To-Bob** and **C1-To-Mary** yields the schema shown in Figure 4.3.

Second attempt: In the second iteration, Mary *evolves* her original model, to obtain **Mary Evolved** (Figure 4.16), addressing the concerns that occurred to her after the first

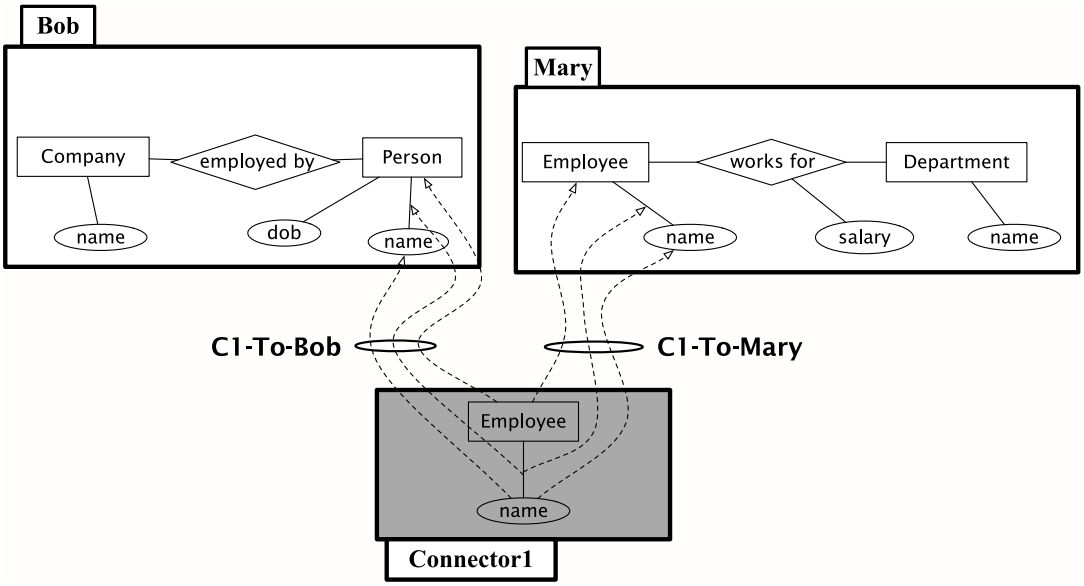


Figure 4.15: Entity-relationship example: Interconnections (Part I)

merge attempt. With Sam's help, she establishes the required interrelationships between her evolved model and Bob's original model through a new connector, **Connector2** (Figure 4.16). If we add these new models and mappings to the interconnection diagram for the first merge attempt and then recompute the merge, we get a schema (not shown) in which the employed by relationship has been refuted and an aggregation link has been introduced between Corporation and Department. Masking the refuted elements of this merge will give us the schema in Figure 4.4.

Third attempt: Finally, in the last iteration, Bob evolves his model (Figure 4.17), capturing his refined beliefs about the employed by relationship. Adding this new model and mapping leads to the full interconnection diagram shown in Figure 4.18. Merging according to this interconnection diagram yields the schema shown in Figure 4.19. In this schema, the annotation computed for the employed by relationship is 'disputed' (ϟ) because Bob and Mary have respectively confirmed and refuted the corresponding elements in their evolved models.

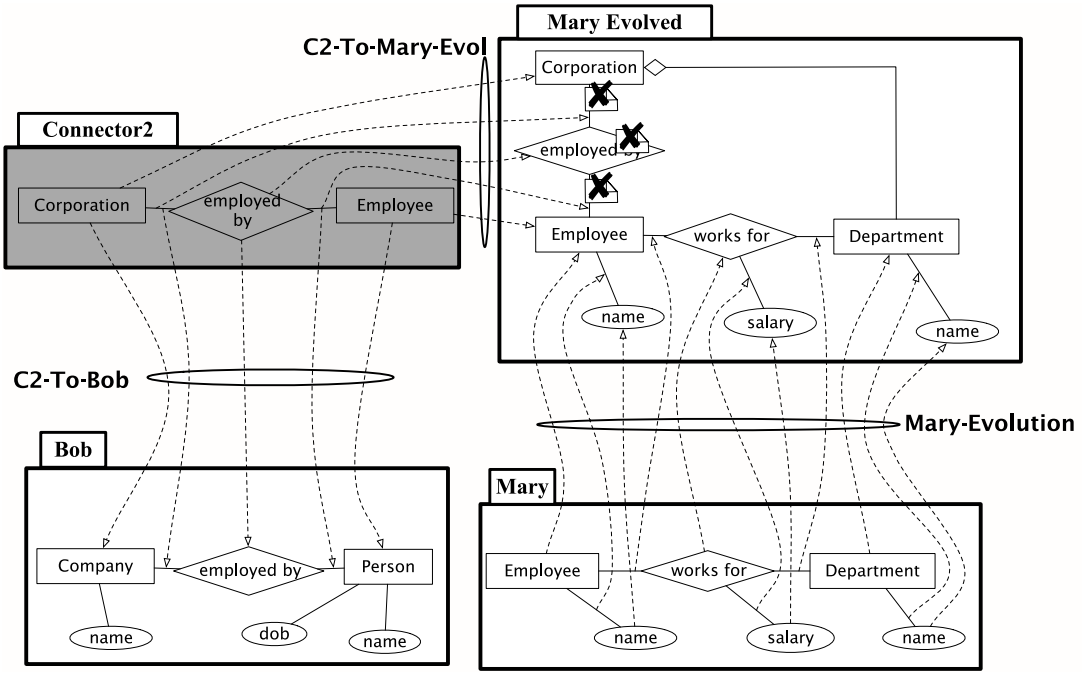


Figure 4.16: Entity-relationship example: Interconnections (Part II)

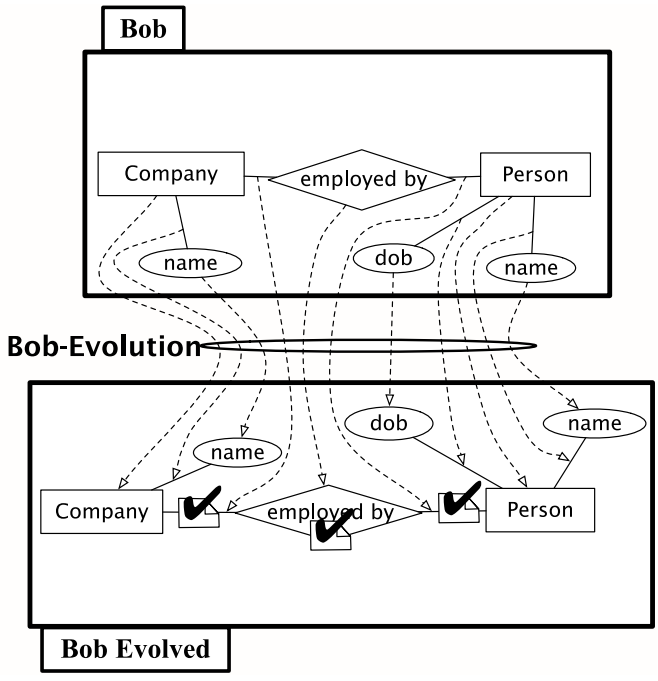


Figure 4.17: Entity-relationship example: Interconnections (Part III)

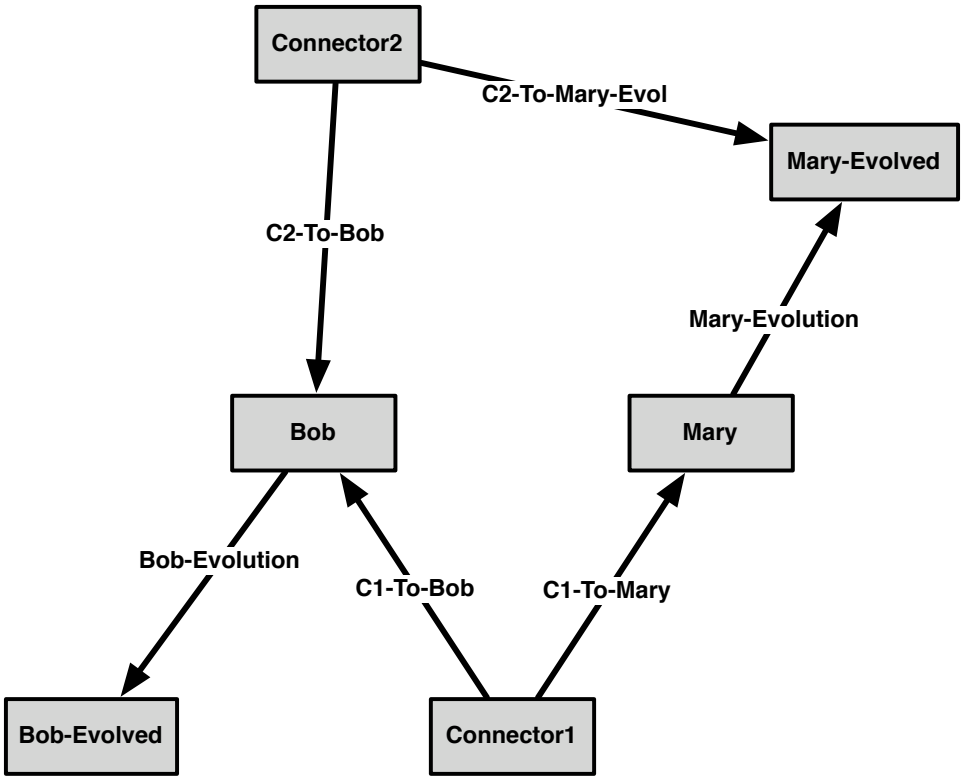


Figure 4.18: Full interconnection diagram for merge

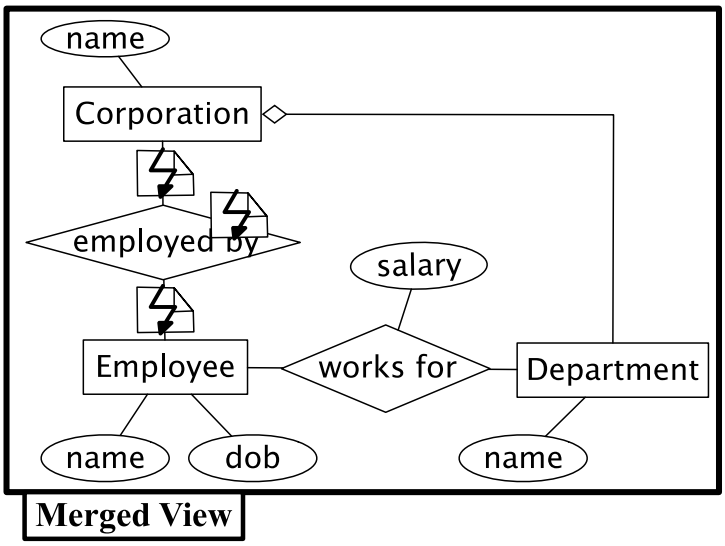


Figure 4.19: Entity-relationship example: The merged model

Since we now have a placeholder in the merge to represent the disagreement between Bob and Mary, there is no need for an immediate resolution – we can delay resolving the conflict, or if it later turns out that the problem is unimportant, we may even elect to ignore it all together.

4.6 Support for Traceability

After a merge is completed, it is often desirable to know how the source artifacts to the merge process, namely models and mappings, influenced the result. Particularly, it is important to be able to trace the elements of the merged model back to the originating models, and to track the correspondence assumptions behind each unification. We call the former notion *origin traceability* and the latter *assumption traceability* (Sabetzadeh & Easterbrook, 2005b). A third traceability notion, which we refer to as *stakeholder traceability*, arises when multiple stakeholders are allowed to work on individual models. To be able to trace decisions back to their human sources in this setting, we need to differentiate between the conceptual contributions of different stakeholders in individual models.

4.6.1 Origin and Assumption Traceability

The merges in Section 4.5 lack the traceability information required for determining where each of their elements originate. To keep track of the origins of the elements in a merge, the merge operator must store proper traceability links in the merged model.

It turns out that unification graphs, as introduced in Section 4.4.1, immediately provide the information needed for supporting origin traceability: For a given merge problem, the set of nodes in each connected component of the unification graph constitutes the origin information for the corresponding merged element. For example, the Available dates be obtained goal in Figure 4.14 should be traceable to Available dates be obtained in **Connec-**

tor1, **Mary** and **Mary Revised**, as well as to Responses be gathered in **Bob** and **Bob Revised**.

To trace the correspondence assumptions involved in a unification, we need to know the details of the interrelations among the source model elements that are unified to form an element of the merged model. In a simple scenario such as the first merge attempt in Section 4.5.3, identifying the correspondence assumptions is trivial because all these assumptions are localized to the mappings **C1-To-Mary** and **C1-To-Bob**. Therefore, if we later need to check why, for example, Person in **Bob** was unified with Employee in **Mary**, we can easily find the chain of correspondences that brought about the unification: Person(**Bob**) = Employee(**Connector1**) by **C1-To-Bob**, and Employee(**Connector1**) = Employee(**Mary**) by **C1-To-Mary**.

However, as merge scenarios get more complex, finding the correspondence assumptions becomes harder. As an example, consider the interconnection diagrams in Figures 4.15–4.17: the assumptions about correspondences between models are scattered among several mappings. For example, the unification of Company in **Bob Evolved** and Corporation in **Mary Evolved** involves **Bob-Evolution**, **C2-To-Bob** and **C2-To-Mary-Evol**; and the unification of Person's name attribute in **Bob Evolved** and Employee's name attribute in **Mary Evolved** involves **Bob-Evolution**, **C1-To-Bob**, **C1-To-Mary** and **Mary-Evolution**. More interestingly, the unification of Employee in **Bob Evolved** and Employee in **Mary Evolved** can be traced to two different correspondence chains, one involving **Bob-Evolution**, **C1-To-Bob**, **C1-To-Mary** and **Mary-Evolution**; and another involving **Bob-Evolution**, **C2-To-Bob** and **C2-To-Mary-Evol**.

The current notion of unification graph is not readily applicable to finding the correspondence chain(s) involved in creating the elements of the merged model. This is because we do not keep track of *which* mapping induces each of the edges in a unification graph. To address this, we label each edge in a unification graph with the name of the mapping that induces the edge. Figure 4.20 shows the extended unification graph for

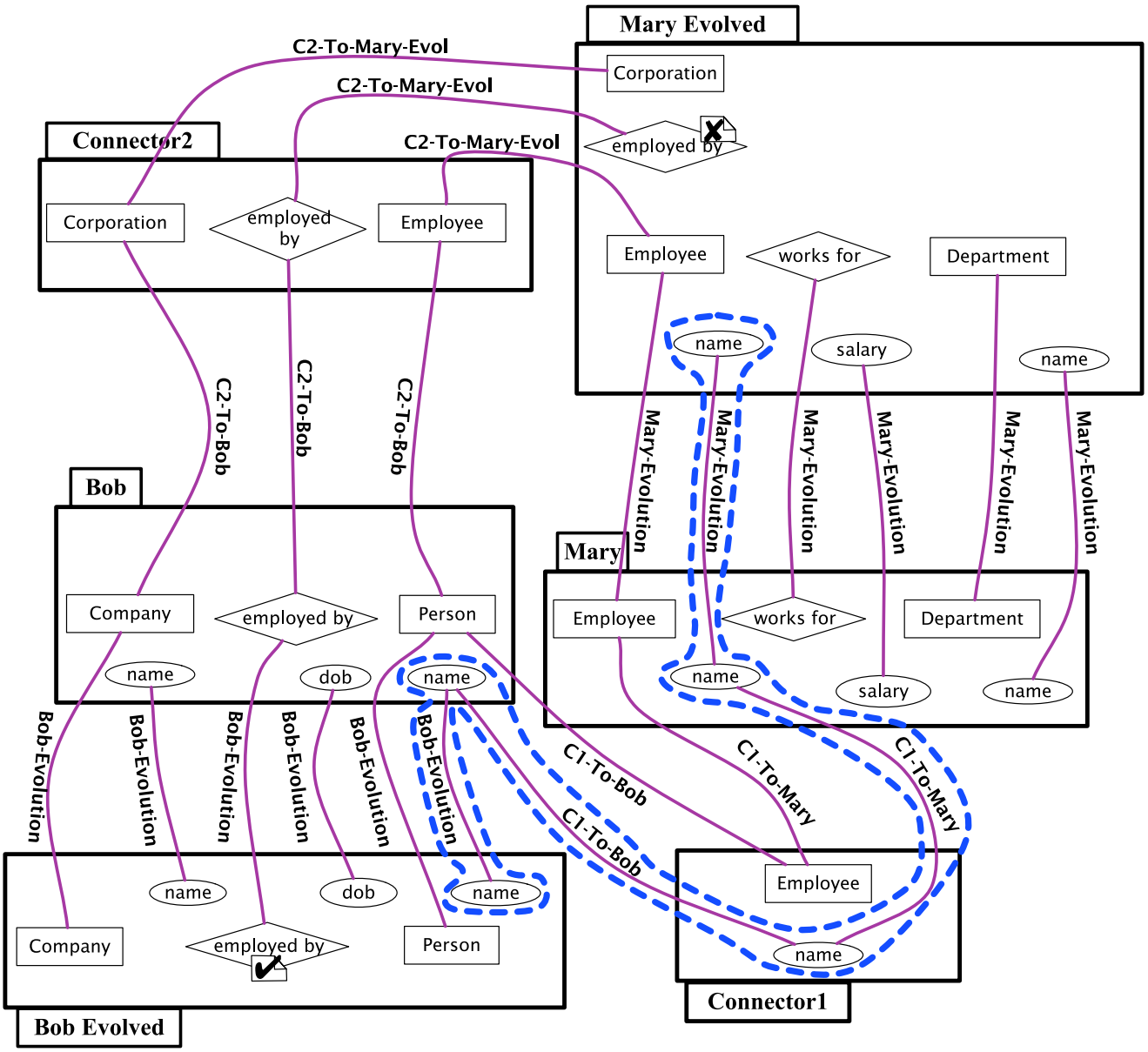


Figure 4.20: Extended unification graph for the node-sets in Figures 4.15–4.17

merging the node-sets of the models in Figures 4.15–4.17. Each connected component of this graph corresponds to one node in the merged model shown in Figure 4.19. As an example, we have explicitly shown in Figure 4.20 the connected component corresponding to the Employee's name attribute.

To support both origin and assumption traceability, we store in each element of the merged model a reference to the corresponding connected component of the extended

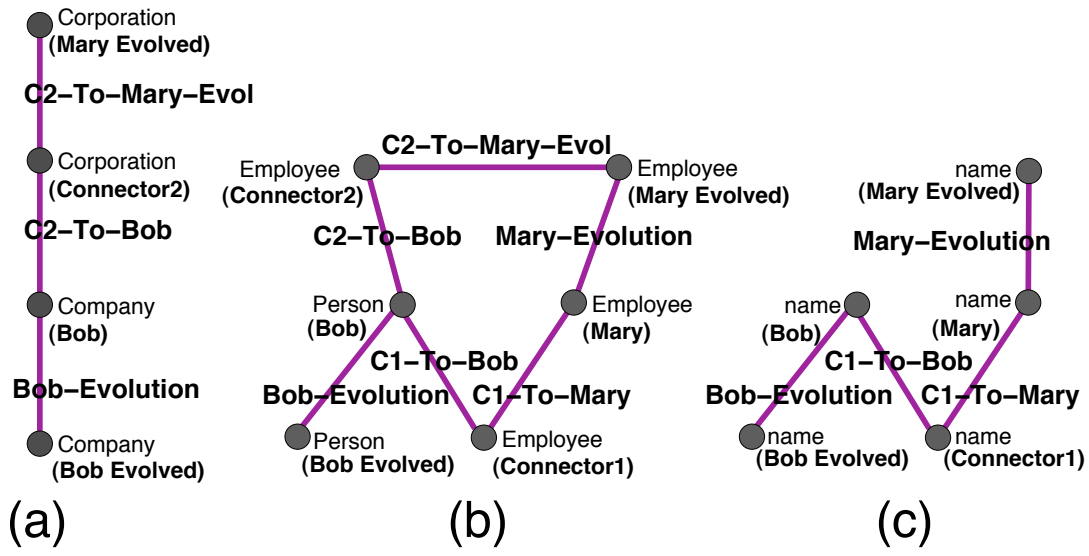


Figure 4.21: Examples of traceability links

unification graph. Figures 4.21(a)–(c) respectively show the stored traceability information for three representative elements of the merged model in Figure 4.19: Corporation, Employee, and Employee's name. In each case, the traceability information makes it possible to trace the respective element back to its origins, and to the related assumptions in the unification. If we want to see why, for example, Employee in **Mary Evolved** was unified with Person in **Bob Evolved**, we find the *(non-looping) paths* between Employee (**Mary Evolved**) and Person (**Bob**) in Figure 4.21(b).

To avoid clutter, we chose not to show the element uid's in Figure 4.21; however, we should emphasize that uid's need to be kept in the traceability links in order to avoid ambiguity, because an element may not be uniquely identifiable by its name.

4.6.2 Stakeholder Traceability

When collaborative work is allowed on an individual model, we can no longer assume that all contributions in a given model come from a single human source. The framework developed in Section 4.5 does not support collaborative work on models because the knowledge labels do not indicate *whose* knowledge is being captured; therefore, we have

to assume all contributions in a given model belong to a single stakeholder.

To support tracing contributions back to their human sources when individual models are collaboratively developed, we introduce a more elaborate annotation scheme: Rather than annotating model elements with single annotations, we attach an annotation-set to each element. Each annotation in the annotation-set has a qualifier denoting the stakeholder whose belief is captured by that annotation.

To ensure that the annotation-set X_e attached to a model element e evolves properly along a mapping **h**, the following condition must be met: Every stakeholder who has an annotation in X_e must have an annotation in the annotation-set of e's image under **h**, and this annotation must be at least as specific as that in X_e. Notice that this condition does not prevent **h** from introducing annotations for stakeholders who do not already have an annotation in X_e – what is required is that the evolution of already-existing annotations along **h** must respect the knowledge order.

To illustrate the new annotation scheme consider the ***i**** merging example: Sam, Mary, and Bob can now manipulate each others' models without compromising traceability. This is because the annotations can keep track of the contributions of individual parties. The new system of interrelated models is shown in Figure 4.22. We use a concise notation to represent the annotation-set for each element. For example, **M:!;B:!** means that both Mary and Bob proposed the element; **B:!;S:✘** means Bob proposed the element and Sam refuted it. Note that in the **Connector** model in Figure 4.22, the elements have no stakeholder annotations, indicated using $\emptyset$. If we were interested in tracking the revisions Sam makes to Bob's and Mary's vocabularies, we would need to use the same interconnection pattern as that in Figure 4.13, but the model elements would be annotated with annotation-sets instead of single knowledge degrees.

Merging the interconnected models in Figure 4.22 yields the model shown in Figure 4.23. The annotation-set for each model element e in the figure is computed by first unioning the stakeholders that have contributed to the elements represented by e; and

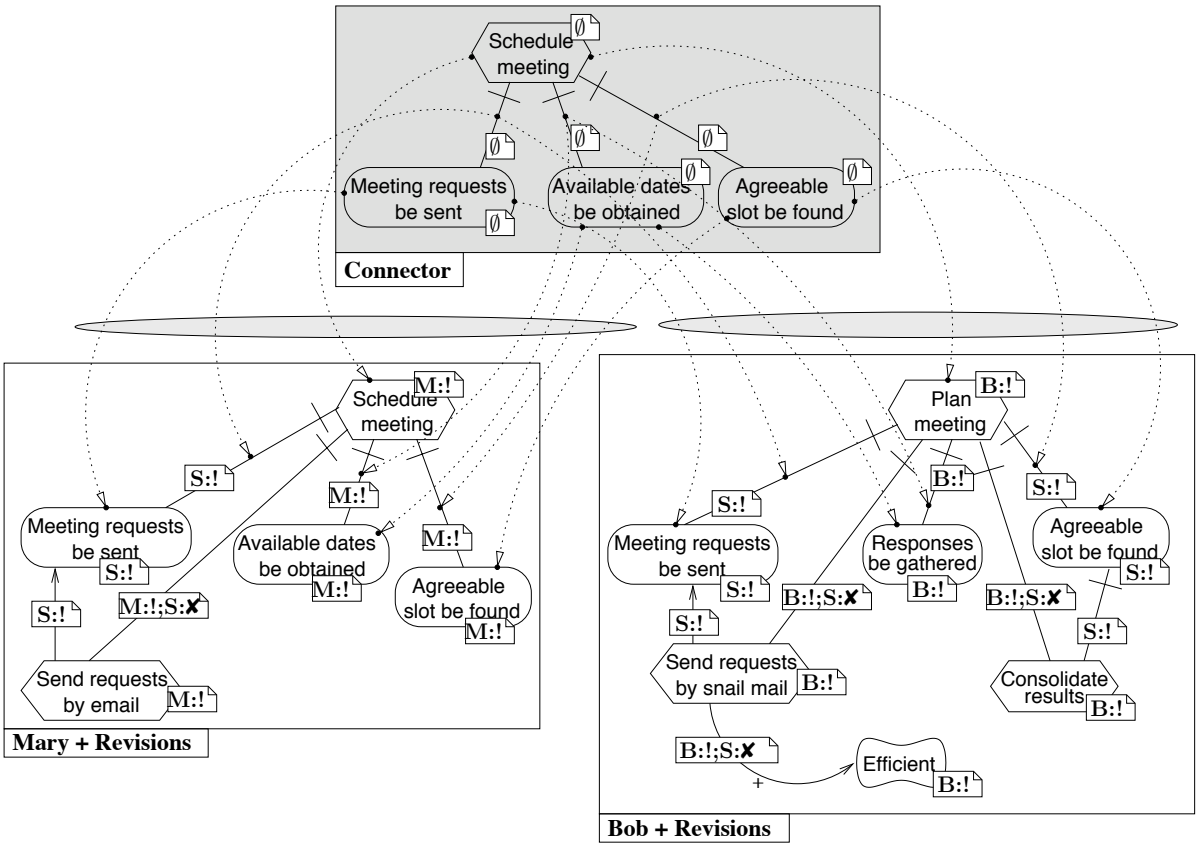


Figure 4.22: New model interconnections

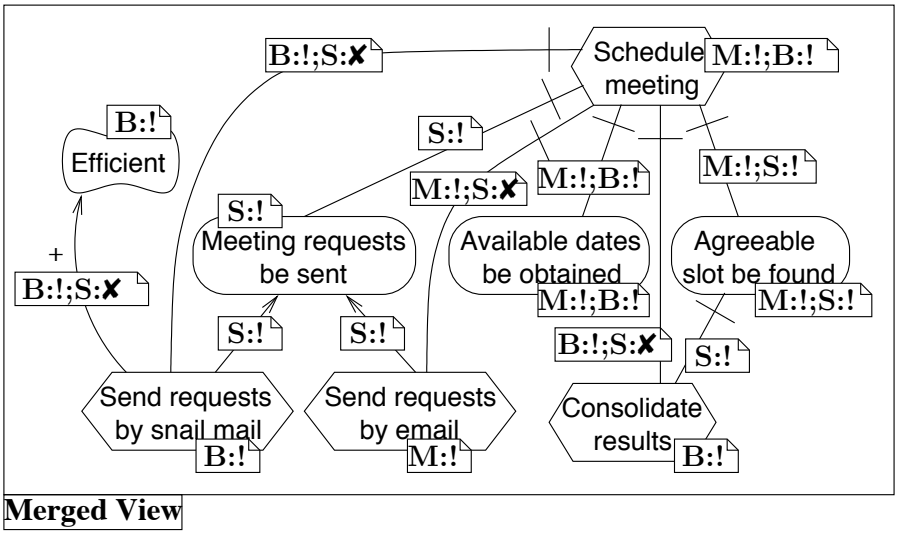


Figure 4.23: Merged model with detailed annotations

then, for every stakeholder s in the union, taking the least upper bound of s's contributions to these elements[11].

The annotation for each element of the merged model in Figure 4.23 reflects the decisions made about the element by the involved stakeholders. Note that when stakeholders do not work on source models collaboratively, origin traceability subsumes stakeholder traceability; that is, we can produce merges with annotations like those in Figure 4.23 based on origin traceability information.

4.7 Discussion

In this section, we discuss some practical considerations concerning our merge framework.

4.7.1 Complexity

Given an interconnection diagram with models $\mathbf{G}_1, \ldots, \mathbf{G}_n$ and mappings $\mathbf{h}_1, \ldots, \mathbf{h}_k$, the space complexity of our framework is dominated by the space required to construct the unification graphs for merging the node-sets and edge-sets of the models. This gives us a space complexity of $O\left((\alpha = \Sigma_{i=1}^{n}|\mathbf{G}_i|) + \left(\beta = \Sigma_{i=1}^{k}|\mathbf{h}_i|\right)\right)$. By $|\mathbf{G}_i|$, we mean the number of nodes and edges of $|\mathbf{G}_i|$; and by $|\mathbf{h}_i : \mathbf{G} \to \mathbf{G}'|$, we mean $|\{(e, \mathbf{h}_i(e)) \mid e \in \mathbf{G}\}|$. Since $\mathbf{h}_i$'s are total functions, we have $|\mathbf{h}_i : \mathbf{G} \to \mathbf{G}'| = |\mathbf{G}|$ for every $\mathbf{h}_i$. The worst-case space complexity of our framework is in the (unrealistic) situation where there is a mapping from every model to all models. In this situation, the space complexity is $O\left(n \times \alpha\right)$.

To determine the time complexity of our framework, we note that the dominant part is unification of elements related by the mappings and computing an appropriate annotation for each unified element. Unification is done by finding the connected components of the (node and edge) unification graphs, in time $O(\alpha + \beta)$. Let K be the knowledge order

[11] This is an informal description of how to compute a least upper bound in a fuzzy powerset lattice. See Theorem 3.80.

from which element annotations are drawn, and let $|K|$ denote the size of the graph representing the covering relation[12] of K. The complexity of computing annotations for the elements of the merged model is $O\left(|K|^2 \times \alpha\right)$, noting that each stakeholder contributes only a single value from K, and that the least upper bound of any two K values can be computed in time $O(|K|^2)$.

If we allow model elements to be annotated with beliefs from multiple stakeholders (see Section 4.6.2), both time and space complexity will be multiplied by the number of stakeholders.

Assuming that the annotation lattice is fixed, and that the number of stakeholders is bounded by some fixed number, we conclude that the space and time complexity of our framework are *linear* in the size of the source models and mappings.

4.7.2 Constraint Checking

The typing mechanism discussed in Section 4.4 is not expressive enough to describe all the well-formedness constraints that need to be enforced over models. For example, in the class diagram example in Figure 4.9, we could not express the constraint that a Java class is not allowed to have multiple parent classes, or that a class cannot extend its subclasses: in both cases, a typing map could be established even though the resulting class diagrams were unsound. To express the former constraint, we would have to require that the class inheritance hierarchy be a many-to-one relation; and to express the latter, we would have to require that the inheritance hierarchy be acyclic. Similarly, our framework cannot express and detect anomalies caused by the annotations. For example, in Section 4.5, it was possible for a model to have a non-refuted edge with a refuted end. In such a case, we would be left with dangling edges if we mask refuted elements. In Chapter 5, we provide a general logic-based platform that enables expressing and checking constraints like the ones mentioned above.

[12] See Definition 3.16.

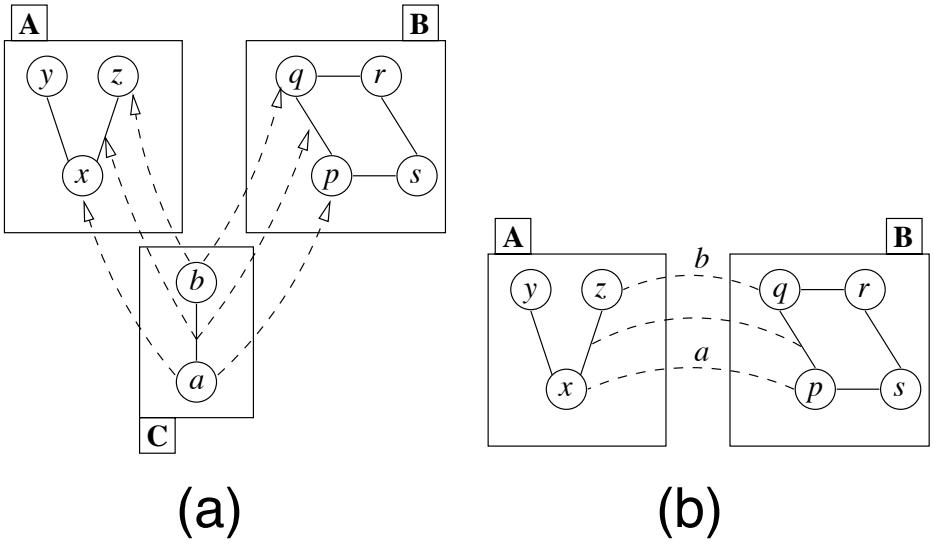


Figure 4.24: Connectors versus direct mappings

4.7.3 Connectors versus Direct Mappings

In our framework, correspondences between independent models are captured using explicit connector models. For example, to describe the overlap of a pair of models A and B, we create a connector model C that captures just the common parts between A and B, and then specify how C is represented in each of the two models using the mappings $C \rightarrow A$ and $C \rightarrow B$, as illustrated in Figure 4.24(a).

Our use of connector models was motivated by theoretical concerns. Briefly, this approach allows us to build the mappings between models using graph homomorphisms – each mapping shows how one model is embedded in another. This in turn allows us to treat models and model mappings as a category, giving us a straightforward way to construct model merges (as colimits), along with standard proofs (e.g., the proof that for this category, the merge always exists and is unique for any set of interconnected models).

One could argue that this approach is less appealing than specifying correspondences between A and B by linking their shared elements directly. Such a scheme could handle naming preferences by encoding the names in labelled binary relations, as illustrated in

Figure 4.24(b). Note that the mapping in Figure 4.24(b) is partial, and does not map the entire structure of one model onto the other.

It would be possible to hide the use of connector models from the user and present them as direct links instead. In fact, we are going to do so in Chapter 5, where we apply our merge technique for consistency checking. Nonetheless, we must emphasize that the use of explicit connector models offers several important methodological advantages when the maintenance overhead imposed by them is justified:

- It allows us to clearly distinguish distinct areas of overlap for a set of models, by using a separate connector model for each area of overlap.
- It generalizes elegantly for cases where more than two models share an overlap, and for cases where we want to indicate overlaps between the connector models themselves.
- It provides a more flexible platform for incorporating various types of preferences into the merge process. A simple example is layout preferences: we may want to choose a certain layout for the parts that are in common between a set of models and preserve that layout in the merge. Layout preferences require explicit models for the shared parts.
- Connectors can be used as requirements baselines when the scope of the elicitation process widens. For example, if the elicitation process starts with two stakeholders and a new stakeholder emerges later on, it is natural for the third stakeholder to use the connectors between the models of the first two stakeholders as baselines for further elaboration of his/her own models.

4.7.4 Information Gaps and Inexact Correspondences

As we already noted in Chapter 2, there may be information gaps between the source models that need to be addressed during merge. For example, the goal Meeting requests

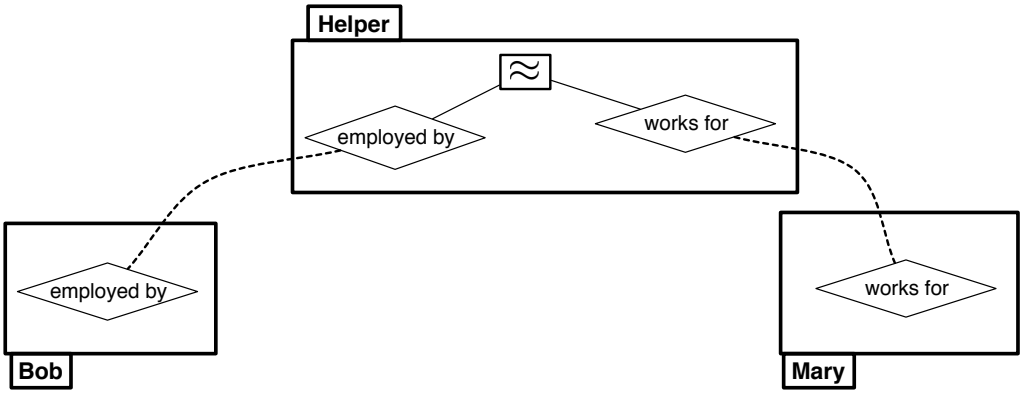


Figure 4.25: Capturing similarity relationships

be sent in Figure 4.1(c) was not explicitly stated by either Mary or Bob in their models (shown in Figures 4.1(a) and 4.1(b)), but Sam had to explicate this unstated goal during merge so that the tasks Email requests to participants and Send request letters could be related. Since our framework can merge multiple models at a time, such gaps can be bridged by evolving the source models, just as we did in the interconnection diagram of Figure 4.13. Alternatively, one can build new models that provide only the missing information and then add these new models to the interconnection diagram for the merge.

Further, from time to time, we may want to express relationships other than exact correspondences. For example, in the schema merging scenario of Section 4.5.3, one could say that the employed by and works for elements are similar (but not equivalent). One way to state such relationships is by extending the modelling language being used with new modelling constructs. Entity-relationship diagrams, for example, have been extended with a special notation, $\approx$, to denote similarity relationships (Pottinger & Bernstein, 2003). Figure 4.25 shows how one can declare the similarity relationship between employed by and works for by reifying the relationship into a new model, **Helper**. The overlaps between **Helper** and the stakeholders' models (only the relevant fragments of which are shown) are captured through exact correspondence mappings. Note that, in light of the discussion in Section 4.7.3, we have used partial binary relations for describing the mappings between the models of Figure 4.25.

4.8 Related Work

Our merge framework is primarily intended to support viewpoint-based modelling. Viewpoints (or views) have been used in the literature in a variety of ways, e.g., to distinguish different users and terminologies (Ross, 1985; Stamper, 1994), to encapsulate knowledge about a system into manageable pieces (Finkelstein *et al.*, 1992), to specify the contexts in which different roles are performed (Easterbrook, 1993), to provide separation of concerns (Nuseibeh *et al.*, 1994), to improve use of abstraction (Egyed, 2000), and so on. For a comparative survey of viewpoint-based techniques, see (Darke & Shanks, 1996).

The underlying principle in viewpoint-based modelling is that, for a given problem, it is better to build several fragmentary models rather than to attempt to construct a single coherent model (Easterbrook *et al.*, 2005). Despite the general desire to maintain viewpoints as separate loosely-coupled objects (Finkelstein *et al.*, 1994), we often need to merge a set of viewpoints in order to explore the relationships between them. Our work on model merging aims to provide a flexible solution for performing such exploration.

As we saw in Chapter 2, many model merging approaches already exist for database schemata, ontologies, and various software engineering models. Our work is distinguished from existing work in two key respects: First, our framework directly applies to systems with an arbitrary number of models and mappings, whereas existing work typically considers only the three-way merge pattern. Generalizability to arbitrary systems comes from the use of category theory and colimits. Hence, in principle, other category-theoretic approaches surveyed in Chapter 2 offer this flexibility as well. However, none of these approaches identify the need for defining merge as an operation over arbitrary systems of models, and therefore, lack a general methodology for applying colimits. In our work, we demonstrate that systems of models arise naturally when modelling is done in an exploratory setting, where models are elaborated incrementally and are continuously evolving. We believe the application of merge for exploratory analysis is novel and has not been considered before.

The second major factor distinguishing our work from existing work is that we can explicitly model incompleteness and inconsistency and enable deferring their resolution. This is in contrast to the body of work surveyed in Chapter 2 where incompleteness and inconsistency are not tolerated and need to be resolved as soon as discovered.

The approach we proposed in this chapter treats models as graphical artifacts and provides a generic algebraic characterization of the merge operation. This makes the approach generalizable to many different modelling notations, and is particularly suited to earlier stages of development, e.g., requirements elicitation, when models usually have loose or tacit semantics. In contrast, our more recent work on behavioural models, (Nejati *et al.*, 2007), has concentrated on providing a *logical* characterization of the merge operation. This recent work has the advantage that it preserves the logical properties of models during merge; however, it is largely inapplicable to structural models such as goal models, entity-relationship diagrams, and class diagrams, because the work is specifically directed towards behavioural properties of models.

The use of multiple-valued logics for merging incomplete and inconsistent models was first proposed in (Easterbrook & Chechik, 2001) and later generalized in (Sabetzadeh & Easterbrook, 2003). These earlier papers, which serve as a precursor to the ideas developed in this chapter provide a formal treatment of incompleteness and inconsistency. However, they were conceived as part of a framework for supporting automated reasoning over state machines. In this chapter, we instead focused on devising a general foundation for model management and defining a mathematically rigorous merge operator within that foundation.

The ability to trace requirements back to their human sources is cited as one of the most important traceability concerns in software development (Gotel & Finkelstein, 1997). To this end, contribution structures (Gotel & Finkelstein, 1995) have been proposed as a way to facilitate cooperative work among teams and to ensure that the contributions of involved parties are properly accounted for throughout the entire development

life-cycle. The notions of origin and stakeholder traceability in our work try to address a similar problem in the context of model merging.

The importance of establishing traceability links between artifacts and the assumptions involved in creating them has been emphasized in the literature on design rationale (Fischer *et al.*, 1996; Gruber & Russell, 1996) and design traceability (Egyed, 2001). However, the focus in these lines of work is on assumptions that relate upstream and downstream artifacts, i.e., vertical traceability. Our work, instead, focuses on horizontal traceability. We discuss the nature of the relationships between collaboratively developed models in a particular stage of development, and provide a technique for keeping track of how assumptions made about model interconnections affect the merge.

4.9 Summary and Future Work

We have presented a flexible and mathematically rigorous framework for merging incomplete and inconsistent models. Our merge framework is general and can be applied to a variety of graphical modelling languages. In this chapter, we presented the core algorithms for computing merges, showed how the framework can handle typing constraints, and how to trace contributions in the merged model back to their sources and to the relevant merge assumptions. We have implemented the algorithms described in this chapter. We describe our implementation in Chapter 6.

An advantage of our framework is the explicit identification of interconnections between models prior to the merge operation rather than relying on implicit conventions to give the desired unification. We believe this offers a powerful tool for exploring inconsistency during exploratory modelling, as it allows an analyst to hypothesize possible interconnections for a set of models, compute the resulting merged models, and trace between the source and the merged models to analyze the results.

Our work can be continued in many directions. A major part of our ongoing research

is to incorporate a model matching operator into our framework. Currently, we assume that model interconnections are identified manually. A natural question to ask is to what extent this task can be automated. Existing matching techniques, e.g., (Bernstein *et al.*, 2004; Aumueller *et al.*, 2005; Mandelin *et al.*, 2006), only consider the three-way pattern. That is, given a pair of models and an incomplete connector, they are concerned with identifying the missing correspondences between the model pair and completing the connector. The problem we would like to address is more general: given an incomplete interconnection diagram over *multiple* models, how can we identify the unstated correspondences and complete the diagram? We anticipate that model merging will play a crucial role in answering this question. In particular, one may be able to identify likely matches by first constructing a merge for an as-yet incomplete interconnection diagram, and then analyzing the resulting merge for redundancies.

Another interesting topic for future work is traceability. Our current work on traceability, reported in this chapter, only deals with the merge operator. To be able to trace between the results of a complex model management manipulation and the source models and mappings, *every* model management operator involved in the manipulation must establish proper traceability links between its operands and its output. In our future work, we aim to study the traceability concerns of model management in a broader context. A related problem to traceability is round-trip engineering. To effectively maintain a set of evolving artifacts in a model management problem, it is important to be able to propagate changes made to one artifact to other related artifacts. In future research, we would like to look into ways to support round-trip engineering in our framework.

Chapter 5

Consistency Checking via Model Merging

In this chapter, we present an approach for consistency checking of distributed models. The key idea behind our approach is to employ model merging to reduce the problem of checking *inter*-model consistency to checking *intra*-model consistency of a merged model. We report on experimental results validating the feasibility and usefulness of our approach.

5.1 Introduction

An important activity in distributed development is consistency checking. As we saw in Chapter 2, consistency is a broad term and may have different interpretations in different contexts. In this chapter, we take consistency to mean *conformance* (see Section 2.2). Hence, consistency checking in our work means verifying models and their mappings against a desired set of properties and generating proper diagnostics if a violation occurs. We concentrate on *structural* properties. These include, among others, well-formedness constraints for the notation being used, and quality constraints, derived from best development practices, for improving understandability, maintenance, and reuse.

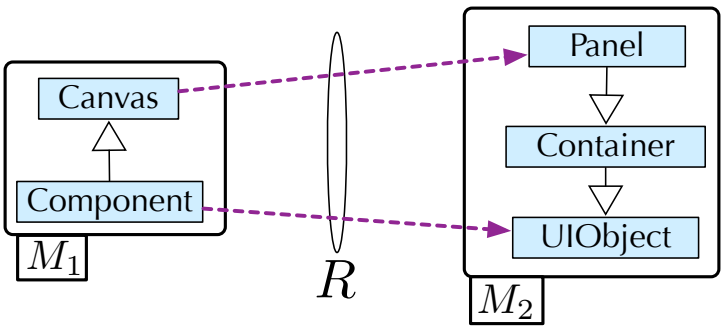


Figure 5.1: Pairwise inconsistency

For example, if models are expressed as class diagrams, well-formedness constraints disallow cyclic inheritance in individual models. Additionally, the mappings defined between models have to respect the acyclic inheritance constraint. To illustrate, consider the class diagrams M_1 and M_2 in Figure 5.1 and the mapping R in the figure, specifying the overlaps between the two models. In this example, R was defined by hand. Larger problems require automation, usually achieved by (1) name matching if models have a common vocabulary, (2) identifier matching if models have a common ancestor, or (3) heuristic matching if models are independently-developed. Regardless of how the mapping R is built, it is important to be able to check whether it respects the consistency properties of interest, in this case acyclic inheritance.

Although M_1 and M_2 are acyclic individually, they are not pairwise consistent because R gives rise to a cycle. Specifically, Component inherits from Canvas $=_R$ Panel which is a descendant of UIObject $=_R$ Component. This indicates either that we misunderstood the nature of the overlaps between M_1 and M_2, or that there is a real disagreement between the models.

The example in Figure 5.1 only considers pairwise consistency, i.e., consistency of a pair of models with respect to a single mapping between them. In practice, we are often faced with systems that have *many* models interrelated by *many* mappings. Therefore, we not only need to check pairwise consistency, but also the consistency of a system as a whole. This is known as *global consistency checking*.

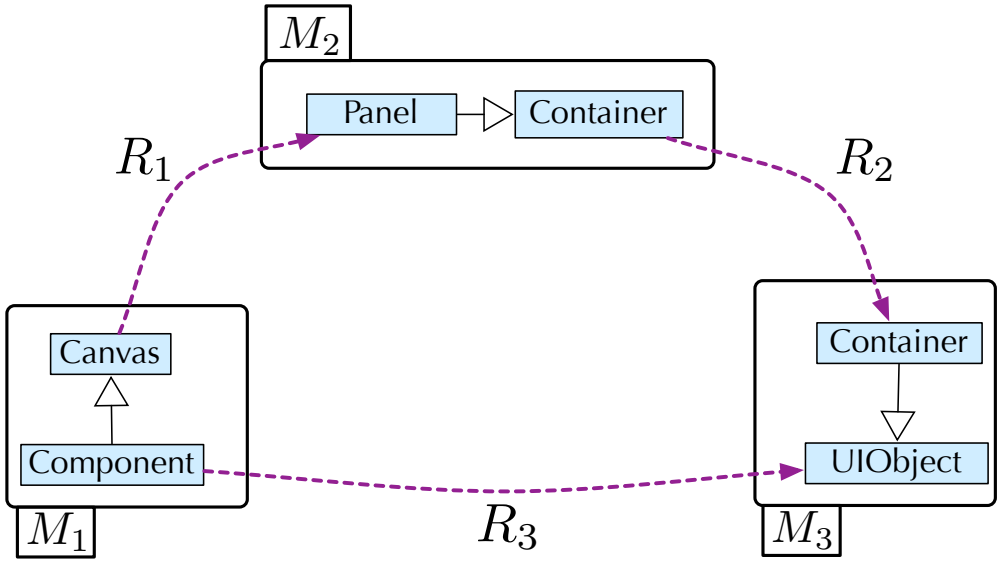


Figure 5.2: Consistency checking of a *system* of interrelated models

Global consistency checking cannot be done through pairwise checking (van Lamsweerde *et al.*, 1998; Nuseibeh *et al.*, 2001). For example, consider the system in Figure 5.2. The models in the figure are pairwise consistent when checked against the acyclic inheritance constraint. But the system is globally inconsistent because the combination of the information provided by R_1, R_2, R_3 implies a loop in the inheritance chain. Specifically, Component inherits from Canvas $=_{R_1}$ Panel which inherits from Container (in M_2) $=_{R_2}$ Container (in M_3) which inherits from UIObject $=_{R_3}$ Component.

This example underscores the need for a consistency checking technique that can *simultaneously* use information from multiple models and mappings. Existing approaches work well for pairwise checking. However, since these approaches do not clearly separate consistency rules from model mappings, it becomes very difficult to generalize the rules beyond pairwise checking.

We propose a consistency checking approach that addresses the above problem for homogeneous models, i.e., models described in the same notation. Our work is motivated by the observation that consistency checking of a set of (homogeneous) models can be done in a more general and succinct way if we first merge the models according to the

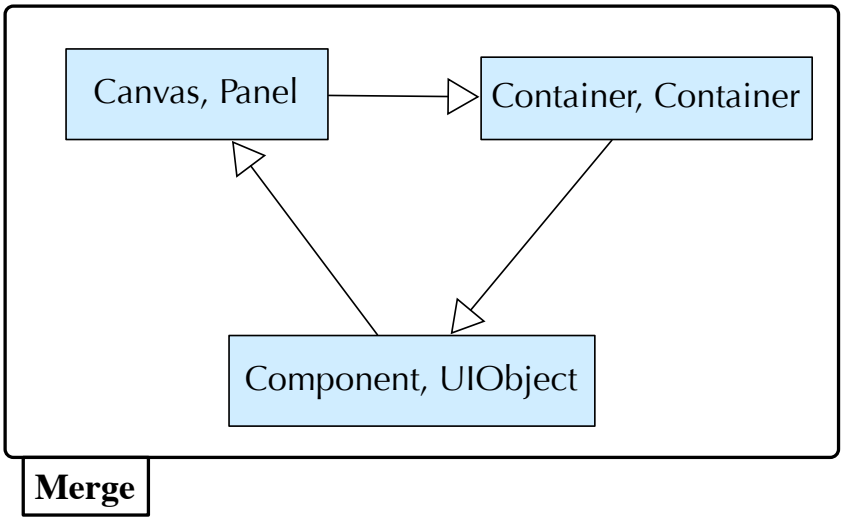


Figure 5.3: Merge of the models in Figure 5.2

mappings defined between them. The implementation of this approach requires a merge operator that is well-defined for *any* system of interrelated models even when they are *inconsistent*. We already developed such a merge operator in Chapter 4.

For example, rather than trying to check the acyclic inheritance constraint on the system in Figure 5.2, we construct a merge, shown in Figure 5.3, and interpret the constraint over it. By keeping proper traceability information, we project the diagnostics obtained from consistency checking of the merge back to the source models and mappings.

In addition to providing a solution for verifying global consistency properties, our approach has the advantage that it requires only a single consistency rule to be developed for each consistency constraint – the rule applies no matter how many models are involved and how they are interrelated with one another.

We provide an implementation of our approach within a logic-based constraint specification framework. To simplify the specification of consistency rules, we develop a set of generic and reusable expressions capturing recurrent patterns across the constraints of different modelling notations. We demonstrate the usefulness of these expressions for describing constraints over class and entity-relationship diagrams, and ***i**** goal models (Yu, 1997).

5.2 Background

Like in Chapter 4, our focus is on models with graph-based notations. However, whereas we were there concerned with manual inspections and negotiation over these models, we are here interested in complementary techniques for *automated* analysis of the models. The kinds of automated analysis that motivate our work are largely orthogonal to the belief annotation scheme we introduced in Chapter 4. Therefore, for capturing models in this chapter, we use typed graphs without annotations, as defined in Section 4.4.3.

We assume every model element i (i.e., node or edge) has an implicit and immutable attribute, called uid, that uniquely identifies i. We write $i.\mathsf{uid}$ to refer to the value of i's uid. To capture properties such as label, stereotype, colour, order, etc., we attach to every element a set of (Property, value) tuples.

5.3 Relational Specification

Consistency constraints are commonly expressed as logical formulas. An important step in developing a consistency checking framework is hence choosing an appropriate logical formalism for specification of consistency rules.

We use the Relational Manipulation Language (RML) (Beyer *et al.*, 2005) for specifying consistency rules. The logical core of RML is first order logic augmented with transitive closure and counting operators. Transitive closure enables capturing reachability constraints that are otherwise inexpressible in first order logic (Libkin, 2004). Counting, in the form used in our work, does not offer additional expressive power; but, having an explicit operator for counting provides a convenient shorthand for writing constraints that involve multiplicities.

As we demonstrate in this chapter, first order logic with transitive closure, also known as transitive closure logic, provides a rich basis for expressing consistency constraints; and yet, the logic is tractable enough to be applicable to large modelling problems (see

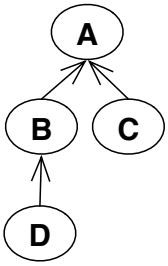
Theorem 3.99). Our focus in this chapter is on structural constraints, but our work can be generalized to behavioural constraints as well. In particular, it is known that any formula written in Computation Tree Logic (CTL) (Clarke *et al.*, 1999) can be translated, in linear time, into an equivalent transitive closure logic formula (Immerman & Vardi, 1997). This makes it possible to use the consistency checking framework to be presented in this chapter as a CTL model checker, by translating CTL properties into transitive closure logic before evaluating the properties.

In addition to meeting the expressive power requirements in our work, there are two factors that make RML particularly appealing:

- RML's domain-independent and easy-to-use syntax;
- RML's highly-optimized interpreter, CrocoPat.

CrocoPat encodes relational predicates as Binary Decision Diagrams (BDDs) (Bryant, 1986) which are compact data structures for representing and manipulating relations. The use of BDDs makes CrocoPat highly scalable in terms of both time and memory.

RML has an imperative style of execution and runs programs statement by statement. A partial and slightly simplified grammar of the language is shown in Figure 5.4. The complete grammar is given in (Beyer & Noack, 2004). For example, consider the graph shown at right, and assume that the relation E(x,y) denotes "there is an edge from x to y". To check if there exists a node without any outgoing edges, we use the existential and universal quantifiers, EX and FA in Figure 5.4, to define the expression EX(x, FA(y, !E(x, y))). This expression holds over our example graph, witnessed by node A.



For another example, suppose we want to count the number of predecessors of A. The expression #(E(x, "A")), for a free variable x, returns the number of all assignments ℓ to x for which E(ℓ, "A") holds. Hence, the expression evaluates to 2. The following program prints to the standard output all these satisfying assignments, i.e., B and C:

stmt	::=	rel_var(*term*, ...) ; |
		rel_var(*term*, ...) := *rel_expr* ; |
		IF *rel_expr* { *stmt* ... } ELSE {*stmt* ... } |
		FOR var IN *rel_expr* {*stmt* ... } |
		WHILE *rel_expr* { *stmt* ... } |
		PRINT *print_expr*;
rel_expr	::=	rel_var(*term*, ...) | *rel_expr* op_1 *rel_expr* |
		FA(var, *rel_expr*) | /* $\forall$ *quantification* */
		EX(var, *rel_expr*) | /* $\exists$ *quantification* */
		TC(*rel_expr*) | /* *transitive closure* */
		!*rel_expr* | /* *negation* */
		rel_expr op_2 *rel_expr* | *num_expr* op_2 *num_expr*
num_expr	::=	num_literal | *num_expr* op_3 *num_expr* |
		#(*rel_expr*) /* *counting* */
print_expr	::=	*rel_expr* | *term*
term	::=	var | str_literal | STRING(*num_expr*)

(1) op_1 can be one of: '|' (or), '&' (and), '–>' (implies).

(2) op_2 can be one of: '=', '!=', '<', '>'.

(3) op_3 can be one of: '+', '-', '*', '/'.

Figure 5.4: Partial grammar of RML

```
FOR n IN E(x, "A") {
  PRINT n;
}
```

Note that the relational expression after IN must have one free variable, here x.

Alternatively, we could use a WHILE loop to print the satisfying assignments:

```
P(x) := E(x,"A");
WHILE EX(x, P(x)) {
  Head(x) := P(x) & FA(y, P(y) -> (x <= y));
  PRINT Head(x);
  P(x) := P(x) & !Head(x);
}
```

In this program, we first define a unary relation P of the satisfying assignments. The body of the WHILE loop is executed repeatedly as long as the condition of the loop holds, i.e., as long as P is not empty. In each iteration, we compute a relation Head containing the (lexicographically) smallest element of P. We then print this smallest element and remove it from P.

For a last example, suppose we want to define a relation Reachable(x, y) that holds iff "there is a path from x to y". This is done by the following statement:

```
Reachable(x, y) := TC(E(x, y));
```

In the above statement, TC denotes the transitive closure operator. The semantics of TC for a relation E with two free variables is described recursively as follows:

$$\mathsf{TC}(E(x,y)) \equiv E(x,y) \vee \exists z.E(x,z) \wedge \mathsf{TC}(E(z,y)) \ ^{1}$$

[1] More precisely, TC coincides with the $\mathbf{trcl}_{\vec{x},\vec{y}}$ operator in Definition 3.97 when $|\vec{x}| = |\vec{y}| = 1$.

5.4 Consistency Checking of Individual Models

As already stated, our approach for checking the consistency of a set of distributed models is to first merge the models into a single model and then check the consistency of this model against the constraints of interest. In this section, we develop a flexible platform for consistency checking of individual models. In Section 5.5, we show how this platform can be used for consistency checking of distributed models.

5.4.1 Translating Models to Relational Predicates

To evaluate relational expressions over a model, we translate it into a set of predicates. In Figure 5.5, we provide an algorithm, GraphToRml, for translating a model described as a typed graph into RML statements.

To illustrate the algorithm, consider the class diagram in Figure 5.6, expressed as a typed graph. We show in Figure 5.7 the translation of this class diagram into RML. To understand the translation, recall from Section 4.4.3 that if the meta-model of the modelling language being used is given by a graph $\mathcal{M}$, every model is a standard graph G equipped with a type function $t : G \to \mathcal{M}$, mapping each element of G to an element of $\mathcal{M}$. The type function respects the structure of G: If t maps an edge e of G to an edge u of $\mathcal{M}$, the endpoints of e are respectively mapped to those of u, as illustrated in Figure 5.6. Note that in the figure, $\mathcal{M}$ is the extends–implements fragment of the meta-model for Java class diagrams. A similar example was given in Figure 4.9 (Chapter 4).

5.4.2 Generic Consistency Checking Expressions

We provide a set of generic expressions for specifying structural consistency constraints over individual models. These expressions capture a number of recurring patterns that we have observed in the consistency constraints of class diagrams, e.g., (Unified Modeling Language, 2003; Egyed, 2000), goal models (Horkoff, 2006; Liaskos, 2007), and database

Algorithm. GraphToRml

Input: Graph $G = (N, E, \mathsf{source}_G, \mathsf{target}_G)$, and type function $t : G \to \mathcal{M}$.

Output: A set of RML statements.

for every node and edge i in G **:**
  if i is a node**:**
    output Node(i.uid)
  else : */* i is an edge */*
    output Edge(i.uid)
    output Source(i.uid, $\mathsf{source}_G(i)$.uid)
    output Target(i.uid, $\mathsf{target}_G(i)$.uid)
  / Translate type information */*
  output Type(i.uid, $t(i)$)
  / Translate properties */*
  for every property $(\text{Property}_k, \text{val}_k)$ of i **:**
    if val_k is boolean **:**
      if $\text{val}_k = \mathit{true}$ **:** output Property_k(i.uid)
    else : output Property_k(i.uid, val_k)

Figure 5.5: GraphToRml algorithm

schemata (Spaccapietra & Parent, 1994).

Table 5.1 shows some illustrative consistency constraints for these notations. Some of these constraints (e.g., **C1**, **C4**, and **C6**) capture fundamental well-formedness criteria, while others (e.g., **C2**, **C5**, and **G1**) describe desirable model qualities. The table provides an implementation of the constraints in RML with occurrences of our generic expressions bolded and underlined. We discuss these expressions under the following three headings:

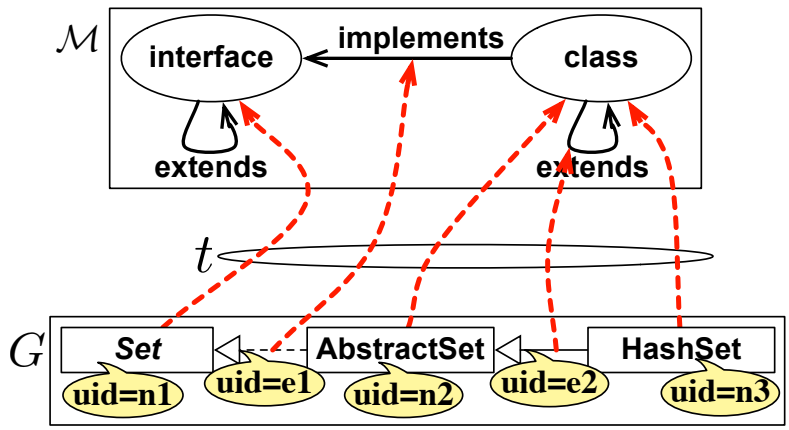


Figure 5.6: Example class diagram described as a typed graph

```
Node("n1");                    Node("n3");                    Type("e1", "implements");
Type("n1", "interface");       Type("n3", "class");           Edge("e2");
Label("n1", "Set");            Label("n3", "HashSet");        Source("e2", "n3");
Node("n2");                    Edge("e1");                    Target("e2", "n2");
Type("n2", "class");           Source("e1", "n2");            Type("e2", "extends");
Label("n2", "AbstractSet");    Target("e1", "n1");
```

Figure 5.7: RML encoding of the model in Figure 5.6

- ***Compatibility expressions***, used for ensuring compatibility of the type of an edge with the types of its endpoints.

- ***Multiplicity expressions***, used for defining a minimum and a maximum number for edges of a given type incident to a node.

- ***Reachability expressions***, used for checking existence of paths of edges of a given type between two nodes.

Lang.	Textual constraint	RML constraint
Class and Entity-Relationship Diagrams	**C1.** An implements edge relates a class to an interface (Unified Modeling Language, 2003)	**C1**() := FA(e, Type(e, "implements") –> **Compatible**$_{\text{class,interface}}$(e));
	C2. Every abstract class has a concrete implementation (Egyed, 2000)	**C2**() :=FA(c1, Type(c1, "class") & Abstract(c1) –> EX(c2, (Concrete(c2) & **Reachable**$_{\text{extends}}$(c2, c1))));
	C3. A class does not extend more than one class (Egyed, 2000)	**C3**() := FA(c, Type(c, "class") –> **SourceMultiplicity**$^{01}_{\text{extends}}$(c));
	C4. Inheritance is acyclic (Unified Modeling Language, 2003)	**C4**():= FA(c, Type(c, "class") –> !**OnCycle**$_{\text{extends}}$(c));
	C5. All classes are reachable from a root class (Egyed, 2000)	**C5**() := FA(c1, Type(c1, "class") –> EX(c2, Type(c2, "class") & IsRoot(c2) & **Reachable**$_{\text{extends}}$(c1, c2)));
	C6. Final classes do not have subclasses (Egyed, 2000)	**C6**() := FA(c, Type(c, "class") & Final(c) –> **TargetMultiplicity**$^{0}_{\text{extends}}$(c));
	C7. Inheritance is redundancy-free (Egyed, 2000)	**C7**() := FA(c1, Type(c1, "class") & FA(c2, Type(c2, "class") –> !**RedundantPaths**$_{\text{extends}}$(c1, c2)));
	C8. Every entity has a unique key (Spaccapietra & Parent, 1994)	**C8**() := FA(e, Type(e, "entity") –> **SourceMultiplicity**$^{1}_{\text{key\_link}}$(e));
i Goal Models*	**G1.** Goal dependencies are acyclic (Horkoff, 2006)	**G1**() := FA(g, Type(g, "goal") –> !**OnCycle**$_{\text{depends}}$(g));
	G2. A resource does not have multiple dependers (Horkoff, 2006)	**G2**() := FA(r, Type(r, "resource") –> **SourceMultiplicity**$^{01}_{\text{depends}}$(r));
	G3. Each loop in a goal decomposition tree includes at least one OR-decomposition (Liaskos, 2007)	**G3**() := FA(g1, Type(g1, "goal") –> !**OnCycle**$_{\text{decomposes}}$(g1) | EX(g2, Type(g2, "goal") & ORNode(g2) & **ReachVia**$_{\text{decomposes}}$(g1, g2, g1)));
	G4. Parallel contribution links do not exist (Liaskos, 2007)	**G4**() := FA(g1, Type(g1, "goal") & FA(g2, Type(g2, "goal") –> !**ParallelEdges**$_{\text{contributes}}$(g1, g2)));
	G5. Goal fulfillment cannot precede subgoal fulfillment (Liaskos, 2007)	**G5**() := FA(g1, Type(g1, "goal") & FA(g2, Type(g2, "goal") & **Reachable**$_{\text{decomposes}}$(g1, g2) –> !E$_{\text{precedes}}$(g2, g1)));

Table 5.1: Examples of well-formedness and quality constraints

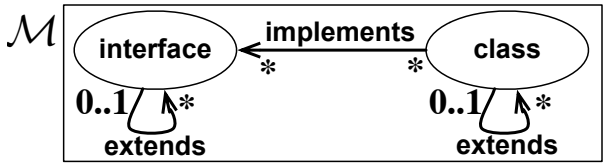


Figure 5.8: Example of multiplicity annotations

Compatibility Expressions

Compatibility constraints constitute the most primitive class of well-formedness criteria for conceptual models. Typed graphs automatically enforce compatibility constraints through their structure-preserving type function t (e.g., see Figure 5.6). We can also easily express these constraints in logical terms: To verify that the source and the target of an edge are respectively of types β and γ, we define $\mathsf{Compatible}_{\beta,\gamma}(\mathsf{x})$ that holds for all edges x satisfying the compatibility constraint:

$$\mathsf{Compatible}_{\beta,\gamma}(\mathsf{x}) := \mathsf{EX}(\mathsf{n},\ \mathsf{EX}(\mathsf{m},\ \mathsf{Source}(\mathsf{x},\ \mathsf{n})\ \&\ \mathsf{Target}(\mathsf{x},\ \mathsf{m})\ \text{->}\ \mathsf{Type}(\mathsf{n},\ \beta)\ \&\ \mathsf{Type}(\mathsf{m},\ \gamma)));$$

There are numerous instances of compatibility constraints in the notations we studied. Since these constraints are very similar, we show only one, namely, **C1**, in Table 5.1.

Multiplicity Expressions

Multiplicity constraints are often specified using annotations over the meta-model graph. Figure 5.8 shows the meta-model of Figure 5.6 annotated with multiplicity constraints. For example, consider the extends self-loop incident to the class node. According to the annotations of this edge, a class can extend at most one (i.e., 0..1) class, but each class can be extended by several (i.e., $*$) classes.

Given a multiplicity-annotated meta-model $\mathcal{M}$, we produce a set of logical expressions for validating conformance of an instance model to the multiplicity constraints prescribed by $\mathcal{M}$. Each edge in $\mathcal{M}$ gives rise to two multiplicity expressions – one for each endpoint.

We consider multiplicity annotations of four kinds: "k", "$k..l$", "$k..*$", and "$*$", where k and l are constants. An RML program for checking a multiplicity annotation "$k..l$" attached to the source side of a meta-model edge α is as follows:

```
SourceMultiplicity^{kl}_α(x):= FALSE(x);
FOR n IN Node(v) { /* v is a dummy free variable */
        IF ( k ≤ #(Source(e, n) & Type(e, α)) ≤ l) {
              SourceMultiplicity^{kl}_α(x):= (x=n) | SourceMultiplicity^{kl}_α(x);
        }
}
```

The program initializes $\mathsf{SourceMultiplicity}^{kl}_{\alpha}(\mathsf{x})$ to an empty unary relation, FALSE(x). If a node n respects the multiplicity constraint, it gets added to the relation $\mathsf{SourceMultiplicity}^{kl}_{\alpha}(\mathsf{x})$. The multiplicity expression $\mathsf{SourceMultiplicity}^{\spadesuit}_{\alpha}(\mathsf{x})$, where $\spadesuit$ is "k" or "$k..*$", can be implemented similarly[2]. Implementation of a multiplicity constraint attached to the target side of a meta-model edge is done by replacing Source(e, n) with Target(e,n) in the above program. In Table 5.1, constraints **C3**, **C6**, **C8**, and **G2** use multiplicity expressions.

Another useful expression similar to multiplicity expressions is for detecting parallel edges of a given type. This is implemented by $\mathsf{ParallelEdges}_{\alpha}(\mathsf{x}, \mathsf{y})$ shown below:

```
ParallelEdges_α(x, y) := FALSE(x, y);
FOR n IN Node(v) { /* v is a dummy free variable */
     FOR m IN Node(w) { /* w is a dummy free variable */
        IF (#(Source(e, n) & Target(e, m), & Type(e, α)) > 1) {
              ParallelEdges_α(x, y):= ((x=n) & (y=m)) | ParallelEdges_α(x, y);
        }
     }
}
```

[2]No expression is needed for the "$*$" multiplicity annotation.

This program initializes ParallelEdges$_\alpha$(x, y) to an empty binary relation, FALSE(x, y), and then, adds to it all pairs (n, m) of nodes between which there are parallel edges. An example constraint using ParallelEdges$_\alpha$(x, y) is **G4** in Table 5.1.

Finally, we note that our multiplicity expressions all work by testing whether "there exists at least, at most, or exactly c elements satisfying some property φ", where c is a *constant* number. These tests are already expressible in first order logic. The formula for the "at least" case is: $\exists x_1 \exists x_2 \ldots \exists x_k \ \bigwedge_{i \neq j}(x_i \neq x_j) \wedge \bigwedge \varphi(x_i)$. The other two cases are similar. Hence, our use of RML's counting operator is merely a matter of convenience. Generally speaking, RML's counting operator does provide additional expressive power. For example, given relations A and B, there is no first order logic formula for testing if the cardinality of A is larger than that of B. This query is easily expressible in RML.

Reachability Expressions

Several consistency constraints involve finding nodes that are reachable or unreachable via edges of a certain type. For example, in goal modelling, we may want to ensure that all goals are reachable via goal decomposition edges. In UML class diagrams, we may want to check that all descendants (via inheritance edges) of a given class have a certain property. For an edge of type α, we define a relation Reachable$_\alpha$(x, y) that holds iff a path from x to y made up of α-edges exists:

```
E_α(x, y) := EX(e, Source(e, x) & Target(e, y) & Type(e, α));
Reachable_α(x, y):= TC(E_α(x, y));
```

For example, in the class diagram of Figure 5.11, Reachable$_{\text{extends}}$("E","A") holds, indicating that E reaches A via extends edges.

A special case of reachability analysis is cycle detection. Cycles of edges of certain types can be indicative of a modelling problem. In class diagrams, for example, inheritance can never be cyclic. For an (edge) type α, the following RML statement creates a

relation $\mathsf{OnCycle}_\alpha(\mathsf{x})$ that holds for all nodes x residing on a cycle of α-edges:

$$\mathsf{OnCycle}_\alpha(\mathsf{x}) := \mathsf{Reachable}_\alpha(\mathsf{x}, \mathsf{x});$$

Reachability is also used for detecting path redundancies. For example, if we have three classes A, B, C, such that C extends B and B extends A, it would be redundant to have an extends edge from C to A because this is already implied by the path $\mathsf{C} \rightarrow \mathsf{B} \rightarrow \mathsf{A}$. Existence of multiple paths of edges of the same type between two nodes can be captured by $\mathsf{RedundantPaths}_\alpha(\mathsf{x}, \mathsf{y})$ defined as follows:

$$\begin{aligned}
&\mathsf{ReachVia}_\alpha(\mathsf{x}, \mathsf{z}, \mathsf{y}) := (\mathsf{Reachable}_\alpha(\mathsf{x}, \mathsf{z}) \mid (\mathsf{z} = \mathsf{x})) \;\&\\
&\qquad\qquad\qquad\qquad \mathsf{Reachable}_\alpha(\mathsf{z}, \mathsf{y})\ ;\\
&\mathsf{DistinctPathEnds}_\alpha(\mathsf{x}, \mathsf{y}) := \mathsf{EX}(\mathsf{v}, \mathsf{EX}(\mathsf{z}, \mathsf{ReachVia}_\alpha(\mathsf{x}, \mathsf{v}, \mathsf{y}) \;\&\\
&\qquad\qquad\qquad\qquad \mathsf{ReachVia}_\alpha(\mathsf{x}, \mathsf{z}, \mathsf{y}) \;\&\; \mathsf{E}_\alpha(\mathsf{z}, \mathsf{y}) \;\&\\
&\qquad\qquad\qquad\qquad \mathsf{E}_\alpha(\mathsf{v}, \mathsf{y}) \;\&\; !(\mathsf{z} = \mathsf{v})));\\
&\mathsf{RedundantPaths}_\alpha(\mathsf{x}, \mathsf{y}) := \mathsf{EX}(\mathsf{z}, (\mathsf{ReachVia}_\alpha(\mathsf{x}, \mathsf{z}, \mathsf{y}) \mid (\mathsf{y} = \mathsf{z})) \;\&\\
&\qquad\qquad\qquad\qquad \mathsf{DistinctPathEnds}_\alpha(\mathsf{x}, \mathsf{z})) \mid \mathsf{ParallelEdges}_\alpha(\mathsf{x}, \mathsf{y});
\end{aligned}$$

$\mathsf{ReachVia}_\alpha(\mathsf{x}, \mathsf{z}, \mathsf{y})$ holds iff there is a path (of length ≥ 1) from x to y passing through z. $\mathsf{DistinctPathEnds}_\alpha(\mathsf{x}, \mathsf{y})$ holds iff there are paths from x to y whose final edges (to y) are different. And, $\mathsf{RedundantPaths}_\alpha(\mathsf{x}, \mathsf{y})$ holds iff there are distinct paths or parallel edges from x to y.

As evidenced by Table 5.1, expressions involving variants of reachability are very common. In particular, **C2**, **C5**, and **G5** require checking reachability in its general form; **C4**, **G1**, and **G3** require checking cyclicity; and **C7** requires checking redundancy.

5.4.3 Instrumentation of Consistency Constraints

To obtain useful diagnostics from the relational interpreter, we instrument each consistency constraint with appropriate messages. The details of instrumentation depend on the nature of the consistency constraint in question and the kinds of feedback users are interested in. We provide two examples below:

Example I: Multiple Inheritance. To facilitate exploration of multiple inheritance violations, it is very helpful to include in the feedback the set of parents (i.e., superclasses) of every offending class. To do so, we instrument $\mathsf{SourceMultiplicity}^{01}_{\mathsf{extends}}(\mathsf{x})$, defined in Section 5.4.2, as follows:

```
1:    FOR n IN !SourceMultiplicity^01_extends(x) {
2:        PRINT n, " violates single inheritance", ENDL;
3:        Parent(y) := EX(e, Source(e,n) & Target(e,y) & Type(e, "extends"));
4:        PRINT ["      Parent: "] Parent(y);
5:    }
```

Assuming that we already computed the relation $\mathsf{SourceMultiplicity}^{01}_{\mathsf{extends}}(\mathsf{x})$, the above code does the following: For every class `n` with multiple parents, it outputs `n` (line 2). It then computes the set of `n`'s parents (line 3), and outputs the set (line 4).

If we execute the above code over the model in Figure 5.11, the resulting feedback is going to be as follows:

```
B violates single inheritance
     Parent:  A
     Parent:  C
```

This transcript, in addition to identifying `B` as an inconsistent element, provides *context* information about the inconsistency. Hence, we can immediately know that it is the relationship of `B` with `A` and `C` that causes `B` to fail the check.

Example II: Cyclic Inheritance. When exploring cyclic inheritance violations, we are interested not only in the individual offending classes, but also in the cyclic paths over which these classes appear. To report the cyclic paths in the feedback, we instrument $\mathsf{OnCycle}_{\mathsf{extends}}(\mathsf{x})$, defined in Section 5.4.2, as follows:

```
1:    FOR n IN OnCycle_extends(x) {
2:        PRINT n, " is on a cycle (";
3:        R(x,y) := EX(e, Source(e, x) & Target(e, y) & Type(e, "extends"));
4:        Current(x) := (x = n);
5:        WHILE (!(Current(z) & Inherits(z, n))) {
6:            Passes(x, y, u) := TC(R(x, y)) & TC(R(y, u));
7:            Admissible(x) := EX(z, Current(z) & R(z, x) & Passes (z, x, n));
8:            Previous(x) := Current(x);
9:            Current(x) := Admissible(x) & FA(y, Admissible(y) -> (x <= y));
10:           PRINT Current(x), "→";
11:           R(x,y) := !Previous(x) & !Current(y) & R(x,y);
12:       }
13:       PRINT "self)", ENDL;
14:   }
```

In the above code, we iterate over the classes in $\mathsf{OnCycle}_{\mathsf{extends}}$ and print a cycle for each. Specifically, for every $\mathsf{n} \in \mathsf{OnCycle}_{\mathsf{extends}}$, we compute a set Admissible of n's successors that have a path back to n (line 7). We choose an arbitrary element from Admissible (line 9) – in our code, the element with the smallest (lexicographic) value. After printing this successor (line 10), the process continues recursively, having removed from the inheritance relation the edge from n to the printed successor (line 11). The process ends when a full cycle has been printed.

Executing the above code over the model in Figure 5.11 results in the following output:

```
B is on a cycle (C → E → self)
C is on a cycle (E → B → self)
E is on a cycle (B → C → self)
```

Note that in the instrumentation code for cyclic inheritance, we ignored symmetries between inconsistencies. Therefore, the diagnostics include distinct errors for the same cycle in Figure 5.11.

5.5 Consistency Checking of Distributed Models

In this section, we generalize the platform developed in Section 5.4 to distributed models. A common approach for extending consistency checks to distributed models is to write consistency rules for the mappings between models, e.g., (Easterbrook & Nuseibeh, 1996; Nentwich *et al.*, 2003). For example, if we have a mapping R that equates elements of two models M_1 and M_2, we may wish to check that the mapping does not introduce cycles. This can be achieved by checking that each model individually is acyclic, and writing a new rule to check the mapping:

$$\mathsf{MCycle}_\alpha(\mathsf{x},\mathsf{y}) := \mathsf{R}(\mathsf{x},\ \mathsf{y})\ \&\ \mathsf{EX}(\mathsf{z},\ \mathsf{EX}(\mathsf{t},\ \mathsf{R}(\mathsf{z},\ \mathsf{t})\ \&\ \mathsf{Reachable}_\alpha(\mathsf{x},\ \mathsf{z})\ \&\ \mathsf{Reachable}_\alpha(\mathsf{t},\ \mathsf{y})));$$

If we apply this rule to M_1 and M_2, the relation MCycle_α holds for all pairs (x,y) of mapped elements that give rise to a cycle across the two models[3].

This pairwise approach is cumbersome for several reasons. First, it requires many new consistency rules: each existing consistency constraint (for a single model) may need to

[3]Formally, MCycle_α is applied to the *disjoint union* of M_1 and M_2.

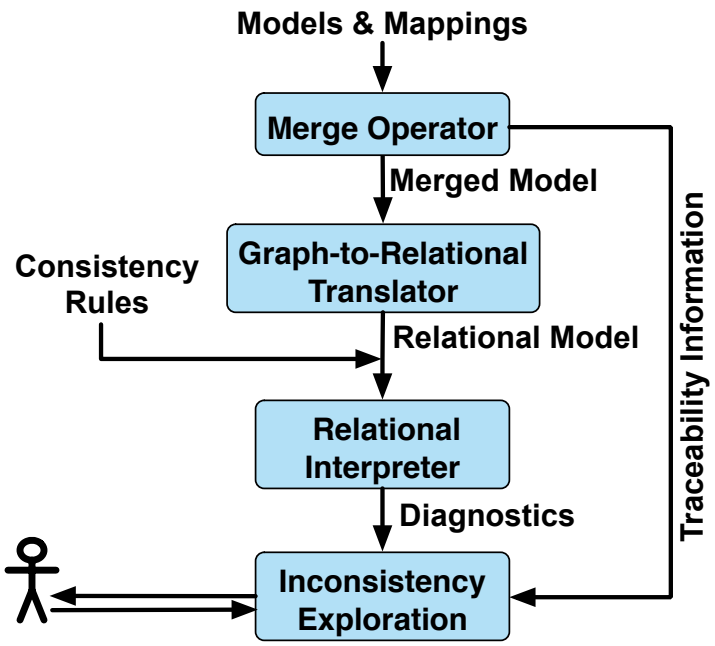


Figure 5.9: Overview of our consistency checking approach

be re-written to take into account each type of mapping that can hold between models. It also introduces an undesirable coupling between consistency rules and model mappings. Consistency rules refer to the possible mappings between models, and model mappings must be checked for their impact on the consistency rules.

Second, this approach does not easily generalize beyond pairwise checking. This is because global consistency rules must consider the interactions between different mappings in the system. For example, a global rule to check for cyclic inheritance in a system such as that of Figure 5.2 would need to refer to all of the mappings between the models in the scope of a single rule. This makes the specification of global consistency rules very complex.

We present an alternative approach that does not suffer from these problems. An overview of our approach is shown in Figure 5.9. Given a set of models and mappings, we begin by constructing a merged model as described in Chapter 4. This model is translated into a set of relational predicates using the algorithm in Figure 5.5. The result, along with the consistency rules of interest, is sent to a relational interpreter for

consistency checking and producing diagnostics for any inconsistencies found. Users can then explore these diagnostics and project them back to the source models and mappings by utilizing the traceability data produced during the merge operation.

To illustrate our approach, consider the example system of models in Figure 5.10. Figure 5.11 shows the merge computed for the system using the merge operator developed in Chapter 4. The traceability information for the classes in the merge is shown in Figure 5.12. Similar traceability information is stored for the inheritance links (not shown). Evaluating the merge against the instrumented constraints of Section 5.4.3 yields two inconsistencies: (1) B has two superclasses; (2) B, C, E form a cycle. We already saw the generated diagnostics in Section 5.4.3. Navigation from these diagnostics to the source models and mappings is made possible by *hyperlinking* the diagnostics to the traceability data associated with the elements of the merge.

For example, Figure 5.13 depicts a hyperlinked version of the multiple inheritance diagnostics in Section 5.4.3. Navigating a link in the diagnostics retrieves the traceability data (projections) for the element in question. Figure 5.13 shows the projections for class A of the merge. Like the diagnostics, projections are in a hyperlinked format, allowing one to navigate to the source models and mappings involved in the inconsistencies.

Note that the projections include *all* available information about the origins of the selected element. This information may not be minimal, i.e., not all models and mappings appearing in the projections are necessarily responsible for the occurrence of the inconsistency in question. For example, model M_4 and mappings R_3, R_4 do not play a role in the violation of single inheritance – the violation would occur even if we removed M_4, R_3, and R_4 from the system in Figure 5.10. However, as shown in Figure 5.13, M_4 and R_3 appear in the projections for A.

For a simple system like the one in Figure 5.10, it may be reasonable to repeat the merge with different subsets of $M_1, \ldots, M_4$ and $R_1, \ldots, R_4$, and identify the minimal subsystem that can produce a particular violation. This is, however, exponential in

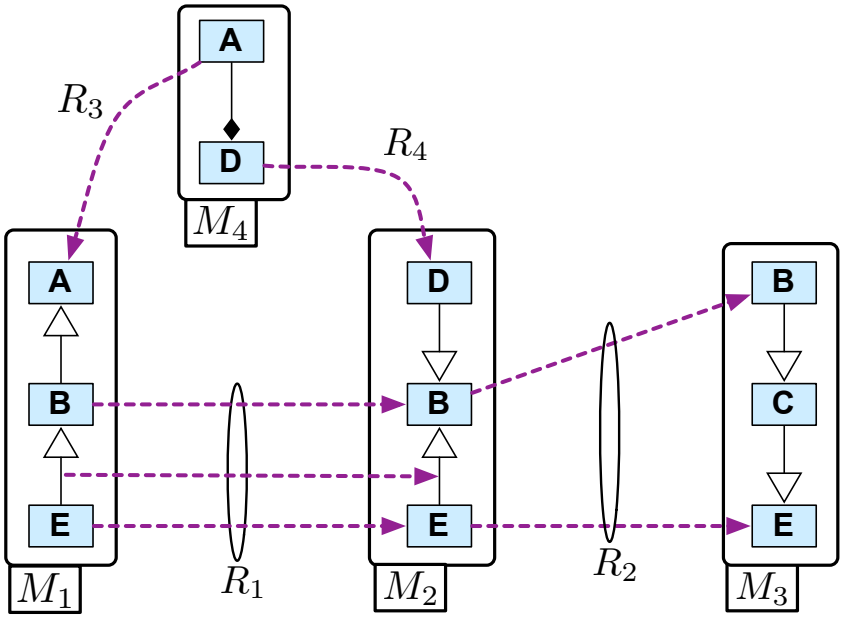


Figure 5.10: Example system with multiple models and mappings

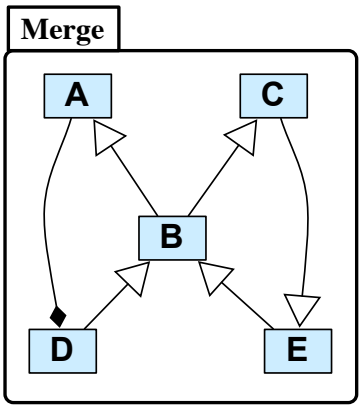


Figure 5.11: Merge of the system in Figure 5.10

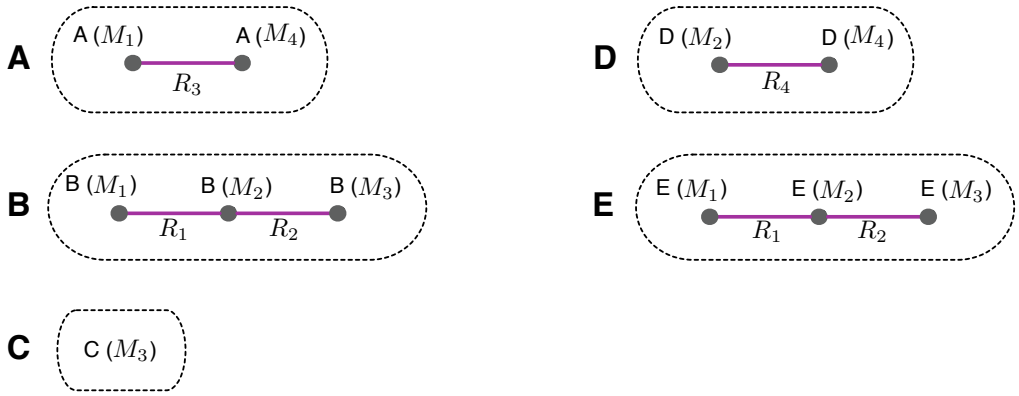


Figure 5.12: Traceability information for the classes in the merge of Figure 5.11

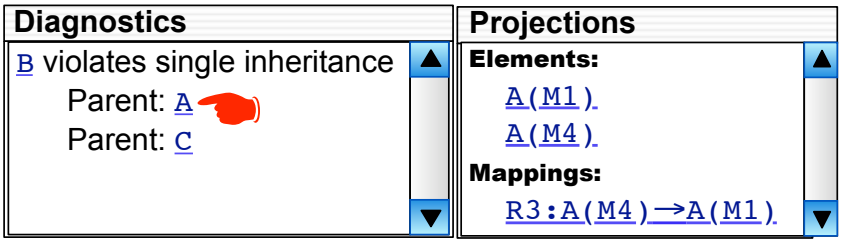


Figure 5.13: Hyperlinking diagnostics to traceability data

the number of models and mappings in the system, and hence not scalable to large systems. Scalable solutions may be found for certain constraints and certain patterns of interconnecting the models, but we have not explored this in our work yet.

On the other hand, filtering out information that is seemingly irrelevant to the occurrence of an inconsistency may not be always desirable. For example, although model M_4 and mapping R_3 are not responsible for the violation of single inheritance, they may still be involved in a resolution of the problem. For example, a possible resolution is to create a new mapping between M_4 and M_3 and unify `A` and `C`. Deducing this resolution needs the knowledge that R_3 unifies `A` in M_4 and `A` in M_1, and filtering out M_4 and R_3 from the diagnostics would effectively eliminate this alternative.

5.6 Preservation of Logical Properties

An interesting question that arises in global consistency checking is whether any of the logical properties of a set of models carry over to their merge. More specifically, given a set of models and a property φ, can we state anything about the satisfaction of φ over the merge of the models, provided that some (or all) of the original models satisfy φ?

We already know that some interesting properties do not carry over – in fact, this is the main reason why we need global consistency checking in the first place. For example, all the models in the systems of Figures 5.2 and 5.10 are acyclic, whereas the merges computed for both systems turn out to be cyclic. For another example, all the models in

5.10 satisfy the single inheritance property, but this property is violated by the merge.

Despite the above, some properties of the source models are indeed preserved across merge. For example, we know from Section 4.4.3 that merges are always well-typed, meaning that a merge satisfies all the compatibility properties (i.e., properties similar to **C1** in Table 5.1) that the source models satisfy. Since type preservation is explicitly guaranteed by the algebraic definition of our merge operator, there is no need for a logic-based machinery to reason about it. However, the algebraic definition of merge, by itself, does not provide mechanisms for reasoning about more advanced logical properties.

In this section, we report on some preliminary results that allow us to establish a connection between the algebraic and logical characteristics of our framework. We use these results to reason about preservation of consistency constraints.

Like in the earlier parts of this chapter, we assume classical semantics for logical properties. In particular, this means that property satisfaction is independent of the belief annotations introduced in Chapter 4. Our preservation results can be used for annotated models if the annotations are treated as recipes for either keeping or filtering elements from the source models. For example, if the annotations are drawn from the lattice in Figure 4.12, we may want to filter out elements marked refuted (✘). Our arguments are valid as long as the source models remain graphs after filtering.

Alternatively, one can use non-classical semantics for evaluating properties in the presence of the annotations (Gurfinkel, 2007). For example, in (Nejati *et al.*, 2007), we use a form of three-valued semantics to reason about preservation of behavioural properties when multiple variants in a product family are merged. Generalizing these existing non-classical results from temporal logics to the more expressive logics used in this thesis has not been attempted here and is left for future work. The key challenge in this regard is to provide a logical characterization of how belief annotations evolve across mappings. This evolution cannot be expressed using classical homomorphism mappings.

5.6.1 General Results

Recall from Chapter 4 that we characterize the merge operation using graph colimits. As we explained in Section 4.3, a colimit-based merge offers three key features:

F1 Merge yields a family of mappings, in our case graph homomorphisms, one from each source model onto the the merged model.

F2 The merged model does not contain any unmapped elements, i.e., every element in the merged model is the image of some element in the source models.

F3 Merge respects the mappings in the source system, i.e., the image of each element in the merged model remains the same, no matter which path through the mappings in the source system one follows.

From ***F1*** and Lemma 3.103 (in Chapter 3), it follows that graph colimits preserve the existential positive fragment of transitive closure logic, and more generally, that of least fixpoint logic (LFP).

Theorem 5.1 *If an existential positive LFP formula φ is satisfied by* ***some*** *source model M, any merge in which M participates will satisfy φ as well.*

By ***F2*** and the above theorem, we obtain the following result regarding preservation of universal properties.

Theorem 5.2 *Let $\varphi(x)$ be an existential positive LFP formula with a free variable x. If the formula $\psi = \forall x\, \varphi(x)$ is satisfied by* ***all*** *the source models, ψ will be satisfied by any merge of the models as well.*

Notice that Theorem 5.2 allows the introduction of *only one* universal quantifier. To gain intuition on what happens when additional universal quantifiers are introduced, consider the system in Figure 5.14(a) and let the relation $E(x, y)$ denote the graph edge relation. Both models in Figure 5.14(a) are complete graphs and hence satisfy the

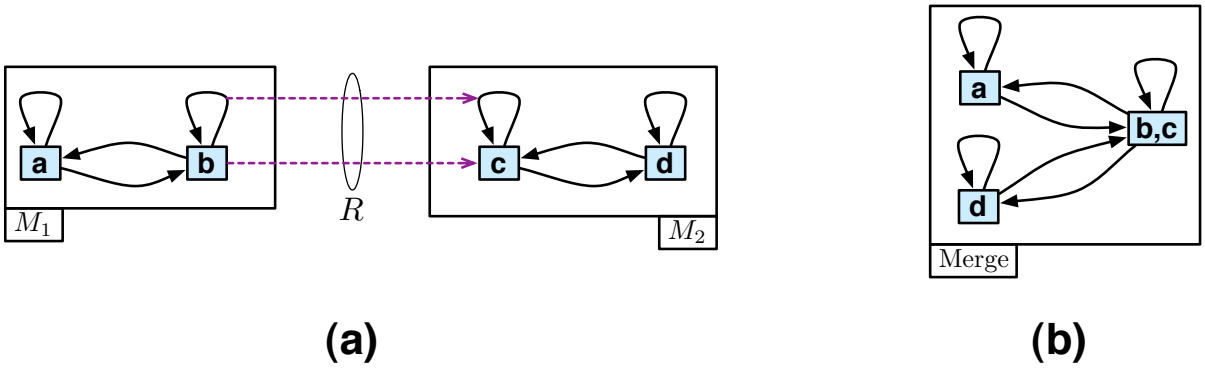


Figure 5.14: Illustration for violation of universal properties

property $\forall x \forall y\ \mathsf{Node}(x) \wedge \mathsf{Node}(y) \Rightarrow E(x,y)$ [4]. However, the property is violated over the resulting merge shown in Figure 5.14(b), because there is no edge from node a to node d and vice versa. The general observation here is that, when there is more than one universal quantifier, universally quantified variables can be assigned values from non-shared parts of *different* source models. In such a case, property satisfaction over the individual source models may not extend to the merge.

Currently, we do not know whether ***F3*** leads to further property preservation results. Also, it may be possible to trade off development flexibility in favour of a broader class of preserved properties, e.g., by using more constrained mappings for relating models, or by placing restrictions on the patterns used for interconnecting the models. We leave an elaboration of these topics to future investigations.

5.6.2 Preservation of Consistency

Below, we employ the results of Section 5.6.1 to reason about preservation of consistency.

Compatibility Properties. Although preservation of compatibility properties is already established through algebraic means, it is interesting to see if the same can be

[4]The property uses implication and hence has negation. But the negation can be resolved, because every element in the universe that is not a node is an edge. Therefore, the property is equivalent to $\forall x \forall y\ \mathsf{Edge}(x) \vee \mathsf{Edge}(y) \vee E(x,y)$.

done through logical means. For example, consider **C1** in Table 5.1, and remember from Section 5.4.2 that $\mathbf{Compatible}_{\mathsf{class,interface}}(\mathsf{e})$ can be written as an existential positive property. The sub-formula $\mathsf{Type(e, "implements")}$ of **C1** appears in negated form, but the negation can be resolved, knowing that (1) the set of types is fixed and, (2) every element has a type. More precisely, if the set of types is $\{t_1, \ldots, t_n\}$, the formula $\neg\mathsf{Type}(e, t_\ell)$ can be replaced with $\bigvee_{i \neq \ell} \mathsf{Type}(e, t_i)$. Hence, by Theorem 5.2, **C1** is preserved.

Multiplicity Properties. One can show through simple counter-examples that *none* of the following lift from the source models to the merge:

- There exists <u>at least</u> c elements satisfying φ.
- There exists <u>exactly</u> c elements satisfying φ.
- There exists <u>at most</u> c elements satisfying φ.

It is easy to see why the "exactly" and "at most" cases do not get preserved, noting that the merge normally has more information than any of the source models. To understand why the "at least" case is not preserved, note that homomorphisms (and functions as well) are not necessarily one-to-one, and can therefore *shrink* the number of elements satisfying a property. For example, consider the system of models in Figure 5.15(a) and its merge in Figure 5.15(b). For simplicity, the models are discrete graphs, i.e., sets, and their mappings are functions. Although M_1, M_2, M_3 all satisfy the property "there exists at least three (distinct) nodes", the merge has only two nodes, hence violating the property. It is important to mention that the flexibility to fuse together multiple elements of the same source model is not an undesirable feature and is indeed valuable when one needs to perform an abstraction during merge (Kalfoglou & Schorlemmer, 2005).

Reachability Properties. An interesting consequence of Lemma 3.103 is preservation of paths, i.e., the expression $\mathsf{Reachable}_\alpha(\mathsf{x}, \mathsf{y})$ defined in Section 5.4.2. To see how this

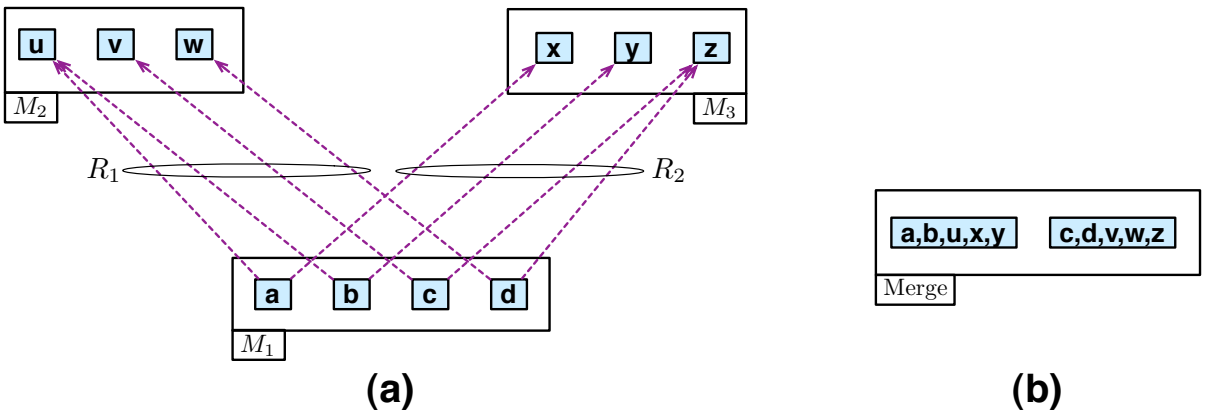


Figure 5.15: Illustration for violation of multiplicity properties

can be used for reasoning about consistency, consider **C2** in Table 5.1. Using the argument we gave when discussing preservation of compatibility properties, we know that the negation of Type(c1, "class") can be resolved. Further ¬Abstract(c1) can be replaced with a positive property, say Concrete(c1). It now follows from Theorem 5.2 that **C2** is preserved. Similarly, it is easy to check that **C5** in Table 5.2 is preserved as well.

5.7 Evaluation

In this section, we provide initial evidence for the feasibility and usefulness of our consistency checking framework.

5.7.1 Tool Support

We have implemented our approach into a prototype tool, TReMer+ (see Chapter 6). The tool allows users to verify a system of interrelated models against a given set of consistency rules using the process shown in Figure 5.9. Consistency rules are evaluated by the CrocoPat relational interpreter (Beyer *et al.*, 2005).

		# elements (nodes + edges)			
		500	**1,000**	**5,000**	**10,000**
Consistency Rule	**Dangling Edges**	< 1 sec	< 1 sec	5 sec	10 sec
	Parallel Edges	< 1 sec	1 sec	15 sec	57 sec
	Type Violations	< 1 sec	< 1 sec	4 sec	11 sec
	Multiple Inheritance	< 1 sec	< 1 sec	7 sec	24 sec
	Cyclic Inheritance	1 sec	3 sec	1 min 6 sec	4 min 48 sec

Table 5.2: Consistency checking running times

5.7.2 Computational Scalability

To validate computational scalability, we need to ensure that both model merging and (intra-model) consistency checking are scalable. The complexity of our merge algorithm is linear in the size of the input models and mappings (see Section 4.7.1). This is dominated by the complexity of consistency checking for most interesting consistency constraints.

To ensure that our approach scales in practice, we used CrocoPat for checking some representative consistency rules over UML domain models with 500 to 10,000 elements. These were structurally realistic models assembled from smaller real-world models. We introduced inconsistencies of various kinds into these models so that about 10% of the elements in each model appeared in the results of inconsistency analysis. Table 5.2 shows the running times for a number of checks on a Linux PC with a 2.8 GHz Pentium CPU and 1 GB of memory. The reported times include finding the inconsistencies and generating proper diagnostics for them. The results indicate that the method is scalable to handle realistically large modelling problems.

5.7.3 Case Study

We motivated our work in this chapter by two main improvements it brings to distributed development: (1) Eliminating the need to have separate rules for checking consistency of

models and consistency of mappings; and (2) Generalization from pairwise consistency checking to global consistency checking where the interactions between different mappings in the system are also considered.

Writing consistency rules is usually a laborious task; hence, the first improvement increases productivity by requiring the development of just a single rule for each consistency constraint. To evaluate the practical utility of the second improvement, we conducted an exploratory study using our tool TReMer+. The study was aimed at investigating how global consistency checking could facilitate the analysis of relationships between distributed models. We based our study on models developed by students as an assignment in a recent offering of a senior undergraduate course on object-oriented analysis and design. To ensure privacy, these models were anonymized by a third party prior to our study.

The assignment had the students write a UML domain model for a hospital based on a short and intentionally ambiguous textual description. This description is provided in Appendix A. We studied five models developed independently by five individual students. These models, shown in Figures A.1–A.5 (of Appendix A), are roughly equal in size, each with 60 to 70 elements; however, there are remarkable discrepancies in the way the models are structured. Other studies suggest that such discrepancies are very common when models are developed independently (Svetinovic *et al.*, 2005; Easterbrook *et al.*, 2005).

The main goals of our study were: (1) to construct a coherent set of mappings to express the overlaps between the studied models; and (2) to systematically explore how these models differed from one another. To achieve these, we began by hypothesizing a set of preliminary mappings between the models, shown in Figure 5.16.

In addition to the source models, **M1**–**M5**, the system in Figure 5.16 includes a model named **Helper1**. This model is used for bridging missing information between the source models. For example, to capture patients' visits to the hospital, all models except

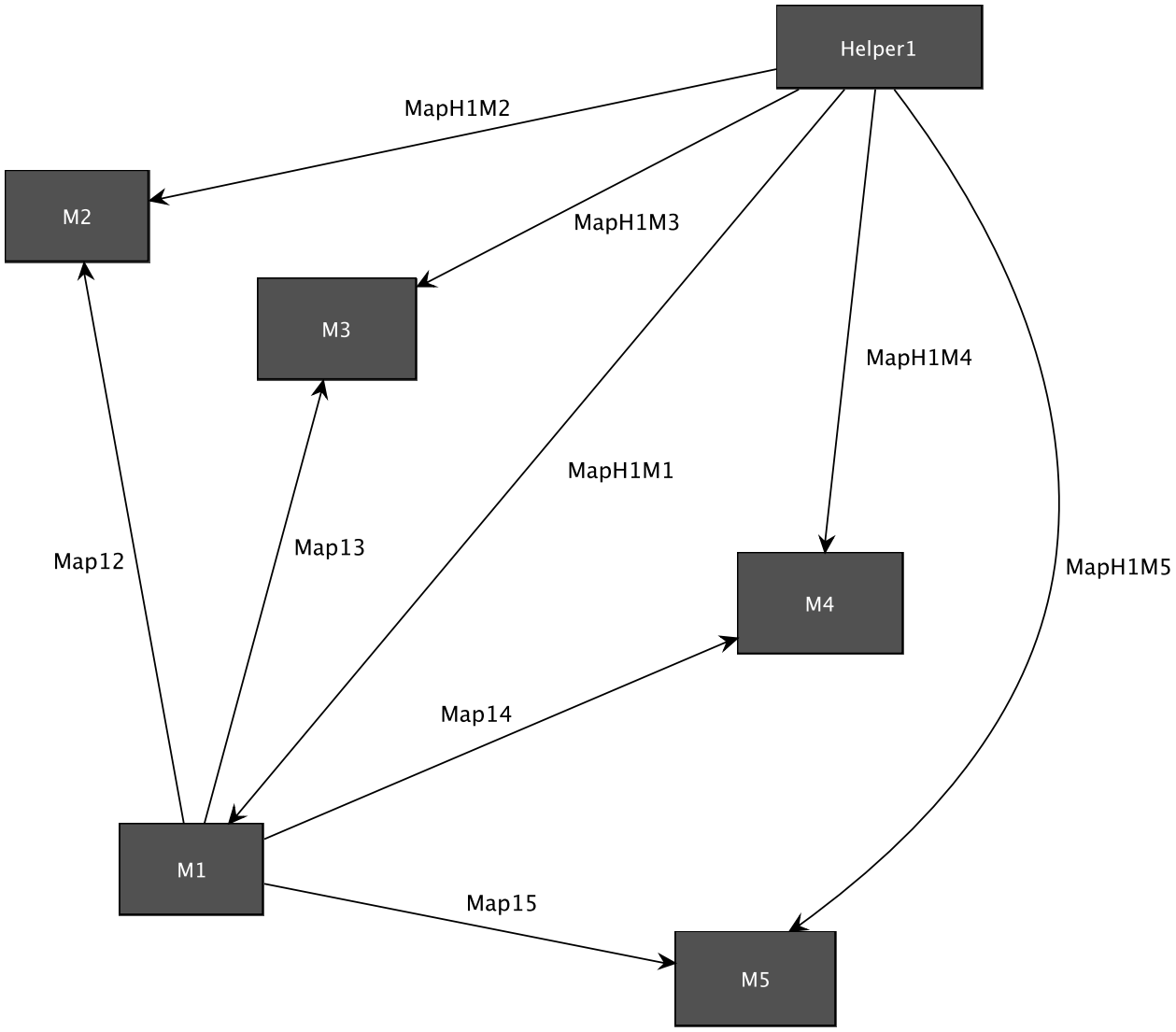


Figure 5.16: Hypothesizing the mappings between the source models

M1 envisage two domain concepts, InpatientVisitRecord and OutpatientVisitRecord, whereas model **M1** envisages only one concept, PatientVisitRecord. To describe the relationships between these concepts, we include in **Helper1** a model fragment like the one shown in Figure 5.17, stating that InpatientVisitRecord and OutpatientVisitRecord are subclasses of PatientVisitRecord. We then map PatientVisitRecord in **Helper1** to the corresponding concept in **M1** (through the mapping **MapH1M1**), and similarly map InpatientVisitRecord and OutpatientVisitRecord in **Helper1** to the corresponding concepts in **M2**–**M5**.

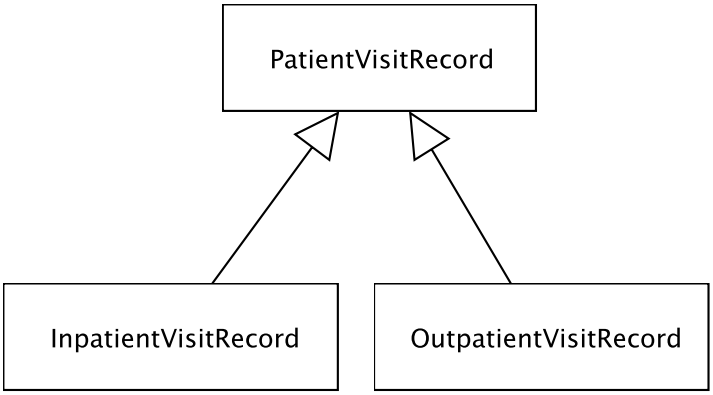


Figure 5.17: Describing missing information using helper models

Further, note that we did not have to state all pairwise mappings in the system of Figure 5.16, because the mappings are transitive. For example, equating Technician in **M1** and Technician in **M2** (through **Map12**), and equating Technician in **M1** and TechnicalStaff in **M3** (through **Map13**), automatically equates Technician in **M2** and TechnicalStaff in **M3**.

After specifying the preliminary mappings between the models, we employed global consistency checking as a way to discover anomalies in these mappings, and later to investigate the differences between the models with respect to the mappings between them. Below, we describe our findings and highlight the advantages of global consistency checking.

First, we automatically constructed a merge based on the preliminary mappings defined between the models. The merge is shown in Figure A.6 (of Appendix A). Consistency checking of this merge revealed several potential anomalies. In particular, the merge had 3 sets of identically-named concepts and 8 sets of parallel links (edges). All of these anomalies were due to the unstated overlaps between the models, which manifested themselves as duplicate elements in the merge. Note that, although not observed in our study, the anomalies could have had other causes. For example, some identically-named concepts could have been homonyms, and some parallel links could in fact have been necessary to distinguish between the different link roles.

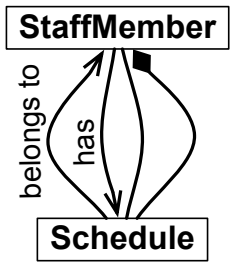


Figure 5.18: Alternative ways of building associations between concept pairs

The generated inconsistency diagnostics along with the traceability data stored for the merge allowed us to quickly identify the origins of duplicate elements and unify them by defining new correspondences. If we wanted to do this by pairwise checking of the five source models, we would have needed to check $(5 * 4)/2 = 10$ individual mappings between model pairs. Constructing a merge and checking global consistency made it possible to perform this task in a single shot. Further, as we mentioned earlier, for global consistency checking, we do not even have to state all mappings between model pairs, because merge automatically accounts for the transitive nature of the mappings.

After refining the preliminary mappings with the newly discovered correspondences, we concentrated on analyzing structural discrepancies between the source models. These discrepancies were primarily due to the use of competing alternatives for capturing the relationships between the concepts in the domain. For example, to relate the StaffMember and Schedule concepts, one could choose among several alternatives, e.g., (1) an unlabelled undirected association, (2) a labelled directed association either saying "StaffMember *has a* Schedule" or "Schedule *belongs to* StaffMember", and (3) a composition link expressing a containment relation between StaffMember and Schedule.

Model merging provided a convenient way to bring together and visualize the alternatives used in different models for relating concept pairs. As an example, Figure 5.18 shows the relevant fragment of the merge for the StaffMember–Schedule pair.

To detect and enumerate alternative choices for relating concept pairs, we developed a variant of the parallel edges rule (see Section 5.4.2), which ignored link types and

directionality. Checking the merged model against this rule yielded 19 groups of links. Each group captured the set of alternatives proposed in different models for relating two specific concepts. The groups referenced a total of 70 links in the source models. Being able to simultaneously view *all* proposed alternatives for relating two concepts is crucial for resolving conflicts and building consensus between the source models. Pairwise checking would have allowed us to deal with only two alternatives at a time.

Finally, we re-examined the source models to apply the knowledge gained from our analysis, by marking ill-conceived elements that needed to be filtered out. At this step, global consistency checking provided us with quick feedback on the impact of removing an element from one model on other models. For example, if we marked concepts Inpatient and Outpatient in **M1** for filtering, we could automatically check all other models which envisage these concepts, to verify that no (non-deleted) links were incident to these concepts. Such links would become anomalous as soon as the Inpatient and Outpatient concepts were actually deleted from **M1**. Employing pairwise checking for performing such sanity checks after a change can be costly or even ineffective when more complex sanity criteria are involved.

In summary, constructing merged models and checking global consistency allowed us to do various types of analysis that would be either expensive or impossible to do by pairwise checking. The traceability information generated during the merge operation made it possible to project inconsistencies back to the originating models and mappings, and take steps to resolve them. Since our merge process is fully automatic, we did not incur overhead costs for generalizing from pairwise to global consistency checking.

5.8 Related Work

Generic constraint expressions. Developing generic expressions for describing correctness properties of models is not a new idea. For example, (Dwyer *et al.*, 1999; Konrad

& Cheng, 2005) provide templates for capturing temporal properties of systems. However, these templates are specifically for behavioural models, and are inapplicable to non-behavioural ones such as class diagrams. The closest work to ours is that of (Wahler *et al.*, 2006), which describes a set of constraint patterns for UML models. However, this work lacks generality and only considers the family of UML notations. In contrast, our work applies to a wider class of notations including those for goal and entity-relationship models.

Consistency checking. As we saw in Chapter 2, consistency checking of distributed models is a well-studied topic in requirements engineering. In this domain, the term "inconsistency" usually refers to a situation where a *pair* of models do not obey a relationship that should hold between them (Nuseibeh *et al.*, 1994). This definition is restrictive in that it makes inconsistency a *pairwise* notion; however, it has the advantage of being easily applicable to *heterogeneous* models. Our work, in contrast, treats inconsistency as a *global* notion, but its scope is currently limited to *homogeneous* models only.

Early approaches to consistency checking of viewpoints use standard first order logic for writing consistency rules (Easterbrook & Nuseibeh, 1996). The expressiveness of these approaches is limited because first order logic cannot capture reachability. Recent work on consistency checking of viewpoints addresses these limitations by using more expressive logics. For example, xlinkit (Nentwich *et al.*, 2003) employs first order logic augmented with a transitive closure operator for describing consistency rules. Similarly, (Paige *et al.*, 2007) explores the use of theorem proving and object-oriented programming to provide a rich platform for constraint specification. These approaches indeed offer sufficient expressive power to cover a wide range of consistency constraints; however, they do not address the key problem tackled in our work, which is consistency checking of *arbitrary* systems of models and mappings.

The idea of consistency checking via merge was first explored in prior work by Easterbrook and Chechik (Easterbrook & Chechik, 2001). The work uses temporal logic model checking to reason about behavioural properties of state machine models when they are merged. This earlier work considers only binary merges, and further, is inapplicable to structural models, because temporal logic cannot capture important structural constraints such as multiplicities (Libkin, 2004). Other consistency checking approaches based on temporal logic suffer from similar expressive power limitations if applied to structural models.

Several consistency checking approaches work by consolidating different modellers' descriptions into a unified knowledge-base and checking its overall consistency. For example, (Gervasi & Zowghi, 2005) translates textual requirements into a knowledge-base of logical statements and finds inconsistencies by applying theorem proving and model checking. These approaches can reason about global consistency; however, to build a unified knowledge-base, they assume that modellers have already agreed on a unified vocabulary. We do not make this assumption in our work and use explicit mappings to capture the relationships between the vocabularies of different models. This makes it possible to hypothesize alternative relationships between these vocabularies and explore how each alternative affects global consistency properties.

There is a large body of research specifically dealing with consistency in UML. For UML models, consistency rules are usually described in the Object Constraint Language (OCL) (Object Constraint Language, 2003). Several tools exist for checking UML models against OCL expressions, e.g., the Dresden OCL toolkit (Hussmann *et al.*, 2002). Despite their merits, these tools are not suited to requirements modelling because of OCL's strong orientation towards the design and implementation stages (Vaziri & Jackson, 1999). Instead, we used a highly expressive and domain-independent language for specifying consistency rules. This makes our framework adaptable to a variety of modelling notations, including those used in requirements engineering.

5.9 Summary and Future Work

We presented an approach for consistency checking of distributed models. The approach enables detecting global inconsistencies that would not otherwise be identified if we only checked the consistency of individual models and individual mappings between them. Our approach can reduce any multi-model consistency checking problem to a single-model consistency checking problem via model merging. Hence, it requires developing only a single rule for each consistency constraint – the rule applies no matter how many models are involved and how they are related to one another.

To simplify the specification of consistency rules, we have developed a set of generic expressions for characterizing recurrent patterns in structural constraints of conceptual models. We demonstrated the usefulness of our expressions for specifying the constraints of class and entity-relationship diagrams, and goal models.

Our work has a number of shortcomings that we plan to address in the future. Particularly, the work currently applies to homogeneous models only. Extending the work to heterogeneous models poses a challenge because this would involve merging models represented in different notations. It is possible to merge a set of heterogeneous models by translating them into a single notation first, but such a translation discards the structural and visual properties of the models. As a result, merges might no longer be an ideal context for exploration of inconsistencies. One possible way to address this problem is to develop techniques that allow users to directly explore inconsistencies over the source models and mappings. These techniques can still utilize merge for automated analysis, but the feedback provided to users should be interpretable independently of the merge.

Our approach would also benefit from a model slicing operator, so that we can extract a desired aspect of a set of models based on a given criterion. In particular, when the source models are large, we may want to hide the complete merge from the user and, at any given time, only show the slice that is relevant to the inconsistency instance being explored. Alternatively, we may want to compute slices of the source models

before constructing a merge. For example, to find cyclic inheritance violations in class diagrams, we only need the class objects and their inheritance links; therefore, we can filter out all the attributes, methods, dependencies, associations, etc. from the source models before merging them.

Finally, we need to further study the usefulness of our approach by conducting user trials and observing how users employ global consistency checking for exploring systems of interrelated models.

Chapter 6

Tool Support

In this chapter, we describe a tool, TReMer+, that implements the model merging and consistency checking approaches discussed in Chapters 4 and 5, respectively.

About the tool. The initial version of our tool was named iVuBlender (Sabetzadeh & Easterbrook, 2005a). This early version only supported the merge approach in Chapter 4. In a later iteration, reported in (Sabetzadeh & Nejati, 2006; Sabetzadeh *et al.*, 2007b), iVuBlender was redesigned so that the merge process was no longer coupled with a particular merge algorithm. The tool was then extended with the behavioural merge operator in (Nejati *et al.*, 2007), and its name was changed to TReMer, which stands for Tool for Relationship-Driven Model Merging. In a third iteration, reported in (Sabetzadeh *et al.*, 2008), TReMer was extended with the consistency checking approach described in Chapter 5, and the tool's name was suffixed by a +, giving the tool its current name.

6.1 Tool Overview

In this section, we discuss TReMer+'s implementation and methodology of use.

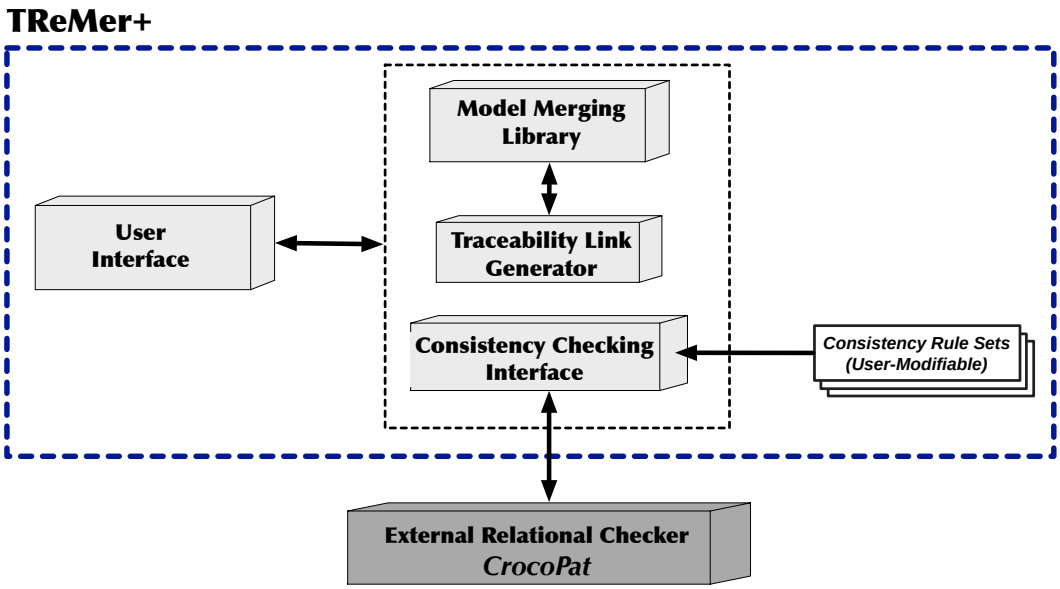


Figure 6.1: Architecture of TReMer+

6.1.1 Implementation

The overall architecture of TReMer+ is shown in Figure 6.1. The main building blocks of the tool are:

User Interface. TReMer+ provides a visual user interface for editting models, building relationships between models, and defining systems of interrelated models. The user interface further enables navigation of inconsistency diagnostics and traceability links. TReMer+ currently supports entity-relationship diagrams, state machines, and simple UML domain models. In the future, we plan to extend the tool to support other notations, such as goal models and detailed class diagrams.

Merge Library. TReMer+ defines a plugin interface for the merge operation and can work with any merge algorithm that realizes this interface. Currently, we provide implementations for two merge algorithms: one is the algorithm in Chapter 4, and the other – the behavioural merge algorithm in (Nejati *et al.*, 2007). In this chapter, we illustrate only the former algorithm. For a detailed treatment and illustrations of the latter, see (Nejati, 2008).

Traceability Link Generator. TReMer+ provides generic primitives for establishing traceability between merged models and their sources. The traceability links for a merged model are stored in an XML document. This document is rendered as hypertext by the user interface for easier navigation.

Consistency Checking Interface. As we explained in Chapter 5, TReMer+ does not implement a consistency checking engine of its own. Instead, it uses an external relational manipulation tool, CrocoPat (Beyer *et al.*, 2005), for verification of consistency properties. To interact with CrocoPat, TReMer+ implements an interface responsible for (1) translating graphical models into CrocoPat's predicate language; (2) invoking CrocoPat with a user-selected set of consistency rules; and (3) communicating the inconsistency diagnostics generated by CrocoPat to the user interface for presentation to the user.

Consistency Rule Sets. TReMer+ has rules for checking well-formedness of entity-relationship diagrams, UML domain models, and state machines. These rules are specified in an XML file which can be easily modified or extended by end-users. To simplify the specification of consistency rules, TReMer+ provides a set of generic expressions capturing common patterns in the structural constraints of graph-based models. We discussed these expressions in Chapter 5.

TReMer+ is written in Java. It is roughly 15K lines of code, of which 8.5K implement the user interface, 5.5K implement the tool's core functions (model merging, traceability, and serialization), and 1K implement the interface for interacting with CrocoPat. The tool uses JGraph (`http://www.jgraph.com/`) for editting and visualizing models, and EPS Graphics2D (`http://www.jibble.org/epsgraphics/`) for exporting models to PostScript vector graphics. TReMer+ was publicly released in May 2007. The tool is freely available at `http://www.cs.toronto.edu/~mehrdad/tremer/`

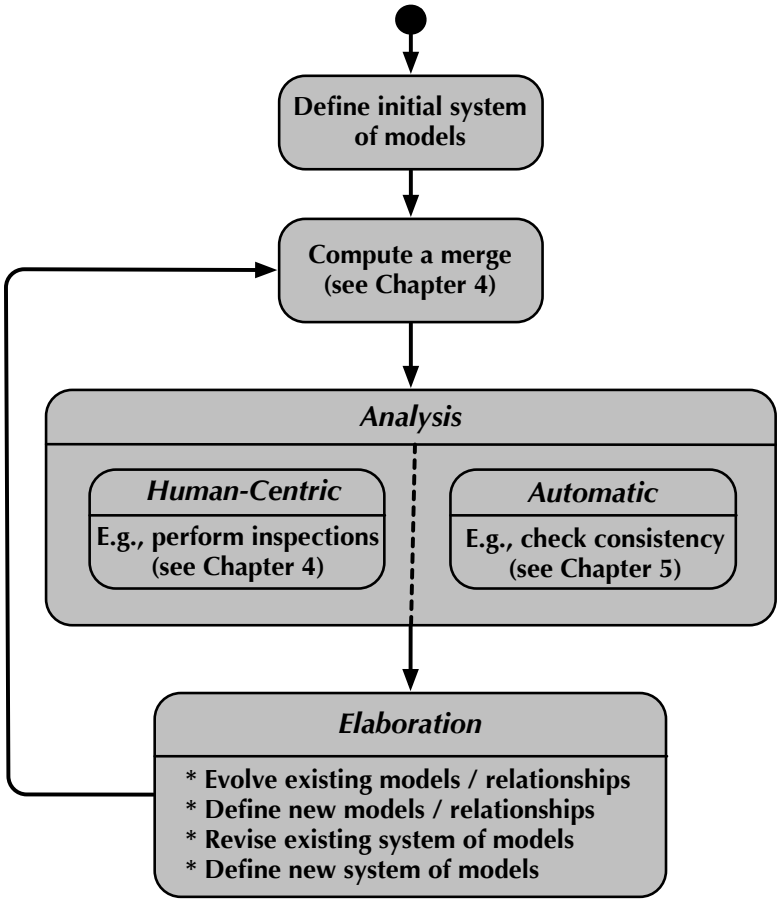


Figure 6.2: Methodology of use of TReMer+

6.1.2 Methodology of Use

Figure 6.2 shows the general methodology of use for TReMer+. We start by defining an initial system of models and then compute the merge of the system. The resulting merge is subjected to various types of exploratory analysis. For example, the merge may be used for manual inspection, or for automatic consistency checking. These analyses often trigger a round of elaboration, where the existing models and relationships are refined and, if necessary, additional models and relationships are introduced to account for any newly discovered aspects of the problem. The interconnection diagram representing the existing system is then revised, or alternatively, a new interconnection diagram is defined, to reflect the elaboration results. After this, we may initiate a new iteration by recomputing the merge and following the subsequent activities.

6.2 Illustrative Applications

In this section, we provide two illustrative use cases for TReMer+: The first use case concerns brainstorming and inspection, and the second – consistency checking.

6.2.1 Brainstorming and Inspection

One of the motivating examples in Chapter 4 was the collaborative development of a database schema (Section 4.5.3). The example showed how merged models can be used for brainstorming and manual inspection. Below, we describe how TReMer+ supports the activities in that motivating example.

Figure 6.3 shows screenshots of the initial models elicited from the stakeholders, Mary and Bob. To describe the relationships between these models, we define a system of models through the interconnection diagram in Figure 6.4. In this interconnection diagram, we declare that we want to relate Mary's and Bob's models through a connector model, **Connector1**, and two mappings **C1-To-Mary** and **C1-To-Bob**. Here, the connector model captures the overlaps between the two source models and the mappings capture how these overlaps are represented in each of the two models.

TReMer+ provides a convenient way for describing mappings. For this purpose, model pairs are shown side-by-side. A correspondence is established by first clicking an element of the model in the left pane and then an element of the model in the right pane. Figure 6.5 illustrates this for the mapping **C1-To-Bob**. To show the desired correspondences, we have added to the screenshot a set of dotted lines indicating the related elements.

After creating the connector model and establishing mappings from it to the source models, we proceed to construct the merge. The result is shown in Figure 6.6. To ensure that the merge is laid out properly, TReMer+ provides a set of automatic layout

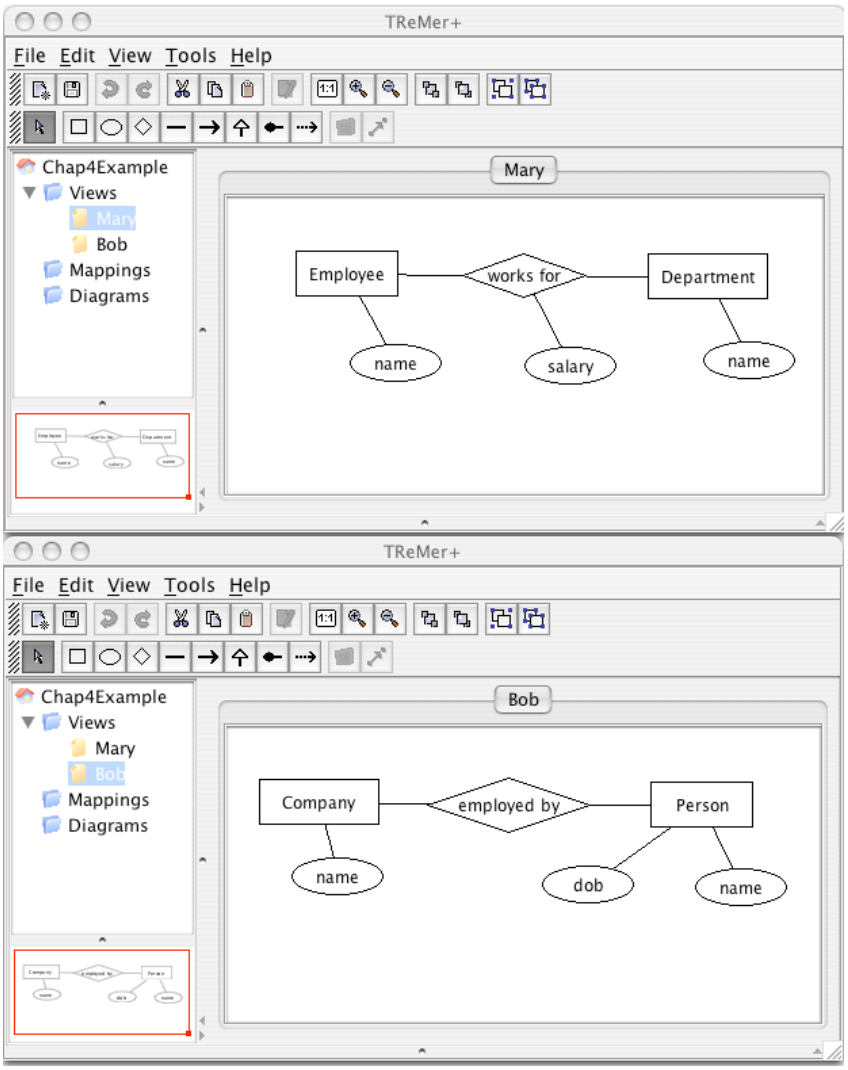


Figure 6.3: Stakeholders' models in the brainstorming and inspection example

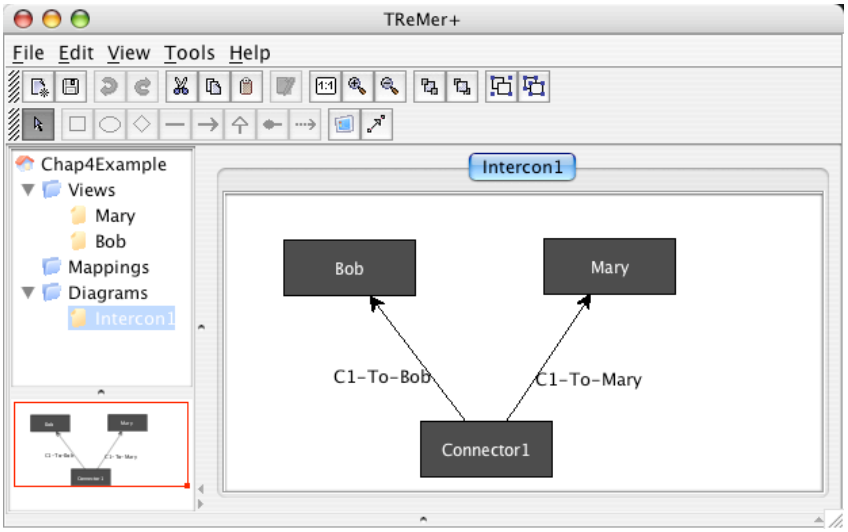


Figure 6.4: Interconnection diagram for relating the stakeholders' models

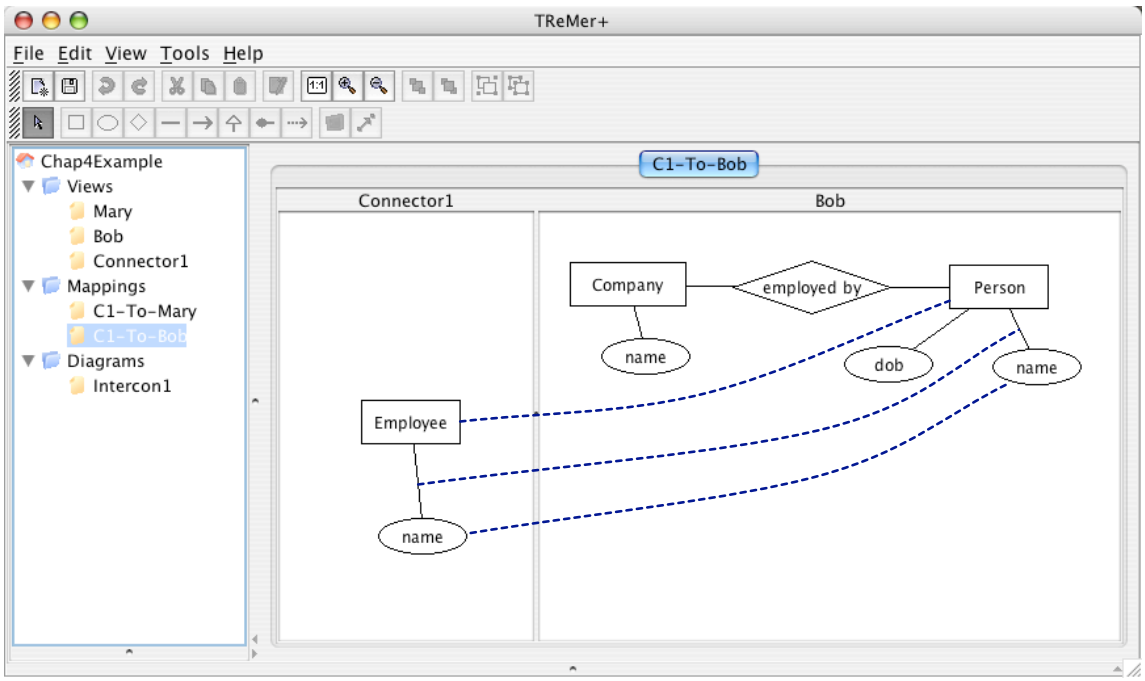


Figure 6.5: Screenshot of the user interface for building mappings

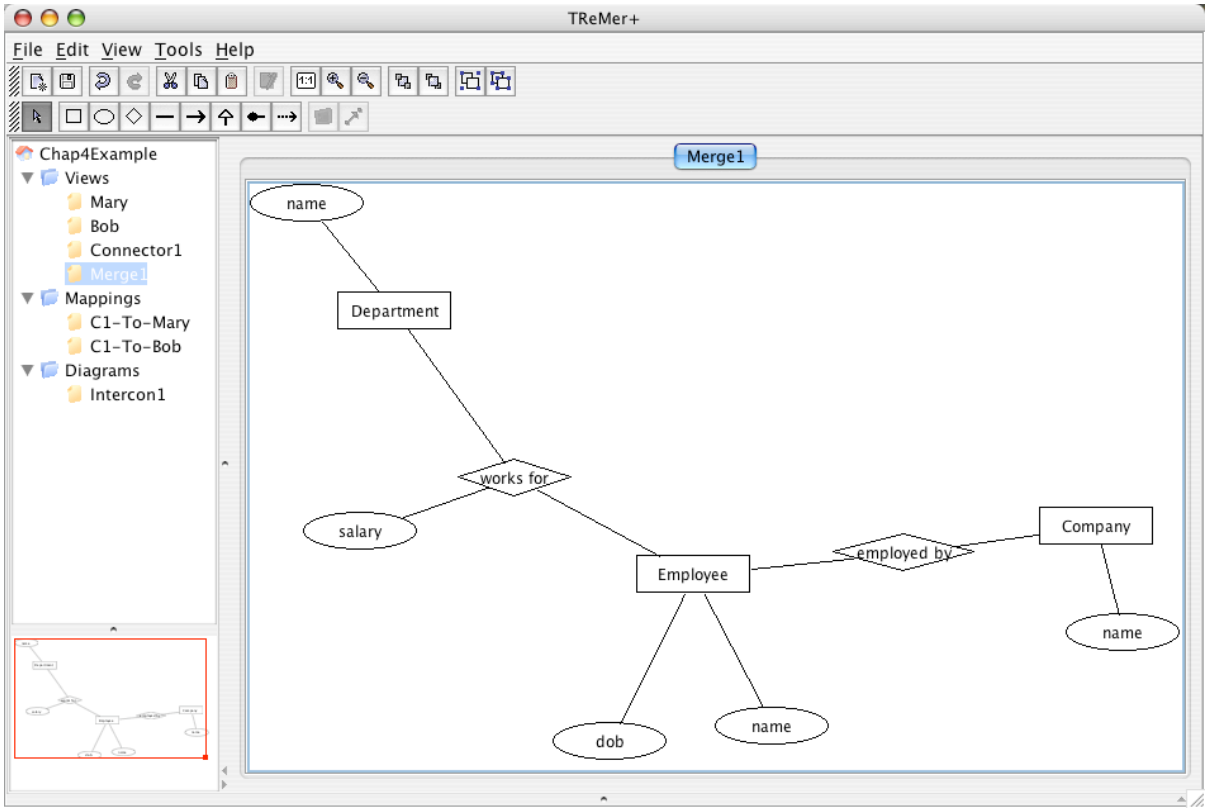


Figure 6.6: Automatically computed merge for the diagram in Figure 6.4

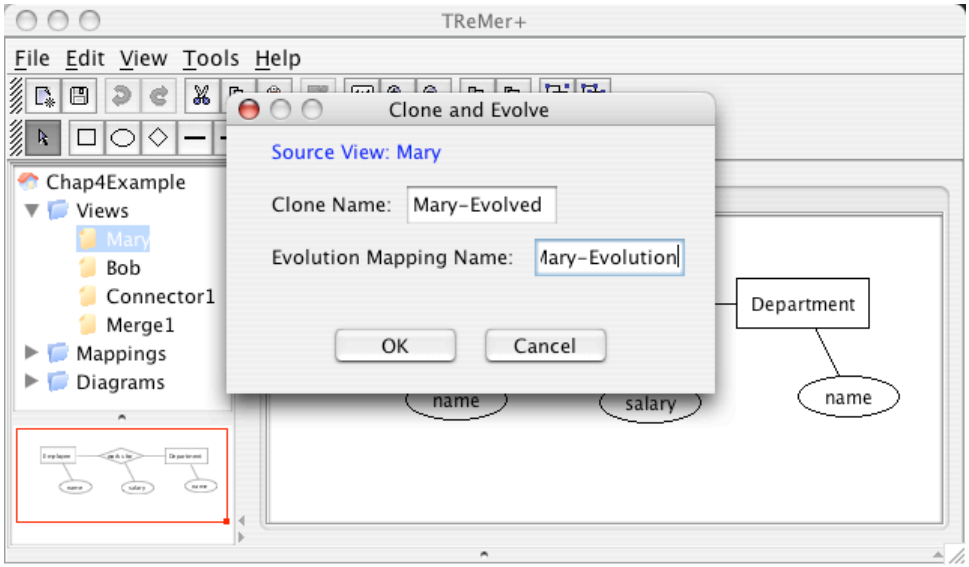


Figure 6.7: The Clone and Evolve feature in TReMer+

algorithms[1]. During the merge process, the tool prompts the user to choose a layout algorithm, and applies the chosen algorithm to the result before displaying it. Currently, TReMer+ cannot preserve the layout of the source models, and ignores all their visual cues during merge. This is a usability issue that we plan to address in the future.

The merge in Figure 6.6 is then manually inspected by the stakeholders. As per the scenario in Chapter 4, Mary (1) deems the employed by relationship redundant, (2) renames Company to Corporation, and (3) adds an aggregation link from Corporation to Department. To reflect these changes, we evolve Mary's original model. To make the evolution process more convenient, TReMer+ provides a "Clone and Evolve" feature. Given a model M, the features creates a clone M' of M and further establishes an identity mapping from M to M' so that the relationship between the clone and its ancestor is preserved. We show in Figure 6.7 how this feature is used to create a clone of Mary's model.

Figure 6.8 shows Mary's evolved model, **Mary-Evolved**, after applying all the desired changes. To represent the belief annotations, we use color coding: The default color represents proposed (**!**), blue and magenta respectively represent confirmed (✔)

[1]These algorithms come with the JGraph editting and visualization framework.

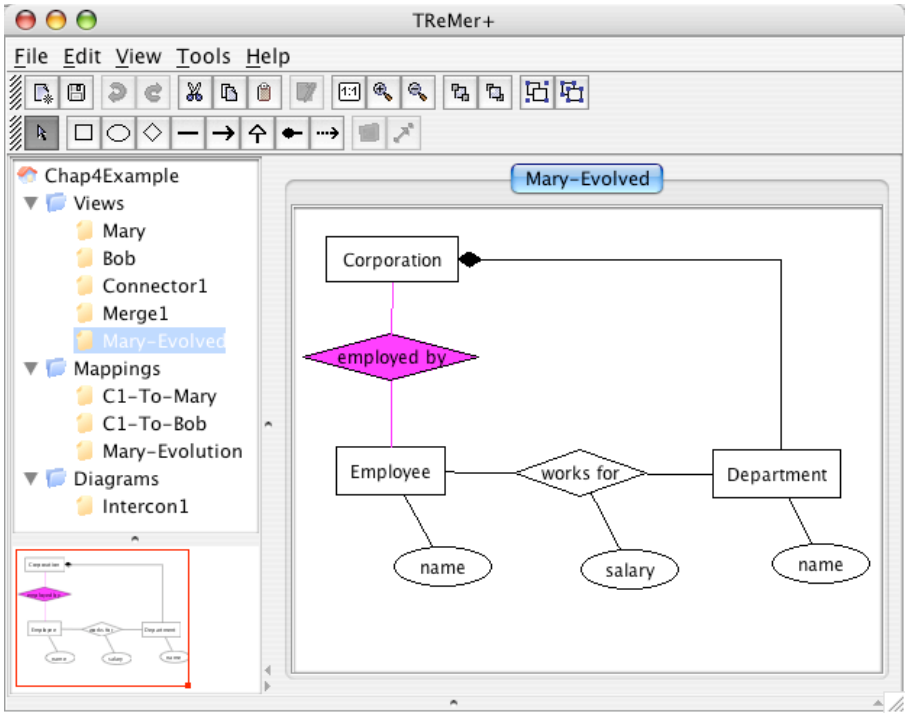


Figure 6.8: Revised perspective of the first stakeholder

and refuted (✘), and red represents disputed (ϟ). For example, in the model shown in Figure 6.8, the employed by relationship as well as the links incident to it are refuted. The remaining elements are at the proposed level.

We must now state that employed by in Mary's model indeed corresponds to the element with the same name in Bob's model. To illustrate TReMer+'s ability to establish direct mappings without introducing connector models (see Section 4.7.3), we declare this correspondence in a slightly different manner than described in Chapter 4: Instead of using a connector model, we define a direct (and partial) mapping between **Mary-Evolved** and **Bob**. This mapping is shown in Figure 6.9. The interconnection diagram after incorporating Mary's evolved model is shown in Figure 6.10. The merge computed for this diagram is shown in 6.11.

Finally, we create an evolved version of Bob's model, again through the Clone and Evolve feature of the tool. Figure 6.12 shows the new model after applying Bob's beliefs, i.e., confirming the employed by relationship (and its incident links). Through this model,

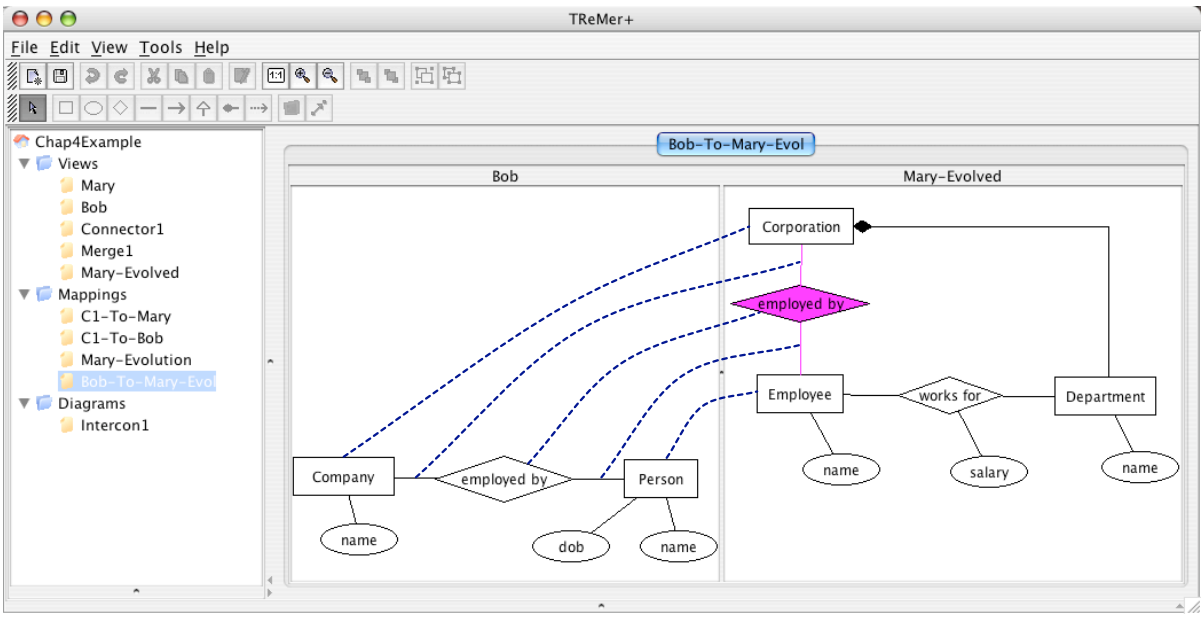


Figure 6.9: Mapping models without introducing connectors

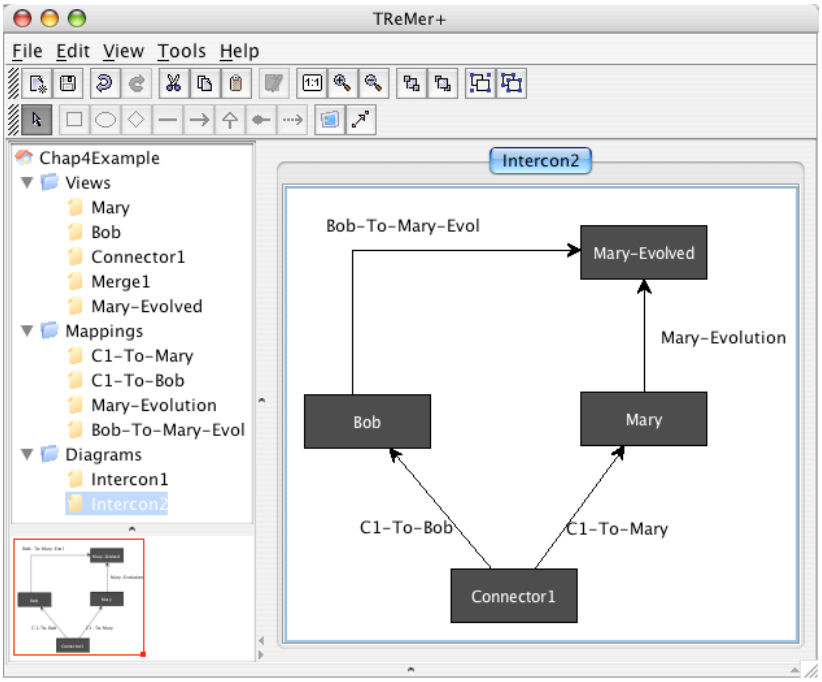


Figure 6.10: Interconnection diagram after first evolution

Bob expresses his certainty about the correctness of the employed by relationship. The interconnection diagram incorporating this evolution and the resulting merge are shown in Figures 6.13 and 6.14, respectively.

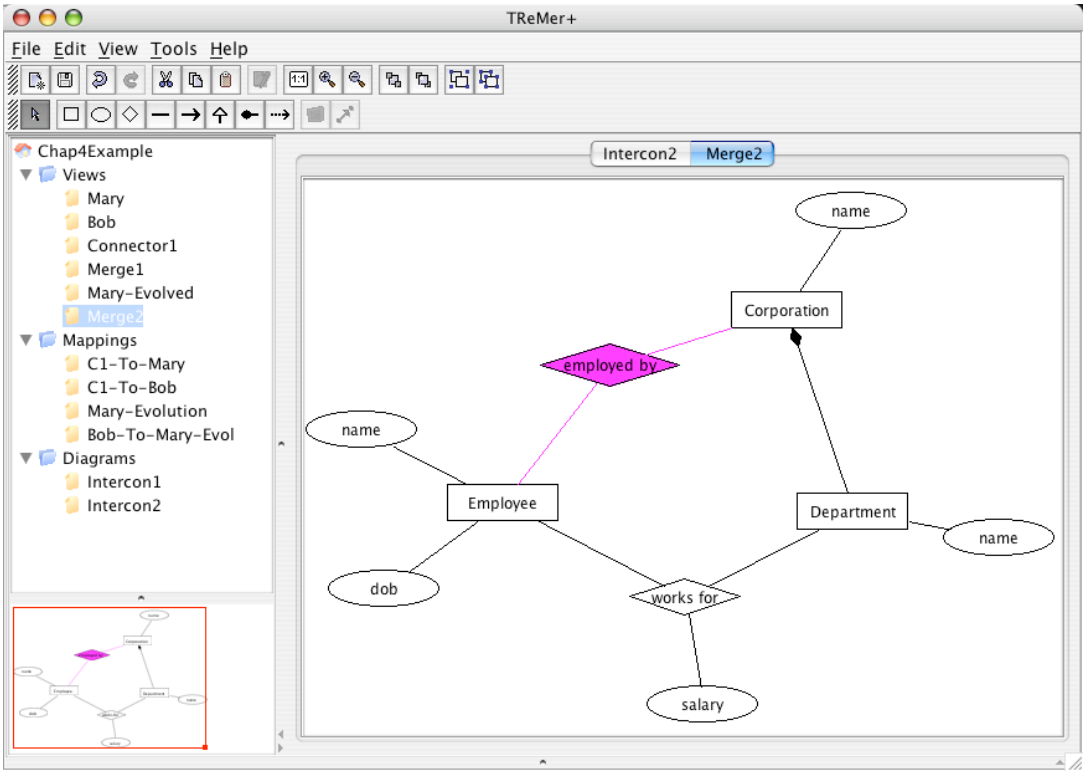


Figure 6.11: Merged model after first evolution

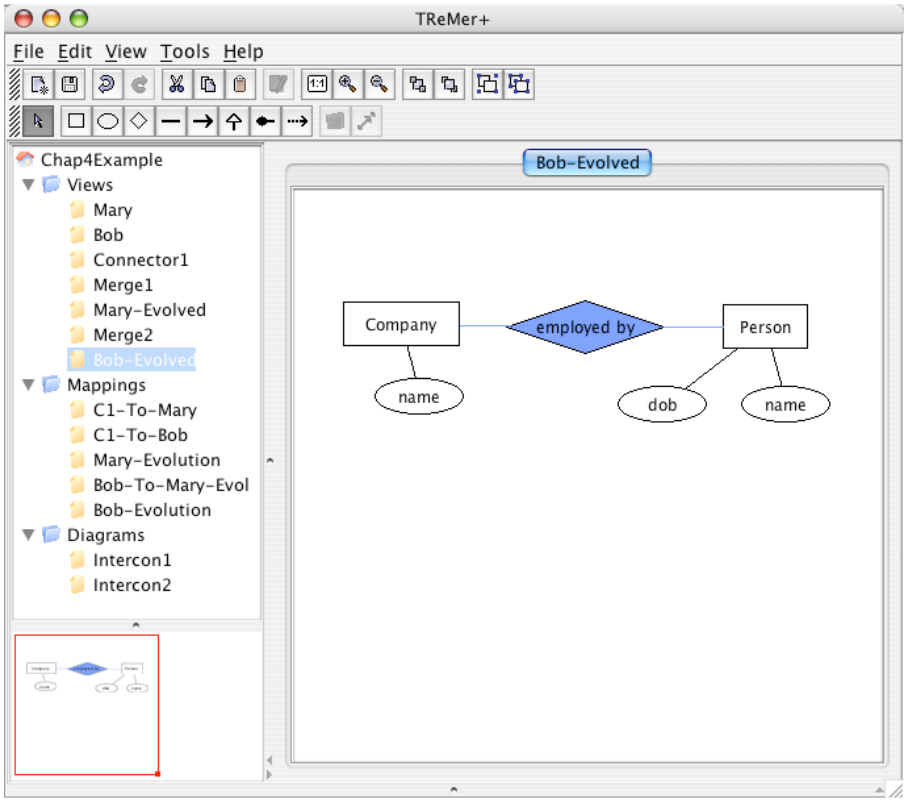


Figure 6.12: Revised perspective of the second stakeholder

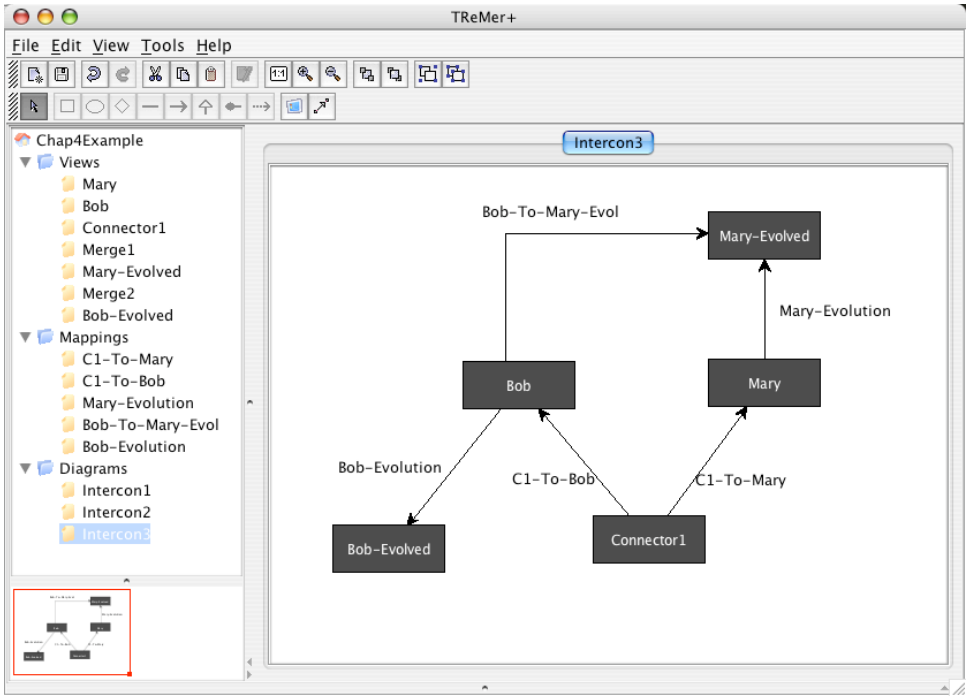


Figure 6.13: Interconnection diagram after second evolution

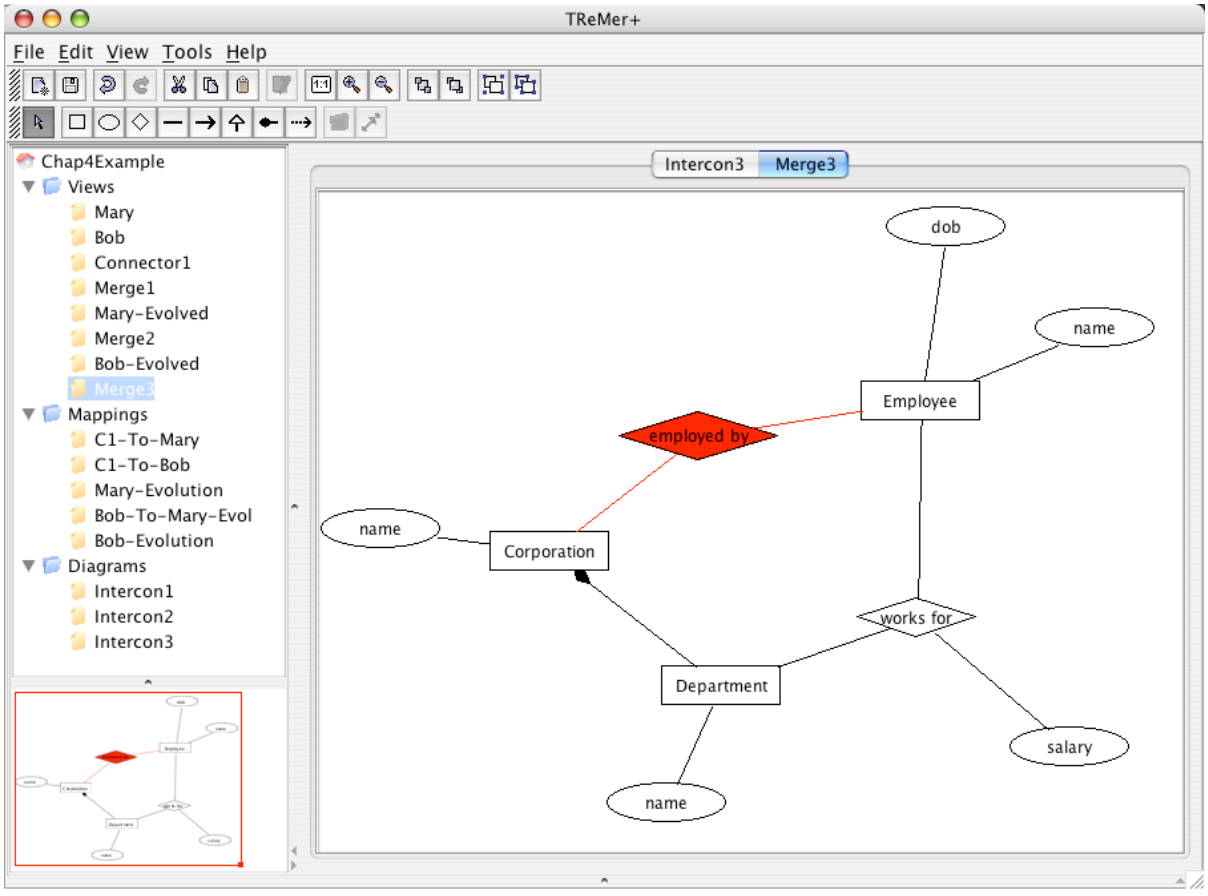


Figure 6.14: Final merge capturing the revisions of both stakeholders

As can be seen from Figure 6.14, the employed by relationship (and its incident links) are displayed in red, meaning that there is a disagreement about the elements. Resolving the disagreement may require a round of negotiation. But, this can be deferred to later stages, because the disagreement is clearly labelled as such in the current merge and will be so represented in any future merge until the disagreement is addressed.

We may now wish to discard the current merge and start over the elaboration cycle, noting that the goal of merge at such an early stage of development is not to create a blueprint, but rather to let the stakeholders and developers experiment with possible alternatives and gain more knowledge about the system being built. The increments in knowledge were explicitly captured by evolving the source models, and the points of disagreement can be reproduced whenever desired, by recomputing the merge. Hence, discarding the current merge does not result in the loss of any conceptual information.

6.2.2 Consistency Checking

In Section 6.2.1, we described how to build systems of interrelated models in TReMer+ and merge them. We now concentrate on the consistency checking and inconsistency navigation capabilities of TReMer+.

Figure 6.15 shows an overview of the consistency checking use case in TReMer+: Having defined a system of interrelated models, we begin by merging the system. This yields a (potentially inconsistent) merged model along with traceability links from it to the source system. In the next step, we check the consistency of this merged model against the (intra-model) constraints of interest using CrocoPat (Beyer *et al.*, 2005), and generate appropriate diagnostics for any violations found. By utilizing the traceability data for the merge, TReMer+ enables navigation from the diagnostics to the source models and mappings involved in every inconsistency instance.

To illustrate consistency checking in TReMer+, we use the system in Figure 5.2 (Chapter 5) as the running example. We have provided in Figure 6.16 screenshots of the

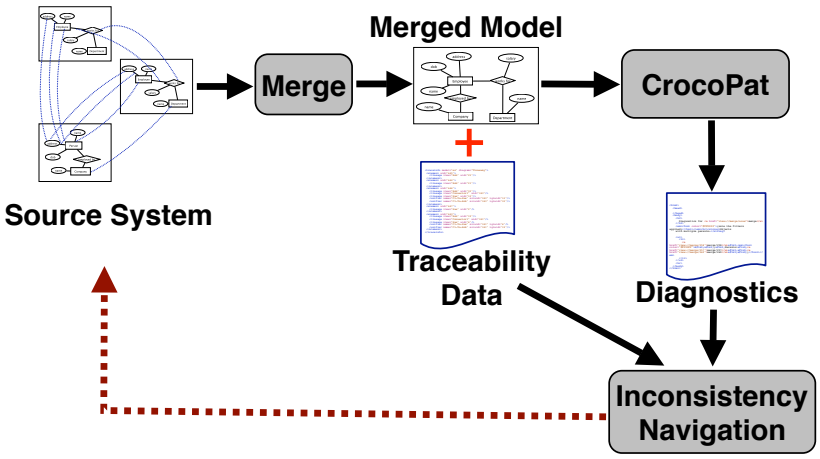


Figure 6.15: Consistency checking with TReMer+.

models and mappings in that system. Note that the mappings were established directly without using connector models.

The interconnection diagram for the system is shown in Figure 6.17(a), and the resulting merge in Figure 6.20 (for now, ignore the diagnostics). Using interconnection diagrams in the consistency checking problem offers two advantages: (1) It allows developers to narrow down the scope of their analysis to a desired subsystem. For example, if we were interested in pairwise checking of M_1 and M_2 (with respect to R_1), we would use the interconnection diagram in Figure 6.17(b). (2) The project may include outdated or competing versions of the models and mappings, in which case one needs to explicitly choose the versions to be included in the analysis.

Figure 6.18 shows how CrocoPat is invoked by TReMer+. In the first step, the model to be checked is translated into a set of first order predicates using the translation algorithm in Figure 5.5 (Chapter 5). The result, along with a user-selected set of consistency rules, is then passed to CrocoPat for consistency checking and generation of diagnostics. Rule selection is done through the dialog box shown in Figure 6.19.

The diagnostics generated by CrocoPat are presented to the user through TReMer+'s user interface. Figure 6.20 shows the diagnostics for the merge in our running example. As can be seen from the figure, the diagnostics refer to model elements through their

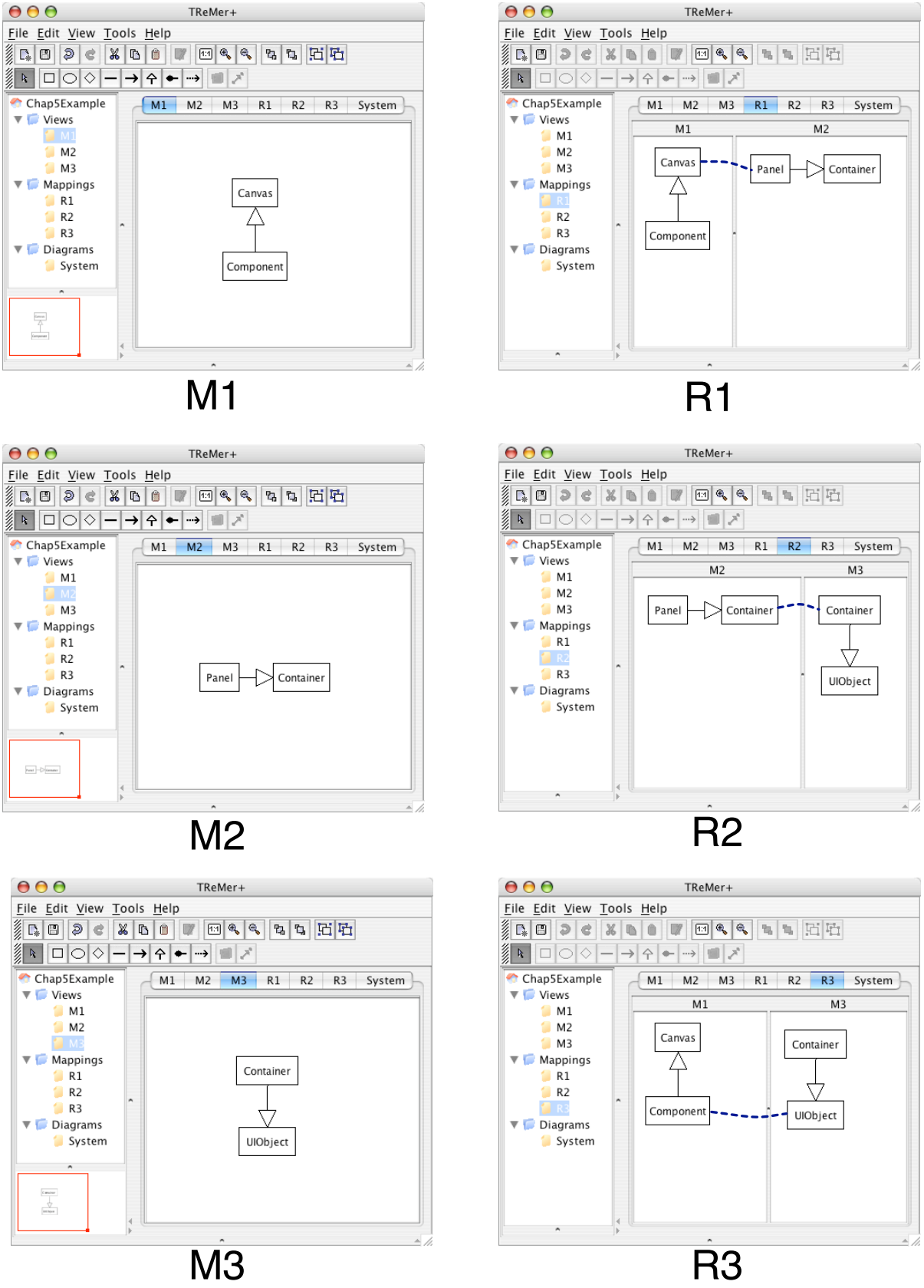


Figure 6.16: Models and mappings in the consistency checking example

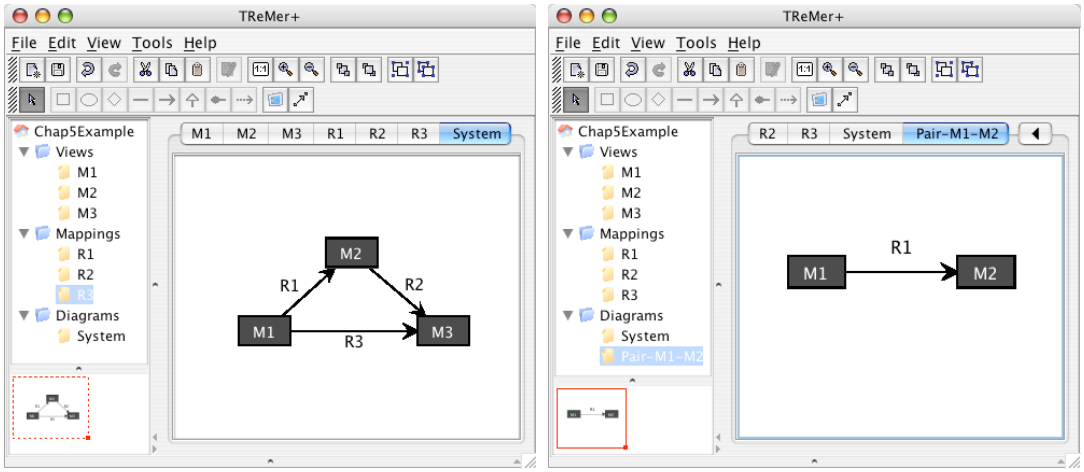


Figure 6.17: Interconnection diagrams for consistency checking

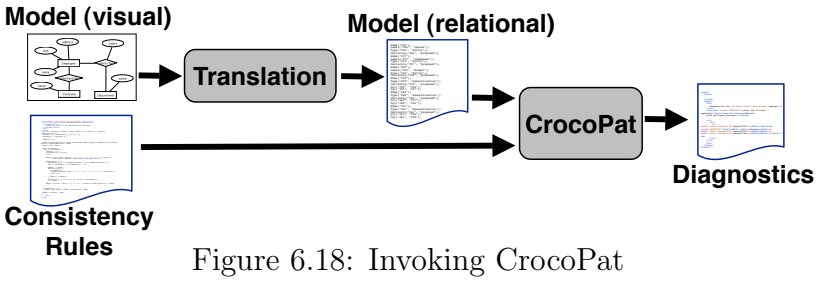


Figure 6.18: Invoking CrocoPat

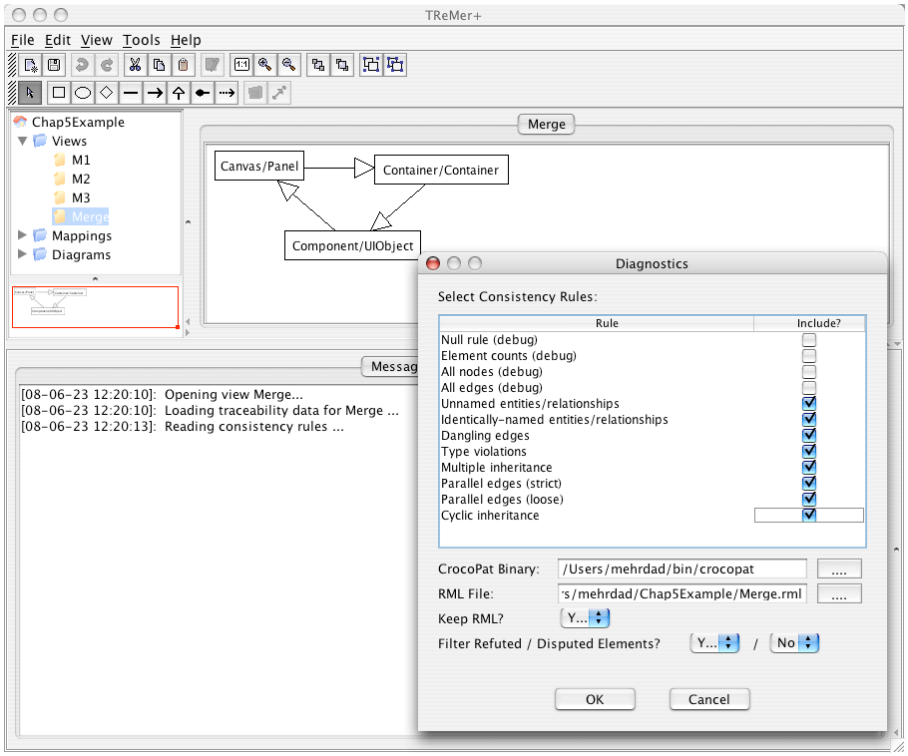


Figure 6.19: Choosing the consistency rules

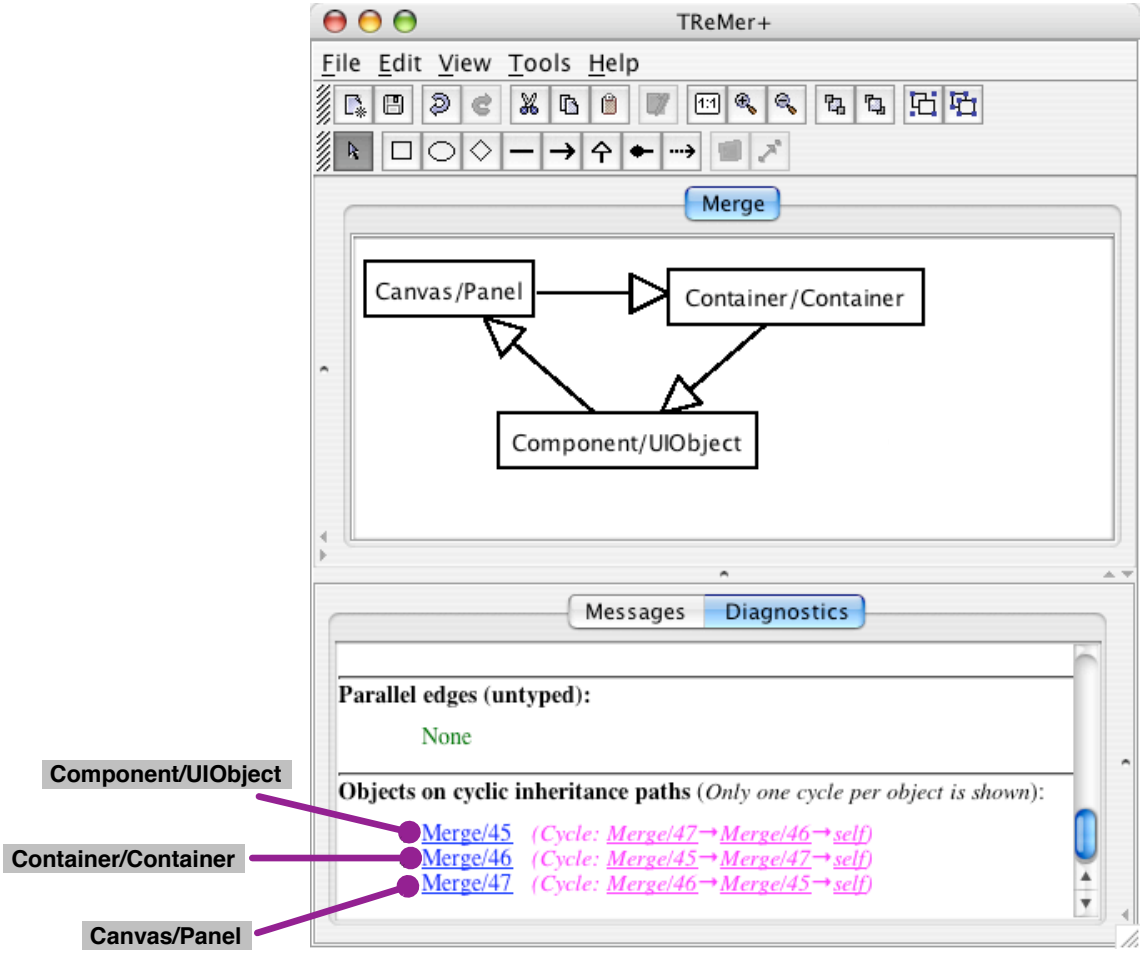


Figure 6.20: Inconsistency diagnostics

unique identifiers (uids). The results for our example indicate that the inheritance hierarchy is cyclic. The inconsistency report includes a list of offending classes, and for each such class, a counterexample path. Currently, we ignore symmetries between inconsistencies; hence, three distinct errors are reported for the same cycle in Figure 6.20.

In the diagnostics, textual references to model elements are hyperlinked to the actual graphical elements. Typically, inconsistencies are explored in the following way:

1. The user clicks a link in the diagnostics. This causes the corresponding graphical element in the (merged) model to be highlighted. At the same time, the traceability data for the clicked element is retrieved and displayed. Figure 6.21 shows the setting immediately after the user has clicked Merge/47 (i.e., Canvas/Panel) in the diagnostics.

The traceability data, shown in the projections pane, makes it possible to trace the element back to its sources, i.e., Canvas in M_1, Panel in M_2, and the correspondence between these two elements defined in R_1.

2. Using the traceability information (which is also in a hyperlinked format), the user navigates to the originating models and mappings behind the inconsistencies. For example, if the user clicks M2/uid:19 in the projections pane of Figure 6.21, she will be taken to the screen shown at the bottom right of the figure, with the Panel element highlighted. Similarly, if she clicks R1/(uid:15, uid:19), she will be taken to the screen at the bottom left, with Canvas and Panel highlighted in the mapping. This helps the user understand why Canvas and Panel were unified.

TReMer+ further provides traceability at the level of interconnection diagrams, shown in Figure 6.22. The navigation process is exactly as explained above, with the only difference being that the data in the projections pane points to the abstract models and mappings in the interconnection diagram inducing the merge. This is useful when the user does not want to zoom into the details of the source models and mappings, and only wants to get a bird's eye view of the models and mappings involved in a particular inconsistency instance.

Finally, we note that, similarly to the brainstorming and inspection use case in Section 6.2.1, the merge in our consistency checking use case was not intended to be the end result. Instead, the merge was used for identification and exploration of anomalies in the source models and mappings.

6.3 Summary and Future Work

We presented a tool, TReMer+, for merging and consistency checking of distributed models. We provided two illustrative use cases, showing how the tool can be applied in practice.

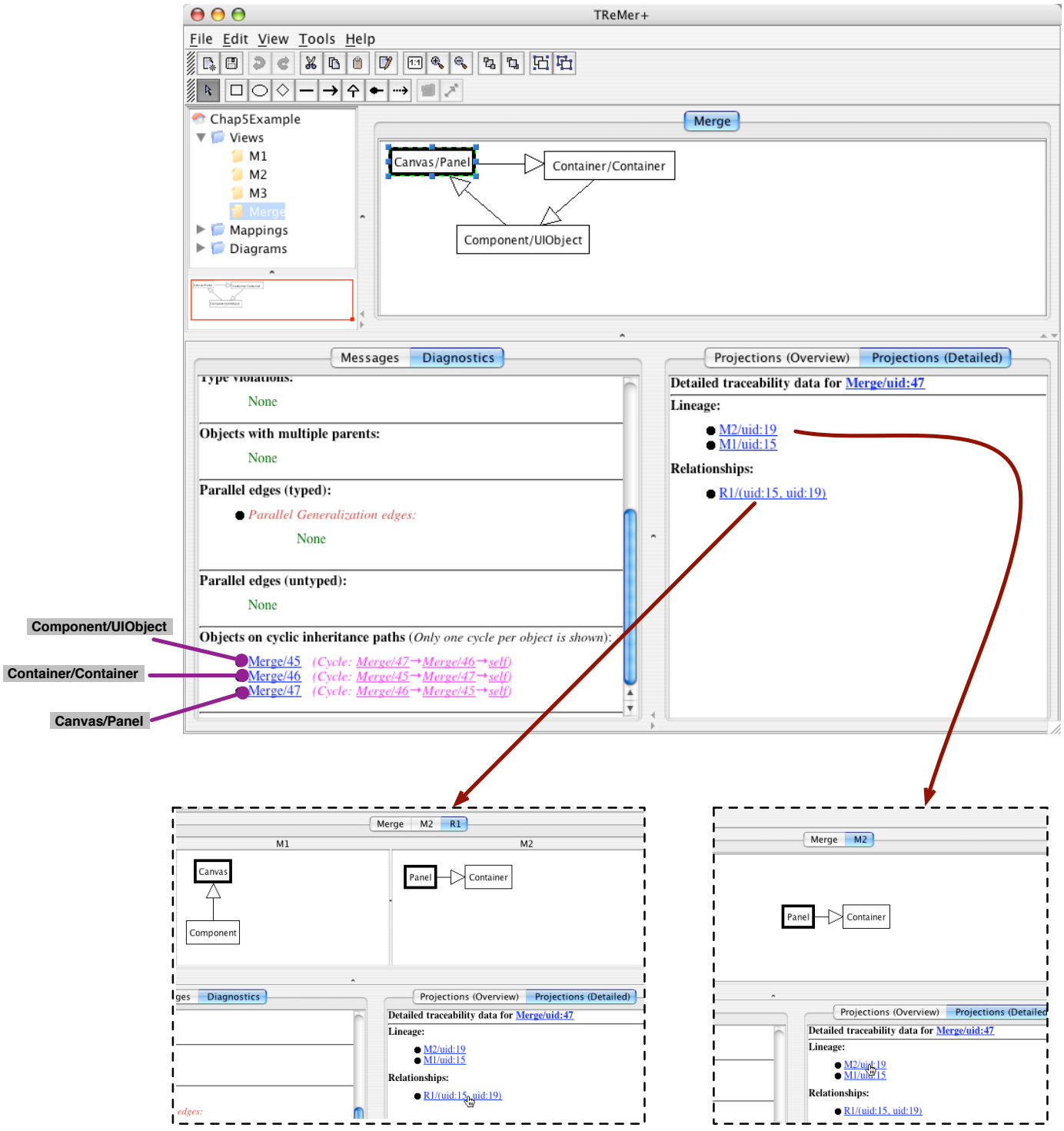


Figure 6.21: Inconsistency navigation

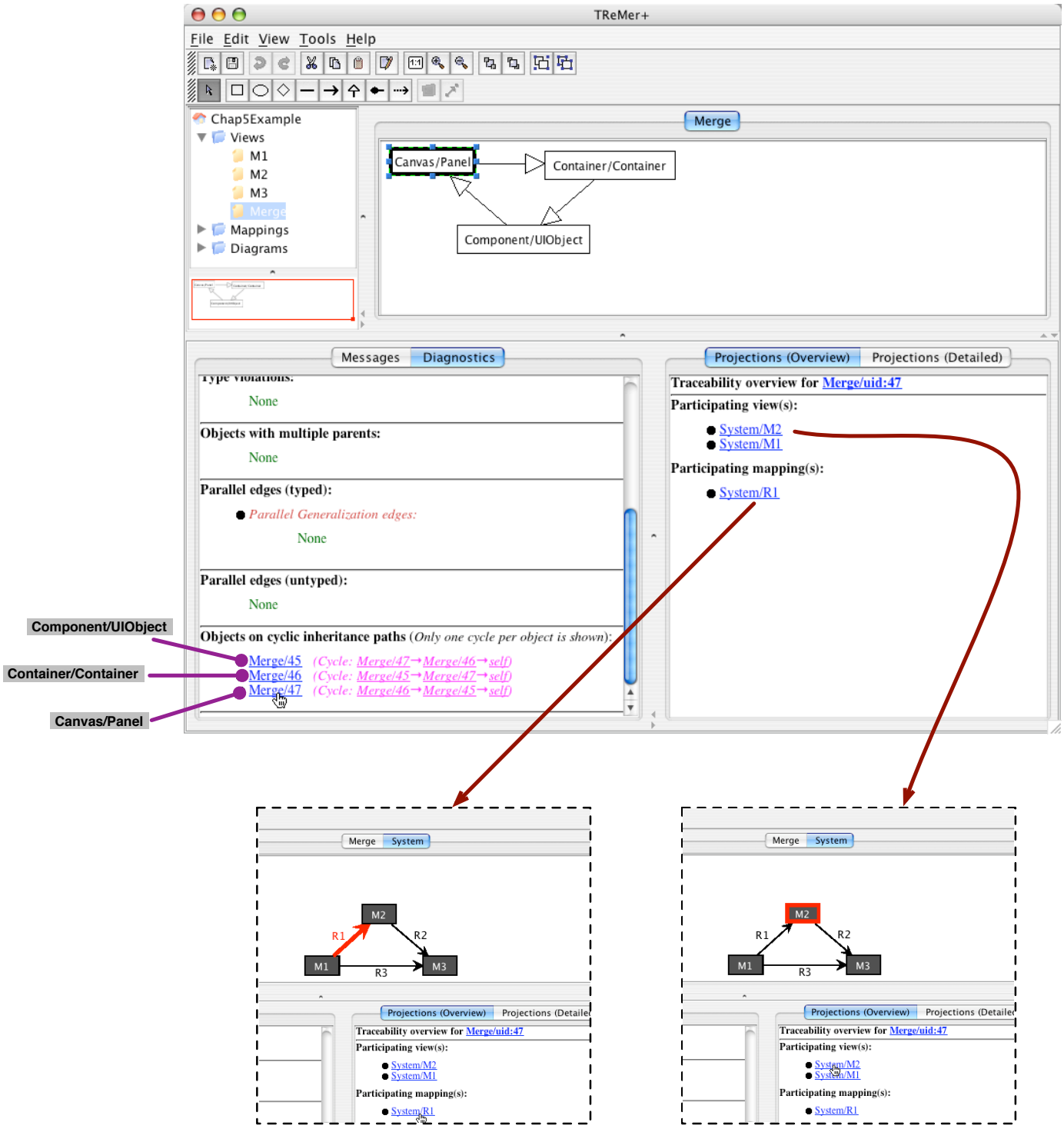


Figure 6.22: Traceability at the level of interconnection diagrams

In the future, we would like to extend TReMer+ with model matching and slicing operators to support more complex exploratory analyses. Another important area for future work is to improve the usability of TReMer+ so that it can be used in large-scale studies.

TReMer+ is part of a broader research effort to build usable model-based development tools. A complementary aspect is a project aimed at developing a customizable Eclipse-based platform for model management. A preliminary outline of this project is given in (Salay *et al.*, 2007).

Chapter 7

Conclusion

To conclude, we summarize our main contributions, describe current limitations, and provide directions for future work.

7.1 Thesis Summary

After reviewing the conceptual and mathematical background for our work in Chapters 2 and 3, we presented the following contributions:

- In Chapter 4, we developed a framework for merging incomplete and inconsistent models. We introduced a formalism, called annotated graphs, with a built-in annotation scheme for capturing incompleteness and inconsistency. We showed how structure-preserving mappings can be used to specify the relationships between disparate models expressed as annotated graphs, and provided a general algorithm for merging arbitrary systems of (interrelated) models. We addressed the problem of traceability in model merging and described methods for establishing traceability links between a merged model and the source models and mappings.

- In Chapter 5, we developed an approach for verifying global consistency properties of systems of models. The approach works by first constructing a merged model

and then verifying this model against the consistency constraints of interest. We showed how the traceability information generated during merge can be utilized to project the inconsistencies found over the merged model back to the source models and mappings. We presented a set of reusable expressions for defining consistency constraints in conceptual modelling. We illustrated the use of the developed expressions in the specification of consistency rules for class, entity-relationship, and goal diagrams. We evaluated our consistency checking approach using a case study.

- In Chapter 6, we described a tool implementing our merge and consistency checking techniques and illustrated the tool through simple exemplars.

Throughout the thesis, we emphasized the need to treat model management as a set of activities over *systems* of models, rather than over individual models and pairs of models. We believe that such treatment holds the potential to address many of the exploration and integration problems that developers face in distributed software projects.

7.2 Limitations

Below, we highlight the important limitations of our work.

Approach Limitations. Our focus in this thesis was on *homogeneous* models. We have not attempted to generalize our work to heterogeneous models yet. One could apply our merge technique to heterogeneous models by translating all the models into a single notation first, but such a translation discards the structure of the original models and the visual aspects of the original notations.

Since we use a merged model for exploration of inconsistencies, our consistency checking technique is currently limited to homogeneous models, too. This limitation can be addressed by augmenting the inconsistency exploration process so that inconsistencies are

not visualized over the merge but directly over the source models and mappings. This makes the model merging step transparent to end-users; therefore, having to translate the models into a single notation before merge will no longer be an issue for consistency checking.

Tool Support and Evaluation Limitations. The ultimate evaluation of our contributions is whether developers faced with real model management tasks find our work useful. In particular, we would like to know (1) whether our merge and consistency checking operators improve time-to-completion of complex modelling activities? and (2) whether the operators improve the quality of the developed artifacts, e.g., in terms of understandability, validity, consistency, abstraction, etc?

Providing answers to these questions depends not only on the technical merits of our operators, but also on the usability of the tool that implements the operators. Our current tool is a prototype developed primarily to prove feasibility. Despite the considerable effort we have invested in building tool support, more work is required before our tool is usable enough for deployment in large-scale projects. The evaluation presented in this thesis only demonstrated the novel analyses that our operators make possible. We do not yet have sufficient empirical evidence to determine how these analyses translate into improved productivity and quality.

7.3 Future Directions

In this section, we outline some of the research threads that we intend to follow in our future work.

7.3.1 Richer Framework for Exploratory Analysis

One of the shorter-term focuses of our future research is to extend our existing set of model management operators to support richer exploratory activities. A key aspect of this

agenda is to provide a semi-automated (interactive) Match operator, so that developers can seed some of the more obvious relations, and iteratively refine the results by applying matching and conducting manual inspections. Alternatively, a developer may want to assess the output of the Match operator through automated analyses, e.g., consistency checking, to ensure that the matching results lead to meaningful relationships between the models.

Another interesting topic would be to develop a Slice operator. Such an operator will be a great aid for exploration by allowing developers to quickly narrow down the scope of their investigation to the desired aspects of a large model or relationship.

7.3.2 Enhancing Usability

With hindsight, one of the most significant outcomes of our efforts towards building distributed modelling tools has been to demonstrate how little infrastructure is available for implementing such tools. The key observation here is that existing modelling platforms, e.g., the Rational Software Architect (IBM Rational Software Architect, 2007), are primarily aimed at centralized development, where all developers contribute to a single holistic model. These platforms lack support, particularly in terms of user interface, for important distributed development activities such as constructing explicit relationships between models, and defining and navigating systems of interrelated models.

In this thesis, we proposed and prototyped a rich graphical front-end that supports some of these activities. In the future, we intend to enhance this front-end so that it can be used by practitioners in production environments. This calls for interdisciplinary research between software engineering, human-computer interaction, and cognitive psychology, with the ultimate goal of developing highly usable interfaces for distributed development.

7.3.3 Formal Semantics for Models

For a long time, models were perceived as peripheral artifacts to programs, and were used only for documentation purposes. As a result, models were seldom subject to rigorous analysis and there was little need to formalize their semantics. The perception of the role of models in software development has changed dramatically in the past few years thanks to the increasing popularity of model-driven engineering, which promotes models, rather than programs, as the focus and primary artifacts of development. With this change of perception has come a growing realization that models need to have formal semantics if they are to be used as the basis for automated verification, transformation, and testing of software.

In future research, we would like to study ways to formalize the semantics of models and leverage the use of models for automated reasoning about safety, security, and reliability of software systems. Our work will focus on domain-specific modelling languages, i.e., languages created specifically to address problems in a particular domain. Examples of such languages include SysML (OMG Systems Modeling Language, 2006), a subset of the UML language tailored to systems engineering applications, and Boxtalk (Zave & Jackson, 2002), a state machine language customized for building telecommunication features. While these languages offer less generality than their general-purpose counterparts, they are much more amenable to formalization, and hence provide a lot more credibility in terms of the results that can be derived from automated analysis of models.

7.3.4 Scalability and Ultra-Large-Scale Systems.

Modelling and analysis techniques that scale well are crucial to the future of software engineering. It is expected that existing development paradigms will fail beyond certain scale thresholds (Cheng & Atlee, 2007). A class of systems that defy existing paradigms are the emerging Ultra-Large-Scale (ULS) systems (Feiler *et al.*, 2006). Examples of ULS

systems include next-generation health-care systems, and critical infrastructure management systems (e.g., systems that control water, communication, and power supplies).

The challenges anticipated in the development of ULS systems have a lot in common with the challenges seen in today's distributed software projects. Specifically, ULS systems will include software created and modified by dispersed teams with different backgrounds, goals, and stakeholders. Therefore, a ULS system will necessarily be made up of overlapping artifacts, with potential inconsistencies in their design and usage. Hence, effective construction of ULS systems will depend on having scalable techniques for inconsistency management. We believe the premises for the research reported in this thesis are very much in line with the needs of ULS systems. In the future, we would like to study the problem of inconsistency management in the context of very large and heterogeneous systems and develop innovative model manipulation techniques with potential for application to ULS systems.

References

Alagic, S., & Bernstein, P. 2001. A Model Theory for Generic Schema Management. *Pages 228–246 of: Proceedings of the 8th International Workshop on Database Programming Languages.*

Alanen, M., & Porres, I. 2003. Difference and Union of Models. *Pages 2–17 of: Proceedings of the 6th International Conference on The Unified Modeling Language, Modeling Languages and Applications.*

Aumueller, D., Do, H., Massmann, S., & Rahm, E. 2005. Schema and Ontology Matching with COMA++. *Pages 906–908 of: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data.*

Baader, F., Calvanese, D., McGuinness, D., Nardi, D., & Patel-Schneider, P. (eds). 2003. *The Description Logic Handbook: Theory, Implementation, and Applications.* Cambridge University Press.

Barr, M., & Wells, C. 1984. *Toposes, Triples and Theories.* Grundlehren Math. Wiss. Springer. `ftp://ftp.math.mcgill.ca/pub/barr`.

Barr, M., & Wells, C. 1999. *Category Theory for Computing Science.* third edn. Montreal, Canada: Les Publications CRM Montréal.

Batini, C., Lenzerini, M., & Navathe, S. 1986. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, **18**(4), 323–364.

Belnap, N. 1977. A Useful Four-Valued Logic. *Pages 5–37 of:* Epstein, G., & Dunn, J. (eds), *Modern Uses of Multiple-Valued Logic*. Dordrecht, Netherlands: Reidel.

Bernstein, P. 2003. Applying Model Management to Classical Meta Data Problems. *Pages 209–220 of: Proceedings of the 1st Biennial Conference on Innovative Data Systems Research*.

Bernstein, P., Melnik, S., Petropoulos, M., & Quix, C. 2004. Industrial-Strength Schema Matching. *SIGMOD Record*, **33**(4), 38–43.

Beyer, D., & Noack, A. 2004. *CrocoPat 2.1 Introduction and reference manual*. Tech. rept. University of California, Berkeley.

Beyer, D., Noack, A., & Lewerentz, C. 2005. Efficient Relational Calculation for Software Analysis. *IEEE Transactions on Software Engineering*, **31**(2), 137–149.

Birkhoff, G. 1979. *Lattice Theory*. third edn. American Mathematical Society.

Blum, B. 1992. *Software Engineering: A Holistic View*. Oxford University Press.

Bolognesi, T., & Brinksma, E. 1987. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, **14**(1), 25–59.

Borceux, F. 1994. *Handbook of Categorical Algebra*. Encyclopedia of Mathematics and its Applications. Cambridge University Press.

Bowman, H., Steen, M., Boiten, E., & Derrick, J. 2002. A Formal Framework for Viewpoint Consistency. *Formal Methods in System Design*, **21**(2), 111–166.

Brunet, G., Chechik, M., Easterbrook, S., Nejati, S., Niu, N., & Sabetzadeh, M. 2006. A manifesto for model merging. *Pages 5–12 of: Proceedings of the 1st International Workshop on Global Integrated Model Management*.

Bryant, R. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, **8**, 677–691.

Buneman, P., Davidson, S., & Kosky, A. 1992. Theoretical Aspects of Schema Merging. *Pages 152–167 of: Proceedings of the 3rd International Conference on Extending Database Technology.*

Cadish, B., & Diskin, Z. 1996. Heterogeneous View Integration via Sketches and Equations. *Pages 603–612 of: Proceedings of the 9th International Symposium on Foundations of Intelligent Systems.*

CASCON Workshop on Model Fusion. 2006. `http://www.cs.toronto.edu/~chechik/ModelFusion/`. Chairs: Marsha Chechik, Steve Easterbrook, Kim Letkeman, Sebastian Uchitel and Jourgen Dingel.

Cheng, B., & Atlee, J. 2007. Research Directions in Requirements Engineering. *Pages 285–303 of: Future of Software Engineering Track of the 29th International Conference on Software Engineering.*

Clarke, E., Grumberg, O., & Peled, D. 1999. *Model Checking.* MIT Press.

Corradini, A., Montanari, U., & Rossi, F. 1996. Graph processes. *Fundamenta Informaticae*, **26**(3–4), 241–265.

Darke, P., & Shanks, G. 1996. Stakeholder viewpoints in requirements definition: A framework for understanding viewpoint development approaches. *Requirements Engineering*, **1**(2), 88–105.

Davey, B., & Priestley, H. 2002. *Introduction to Lattices and Order.* second edn. Cambridge, UK: Cambridge University Press.

Delugach, H. 1992. *Conceptual structures: current research and practice.* Ellis Horwood. Chap. Analysing multiple views of software requirements, pages 391–410.

Dwyer, M., Avrunin, G., & Corbett, J. 1999. Patterns in property specifications for finite-state verification. *Pages 411–420 of: Proceedings of the 21st International Conference on Software Engineering.*

Easterbrook, S. 1993. Domain Modeling with Hierarchies of Alternative Viewpoints. *Pages 65–72 of: Proceedings of the 1st International Symposium on Requirements Engineering.*

Easterbrook, S. 1994. Resolving Requirements Conflicts with Computer-Supported Negotiation. *Pages 41–65 of:* Jirotka, M., & Goguen, J. (eds), *Requirements Engineering: Social and Technical Issues.* Academic Press.

Easterbrook, S., & Chechik, M. 2001. A Framework for Multi-Valued Reasoning over Inconsistent Viewpoints. *Pages 411–420 of: Proceedings of the 23rd International Conference on Software Engineering.*

Easterbrook, S., & Nuseibeh, B. 1996. Using Viewpoints for Inconsistency Management. *Software Engineering Journal*, **11**(1), 31–43.

Easterbrook, S., Finkelstein, A., Kramer, J., & Nuseibeh, B. 1994. Coordinating Distributed ViewPoints: The Anatomy of a Consistency Check. *International Journal on Concurrent Engineering: Research and Applications*, **2**, 209–222.

Easterbrook, S., Yu, E., Aranda, J., Fan, Y., Horkoff, J., Leica, M., & Qadir, R. 2005. Do Viewpoints Lead to Better Conceptual Models? An Exploratory Case Study. *Pages 199–208 of: Proceedings of the 13th International Requirements Engineering Conference.*

Egyed, A. 2000. *Heterogeneous View Integration and its Automation.* Ph.D. thesis, University of Southern California, USA.

Egyed, A. 2001. A scenario-driven approach to traceability. *Pages 123–132 of: Proceedings of the 23rd International Conference on Software Engineering.*

Egyed, A. 2006. Instant consistency checking for the UML. *Pages 381–390 of: Proceedings of the 28th International Conference on Software Engineering.*

Ehrig, H., & Pfender, M. 1972. *Kategorien und Automaten.* de Gruyter Lehrbuch. Walter de Gruyter.

Ehrig, H., & Taentzer, G. 1996. Computing by Graph Transformation, A Survey and Annotated Bibliography. *Bulletin of the European Association for Theoretical Computer Science*, **59**, 182–226.

Elaasar, M., & Briand, L. 2004. *An Overview of UML Consistency Management.* Tech. rept. SCE-04-18. Carleton University.

Feather, M., Fickas, S., Finkelstein, A., & van Lamsweerde, A. 1997. Requirements and Specification Exemplars. *Automated Software Engineering*, **4**(4), 419–438.

Feiler, P., Gabriel, R., Goodenough, J., Linger, R., Longstaff, T., Kazman, R., Klein, M., Northrop, L., Schmidt, D., Sullivan, K., & Wallnau, K. 2006. *Ultra-Large-Scale Systems: The Software Challenge of the Future.* SEI Technical Reports. Carnegie Mellon University.

Fiadeiro, J., & Maibaum, T. 1992. Temporal Theories as Modularisation Units for Concurrent System Specification. *Formal Aspects of Computing*, **4**(3), 239–272.

Filman, R., Elrad, T., Clarke, S., & Aksit, M. 2004. *Aspect-Oriented Software Development.* Reading, USA: Addison-Wesley.

Finkelstein, A., & Sommerville, I. 1996. The Viewpoints FAQ. *Software Engineering Journal: Special Issue on Viewpoints for Software Engineering*, **11**(1), 2–4.

Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., & Goedicke, M. 1992. Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. *International Journal of Software Engineering and Knowledge Engineering*, **2**(1), 31–58.

Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J., & Nuseibeh, B. 1994. Inconsistency Handling in Multi-perspective Specifications. *IEEE Transactions on Software Engineering*, **20**(8), 569–578.

Fischer, G., Lemke, A., McCall, R., & Morch, A. 1996. Making argumentation serve design. *Pages 267–293 of:* Moran, T., & Carroll, J. (eds), *Design rationale: concepts, techniques, and use*. Mahwah, NJ, USA: Lawrence Erlbaum Associates.

Fradet, P., Métayer, D. Le, & Périn, M. 1999. Consistency Checking for Multiple View Software Architectures. *Pages 410–428 of: Proceedings of the 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Lecture Notes in Computer Science, vol. 1687.

Ganter, B., & Wille, R. 1998. *Formal Concept Analysis: Mathematical Foundations*. Secaucus, USA: Springer.

Gervasi, V., & Zowghi, D. 2005. Reasoning about inconsistencies in natural language requirements. *ACM Transactions on Software Engineering and Methodology*, **14**(3), 277–330.

Ginsberg, M. 1988. Multivalued Logics: A Uniform Approach to Reasoning in Artificial Intelligence. *Computational Intelligence*, **4**, 265–316.

Glinz, M. 1995. An Integrated Formal Model of Scenarios Based on Statecharts. *Pages 254–271 of: Proceedings of the 5th European Software Engineering Conference*.

Goal-oriented Requirement Language. 2004. *The Goal-oriented Requirement Language (GRL)*. `http://www.cs.toronto.edu/km/GRL`.

Goguen, J. 1968. *Categories of Fuzzy Sets: Applications of Non-Cantorian Set Theory*. Ph.D. thesis, University of California, Berkeley.

Goguen, J. 1974. Concept representation in natural and artificial languages. *International Journal of Man-Machine Studies*, **6**(5), 513–561.

Goguen, J. 1991. A Categorical Manifesto. *Mathematical Structures in Computer Science*, **1**(1), 49–67.

Goguen, J. 2005. Data, Schema, Ontology and Logic Integration. *Logic Journal of the IGPL*, **13**(6), 685–715.

Goguen, J., & Burstall, R. 1984. Some fundamental algebraic tools for the semantics of computation, Part I: Comma Categories, Colimits, Signatures and Theories. *Theoretical Computer Science*, **31**, 175–209.

Goguen, J., & Burstall, R. 1992. Institutions: abstract model theory for specification and programming. *Journal of the ACM*, **39**(1), 95–146.

Goguen, J., & Rosu, G. 2002. Institution Morphisms. *Formal Aspects of Computing*, **13**(3-5), 274–307.

Goguen, J., Thatcher, J., & Wagner, E. 1987. An initial algebra approach to the specification, correctness and implementation of abstract data types. *Chap. 5 of:* Yeh, R. (ed), *Current Trends in Programming Methodology*, vol. 4. Prentice-Hall.

Gotel, O., & Finkelstein, A. 1994. An Analysis of the Requirements Traceability Problem. *Pages 94–101 of: Proceedings of the 1st International Conference on Requirements Engineering.*

Gotel, O., & Finkelstein, A. 1995. Contribution structures (Requirements artifacts). *Pages 100–107 of: Proceedings of the 2nd International Symposium on Requirements Engineering.*

Gotel, O., & Finkelstein, A. 1997. Extended requirements traceability: results of an industrial case study. *Pages 169–178 of: Proceedings of the 3rd International Symposium on Requirements Engineering.*

Gruber, T., & Russell, D. 1996. Generative design rationale: beyond the record and replay paradigm. *Pages 323–349 of:* Moran, T., & Carroll, J. (eds), *Design rationale: concepts, techniques, and use.* Mahwah, NJ, USA: Lawrence Erlbaum Associates.

Gurfinkel, A. 2007. *Model-Checking with Many Values.* Ph.D. thesis, University of Toronto, Canada.

Hay, J., & Atlee, J. 2000. Composing features and resolving interactions. *Pages 110–119 of: Proceedings of the 8th International Symposium on Foundations of Software Engineering.*

Heckel, R., Engels, G., Ehrig, H., & Taentzer, G. 1999. A view-based approach to system modeling based on open graph transformation systems. *Pages 639–668 of:* Ehrig, H., Engels, G., Kreowski, H., & Rozenberg, G. (eds), *Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools.* River Edge, USA: World Scientific.

Herbsleb, J. 2007. Global Software Engineering: The Future of Socio-Technical Coordination. *Pages 188–198 of: Future of Software Engineering Track of the 29th International Conference on Software Engineering.*

Hitzler, P., Krötzsch, M., Ehrig, M., & Sure, Y. 2005. What Is Ontology Merging? - A Category-Theoretical Perspective Using Pushouts. *Pages 104–107 of: Proceedings*

of the First International Workshop on Contexts and Ontologies: Theory, Practice and Applications. Frontiers in Artificial Intelligence and Applications, vol. 150.

Höhle, U., & Rodabaugh, S.E. (eds). 1999. *Mathematics of Fuzzy Sets: Logic, Topology, and Measure Theory*. Kluwer Academic Publishers.

Horkoff, J. 2006. *Using i* Models for Evaluation*. M.Sc. thesis, University of Toronto.

Hunter, A., & Nuseibeh, B. 1998. Managing Inconsistent Specifications: Reasoning, Analysis, and Action. *ACM Transactions on Software Engineering and Methodology*, **7**(4), 335–367.

Hussmann, H., Demuth, B., & Finger, F. 2002. Modular architecture for a toolset supporting OCL. *Science of Computer Programming*, **44**(1).

IBM Rational Software Architect. 2007. `http://www.ibm.com/software/awdtools/architect/swarchitect/`.

Immerman, N., & Vardi, M. 1997. Model Checking and Transitive-Closure Logic. *Pages 291–302 of: Proceedings of the 9th International Conference on Computer Aided Verification*.

Jackson, D. 2006. *Software Abstractions Logic, Language, and Analysis*. The MIT Press.

Jannink, J., Pichai, S., Verheijen, D., & Wiederhold, G. 1998. Encapsulation and composition of ontologies. *In: Proceedings of the AAAI Workshop on AI and Information Integration*.

Kalfoglou, Y., & Schorlemmer, M. 2005. Ontology Mapping: The State of the Art. *In:* Kalfoglou, Y., Schorlemmer, M., Sheth, A., Staab, S., & Uschold, M. (eds), *Semantic Interoperability and Integration*. Dagstuhl Seminar Proceedings, no. 04391. IBFI.

Kolovos, D., Paige, R., & Polack, F. 2006. Merging Models with the Epsilon Merging Language (EML). *Pages 215–229 of: Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems.*

Konrad, S., & Cheng, B. 2005. Real-Time Specification Patterns. *Pages 372–381 of: Proceedings of the 27th International Conference on Software Engineering.*

Laddad, R. 2003. *AspectJ in Action: Practical Aspect-Oriented Programming.* Greenwich, CT, USA: Manning Publications.

Larsen, K., & Thomsen, B. 1988. A Modal Process Logic. *Pages 203–210 of: Proceedings of the 3rd Annual Symposium on Logic in Computer Science.*

Letkeman, K. 2005. *Comparing and Merging UML Models in IBM Rational Software Architect.* `http://www.ibm.com/developerworks/rational/library/05/712_comp/`.

Liaskos, S. 2007. *Quality Criteria for Variability Modelling.* Tech. rept. CSRG-549. University of Toronto.

Libkin, L. 2004. *Elements Of Finite Model Theory.* Texts in Theoretical Computer Science. An EATCS Series. Springer.

Mandelin, D., Kimelman, D., & Yellin, D. 2006. A Bayesian Approach to Diagram Matching with Application to Architectural Models. *Pages 222–231 of: Proceedings of the 28th International Conference on Software Engineering.*

Mehra, A., Grundy, J., & Hosking, J. 2005. A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design. *Pages 204–213 of: Proceedings of the 20th International Conference on Automated Software Engineering.*

Melnik, S. 2004. *Generic Model Management: Concepts and Algorithms.* Lecture Notes in Computer Science, vol. 2967. Berlin, Germany: Springer.

Melnik, S., Rahm, E., & Bernstein, P. 2003. Rondo: a programming platform for generic model management. *Pages 193–204 of: SIGMOD Conference*.

Mens, T. 2002. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering*, **28**(5), 449–462.

Milner, R. 1989. *Communication and Concurrency*. New York, USA: Prentice-Hall.

Mylopoulos, J. 1998. Information Modeling in the Time of the Revolution. *Information Systems*, **23**(3–4), 127–155.

Nejati, S. 2008. *Behavioural Model Fusion*. Ph.D. thesis, University of Toronto.

Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., & Zave, P. 2007. Matching and Merging of Statecharts Specifications. *Pages 54–64 of: Proceedings of the 29th International Conference on Software Engineering*.

Nentwich, C., Emmerich, W., Finkelstein, A., & Ellmer, E. 2003. Flexible Consistency Checking. *ACM Transactions on Software Engineering and Methodology*, **12**(1), 28–63.

Niu, N., Easterbrook, S., & Sabetzadeh, M. 2005. A Category-Theoretic Approach to Syntactic Software Merging. *Pages 197–206 of: Proceedings of the 21st International Conference on Software Maintenance*.

Noy, N., & Musen, M. 2000. PROMPT: Algorithm and Tool for Automated Ontology Merging and Alignment. *Pages 450–455 of: Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence*.

Nuseibeh, B., Kramer, J., & Finkelstein, A. 1994. A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification. *IEEE Transactions on Software Engineering*, **20**(10), 760–773.

Nuseibeh, B., Easterbrook, S., & Russo, A. 2000. Leveraging Inconsistency in Software Development. *IEEE Computer*, **33**(4), 24–29.

Nuseibeh, B., Easterbrook, S., & Russo, A. 2001. Making Inconsistency Respectable in Software Development. *Journal of Systems and Software*, **58**(2), 171–180.

Object Constraint Language. 2003. *The Object Constraint Language*. `http://www.omg.org/technology/documents/formal/ocl.htm`.

OMG Systems Modeling Language. 2006. `http://www.sysml.org/`.

Osborn, A. 1979. *Applied Imagination*. Scribner.

Paige, R., Brooke, P., & Ostroff, J. 2007. Metamodel-Based Model Conformance and Multiview Consistency Checking. *ACM Transactions on Software Engineering and Methodology*, **16**(3).

Pottinger, R., & Bernstein, P. 2003. Merging Models Based on Given Correspondences. *Pages 862–873. of: Proceedings of the 29th International Conference on Very Large Data Bases*.

Richards, D. 2003. Merging individual conceptual models of requirements. *Requirements Engineering*, **8**(4), 195–205.

Rosen, E. 2002. Some Aspects of Model Theory and Finite Structures. *The Bulletin of Symbolic Logic*, **8**(3), 380–403.

Ross, D. 1985. Applications and Extensions of SADT. *IEEE Computer*, **18**(4), 25–34.

Rossman, B. 2005. Existential Positive Types and Preservation under Homomorphisisms. *Pages 467–476 of: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*.

Rossman, B. 2008. *Homomorphism Preservation Theorem on Finite Structures.* Tech. rept. Massachusetts Institute of Technology. Manuscript.

Rozenberg, G. (ed). 1997. *Handbook of graph grammars and computing by graph transformation: Foundations.* Vol. 1. River Edge, NJ, USA: World Scientific.

Rydeheard, D., & Burstall, R. 1988. *Computational Category Theory.* Hertfordshire, UK: Prentice Hall.

Sabetzadeh, M. 2003. *A Category-Theoretic Approach to Representation and Analysis of Inconsistency in Graph-Based Viewpoints.* M.Sc. thesis, University of Toronto.

Sabetzadeh, M., & Easterbrook, S. 2003. Analysis of Inconsistency in Graph-Based Viewpoints: A Category-Theoretic Approach. *Pages 12–21 of: Proceedings of the 18th International Conference on Automated Software Engineering.*

Sabetzadeh, M., & Easterbrook, S. 2005a. iVuBlender: A Tool for Merging Incomplete and Inconsistent Views. *Pages 453–454 of: Proceedings of the 13th International Requirements Engineering Conference.* Tool Demo Paper.

Sabetzadeh, M., & Easterbrook, S. 2005b. Traceability in Viewpoint Merging: A Model Management Perspective. *Pages 44–49 of: Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering.*

Sabetzadeh, M., & Easterbrook, S. 2006. View Merging in the Presence of Incompleteness and Inconsistency. *Requirements Engineering Journal*, **11**(3), 174–193.

Sabetzadeh, M., & Nejati, S. 2006. TReMer: A Tool for Relationship-Driven Model Merging. *In: Posters and Research Tools Track of Formal Methods.* No published proceedings.

Sabetzadeh, M., Nejati, S., Liaskos, S., Easterbrook, S., & Chechik, M. 2007a. Consistency Checking of Conceptual Models via Model Merging. *Pages 221–230 of: Proceedings of the 15th International Requirements Engineering Conference.*

Sabetzadeh, M., Nejati, S., Easterbrook, S., & Chechik, M. 2007b. A Relationship-Driven Framework for Model Merging. *In: Proceedings of the International Workshop on Modeling in Software Engineering.*

Sabetzadeh, M., Nejati, S., Easterbrook, S., & Chechik, M. 2008. Global Consistency Checking of Distributed Models with TReMer+. *Pages 815–818 of: Proceedings of the 30th International Conference on Software Engineering.* Formal Demonstration.

Salay, R., Chechik, M., Easterbrook, S., Diskin, Z., McCormick, P., Nejati, S., Sabetzadeh, M., & Viriyakattiyaporn, P. 2007. An Eclipse-Based Tool Framework for Software Model Management. *In: Proceedings of the OOPSLA Workshop on Eclipse Technology eXchange.*

Sannella, D., & Tarlecki, A. 1999. Algebraic Preliminaries. *Chap. 2 of:* Astesiano, E., Kreowski, H., & Krieg-Brückner, B. (eds), *Algebraic Foundations of Systems Specification.* Springer.

Schorlemmer, M., Potter, S., & Robertson, D. 2002. *Automated Support for Composition of Transformational Components in Knowledge Engineering.* Tech. rept. EDI-INF-RR-0137. University of Edinburgh.

Spaccapietra, S., & Parent, C. 1994. View Integration: A Step Forward in Solving Structural Conflicts. *Knowledge and Data Engineering*, **6**(2), 258–274.

Spanoudakis, G., & Zisman, A. 2001. *Handbook of Software Engineering and Knowledge Engineering.* World scientific. Chap. Inconsistency management in software engineering: Survey and open research issues, pages 329–380.

Spanoudakis, G., Finkelstein, A., & Till, D. 1999. Overlaps in Requirements Engineering. *Automated Software Engineering*, **6**(2), 171–198.

Spivey, J. 1992. *The Z Notation: A Reference Manual*. second edn. Hertfordshire, UK: Prentice Hall.

Stamper, R. 1994. Social norms in requirements analysis: an outline of MEASUR. *Pages 107–139 of:* Jirotka, M., & Goguen, J. (eds), *Requirements engineering: social and technical issues*. London, UK: Academic Press.

Stumme, G., & Maedche, A. 2001. FCA-Merge: Bottom-Up Merging of Ontologies. *Pages 225–234 of: Proceedings of the 17th International Joint Conference on Artificial Intelligence*.

Svetinovic, D., Berry, D., & Godfrey, M. 2005. Concept Identification in Object-Oriented Domain Analysis: Why Some Students Just Don't Get It. *Pages 189–198 of: Proceedings of the 13th International Requirements Engineering Conference*.

Tarlecki, A. 1986. Bits and pieces of the theory of institutions. *Pages 334–363 of:* Abramsky, S., Poigné, A., & Rydeheard, D. (eds), *Summer Workshop on Category Theory and Computer Programming*. Springer.

Tarlecki, A. 1999. Institutions: An Abstract Framework for Formal Specifications. *In:* E. Astesiano, H. Kreowski, B. Krieg-Brückner (ed), *Algebraic Foundations of System Specification*. Springer.

Uchitel, S., & Chechik, M. 2004. Merging partial behavioural models. *Pages 43–52 of: Proceedings of the 12th International Symposium on Foundations of Software Engineering*.

Unified Modeling Language. 2003. *The Unified Modeling Language*. `http://www.uml.org/`.

van Lamsweerde, A. 2000. Formal Specification: A Roadmap. *Pages 147–159 of: Proceedings of the Conference on The Future of Software Engineering.*

van Lamsweerde, A., Darimont, R., & Massonet, P. 1995. Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt. *Pages 194–203 of: Proceedings of the 2nd International Symposium on Requirements Engineering.*

van Lamsweerde, A., Darimont, R., & Letier, E. 1998. Managing Conflicts in Goal-Driven Requirements Engineering. *IEEE Transactions on Software Engineering*, **24**(11), 908–926.

Vardi, M. 1982. The Complexity of Relational Query Languages. *Pages 137–146 of: Proceedings of the 14th Annual ACM Symposium on Theory of Computing.*

Vaziri, M., & Jackson, D. 1999. *Some Shortcomings of OCL, the Object Constraint Language of UML.* Tech. rept. Massachusetts Institute of Technology.

Wahler, M., Koehler, J., & Brucker, A. 2006. Model-Driven Constraint Engineering. *Pages 111–125 of: Proceedings of the MoDELS Workshop on OCL for (Meta-)Models in Multiple Application Domains.*

Yu, E. 1997. Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering. *Pages 226–235 of: Proceedings of the 3rd International Symposium on Requirements Engineering.*

Zave, P., & Jackson, M. 1993. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology*, **2**(4), 379–411.

Zave, P., & Jackson, M. 2002. A Call Abstraction for Component Coordination. *Electronic Notes in Theoretical Computer Science*, **66**(4).

Zave, P., Goguen, H., & Smith, T. 2004. Component coordination: a telecommunication case study. *Computer Networks*, **45**(5), 645–664.

Zimmermann, A., Krötzsch, M., Euzenat, J., & Hitzler, P. 2006. Formalizing Ontology Alignment and its Operations with Category Theory. *Pages 277–288 of: Proceedings of the Fourth International Conference on Formal Ontology in Information Systems.* Frontiers in Artificial Intelligence and Applications, vol. 150.

Zisman, A., & Kozlenkov, A. 2001. Knowledge Base Approach to Consistency Management of UML Specification. *Pages 359–363 of: Proceedings of the 16th International Conference on Automated Software Engineering.*

Appendix A

Hospital Case Study Information

In this appendix, we provide additional information about the case study in Chapter 5.

A.1 Problem Description

The hospital system description handed out to students is as follows[1].

A.1.1 Background

The hospital gives around-the-clock medical care, diagnosis and treatment to the sick and injured on both an inpatient and an outpatient basis. The hospital is divided into several wards, described below. The medical team is comprised of physicians, technicians (technical staff), nurses, and administrative assistants.

A.1.2 Wards, Rooms, Units, Beds and Equipment

Wards have rooms for patients to stay in. Rooms have beds, and beds are numbered per room, so a combination of the ward, room, and bed number will uniquely identify every bed in the hospital.

[1] I thank Paul Gries for his invaluable help with preparing this description.

The pieces of equipment in a ward can be mobile or stationary. If stationary, a piece will be assigned to a single ward room or unit; and if mobile, it will be assigned to the general storage area in the ward. Equipment is not shared between wards. For each piece of mobile equipment, there is a booking calendar to keep track of when and where the piece is needed.

Wards also have units, such as critical care units and operating room units. The set of units varies from ward to ward, but every ward has a single general storage area unit. Each unit has a textual description outlining its responsibilities, as well as a list of equipment in that unit.

A.1.3 Display Units and Scanners

Every ward unit and room has a display and a scanner. The display can show everything from a medical team member's schedule to a patient's history. The scanner is connected to the display, and can be used to scan both the wrist band of a patient (described below) and the id card of a member of the medical team.

A.1.4 Medical Team

Each medical team member has a name and a unique id number, and all team members carry an id card containing that information.

Nurses are affiliated with a single ward, while physicians and technicians can be affiliated with several different wards. All personnel have access to a calendar detailing the hours that they need to be present at the various wards. Nurses record physicians' decisions such as diagnoses, medical procedures that are to be performed (and when), new prescriptions (including medication, amount, and schedule), cancelled prescriptions, whether a patient needs to be transferred (and to which ward), and whether a patient can check out of the hospital. These are written on paper and handed to an administrative assistant to enter. The administrative assistant needs to figure out who needs to be at a

particular procedure before they enter it in the system.

Here are some necessary items for the system:

- Each procedure has a list of medical equipment necessary to perform that procedure. A procedure will only be scheduled when all pieces of equipment are available.
- The technicians can view their schedule and move equipment as necessary.
- Physicians have a set of patients they are assigned to. Nurses and physicians can review a patient's medical history.
- Everyone needs to be able to see a list of the medical procedures in which they are involved.
- Physicians need to be able to get a list of their patients to be visited in a ward.

A.1.5 Patients

Unless new patients are in a life-threatening situation, they must register at the hospital. This involves an administrative assistant scanning their health card, which contains their name, address, and a unique health card number. After registering they receive an early diagnosis by a physician in which they are classified as either an "inpatient" or an "outpatient". An inpatient is a person who is admitted at least for one night to the hospital; and an outpatient is a person who visits the hospital for diagnosis or treatment without spending the night. This is determined by the physician doing the initial assessment.

Outpatients on a return visit may arrive at the hospital and go straight to the ward where they are to receive treatment.

Each patient, be that an inpatient or an outpatient, has a profile capturing the patient's name, address, health card number, and their general health-related remarks. For each outpatient visit, an outpatient visit record is created storing the date and time of the visit, the patient's health problem, the name(s) of the physician(s) who attended to the patient, the diagnosis, and the prescribed medication. For each inpatient visit,

an inpatient visit record is created storing the date and time of admission, the patient's health problem(s), the early diagnosis, and the name(s) of the physician(s) involved in making the early diagnosis. To facilitate the management of inpatients, a wristband is produced for each inpatient at admission. The hospital's medical team will use special handheld scanners to scan the wristbands and fetch the inpatient visit records, described below.

During each stay at the hospital, an inpatient may be transferred several times between different wards and between different parts of a single ward. For each ward stay, a ward stay record is created. The information stored in a ward stay record includes a list of intra-ward stays, a list of medical procedures performed on the patient during their stay at the ward, the name(s) of the physicians attending to the patient while at the ward, a ward-specific diagnosis and a ward-specific medication chart. The ward nurses use these charts to administer the required medication. The information stored in an intra-ward stay record includes the date and time when the patient checked in the corresponding ward room or unit, the checkout date and time, and if applicable, the id of the bed assigned to the patient within the room or unit.

Patients in a life-threatening situation are assessed and treated as normal, but without registering them first. They are all issued a wristband at some point for identification, and flagged in the system as not having registered.

A.2 Source Models and Their Merges

Figures A.1–A.5 show the five source models in our study. Figure A.6 shows the merge of these models with respect to the preliminary mappings that we manually defined between the models. Additionally, we provide a coherent merge after evolving the models and refining their mappings through multiple rounds of global consistency checking and manual inspections. Note that elements which were found to be ill-conceived during our analysis were filtered out from this final merge.

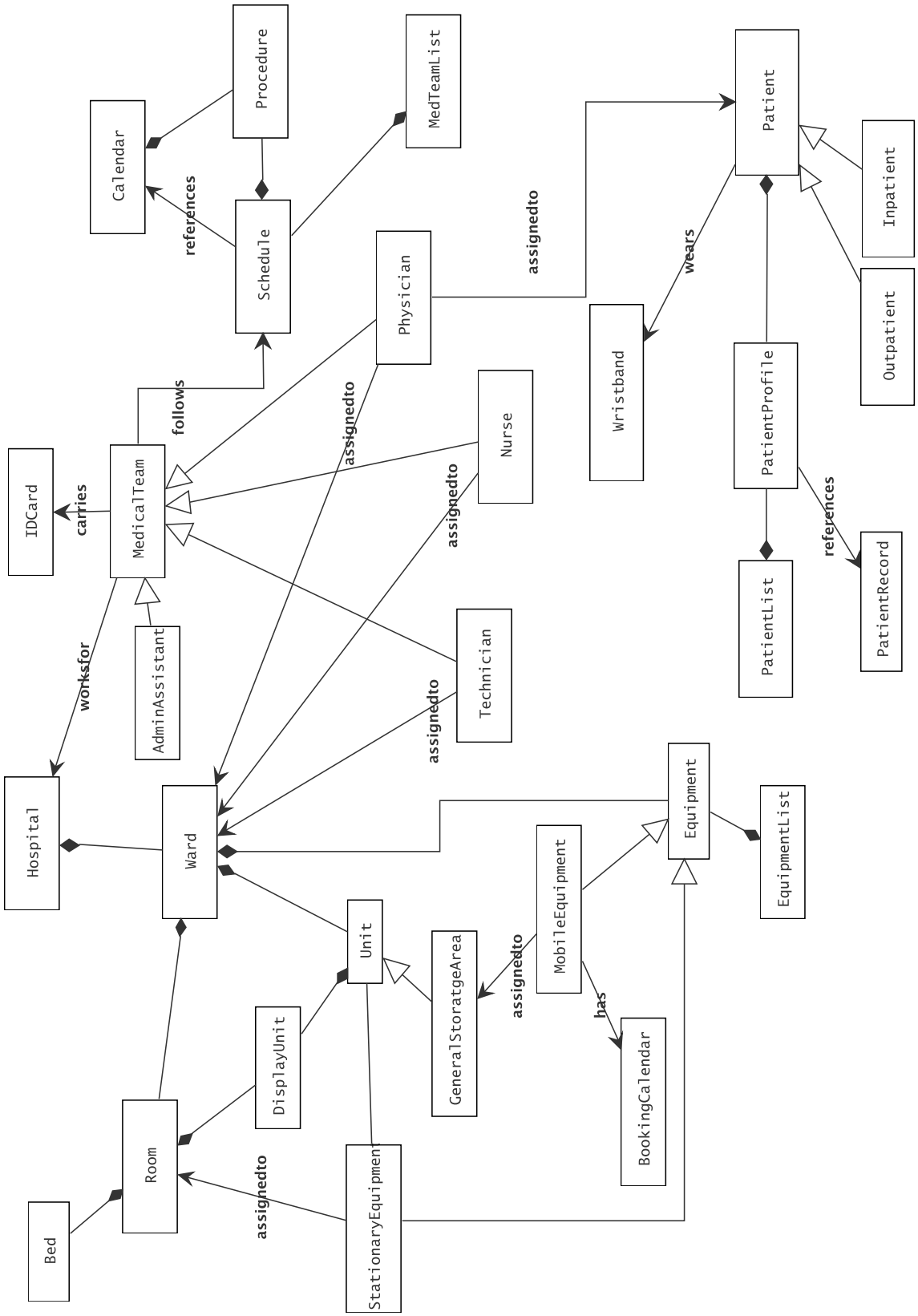


Figure A.1: Source model I

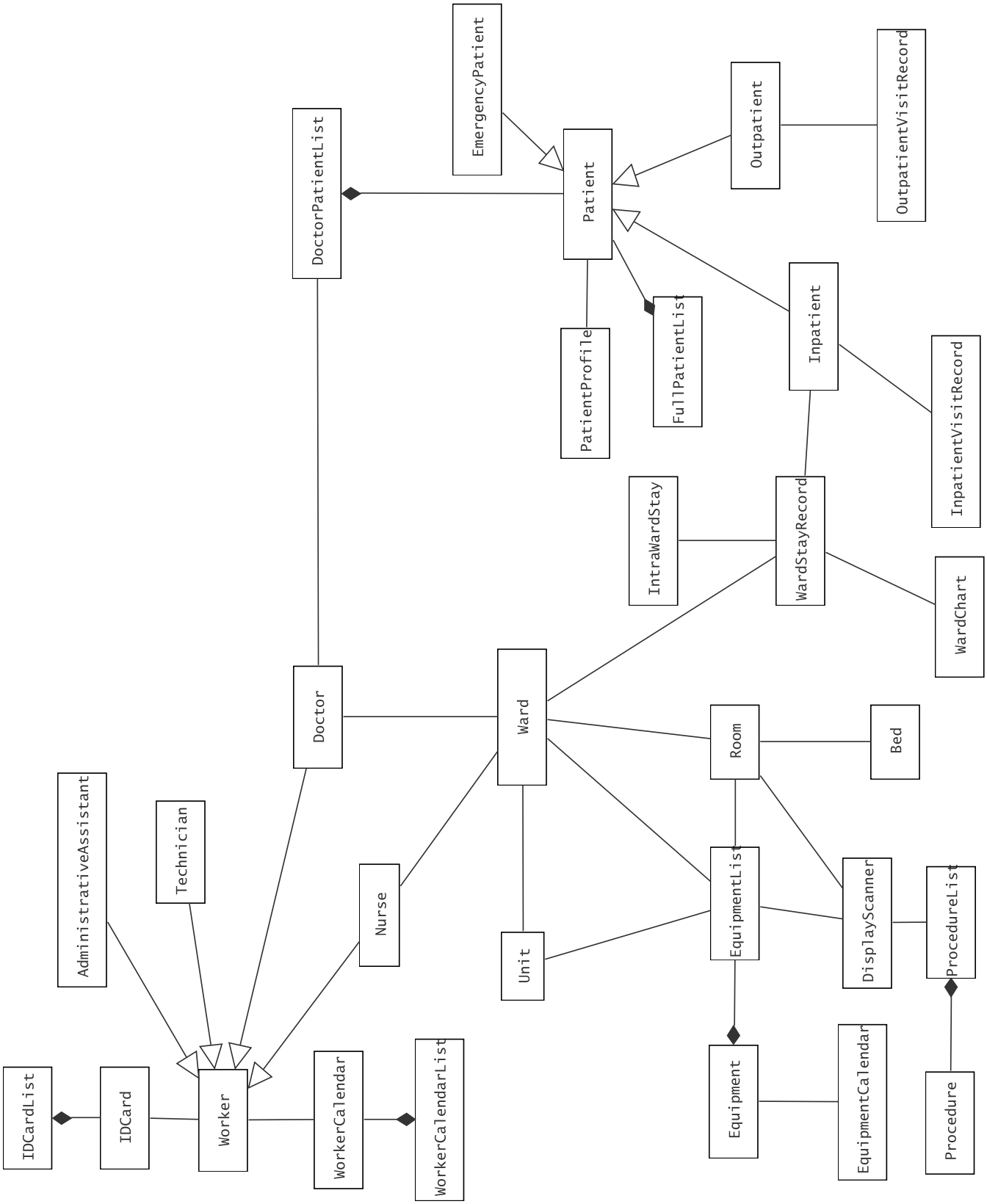


Figure A.2: Source model II

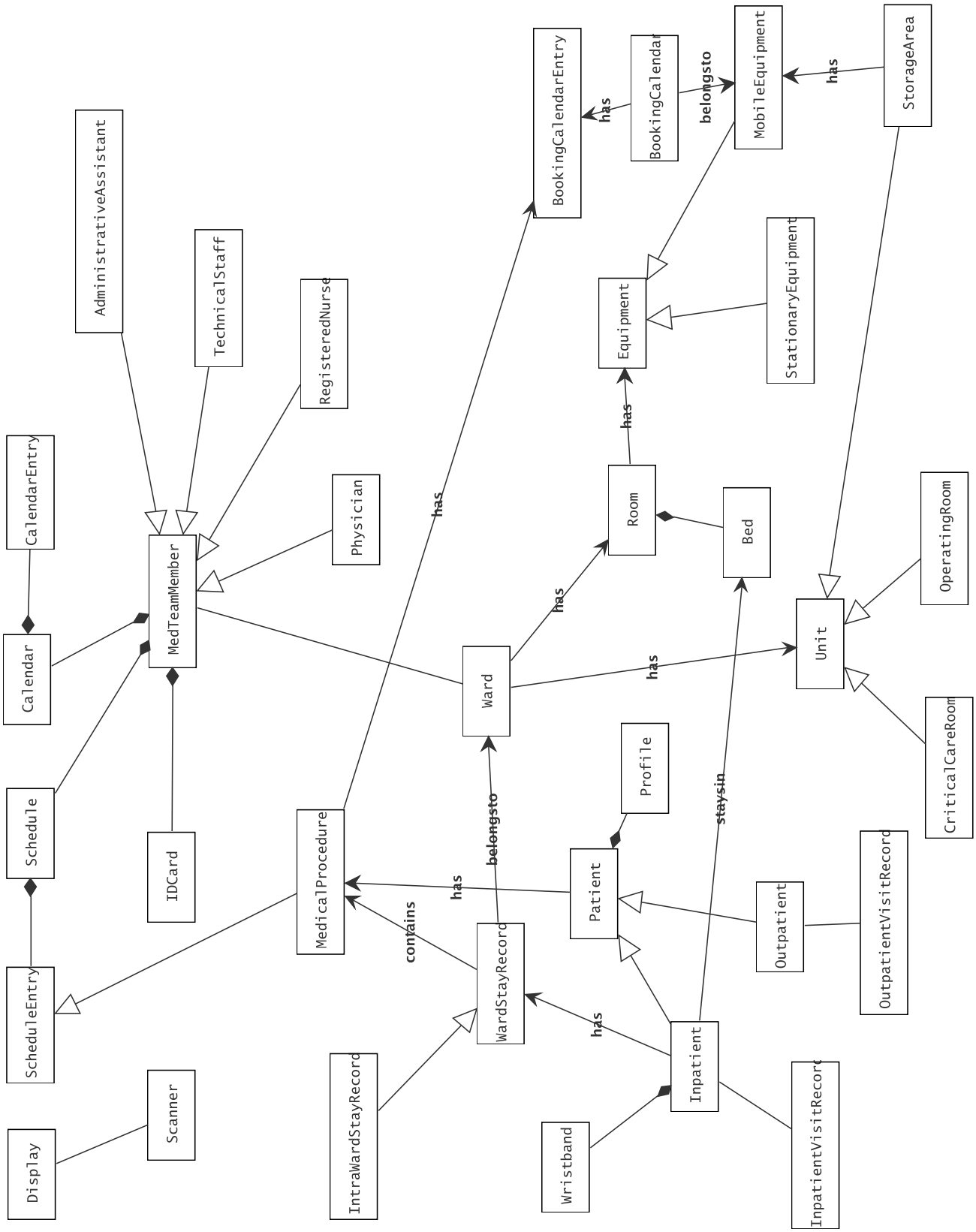


Figure A.3: Source model III

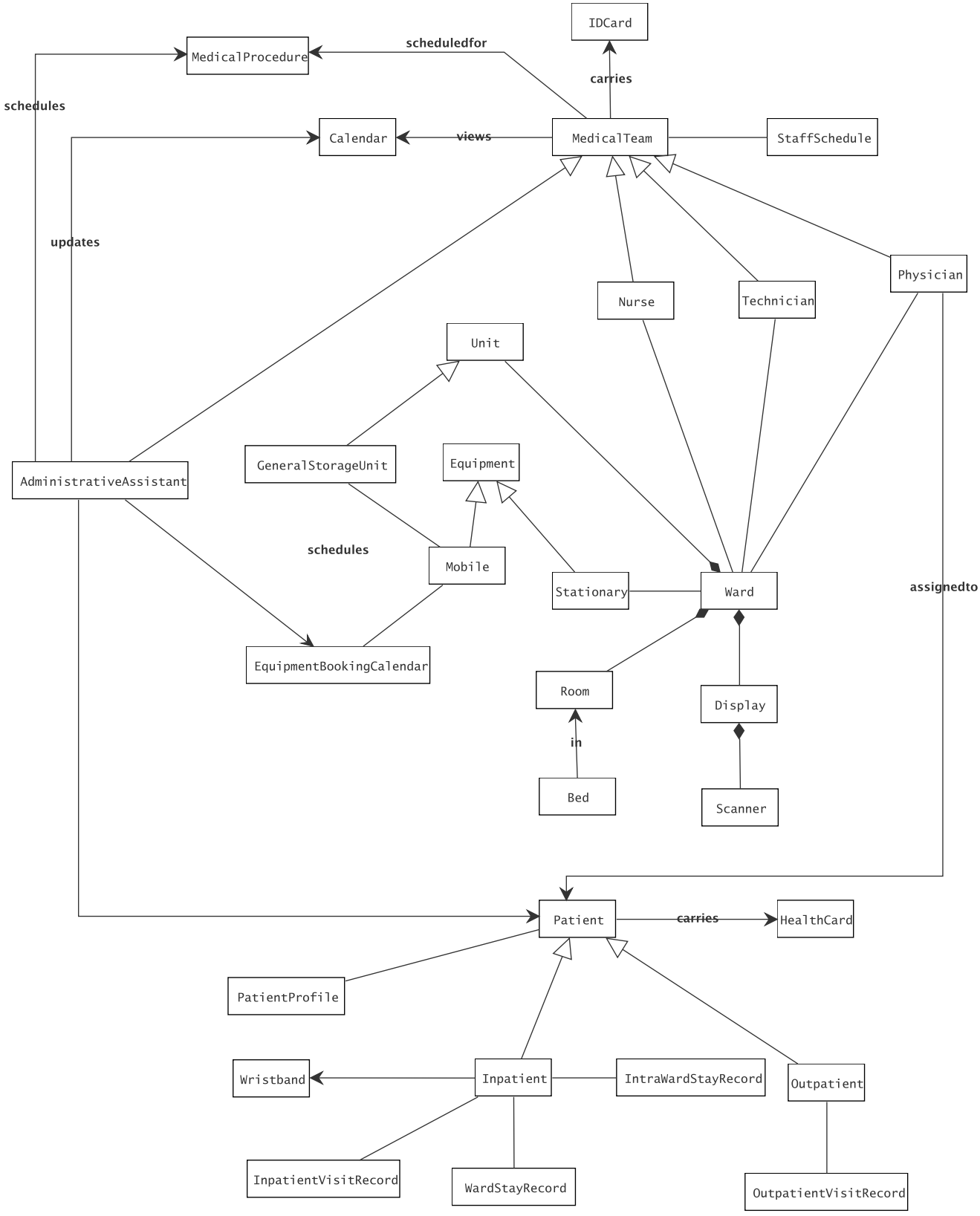


Figure A.4: Source model IV

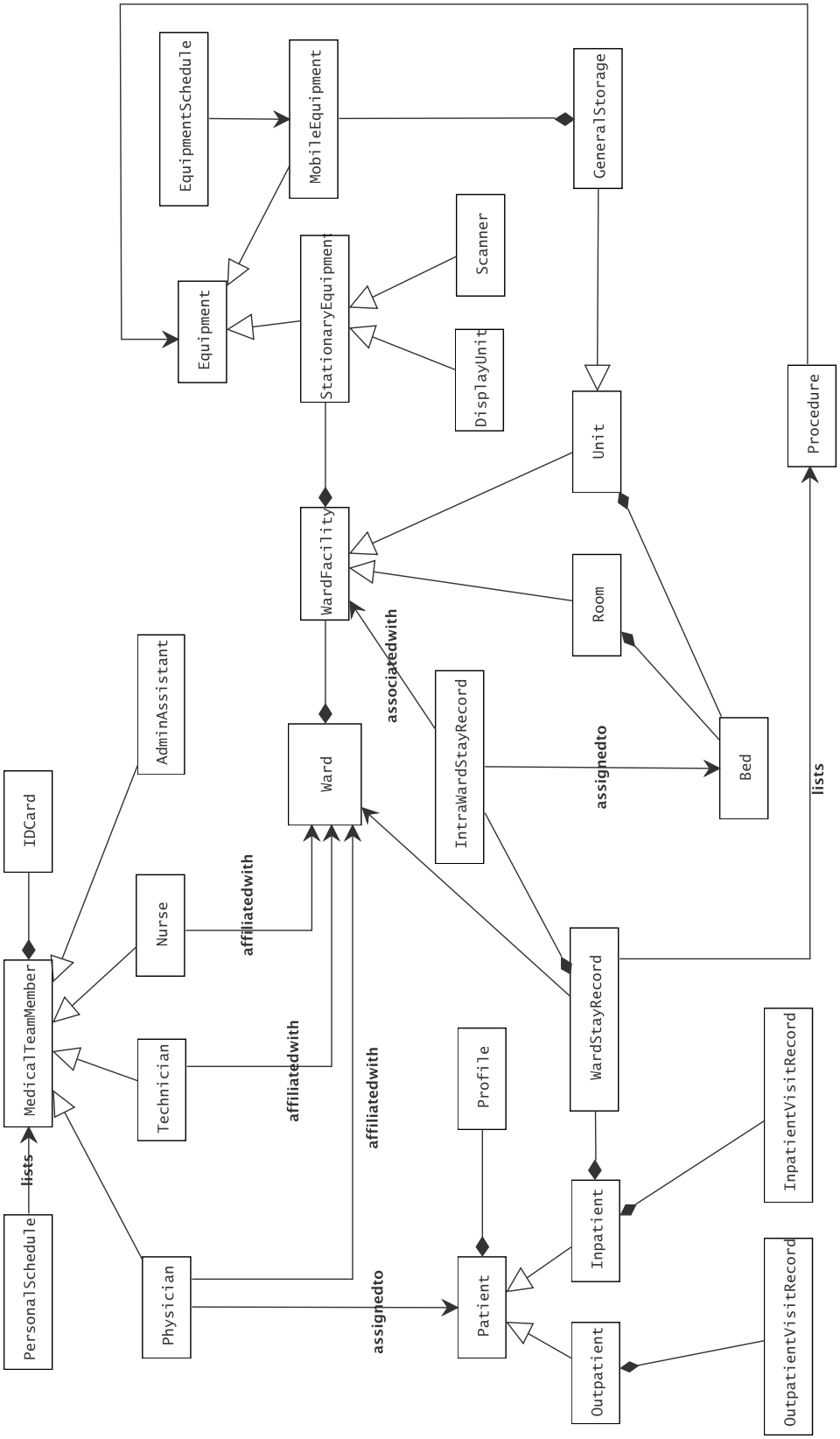


Figure A.5: Source model V

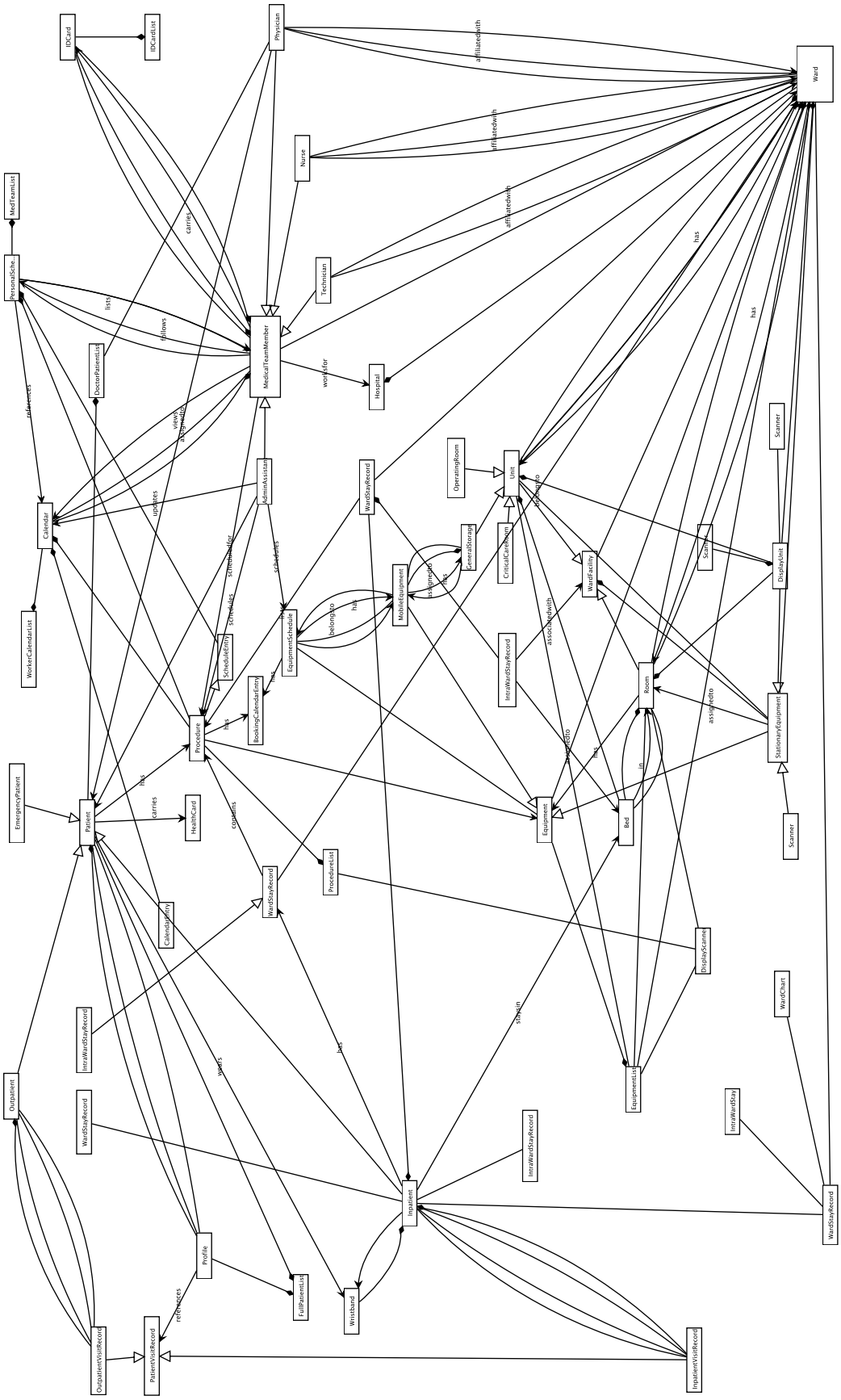


Figure A.6: Merge with respect to our preliminary mappings

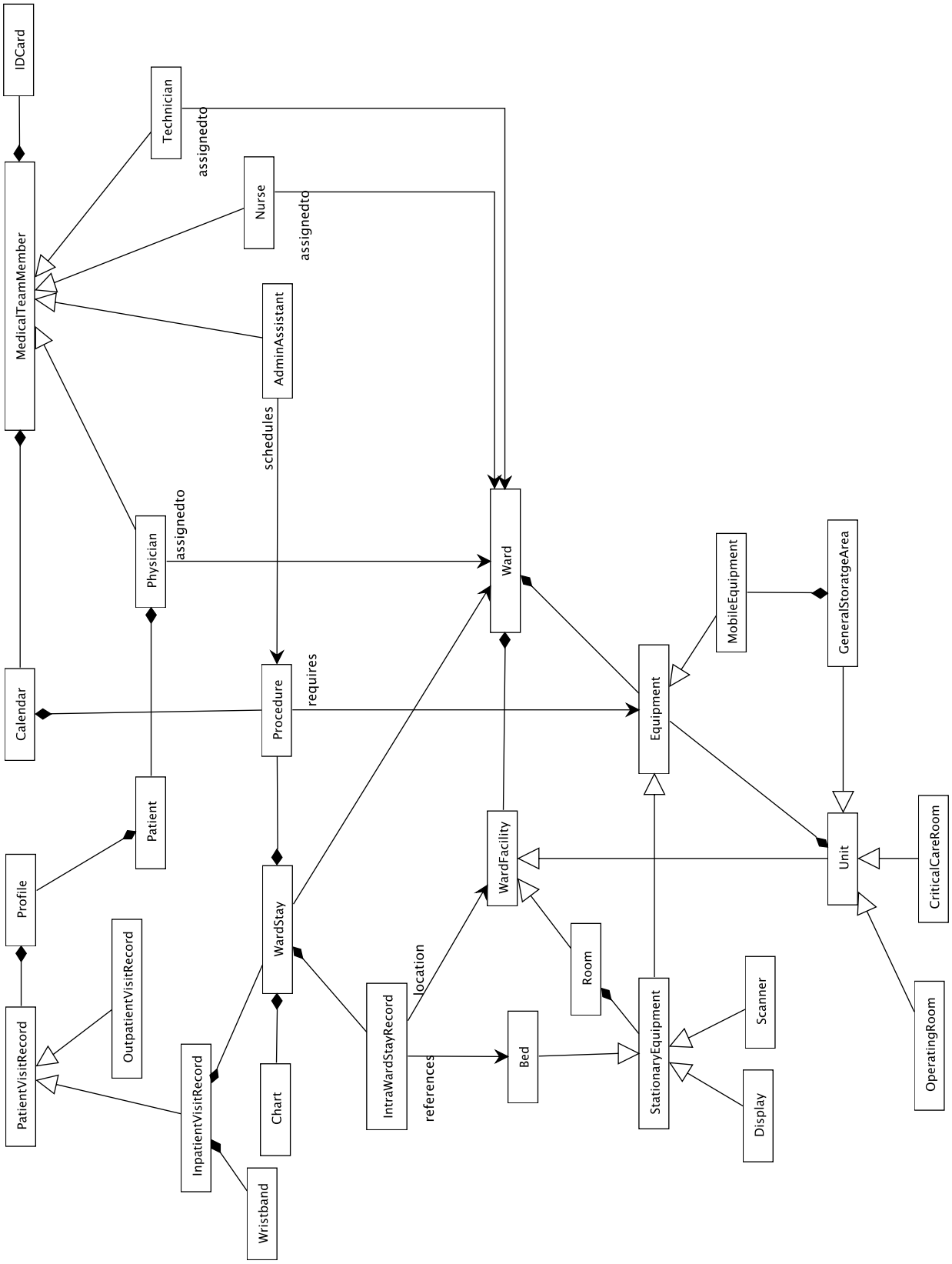


Figure A.7: Final merge after refining the source models and mappings